Université de Reims Champagne-Ardenne
École Doctorale Sciences Technologies et Santé

Thèse

présentée par **Cyril RABAT**

pour l'obtention du grade de

Docteur de l'Université de Reims Champagne-Ardenne

Spécialité : Informatique

Étude et simulation de solutions pour les grilles et systèmes pair-à-pair : application à la gestion des ressources et des tâches

Président	Pierre Fraigniaud	Directeur de Recherche à l'Université Paris Diderot (Paris 7)
Rapporteur	Sébastien Tixeuil	Professeur à l'Université Pierre et Marie Curie (Paris 6)
Rapporteur	Pascal Felber	Professeur à l'Université de Neuchâtel
Examinateur	Eddy Caron	Maître de conférences à l'École Normale Supérieure de Lyon
Examinateur	Hacène Fouchal	Professeur à l'Université des Antilles et de la Guyane
Directeur	Alain Bui	Professeur à l'Université de Reims Champagne-Ardenne
Co-directeur	Olivier Flauzac	Professeur à l'Université de Reims Champagne-Ardenne

---

Centre de Recherche en STIC équipe Systèmes Communicants

"À ma femme et à mon fils..."

Remerciements

Je tiens à exprimer mes remerciements et toute ma gratitude à *Alain Bui* et à *Olivier Flauzac*, Professeurs à l'Université de Reims Champagne-Ardenne, pour avoir accepté de m'encadrer pour cette thèse ainsi que pour la confiance qu'ils m'ont manifestée tout au long de ces années. Les pistes qu'ils m'ont données m'ont permis d'avancer dans mes recherches, de même que l'indépendance qu'ils m'ont accordée. J'espère avoir été à la hauteur de leur espérance.

Je tiens à remercier *Hacène Fouchal*, Professeur à l'Université des Antilles et de la Guyane, qui m'a fait découvrir le monde passionnant de la Recherche et qui m'a permis d'étendre mes connaissances vers d'autres domaines que ceux abordés dans ces travaux. C'est grâce à lui que je me suis intéressé à la recherche et dirigé vers cette thèse.

Je remercie le Professeur *Pascal Felber* de l'Université de Neuchâtel, en Suisse, pour m'avoir accueilli au sein de son laboratoire pendant deux semaines en juin 2007 et avec qui j'ai eu des échanges très enrichissants. Je le remercie aussi pour avoir accepté d'être le rapporteur de cette thèse.

Merci également au Professeur *Tixeuil*, de l'Université Pierre et Marie Curie, pour avoir accepté de rapporter ce manuscrit et pour toutes ces remarques constructives.

Je remercie le Professeur *Pierre Fraigniaud*, Directeur de recherche à l'Université Paris Diderot, pour l'honneur qu'il me fit d'accepter la présidence du jury de la soutenance publique de mes travaux de recherche.

Je tiens également à remercier *Eddy Caron* de l'École Normale Supérieure de Lyon, pour l'intérêt qu'il porte à mes travaux et pour avoir accepté de faire partie de ce même jury.

La recherche est aussi un travail d'équipe et je tiens tout d'abord à remercier mon collègue *Thibault Bernard* avec qui j'ai co-écrit plusieurs articles, mes différents collègues *Christophe Jaillet* et *Devan Sohier*, avec qui j'ai eu de nombreuses discussions, me permettant d'avancer dans mes travaux. Tous les trois m'ont grandement aidé en relisant ce manuscrit. Merci également aux collègues de bureau qui ont eu à me supporter durant ces dernières années *Arnaud Renard*, *Antoine Rollet* et *Sylvain Rampacek*.

La vie professionnelle et la vie privée sont difficilement dissociables au cours d'une thèse. Aussi, un grand merci à ma femme *Frédérique* qui a du supporter mon ordinateur durant toutes ces années, qui a toujours été une grande source de motivation et qui m'a aidé à garder un bon équilibre.

Table des matières

Table des figures

Liste des algorithmes

Introduction

La puissance des processeurs n'a cessé de progresser durant les dernières années. Dans la plupart des ordinateurs de bureau actuels, les processeurs sont en mesure de réaliser plusieurs milliards d'opérations à la seconde. L'aspect économie d'énergie a fait évoluer leur architecture vers une autre voie : les fondeurs préfèrent dorénavant augmenter le nombre de cœurs au sein des processeurs plutôt que d'augmenter encore leur fréquence qui est source de chaleur et donc de perte d'énergie. De plus, Gordon Moore[1] a annoncé récemment que sa loi ne serait plus applicable d'ici une dizaine d'années, la miniaturisation devrait alors atteindre une borne infranchissable.

Mais parallèlement à cette augmentation de la puissance de calcul, les applications et les problèmes ont des besoins de plus en plus importants. Paradoxalement, plus les processeurs deviennent puissants, plus les modèles utilisés sont complexes et donc, plus les applications basées sur ces modèles ont besoin de puissance de calcul. L'un des exemples est la prévision météorologique. Pour améliorer la précision, les modèles climatiques se complexifient pour prendre en compte plus de paramètres. Ainsi, les simulations ont besoin de beaucoup de puissance de calcul pour analyser les données et obtenir des résultats dans un délai raisonnable.

Si la puissance des processeurs augmente constamment, nous en trouvons de plus en plus dans notre vie quotidienne sous de multiples formes. Tout d'abord, nous avons les appareils portables comme les PDA. Ils possèdent un système d'exploitation propre et exécutent des applications de plus en plus complexes. Les téléphones portables ont eux-aussi envahi notre quotidien. Selon l'ARCEP[2], plus de 53 millions de téléphones étaient en circulation en France, fin septembre 2007. De plus, les PDA et les téléphones possèdent de multiples interfaces de communication comme le *WiFi* ou le *BlueTooth*. Si la puissance de calcul de ces appareils n'est pas très élevée, l'accumulation de toute cette puissance est loin d'être négligeable.

De plus en plus de particuliers sont équipés d'un ordinateur personnel dont le processeur central n'est plus la seule source de calcul. Les industriels se sont intéressés depuis plusieurs années à l'exploitation du processeur de la carte graphique, appelé GPU, qui est massivement parallèle. Enfin, les consoles de jeu ont elles-aussi énormément progressées et produisent une puissance de calcul importante, voir supérieure aux ordinateurs de bureau. Ainsi, en août 2007, le projet *Folding@home*[3] battait le record du point de vue de la puissance de calcul *virtuelle*

[1] Gordon Moore avait énoncé en 1965 la Loi qui prévoyait le doublement du nombre de transistors tous les 18 mois dans un processeur de taille équivalente.

[2] L'ARCEP est l'autorité française de régulation des télécommunications (`http ://www.arcep.fr/`).

[3] Le site officiel de *Folding@home* se trouve à l'adresse `http ://folding.stanford.edu/`.

dépassant largement *BlueGene/L* le plus gros ordinateur parallèle actuel[4]. Or, sur la puissance totale disponible dans *Folding@Home*, plus de 50% est fournie par des consoles de jeu *Playstation®3* de *Sony*.

Cependant, pour exploiter toutes ces ressources distribuées, nous avons besoin d'outils. L'une des possibilités est l'utilisation d'une grille informatique. Le principe des grilles est de mettre en commun des ressources partagées, distribuées et hétérogènes. Les ressources peuvent être de la puissance de calcul, de la capacité de stockage, des données ou encore des applications. Cependant, l'utilisation de ressources distribuées (et spécialement lorsqu'elles sont dynamiques) implique la mise en place de mécanismes particuliers. La centralisation des services sur des serveurs rencontrée habituellement dans les grilles n'est pas envisageable. La conception des grilles s'est donc tournée vers une approche pair-à-pair. Dans un tel modèle, les services sont répartis sur l'ensemble des acteurs qui ont tous un rôle identique. Les connexions et déconnexions fréquentes (principe du *churn*) ont donc moins d'impact sur l'ensemble de l'application. De plus, les communications se font directement entre les acteurs, limitant ainsi la charge sur certains nœuds et favorisant le passage à l'échelle pour l'application.

L'exploitation de ressources distribuées à l'échelle planétaire implique de nombreux défis. Premièrement, les ressources sont très hétérogènes. Il est donc important de prendre en compte les caractéristiques propres de chaque entité (architecture, système d'exploitation). De même, les réseaux d'interconnexion sont de natures très différentes. Les débits sont très variés et les protocoles de communication sont eux-aussi très hétérogènes. Deuxièmement, les ressources sont très dynamiques. Les entités sans fils ont généralement une durée d'utilisation limitée et une mobilité importante. Les consoles de jeu ou les ordinateurs de salon ne sont pas allumés en permanence. Ainsi, pour utiliser cette puissance de calcul, il est nécessaire de gérer ce fort dynamisme. Enfin, le nombre de ressources à regrouper est important. L'application doit être en mesure de passer à l'échelle tout en limitant les coûts de gestion et de communication pour maximiser l'utilisation des ressources.

Toutes ces contraintes doivent être prises en compte lors de la conception d'applications pour les grilles ou les systèmes pair-à-pair. Une modélisation précise est donc nécessaire. Elle doit mettre en évidence toutes les interactions sous-jacentes à l'application afin d'aider le concepteur à concevoir des mécanismes appropriés. La difficulté est de trouver un modèle adéquat. Trop proche du matériel, il ne permet pas de mettre en évidence les conséquences de changements topologiques sur l'application. À l'opposé, si le modèle est trop éloigné du matériel et qu'il se focalise sur l'application, il devient difficile de détecter l'origine d'une panne au niveau de l'application et donc de proposer un mécanisme de correction adapté. Dans nos travaux, nous avons ainsi proposé un modèle original constitué de 5 couches superposées qui se place à la fois au niveau du matériel et de l'application. Chaque couche se focalise sur une fonctionnalité précise comme le routage ou les communications. Au sein de chaque couche, nous proposons de modéliser le réseau sous la forme d'un graphe, chaque nœud et lien ayant une spécificité déterminée. Le modèle permet ainsi d'analyser les impacts des fautes au travers des différentes couches et sa généricité nous permet de plonger des solutions existantes afin de comparer leur

[4]Selon le classement mondial `http ://www.top500.org/`.

performances en fonction de l'environnement de destination de l'application.

Une fois les différents algorithmes de l'application développés, une autre étape importante est la simulation. Elle permet d'observer le comportement des algorithmes en fonction des caractéristiques de l'environnement de destination. Nous avons développé une bibliothèque qui permet de construire des simulateurs dont le modèle d'exécution est calqué sur notre modèle théorique. La simulation est donc plus précise car elle permet au concepteur de jouer sur chaque couche du modèle et d'observer le comportement de ses algorithmes. L'écriture d'un simulateur est réalisée indépendamment des modèles de simulation. Ainsi, il est possible de jouer sur la finesse de la simulation en appliquant des modèles très précis pour simuler les protocoles de bas niveau ou au contraire appliquer des modèles plus abstraits pour diminuer le temps de calcul de la simulation. Ce changement du degré d'abstraction est réalisé à l'exécution et n'implique aucune modification du code du simulateur.

À partir de l'architecture et du modèle théorique, la conception d'une application passe par le choix d'outils adaptés. Nous nous sommes intéressés en particulier à concevoir des solutions complètement décentralisées et supportant un fort dynamisme des ressources. Nous avons eu recours à deux outils : la marche aléatoire et le mot circulant. Nos solutions sont basées sur le transfert successif d'un message, appelé jeton, entre les nœuds du graphe, le destinataire étant choisi aléatoirement parmi les voisins de l'émetteur. Le déplacement du jeton peut être vu comme une marche aléatoire. Les résultats théoriques sur les marches aléatoires nous permettent de calculer la complexité d'un algorithme basé sur la circulation aléatoire d'un jeton. Ce paradigme de circulation ne nécessite aucune connaissance globale du système, ce qui permet de concevoir des solutions complètement décentralisées. Les nœuds possèdent tous la même application : le déploiement est simplifié et la tolérance aux pannes est plus grande. De plus, le nombre de messages échangés est faible et ce, quel que soit le nombre de pannes dans le système. Le mot circulant est lui utilisé pour récolter de l'information au fur et à mesure de ses déplacements dans un réseau. En particulier, couplé avec une marche aléatoire, il peut récupérer des informations sur ses déplacements et donc, sur la topologie du réseau. Le contenu du mot circulant est ensuite utilisé pour la structuration des ressources.

À partir du modèle, nous avons proposé une solution pour gérer les ressources dans un réseau dynamique qui exploite ce couple d'outils. Une image partielle du graphe de communication est maintenue dans le mot circulant afin de répertorier les différentes ressources et de construire des chemins pour les atteindre. Cette solution est complètement distribuée et est peu coûteuse en terme de messages : à tout moment, seul un message transite dans tout le système quel que soit le nombre de pannes. Certains matériels réseau comme les pare-feux empêchent l'établissement de connexion entre les nœuds. Or, la gestion du mot circulant telle qu'elle a été proposée dans la littérature ne permet de gérer cette orientation des liens de communication. Nous avons donc proposé une nouvelle gestion pour construire des arbres aléatoires dans un réseau orienté.

Outre la gestion des ressources, l'un des composants essentiel dans les systèmes de calcul est la gestion des tâches. Nous nous sommes intéressés en particulier aux tâches indépendantes, indivisibles et irrégulières. Le calcul de ce type de tâche est en effet particulièrement adapté

dans les réseaux dynamiques. Nous proposons une solution complètement décentralisée basée sur les marches aléatoires qui laisse les nœuds en charge de l'assignation des tâches. Cette politique locale évite le recours à des nœuds distingués et limite l'émission de messages de contrôle. La gestion des tâches doit gérer l'assignation des tâches aux nœuds pour qu'elles puissent être calculées. Nous comparons deux méthodes d'assignation, appelées passive et active, en fonction des caractéristiques physiques de la grille et des tâches. À partir de ces résultats, nous proposons plusieurs optimisations afin d'améliorer leur efficacité. Ces différentes solutions proposent une efficacité très intéressante et ce, malgré un dynamisme important des ressources de calcul.

L'organisation de cette thèse est la suivante. Dans le premier chapitre, nous présentons les différentes applications de grille ou pair-à-pair existantes en nous focalisant sur le degré de centralisation de leur architecture ainsi que les différents mécanismes mis en place pour la gestion des pannes. Nous introduisons la notion de graphe qui nous permet de modéliser un système distribué et nous détaillons les deux outils que nous utilisons dans nos solutions : la marche aléatoire et le mot circulant. Nous présentons dans le chapitre 2, le modèle théorique que nous avons proposé en décrivant les mécanismes au sein de chaque couche. Nous présentons ensuite notre bibliothèque de simulation en détaillant ses différents composants et son utilisation. Dans le chapitre 3, nous présentons une gestion des ressources originale basée sur les marches aléatoires et le mot circulant. Nous décrivons la nouvelle gestion du mot circulant adaptée aux réseaux orientés. Dans le chapitre 4, nous détaillons deux méthodes d'assignation des tâches et nous en présentons une comparaison détaillée. Nous proposons différentes améliorations en exploitant notamment le contenu d'un mot circulant. Enfin, dans le dernier chapitre, nous proposons différentes optimisations que nous pouvons apporter aux applications exploitant les marches aléatoires et le mot circulant. Ces optimisations sont indépendantes des applications présentées dans les chapitres précédents.

Chapitre 1

Grilles et systèmes pair-à-pair : architecture et outils de conception

Résumé : *Ce chapitre propose un aperçu sur les différentes applications de grille ou systèmes pair-à-pair actuels. Quelle que soit la nature de l'application (grille, système d'échange de fichiers), nous détaillons leur architecture ainsi que les différents mécanismes déployés pour gérer le dynamisme des ressources (*i.e. *la connexion et la déconnexion des nœuds). Dans la première partie de ce chapitre, nous proposons de les classer en quatre catégories principales selon leur degré de centralisation.*

Les systèmes distribués (applications ou réseaux physiques) sont généralement représentés sous la forme de graphes. Dans la deuxième partie, nous donnons quelques définitions sur les graphes et nous décrivons les topologies les plus couramment rencontrées, utilisées dans les chapitres suivants pour simuler nos solutions.

À la fin de ce chapitre, nous présentons deux outils adaptés aux systèmes dynamiques et permettant de créer des solutions totalement décentralisées. D'une part, nous présentons les marches aléatoires dont les différentes grandeurs caractéristiques sont utilisées pour calculer la complexité d'algorithmes basés sur la circulation aléatoire d'un jeton. D'autre part, le mot circulant nous permet de structurer le réseau et nous détaillons les différents algorithmes utilisés pour sa gestion.

1.1 Introduction

Le paradigme de la grille informatique a été introduit dans [FK99]. C'est une infrastructure matérielle et logicielle dont le but est de connecter des ressources partagées, distribuées et hétérogènes. Les ressources propres d'un ordinateur qui se connecte à la grille sont mises à la disposition de l'ensemble des autres nœuds. Dans le même temps, l'ordinateur accède de manière transparente pour l'utilisateur, à l'ensemble des ressources de la grille. Celle-ci se comporte alors comme un ordinateur unique. L'objectif de la grille est donc d'apporter à l'utilisateur final la possibilité d'utiliser des ressources distantes ou de lancer une application qui demande beaucoup de ressources non disponibles localement.

La nature des ressources partagées est très diverse comme la capacité de calcul, les moyens de stockage, les applications ou encore le matériel scientifique. Comme le proposent les auteurs de [BBL02], il est possible de classer les grilles en fonction des services qu'elles apportent vis-à-

vis de l'utilisateur final. Ainsi, les **grilles de calcul** ont pour but de partager de la puissance de calcul en répartissant les tâches sur tous les nœuds. Les **grilles de données** sont destinées au partage, à l'accès et à la sécurité de données. Le principe de la réplication est mis en œuvre afin de diminuer le temps d'accès aux données ou pour augmenter la pérennité des données. Les grilles peuvent être aussi utilisées pour partager des applications. Des mécanismes sont mis en place afin de supporter l'accès à distance de bibliothèques ou d'applications de manière transparente.

Depuis plusieurs années, les systèmes pair-à-pair se sont développés, rendus populaires avec l'échange de fichiers via Internet, comme *Napster* [Nap]. Ils ont progressivement évolué eux-aussi vers d'autres services comme le calcul avec *CONFIIT* [FKF03], le stockage distribué avec *US* [RSUW05] ou la téléphonie IP avec *Skype*™[1]. La principale caractéristique par rapport à la grille réside dans leur architecture : dans les systèmes pair-à-pair, les nœuds possèdent les fonctions à la fois d'un serveur et d'un client (ils sont appelés *servent*) et peuvent communiquer directement entre eux. Les systèmes pair-à-pair sont aussi caractérisés par des communications ad-hoc, c'est-à-dire des communications point-à-point. Cette décentralisation permet le passage à l'échelle en réduisant la charge de nœuds dédiés. Les systèmes actuels comportent ainsi des millions d'utilisateurs. Les auteurs de [FI03] précisent cependant que les grilles et les systèmes pair-à-pair ont tendance à converger vers le même objectif, tout en partant d'un point de départ différent. Ils sont développés dans le but d'être tolérants aux pannes et de supporter une mise à l'échelle importante. Une des approches pour la conception d'applications de grille est donc basée sur le modèle pair-à-pair : les serveurs sont supprimés et les services rendus par ces serveurs sont répartis dans la grille.

Les systèmes de calcul (grilles ou systèmes pair-à-pair) peuvent être regroupés en plusieurs catégories comme le proposent les auteurs de [FK99]. Le **calcul haute-performance** (*distributed supercomputing*) est caractérisé par l'exploitation de ressources de calcul importantes comme des grappes de serveurs ou de machines parallèles. Dans ce cas, les communications entre les sites sont faibles. A l'opposé, le **calcul à la demande** (*on-demand computing*) est tourné vers la notion de performance à court terme : un calcul soumis doit être réalisé dans un temps le plus court possible. Il nécessite donc des communications plus importantes. Le **calcul axé sur les données** (*data intensive computing*) nécessite autant de puissance de calcul qu'une grosse capacité de communication entre les sites. Le but est de pouvoir exploiter des données, généralement de volume très important, réparties sur plusieurs sites distants. Le **calcul collaboratif** permet d'exploiter les ressources de la grille en fonction des interactions avec les hommes. La puissance de calcul est utilisée afin de répondre en temps réel aux attentes de l'utilisateur. Enfin, la dernière classe est le **calcul basé sur l'exploitation des ressources inutilisées** (*high-throughput computing*). On parle aussi de vol de cycles. Un type particulier est la *desktop grid*, autrement dit la grille basée sur l'exploitation des ordinateurs de bureau comme *SETI@home* [ACK+02]. Cette classe de grille est particulièrement adaptée pour la résolution de nombreuses tâches indépendantes.

La caractérisation des systèmes de calcul donne une indication sur le type d'applications qui peut y être exécuté. Pour les systèmes pair-à-pair de calcul répartis au travers Internet, les

[1]Site officiel de Skype http ://www.skype.com

ressources sont généralement de plus faible puissance et plus dynamiques. De même, le débit du réseau d'interconnexion est très variable. Les applications basées sur le vol de cycles sont donc plus adaptées que celles nécessitant du calcul haute-performance ou à la demande.

Une fois l'application et les ressources caractérisées, il est nécessaire de choisir une architecture pour l'application. Elle guide le développement de tous les composants de l'application. Le degré de centralisation de cette architecture est important du point de vue de la tolérance aux pannes, ainsi que de la possibilité pour l'application de passer à l'échelle. Nous présentons dans la première partie de ce chapitre, un aperçu sur les différentes architectures des applications de grille ou pair-à-pair existantes. Nous détaillons en particulier les différents mécanismes mis en place pour la gestion du dynamisme des ressources. Ensuite, nous introduisons les graphes qui sont une représentation classique des systèmes distribués. Nous détaillons les topologies de graphes les plus couramment rencontrées. Dans les chapitres suivants, nous nous basons sur ces topologies pour réaliser les simulations de nos solutions.

L'architecture totalement décentralisée est très souvent rencontrée dans les réseaux pair-à-pair. Aucun nœud n'est différencié et s'il tombe en panne, l'efficacité de l'application n'est que très peu affectée. D'autre part, l'absence de centralisation empêche les goulots d'étranglement. Cependant, il est nécessaire de structurer les nœuds pour accéder aux ressources. Pour cela, nous avons besoin d'outils adaptés. Nous nous sommes intéressés plus particulièrement aux marches aléatoires et au mot circulant. Ce couple d'outils permet de construire des structures aléatoires qui s'adaptent aux changements topologiques. De plus, cette construction est réalisée de manière totalement décentralisée et est peu coûteuse en terme d'échanges de messages. Dans la dernière partie de ce chapitre, nous détaillons ces deux outils.

1.2 Architectures et degré de centralisation

Le choix de l'architecture d'une application dépend de nombreuses considérations. Par exemple, il faut tenir compte des ressources disponibles dans le système et les mettre en regard avec les ressources nécessaires pour l'application. De même, l'architecture joue un rôle essentiel dans le taux de dynamisme des nœuds qui est supporté au cours de l'exécution. Dans cette section, nous proposons d'étudier quelques solutions existantes en fonction du degré de centralisation de leur architecture. Nous présentons aussi bien des applications de calcul, que des applications d'échange de fichiers. Nous distinguons principalement quatre niveaux de centralisation : centralisé, semi-centralisé, hiérarchique et décentralisé.

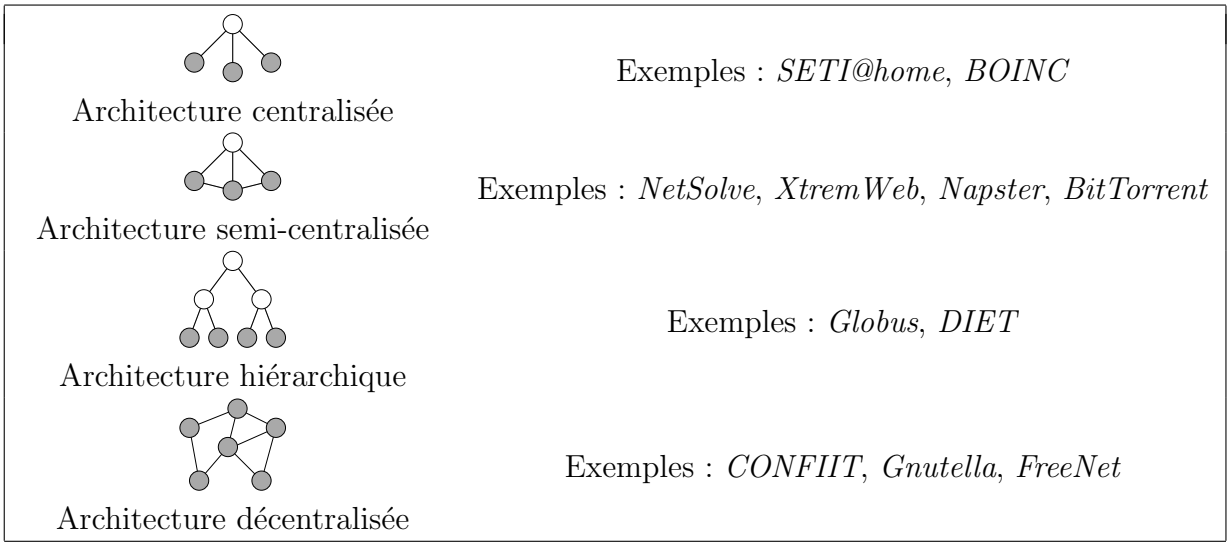


FIG. 1.1 – Les différents degrés de centralisation des applications de grille ou pair-à-pair.

1.2.1 Architectures centralisées

Une architecture complètement centralisée est constituée d'un serveur central sur lequel se connectent tous les clients. Ce type d'architecture est assez peu rencontré dans les grilles ou les réseaux pair-à-pair. En effet, le serveur central est un point critique. Premièrement, s'il tombe en panne, l'ensemble du système est inutilisable. Deuxièmement, le passage à l'échelle est plus difficile. Le serveur doit être en mesure d'accepter un grand nombre de connexions simultanément, ce qui implique un débit extrêmement important. De plus, pour traiter toutes les demandes, le serveur doit avoir une puissance de calcul suffisamment importante et un accès disque très rapide.

Le projet de calcul distribué le plus populaire est sans aucun doute *SETI@home* [ACK+02]. Il est basé sur le volontariat du public (en anglais *Public-Resource Computing*) et il compte des millions d'utilisateurs. Il est tourné vers une seule et unique application : la recherche d'intelligence artificielle (*SETI* pour *S*earch for *E*xtra*T*errestrial *I*ntelligence) en analysant des signaux collectés par le radio-télescope d'*Arecibo* dans sa première version et actuellement par le *Allen Telescope Array*.

Depuis sa mise-en-place, la version originale a été remplacée par une version à base de *BOINC* [And03] (pour *B*erkeley *O*pen *I*nfrastructure for *N*etwork *C*omputing). Son principe est identique à celui de *SETI@home* mais il permet de le généraliser pour d'autres applications du même genre comme *Einstein@home*[2] qui permet de détecter des pulsars dans l'espace ou *Chess960@home*[3] qui calcule des combinaisons du jeu d'échecs. Chacune est appelée un projet et est identifiée par une adresse Internet qui correspond à la page d'accueil du site Web. Les participants s'enregistrent à un projet en remplissant un formulaire et en téléchargeant un logiciel client *BOINC* à partir de ce site.

Lorsque le logiciel client s'exécute (pendant la mise en veille de l'ordinateur, par exemple),

[2]L'adresse du site Web de *Einstein@home* est http ://einstein.phys.uwm.edu/.
[3]La page officiel du projet *Chess960@home* est à l'adresse http ://www.chess960athome.org/alpha/.

il interroge le serveur (situé à Berkeley pour *SETI@home*) pour récupérer des données à analyser. Une fois les résultats obtenus, le logiciel client se connecte à nouveau au serveur pour lui envoyer les résultats et récupérer de nouvelles données.

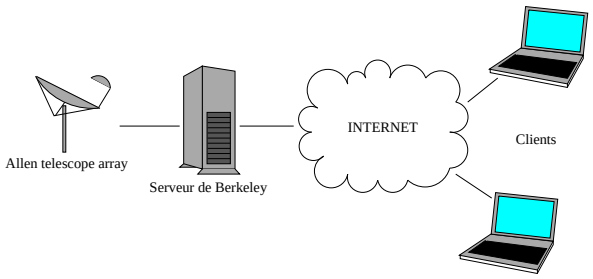


FIG. 1.2 – Architecture générale de *SETI@home* : les clients se connectent au serveur de Berkeley pour récupérer les données à analyser provenant du télescope ou pour retourner des résultats d'analyse.

La surcharge du serveur est importante dans une telle architecture, en particulier si des connexions simultanées des clients qui téléchargent les données sont très nombreuses. Si le serveur ne répond pas, un mécanisme de *backoff* est mis en place. Le client attend un certain temps qui croît exponentiellement avant de contacter à nouveau le serveur.

Des erreurs de calcul peuvent intervenir soit à cause des processeurs (dans le cas de *SETI@home*, l'analyse correspond au calcul d'une transformée de Fourrier) ou bien volontairement de la part de clients malicieux. Une redondance dans les calculs est nécessaire. Pour chaque projet, une constante est fixée correspondant au nombre de fois qu'un calcul doit être réalisé pour être validé (2 à 3 pour *SETI@home*). Le système s'assure que les mêmes calculs ne sont pas envoyés aux mêmes participants et pour distinguer les erreurs provenant de l'architecture de l'ordinateur, les calculs sont envoyés à une même communauté comportant une architecture semblable (processeur, système d'exploitation...).

Si un nœud tombe en panne pendant la transmission des données à analyser ou des résultats, il retente l'opération après un temps donné. Si la panne intervient pendant le calcul, ce dernier est relancé dès que possible. Enfin, si un nœud ne répond plus, les données à analyser sont attribuées à nouveau à un autre nœud après un délai variable.

Le choix d'une architecture totalement centralisée n'est pas préjudiciable pour de telles applications. Toutes les données à analyser provenant du télescope sont stockées sur des serveurs de données. Le délai entre la réception des données sur le serveur et leur traitement par les clients n'est pas critique, excepté en terme de stockage (si le nombre de clients est insuffisant). Ainsi, la perte d'un calcul comparée à la puissance disponible est négligeable[4]. Les tâches sont d'ailleurs calculées plusieurs fois. De même, les calculs ne nécessitent pas une connexion permanente entre le serveur et les clients et durent un temps relativement long. Une panne courte du serveur n'est pas préjudiciable comparée à la puissance de calcul disponible. Pour un système

[4]Selon le site de statistiques de *BOINC* à l'adresse http ://fr.boincstats.com/, la puissance cumulée dépassait les 500 TFlops en août 2007.

pair-à-pair dont les capacités de communication sont hétérogènes, une telle architecture est difficilement envisageable : le serveur doit être robuste et posséder un réseau très haut-débit.

1.2.2 Architectures semi-centralisées

Une architecture semi-centralisée est caractérisée par la présence d'un agent central qui sert d'intermédiaire entre les nœuds. Dans les systèmes pair-à-pair, il est appelé *agent de négociation* (pour *agent mediated*). Un cas particulier d'architecture semi-centralisée est l'architecture trois-tiers (les nœuds sont distingués en deux catégories : les clients et les serveurs). Contrairement à une architecture centralisée, les nœuds peuvent avoir des rôles différents : l'émetteur d'une requête (fichier ou calcul) et le (ou les) nœud(s) qui répond(ent) à cette requête. Dans les systèmes pair-à-pair, le rôle des nœuds n'est pas fixé et change tout au long de l'exécution. De plus, un pair peut être à la fois demandeur d'un fichier et source pour un autre fichier.

Les systèmes de calcul. Nous pouvons citer deux systèmes particuliers qui sont tous les deux basés sur une architecture trois-tiers : *NetSolve* (ou *GridSolve*) décrit dans [AAB+02], développé à l'Université du *Tennessee's Computer Departement*, et *XtremWeb* [CDF+04], développé par l'équipe du *LRI* de l'Université de Paris XI. Ils sont tous les deux constitués des entités suivantes :

- *Les clients* : ils soumettent des calculs dans la grille par l'intermédiaire de l'agent central et attendent les résultats.
- *Les serveurs* (ou *worker* pour *XtremWeb*) : ce sont les ressources de calcul. Dans *NetSolve*, ils exécutent des services particuliers (fonctions mathématiques, par exemple) alors que dans *XtremWeb*, ils sont en mesure d'accepter des applications transmises par le client.
- *L'agent* (ou *coordinateur* pour *XtremWeb*) : il traite les requêtes des clients.

La principale différence entre ces deux systèmes réside dans les communications entre les clients et les serveurs : dans *NetSolve*, les clients sont mis en relation directement avec les serveurs alors que dans *XtremWeb*, toutes les communications passent nécessairement par l'agent ce qui augmente la sécurité.

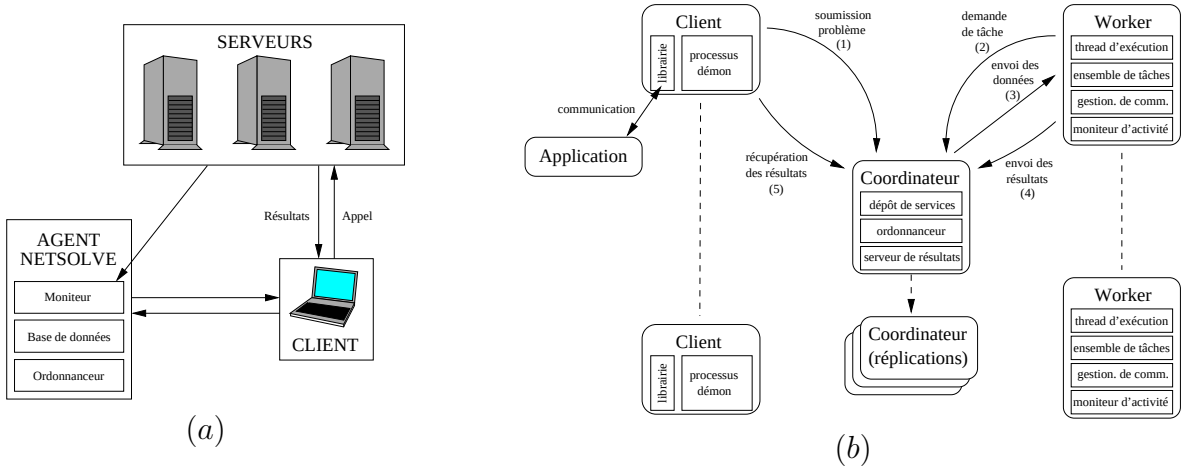


FIG. 1.3 – Exemple de deux architectures semi-centralisées : *NetSolve* (*a*) et *XtremWeb* (*b*).

Lorsqu'un client *NetSolve* désire soumettre un calcul, il interroge l'agent qui lui retourne une liste des serveurs en mesure de répondre à la requête. Le client interroge ensuite les serveurs dans l'ordre de la liste fournie par l'agent jusqu'à trouver un serveur disponible. Le client envoie ensuite à ce serveur les données pour le calcul et attend les résultats. Si une panne est détectée dans le système, le calcul est automatiquement soumis à un autre serveur.

Pour *XtremWeb*, les clients envoient leurs requêtes au *coordinateur*. Les serveurs interrogent ce dernier pour obtenir ces requêtes. Lorsque l'une d'entre elles est calculée, les résultats sont envoyés à l'agent (ou à un serveur dédié) où le client les récupèrent. Des mécanismes sont mis en place pour gérer la panne d'un *worker* et du *coordinateur*. Lorsqu'un calcul est exécuté sur un *worker*, ses paramètres sont sauvegardés au fur et à mesure de son exécution. En cas de panne, le calcul recommence à partir des derniers paramètres. De même, les résultats ne sont effacés sur le *worker* que lorsque le *coordinateur* lui envoie un accusé de réception. La panne d'un *worker* peut être détectée par le *coordinateur*. Si nécessaire, le calcul est réaffecté à un autre *worker*. Pour le *coordinateur*, un mécanisme de réplication est prévu. Dans ce cas, chacune de ses actions est répliquée sur un autre serveur qui peut le remplacer à tout moment.

Systèmes pair-à-pair d'échange de fichiers. L'architecture semi-centralisée est aussi rencontrée dans ces systèmes. Cependant, les pairs possèdent à la fois les fonctions des clients et des serveurs. *Napster*([Nap]) est considéré comme l'un des plus anciens réseau pair-à-pair d'échange de fichiers. Créé par Shawn Fanning en 1999, *Napster* était destiné à l'échange de fichiers musicaux. L'agent est une grappe centrale composée d'un grand nombre de serveurs dédiés qui maintient un index de tous les fichiers partagés par les pairs actifs. Chaque fois qu'un pair désire partager un fichier, il envoie sa description au serveur auquel il est connecté. Ce fichier est alors accessible par tous les autres pairs qui peuvent le récupérer en interrogeant les serveurs. Le transfert est réalisé par l'établissement d'une connexion directe entre le demandeur et le détenteur. Pendant un transfert, si l'émetteur du fichier tombe en panne, une nouvelle requête doit être émise. D'autres protocoles sont basés sur le principe de *Napster* comme *EDonkey* qui est le plus utilisé à l'heure actuelle, exploité notamment par le logiciel *EMule* [5].

Un autre protocole pour l'échange de fichiers a fait son apparition en 2002, développé par Bram Cohen et appelé *BitTorrent* [Coh03]. En particulier, il apporte une nette amélioration en terme de performances par rapport aux autres systèmes existants (*Napster*, *Gnutella*), en répartissant la charge du téléchargement sur l'ensemble des pairs. Lorsqu'un fichier est mis à disposition, il est publié sur un site Web sous la forme d'un fichier *.torrent* contenant le nom du fichier, sa taille, des informations de hachage ainsi que l'adresse Internet du *traqueur* (pour *traker*). Un *traqueur* met en relation l'ensemble des pairs qui téléchargent le fichier. Ces derniers envoient régulièrement des informations au *traqueur* (comme des statistiques d'utilisation) et en échange, ils reçoivent des informations sur les autres pairs.

Un fichier est décomposé en plusieurs blocs. Chaque fois qu'un pair télécharge un bloc, il le met à disposition pour les autres pairs. Ainsi, une graine (un nœud qui possède le fichier dans sa totalité) n'émet qu'une seule fois l'intégralité du fichier. La figure 1.4 montre un exemple du protocole pour l'échange d'un fichier composé de 4 blocs, entre une graine et quatre pairs.

[5] Le site officiel de *Emule* est à l'adresse http ://www.emule.com/.

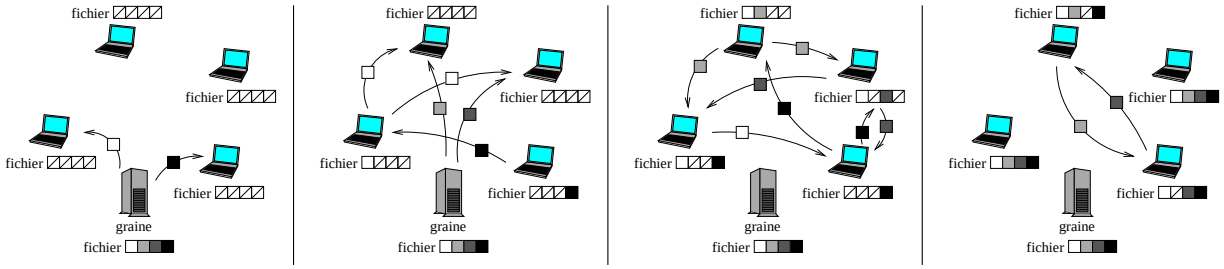


FIG. 1.4 – Exemple du transfert d'un fichier constitué de 4 blocs depuis une graine, avec le protocole *BitTorrent*.

Ces différentes solutions semi-centralisées sont basées sur un agent central servant d'intermédiaire entre le demandeur de la ressource (le client) et le possesseur de la ressource. Afin d'éviter que l'agent ne tombe en panne, il est nécessaire de prévoir des mécanismes qui peuvent être assez lourds et difficiles à mettre en place dans des réseaux totalement dynamiques. La réplication, par exemple, nécessite l'utilisation d'un autre serveur dédié et d'une capacité de communication importante entre les deux serveurs. Dans ce cas, cela implique une perte de la puissance de calcul. C'est pourquoi la gestion des pannes de ces solutions est principalement axée sur la déconnexion des clients ou des serveurs. L'agent central, quant à lui, doit être confié à un ordinateur très stable (ou à une grappe de serveurs dédiés comme c'est le cas pour *Napster*).

1.2.3 Architectures hiérarchiques

Dans une architecture semi-centralisée, l'agent central peut être un point critique, aussi bien au niveau de la tolérance aux pannes qu'au point de vue de la capacité à passer à l'échelle. Les solutions hiérarchiques proposent de diviser les fonctionnalités de l'agent sur plusieurs agents. La charge est donc répartie et si l'un des agents tombe en panne, il peut être remplacé dynamiquement.

Développé en 1996 par Ian Foster et Carl Kesselman, *Globus* [Fos06] fournit un ensemble de composants, appelé *Globus toolkit*, permettant de construire des applications de plus haut niveau. Parmi ces composants, se trouve en particulier *GRAM* [CFK+98] (pour *G*rid *R*esource *A*llocation *M*anager) qui gère la soumission des problèmes distants et leur exécution, ainsi que *MDS* [FFK+97] (pour *M*etacomputing *D*irectory *S*ervice) qui récupère des informations sur les ressources présentes dans la grille. *Globus* fournit de nombreux autres services que nous ne détaillons pas ici, comme des mécanismes d'authentification et de sécurité, de transfert de données, etc...

MDS est constitué de deux composants : un fournisseur d'informations *GRIS* (pour *G*rid *R*esource *I*nformation *S*ervice) et un annuaire global *GIIS* (pour *G*rid *I*ndex *I*nformation *S*ervice). Le *GRIS* fournit des informations sur une ressource spécifique, par exemple une grappe de serveurs. Un client interroge le *GRIS* pour obtenir des informations sur la grappe. Le *GIIS* rassemble l'ensemble des informations de tous les *GRIS* et peut être aussi organisé hiérarchiquement. Dans ce cas, les *GRIS* et *GIIS* s'enregistrent les uns sur les autres.

Toutes les requêtes qui sont échangées entre les différents composants pour la gestion des ressources sont exprimées dans le langage *RSL* (pour *R*esource *S*pecification *L*anguage). Les fonctions des différents composants sont les suivantes :

- *Resource Broker* : ils sont chargés de traduire les requêtes *RSL* en des spécifications concrètes (processus appelé spécialisation).
- *Co-allocateurs* : ils sont responsables de coordonner l'allocation et la gestion des ressources sur des sites multiples. Ils découpent les requêtes demandant des ressources situées sur plusieurs sites et diffusent les différentes parties aux gestionnaires de ressources appropriés.
- *Service d'information* : il permet de donner des informations pertinentes sur la disponibilité et la capacité des ressources. C'est le composant *MDS* qui est utilisé.
- *Gestionnaire de ressources* : ils sont responsables de récupérer les requêtes *RSL* et de les traduire en opérations sur le système de gestion de ressources local et spécifique à chaque site (exemples *LSF*, *EASY*, *NQE*, *Condor*, *Fork* ou *LoadLeveler*).

La gestion locale des ressources est implémentée par le composant *GRAM*. Il sert d'interface entre l'environnement de métacomputing et l'entité capable de créer des processus comme l'ordonnanceur d'un ordinateur parallèle ou *CONDOR* [FTL+02] par exemple. Ses interactions avec les autres composants de *Globus* sont décrits sur la figure 1.5.

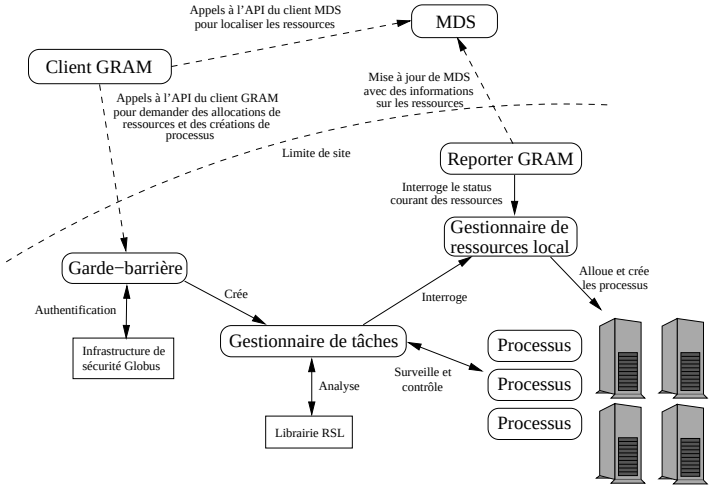


FIG. 1.5 – Aperçu des différentes interactions entre les composants de *Globus*.

DIET [CDL+02] (pour *D*istributed *I*nteractive *E*ngineering *T*oolbox) a été développé par l'équipe GRAAL[6]. C'est un ensemble hiérarchique de composants pour la construction d'applications basées sur les serveurs de calcul. L'objectif est de permettre à un client de localiser un serveur approprié à son problème. La figure 1.6 montre un aperçu de l'architecture de *DIET*. Les fonctionnalités des différents composants sont les suivantes :

- *Client*. C'est une application qui utilise *DIET* afin de résoudre des problèmes. Il est possible d'avoir plusieurs clients. Ils communiquent directement avec les *MAs* auxquels ils envoient leurs requêtes. Les problèmes sont soumis de manière synchrone ou asynchrone.
- *Master Agent (MA)*. C'est un serveur qui est directement relié aux clients. Il reçoit leurs requêtes et est capable de choisir des *SeDs* qui sont capables de les résoudre. Il les met

[6]http ://graal.ens-lyon.fr/DIET

alors en relation avec le client. Pour accélérer la recherche, il possède une vue globale de toute sa hiérarchie : problèmes pouvant être résolus, données distribuées.

– *Leader Agent (LA).* Il compose l'un des niveaux hiérarchiques de *DIET*. Il fait le lien entre un *MA* et un *SeD*, un autre *LA* et un *SeD* ou bien encore entre deux *LAs*. Son rôle est de diffuser les requêtes des *MAs* vers les *SeDs*. Il contient une liste des problèmes en cours ainsi que la liste de tous les serveurs situés dans ses sous-arbres avec le type de problème qu'ils peuvent résoudre.
– *Server Daemon (SeD).* Points d'entrée d'un serveur de calcul, ils gèrent des ensembles de *CRDs*. Ils sont sous la responsabilité d'un *LA*. Ils maintiennent la liste des données disponibles sur un serveur, une liste des problèmes qui peuvent être résolus et toutes les informations concernant sa charge. Sur une machine parallèle, un *SeD* est installé sur le frontal et sur chaque noeud se trouve un *CRD*.
– *Computational Resources Daemons (CRD).* Il constitue un ensemble de composants matériels et logiciels qui peuvent effectuer des calculs sur des données envoyées par un client ou un autre serveur. Il est en attente d'une demande de fonction de la part d'un *SeD*.

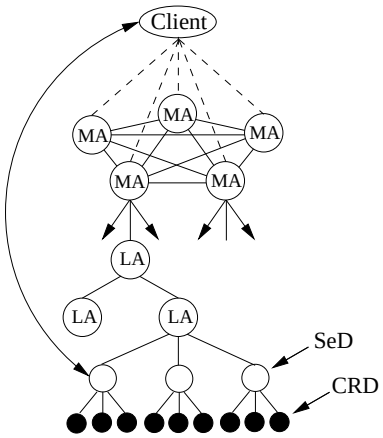


FIG. 1.6 – Hiérarchie des différents agents dans *DIET*.

La connexion de l'ensemble des composants de *DIET* se fait de manière hiérarchique du haut vers le bas : les *master agents* sont les premiers à s'initialiser. Puis les *leader agents* viennent s'y connecter et ainsi de suite jusqu'aux ressources. A tout instant, une nouvelle branche peut venir se connecter à la structure courante. Lorsqu'un *LA* est lancé, l'identité de son père doit lui être communiqué. Le *LA* lui transmet l'état de son arborescence et le prévient ensuite de changements locaux.

Lorsqu'un client désire résoudre un problème, il envoie une requête à son *MA* qui la transmet à toute son arborescence dans le but de trouver le meilleur *SeD* et, le cas échéant, elle est transmise aux autres *MAs* si elle ne peut pas être résolue. C'est l'outil *SLiM* qui permet de déterminer quel serveur est capable de résoudre le problème. Le choix entre les différents candidats potentiels est réalisé par l'outil *FAST* [DQS01] qui prend en compte les contraintes de charge ou de distance par rapport aux données. Lorsque le *SeD* a été choisi, il est mis directement en relation avec le client.

Pour la tolérance aux pannes, deux types d'erreurs sont reconnus : la mort d'un serveur et d'un agent. Dans le premier cas, un serveur de sauvegarde peut être utilisé avec des points de

recouvrement. Pour la mort d'un agent, c'est un fils qui la détecte. Dans ce cas, il y a plusieurs possibilités :

- le fils tente de contacter une nouvelle instance de son père ;
- en cas d'échec, le fils tente de contacter un agent situé au-dessus dans l'architecture (chaque agent ou serveur possède la liste de ses pères). Dans tous les cas, les agents tentent régulièrement de relancer leurs fils afin de reconstruire l'architecture originale.

Si *DIET* est particulièrement adapté pour les réseaux hiérarchiques, la mise en place de son architecture est assez délicate. Il est nécessaire de placer les agents au bon endroit dans le réseau pour optimiser les communications, ce qui nécessite de configurer convenablement chaque agent via leur fichier d'initialisation. Afin de faciliter le déploiement, un outil *GoDIET* [DC05] a été développé pour configurer plus aisément tous les composants par l'intermédiaire d'une interface graphique.

Une architecture hiérarchique propose une tolérance aux pannes accrue par rapport aux deux types d'architectures précédentes. Cependant, elle nécessite la mise-en-place de différents agents sur des serveurs qui doivent être configurés. *Globus* et *DIET* sont destinés à exploiter des machines puissances comme des ordinateurs parallèles ou des grappes de serveurs. De telles solutions sont en effet difficiles à adapter aux réseaux dynamiques comme les réseaux pair-à-pair. L'architecture mise en place n'est pas très flexible : dans le cas de *DIET*, un *master-agent* ne peut être remplacé dynamiquement par un agent inférieur. De plus, si un agent est installé sur un pair, la puissance de calcul de ce pair est détournée à la gestion et non plus au calcul proprement dit.

1.2.4 Architectures décentralisées

Dans les architectures précédentes, les rôles attribués aux nœuds sont différents, ce qui implique la mise en place de mécanismes pour pallier les pannes. Dans le cas contraire, une panne implique une indisponibilité du service (gestion des connexions et déconnexion, gestion des requêtes). Dans une architecture décentralisée, tous les nœuds possèdent potentiellement les mêmes capacités. En cas de panne, tout nœud est en mesure de prendre la place d'un autre. L'ensemble des services est ainsi réparti, ce qui implique aussi une possibilité accrue de passer à l'échelle. Dynamiquement, un nœud peut assurer un service et modifier ainsi son rôle dans l'application. La structuration des nœuds est utilisée dans certaines solutions basées sur une architecture décentralisée. Dans ce cas, la structure est dynamique, adaptative et tolérante aux pannes. Dans les réseaux pair-à-pair, nous parlons de réseaux *structurés* et *non-structurés*.

CONFIIT [FKF03] est un intergiciel de calcul pair-à-pair complètement distribué développé à l'Université de Reims Champagne-Ardenne. Tous les noeuds de la grille possèdent le même programme et ont les fonctionnalités à la fois d'un serveur (ordonnancement, gestion de la topologie) et d'un client (ressources de calcul et de stockage). *CONFIIT* a été écrit en *Java*, ce qui lui permet de pouvoir s'exécuter sur toutes les machines, quel que soit le système d'exploitation.

La topologie est gérée à l'aide d'un anneau virtuel mis en place au-dessus de la grille. Les noeuds communiquent par l'intermédiaire d'un jeton qui diffuse l'état du calcul (l'état de chaque tâche de chaque problème en cours). Chaque noeud possède trois services principaux : un gestionnaire de topologie et de communication, un gestionnaire de tâches et un ou plusieurs

solveurs dédiés à un problème. Pour une machine possédant plusieurs processeurs, un solveur par processeur est nécessaire.

Pour la gestion des tâches, tous les noeuds connaissent l'intégralité des tâches du système (*i.e.* les paramètres). Lorsque l'un d'entre eux désire débuter une tâche, il en sélectionne une au hasard. L'état de la tâche, marqué *en cours*, est diffusé dans l'anneau par le jeton afin d'éviter qu'un autre noeud ne la sélectionne. Enfin, lorsqu'un noeud finit une tâche, il marque son état comme *terminée* et en sélectionne une autre. Lorsque toutes les tâches sont calculées, les noeuds sélectionnent une tâche marquée comme *en cours*. Cela permet de gérer le cas où un nœud sélectionne une tâche et tombe en panne avant de l'avoir terminée. De plus, en affectant de nouveau cette tâche à un autre nœud éventuellement plus rapide, son calcul peut être obtenu plus rapidement.

Lorsqu'un nouveau noeud désire intégrer la grille, il contacte l'un des noeuds déjà présents appelé alors *point d'entrée*. Il vient se placer comme nouveau voisin de ce point d'entrée dans l'anneau virtuel. Les pannes sont détectées grâce à un autre jeton qui circule à contre-sens dans l'anneau. Si un nœud ne répond pas, l'émetteur du jeton met à jour la topologie et envoie le jeton au noeud suivant dans la grille. Chaque noeud connaît donc n voisins, où n est un paramètre de l'exécution. Dans ce cas, l'anneau accepte la déconnexion simultanée de n nœuds.

D'autres applications sont basées sur l'utilisation d'un anneau virtuel. *Pastry* [RD01] et *Chord* [SMK+01] ont pour but d'indexer les ressources dans les réseaux pair-à-pair en associant un identifiant unique à chacune et un mécanisme de routage pour les atteindre à partir de leur identifiant.

Gnutella [Gnu] est un protocole d'échange de fichiers. Il existe plusieurs applications qui exploitent ce même protocole. Contrairement à *Napster*, il n'y a aucune centralisation. Chaque pair qui désire intégrer un réseau *Gnutella*, doit contacter un noeud existant. Pour cela, il contacte un serveur qui lui envoie une liste aléatoire de nœuds. Après un certain nombre d'échanges, le nouveau noeud se connecte à plusieurs autres pairs. Il communique ensuite en envoyant et en recevant des messages du protocole *Gnutella* : *ping*, *pong*, *query*, *queryHit* et *push*.

Les messages *ping* et *pong* sont émis pour maintenir la structure du réseau et gérer ainsi les connexions et les déconnexions des nœuds. Lorsqu'un pair reçoit un message *ping* d'un voisin, il lui retourne un message *pong* et diffuse à tous ses autres voisins un message *ping*. Afin d'éviter la surcharge du réseau entraîné par ces messages, un mécanisme de durée de vie (ou *TTL* pour *Time-To-Live*) permet de limiter en profondeur le transfert des messages.

Pour la recherche de fichiers, le protocole *Gnutella* fonctionne aussi en utilisant l'inondation. Lorsqu'un pair recherche un fichier, il envoie à tous ses voisins un message *query* correspondant à sa requête. A sa réception, un pair commence par vérifier si le fichier existe dans son cache. Si c'est le cas, il renvoie un message *queryHit* à l'émetteur de la requête. Dans le cas contraire, il diffuse la requête à tous ses voisins. Dès que le fichier est trouvé, le pair qui a émis la requête commence le téléchargement du fichier en établissant une connexion directe avec le pair qui possède le fichier. La gestion des pairs protégés par des pare-feux est aussi prévue. Dans ce cas, l'émetteur de la requête demande explicitement au détenteur du fichier d'établir lui-même la connexion (messages *push*).

Le concept de *UltraKeeper* a été introduit depuis la version 0.6 du protocole *Gnutella*. L'objectif est de classer les noeuds du réseau en deux catégories : les clients normaux (*regular clients*) et les super noeuds (*super nodes*). Les super noeuds ont une longévité plus importante

dans le réseau et possèdent une bande passante plus importante. Ils servent de proxy pour les autres noeuds du réseau et sont les points d'entrée pour les clients normaux. De plus, ils permettent de réduire le nombre de messages générés dans le réseau en limitant le nombre de noeuds concernés dans le routage des messages.

Le stockage de fichiers est un autre champ d'application pour les systèmes pair-à-pair comme *US* [RSUW05] ou *FreeNet* [CSWH01] (et non pas d'échange comme c'est le cas pour *Gnutella* ou pour *Napster*). La particularité de *FreeNet* réside dans le total anonymat des participants et propose des communications sécurisées. Par contre, les utilisateurs ne peuvent pas lancer des recherches arbitraires : à chaque fichier est associé une clef et les recherches ne sont réalisées que sur ces clefs.

Au niveau de l'architecture, chaque noeud ne connaît que ses voisins proches dans le réseau. Ainsi, personne n'est en mesure de déterminer où se trouvent les données voulues. De même, chaque noeud est dans l'incapacité de savoir quelles données se trouvent actuellement en local : les fichiers sont cryptés et la clef se trouve sur d'autres noeuds. La clef de chaque fichier étant calculée par SHA-1, sa connaissance est insuffisante pour déterminer le contenu d'un fichier. Pour se connecter au réseau, un noeud doit nécessairement connaître l'adresse d'un noeud déjà présent dans le réseau.

Les architectures décentralisées sont adaptées aux réseaux dynamiques et sont plus aptes à passer à l'échelle. Sans structuration, le nombre de messages échangés est assez important dans des réseaux de type *Gnutella*. Pour réduire le trafic, une solution consiste à structurer le réseau, soit en anneau comme *CONFIIT*, ou hiérarchiquement (un arbre) comme les versions plus récentes de *Gnutella*. Contrairement aux architectures hiérarchiques décrites dans la section précédente, ses structures sont virtuelles et s'adaptent au dynamisme du réseau. Chaque nœud possède en effet la même application. Mais en contrepartie, la maintenance implique de nombreux messages de contrôle si le dynamisme des ressources est important.

1.2.5 Des grilles et systèmes pair-à-pair aux systèmes distribués

Le choix de l'architecture d'une application dépend de nombreux paramètres : le type de l'application, la tolérance aux pannes, les ressources du système (type de ressource, dynamisme des ressources) et la capacité à passer à l'échelle. Le choix d'une architecture centralisée permet de simplifier la maintenance du système. L'ensemble des services et des informations est regroupé sur une seule machine qui possède alors une vue d'ensemble. Cependant, chaque point de centralisation dans l'application implique une spécialisation des nœuds. Une telle spécialisation devient un point de fragilité du système. Si un nœud spécialisé tombe en panne ou s'il est saturé par un nombre trop important de requêtes, l'ensemble des services qu'il fournit devient inaccessible pour les autres nœuds. La décentralisation permet donc une tolérance aux pannes accrue mais complexifie la gestion. Les services sont distribués dans le système et des moyens de communication doivent être déployés pour mettre à jour une connaissance globale (état des calculs, localisation ou disponibilité des ressources). Dans ce cas, une structuration des nœuds peut être mise en place.

Pour apporter des solutions pour les grilles ou les systèmes pair-à-pair, nous nous basons sur la théorie des systèmes distribués. Ces derniers sont caractérisés par des entités reliées par

des canaux de communication et ils sont représentés sous la forme de graphes. Nous détaillons dans la section suivante, les différentes notions autour des graphes.

1.3 Représentation d'un système distribué

Un système distribué est classiquement représenté sous la forme d'un graphe. Chaque nœud de ce graphe représente une ressource ou un ensemble de ressources et les liens représentent les connexions physiques entre elles. Dans un premier temps, nous donnons quelques définitions à propos des graphes. Ensuite, nous présentons les topologies de graphe les plus couramment rencontrées dans la littérature.

1.3.1 Définitions

Un système distribué se modélise par un graphe $G = (V, E)$ où V est l'ensemble des nœuds et E l'ensemble des liens, avec $|V| = n$ et $|E| = m$. Un élément de V ou de E peut avoir plusieurs significations suivant la modélisation du système choisie : un réseau et ses composants physiques, une grille et ses différentes ressources ou un système pair-à-pair et ses pairs.

Définition 1.1 (Nœud) *Suivant le système modélisé, un nœud peut être un composant du réseau (ordinateur, machine parallèle ou routeur) ou bien un élément (appelé alors ressource) d'un composant du réseau.*

Ainsi, une grappe de serveurs peut être représentée par un seul nœud ou bien par un ensemble de nœuds, chacun représentant un serveur. Quel que soit le système, nous supposons que chaque nœud possède une identité unique.

Définition 1.2 (Lien) *Lorsque deux nœuds* i *et* j *du graphe sont reliés, il existe un lien* $(i, j) \in E$. *Si le graphe est non-orienté, nous avons* $(i, j) \in E \Leftrightarrow (j, i) \in E$.

Suivant le choix de la modélisation, un lien peut être de plusieurs types. En particulier, nous distinguons un lien physique qui représente un câble ou un bus, ainsi qu'un lien de communication qui représente un chemin parcourant plusieurs nœuds. Ainsi, $(i, j) \in E$ peut signifier qu'il existe un câble entre i et j ou bien que i peut envoyer des messages à j, éventuellement via l'intermédiaire d'autres nœuds.

Définition 1.3 (Voisin) *Soient deux nœuds* i *et* j, i *est dit voisin de* j *s'il existe un lien* $(j, i) \in E$. *Si le graphe est orienté,* j *n'est pas nécessairement voisin de* i.

La signification d'un voisin dans le cadre des systèmes pair-à-pair peut dépendre de l'application. Comme nous l'avons vu précédemment pour *Gnutella*, lorsqu'un nœud se connecte via un serveur, celui-ci lui fournit un sous-ensemble de nœuds avec lesquels il établit des connexions. Ces nœuds deviennent alors ses voisins dans l'application. Physiquement, ces nœuds sont reliés via Internet et peuvent être distants.

Définition 1.4 (Voisinage d'un nœud et degré) *L'ensemble de tous les voisins d'un nœud* i *est appelé le* voisinage *et est noté* $Vois_i$. *Le nombre de voisins, égal à* $|Vois_i|$, *est appelé son* degré *et est noté* $degré(i)$.

Définition 1.5 (Chemin) *Un chemin existe entre deux nœuds* i *et* j*, s'il existe une suite finie de liens consécutifs reliant* i *à* j*. Il est noté* $Ch(i,j)$ *et* $longueur(Ch(i,j))$ *est le nombre de liens de ce chemin.*

Définition 1.6 (Graphe connexe) *Un graphe non-orienté est dit* connexe *si et seulement si* $\exists i \in V, \forall j \in V, \exists Ch(i,j)$*. Dans le cadre des graphes orientés, un graphe est* fortement connexe *si* $\forall (i,j) \in V^2, \exists Ch(i,j)$*.*

Dans nos travaux, nous supposons que les graphes sont fortement connexes. Cependant, les solutions que nous proposons tolèrent la non-connexité du réseau durant un temps relativement court à la suite de changements topologiques. Au-delà d'un temps borné, les nœuds non-atteignables sont considérés comme déconnectés.

1.3.2 Topologies de base

Définition 1.7 (Graphe complet ou clique) *Un graphe est dit* complet *(on parle aussi de* clique*) si* $\forall (i,j) \in V^2, i \neq j \wedge j \in Vois_i$*.*

Le réseau complet est souvent utilisé pour représenter les systèmes pair-à-pair. La modélisation la plus courante consiste à supposer que chaque pair est en mesure de contacter tous les autres pairs du réseau à condition qu'il connaisse leur adresse Internet.

Définition 1.8 (Chaîne) *Une chaîne de taille* n *est un graphe connexe de degré maximum 2. Nous appelons les extrémités de la chaîne, les deux nœuds ayant un degré égal à 1.*

Dans les représentations courantes, les réseaux éthernet sont représentés sous la forme d'une chaîne. Il s'agit d'un bus sur lequel chaque ordinateur se connecte.

Définition 1.9 (Anneau) *Un anneau est une chaîne reliée par ses extrémités. Il peut être orienté ou non.*

L'intergiciel de calcul pair-à-pair *CONFIIT* maintient ses nœuds dans un anneau virtuel. D'autres systèmes comme *Chord* ou *Pastry* utilisent aussi l'anneau pour la recherche de données dans les réseaux pair-à-pair.

Définition 1.10 (Arbre) *Un arbre est un graphe connexe sans cycle. L'un des nœuds est appelé la* racine*.*

S'il existe un lien entre deux nœuds i et j d'un arbre, si i est plus proche de la racine, alors i est dit *père* de j et j est un fils de i. Un nœud qui ne possède pas de fils est appelé une *feuille*.

Dans *DIET*, les agents sont organisés hiérarchiquement sous la forme d'un arbre. De même, dans le protocole *Gnutella*, les nœuds sont organisés sous forme d'arbres dynamiques.

Définition 1.11 (Lollipop) *Un graphe* lollipop *est constitué d'une clique de* $n/2$ *nœuds reliée par l'un de ses nœuds à une chaîne de* $n/2$ *nœuds.*

Les graphes *lollipop* sont rarement rencontrés dans les systèmes distribués. Ils fournissent cependant un cas d'étude intéressant dans le cadre des marches aléatoires. Les auteurs de [BW90] montrent que sur ce type de graphe, les marches aléatoires présentent des résultats extrêmes. La figure 1.7 montre un graphe *lollipop* de 10 nœuds.

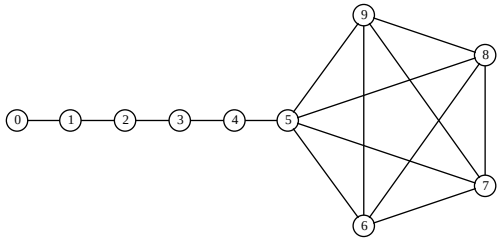


FIG. 1.7 – Exemple d'un graphe *lollipop* de 10 nœuds.

1.3.3 Topologies évoluées

Dans le cas le plus général, les graphes aléatoires sont utilisés pour représenter la topologie de réseaux quelconques. L'algorithme de génération a été proposé dans [ER59] : pour chaque couple de nœuds (i, j), il existe une probabilité p que le lien existe. Dans le cadre des graphes non-orientés, l'existence de (i, j) implique celle de (j, i).

Le graphe complet est souvent utilisé dans les réseaux pair-à-pair, en particulier dans les applications où des connexions sont établies entre chaque pair de nœuds. Pour un nombre important de nœuds, cette stratégie est plus difficile à mettre en œuvre, le nombre de connexions à maintenir sur chaque nœud étant trop grand. Dans *Gnutella*, lorsqu'un nœud se connecte, un ensemble de k voisins lui est attribué. Les nœuds qui restent longtemps dans le réseau possèdent alors un nombre de voisins important alors que les voisins qui sont connectés depuis peu ne possèdent que k voisins. Le graphe obtenu est donc un graphe à degré minimum qui dépend de k.

Définition 1.12 (Graphe à degré minimum) *Un graphe est dit à degré minimum δ si* $\forall i \in V, \text{degré}(i) \geq \delta$.

Plus généralement, des études sur les systèmes pair-à-pair d'échange de fichiers ont montré que les réseaux de recouvrement (le réseau formé par les pairs et les connexions entre les pairs) forment des graphes avec des propriétés particulières appelés graphes petit-monde [Mil67] : il existe une relation de distance maximale entre tous les nœuds du réseau. Dans la suite, nous utilisons l'algorithme de génération proposé dans [WS98]. Les auteurs partent d'un anneau à k voisins et avec une probabilité p, pour chaque nœud, les arêtes sont déplacées vers un nœud choisi aléatoirement. p est appelé la probabilité d'instabilité.

Enfin, nous pouvons citer les graphes *caveman*. Ils ont été introduits dans [Wat99] et sont formés par k sous-graphes disjoints fortement connexes (appelés *caves*) reliés entre eux

aléatoirement avec une probabilité p. Les réseaux *caveman* permettent, par exemple, de représenter des réseaux locaux, considérés alors comme les *caves*, reliés via Internet. La figure 1.8 montre un exemple de graphe *caveman* formé de 5 cliques.

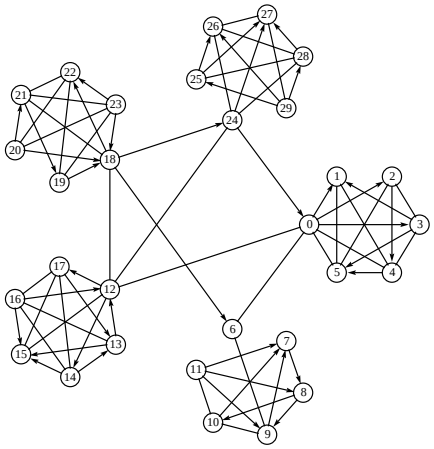


FIG. 1.8 – Exemple d'un graphe de type *caveman* constitué de 5 cliques de dimensions égales.

1.4 Les marches aléatoires

Dans les réseaux dynamiques, la conception d'applications décentralisées et tolérantes aux pannes est basée sur l'utilisation d'outils adéquats. Nous proposons ainsi d'utiliser les marches aléatoires. En effet, aucune structure n'est nécessaire pour la communication entre les nœuds. Le nombre de messages échangés est donc faible. De plus, les marches aléatoires permettent de créer des solutions complètement décentralisées. Comme nous l'avons vu précédemment, ce type d'architecture est plus adapté aux environnements dynamiques.

1.4.1 Introduction

Un message qui se déplace aléatoirement dans un graphe peut être vu comme une marche aléatoire. Ce message est appelé *jeton*. L'algorithme de circulation du jeton n'a besoin que du voisinage du nœud courant et aucune connaissance globale du système n'est nécessaire. Il est donc complètement décentralisé. Le calcul de la complexité d'une solution utilisant un jeton est basé sur les propriétés des marches aléatoires.

De nombreux algorithmes distribués sont basés sur la construction et la maintenance de structures couvrantes qui sont utilisées pour la communication entre les nœuds. De telles structures doivent être maintenues ce qui induit l'échange de messages de contrôle. Plus le réseau est dynamique, plus les échanges sont nombreux, provoquant une surcharge du réseau. De plus, dans de telles conditions, la structure risque d'être instable empêchant son exploitation. Or, une solution à base de marches aléatoires ne nécessite pas la mise en place de structure pour les communications, ce qui limite le nombre de messages de contrôle échangés. À tout moment, un seul message transite dans tout le système, ce qui économise la bande passante du réseau et

évite les phénomènes de congestion. De telles solutions sont donc adaptées aux réseaux à faible débit ou possédant des goulots d'étranglement.

Comme nous l'avons dit précédemment, une solution basée sur les marches aléatoires peut être complètement décentralisée, ce qui apporte plusieurs avantages. D'une part, le déploiement est facilité. Aucun nœud du réseau n'est différencié comme c'est le cas pour des solutions centralisées ou semi-distribuées. De telles solutions impliquent en effet le déploiement d'agents sur des nœuds dédiés. Avec une solution totalement décentralisée, chaque nœud du réseau possède la même application. D'autre part, cette facilité de déploiement est très avantageuse dans les réseaux dynamiques. L'architecture physique évolue constamment et la durée de vie des nœuds est difficilement prévisible. Or, une solution à base de marches aléatoires impose peu de conditions sur l'architecture et la durée de vie des nœuds. Cependant, des mécanismes doivent être mis en place pour gérer la perte ou la corruption éventuelle du jeton.

Les solutions basées sur les marches aléatoires sont multiples et couvrent un grand nombre de domaines. Tout d'abord, elles peuvent être utilisées pour la construction de structures couvrantes aléatoires. Les auteurs de [BIZ89] proposent un algorithme pour construire des arbres couvrants aléatoires. Des sous-arbres sont construits indépendamment dans le réseau et sont fusionnés au fur et à mesure pour n'obtenir qu'un seul arbre couvrant à la fin de l'exécution. Dans [Fla01], l'auteur propose un algorithme qui maintient un arbre couvrant qui peut être utilisé pour la construction de tables de routage. Le jeton circule infiniment dans le réseau et sans émission de message de contrôle supplémentaire, les tables de routage sont mises à jour.

Les auteurs de [BBFN06] proposent une application pour les réseaux ad-hoc. Ils utilisent les marches aléatoires dans le cadre de l'allocation de ressources : seul le nœud qui possède un jeton peut accéder à la ressource correspondante. Leur algorithme est basé sur la maintenance d'un arbre couvrant aléatoire qui évolue constamment au fil des déplacements de la marche aléatoire. À intervalles réguliers, des *vagues de réinitialisation* sont émises le long de cet arbre afin de prévenir les autres nœuds de l'existence du jeton et pour éviter ainsi, sa duplication.

Dans le cadre des réseaux pair-à-pair, les auteurs de [GMS04] montrent que les marches aléatoires produisent moins de messages par rapport à d'autres solutions basées sur l'inondation comme c'est le cas dans les réseaux de type *Gnutella*. L'un de ces algorithmes, proposé dans [DSW02], est basé sur l'émission de plusieurs marches aléatoires pour la recherche de fichiers dans un réseau pair-à-pair. Pour éviter l'inondation qui est coûteuse en terme de messages, un nœud envoie un nombre fixé de marches aléatoires dans le réseau, chacune avec une durée de vie limitée. Lorsqu'une marche arrive sur un nœud qui possède le fichier correspondant à la requête, une connexion est établie entre ce nœud et l'émetteur de la requête et le fichier est transféré.

Dans la section suivante, nous donnons différentes définitions relatives aux marches aléatoires. En particulier, nous présentons les grandeurs caractéristiques qui sont utilisées pour calculer la complexité d'algorithmes basés sur les marches aléatoires. Nous présentons aussi des mécanismes afin de pallier les différentes pannes qui peuvent intervenir dans le réseau. Ces mécanismes nous assurent qu'une marche aléatoire circule toujours dans le réseau et si ce n'est plus le cas, une nouvelle marche est créée. Nous présentons ensuite un outil, appelé le *mot circulant*, qui peut être utilisé conjointement à une marche aléatoire pour collecter de l'information sur les entités

du réseau. À l'aide de différentes méthodes appliquées sur ce mot, une image partielle du graphe de communication est maintenue et s'adapte aux changements topologiques.

1.4.2 Marches aléatoires et chaînes de Markov

La complexité d'un algorithme basé sur la circulation aléatoire d'un jeton se calcul à l'aide des grandeurs caractéristiques des marches aléatoires. La théorie des marches aléatoires est basée sur les chaînes de Markov.

Définition 1.13 (Chaîne de Markov) *La suite $X = (X_n/n \in \mathbb{N})$ de variables aléatoires à valeurs dans l'ensemble E fini ou dénombrable, est une chaîne de Markov si, pour tout entier positif n et tous $i, j, i_0, \ldots, i_{n-1} \in E$, nous avons :*

$$\begin{aligned}\mathbb{P}(X_{n+1} = j | X_0 = i_0, X_1 = i_1, \ldots, X_{n-1} = i_{n-1}, X_n = i) \\ = \mathbb{P}(X_{n+1} = j | X_n = i) = p_n(i, j)\end{aligned}$$

En d'autres termes, une chaîne de Markov est un processus stochastique qui possède la propriété markovienne : la distribution conditionnelle de probabilité des états futurs ne dépend que de l'état présent et non des états passés. On dit aussi qu'une chaîne de Markov est un processus sans mémoire. Une marche aléatoire dans un graphe est une chaîne de Markov finie.

Définition 1.14 (Graphe biparti) *Un graphe est dit* biparti, *si et seulement s'il existe une partition en deux sous ensembles de sommets dans lesquels les extrémités de chaque arête sont partagées.*

Définition 1.15 (Marche aléatoire) *Soit $G = (V, E)$ un graphe connexe, non biparti et non orienté, où $|V| = n$ et $|E| = m$, la marche aléatoire M_G est définie par :*

- *les états de M_G sont les sommets de G ;*
- *pour tout couple de sommets (i, j) de V, la probabilité de transition de i à j :*

$$P_{ij} = \begin{cases} 1/\textit{degré(i)} & \textit{si } (i,j) \in E \\ 0 & \textit{sinon} \end{cases}$$

La matrice $P = (P_{ij})_{(i,j)\in V^2}$ est la matrice de transition et on note P^t la $t^{ième}$ puissance de P.

Définition 1.16 (Régime permanent) *Si G est connexe et non biparti, alors $\lim_{t\to\infty} P^t$ existe. C'est une matrice Q dont les colonnes identiques $\pi = (\pi_i, i \in V)$, c'est-à-dire $\forall (i, j) \in V^2, \lim_{t\to\infty} P^t(i, j) = \pi_i$. On dit alors que la marche a atteint le régime permanent.*

Intuitivement, le régime permanent est atteint lorsque la marche aléatoire est suffisamment loin de son point de départ pour que ses déplacements soient indépendants de ce point de départ.

Définition 1.17 (Distribution stationnaire) *Le régime permanent est atteint lorsque :*

$$\Pi = \Pi \times P$$

Nous appelons Π la distribution stationnaire qui est unique.

La distribution stationnaire d'une marche aléatoire est égale à :

$$\pi_i = \frac{\textit{degré}(i)}{2.m}$$

1.4.3 Grandeurs caractéristiques et bornes

Nous distinguons trois principales grandeurs caractéristiques sur les marches aléatoires. Nous donnons ici leur définition ainsi que leurs bornes.

Définition 1.18 (Temps de percussion) *Le* temps de percussion *(ou* temps de visite *et en anglais* hitting time *ou* access time*) est le nombre moyen d'étapes ou de sauts nécessaires pour qu'un jeton partant du nœud i atteigne le nœud j. Il est noté $h(i,j)$.*

Dans [Lov93], l'auteur rappelle les bornes supérieures suivantes pour le temps de percussion entre deux nœuds arbitraires :

$$(4/27)n^3 - (1/9)n^2 + (2/3)n - 1 \text{ si } n \equiv 0 \ (mod\ 3)$$

$$(4/27)n^3 - (1/9)n^2 + (2/3)n - (29/27) \text{ si } n \equiv 1 \ (mod\ 3)$$

$$(4/27)n^3 - (1/9)n^2 + (4/9)n - (13/27) \text{ si } n \equiv 2 \ (mod\ 3)$$

Nous rencontrons aussi la notion de *temps de retour* qui est le temps moyen pour qu'un jeton partant du nœud i retourne sur le nœud i. En régime permanent, la valeur du temps de retour est :

$$h(i,i) = \frac{2.m}{degré(i)}$$

Définition 1.19 (Temps de commutation) *Le* temps de commutation *(en anglais* commute time*) est le nombre moyen d'étapes ou de sauts nécessaires pour qu'un jeton partant du nœud i atteigne le nœud j et retourne sur le nœud i. Il est noté $C(i,j)$ et vaut :*

$$C(i,j) = h(i,j) + h(j,i)$$

Définition 1.20 (Temps de couverture) *Le* temps de couverture *(en anglais* cover time*) est le nombre moyen d'étapes ou de sauts nécessaires pour qu'un jeton parcoure tous les nœuds du graphe. On note $C(i)$ le temps de couverture d'une marche partant du sommet i et $C = \max_{i \in V} C(i)$ le temps de couverture du graphe G où V est l'ensemble des nœuds de G.*

D'après le théorème de Matthews [Mat88], le temps de couverture est borné par :

$$\ln n \min_{i,j \in V} h(i,j) \leq C \leq \ln n \max_{i,j \in V} h(i,j)$$

Dans [Fei95a, Fei95b], les bornes sont données en fonction du nombre de nœuds. L'auteur montre que le temps de couverture est borné par :

$$(1 - o - (1))n \ln n \leq C \leq 4/27n^3$$

Définition 1.21 (Temps de couverture partiel) *Le* temps de couverture partiel *est le nombre moyen d'étapes ou de sauts nécessaires pour qu'un jeton parcoure une proportion p du graphe, p étant exprimé en pourcentage. On note $C_p(i)$ le temps de couverture partiel d'une marche partant du sommet i et $C_p = \max_{i \in V} C_p(i)$ le temps de couverture partiel du graphe G où V est l'ensemble des nœuds de G.*

Une proportion de 100% signifie que l'ensemble du graphe a été visité par une marche partant d'un nœud donné, le temps moyen nécessaire pour l'obtenir étant appelé le temps de couverture. Le temps de couverture partiel permet d'observer le temps nécessaire pour couvrir une proportion du graphe.

Définition 1.22 (Temps de rencontre) *Le* temps de rencontre *(en anglais* meeting time*) est le nombre moyen d'étapes ou de sauts nécessaires pour que deux marches aléatoires se rencontrent sur un nœud unique. Il est noté $M_G(u, v)$ avec u et v deux marches.*

Le temps de rencontre entre deux marches dépend de la méthode de déplacement de celles-ci. La plus générale est la méthode asynchrone, c'est-à-dire que les marches se déplacent l'une après l'autre, celle qui se déplace étant choisie aléatoirement. Dans [TW93], les auteurs proposent les bornes suivantes sur le temps de rencontre de deux marches u et v avec ce type de déplacement :

$$(1/27)n^3 \leq M_G(u, v) \leq (4/27)n^3$$

Les différentes bornes données précédemment sont en fonction du nombre de nœuds dans le graphe (excepté pour le théorème de Matthews). Or, le comportement d'une marche aléatoire varie énormément en fonction de la topologie du graphe sur lequel elle circule. C'est pourquoi les auteurs de [SB04, BS05] proposent des méthodes pour calculer la valeur exacte des différentes grandeurs. Ces méthodes de calcul sont basées sur le parallèle entre le comportement d'une marche aléatoire et celui d'un réseau électrique [Tet91, CRR+97]. En utilisant les lois classiques de l'électrocinétique, notamment la loi d'Ohm, de Kirchhoff et le théorème de Millman, les auteurs proposent des formules et des algorithmes pour calculer ces valeurs exactes. Cependant, ces méthodes nécessitent la connaissance complète du graphe.

1.4.4 Les marches aléatoires dans les systèmes distribués

Nous définissons le temps de communication d'un jeton t_c comme le temps nécessaire pour qu'un nœud qui a reçu le jeton, le traite (mise-à-jour des données du jeton et du nœud) et l'envoie à l'un de ses voisins. Il est donc composé de deux étapes : le temps de traitement local noté δ et le temps de transfert entre les deux nœuds noté τ. δ dépend du contenu du jeton, c'est-à-dire des données qu'il contient, et de la puissance de l'ordinateur qui le traite (puissance de calcul, mémoire ou espace disque). τ dépend aussi des données du jeton (plus la taille est importante, plus le temps de transfert est important), ainsi que des propriétés physiques du réseau : son débit, sa latence et son utilisation. La saturation du réseau entraîne une augmentation du temps de communication.

Le temps de traitement utilise de la puissance de calcul des nœuds qui ne peut être exploitée pour le calcul des tâches. Cette perte est d'autant plus importante si les nœuds reçoivent très souvent le jeton. De plus, le jeton est généralement utilisé pour mettre à jour des données sur les nœuds ou récupérer des informations topologiques. Une circulation trop rapide du jeton est donc inutile. Il est nécessaire de mettre en place un mécanisme pour le ralentir artificiellement. Nous introduisons donc le *temps de ralentissement*.

Définition 1.23 (Temps de ralentissement) *Le temps de ralentissement, noté t_r, est le temps pendant lequel un jeton est capté par les nœuds.*

Le temps de communication devient donc $t_c = \tau + \delta + t_r$. Le temps de ralentissement dépend du réseau, des capacités matériels ainsi que de l'application.

Le temps de rencontre entre deux marches est borné si plusieurs propriétés sont respectées. Par exemple, si les communications sont synchrones, il est possible que les marches se croisent sans jamais se rencontrer (cas du graphe biparti). Or, comme nous l'avons dit précédemment, le temps de communication fluctue en fonction du nombre de nœuds, des propriétés physiques du réseau traversé et varie au cours du temps. Nous sommes bien dans le cas de communications asynchrones, mais nous supposons que τ est borné. Cependant, si le temps de traitement du jeton est relativement court, il peut être traité et renvoyé à un voisin en un temps très court. La rencontre de deux jetons en un nœud unique est donc très difficile voire impossible. Or, en introduisant le temps de ralentissement, les jetons sont captés par les nœuds et placés dans une file d'attente durant le temps t_r, ce qui augmente la probabilité de rencontre entre plusieurs jetons sur un nœud unique. Ainsi, à la réception d'un jeton, le nœud vérifie si d'autres jetons sont actuellement dans sa file d'attente.

À partir des différentes observations précédentes, nous proposons l'algorithme 1 qui décrit le comportement général d'un nœud à la réception d'un jeton. Lorsqu'un jeton est reçu, il doit être comparé aux autres jetons présents éventuellement dans la file d'attente : sa validité est testée et des données peuvent être transférées d'un jeton à l'autre. En cas de validité, les données du jeton et les données locales du nœud sont mises à jour. Le jeton est ensuite placé dans la file d'attente. Après un temps t_r, il est retiré de la file d'attente. Si t_r est assez grand, les données locales peuvent avoir changées et les données du jeton doivent être mises à jour à nouveau. Dans le cas contraire, cette phase de mise-à-jour peut être ignorée. Enfin, le jeton est envoyé aléatoirement à l'un des voisins du nœud.

Algorithme 1 Réception du jeton J sur un nœud i

```
Si il y a plusieurs jetons dans la file d'attente Alors
  Si J est valide Alors
    Fusionner les informations de J et des autres jetons
  Sinon
    Détruire J
    Fin de l'algorithme
  Fin Si
Fin Si
Mise-à-jour des données de J et des données locales de i
Placer J dans la file d'attente
Attendre un temps t_r
Mise-à-jour des données de J /* si les données locales ont changé */
Envoyer J à un voisin de i
```

1.4.5 Tolérance aux pannes de solutions à base de jeton

Dans l'introduction de ce chapitre, nous avons avancé que les marches aléatoires sont particulièrement adaptées aux systèmes dynamiques. En particulier, le mouvement du jeton est

aléatoire et il n'est donc pas nécessaire de mettre en place de structure sous-jacente, ce qui limite le nombre de messages de contrôle émis dans le réseau. Cependant, une marche aléatoire est basée sur le transfert du jeton d'un nœud à l'autre et, en fonction du réseau et des différents protocoles sous-jacents, il est essentiel de s'assurer qu'il ne disparaisse pas du système suite à une panne quelconque. Nous avons vu dans la section 2.2.2 que les algorithmes basés sur les jetons doivent gérer plusieurs cas, en accord avec notre modèle théorique.

La perte de jeton. Par exemple, une perte de jeton survient lorsqu'un nœud possède le jeton et tombe en panne avant de pouvoir le transférer à l'un de ses voisins. Mais il est aussi possible que le protocole de communication ne prévienne pas la perte des messages. Dans ce cas, un message envoyé peut être perdu sans que l'émetteur ne s'en aperçoive, entraînant ainsi la perte du jeton et l'arrêt de la marche.

La duplication de jeton. Suivant le protocole de communication sous-jacent et suite à des problèmes de congestion ou d'émission, un message peut être dupliqué, ce qui entraîne la création de plusieurs marches aléatoires. Afin de limiter le nombre de messages produits, il est important de pouvoir contrôler le nombre de jetons en supprimant ceux qui sont superflus. La fusion des informations contenues dans chacun d'entre eux peut aussi permettre d'augmenter l'efficacité de l'application.

La corruption de jeton. Dans le cas d'une "*simple*" marche aléatoire, aucune corruption du jeton n'est possible puisqu'aucune donnée n'est contenue dans le message. Cependant, le jeton est souvent utilisé pour diffuser ou récolter de l'information. Il est donc important de prévoir un mécanisme pour corriger les erreurs éventuelles. Comme il dépend de l'application, nous ne nous intéressons pas à ce point dans cette section.

La gestion de la topologie dans l'intergiciel *CONFIIT* est basée sur la circulation d'un jeton au sein d'un anneau virtuel. Les auteurs de [FKF03] proposent de mettre en place sur chaque nœud un compte-à-rebours T_i qui est réinitialisé à chaque passage du jeton. Lorsque le compte-à-rebours d'un nœud arrive à terme, le nœud considère que le jeton a disparu du réseau. Il en crée un nouveau à partir de ses informations locales. Cette méthode est très simple à mettre en œuvre mais peut entraîner la création de plusieurs jetons simultanés. C'est pourquoi chaque jeton est marqué par une identité unique $id_J = (idNœud, idMsg)$, où $idNœud$ est l'identité du nœud qui a créé le jeton et $idMsg$ est la valeur du compteur de messages du nœud au moment de la création du jeton. Ce compteur est initialisé à 0 et est incrémenté à chaque fois qu'un nœud crée un nouveau jeton.

Lorsqu'un jeton est reçu sur un nœud i, une comparaison est réalisée entre id_J et $id_i = (idNœud_i, idMsg_i)$ qui est l'identité du dernier jeton valide reçu. Cette comparaison permet au nœud de vérifier si le jeton est valide. Si c'est le cas il le transmet à l'un de ses voisins, sinon il le détruit. La comparaison suit les règles suivantes :

- $idMsg < idMsg_i$: le jeton n'est pas valide et doit être supprimé ;
- $idMsg > idMsg_i$: le jeton est valide et la trace sur le nœud peut être mise à jour ;
- $idMsg = idMsg_i$: cette donnée est insuffisante pour déterminer la validité du jeton, il est nécessaire de comparer l'identité du nœud qui a créé le jeton :
  - $idNœud < idNœud_i$: le jeton est obsolète et peut donc être supprimé ;

– $idNœud \geq idNœud_i$: le jeton est valide et la trace sur le nœud est mise à jour.

Cette solution a été proposée dans le cadre d'un déplacement d'un jeton dans un anneau. En choisissant des bornes adéquates pour l'initialisation des compte-à-rebours, nous proposons d'exploiter cette solution dans le cadre des marches aléatoires.

Cette méthode est satisfaisante dans la plupart des cas, mais certaines solutions nécessitent l'unicité du jeton dans le système. Dans [BBF04b], les auteurs proposent de contrôler de manière plus stricte le nombre de jetons créés. Pour cela, en plus des compte-à-rebours sur chaque nœud, un arbre couvrant est maintenu au sein du jeton et mis à jour au fur et à mesure des déplacements. A intervalles réguliers et à l'initiative du jeton, une vague appelée *vague de réinitialisation*, est émise le long de cet arbre. Les nœuds sont ainsi prévenus de l'existence du jeton et leur compte-à-rebours est réinitialisé. Les auteurs prouvent en particulier que dans le cadre normal, c'est-à-dire sans faute, et si tous les nœuds ont été visités, il n'y aura aucun jeton superflu créé. Une des applications possibles proposée est une solution à l'allocation de ressources. Le nœud qui possède le jeton est le seul à pouvoir accéder à la ressource critique.

L'arbre maintenu dans le jeton se présente sous la forme d'un tableau de pères : à chaque nœud, l'identité de son père lui est associée, le nœud racine étant lui-même son père. Lorsqu'un nœud reçoit le jeton, il devient la racine de l'arbre et l'émetteur devient son fils. Une fois tous les nœuds du réseau visités, le tableau est complet et forme un arbre couvrant qui est toujours enraciné sur le nœud possédant le jeton. Il faut noter que l'algorithme présenté fonctionne avant que le tableau ne soit complet, mais dans ce cas, la création d'un jeton superflu est possible.

Exemple 1.1 *La figure 1.9 présente un exemple de la vague de réinitialisation. Le jeton, représenté par le point noir, s'est déplacé suivant le parcourt : 0, 2, 1, 2, 4, 5, 3. La figure montre le tableau des pères obtenu par cette circulation, ainsi que l'arbre couvrant utilisé pour la diffusion de la vague (indiqué sur la figure par les flèches). La figure de droite montre la modification de cet arbre après le déplacement sur le nœud 2 qui devient la racine.*

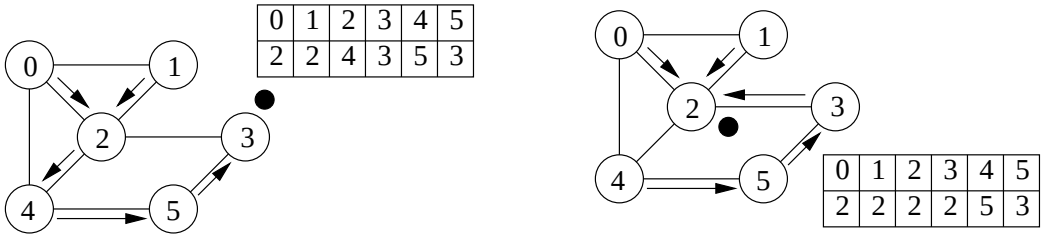


FIG. 1.9 – Exemple d'un arbre créé par la circulation d'un jeton arrivé sur le nœud 3 (a) puis après le déplacement sur le nœud 2 (b), avec l'algorithme proposé dans [BBF04b].

1.5 Le mot circulant

Comme nous l'avons précisé précédemment, la marche aléatoire est un mécanisme sans mémoire. Lorsqu'un nœud reçoit le jeton, il est dans l'incapacité de connaître le chemin qui a été parcouru par le jeton pour arriver jusqu'à lui, exceptée la dernière étape (l'identité de

l'émetteur du jeton est supposée connue). Le mot circulant est un outil qui permet de collecter des informations sur les nœuds au cours de sa visite. En particulier, il peut être utilisé pour récupérer les identités des nœuds visités par une marche aléatoire. Nous présentons dans la première section les méthodes utilisées pour gérer le contenu de ce mot qui ont été introduites dans [Fla01]. Dans la section suivante, nous détaillons sa gestion dans le cadre de pannes proposée dans [BBF04a].

1.5.1 Gestion du contenu du mot circulant

Le mot circulant a été introduit dans [Lav86] pour la détection de terminaison d'algorithmes distribués. À l'aide de mots émis et transmis à tous les nœuds du graphe, la suite des identités collectées permet de détecter les cycles éventuels dans le graphe de communication. Deux identités successives dans le mot signifient qu'il existe un arc entre ces deux identités (ou une arête si nous travaillons dans des graphes non-orientés).

Le mot circulant noté W peut donc être considéré comme une liste d'identités et nous notons W_i le i^{eme} élément de la liste où $W_i \in [1, n]$ avec "n" le nombre de nœuds du graphe. Les fonctions suivantes sont utilisées pour la gestion de cette liste :

- *taille(W)* : retourne le nombre d'identités contenues dans le mot ;
- *identités(W)* : retourne l'ensemble des identités (disjointes) contenues dans le mot ;
- *ajouter(W,e)* : ajoute l'identité "e" dans le mot (*i.e.* après l'ajout $W_1 = e$) ;
- *insérer(W,i,e)* : insère dans le mot l'identité "e" à la position "i" ;
- *supprimer(W,i,j)* : supprime le sous-mot $< W_i, \ldots, W_j >$. On utilise aussi la variante $supprimer(W, i)$ qui supprime l'identité à la position "i" ;
- *droite(W,i)* : retourne le sous-mot $< W_i, \ldots, W_{taille(W)} >$;
- *gauche(W,i)* : retourne le sous-mot $< W_1, \ldots, W_i >$;
- *premièreOccurrence(W,e)* : retourne la position de la première occurrence de l'identité "e" à partir de "W_1" ;
- *déplacer(W,i,j)* : déplace W_i à la position j.

À partir de cette liste, il est possible de construire un arbre couvrant à l'aide de l'algorithme 2. Il faut parcourir le mot de gauche à droite et ajouter toute identité W_i comme étant père de l'identité W_{i+1}, sauf si W_{i+1} est déjà dans l'arbre. Par défaut, l'arbre est enraciné en W_1. En sachant que le graphe est non-orienté, l'arbre peut donc être enraciné en tout nœud.

Algorithme 2 Construction de l'arbre $\mathcal{A}$ à partir du mot circulant W

```
A ← W_1
Pour k = 1 à taille(W) − 1 Faire
  Si (W_{k+1} ∉ A) Alors
    ajouter(A, W_k, W_k + 1)
  Fin Si
Fin Pour
```

Dans [Fla01], l'auteur propose une adaptation du mot circulant pour la construction d'un arbre couvrant dans un graphe non-orienté. Un mot circulant se déplace aléatoirement et à partir des identités des nœuds visités, un arbre est maintenu au sein du mot. Afin qu'il s'adapte

au dynamisme du graphe, le jeton circule infiniment. Puisqu'à chaque réception du mot, une nouvelle identité est ajoutée au mot, sa taille croît à l'infini. L'auteur propose donc plusieurs coupes pour le réduire. Pour cela, il introduit la notion d'occurrence constructive.

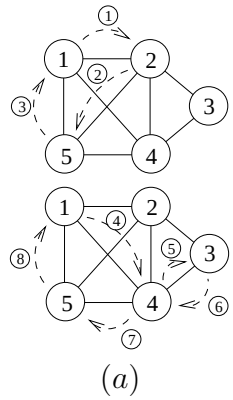
Définition 1.24 (Occurrence constructive) *Une occurrence dans le mot est dite constructive si elle permet d'ajouter un nouveau nœud dans l'arbre couvrant construit.*

En particulier, l'auteur prouve qu'une occurrence d'un nœud i à la fin du mot est constructive si et seulement si elle n'existe pas dans le reste du mot (cette coupe est appelée *coupe terminale*). De même, pour une identité i à la position k dans le mot, si i existe avant la position k et que l'identité à la position $k+1$ existe avant la position $k+1$, alors l'occurrence à la position k n'est pas constructive et peut être supprimée (cette coupe est appelée *coupe interne*). L'auteur propose l'algorithme 3 afin de réduire la taille du mot circulant à au plus $2n-1$ identités.

Algorithme 3 Réduction du mot circulant W

visités $\leftarrow \emptyset$
Pour $k = 1$ à $taille(W)$ **Faire**
  Si $k < taille(W)$ **Alors**
    Si $((W_k \in visités) \wedge (W_{k+1} \in visités)) \vee (W_k = W_{k+1})$ **Alors**
      $supprimer(W, k)$
    Fin Si
  Sinon
    Si $W_k \in visités$ **Alors**
      $supprimer(W, k)$
    Fin Si
  Fin Si
  visités $\leftarrow$ *visités* $\cup \{W_k\}$
Fin Pour

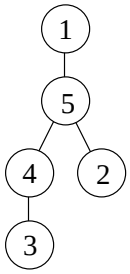
Exemple 1.2 *La figure 1.10 montre un exemple de circulation d'un mot circulant dans un graphe. Le contenu du mot est mis à jour à chaque déplacement et les informations redondantes sont supprimées à l'aide des coupes interne et terminale.*



(*a*)

- $< 1 >$ (création par le nœud 1)
(1) $< 2, 1 >$
(2) $< 5, 2, 1 >$
(3) $< 1, 5, 2, \boxed{\cancel{1}} >$ (coupe terminale)
(4) $< 4, 1, 5, 2 >$
(5) $< 3, 4, 1, 5, 2 >$
(6) $< 4, 3, 4, 1, 5, 2 >$
(7) $< 5, 4, 3, 4, 1, 5, 2 >$
(8) $< 1, 5, 4, 3, \boxed{\cancel{4,1}}, 5, 2 >$ (coupe interne)

(*b*)



(*c*)

FIG. 1.10 – Exemple de déplacements d'un mot circulant dans un graphe (*a*) avec les différentes coupes effectuées (*b*). L'arbre obtenu est représenté sur la figure (*c*).

1.5.2 Gestion des fautes liées au mot circulant

Par nature, le mot circulant doit être placé dans un jeton qui circule dans le réseau. Nous avons montré dans la section 1.4.5 différents mécanismes pour gérer la duplication ou la disparition d'un jeton. S'il contient un mot circulant, ces mécanismes peuvent être appliqués et différentes optimisations sont possibles.

La perte de jeton. Lorsqu'un jeton disparaît du réseau, les informations contenues dans le mot circulant sont perdues. Comme nous avons vu dans la section 1.4.5, un nouveau jeton peut être recréé à l'aide du mécanisme de compte-à-rebours. Dans ce cas, il contient un mot réduit à une seule identité qui doit recouvrir le graphe pour se compléter. Pour limiter ce phénomène, lorsqu'un mot est reçu sur un nœud, une trace peut être sauvegardée. Ainsi, lors de la régénération, elle est placée dans le mot. Les informations topologiques sont anciennes, mais nous verrons dans la suite qu'elles sont corrigées si nécessaire étape-par-étape, uniquement avec la circulation du jeton et des données locales aux nœuds (leur voisinage).

La duplication de jeton. Les différents mécanismes expliqués dans la section 1.4.5 visent à réduire le nombre de jetons dans le graphe. Pour le mot circulant, ils impliquent des conséquences très différentes : si des jetons se retrouvent sur le même nœud, il est possible de fusionner les informations topologiques et de ne garder qu'un seul jeton. Par contre, si nous utilisons le mécanisme des identités expliqué dans la section 1.4.5, seul le mot circulant le plus récent est conservé. Or, il y a une probabilité plus grande qu'un ancien mot contienne plus d'informations topologiques qu'un mot récent. Si les nœuds possèdent une trace du dernier mot circulant, il peut être intéressant d'injecter les informations topologiques contenues dans ce mot dans le jeton qui est plus récent. L'algorithme 4 décrit ce mécanisme. Il ne doit être exécuté que lorsqu'un jeton avec une identité différente et plus récente est reçue.

Algorithme 4 Injection des données du mot contenu sur un nœud W_N dans le mot du jeton W_J

Pour tout $i \in identités(W_N) \backslash identités(W_J)$ **Faire**
  $père \leftarrow W_{premièreOccurrence(W_N,i)-1}$
  Si $W_{J\,taille(W)} \neq père$ **Alors**
    $insérer(W_J, taille(W_J)+1, père)$
  Fin Si
  $k \leftarrow premièreOccurrence(W_N, i)$
  Tant que $W_{Nk} \notin gauche(W_N, k-1) \wedge W_{Nk} \notin identités(W_J)$ **Faire**
    $insérer(W_J, taille(W_J)+1, W_{Nk})$
  Fin Tant que
Fin Pour

Exemple 1.3 *Soient deux mots circulants* $W1 =< 1,2,3,2,5 >$ *et* $W2 =< 1,3,6,4,2 >$ *reçus simultanément sur le nœud 1. Les deux mots possèdent tous les deux des informations topologiques originales. Suivant l'algorithme 4, les données de* $W2$ *sont insérées dans* $W1$*. La figure 1.11 présente les arbres qui correspondent aux deux mots ainsi que l'arbre obtenu après la fusion, le mot circulant obtenu étant* $W =< 1,2,3,2,5,3,6,4 >$.

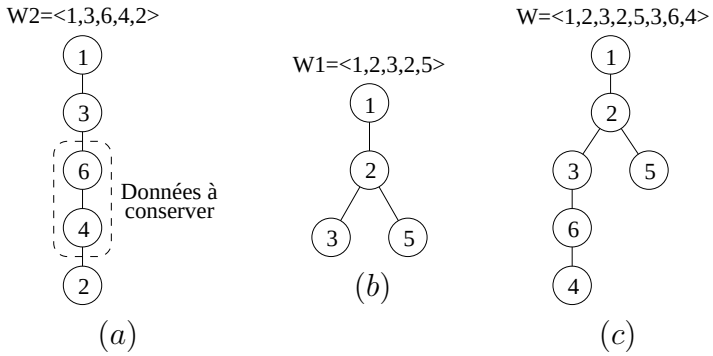


FIG. 1.11 – Exemple de fusion d'un mot circulant (a) dans un autre (b). Le résultat est représenté sur la figure (c).

La corruption du jeton. Le mot circulant peut être vu comme une image partielle du graphe de communication. Cette image a donc une durée de vie limitée dans le cadre des réseaux dynamiques. Ainsi, différentes incohérences topologiques peuvent apparaître dans le mot circulant. Si une corruption du jeton survient, elle peut être assimilée à une incohérence topologique. Dans [Fla01], les incohérences ne sont pas corrigées. Si un arbre est construit avec une branche (i, j) et que le nœud j disparaît, l'arbre n'est pas modifié et le sous-arbre enraciné en j est conservé dans le mot. Les auteurs de [BBF04a] proposent donc un mécanisme pour supprimer ces informations inutiles. Afin de ne produire aucun message autre que le jeton, ils proposent de corriger le mot à partir des informations locales des nœuds. À la réception du mot, chaque nœud vérifie pour toutes les occurrences de son identité dans le mot si les nœuds enracinés sur lui sont toujours dans son voisinage. Si ce n'est pas le cas, la partie de droite depuis cette occurrence est supprimée. Ainsi, si des incohérences se trouvent dans le mot, elles sont corrigées étape par étape en visitant tous les nœuds concernés.

L'algorithme 5 présente l'algorithme du test de cohérence local exécuté sur chaque nœud à la réception d'un mot circulant.

Algorithme 5 Test de cohérence local sur le nœud i à la réception du mot W

$k \leftarrow 1$
Tant que $k < taille(W)$ **Faire**
  Si $W_k = i$ **Alors**
    Si $(W_{k+1} \notin identités(gauche(W,k)) \wedge (W_{k+1} \notin Vois_k)$ **Alors**
      $W \leftarrow gauche(W,k)$
    Fin Si
  Fin Si
  $k \leftarrow k + 1$
Fin Tant que
Réduction du mot circulant (voir algorithme 3)

Exemple 1.4 *La figure 1.12 montre un exemple de correction d'un mot circulant. Sur la figure (a), le nœud 4 reçoit un mot circulant et après mise-à-jour, le mot devient* $W =< 4, 5, 1, 2, 1, 0, 3 >$. *Le mot est envoyé au nœud 1 qui est l'un des voisins de 4. Suite à la panne du nœud 0, le mot reçu contient des incohérences topologiques : le nœud 0 est enraciné sur le*

nœud 1 mais n'appartient plus à son voisinage. La partie de droite à partir de l'occurrence de 1 est supprimée et le mot devient : $W =< 1, 4, 5, 1, 2 >$.

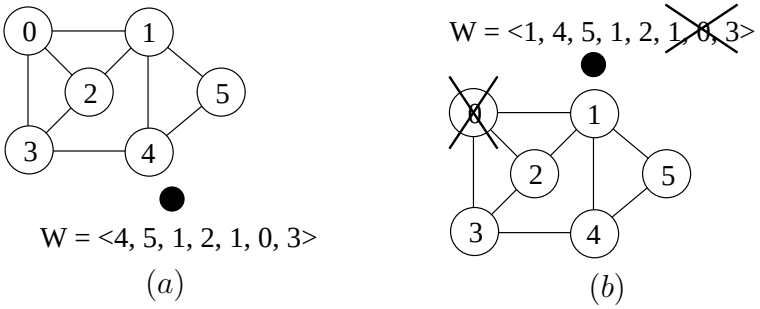


FIG. 1.12 – Exemple de correction d'incohérences topologiques dans un mot circulant d'après [BBF04a].

1.6 Conclusion

Nous avons présenté dans ce chapitre un aperçu sur les grilles et les systèmes pair-à-pair en nous intéressant à leur architecture. En effet, une fois l'architecture fixée, les différents composants de l'application s'organisent autour de cette architecture. Le degré de centralisation est aussi un élément important à prendre en compte en particulier au niveau de la tolérance aux pannes et à la capacité de passage à l'échelle. Les réseaux pair-à-pair décentralisés comme *Gnutella* peuvent regrouper des millions d'utilisateurs.

Nous avons introduit la représentation des systèmes distribués sous la forme de graphes en énonçant des définitions et en présentant les topologies les plus couramment utilisées. Cependant, un système peut être modélisé sous plusieurs formes suivant la partie du système concernée (architecture physique ou composants de l'application) et en fonction du niveau d'abstraction (réseau physique des systèmes pair-à-pair ou réseau de recouvrement). Il est donc important de trouver un modèle adéquat pour concevoir de nouvelles applications.

Nous avons présenté deux outils permettant de construire des applications distribuées complètement décentralisées et tolérantes aux pannes. Nous avons introduit les différentes grandeurs caractéristiques associées aux marches aléatoires qui sont utilisées pour calculer la complexité d'algorithmes basés sur la circulation aléatoire d'un jeton. Nous avons aussi décrit un algorithme général permettant d'utiliser une marche aléatoire dans un système distribué qui prend en compte le temps de communication du jeton.

Nous avons vu que le mot circulant utilisé conjointement à la marche aléatoire permet de construire des structures couvrantes dynamiques et aléatoires en collectant les identités rencontrées lors de la circulation du jeton. Nous avons rappelé dans ce chapitre les différentes méthodes rencontrées dans la littérature permettant de gérer son contenu : la réduction des données redondantes afin de limiter sa taille ainsi que l'adaptation aux changements topologiques.

Chapitre 2

Conception d'applications de grille ou pair-à-pair : modèle théorique et simulateur

Résumé : *La conception d'une application distribuée doit suivre une succession d'étapes prédéterminées et passe notamment par le choix d'un modèle théorique. Nous présentons dans ce chapitre, un modèle original destiné aux applications de grille. Il est constitué de 5 couches superposées qui mettent en évidence les différents mécanismes sous-jacents à la grille ainsi que les interactions entre les composants de l'application. Conjointement à ce modèle, nous présentons la bibliothèque de simulation à événements discrets appelée* Dasor. *Elle permet de construire des simulateurs dont le modèle d'exécution est basé sur le modèle théorique. L'écriture d'un simulateur est réalisée indépendamment du réseau et des modèles de simulations qui ne sont appliqués qu'au moment de l'exécution. Ainsi, le degré d'abstraction de la simulation est ajustable sans modifier le code du simulateur. Le couple formé du modèle et de la bibliothèque est un outil complet pour la conception d'applications de grille ou pair-à-pair. Ces travaux ont fait l'objet d'une publication [RBF06]. Nous présentons à la fin de ce chapitre, quelques résultats de simulations sur les marches aléatoires et le mot circulant.*

2.1 Introduction

La conception d'applications distribuées doit suivre un schéma très strict afin de s'assurer de leur bon fonctionnement une fois déployées dans l'environnement de destination (réseau éthernet, réseau sans fil). La figure 2.1 présente ce schéma de manière générale : nous distinguons 5 étapes importantes.

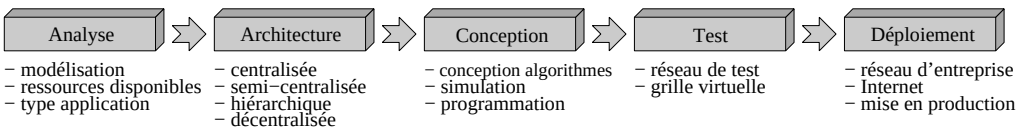


Fig. 2.1 – Schéma général de conception d'une application distribuée.

Analyse. La première étape consiste à analyser les besoins de l'application. Il s'agit de caractériser son fonctionnement général en déterminant, par exemple, ses besoins en capacité de stockage ou en communications. Les auteurs de [FK99] parlent de classes d'applications (voir

section 1.1). Ces besoins doivent être mis en parallèle avec les ressources disponibles. Le choix d'un modèle théorique simplifie cette étape. Il met en évidence toutes les contraintes liées à l'environnement et guide les étapes suivantes.

Architecture. À partir de l'étape d'analyse, le choix de l'architecture de l'application est réalisé. L'architecture détermine notamment la capacité de l'application à passer à l'échelle ou à supporter un nombre plus ou moins important de connexions et déconnexions dans le système. Certain types d'architectures impliquent l'existence de certains matériels sur lesquels des composants particuliers de l'application devront être installés. Dans le cas de *SETI@home*, le serveur central doit être adapté pour supporter un nombre important de connexions et doit avoir une grande fiabilité.

Conception. La conception de l'application s'appuie sur les deux étapes précédentes (analyse et architecture). Généralement, elle est modulaire pour simplifier la programmation. Le comportement de chaque composant peut être observé indépendamment à l'aide d'un simulateur. La simulation est une étape importante car elle peut mettre en évidence certains problèmes en fonction de stress particuliers (taux de pannes, nombre de nœuds, protocoles de communication). La programmation de l'application clôture l'étape de conception. Il est à noter que certains outils de simulation comme *SimGrid* [LMC03] proposent d'exporter directement les algorithmes simulés sous forme d'une application finale.

Test et déploiement. L'étape suivante consiste à tester l'application afin de s'assurer qu'elle réagit bien aux conditions initiales. Elle permet de détecter les erreurs de programmation éventuelles ainsi que les incompatibilités matérielles ou logicielles qui n'ont pas été prises en compte dans les étapes précédentes. Pour tester l'application, il est possible de la placer dans un environnement expérimental comme *Grid'5000*[1]. Dans ce cas, pour tester la tolérance aux pannes, des mécanismes particuliers d'injection de pannes sont nécessaires comme *FAIL-FCI* [HTV07]. Une autre solution consiste à utiliser des outils comme *MicroGrid* [LXC04] qui simule une grille virtuelle sur laquelle les applications s'exécutent de manière transparente. Enfin, une fois que les tests sont concluants, l'application peut être déployée dans son environnement de destination.

Dans nos travaux, nous nous sommes intéressés à la conception des applications de grille. Dans le chapitre précédent, nous avons vu que la modélisation d'un système distribué peut avoir plusieurs objectifs. Il existe ainsi plusieurs modèles théoriques dans la littérature que nous pouvons regrouper en deux types principaux : ceux axés sur l'architecture de l'application et ceux qui se focalisent sur le matériel et sur les protocoles.

Dans [LPP04], les auteurs proposent de modéliser la topologie du réseau d'une grille. La topologie est représentée sous la forme d'un graphe où chaque nœud représente aussi bien un site (appelé un hôte), que des composants réseaux comme des commutateurs, des routeurs ou un type de connexion particulier (gigabit, ethernet ou Myrinet). Ce type de modèle permet de mettre en avant les problèmes liés à l'architecture physique et ainsi étudier les phénomènes de congestion lors des communications entre les nœuds.

Les auteurs de [BBL02] proposent un modèle basé sur la notion de fabrique. Les composants et l'architecture de la grille sont organisés en couches superposées. La figure 2.2 (a) en propose une vue simplifiée. La couche la plus basse est proche du réseau et concerne les res-

[1]Site officiel de Grid'5000 https ://www.grid5000.fr

sources matérielles proprement dites, accessibles depuis des gestionnaires de ressources locaux. La seconde couche est axée sur la sécurité et à l'accès aux ressources à l'aide de mécanismes d'authentification. La troisième couche représente l'intergiciel qui sert d'interface pour l'accès aux ressources. Il peut être sous la forme d'un langage de programmation comme JXTA [Li03] ou sous la forme de bibliothèques comme *Globus*. La dernière couche représente les applications exploitant l'intergiciel.

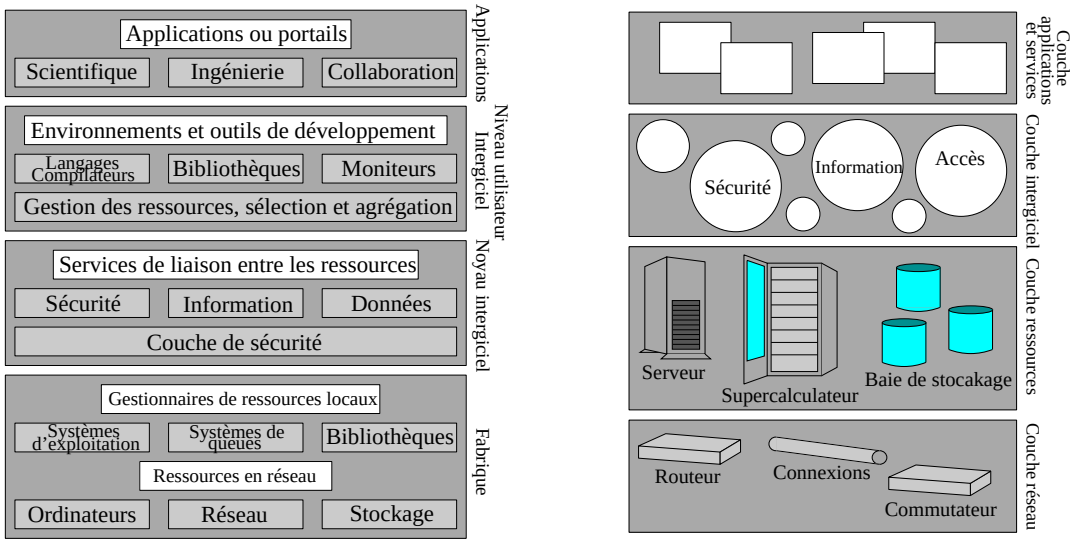


FIG. 2.2 – Modèle proposé dans [BBL02] basé sur la notion de fabrique (a) et modèle basé sur le matériel constituant la grille (b).

Un autre modèle couramment rencontré sur lequel s'appuie notamment *Globus*, est basé sur le matériel constituant la grille illustré sur la figure 2.2 (b)[2]. Il est lui-aussi constitué de 4 couches, la plus haute étant axée sur le point de vue utilisateur et la plus basse sur le matériel réseau. Au-dessus du réseau, nous trouvons les ressources qui peuvent être exploitées par la grille comme la puissance de calcul, la capacité de stockage ou les données distribuées. La troisième couche représente l'intergiciel qui fournit aux applications différents services comme la sécurité, l'authentification des utilisateurs, l'accès aux données ou aux autres ressources. La dernière couche concerne les applications qui exploitent les services de l'intergiciel.

Le premier modèle [LPP04] permet de mettre en évidence les problèmes matériels mais se détache des contraintes de l'application. Les deux suivants, au contraire, se focalisent sur la grille en omettant les mécanismes sous-jacents. Dans [RBF06], nous avons proposé un modèle théorique original permettant de modéliser une application de grille sous la forme de graphes. Il est constitué de 5 couches, les trois couches inférieures représentant les mécanismes sous-jacents à la grille. L'étude de ces mécanismes permet de s'assurer du bon fonctionnement de l'application en mettant en place des moyens de correction nécessaires. Les deux couches supérieures représentent les différents composants de l'application qui représente l'intergiciel proprement dit. En nous basant sur ce modèle, nous avons développé une bibliothèque de simulation qui

[2]Inspiré du site *GridCafé* à l'adresse http ://gridcafe-f.web.cern.ch/gridcafe-f/

permet ainsi de simuler des algorithmes en modifiant indépendamment les paramètres de chaque couche.

Dans la section 2.2, nous détaillons les couches de notre modèle théorique et nous présentons une étude des différents impacts des pannes sur les couches de l'intergiciel. Nous décrivons dans la section 2.3, notre bibliothèque de simulation à événements discrets appelée *Dasor*. Enfin, dans la section 2.4, nous proposons des simulations sur les marches aléatoires et le mot circulant.

2.2 Modèle théorique

Afin de présenter les différents mécanismes interagissant dans une grille ou un système pair-à-pair, nous avons proposé dans [RBF06] un modèle original constitué de plusieurs couches représentant chacune une fonctionnalité particulière (par exemple le routage, les communications...). Le schéma de la figure 2.3 en propose un aperçu général.

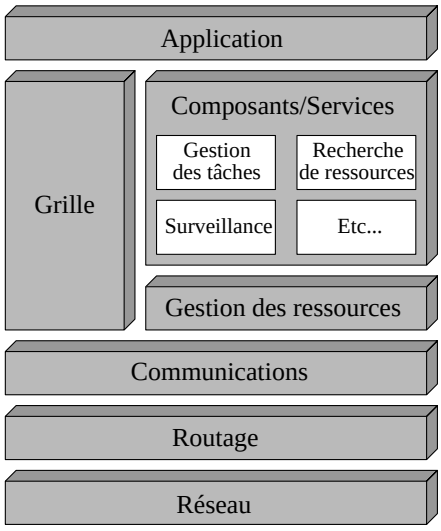


FIG. 2.3 – Présentation des différentes couches du modèle.

2.2.1 Présentation des couches

Chaque couche est caractérisée de la manière suivante.

Réseau (couche 1) : le réseau est représenté sous la forme d'un graphe orienté $G_1 = (V, E)$ dont les arêtes représentent les liens physiques (les bus, les câbles ou les connexions sans fil) entre les différentes entités du réseau et les noeuds représentent les éléments du réseau. Ces éléments peuvent être des ordinateurs, des machines parallèles ou bien des éléments passifs du réseau comme des routeurs ou des concentrateurs. Au sein de cette couche, un protocole calcule et met à jour l'ensemble $Vois_i^1$ sur chaque nœud i contenant les identités (les adresses IP, par exemple) des voisins de i. Un câble réseau est naturellement non-orienté. Mais les différences de portée entre les nœuds dans les réseaux sans fil impliquent une orientation de certain liens de G_1. Ainsi, si un nœud i est dans la zone de couverture d'un nœud j mais que sa propre portée

est insuffisante, cela se traduit par un lien orienté (j, i)[3].

Routage (couche 2) : au-dessus des liens physiques, nous ajoutons les différents chemins possibles entre les nœuds, calculés à l'aide d'un protocole de routage. Ce dernier calcule des tables de routage locales qui correspondent aux chemins entre les nœuds. Elles se traduisent par la maintenance d'ensembles $Vois_i^2$ qui contiennent les identités des noeuds accessibles à partir d'un noeud i. Le réseau est représenté sous la forme d'un graphe $G_2 = (V, E')$ où E' est l'ensemble des routes possibles entre les différentes entités du réseau. Induit par la restriction des droits d'accès ou de l'accessibilité limitée, chaque lien peut être orienté. Par exemple, un nœud qui est situé derrière un pare-feu, peut être inaccessible pour une partie des nœuds du réseau. De même, le protocole de translation d'adresses NAT permet un accès à Internet à un ensemble de nœuds d'un réseau local. Ceux-ci ne possèdent pas d'adresse IP propre et ne sont donc pas accessibles directement depuis l'extérieur (*i.e.* si aucune connexion n'est établie).

Échange de messages (couche 3) : un protocole permet l'échange de messages ou de données entre les entités du réseau via les chemins de communication construits dans la couche précédente. Suivant ce protocole, différentes options sont disponibles. Avec un mécanisme d'accusés de réception, un nœud est assuré que le message a bien été transmis à son destinataire. Un système de *hachage* sert à vérifier l'intégrité des messages. Dans le cas des réseaux IP, des mécanismes assurent la ré-émission des messages non-acquittés en cas de perte ou de corruption des données. Les ensembles de voisins $Vois_i^3$ sont identiques à ceux de la couche précédente et nous avons $G_3 = G_2(V, E')$.

Gestion des ressources (couche 4) : nous distinguons au sein de cette couche deux types de noeuds en fonction de leur activité dans la grille. Les noeuds *actifs* sont les noeuds connectés à la grille, c'est-à-dire qui partagent ou utilisent des ressources. Les noeuds *inactifs* sont les éléments passifs du réseau comme les routeurs ou les commutateurs ainsi que les ordinateurs qui ne sont pas connectés dans la grille. La grille est donc représentée sous la forme d'un graphe $G_4 = (V', E'')$ où V' est l'ensemble des noeuds actifs et l'ensemble E'' comprend l'ensemble des routes possibles entre chaque noeud actif. Un protocole doit maintenir la topologie de la grille, c'est-à-dire être capable de détecter la connexion ou la déconnexion des nœuds. Par exemple, dans le cas du protocole *Gnutella*, des connexions TCP sont établies et l'ensemble de voisins $Vois_i^4$ est un sous-ensemble des voisins atteignables. Dans ce cas, les chemins orientés de la couche précédente deviennent non-orientés.

Services et composants de la grille (couche 5) : cette couche comprend tous les services et les composants d'une grille : la recherche de ressources, l'ordonnancement des tâches, le monitoring, le transfert de données, etc... Par exemple, les grilles de calcul sont constituées de composants comme la gestion des tâches dont les principales fonctions sont la diffusion des paramètres des tâches dans le réseau et leur attribution à des ressources disponibles et éventuellement, des composants comme le transfert de fichiers pour retourner les résultats. Au sein de cette couche, la représentation de la grille dépend de la couche inférieure. Si des connexions sont établies entre chaque nœud, les liens sont non-orientés. Dans ce cas, les mes-

[3] À condition que le protocole de bas niveau le permette.

sages échangés par les nœuds de la grille passent par ces canaux de communication.

Applications (couche 5 bis ou couche 6) : au-dessus de la grille, nous trouvons les applications qui exploitent les différents composants mis à disposition au sein de la grille. L'accès aux ressources doit être transparent pour l'utilisateur.

À partir de ce modèle, une grille de calcul peut être représentée sous la forme de graphes en fonction des différentes couches du modèle, comme le montre le tableau de la figure 2.4.

Couches	Graphe	Nœuds	Liens	Données locales
1	$G_1(V, E)$	Éléments réseau	Lien physique	$Vois_i^1$, voisins physiques
2	$G_2(V, E')$	Éléments réseau	Lien de communication	$Vois_i^2$, voisins atteignables
3	$G_3(V, E')$	Éléments réseau	Lien de communication	$Vois_i^3$, voisins atteignables
4	$G_4(V', E'')$	Nœuds actifs	Lien ou connexion	$Vois_i^4$, voisins dans la grille
5	$G_5(V', E'')$	Nœuds actifs	Lien ou connexion	$Vois_i^5$, voisins dans la grille

FIG. 2.4 – Résumé des données au sein de chaque couche du modèle.

Exemple 2.1 *La figure 2.5 montre un exemple de grille constituée d'un réseau local qui comprend deux ordinateurs* (1) *et* (2) *et un serveur* (3)*. Ces trois machines sont reliées via un routeur/pare-feu* (4)*. Une machine parallèle* (6) *est connectée à un routeur* (5) *lui-même relié à* (4)*. Une grille peut aussi être constituée de matériels sans fil. Un* PDA (8) *et un ordinateur portable* (9) *sont reliés à une borne* WIFI (7)*, elle-même reliée au routeur/pare-feu* (5)*. La portée de* (9) *est supérieure à* (8)*, nous notons ainsi un lien orienté. Dans cette exemple, nous supposons que pour des raisons de sécurité, les ordinateurs* (1) *et* (2) *ne sont pas accessibles directement de l'extérieur. Par contre, ils ont un accès direct à la machine parallèle. Cette restriction entraîne donc des liens de communication orientés au sein de la couche 4. En fonction du protocole de la couche 4, des connexions peuvent être établies entre les nœuds et le voisinage réduit à un sous-ensemble de nœuds. Dans ce cas, les liens ne sont pas orientés et le nombre de liens est réduit. Sur la figure 2.5, nous montrons les deux cas de figure, orienté ou non.*

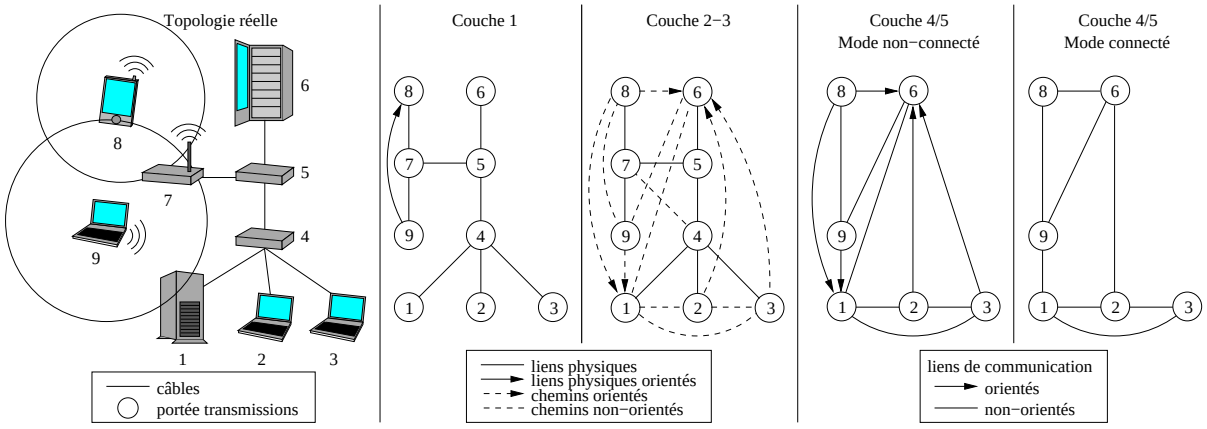


FIG. 2.5 – Exemple de représentation d'une grille de calcul au travers des couches du modèle théorique.

2.2.2 Impacts des fautes

À partir du modèle, il est maintenant possible de distinguer pour chaque couche les différentes fautes pouvant intervenir dans le système et d'analyser les différents impacts possibles sur les couches supérieures. Le schéma 2.6 résume une partie des fautes possibles avec leurs impacts respectifs.

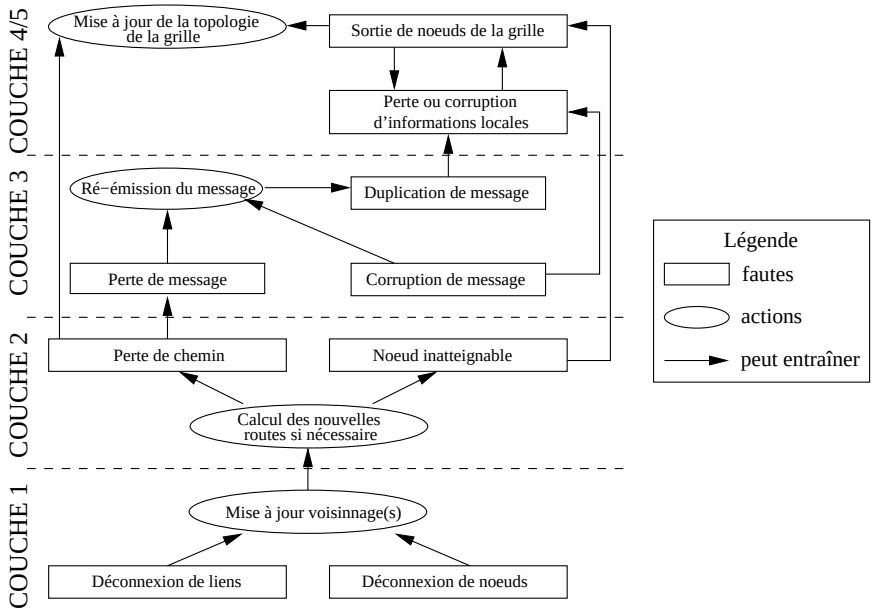


FIG. 2.6 – Impacts des fautes au travers des couches du modèle théorique.

Afin de comprendre les différentes fautes qui interviennent dans les couches de la grille (couches 4 et 5), il est nécessaire de s'intéresser d'abord aux couches inférieures. En comprenant l'impact des fautes qui sont générées dans ces couches, il est plus simple de mettre en place des mécanismes appropriés. Ces mécanismes dépendent de l'application développée, de l'environnement de destination mais aussi des fautes acceptées (dans la section 1.2.2, nous avons vu que les applications de type *Napster* se reposent sur un agent central qui ne doit pas tomber en panne).

La **couche 1** est la couche la plus proche du matériel. Aussi, les fautes possibles à ce niveau sont des pannes physiques : la déconnexion d'un lien (un câble débranché, coupé ou défectueux) et la panne d'un noeud (coupure de courant ou ordinateur éteint). Ces deux pannes entraînent des modifications sur le voisinage que le protocole répercute dans les différents ensembles locaux $Vois_i^1$. En fonction du temps de rafraîchissement de ces ensembles, il peut y avoir des conséquences au niveau supérieur. Dans le cas des réseaux sans fil, la mobilité des nœuds implique aussi une mise-à-jour de ces ensembles.

Dans la **couche 2**, la mise-à-jour des voisinages de chaque noeud du réseau entraîne différentes mises-à-jour. Le protocole de routage doit recalculer les chemins invalides. Cependant, les pannes de la couche 1 peuvent avoir des conséquences dans cette couche : l'impossibilité de recalculer un chemin entre deux noeuds et l'impossibilité de rejoindre un noeud donné. De même, un chemin peut être invalide durant une certaine période de temps, induit par le temps

nécessaire au protocole pour le recalculer.

En fonction du protocole d'échange de messages de la **couche 3**, des fautes peuvent survenir. Cela dépend des mécanismes de vérification mis en place. Nous comptabilisons trois principaux types de fautes : la perte de messages, la duplication de messages et la corruption de messages. Ces fautes interviennent à cause de perturbations ou suite à des fautes sous-jacentes. Un changement de route peut entraîner une perte de messages ou des duplications. Dans un environnement de type TCP/IP, l'envoi de données est suivi d'un accusé de réception qui assure que les données sont envoyées et reçues. Un mécanisme de hachage permet aussi de vérifier l'intégrité des données. Si un message invalide est reçu, il est re-émis. Ce qui n'est pas forcément le cas avec tous les protocoles de communication.

Dans les **couches 4 et 5**, la perte de chemin ou un site non joignable entraînent des changements de topologie au sein de la grille (déconnexion d'un nœud de la grille ou déconnexion physique d'un nœud). Il est à noter que ces changements peuvent induire aussi la perte d'informations locales (si une tâche est en cours d'exécution). Si nous utilisons la circulation d'un jeton, il faut introduire des mécanismes de récupération de ce jeton. De même, pour les grilles de calcul, il faut prendre en compte les tâches en cours sur un site qui se déconnecte : assignation des tâches sur un autre nœud, sauvegardes régulières de l'avancement du calcul ("*points de contrôle*"), etc...

Toutes les fautes ne sont pas obligatoirement gérées dans toutes les applications distribuées. Les mécanismes à mettre en place dépendent de l'environnement de destination et des protocoles mis en place, ainsi que de l'application.

2.3 Dasor, une bibliothèque de simulation

Lorsqu'une application est diffusée à grande échelle, son comportement peut être difficile à prévoir. Même si l'application et ses différents composants ont été développés en se basant sur un modèle théorique, de nouveaux mécanismes apparaissent dus aux effets du matériel physique, des protocoles déployés sur le réseau ou bien de l'interaction avec d'autres applications. Aussi, l'efficacité de l'application peut être imprévisible et très variable. L'exécution sur des milliers de nœuds est bien souvent difficile voire impossible à réaliser en phase de test. En effet, le matériel disponible est bien souvent insuffisant pour faire apparaître les problèmes de mise à l'échelle et l'utilisation de réseaux dédiés ne reflètent généralement pas le cas réel, comme une exécution sur Internet. C'est pourquoi la simulation est une étape essentielle. Même si elle ne peut être utilisée pour valider les algorithmes simulés comme une exécution en conditions réelles, elle permet cependant de contrôler leurs comportements lorsqu'ils sont soumis à des contraintes spéciales.

La simulation apporte cependant de nouvelles contraintes. Il est souvent difficile de choisir des modèles de simulation adéquats, choix qui doit être guidé par le temps de calcul de la simulation et le degré de finesse de la simulation désiré. En particulier, une application pair-à-pair est distribuée au travers Internet. Doit-on simuler jusqu'au moindre échange de paquets entre les nœuds ou bien choisir un degré d'abstraction plus élevé ? Dans le premier cas, le temps

de simulation devient exorbitant pour des réseaux de plusieurs dizaines de milliers de nœuds mais parfois essentiel pour analyser l'évolution de performances (phénomènes de congestion ou de latence). Dans le second cas, une bonne modélisation est nécessaire pour prendre en compte les effets de bord. Pour une même application, ces deux orientations sont parfois nécessaires pour tester ses composants.

Certaines bibliothèques de simulation actuelles proposent plusieurs niveaux d'abstraction. Cependant, dans la plupart des cas, le choix doit être pris avant d'écrire le simulateur car le changement du niveau d'abstraction s'accompagne bien souvent de la reprogrammation partielle, voire complète, du simulateur. De plus, les niveaux d'abstraction proposés se résument bien souvent à deux niveaux : le niveau le plus faible qui consiste à simuler tous les échanges de messages des protocoles réseaux et le niveau le plus élevé qui consiste à les ignorer et considérer, par exemple, les flux de données entre les nœuds. Or, il est intéressant de proposer des choix intermédiaires entre ces deux niveaux et de pouvoir les modifier à volonté suivant le composant de l'application à tester ou en fonction des conditions de test.

C'est pourquoi nous avons développé *Dasor* [Rab]. C'est une bibliothèque de simulation à événements discrets basée sur notre modèle théorique. Les simulateurs générés à partir de cette bibliothèque sont écrits indépendamment du réseau sur lequel les simulations sont exécutées ainsi que des modèles de simulation (communication, routage, caractéristiques matérielles, réseau...). Le réseau et ces modèles ne sont renseignés qu'au moment de l'exécution du simulateur via un fichier externe écrit dans un langage simplifié, appelé *fichier de description*. Nous apportons ainsi un couple d'outils puissant (formé du modèle théorique et de *Dasor*) pour la conception d'applications distribuées.

Dans les sections suivantes, nous présentons l'architecture et les différents composants de notre bibliothèque. Nous détaillons notamment le contenu des fichiers de description, les différents outils présents dans la bibliothèque et la structure principale d'un simulateur écrit avec *Dasor*.

2.3.1 Aperçu des différents simulateurs d'applications distribuées

Avant de présenter notre bibliothèque, nous proposons un aperçu sur différents outils de simulation existants. La conception de ces outils est bien souvent guidée par le type d'applications à simuler. Nous nous sommes intéressés plus particulièrement à ceux qui permettent de simuler des applications de grille. Nous distinguons trois catégories principales : les outils de simulation qui se placent au niveau des protocoles réseau (échanges de paquets, couche de protocoles), les outils de simulation qui proposent de simuler les composants spécifiques aux grilles (ordonnancement, transfert de données) et les simulateurs de réseaux pair-à-pair ou d'applications pair-à-pair (protocoles pair-à-pair, échanges de fichiers).

Les outils de simulation réseau

Les applications distribuées sont interconnectées à l'aide de réseaux physiques. Au sein de ces réseaux, des protocoles sont déployés pour échanger de l'information. En général, ils sont empilés, la pile la plus connue étant celle des protocoles Internet TCP/IP. Au niveau de l'application, un seul message est envoyé mais physiquement, il est encapsulé, fragmenté en plusieurs

messages et envoyé au travers le réseau avant d'être reconstitué une fois arrivé à destination. Pour des applications de grille, la connaissance exacte du comportement de ces protocoles en rapport avec l'application est cruciale afin d'apporter une efficacité optimale dans toutes les communications. L'étude des différents paquets et de leur route met en évidence les problèmes de congestion ou de latence dans le réseau.

OMNeT++ [Var01] est une bibliothèque pour la simulation à événements discrets écrit en *C++*. Elle a été proposée dans un premier temps pour simuler les réseaux de communication. Elle est basée sur la notion de *modules* qui communiquent par passage de messages. Un nœud est représenté par un module ou une pile de modules, ce qui permet de simuler une pile de protocoles. Ainsi, lorsqu'un message est envoyé d'un nœud à un autre, soit la communication est établie directement entre les modules associés ou bien le message est transféré au travers des couches de la pile de manière transparente. L'ensemble de la structure du réseau et des interconnexions est décrit à l'aide d'un langage propre appelé "*NED*". À partir de ce fichier et d'un outil de pré-compilation, des classes sont générées, formant la structure de base du simulateur. L'utilisateur renseigne ensuite chaque méthode qui sont exécutées lors de la réception d'un message sur un nœud. Une simulation correspond aux échanges de messages dans le système, chacun provoque l'exécution d'un code.

Ns-2 [NS] propose aussi la simulation de protocoles réseaux. Il permet de créer et de tester de nouveaux protocoles ou de comparer des protocoles similaires. Il utilise des scripts *OTcl* (variante objet de *Tcl*) pour décrire des événements associés à un temps donné ou de construire un réseau en décrivant le débit associé aux liens. De plus, il est possible de sélectionner le protocole de routage utilisé, les protocoles pour le transport (*TCP*, *UDP*) et pour les applications (*FTP*, *Telnet*).

Ces différents outils sont utiles pour simuler avec précision les protocoles réseaux, mais il est possible de simuler aussi le comportement d'applications de plus haut niveau. Cependant, les besoins particuliers des applications de grille ou pair-à-pair ne sont pas pris en compte comme la gestion des ressources, le calcul de tâches ou le transfert de fichiers. D'autres simulateurs ont donc été proposés afin de s'intéresser plus particulièrement à ces composants en les intégrant directement aux simulateurs.

Les simulateurs de grilles

En général, une application de grille fait appel à plusieurs composants comme la gestion des ressources ou la gestion des tâches. Chacun doit être analysé afin de vérifier son comportement (efficacité pour les méthodes d'ordonnancement, coûts de communication associés) avant de les intégrer dans l'application finale. Les simulateurs de grilles sont très variés et sont généralement destinés à simuler un seul type de composant.

Pour les grilles de calcul, *SimGrid* [LMC03] est une boîte à outils pour créer des simulateurs à événements discrets d'applications d'ordonnancement. Il est consisté de 5 éléments :

- *Agents* : chacun est chargé d'une partie de l'ordonnancement ;
- *Endroits* : il s'agit d'un endroit dans la topologie où un agent est exécuté et qui possède

des ressources de calcul ou des données ;
- *Tâches* : dans *SimGrid*, une tâche peut être aussi bien un calcul qu'un transfert de données ;
- *Chemins* : ils correspondent aux chemins entre les sites et sont des ensembles de ressources de communication ;
- *Canaux* : lorsque deux agents communiquent entre eux, une communication est établie via un canal.

SimGrid étant destiné à simuler des applications d'ordonnancement, il fournit des fonctions avancées pour la gestion des tâches avec la prise en compte des dépendances.

OptorSim [BCC+02] est un paquetage *Java* pour simuler des grilles de données. Il est basé sur l'architecture de *DataGrid*[4] qui est une grille européenne construite dans l'objectif d'étudier le calcul intensif et l'interaction avec des bases de données à grande échelle (de 100 à 1000 TBytes). Lors de la simulation, chaque site fournit des ressources de calcul et de stockage. Lorsqu'une tâche est exécutée, elle utilise des ressources locales. Un gestionnaire de réplication est chargé du transfert des données d'un site à un autre, en fonction des besoins. Pour cela, il fait appel à l'*optimiseur de réplication* appelé *Optor*. Il est ainsi possible de comparer différentes stratégies de réplication.

Contrairement aux précédents simulateurs, *MicroGrid* [LXC04] émule une grille virtuelle en modélisant un réseau, des ressources ainsi que des services d'information. Des applications *réelles* peuvent être exécutées au-dessus de cette grille virtuelle de manière transparente.

D'autres simulateurs, comme *GridG* [LD03] se concentrent sur la génération de topologies de grilles basées sur les caractéristiques de réseaux de type Internet. Pour chaque nœud et chaque lien, le matériel et les logiciels disponibles sont ainsi détaillés (mémoire, capacité disque, bande passante, latence). Ces topologies peuvent ensuite être utilisées dans des programmes ou d'autres simulateurs.

Les simulateurs de réseaux et d'applications pair-à-pair

Tout comme les grilles, les réseaux pair-à-pair ont des spécificités propres. Tout d'abord, contrairement aux grilles "*classiques*", les réseaux pair-à-pair sont généralement déployés à très grande échelle au travers Internet. Il n'y a pas de réseau dédié et les ressources sont dynamiques. De plus, le réseau d'interconnexion est souvent très disparate entre les nœuds. Les moyens de connexion à Internet étant très variés (RTC, ADSL, fibre ou câble) et ont des débits très différents et fluctuants. Aussi, pour simuler des applications comme *Gnutella* qui regroupe une très grande quantité de nœuds, il est difficile voire impossible d'utiliser des simulateurs orientés réseaux ou de grille. Nous présentons ici quelques outils de simulation dédiés aux problématiques des systèmes pair-à-pair.

3LS [TD03] est orienté sur trois niveaux : le niveau réseau, le niveau protocole et le niveau utilisateur. Le premier niveau consiste à décrire les nœuds en terme de distance entre les nœuds.

[4] L'adresse officiel du projet est http ://eu-datagrid.web.cern.ch/eu-datagrid/

Le niveau protocole est le protocole pair-à-pair à utiliser et le niveau utilisateur permet d'injecter des données à partir d'un fichier. En séparant les trois niveaux, il est possible d'exécuter des simulations en choisissant parmi les modèles de la bibliothèque ou des modèles utilisateurs. Le comportement de l'application peut donc être observé sans modification du code du simulateur. Actuellement, il n'existe qu'une simulation du protocole *Gnutella* qui ne supporte qu'un nombre très faible de nœuds.

Les auteurs de *Narse* [GB02] proposent une approche différente. C'est un simulateur écrit en *Java* dont le but est de simuler des applications réparties à grande échelle. Il fournit une interface pour le protocole de transport pour l'envoi des données, permettant de caractériser les applications. Il repose sur la supposition que les paquets correspondants à une transmission peuvent être modélisés sous forme d'un flux. De plus, les transferts ne sont limités que par la bande passante la plus faible entre celle de l'émetteur et celle du destinataire, la bande passante des routeurs intermédiaires n'étant pas prise en compte. L'approche proposée dans *Narse* permet ainsi de simuler des réseaux à très grande échelle pour un temps de calcul raisonnable. L'inconvénient majeur est qu'il ne peut être utilisé sur des topologies contenant des goulots d'étranglement.

NeuroGrid [NEU, Jos03] a été proposé afin de comparer les systèmes pair-à-pair d'échange de fichiers comme *Gnutella* ou *FreeNet*. Il utilise un fichier de configuration afin de définir le protocole à simuler et les propriétés du réseau comme le nombre de nœuds ou le nombre de requêtes à exécuter. Il fournit alors des statistiques en rapport avec le taux de réussite des requêtes. Actuellement, des simulations sont disponibles pour les protocoles de *Gnutella*, *FreeNet* et *Pastry*.

Le simulateur *PeerSim* [PEE] est codé en *Java* sous licence *GPL* et fait partie du projet *Bison* de l'Université de Bologne[5]. Il propose quant à lui deux modes de fonctionnement : par cycles ou par événements. Dans le mode par cycles, il autorise un degré d'abstraction plus important. Les protocoles s'exécutent à chaque cycle ce qui implique une absence de concurrence et la couche de transport est ignorée. Le mode par événements est basé sur l'échange de messages et propose de simuler la couche de transport. Le principe général du simulateur repose sur la notion de composants qui peuvent être ajoutés à la volée à l'aide d'un fichier texte. Il se focalise sur la simulation des protocoles proprement dit et il est en mesure de simuler des réseaux de plusieurs centaines de milliers de nœuds.

2.3.2 Présentation de *Dasor*

La conception de *Dasor* a été motivée par plusieurs points. D'une part, nous voulions proposer une bibliothèque qui laisse le choix à l'utilisateur du simulateur, du degré d'abstraction de la simulation. Ainsi, à partir d'un simulateur donné, il est possible de régler chaque modèle de simulation afin de jouer sur le degré de réalisme de la simulation et sur le temps de simulation. Ce réglage doit être réalisé sans modifier le code du simulateur. D'autre part, nous voulions proposer des simulateurs dont le modèle d'exécution est calqué sur notre modèle théorique.

[5]L'adresse Internet du site de Bison est http ://www.cs.unibo.it/bison/

Ainsi, une application conçue à partir du modèle peut être précisément simulée avec *Dasor*. L'utilisateur du simulateur peut ainsi jouer sur chaque composant de chaque couche du modèle.

Les auteurs de [LK99] décrivent la simulation à événements discrets comme étant un moyen de modéliser un système qui évolue en temps réel en représentant ses états au cours du temps. Il change spontanément d'état à un temps donné qui est appelé *événement*. *Dasor* permet de construire des simulateurs à événements discrets d'applications distribuées complètes ou de composants d'applications de grille ou pair-à-pair comme l'ordonnancement de tâches ou le transfert de fichiers. En ignorant tous les composants relatifs aux grilles, il est possible aussi de simuler des algorithmes distribués.

Le modèle d'exécution des simulateurs est basé sur le modèle théorique que nous avons proposé dans la section 2.2. La figure 2.7 présente ainsi les différents composants associés aux couches du modèle. La bibliothèque fournit différents modèles de simulation qui correspondent aux trois premières couches du modèle. Les deux couches les plus hautes sont les algorithmes à simuler et doivent être programmées par le créateur du simulateur.

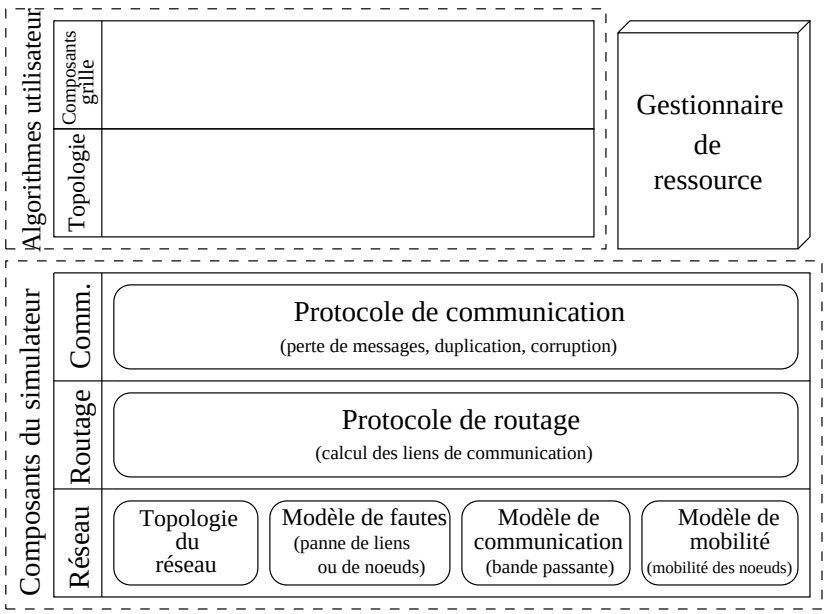


FIG. 2.7 – Modèle d'exécution de simulateurs écrits à l'aide de *Dasor*, basé sur le modèle théorique.

L'écriture d'un simulateur est réalisée indépendamment du réseau sur lequel il est exécuté, ainsi que des différents modèles appliqués. Le réseau et le modèle ne sont renseignés qu'à l'exécution à l'aide d'un fichier de description externe passé en paramètre. Ainsi, pour un simulateur donné, il est possible de l'exécuter dans des environnements différents sans modification de son code.

La figure 2.8 schématise le fonctionnement de la bibliothèque. Elle permet de construire des simulateurs qui acceptent en paramètre trois types de fichier : le fichier de description (*.net*) qui décrit le réseau et les modèles de simulation, le fichier d'initialisation (*.ini*) contenant les différents paramètres de simulation (le nombre de simulations, le temps maximal de simulation) et les fichiers de données éventuels (durées des tâches sous la forme d'un fichier texte,

par exemple). L'exécution d'un simulateur peut produire deux types de fichiers. Le fichier de statistiques (*.sta*) correspond aux valeurs retournées par le simulateur. Elles peuvent être utilisées pour produire des courbes à l'aide de l'outil *GnuPlot*[6]. Le fichier de description (*.des*) est la trace d'exécution, c'est-à-dire la description de tous les événements produits lors des simulations. Dans les sections suivantes, nous en proposons différentes utilisations comme la construction de diagrammes d'exécution ou l'analyse des échanges de messages.

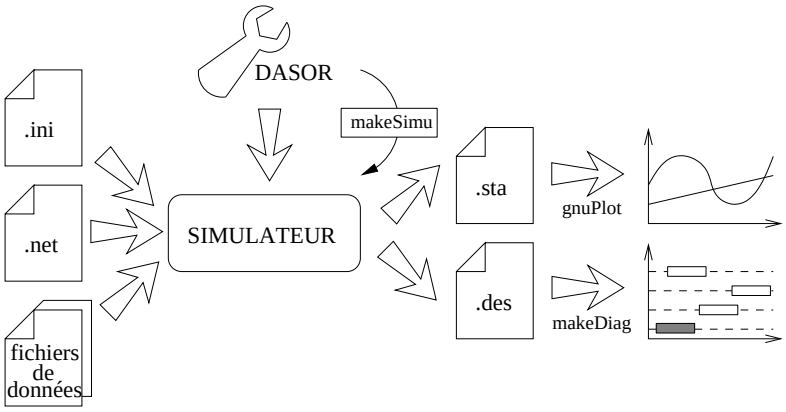


FIG. 2.8 – Interactions entre différents composants de la bibliothèque *Dasor*.

2.3.3 Le fichier de description

Lors de l'exécution d'un simulateur, il est nécessaire de passer en paramètre le nom du fichier de description. Il s'agit d'un fichier texte qui décrit les différents modèles appliqués lors de l'exécution. Sa syntaxe a été simplifiée, ce qui permet de décrire assez rapidement le réseau ainsi que les différents modèles choisis.

Dasor supporte les réseaux filaires et les réseaux sans fil (nous planifions de rendre possible la construction de réseaux hybrides). Suivant le type choisi, la topologie du réseau est décrite différemment : dans le cas des réseaux filaires, il faut décrire chaque lien entre les nœuds alors que pour les réseaux sans fil, c'est la position des nœuds qui doit être renseignée. Ici, nous nous intéressons plus particulièrement aux réseaux filaires.

Un réseau doit comporter un nombre déterminé de nœuds qui ne peut être modifié au cours d'une simulation, éventuellement, certains nœuds peuvent être inactifs. Il est renseigné à l'aide du mot-clef "`NODES= 10`" pour un réseau de 10 nœuds. Le fichier de description permet de spécifier différentes options générales à l'aide du mot-clef "`OPTIONS=`" suivi des mots-clefs suivants :

- `WIRED` ou `WIRELESS` : détermine si le réseau est filaire ou sans fil ;
- `DIRECTED` ou `UNDIRECTED` : détermine si les liens de communication sont orientés ou non ;
- `WEIGHTED` ou `UNWEIGHTED` : indique si les liens possèdent des poids (utile lors de la simulation de certains algorithmes distribués).

Chaque modèle appliqué lors de l'exécution correspond à un mot-clef dans le fichier de description. Ce mot-clef peut être employé de deux manières : suivi du symbole "`=`", ce qui correspond à un modèle et suivi du symbole "`:`", ce qui correspond à un événement du modèle.

[6]Le site officiel de GnuPlot est http ://www.gnuplot.info

L'utilisateur peut donc choisir des modèles existants dans la bibliothèque, en créer de nouveaux en détaillant chaque événement ou encore modifier un modèle en y ajoutant des événements.

Les liens sont décrits en utilisant le mot-clef "`LINK`". Toute une série de topologies de base est proposée comme l'anneau, la chaîne ou la grille mais aussi des topologies évoluées comme les réseaux petit-monde ou à degré moyen. Chaque topologie correspond à un mot-clef particulier et représente un modèle, un événement étant l'ajout ou la suppression d'un lien. Voici plusieurs exemples de fichiers de description qui décrivent un anneau orienté :

```
[GRAPH]
NODES= 5
OPTIONS= directed
LINK= directed ring
```

```
[GRAPH]
NODES= 5
OPTIONS= directed
POSITION= circle
LINK: (0->1)%1
LINK: 4->0
```

```
[GRAPH]
NODES= 5
AREA= 5, 5
OPTIONS= directed, unweighted, wired
POSITION:0= 2.5, 5
POSITION:1= 4.8, 3.2
POSITION:2= 3.9, 0.4
POSITION:3= 1.0, 0.4
POSITION:4= 0.1, 3.2
LINK:
[[1, 1, 0, 0, 0]
 [0, 1, 1, 0, 0]
 [0, 0, 1, 1, 0]
 [0, 0, 0, 1, 1]
 [1, 0, 0, 0, 1]]
```

2.3.4 Le choix du niveau d'abstraction

Comme le montre la figure 2.7, le modèle du simulateur est calqué sur notre modèle théorique. Chaque couche peut comprendre un ou plusieurs modèles de simulation, chacun étant paramétrable par l'utilisateur du simulateur via le fichier de description. La couche réseau, par exemple, comprend les modèles sur les propriétés physiques des nœuds, le réseau (bande passante) ainsi que sur les pannes ou la mobilité des nœuds.

Le choix du degré d'abstraction de la simulation n'est pris qu'au moment de l'exécution du simulateur, via le fichier de description. *Dasor* propose ainsi plusieurs dimensions d'abstraction. Tout d'abord, la plupart des modèles de simulation ne sont pas obligatoires. En omettant certains modèles, le temps d'exécution de la simulation est plus court et l'espace mémoire utilisé est plus faible. D'autre part, il est possible de choisir entre des modèles plus ou moins complexes, ce qui permet d'augmenter la précision des simulations. Par exemple, le modèle de communication peut prendre en compte le transfert de paquets dans le réseau ou gérer des flux de données.

Exemple 2.2 *La figure 2.9 montre l'effet du choix des composants sur deux exemples de fichiers de description. Dans le premier, une topologie en étoile est sélectionnée et un modèle de routage est appliqué. Ainsi, un message envoyé du nœud 4 vers le nœud 1 transite par le nœud 0. Dans le deuxième fichier, une topologie complète est sélectionnée et aucun routage n'est appliqué. Dans ce cas, le message ne transite plus par le nœud 0. Le niveau d'abstraction est plus élevé,*

ce qui réduit le nombre de messages échangés et diminue le temps d'exécution de la simulation. Mais les phénomènes de congestion qui pourraient apparaître sur le nœud 0 ne pourront pas être détectés.

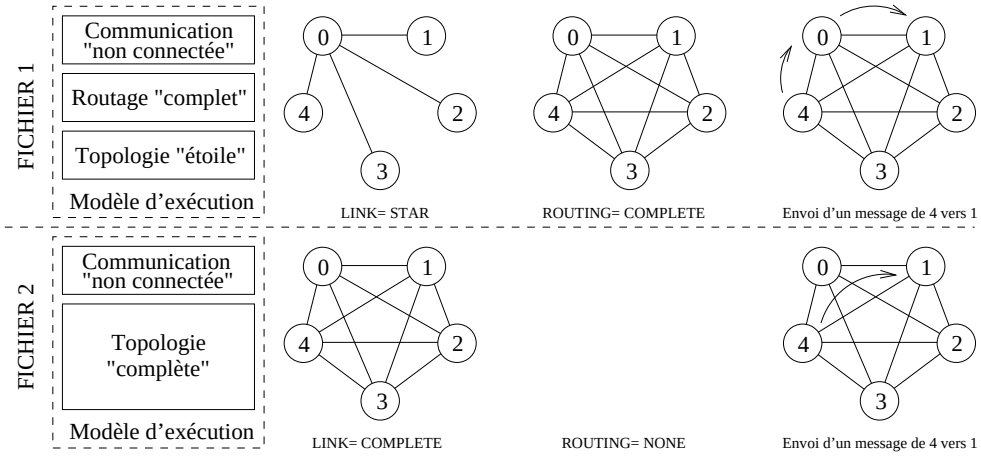


FIG. 2.9 – Effet de l'envoi d'un message en fonction du niveau d'abstraction

L'exemple précédent montre que le choix des modèles dépend du comportement à mettre en évidence et ce choix peut à tout moment être modifié indépendamment du simulateur. Par exemple, le protocole *Gnutella* a été largement étudié dans la littérature. Nous pouvons réaliser des séries de simulations en étudiant les phénomènes de congestion du réseau liés au grand nombre de messages échangés. Dans ce cas, nous appliquons des modèles de routage pour observer le trafic dans le réseau. Avec le même simulateur, nous pouvons étudier le taux de réussite des requêtes de fichiers. Dans ce cas, le protocole de routage n'est plus utile, ce qui permet d'accélérer la simulation.

2.3.5 Les composants de *Dasor*

Comme le montre la figure 2.7, un ou plusieurs composants peuvent être appliqués à chaque couche. Dans la première, nous trouvons tous les composants relatifs aux propriétés matérielles des nœuds. En particulier, nous nous focalisons sur trois types de modèles que nous exploitons dans la suite[7] : la topologie du réseau (mot-clef "`LINK`"), le modèle de pannes appliqué aux nœuds (mot-clef "`FAILURE`") et les capacités de calcul des nœuds (mot-clef "`COMPUTATION`").

Différents modèles de pannes sur les nœuds du réseau sont disponibles. Nous avons utilisé deux modèles en particulier qui sont des modèles génériques : le *modèle aléatoire* et le *modèle fixe*. Dans le premier cas, les pannes interviennent aléatoirement. L'utilisateur doit spécifier la probabilité de panne p qui est constante au cours de l'exécution et identique pour tous les nœuds du réseau. Il doit aussi indiquer la durée minimale m et maximale M des pannes. Lorsqu'un nœud tombe en panne (avec la probabilité p), il le reste durant un temps choisi uniformément aléatoirement dans l'intervalle $[m, M]$. La figure 2.10 (a) montre ainsi un exemple de diagramme

[7]Pour la description des autres composants, il faut se reporter à la notice d'utilisation de *Dasor* [Rab07b].

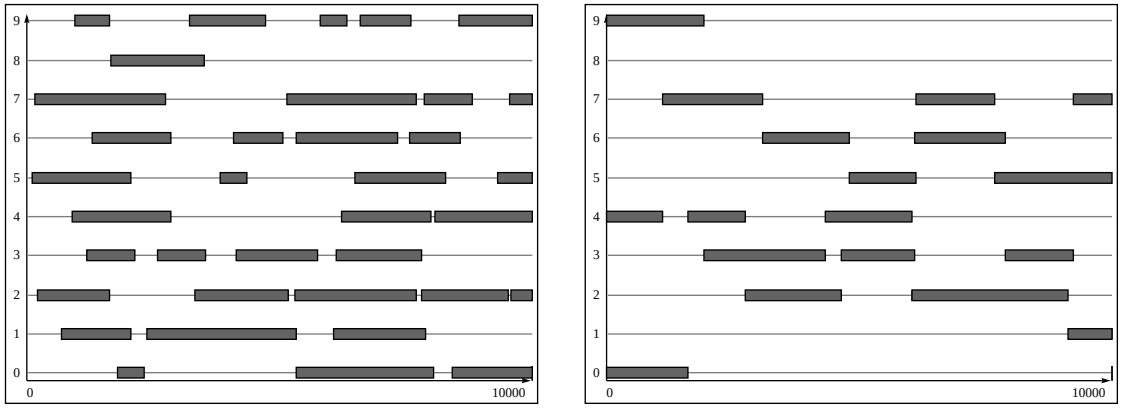
d'exécution obtenu avec ce modèle. Dans le modèle fixe, à tout moment de l'exécution de la simulation, seul un nombre n de nœuds sont en panne pour une durée choisie elle-aussi dans un intervalle. La figure 2.10 (b) montre le résultat d'une exécution avec $n = 3$. Nous observons ainsi qu'à tout moment, seuls 3 nœuds sont en panne. Ce modèle est particulièrement utile lorsque nous désirons analyser le comportement d'une application soumise à des pannes de nœuds. Dans cet exemple, nous pouvons ainsi comparer l'efficacité de la solution dans un réseau sans panne de 7 nœuds par rapport à un réseau de 10 nœuds avec 3 nœuds en panne.

Le modèle de calcul spécifie les capacités de calcul des nœuds. Par défaut, chaque nœud possède un nombre de processeurs p, chacun de rapidité r. Le nombre de processeurs détermine le nombre de tâches qui peuvent être exécutées simultanément (dans le cas du modèle de calcul simple). r est un coefficient qui permet de déterminer le temps nécessaire pour calculer une tâche de longueur l. Pour $r = 0.5$, il faut donc un temps de $2 \times l$ pour calculer la tâche. Dans nos travaux, nous utilisons le modèle fixe qui consiste à appliquer à tous les nœuds une puissance de calcul qui dépend de la loi de probabilité choisie (il peut aussi s'agir d'une constante).

Dans la seconde couche, un modèle de routage peut être appliqué. Suivant le modèle choisi, il permet de calculer les chemins entre les nœuds plus ou moins dynamiquement. Si le modèle de routage est ignoré, un voisin physique correspondant à la couche 1 représente un voisin atteignable via le routage. Par exemple, certains réseaux de recouvrement de protocoles pair-à-pair sont représentés par des réseaux petit-monde. Dans la couche 1, nous appliquons une topologie petit-monde et dans la couche 2, nous pouvons ignorer le protocole de routage. Comme nous l'avons précisé précédemment, ce choix empêche alors l'étude des phénomènes de congestion au niveau du réseau physique.

La troisième couche correspond au protocole de communication entre les nœuds. Par défaut, aucun modèle n'est appliqué. En fonction des couches sous-jacentes, l'envoi d'un message entre deux nœuds ne prend pas en compte les phénomènes de congestion des liens. Ainsi, tous les messages sont transmis directement, sans latence. Un modèle simpliste consiste à jouer sur le temps de communication des messages, indépendamment de leur taille. Différentes lois de probabilité sont utilisées pour calculer le temps mis par un message pour atteindre son destinataire.

Si *Dasor* propose des modèles pour les trois premières couches, il comporte aussi un gestionnaire de ressources pour simuler le calcul de tâches ou le transfert de ressources entre les nœuds. En particulier, il est possible de décrire les propriétés des tâches (puissance de calcul nécessaire, dépendances entre les tâches) dans le fichier de description. Elles peuvent être générées à l'aide d'une loi de probabilité ou bien à partir d'un fichier texte. La notion d'image de ressources est utilisée pour le transfert d'un fichier, d'une application ou des paramètres d'une tâche. Un message contenant une image de fichier, par exemple, permet de simuler le transfert de ce fichier dans le réseau.



`FAILURE= random 0.01 100 500 3000` (*a*)

`FAILURE= fixed 3 1000 2000` (*b*)

FIG. 2.10 – Diagramme d'exécution obtenu en appliquant un modèle de pannes aléatoire sur un réseau de 10 nœuds (*a*) et un modèle de pannes fixe (*b*).

2.3.6 La boîte à outils *Dasor*

Autour de la bibliothèque, nous avons développé différentes applications qui forment la boîte à outils *Dasor*. Elles sont utilisées dans la conception de nouveaux simulateurs ou pour exploiter les résultats obtenus lors de simulations.

`makeSimu` : il construit la structure nécessaire (les différentes classes) pour la création d'un nouveau simulateur. Les classes produites contiennent des méthodes sans code que le créateur de simulateur complète en détaillant les actions associées à chaque événement.

`statNet` : il extrait des statistiques ou des informations sur le réseau et sur les différents modèles contenus dans un fichier de description.

`convertNet` : un fichier de description contient la description d'un réseau. Cet outil propose de le convertir sous forme d'un fichier *postscript* (voir figures 1.7 et 1.8) ou d'un autre fichier de description en supprimant tout ou une partie des éléments aléatoires (conversion des modèles en événements, détail de chaque lien).

`makeDiag` : à partir des fichiers de description des simulations, cet outil construit des diagrammes d'exécution pour mettre en évidence le calcul des tâches, des pannes de nœuds (voir figure 2.10) ou encore les échanges de messages entre les nœuds. Il permet aussi d'extraire les différents événements qui ont été générés lors d'une simulation. `makeDiag` est utilisé à la fois pour débugger un simulateur que pour vérifier le comportement de les algorithmes simulés.

`simServer` et `simClient` : un simulateur est en général exécuté plusieurs fois pour obtenir des statistiques moyennes. Ces deux outils permettent d'exécuter des simulations en parallèle. Ils ont été utilisés pour réaliser différentes séries de simulations sur la machine parallèle Roméo 2 du centre de calcul régional de l'Université de Reims Champagne-Ardenne[8]. `simServer` doit être exécuté sur la machine parallèle ou sur une grappe de serveurs et se charge de la distribution des simulations. Il reconstruit un seul fichier de statistiques à l'aide des fichiers obtenus sur chaque nœud ou processeur. `simClient` soumet les simulations à `simServer` et exécute des requêtes pour obtenir l'état courant des calculs.

[8]Le site de Romeo2 est accessible à l'adresse http ://www.romeo2.fr.

2.3.7 Conception d'un simulateur

L'outil *makeSimu* automatise la construction de toutes les classes nécessaires pour l'écriture d'un nouveau simulateur. Chaque type de nœud du réseau correspond à une classe héritant des propriétés génériques d'un nœud, de même que chaque type de message correspond à une classe héritant des propriétés génériques d'un message.

Une classe correspondant à un type de nœud possède plusieurs méthodes, chacune étant exécutée lors de la réception d'un événement particulier qui survient sur le nœud. Il n'est pas nécessaire qu'elles soient toutes renseignées. Ainsi, les méthodes associées au calcul de tâches peuvent être ignorées si l'application n'est pas dédiée au calcul. L'écriture d'un simulateur consiste donc en la description des actions associées à chaque événement du système. Nous distinguons les événements suivants :

- *Initialisation* et *stop* : avant que la simulation ne débute, l'événement *initialisation* est exécuté sur chaque nœud de la grille. Il est utilisé pour initialiser les différentes variables locales ou pour mettre en place des compte-à-rebours. Lorsque la simulation s'arrête, chaque nœud exécute l'événement *stop* ;
- *Panne* et *réveil* : ces deux événements interviennent lorsqu'un modèle de panne est appliqué sur les nœuds du réseau. L'événement *réveil* correspond à l'initialisation du nœud lorsqu'il se réveille alors que l'événement *panne* permet de réaliser des actions lorsque le nœud tombe en panne ;
- *Début de tâche* et *fin de tâche* : lorsqu'une tâche est assignée à un nœud, un événement est généré pour détecter le début effectif de la tâche. De même, la fin d'une tâche provoque l'événement *fin de tâche* ;
- *Fin de compte-à-rebours* : sur chaque nœud, un ensemble de compte-à-rebours ou horloges peut être associé. Il permet de réaliser des actions à intervalles réguliers comme l'envoi de messages *ping* dans le protocole *Gnutella*. Lorsqu'un compte-à-rebours se termine, un événement est provoqué sur le nœud correspondant ;
- *Réception de messages* : lorsqu'un message est reçu sur un nœud, cet événement est généré afin d'analyser les données reçues. À la fin de la méthode, le message est automatiquement détruit à moins qu'il ne soit transmit à un autre nœud.

Lors de l'exécution d'un simulateur, l'utilisateur doit fournir un fichier de description qui permet de mettre en place les différents modèles. Une simulation se termine suivant trois cas : s'il n'y a plus d'événements à traiter, si le temps a atteint la borne maximale fixée en paramètre (ou dans le fichier d'initialisation) ou si un nœud demande explicitement la fin de la simulation. Le modèle d'exécution des simulateurs est détaillé sur la figure 2.11.

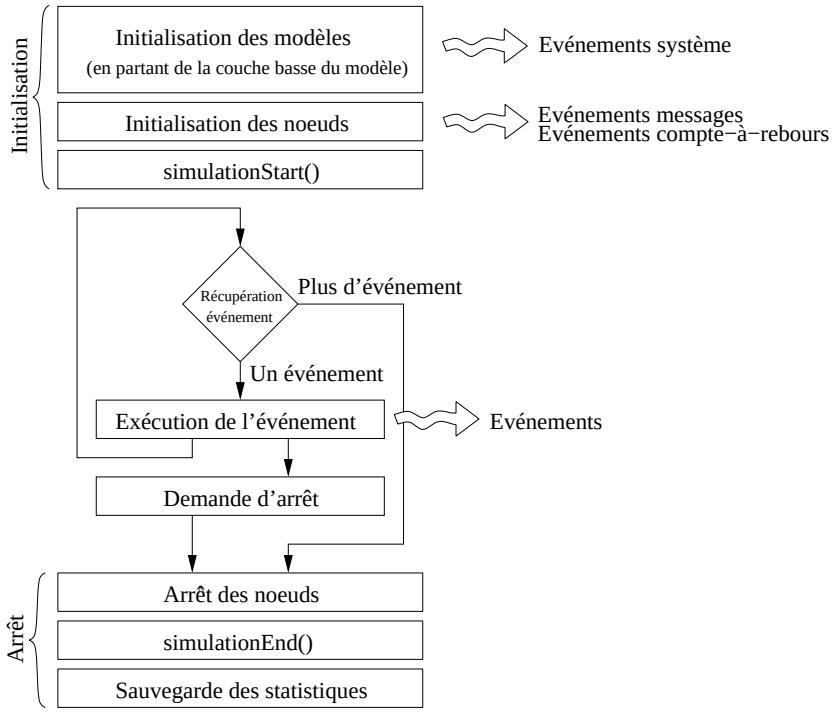


FIG. 2.11 – Diagramme d'exécution d'un simulateur écrit à l'aide de *Dasor*.

2.4 Simulations sur les marches aléatoires et le mot circulant

Nous présentons dans cette section, différents résultats de simulations concernant le temps de couverture des marches aléatoires et les conséquences des mécanismes de gestion de pannes exposés dans la section 1.4.5. Nous présentons ensuite des statistiques à propos du mot circulant, comme les caractéristiques des arbres générés ainsi que sur la validité des données en fonction des pannes dans le système. Les différentes simulations ont été réalisées à l'aide de la bibliothèque *Dasor*.

2.4.1 Simulations sur les marches aléatoires

Les différentes solutions que nous avons proposées et que nous exposons dans les chapitres suivants, sont à base de marches aléatoires et leurs efficacités dépendent en partie du temps de couverture du réseau. Dans cette section, nous proposons de simuler la circulation d'une marche aléatoire et d'observer son comportement en fonction de plusieurs critères comme le nombre de nœuds ou le degré moyen du graphe.

Simulations sans modèle de pannes

Les différentes bornes présentées dans la section 1.4.3 sont toutes exprimées en fonction de n (sauf le théorème de Matthews). Le temps de couverture dépend aussi des caractéristiques du graphe comme sa densité. Nous exécutons des simulations qui consistent à faire circuler une

marche partant d'un sommet choisi aléatoirement et à calculer le temps nécessaire pour obtenir des temps de couverture partiels C_p avec p égal à 25%, 50%, 75% et 100%. Nous augmentons le nombre de nœuds dans le graphe de 1000 à 10000 et nous obtenons les résultats présentés sur la figure 2.12. Les graphes utilisés pour les simulations sont des graphes aléatoires avec une probabilité de 0.6 d'avoir un lien (i, j).

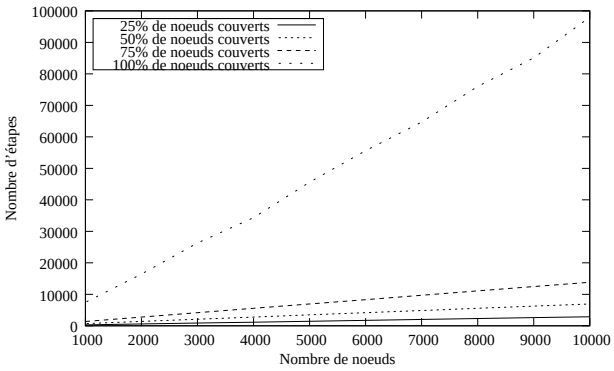


FIG. 2.12 – Temps de couverture partiels d'une marche aléatoire sur des graphes aléatoires en fonction du nombre de nœuds.

Sur l'intervalle de nœuds choisi, nous remarquons que le temps de couverture semble augmenter linéairement en fonction du nombre de nœuds, tout en restant très en dessous de n^3 : les graphes générés ont une densité importante, le temps de couverture des marches aléatoires est donc très court. De plus, nous observons qu'un temps de couverture partiel de 75% du graphe est obtenu rapidement comparé à la couverture complète. Pour un réseau de 1000 nœuds, il existe un facteur supérieur à 5 entre ces deux temps et cette différence s'amplifie jusqu'à un facteur 7 lorsque le nombre de nœuds atteint 10000.

Le temps de couverture dépend aussi de la nature du graphe. Nous réalisons des simulations sur des topologies particulières de 100 nœuds décrites dans la section 1.3.2. La figure 2.13 (a) présente les différents temps de couverture partiels obtenus et la figure 2.13 (b) présente l'écart type observé sur les temps de couverture d'une série de simulations.

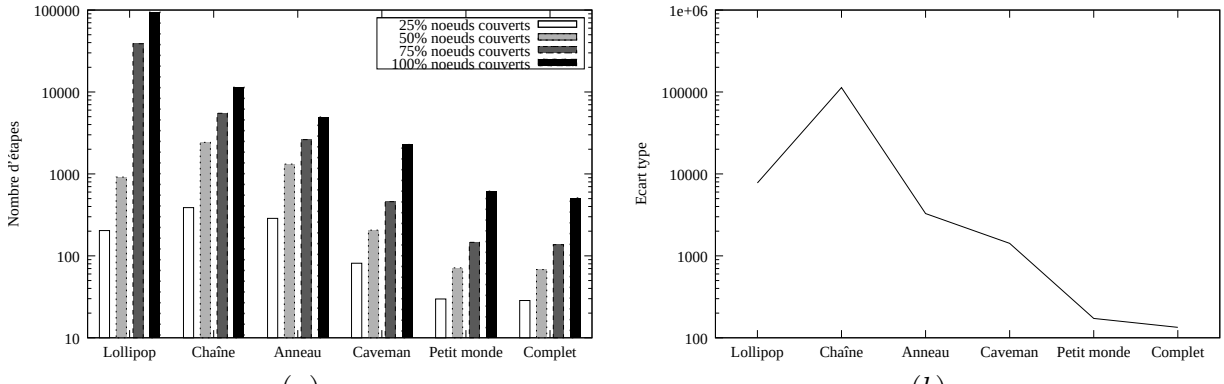


FIG. 2.13 – Temps de couverture partiels d'une marche aléatoire sur certaines topologies particulières avec 100 nœuds (a) et écarts types observés sur le temps de couverture complet (b).

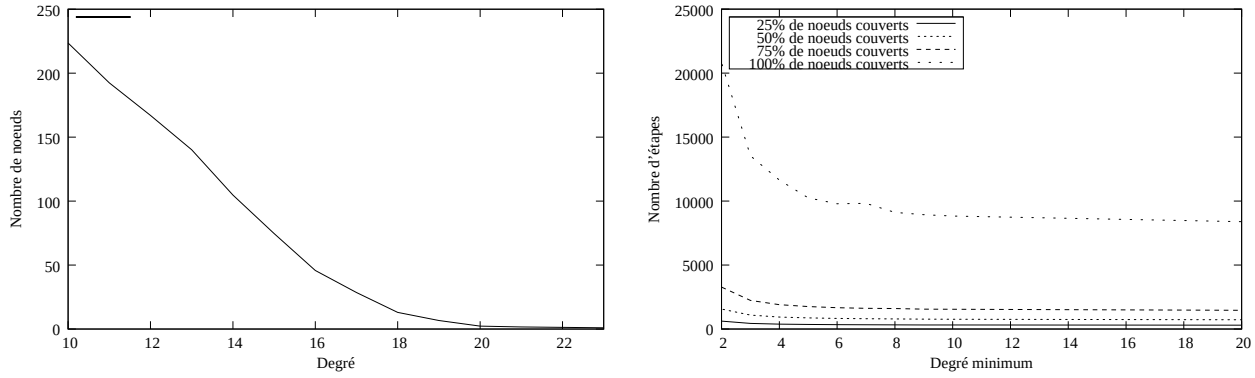
Les graphes de type *lollipop* sont souvent cités dans la littérature lors de l'étude sur les marches aléatoires. La clique favorise la circulation du jeton alors que les nœuds en bout de chaîne sont difficilement atteints. Ce phénomène explique ainsi l'écart important observé entre le temps nécessaire pour une couverture 50% et une couverture de 75%. La clique contient en effet 50% des nœuds. Les temps de couverture partiels sur les graphes lollipop sont donc très grands comparés aux autres topologies.

Lorsqu'un jeton se déplace dans une chaîne, il circule difficilement d'une extrémité à l'autre. Les temps de couverture partiels sont en moyenne plus faibles que sur une lollipop. Cependant, nous observons un écart type important sur le temps de couverture qui dépend du point de départ du jeton (choisi aléatoirement dans ces simulations). Lorsque les deux extrémités sont reliées, nous obtenons un anneau. Sans extrémité, le point de départ du jeton n'a plus d'importance. L'écart type sur le temps de couverture est donc plus faible. De même, les différents temps de couverture partiels sont légèrement plus faibles que sur une chaîne.

Les réseaux *caveman* sont constitués de sous-graphes connexes qui sont reliés entre eux (les graphes générés pour ces simulations ont comme paramètres $p_1 = 0.6$, $p_2 = 0.6$ et possèdent 10 caves). De la même manière que dans une lollipop, la marche aléatoire circule facilement au sein d'une cave mais passe difficilement entre les différentes caves. Nous pouvons observer cependant, que les temps de couverture partiels sont très largement inférieurs dans les réseaux *Caveman* que dans les réseaux *lollipop*.

Enfin, les réseaux petit-monde et les réseaux complets induisent les temps de couverture les plus faibles. Ils sont tous les deux particulièrement favorables à la circulation du jeton. L'écart type observé sur le temps de couverture est donc plus faible que dans tous les autres graphes présentés précédemment.

Les performances de couverture d'une marche aléatoire dépendent aussi du degré moyen des nœuds. Si nous nous intéressons plus particulièrement à des protocoles de type *Gnutella*, les graphes à degré minimum représentent assez bien l'aspect du réseau pair-à-pair obtenu. La figure 2.14 (a) montre la distribution moyenne des degrés dans des réseaux de degré minimum générés par *Dasor*. Nous réalisons des séries de simulations sur de tels graphes afin d'observer le comportement des marches aléatoires en fonction du degré minimum. La figure 2.14 (b) montre les résultats obtenus. Pour un degré faible, le temps de couverture est très important : la marche aléatoire couvre difficilement tout le réseau. Lorsque le degré augmente, le temps de couverture diminue assez rapidement et atteint un temps acceptable pour des réseaux dont le degré minimum est au moins de 8. Dans un réseau pair-à-pair, le nombre de voisins est en général assez important. Les graphes obtenus permettent donc aux marches aléatoires d'avoir des temps de couverture assez faibles.



(*a*) (*b*)

FIG. 2.14 – Distribution moyenne des degrés dans des réseaux aléatoires de 1000 nœuds à degré minimum égal à 10 (*a*) et temps de couverture d'une marche aléatoire en fonction du degré minimum (*b*).

Simulations avec modèle de pannes

Dans la section 1.4.5, nous proposons de gérer la disparition des jetons à l'aide de compte-à-rebours T_i. Leur valeur d'initialisation, notée t_{ini}, n'est pas simple à déterminer car elle dépend de plusieurs paramètres. En particulier, elle dépend du temps de retour du jeton : lorsque le compte-à-rebours est initialisé, il faut attendre que le jeton retourne sur le nœud pour le réinitialiser. Pour déterminer t_0, nous pouvons calculer la valeur du temps de retour pour chaque nœud. Nous avons expliqué que ces valeurs peuvent être calculées de manière exacte (voir [BS07]). Cependant, le dynamisme du réseau d'une part, et le point de vue uniquement local des nœuds d'autre part, nous empêchent d'utiliser de telles méthodes. Nous devons donc appliquer une méthode générale qui consiste à sélectionner la valeur t_{ini} indépendamment du nœud. Mais afin d'éviter la régénération simultanée de nombreux jetons lors de la perte du jeton, chaque nœud choisit t_{ini} aléatoirement dans un intervalle I_{ini}. Les bornes de cet intervalle dépendent du nombre de nœuds dans le réseau et du temps de communication.

À l'aide de différentes simulations, nous cherchons à observer l'influence des valeurs de t_{ini}. Sur des réseaux aléatoires de 1000 nœuds, nous appliquons un modèle de pannes fixe : à tout moment, seulement 80% des nœuds sont présents dans le réseau. Nous faisons varier la valeur de l'intervalle $I_{ini} = [n.t_c.j, n.t_c.(j+1)]$ avec $1 < j < 9$ et t_c étant le temps de communication moyen. Nous calculons deux grandeurs différentes : la *période d'invalidité* pendant laquelle il existe au moins deux jetons et la *période de famine* pendant laquelle il n'en existe aucun. Les résultats sont présentés sur la figure 2.15.

Pour des valeurs de j faibles, la période d'invalidité représente une grande partie du temps d'exécution : des jetons sont créés alors qu'un jeton existe toujours dans le graphe. Cette période diminue fortement lorsque nous augmentons j alors que dans le même temps, la période de famine augmente régulièrement. Lorsque j est supérieur à 8, il n'existe plus de période d'invalidité. Un compromis doit donc être trouvé et il dépend de l'application et du graphe. Dans le cadre de l'allocation de ressources, une valeur élevée de j est recommandée. En effet, il est préférable de n'avoir aucun jeton dans le réseau que d'en avoir plusieurs pour éviter que des conflits apparaissent. Au contraire, pour une diffusion d'informations basée sur une marche aléatoire, il faut éviter les périodes de famine. Une valeur de j plus faible doit donc être

envisagée. La figure 2.15 montre aussi la période d'instabilité qui est la somme des périodes d'invalidité et de famine. Nous observons que sur les graphes aléatoires de 1000 nœuds, la période d'instabilité est au minimum pour $j \approx 5$.

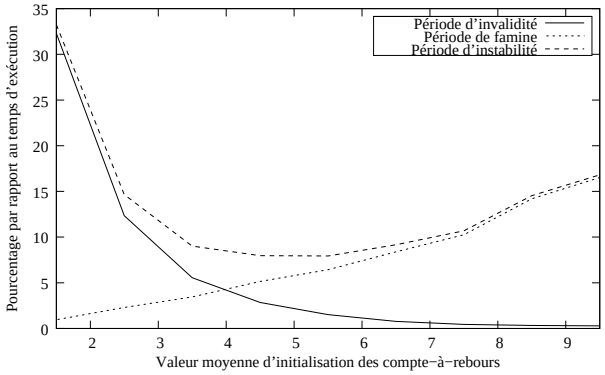
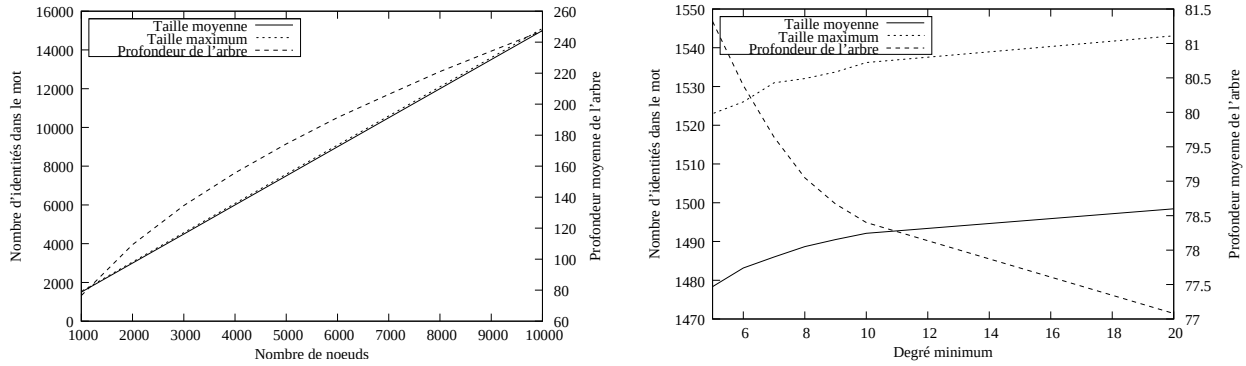


FIG. 2.15 – Périodes d'invalidité et de famine lors d'une exécution sur un réseau aléatoire de 1000 nœuds en fonction de l'intervalle $T_{ini} = [n.t_c.j, n.t_c.(j+1)]$ avec $1 < j < 9$.

2.4.2 Observations sur les performances du mot circulant

La borne maximum sur la taille du mot circulant est de $2n - 1$. Mais sa taille en cours d'exécution dépend de la circulation du jeton qui dépend elle-même des caractéristiques du graphe. Nous exécutons plusieurs séries de simulations afin d'observer la taille moyenne et maximum du mot circulant. La figure 2.16 montre les résultats obtenus sur des réseaux aléatoires dans lesquels nous faisons varier le nombre de nœuds. Nous observons que la taille moyenne augmente linéairement mais reste en-dessous de la borne maximum : nous obtenons une taille moyenne proche de $1.5 \times n$ pour des graphes entre 1000 et 10000 nœuds.

Le mot circulant permet de maintenir un arbre couvrant du sous-graphe visité. Nous remarquons que la profondeur moyenne des arbres est relativement importante. Pour des graphes de 1000 nœuds, elle est proche de 80 (dans ces simulations sur des réseaux aléatoires). La figure 2.16 (b) présente les résultats de simulations sur des graphes à degré minimum. Lorsque le degré est faible, la taille du mot est elle aussi assez faible et la profondeur de l'arbre importante. Cela s'explique par le degré moyen des nœuds qui plus faible dans l'arbre, ce qui diminue le nombre d'occurrences constructives dans le mot. Lorsque le degré minimum du graphe augmente, la hauteur de l'arbre diminue, ce qui implique que la taille du mot augmente. Dans ce cas, le mot passe plusieurs fois par un même nœud en arrivant de voisins différents. Ce nœud possède donc plus de fils dans l'arbre.

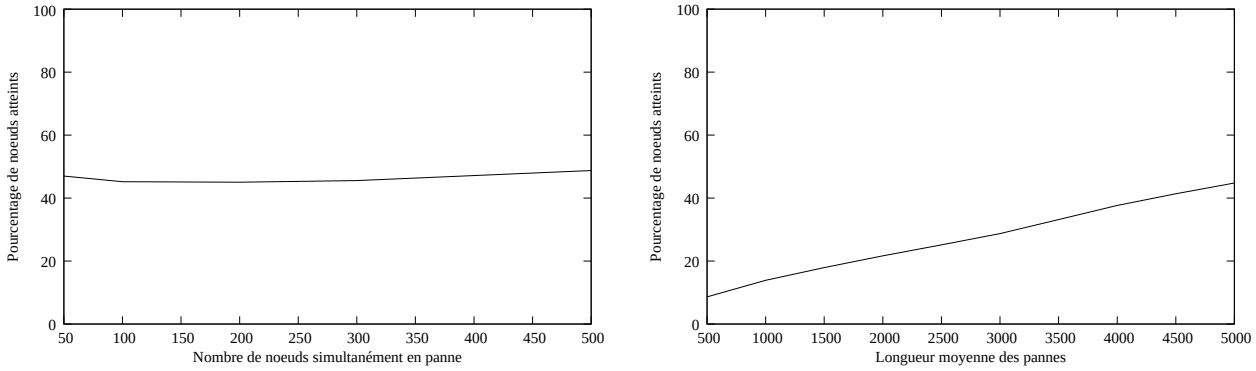


(*a*) (*b*)

FIG. 2.16 – Taille moyenne et maximum du mot circulant et profondeur moyenne des arbres obtenus en fonction du nombre de nœuds dans des graphes aléatoires (*a*) et en fonction du degré minimum de graphes de 1000 nœuds (*b*).

L'arbre construit dans le mot circulant a une durée de validité qui dépend du degré de dynamisme du réseau. Nous cherchons à calculer le pourcentage de nœuds contenus dans le mot circulant qui sont effectivement atteignables en fonction de l'arbre construit. Le simulateur que nous avons écrit construit un arbre à partir de chaque mot circulant reçu, enraciné sur le nœud courant. Nous comparons ensuite l'arbre avec la topologie réelle du réseau. Nous obtenons un taux de couverture du mot circulant en comparant les liens de communication du réseau et les relations père-fils dans l'arbre. Nous proposons d'appliquer le modèle de pannes fixe présenté dans la section 2.3.5. Dans un premier temps, nous analysons le taux de couverture en fonction du nombre de nœuds qui sont en panne dans le réseau. La figure 2.17 (*a*) présente les résultats sur des réseaux aléatoires de 1000 nœuds. Nous observons que le taux de couverture du mot est sensiblement constant même pour un nombre important de pannes (jusqu'à 50%) et ce pour des pannes durant en moyenne 5000 étapes.

Le modèle permet aussi de spécifier la durée des pannes qui est choisie aléatoirement dans un intervalle. La figure 2.17 (*b*) présente l'évolution du taux de couverture du mot en fonction de la longueur moyenne des pannes. Le modèle choisi implique que le réveil d'un nœud correspond à la panne d'un autre nœud dans le réseau. Aussi, pour des longueurs moyennes très courtes, le taux de couverture du mot est très faible : le contenu du mot ne peut pas être mis à jour assez rapidement par rapport aux pannes survenant dans le réseau.



(*a*) (*b*)

FIG. 2.17 – Taux de couverture du mot circulant en fonction du nombre de nœuds simultanément en panne dans le réseau (*a*) et en fonction de la longueur moyenne des pannes (*b*).

2.5 Conclusion

La conception d'applications distribuées repose sur le choix d'un modèle théorique. Dans ce chapitre, nous avons présenté un modèle original pour les applications de grille ou pair-à-pair. Il est consisté de 5 couches superposées qui interagissent les unes avec les autres. En particulier, il met en évidence les mécanismes sous-jacents à une grille dans le but de mettre en place des solutions de gestion de pannes.

D'autre part, nous avons présenté *Dasor* qui est une bibliothèque de simulation à événements discrets. Elle permet de construire des simulateurs dont le modèle d'exécution est basé sur le modèle théorique. Elle propose un choix du niveau d'abstraction de la simulation très large. L'écriture d'un simulateur est ainsi réalisée indépendamment du réseau et des différents modèles de simulation qui sont appliqués sur les nœuds. Sans modification du code du simulateur, il est possible de réaliser des simulations dans des environnements complètement différents.

Le couple formé du modèle théorique et de la bibliothèque de simulation est un outil puissant pour la conception d'applications de grille ou pair-à-pair. Il permet de guider le concepteur depuis l'étape d'analyse jusqu'à l'étape de conception proprement dite.

À l'aide de la bibliothèque *Dasor*, nous avons écrit plusieurs simulateurs afin d'observer le comportement des marches aléatoires et du mot circulant présentés dans le chapitre précédent. Les différents résultats obtenus corroborent les différentes bornes théoriques. De plus, nous avons vu que ces outils supportent un degré de dynamisme des nœuds assez important et proposent de très bonnes performances avec une facilité de mise en œuvre.

Chapitre 3

Gestion des ressources dans des réseaux dynamiques

Résumé : *La gestion des ressources est un mécanisme essentiel dans toute application distribuée et plus particulièrement pour les applications de grille où les ressources mises en commun sont souvent très hétérogènes et dynamiques. La gestion des ressources est constituée d'un ensemble de composants et nous trouvons entre autres la gestion de la topologie, la recherche et le transfert de ressources. La complexité de ces composants est importante si nous prenons en compte les différents problèmes de sécurité, le dynamisme des ressources ainsi que le passage à l'échelle de l'application. Nous proposons dans ce chapitre, une solution complètement distribuée exploitant un mot circulant contenu dans un jeton circulant aléatoirement. Cette solution est basée sur le modèle théorique que nous avons présenté dans la section 2.2 : le réseau est représenté par un graphe orienté. Les différents algorithmes de gestion du mot circulant présentés dans la section 1.5 fonctionnent sur des graphes non-orientés. Nous présentons donc dans ce chapitre une nouvelle méthode pour gérer le mot circulant et nous en proposons différentes applications, comme la recherche de ressources ou la construction d'un réseau recouvrant pour les réseaux pair-à-pair. Nous analysons aussi des résultats de simulations obtenus avec la bibliothèque* Dasor. *Les travaux présentés dans ce chapitre ont été sujets à une présentation lors du congrès de la* ROADEF *en 2006 [BBFR06b] et publiés dans [BBFR06a].*

3.1 Introduction

La gestion des ressources est essentielle dans toute application distribuée. Cette gestion peut être statique, auquel cas les ressources doivent être connectées au démarrage de l'application. Le nombre de ressources utilisées est constant tout au long de l'exécution. Avec une gestion dynamique, les ressources peuvent se connecter et se déconnecter tout au long de l'exécution. Les grilles sont caractérisés notamment par le dynamisme de leurs ressources. C'est donc une gestion des ressources dynamique qui est nécessaire.

La gestion des ressources pour les applications de grille peut être vue comme un ensemble de composants qui dépendent du type de l'application et des ressources à gérer. La figure 3.1 montre quelques composants rencontrés dans la plupart des intergiciels de grilles comme *Globus* ou *DIET* plongés dans notre modèle théorique. Ils sont à la fois dans la couche 4 et dans la couche 5.

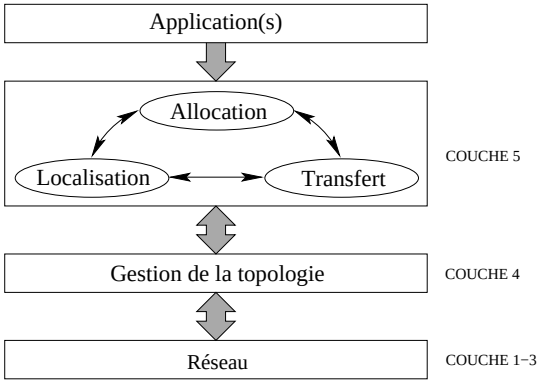


FIG. 3.1 – Les principaux composants de la gestion des ressources plongés dans notre modèle théorique.

La gestion de la topologie (couche 4) assure la maintenance de la connectivité entre les différents nœuds de l'application. Elle gère donc les connexions des nouvelles ressources en les rendant accessibles pour les autres nœuds. De même, lorsqu'un ou plusieurs nœuds se déconnectent, l'ensemble du système doit continuer à fonctionner. Dans des systèmes pair-à-pair, cela se traduit par la création d'une surcouche du réseau appelé *réseau recouvrant* ou *réseau de recouvrement* (pour *Overlay Network*) : des mécanismes de routage peuvent être appliqués et la volatilité des nœuds doit être gérée. La construction de cette surcouche dépend des contraintes de l'application et est confrontée la plupart du temps à des problèmes liés aux politiques de sécurité des différents domaines traversés.

Dans le cadre du calcul haute-performance qui consiste à connecter des machines parallèles ou des grappes de serveurs, les différentes entités sont reliées via des réseaux haut-débit dédiés comme le réseau *Renater* pour le projet *Grid'5000* [GRI] et ont en général une fiabilité importante. La gestion de la mobilité est inutile comparée à une application pair-à-pair d'échange de fichiers dont les différentes entités sont reliées via Internet et qui sont caractérisées par leur hétérogénéité et leur volatilité. Comme expliqué précédemment, la gestion de la topologie dépend aussi de l'application. Dans le cas de *SETI@home*, le choix s'est porté sur une architecture totalement centralisée malgré le nombre important de nœuds. Cependant, les données sont générées sur un serveur unique et les connexions qui sont établies entre les clients et le serveur ne sont maintenues que pendant un temps très court : les données nécessaires au calcul sont envoyées aux clients et les résultats sont retournés plusieurs heures après. Cela permet de gérer de nombreuses connexions qui ne sont jamais établies au même moment (un mécanisme de retardement permet d'éviter la saturation du serveur). La gestion de la topologie est donc simplifiée, ce qui permet un passage à l'échelle plus aisé. Dans *DIET*, les différentes ressources sont connectées via une hiérarchie de serveurs appelés agents. La charge du serveur unique est allégée, ce qui augmente la capacité de traitement des requêtes et entraîne une plus grande réactivité. Cette hiérarchisation améliore aussi la tolérance aux pannes : si un agent tombe en panne, les agents situés en-dessous dans la hiérarchie, se connectent à l'agent du dessus.

Au-dessus de la gestion de la topologie, au sein de la couche 5, nous trouvons les autres composants de la gestion des ressources. La localisation de ressources permet aux nœuds d'en-

trer en contact avec les ressources correspondantes à une requête donnée. Pour des applications qui mettent en commun des ressources différentes, un catalogue est généralement mis en place. Il peut être complètement centralisé sur un unique serveur comme *NetSolve*, partiellement décentralisé sur plusieurs serveurs comme *Globus* ou encore complètement décentralisé en utilisant le mécanisme de *Chord* [SMK+01] qui propose des tables de hachage distribuées (ou *DHT* pour *Distributed Hash Table*). Lorsqu'un catalogue est présent au sein de l'application, toutes les requêtes concernant les ressources passent par des nœuds intermédiaires qui se chargent de retrouver la ressource correspondante suivant différents critères (charge actuelle, localité, architecture matérielle...). La stratégie de déploiement de ce catalogue est donc cruciale : s'il est placé sur un nœud inadéquat, les performances de toute l'application chutent.

Une fois les ressources localisées, il est nécessaire de les allouer. Cette étape dépend de l'architecture de la machine qui possède la ressource. Elle est souvent laissée à la charge des gestionnaires de ressources locaux fournis par le système d'exploitation dans le cas de machines parallèles ou de grappes de serveurs. Aussi, dans *Globus*, le service *GRAM* est mis en relation avec des gestionnaires locaux très différents comme *LSF*, *EASY-LL* ou *NQE* comme illustré sur la figure 3.2 (*a*). La figure 3.2 (*b*) présente une autre stratégie qui est celle utilisée dans *CONFIIT*. Pour gérer la volatilité des nœuds, une approche pair-à-pair a été choisie. Totalement distribuée, cette solution ne nécessite pas la mise en place d'un catalogue et l'allocation des ressources est laissée à la charge des nœuds : si les ressources sont disponibles, la tâche est exécutée sans appel à un service ou un serveur externe car les paramètres sont connus localement. Les coûts de communication sont réduits et la gestion des ressources est répartie sur les nœuds de la grille.

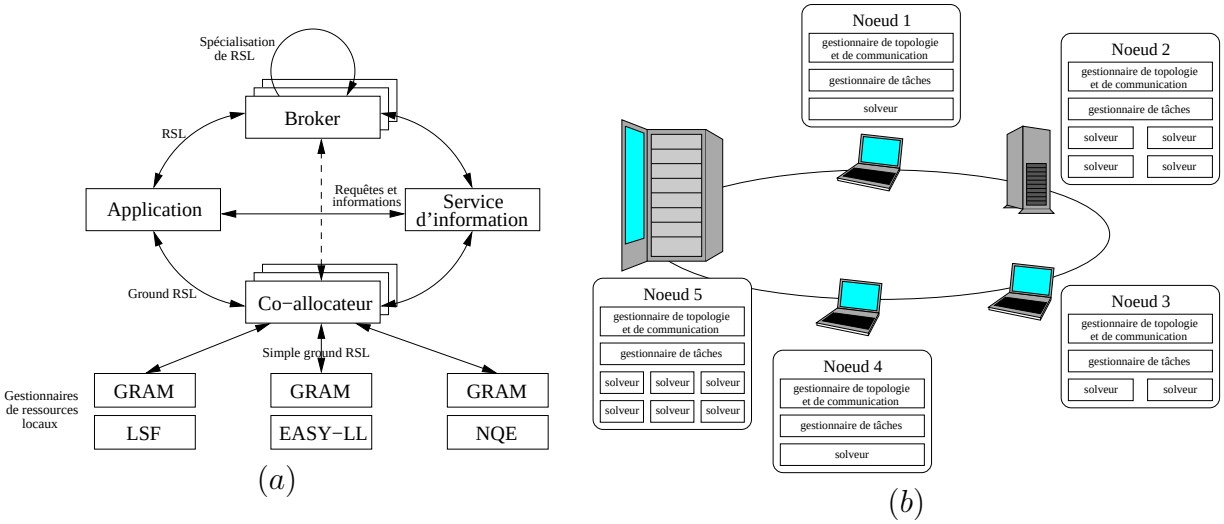


Fig. 3.2 – Illustration du mécanisme d'allocation de ressources dans *Globus* via le service *GRAM* (*a*) et organisation totalement distribuée du gestionnaire de ressources (ici, gestionnaire de tâches) de *CONFIIT* (*b*).

La plupart des solutions qui nécessitent la mise en place de services sur des nœuds particuliers ne sont pas efficaces lorsque l'architecture physique du réseau n'est pas connue et que les ressources sont dynamiques. Or, les nœuds d'une application de grille peuvent être disséminés

dans le monde entier et les connexions et déconnexions sont très nombreuses. Ce phénomène observé dans les réseau pair-à-pair est appelé le "*churn*" : l'application doit gérer le va-et-vient des nœuds. Dans le cas de *DIET*, les serveurs sont interconnectés hiérarchiquement afin de limiter la congestion sur un serveur unique. Pour une question de performances, ces serveurs doivent être placés correctement et calqués sur l'architecture du réseau local. Pour *CONFIIT*, les différents nœuds sont placés dans un anneau virtuel comme dans *Chord*. Si le choix d'une solution complètement décentralisée paraît plus tolérante en panne, la mise en place et la maintenance de cette structure virtuelle doit prendre en compte les problèmes liés aux pare-feux qui empêchent l'établissement de certaines connexions. De plus, en cas de pannes, elle peut être coûteuse en terme de messages.

Nous proposons dans ce chapitre, une gestion des ressources basée sur l'utilisation des marches aléatoires et le mot circulant. Elle est complètement distribuée et supporte l'hétérogénéité des ressources (puissances de calcul diverses et réseau d'interconnexion hétérogène). L'utilisation de ce couple d'outils permet à l'application d'être particulièrement adaptée aux réseaux dynamiques. Le nombre de messages de contrôle échangés est plus faible quelles que soient les pannes survenant dans le système : cette solution est donc adaptée pour des réseaux à faible débit. De plus, aucune route prédéfinie n'est utilisée et aucun nœud n'est différencié, ce qui augmente encore la tolérance aux pannes de cette solution.

Ce chapitre est organisé de la manière suivante. Dans la prochaine section, nous exposons le problème lié à l'utilisation du mot circulant dans un réseau orienté. Nous proposons notamment de nous intéresser aux cycles formés dans le mot afin d'accélérer la gestion du mot. Ensuite, nous proposons un algorithme général pour exploiter cet outil. Dans la section 3.4, nous nous intéressons aux pannes pouvant intervenir au sein de l'application avec les conséquences sur le mot circulant. Pour illustrer cette nouvelle gestion, nous proposons des résultats de simulations dans la section 3.5. Dans la dernière section, nous proposons plusieurs applications au mot circulant orienté.

3.2 Gestion du mot circulant dans un graphe orienté

Comme nous l'avons précisé précédemment, nous nous intéressons dans ce chapitre à une solution complètement distribuée basée sur le mot circulant. Nous proposons une nouvelle gestion du mot afin de l'adapter à l'orientation des liens de communication. Cette gestion est basée sur l'exploitation des cycles formés dans le mot par la circulation du jeton dans le réseau.

3.2.1 Présentation générale

L'algorithme de circulation du mot circulant est identique à celui du mot circulant non-orienté[1] : lorsque le mot est reçu par un nœud, nous ajoutons son identité dans la liste à la première position (*i.e.* la nouvelle identité devient W_1). La gestion des ressources correspond à la couche 4 de notre modèle théorique. Or, les différents liens de communication peuvent être orientés dus aux différentes politiques de sécurités rencontrées dans le réseau. Ainsi, cela se

[1]Par abus de langage, nous appelons *orienté* un mot circulant contenant des liens de communication orientés. De même, nous parlerons de *mot circulant non-orienté* la solution proposée dans [Fla01, BBF04b].

traduit dans le mot circulant par une orientation de la liste : deux identités successives dans le mot représentent un lien orienté de droite à gauche.

Exemple 3.1 *La figure 3.3 (a) montre un exemple de circulation d'un mot dans un graphe non-orienté. Lorsque le jeton arrive sur le nœud 3, il est possible de construire un chemin du nœud 3 vers le nœud 1 en passant par le nœud 2. Par contre, sur la figure 3.3 (b), la connaissance topologique incluse dans le mot est insuffisante pour atteindre le nœud 1 depuis le nœud 3.*

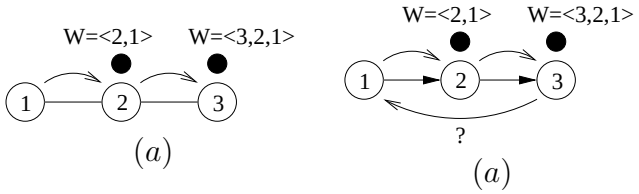


FIG. 3.3 – Problème de l'orientation des liens avec le contenu du mot.

La gestion des ressources consiste à maintenir l'interconnexion entre les nœuds de la grille. Autrement dit, cela peut être vu comme la maintenance d'un chemin entre chaque couple de nœuds de la grille. Or, contrairement au mot circulant non-orienté, lorsque nous ajoutons une nouvelle identité, nous ne connaissons que le chemin pour atteindre le nœud courant à partir des autres nœuds contenus dans le mot et non l'inverse. La gestion proposée dans [Fla01] n'est donc pas applicable dans notre cas, seul un arbre couvrant étant maintenu. Notre but est de maintenir un sous-ensemble du graphe de communication dans le mot circulant comportant un chemin entre tout couple de nœuds. Nous devons donc modifier complètement la gestion du contenu du mot afin d'éviter de perdre des informations topologiques.

Nous pouvons remarquer une position particulière au sein du mot circulant. Il est possible de construire un chemin depuis l'identité située à cette position vers toutes les autres identités. Cette position n'est pas unique et la dernière position du mot répond tout le temps à cette propriété. Aussi, parmi toutes les positions répondant à cette propriété, nous nous intéressons plus particulièrement à celle située la plus à gauche possible : nous l'appelons la *position minimale.*

Définition 3.1 *Une position est appelée* minimale *et notée* pos_{min} *dans un mot* W*, si elle satisfait la propriété suivante :*

$$pos_{min} = min\,\{i \in [1, taille(W)] / \forall k \in identités(W), \exists j \leq i, W_j = k\}$$

La position minimale dans un mot peut être calculée grâce à l'algorithme 6.

Remarque 3.1 *La position minimale est unique dans le mot.*

Algorithme 6 Calcul de la position minimale pos_{min} dans un mot circulant W

$pos_{min} \leftarrow 1$
$visités \leftarrow \emptyset$
$i \leftarrow 1$
Tant que $i \leq taille(W)$ **Faire**
  Si $W_i \notin visités$ **Alors**
    $visités \leftarrow visités \cup \{W_i\}$
    $pos_{min} \leftarrow i$
  Fin Si
  $i \leftarrow i + 1$
Fin Tant que

Exemple 3.2 *La figure 3.4 montre un exemple de mot circulant après des déplacements aléatoires dans un graphe. Le mot circulant obtenu est* $W =< 1, 3, 2, 4, 5, 3, 2 >$. *La position minimale est donc* $W_5 = 5$ *et il est possible de construire un arbre couvrant enraciné en 5. Le mot circulant n'a pas encore visité le nœud 6, mais la construction de l'arbre est tout de même possible.*

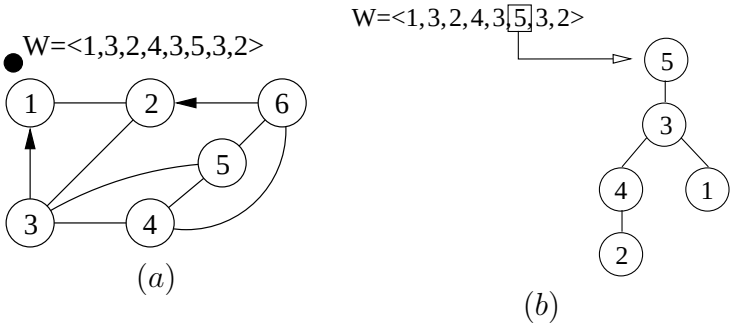


FIG. 3.4 – Mot circulant résultant de la circulation d'un jeton dans un graphe (a) et construction d'un arbre couvrant enraciné sur la position minimale (b).

Depuis la position minimale, il est possible de construire des chemins vers tous les autres nœuds. Mais elle est insuffisante pour construire tous les chemins entre tout couple de nœuds. Nous sommes dans le cadre orienté, aussi si un nœud exécute une requête diffusée le long de cet arbre, la réponse ne peut être retournée sur ce même arbre. Si nous reprenons l'exemple précédent, le nœud 1 est le fils du nœud 3 dans l'arbre enraciné en $W_{pos_{min}}$. S'il peut recevoir un message provenant de 3, il ne peut lui envoyer de message, le lien correspondant dans le graphe étant orienté.

La position minimale est cependant stratégique dans le mot : si nous trouvons un élément à la position j tel que $W_1 = W_j$ et que $j > pos_{min}$, il est possible de construire un arbre couvrant enraciné en tout nœud. Cela se traduit par l'existence d'un cycle dans le mot que l'on peut écrire sous la forme $< W_1, \ldots, W_{pos_{min}}, \ldots, W_j, \ldots >$. Notre méthode de gestion du mot circulant se base donc sur cette observation et sur la détection de tels cycles.

3.2.2 Cycle constructeur et réductions du mot

Dans [Fla01], l'auteur propose de réduire la taille du mot en utilisant plusieurs méthodes mais qui ne peuvent être exploitées dans un graphe non-orienté. Comme dit précédemment et en accord avec notre modèle, nous devons prendre en compte l'orientation des liens de la grille afin d'atteindre le maximum de ressources.

Pour réduire la taille du mot, une solution simple est de calculer tous les chemins entre chaque couple de nœuds dans le but de déterminer les parties inutiles du mot et de les supprimer. Cependant, la complexité d'un tel algorithme est trop importante sachant que cette réduction doit être réalisée à chaque réception du jeton. Nous proposons donc ici une solution non-optimale mais satisfaisante en terme de complexité. Notre méthode est basée sur les cycles contenus dans le mot.

Définition 3.2 *Un mot* W *contient un cycle s'il existe* $(i, j) \in [1, taille(W)]^2$, $i < j$ *et* $W_i = W_j$. *Nous notons ce cycle* $\mathcal{C}(i, j)$.

Les propriétés d'un cycle impliquent que pour chaque couple d'identités incluses dans un cycle, il est possible de construire un chemin de l'une vers l'autre et inversement. Si ce cycle contient toutes les identités du sous-graphe visité par le jeton, il est alors possible de construire des chemins de toutes les identités contenues dans le mot vers toutes les autres. Nous appelons un tel cycle, un *cycle constructeur*.

Définition 3.3 *Un cycle* $\mathcal{C}(i, j)$ *contenu dans un mot* W *est dit* constructeur *si :*

$$\forall k \in identités(W), \exists l \in [i, j]/W_l = k$$

Un tel cycle est noté $\mathcal{C}_C(i, j)$.

Exemple 3.3 *Soit le mot circulant* $W =< 1, 2, 4, 3, 1 >$. *Ce mot contient un cycle constructeur* $\mathcal{C}_C(1, 5)$. *En particulier, entre les identités 2 et 3, nous pouvons construire un chemin : de 2 vers 1 puis de 1 vers 3. Le chemin de 3 vers 2 est : de 3 vers 4 puis de 4 vers 2.*

Remarque 3.2 *Lorsqu'un nouveau mot est créé (*i.e. *il ne contient alors que l'identité du nœud qui l'a créé), le premier cycle constructeur est obtenu lorsque le mot revient sur le nœud de départ.*

Avoir un cycle constructeur dans le mot circulant est suffisant mais pas nécessaire pour construire les chemins entre tout couple d'identités. Par exemple, le mot $< 1, 2, 3, 1, 4, 3 >$ ne contient pas de cycle constructeur et pourtant, il est possible de construire un chemin entre chaque couple d'identités. Nous pouvons remarquer que sur cet exemple, il est possible de récrire le mot comme le montre la figure 3.5 pour obtenir un cycle constructeur. Nous choisissons de ne pas détailler cette technique ici : nous verrons dans la suite, qu'un cycle constructeur est construit en un temps fini, ce qui limite les manipulations sur le mot et donc accélère son traitement.

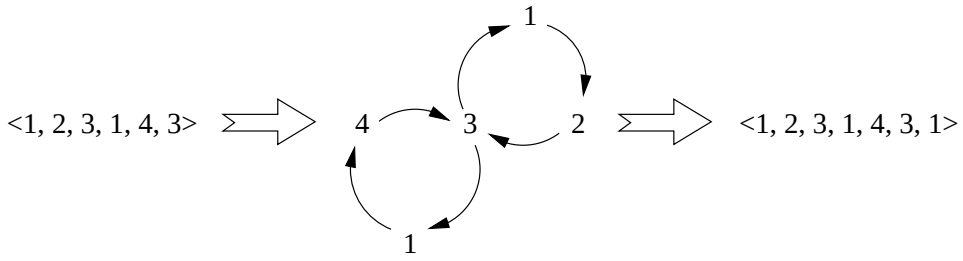


FIG. 3.5 – Exemple de réécriture d'un mot contenant deux cycles imbriqués pour obtenir un cycle constructeur grâce à une rotation de ces cycles.

Notre but étant de réduire la taille du mot sans perdre d'informations topologiques utiles, nous pouvons observer que si un mot W possède un cycle constructeur $\mathcal{C}_C(i,j)$, il est possible de supprimer la partie $droite(W, j+1)$, ce que nous appelons une *réduction terminale*. À noter que la partie à droite du mot est la partie la plus ancienne, les identités étant ajoutées à gauche du mot.

Définition 3.4 (Réduction terminale) *Soit un mot contenant un cycle constructeur $\mathcal{C}_C(i,j)$, la partie du mot $< W_{j+1}, \ldots, W_{taille(W)} >$ contient des informations redondantes et peut être supprimée. Nous appelons une telle réduction une* réduction terminale.

Notre méthode est basée sur les cycles constructeurs. Mais est-il possible de réduire la taille du cycle constructeur ? La première remarque que nous pouvons faire est que toute partie supprimée du mot ne doit pas entraîner d'incohérences topologiques en ajoutant par exemple des liens inexistants. En effet, si nous supprimons une partie du mot $< W_i, \ldots, W_j >$, nous devons nous assurer que le lien de communication (W_{i-1}, W_{j+1}) existe. Si nous avons un cycle $\mathcal{C}(i,j)$, il est possible de supprimer sans ajouter de lien la partie $< W_{i+1}, W_j >$ puisque le lien de communication (W_j, W_{j+1}) est équivalent à (W_i, W_{j+1}). Le cycle $\mathcal{C}(i,j)$ peut être supprimé si il ne contient que des informations redondantes. Nous l'appelons alors *cycle redondant*.

Définition 3.5 (Cycle redondant) *Un cycle $\mathcal{C}(i,j)$ est dit redondant*[2] *si :*

$$\forall k, i < k < j, \exists l \in [1, i[\cup]j, taille(W)] / W_k = W_l$$

Un tel cycle est noté $\mathcal{C}_R(i,j)$.

Un cycle redondant ne peut être réduit que si sa suppression n'entraîne pas la perte du cycle constructeur.

Remarque 3.3 *Lorsqu'un cycle redondant $\mathcal{C}_R(i,j)$ est détecté à l'intérieur d'un cycle constructeur, il peut être supprimé en supprimant la partie du mot $< W_{i+1}, \ldots, W_j >$.*

Le but de l'algorithme 7 est de rechercher et de supprimer tous les cycles redondants inclus dans un mot W. La recherche s'effectue de droite à gauche, c'est-à-dire de la partie la plus ancienne vers la partie la plus récente. Pour chaque identité, il faut rechercher si une identité

[2]Dans [BBFR06a], un tel cycle est appelé non-constructeur. Pour éviter la confusion entre un cycle non-constructeur et un cycle qui n'est pas constructeur, nous le renommons *cycle redondant*.

identique existe à gauche de celle-ci. Dans ce cas, un cycle est trouvé et il reste à déterminer si les identités incluses dans ce cycle existent aussi à l'extérieur du cycle. Pour accélérer la recherche, un ensemble $\mathcal{V}$ est mis à jour contenant toutes les identités rencontrées à droite du cycle. Il reste donc à rechercher les identités du cycle privées de celles contenues dans $\mathcal{V}$ à gauche du cycle.

Algorithme 7 Réduction des cycles redondants dans un mot circulant W

$\mathcal{V} \leftarrow W_{taille(W)}$
$pos \leftarrow taille(W)$
Tant que $pos > 2$ **Faire**
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{W_{pos}\}$
  $i \leftarrow pos - 1$ /* *Il faut rechercher un cycle* $\mathcal{C}(i, pos)$ */
  $\mathcal{V}_C \leftarrow \emptyset$
  Tant que $(i > 1) \wedge (W_i \neq W_{pos})$ **Faire**
    Si $W_i \notin \mathcal{V}$ **Alors**
      $\mathcal{V}_C \leftarrow \mathcal{V}_C \cup \{W_i\}$
    Fin Si
    $i \leftarrow i - 1$
  Fin Tant que
  Si $W_i = W_{pos}$ **Alors**
    $j \leftarrow i - 1$ /* *Cycle trouvé : recherche des éléments* $\mathcal{V}_C$ *à gauche de* i */
    Tant que $(j \geq 1) \wedge (\mathcal{V}_C \neq \emptyset)$ **Faire**
      $\mathcal{V}_C \leftarrow \mathcal{V}_C \backslash \{W_j\}$
      $j \leftarrow j - 1$
    Fin Tant que
    Si $\mathcal{V}_C = \emptyset$ **Alors**
      $supprimer(W, i + 1, pos)$ /* *Le cycle* $\mathcal{C}(i, pos)$ *est redondant* */
      $pos \leftarrow i$
    Sinon
      $pos \leftarrow pos - 1$
    Fin Si
  Sinon
    $pos \leftarrow pos - 1$
  Fin Si
Fin Tant que

Exemple 3.4 *Soit le mot circulant* $W =< 4, 3, 2, 5, 2, 4, 1, 5, 6, 4, 3 >$. *Nous remarquons que* W *possède un cycle constructeur* $\mathcal{C}_C(1, 11)$ *: ce cycle contient les 6 identités contenues dans* identités(W). *D'après la définition 3.4, nous pouvons supprimer la partie à droite (ici, seule l'identité 3 est supprimée). Enfin, le cycle* $\mathcal{C}(3, 5) =< 2, 5, 2 >$ *est redondant, l'identité 5 étant présente à droite de ce cycle. Il peut être réduit et le mot devient* $W =< 4, 3, 2, 4, 1, 5, 6, 4 >$

Lemme 3.1 (Taille maximum du cycle constructeur) *La taille du cycle constructeur est égale à* $\frac{n^2}{4} + n$ *dans le pire des cas.*

Démonstration Soit m l'identité qui apparaît la plus fréquemment dans le cycle constructeur et k le nombre d'occurrences de m dans le cycle constructeur. Il y a donc $k-1$ cycles qui ne sont pas redondants (sinon, ils auraient été supprimés selon la remarque 3.3) et donc $k-1$ identités qui apparaissent qu'une unique fois dans le cycle constructeur et $n-(k-1)$ identités qui peuvent apparaître au plus k fois (selon la définition de k). La taille maximale du mot T peut donc s'écrire en fonction de k :

$$T(k) = (k-1) + (n-(k-1)) * k = -k^2 + (n+2)k - 1$$

La fonction $T(k)$ est croissante lorsque $T'(k)$ est positive et admet un extremum lorsque $T'(k)$ s'annule, c'est à dire :

$$T'(k) > 0 \Leftrightarrow -2k + n + 2 \geq 0 \Leftrightarrow k \leq \frac{n+2}{2}$$

La taille du cycle constructeur admet donc un maximum en :

$$k = \frac{n+2}{2}$$

Donc :

$$\begin{aligned}
T_{max} &= \begin{cases} T(\frac{n+2}{2}) & \text{si } n \text{ pair} \\ max\{T(\frac{n+1}{2}), T(\frac{n+3}{2})\} & \text{si } n \text{ impair} \end{cases} \\
&= \begin{cases} -\left(\frac{n+2}{2}\right)^2 + (n+2) * \frac{n+2}{2} - 1 \\ max\left\{\left(-\left(\frac{n+1}{2}\right)^2 + (n+2)\left(\frac{n+1}{2}\right) - 1\right), \left(-\left(\frac{n+3}{2}\right)^2 + (n+2)\left(\frac{n+3}{2}\right) - 1\right)\right\} \end{cases} \\
&= \begin{cases} \frac{n^2+4n}{4} \\ max\left\{\left(\frac{n^2+4n-1}{4}\right), \left(\frac{n^2+4n-1}{4}\right)\right\} \end{cases} \\
&= \begin{cases} \frac{n^2+4n}{4} & \text{si } n \text{ pair} \\ \frac{n^2+4n-1}{4} & \text{si } n \text{ impair} \end{cases}
\end{aligned}$$

□

3.3 Algorithme principal

Nous distinguons trois phases dans l'algorithme principal. Tout d'abord, la *phase d'initialisation* consiste à ajouter les identités au mot jusqu'à obtenir le premier cycle constructeur, c'est-à-dire que le mot retourne sur le site qui a créé le jeton comme précisé par la remarque 3.2. La phase suivante est la *phase de maintenance* pendant laquelle un cycle constructeur est maintenu dans le mot et enfin, la *phase de collecte* lorsque le cycle constructeur est détruit suite à la découverte d'une nouvelle identité, par exemple.

3.3.1 Phase d'initialisation

Lorsque le mot est créé, il ne contient que l'identité du nœud qui l'a créé. Le but de la phase d'initialisation est de faire circuler le mot jusqu'à obtenir le premier cycle constructeur.

Dans ce cas, l'algorithme passe en phase de maintenance. La taille du mot circulant peut cependant grandir assez rapidement durant la phase d'initialisation. Aussi, la réduction des cycles redondants est appliquée à chaque déplacement du mot. Si le mot contient un cycle redondant $\mathcal{C}_R(i,j)$, alors la partie $< W_{i+1}, \ldots, W_j >$ peut être supprimée.

Remarque 3.4 *La suppression des cycles redondants durant la phase d'initialisation réduit la taille du mot mais peut entraîner la perte d'informations constructives. Par exemple, le mot* $< 3,4,1,3,1,2,1 >$ *contient un cycle redondant* $< 1,3,1 >$*. S'il est supprimé, il n'est plus possible de construire un chemin entre 3 et 4. Cependant, si cette réduction peut supprimer des informations topologiques, l'ensemble* identités(W) *n'est pas modifié et un cycle constructeur finit par apparaître (lorsque la première identité est rencontrée).*

Remarque 3.5 *L'exécution de l'algorithme 7 peut être coûteuse mais n'est pas nécessaire à chaque déplacement du jeton. En fait, il est possible de l'exécuter uniquement lorsque l'algorithme entre en phase de maintenance ou périodiquement.*

La taille du mot dans la phase d'initialisation est bornée par $2n-1$. En effet, tout comme dans le mot non-orienté, il ne peut y avoir d'informations non constructives, les cycles redondants étant supprimés au fur et à mesure. Pour la durée de cette phase, elle est égale au temps de retour sur le premier nœud qui a généré le mot. Dans la section 1.4, nous avons vu que le temps de retour d'une marche aléatoire est en $O(n^3)$.

3.3.2 Phase de maintenance

Lorsque l'algorithme passe dans cette phase, il existe un cycle constructeur $\mathcal{C}_C(1, taille(W))$ si nous tenons compte de la réduction des informations à sa droite. Le but de la phase de maintenance est de maintenir le cycle constructeur et d'ajouter les identités visitées à gauche de celui-ci (cette partie est appelée la *tête*) jusqu'à ce qu'un nouveau cycle constructeur apparaisse. Ainsi, le cycle constructeur contenu dans le mot évolue en fonction des visites du jeton. Lorsque i identités ont été ajoutées, le cycle constructeur devient $\mathcal{C}_C(i+1, taille(W))$. La structure du mot est donc la suivante :

$$< \underbrace{W_1, \ldots, W_i}_{tête}, \underbrace{W_{i+1}, \ldots, W_{taille(W)}}_{\mathcal{C}_C(i+1, taille(W))} >$$

Lorsqu'une nouvelle identité est ajoutée, il y a deux cas à considérer suivant si l'identité est déjà dans le mot ou non.

L'identité n'est pas dans le mot : si une nouvelle identité apparaît au début du mot, le cycle constructeur est cassé puisqu'il ne contient pas cette identité. Dans ce cas, l'algorithme entre dans la phase de collecte.

L'identité est déjà dans le mot : lorsqu'une identité est ajoutée dans le mot, nous devons rechercher si un cycle redondant peut être supprimé dans la tête du mot.

Remarque 3.6 *La recherche des cycles redondants ne doit pas être réalisée en dehors de la tête. En effet, cela pourrait induire des pertes d'informations. Par exemple, le mot* <4, 2, 1, 5, 2, 4, 2, 3, 1> *contient le cycle redondant* $< 2, 4, 2 >$ *qui ne doit pas être réduit. Dans le cas contraire, l'identité 4 disparaîtrait du cycle constructeur.*

Propriété 3.1 *Si un mot contient un cycle constructeur* $\mathcal{C}_C(i+1, taille(W))$*, alors tout cycle* $\mathcal{C}(j,k)$ *avec* $j < k < i+1$ *est redondant.*

En effet, un tel cycle contient des identités qui sont forcément incluses dans le cycle constructeur. Cette propriété accélère considérablement la recherche des cycles redondants. De plus, s'ils sont réduits au fur et à mesure, le seul cycle redondant possible est le cycle $\mathcal{C}(1,j)$ avec $j \leq i+1$.

Remarque 3.7 *Si un cycle redondant est détecté dans la tête, alors il ne peut y avoir de nouveau cycle constructeur.*

Lorsqu'aucun cycle redondant n'est détecté dans la tête du mot, nous devons rechercher la présence éventuelle d'un nouveau cycle constructeur.

Propriété 3.2 *Si un mot contient un cycle constructeur* $\mathcal{C}_C(i+1, taille(W))$ *alors la position minimale se trouve dans ce cycle.*

Démonstration Supposons que pos_{min} est dans l'intervalle $[1,i]$ alors il existe une occurrence de chaque identité de *identités(W)* dans $[1, pos_{min}]$ et dans $[i+1, taille(W)]$. En particulier, il existe une position j dans $[i+1, taille(W)]$ telle que $W_j = W_1$. Alors $\mathcal{C}(1,j)$ est un cycle constructeur, ce qui est impossible en considérant la réduction décrite dans la section 3.2.2. Donc $W_{pos_{min}}$ est dans $droite(W, i+1)$.

□

Propriété 3.3 *Supposons qu'un mot contienne un cycle constructeur* $\mathcal{C}_C(i+1, taille(W))$*. Si un nouvel élément est ajouté au mot, alors si un nouveau cycle constructeur apparaît, sa borne droite est obligatoirement dans l'intervalle* $[pos_{min}, taille(W)]$.

Démonstration Supposons j une position telle que $j < pos_{min}$ et que nous ayons un nouveau cycle constructeur $\mathcal{C}_C(1,j)$. Alors l'élément $W_{pos_{min}}$ devrait aussi apparaître dans $gauche(W,j)$. Or, comme le précise la remarque 3.1, c'est impossible. Donc $j \leq pos_{min}$.

□

La recherche de W_j, la borne droite de l'éventuel nouveau cycle constructeur, est limitée dans le sous-mot $droite(W, pos_{min})$. L'algorithme 8 résume les différentes étapes de la phase de maintenance.

D'après le lemme 3.1, la taille du cycle constructeur est bornée par $\frac{n^2}{4} + n$. D'autre part, la taille de la tête est bornée par $n-1$, cette partie du mot ne contenant aucun cycle. La complexité de l'algorithme pour la phase de maintenance dépend de la découverte d'un nouveau cycle constructeur. Dans le cas où aucun cycle constructeur n'est trouvé, la recherche est limitée au parcours de la tête et du sous-mot $droite(W, pos_{min})$. Par contre, en cas de succès, la complexité est augmentée suivant l'algorithme 7.

Algorithme 8 Ajout d'une nouvelle identité dans un mot circulant W contenant un cycle constructeur $\mathcal{C}_C(i+1, taille(W))$

```
/* Le mot possède un cycle constructeur C_C(i+1, taille(W)) */
/* Recherche d'un cycle redondant dans la tête du mot */
pos ← 2
Tant que (W_1 ≠ W_pos) ∧ (pos < i+1) Faire
  pos ← pos + 1
Fin Tant que
Si W_1 = W_pos Alors
  /* Cycle redondant trouvé */
  supprimer(W, 1, pos − 1)
Sinon
  /* Recherche d'un nouveau cycle constructeur */
  pos ← pos_min
  Tant que (W_1 ≠ W_pos) ∧ (pos < taille(W) Faire
    pos ← pos + 1
  Fin Tant que
  Si W_1 = W_pos Alors
    /* Nouveau cycle constructeur trouvé */
    supprimer(W, pos + 1, taille(W)
    Recherche des cycles redondants
    Calcul de pos_min
    /* C_C(i+1, taille(W) devient C_C(1, W_pos) */
  Fin Si
Fin Si
```

3.3.3 Phase de collecte

Comme dit précédemment, lorsque l'algorithme est en phase de maintenance et qu'il rencontre une nouvelle identité, il passe dans la phase de collecte. À cet instant, il n'existe plus de cycle constructeur (au moins une identité n'existe pas à l'intérieur du cycle constructeur). Comme la phase d'initialisation est coûteuse en terme de réduction des cycles redondants, la phase de collecte se contente de maintenir un cycle maximal (l'ancien cycle constructeur de la phase de maintenance) afin d'accélérer la construction d'un nouveau cycle constructeur.

Pour cela, lorsqu'une identité est ajoutée dans la tête du mot, nous vérifions si elle existe dans le cycle maximal. Si c'est le cas, le mot est de la forme suivante :

$$< W_1, \ldots, W_i, \ldots, W_j, \ldots, W_{taille(W)} >, \text{ avec } W_1 = W_j \text{ et } W_i = W_{taille(W)}$$

Il est alors possible de le récrire sous la forme suivante :

$$< W_1, \ldots, W_i, \ldots, W_j, \ldots, W_{taille(W)}, \ldots, W_j >, \text{ avec } W_1 = W_j \text{ et } W_i = W_{taille(W)}$$

Nous obtenons alors un nouveau cycle constructeur et l'algorithme peut retourner dans la phase de maintenance. Le nouveau cycle constructeur peut cependant contenir des cycles redondants et il est nécessaire de lui appliquer l'algorithme 7.

3.3.4 Algorithme général

Nous avons défini un jeton comme étant un mot circulant, c'est-à-dire uniquement une liste d'identités. La position minimale, la borne gauche du cycle constructeur ainsi que la phase dans laquelle se trouve l'algorithme ne sont pas connues par le nœud recevant le jeton. L'algorithme 9 doit être exécuté à chaque réception du mot. Tout d'abord, nous calculons la position minimale : si elle se trouve en dernière position, l'algorithme est en phase d'initialisation. Pour différencier les deux autres phases, il faut s'intéresser au cycle maximal. On recherche une occurrence de la dernière identité du mot (*i.e.* $W_{taille(W)}$) à partir du début du mot. Pour déterminer si ce cycle est constructeur, il faut rechercher s'il contient la première identité du mot. Dans ce cas, l'algorithme est dans sa phase de maintenance.

Algorithme 9 Algorithme général à la réception d'un mot circulant W sur un nœud i

```
ajouter(W, i)
Calcul de pos_min (algorithme 6)
Si pos_min ≠ taille(W) Alors
   Recherche de W_j à partir de W_1 tel que W_j = W_taille(W)
   Recherche de W_k à partir de W_j tel que W_k = W_1
   Si W_k n'existe pas Alors
      Phase de collecte
   Sinon
      Phase de maintenance
   Fin Si
Sinon
   Phase d'initialisation
Fin Si
Envoyer le jeton à j choisi uniformément au hasard dans Vois_i
```

Pour éviter tous ces calculs, il suffit d'ajouter à l'intérieur du jeton la position minimale, la position de la borne gauche du cycle constructeur et enfin, la phase courante. Cependant, si nous sommes dans un environnement où le contenu des messages peut être corrompu, l'algorithme 9 est essentiel.

3.4 Gestion des fautes

Les différentes pannes pouvant subvenir dans une grille peuvent être regroupées en deux catégories : les changements topologiques (si un nœud se connecte ou se déconnecte) et les erreurs de communication (perte de messages). Ces deux types de panne ont des conséquences sur le mot circulant.

3.4.1 Changements topologiques dans la grille

D'après notre modèle, chaque nœud i gère son voisinage noté $Vois_i$. Ainsi, si un nœud voisin se déconnecte de la grille, $Vois_i$ est mis à jour. Cependant, les informations topologiques globales sont toujours incluses dans le mot circulant. Aussi, lorsque le jeton arrive sur le nœud

i, celui-ci doit vérifier la cohérence des informations topologiques contenues dans le mot en le comparant avec son voisinage.

Ainsi si le jeton possède un lien (i, j) et que j n'existe pas dans $Vois_i$, alors le nœud i doit corriger cette erreur. Pour cela, il doit supprimer la partie $< W_{k+1}, \ldots, W_l, W_m >$ avec $W_l = j$, $W_m = i$ et $W_k \in N_i$. Par exemple, si le nœud d'identité 4 a un voisinage $Vois_4 = \{1, 2\}$ et que le mot est $< 1, 2, 3, 4, 2, 4 >$, alors le mot est réduit par $< 1, 2, 3, 2, 4 >$. L'algorithme 10 décrit cette réduction.

Algorithme 10 Réduction des incohérences topologiques dans un mot circulant W

$pos \leftarrow taille(W)$
Tant que $pos > 1$ **Faire**
  Si $W_{pos} = i$ **Alors**
    $gauche \leftarrow pos - 1$
    Tant que $(gauche > 1) \wedge (W_{gauche} \neq i) \wedge (W_{gauche} \notin Vois_i)$ **Faire**
      $gauche \leftarrow gauche - 1$
    Fin Tant que
    Si $((W_{gauche} = i) \vee (W_{gauche} \in Vois_i)) \wedge (gauche + 1 \neq pos)$ **Alors**
      $supprimer(W, gauche + 1, pos - 1)$
    Sinon
      Si $gauche = 1$ **Alors**
        $supprimer(W, 1, pos - 1)$
      Fin Si
    Fin Si
    $pos \leftarrow gauche$
  Sinon
    $pos \leftarrow pos - 1$
  Fin Si
Fin Tant que

Remarque 3.8 *L'algorithme 10 corrige les incohérences contenues dans le mot à partir des informations topologiques locales d'un nœud. Après son exécution, il est possible que des erreurs subsistent encore dans le mot. La circulation du jeton assure qu'elles seront corrigées en un temps fini, c'est-à-dire en un nombre fini d'étapes.*

3.4.2 Pannes de communication

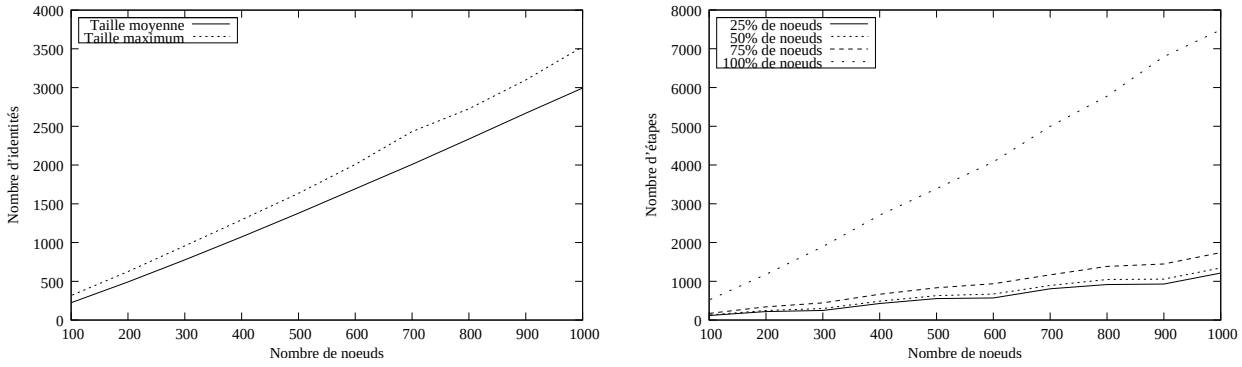
Une panne de communication peut avoir plusieurs conséquences. Tout d'abord, le mot peut être modifié si nous considérons que les communications ne sont pas fiables. Cela se traduit par des informations erronées ou supprimées à l'intérieur du mot. Or, lors de la réception du jeton, l'algorithme 10 est appliqué sur le contenu du mot. Les incohérences produites par les erreurs de communication sont alors vues comme des changements topologiques et sont corrigées sans mécanisme supplémentaire.

Notre solution étant basée sur la circulation d'un jeton, il est essentiel qu'il ne soit pas détruit. Pour cela, nous plaçons un compte-à-rebours sur chaque nœud de la grille. Ce mécanisme est décrit dans la section 1.5.2. Il peut impliquer la duplication du jeton et nous avons vu qu'il

était possible de fusionner les informations de plusieurs mots circulants. Dans un cadre orienté, la procédure est identique. Cependant, l'insertion d'informations ne doit pas ajouter de liens qui n'existent pas dans le graphe de communication original. Pour limiter la complexité de la fusion, nous posons comme condition qu'au moins un des deux jetons possède un cycle constructeur. Dans ce cas, nous comparons les différentes informations contenues dans les deux mots et nous les fusionnons en insérant le cycle constructeur dans le second mot. Si aucun jeton ne possède de cycle constructeur, ils continuent leur marche. Cette méthode permet de réduire la complexité de la fusion mais peut accélérer la reconstruction d'un mot complet.

3.5 Mot circulant orienté : simulations

Nous avons réalisé des simulations afin d'observer le comportement du mot circulant orienté. Tout d'abord, nous avons observé la taille moyenne et maximum du mot circulant en fonction du nombre de nœuds dans le réseau. Les résultats de simulations sur des réseaux aléatoires est présentée sur la figure 3.6 (a). Tout comme le mot circulant non-orienté, sa taille augmente proportionnellement au nombre de nœuds. Elle reste cependant largement inférieure à la borne maximum. Pour des réseaux de 1000 nœuds, sa taille ne dépasse pas 3500 identités. La figure 3.6 (b) présente le temps nécessaire pour obtenir des taux de couverture de 25% à 100% dans le cycle constructeur du mot circulant. Tout comme nous avions vu dans la section 2.4, un taux de couverture de 75% est rapidement atteint (un peu plus de 1000 étapes pour un réseau de 1000 nœuds). La couverture totale est cependant beaucoup plus longue (près de 7500 étapes).

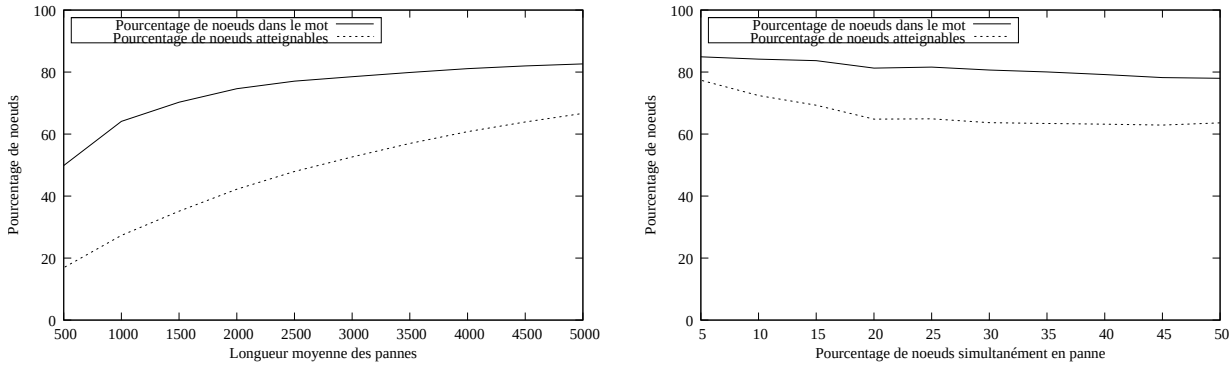


(a) (b)

FIG. 3.6 – Taille moyenne et maximum du mot circulant en fonction du nombre de nœuds dans des graphes aléatoires (a) et taux de couverture en fonction du nombre de nœuds (b).

La gestion du mot circulant permet de corriger les incohérences topologiques. Les deux courbes de la figure 3.7 présentent le taux de couverture du mot circulant, c'est-à-dire le nombre de nœuds qu'il est possible d'atteindre à l'aide des informations contenues dans le mot circulant. Les simulations ont été réalisées sur des graphes aléatoires de 100 nœuds sur lesquels nous appliquons un modèle de panne fixe. Nous faisons varier la durée moyenne des pannes (figure (a)) et le nombre de nœuds simultanément en panne (figure (b)). Là encore, le comportement du mot circulant orienté est sensiblement identique à celui du mot circulant non-orienté. Même avec un taux de panne important (près de 50% de nœuds en panne), le taux de couverture reste

important (presque 80% avec des pannes de longueur 5000). La durée moyenne des pannes a une grande influence sur le taux de couverture. Les incohérences topologiques étant corrigées localement, il est nécessaire que le mot visite les nœuds concernés. La vitesse de correction dépend du nombre de nœuds dans le réseau, mais aussi de la topologie. Plus le réseau est dense, plus vite les incohérences sont corrigées.



(*a*) (*b*)

FIG. 3.7 – Taux de couverture du mot circulant sur des graphes aléatoires de 100 nœuds en fonction de la longueur moyenne des pannes (*a*) et en fonction du nombre de nœuds simultanément en panne (*b*).

3.6 Applications du mot circulant orienté

Dans un premier temps, nous proposons dans cette section une solution qui exploite le contenu du mot circulant pour rechercher une ressource dans le réseau. Pour cela, des arbres couvrants sont construits à partir des informations topologiques afin d'atteindre tous les nœuds, quelle que soit l'orientation des liens de communication. Dans un deuxième temps, nous proposons une solution pour construire un réseau de recouvrement pour les réseaux pair-à-pair. Cette solution prend en compte l'orientation des liens et maximise ainsi le nombre de pairs connectés.

3.6.1 Recherche de ressources dans une grille

À partir des informations topologiques inclues dans le mot, il est possible de construire des structures couvrantes de la grille. En particulier, si le mot possède un cycle constructeur, il est possible de construire un arbre couvrant enraciné en chaque nœud du graphe. Cette construction est décrite dans l'algorithme 11.

Algorithme 11 Construction d'un arbre couvrant $\mathcal{A}$ enraciné sur le nœud r à partir du mot circulant W

```
A ← r
i ← 1
pos ← 1
Tant que i < taille(W) Faire
  /* Recherche d'un nœud déjà dans l'arbre */
  Tant que (i < taille(W)) ∧ (W_i ∉ A) Faire
    i ← i + 1
  Fin Tant que
  Si W_i ∈ A Alors
    /* Ajout de la nouvelle branche enracinée en i */
    j ← i − 1
    Tant que j ≥ pos Faire
      Si W_j ∉ A Alors
        ajouter(A, W_j, W_{j+1})
      Fin Si
      j ← j − 1
    Fin Tant que
  Fin Si
  i ← i + 1
  pos ← i
Fin Tant que
```

Pour localiser une ressource dans la grille (les résultats d'un calcul, une application, un fichier), il est possible d'utiliser le mécanisme de propagation avec remontée d'informations proposé dans [Seg83]. La diffusion est amorcée le long d'un arbre couvrant et les nœuds envoient leur réponse dans ce même arbre. Or, l'algorithme 11 permet de construire uniquement un arbre de diffusion et non pas de retour, les communications étant orientées. Nous avons donc besoin de deux arbres différents : un arbre de propagation $\mathcal{A}_P$ qui est orienté du nœud émettant la requête vers les nœuds de la grille et d'un arbre de retour $\mathcal{A}_R$ orienté des nœuds de la grille vers le nœud émetteur. L'arbre $\mathcal{A}_P$ est construit grâce à l'algorithme 11. Pour l'arbre de remontée, nous remarquons que la première identité est toujours celle du nœud émetteur. Aussi, il suffit d'ajouter toutes les identités du mot comme étant fille de l'identité de gauche. L'algorithme 12 montre cette construction.

Algorithme 12 Construction d'un arbre de retour $\mathcal{A}_R$ à partir du mot circulant W

```
A_R ← W_1
k ← 2
Tant que identités(W) ≠ identités(A_R) Faire
  Si W_k ∉ A_R Alors
    ajouter(A_R, W_k, W_{k−1})
  Fin Si
  k ← k + 1
Fin Tant que
```

Exemple 3.5 *La figure 3.8 montre un exemple de requête sur une ressource quelconque (ici, un fichier). Le nœud 0 désire accéder au fichier $F1$ qui est disséminé dans le réseau. Lorsqu'il reçoit le mot circulant (ou à partir de l'image du dernier mot reçu), il peut construire les arbres $\mathcal{A}_P$ et $\mathcal{A}_R$. Il envoie alors sa requête qui est diffusée le long de $\mathcal{A}_P$. Les nœud 2 et 4, qui possèdent tous les deux une partie du fichier, envoient une réponse au nœud 0 le long de l'arbre $\mathcal{A}_R$. Le nœud 0 peut ainsi localiser chaque partie de $F1$ et éventuellement le reconstituer.*

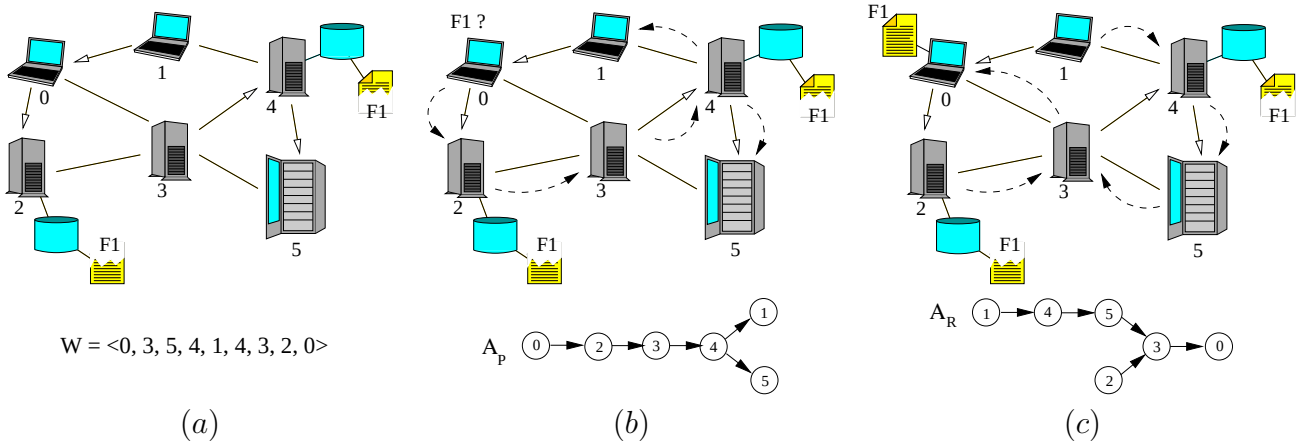


FIG. 3.8 – Le nœud 0 a besoin du fichier $F1$. À partir du mot circulant (a), il construit $\mathcal{A}_P$ et$\mathcal{A}_R$. Il transfert sa requête le long de $\mathcal{A}_P$ (b) et la réponse est retournée le long de $\mathcal{A}_R$ (c) : le fichier est localisé sur les nœuds 4 et 2.

Dans cet exemple, nous montrons que nous pouvons localiser une ressource à l'aide du contenu du mot circulant (ici, un fichier) et que nous pouvons aussi transférer cette ressource dans la grille.

3.6.2 Construction d'un réseau recouvrant

Dans un réseau pair-à-pair, la construction d'une surcouche au réseau est nécessaire : elle évite l'établissement de nouvelles connexions entre les pairs à chaque envoi de données. Comme le nombre de nœuds peut être relativement important, il n'est pas possible d'établir des connexions entre chaque paire de nœuds. Une stratégie de sélection doit être mise en place. Nous avons déjà présenté le protocole *Gnutella* (voir section 1.2.4). Lorsqu'un nœud se connecte au réseau, il contacte un serveur qui lui retourne un ensemble de pairs choisis aléatoirement parmi les pairs connectés. Des connexions sont établies entre ces pairs et le nouveau nœud qui deviennent ses voisins dans le réseau recouvrant. Les messages de contrôle sont ensuite envoyés au travers ces connexions.

Nous proposons ici d'utiliser le mot circulant orienté pour construire un réseau recouvrant. Afin de garder une décentralisation totale, nous proposons un mode de connexion des nouveaux nœuds similaire à *CONFIIT* : lorsqu'un nouveau nœud désire se connecter, il contacte l'un des nœuds déjà présent dans le réseau. Une liste d'adresses IP est supposée connue afin de contacter d'autres nœuds dans le cas où le premier ne réponde pas.

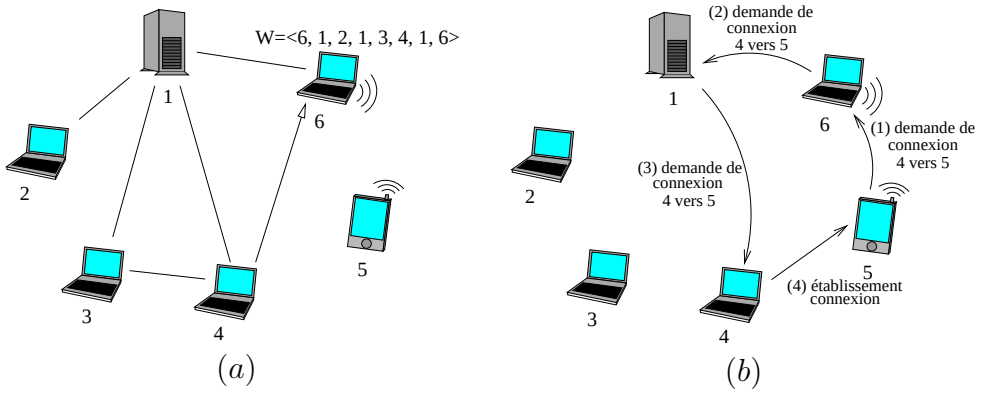


FIG. 3.9 – Exemple de connexion d'un nœud à un réseau pair-à-pair. Le nœud 5 choisit comme point d'entrée le nœud 6 qui lui envoie sa trace du mot circulant (a). La connexion entre le nœud 4 et le nœud 5 doit être établie à l'initiative du nœud 4 qui est contacté à l'aide du contenu du mot circulant (b).

Nous supposons qu'un mot circulant orienté se déplace infiniment dans le réseau et que les nœuds gardent une trace du dernier mot reçu. Lorsqu'un nœud se connecte à un point d'entrée, celui-ci lui envoie cette trace. À partir des identités contenues dans le mot, le nouveau nœud en sélectionne un sous-ensemble choisi aléatoirement : nous notons cet ensemble $Vois_i^5$ (il s'agit du voisinage correspondant à la couche 5 du modèle théorique). Pour chaque nœud de cet ensemble, des connexions doivent être établies. Si un nœud donné ne peut être contacté directement, une requête est envoyée dans le réseau à l'aide du mot circulant.

Exemple 3.6 *La figure 3.9 montre un exemple de connexion du nœud 5 au réseau. Le point d'entrée est le nœud 6. Le nœud 5 le contacte et reçoit le mot circulant* $W =< 6, 1, 2, 1, 3, 4, 1, 6 >$. *Nous supposons ici que le nombre de voisins dans le réseau de recouvrement est de 3. Le nœud 5 choisit pour voisinage* $\text{Vois}_5^5 = \{2, 4, 6\}$. *Pour l'établissement des connexions vers les nœuds 2 et 6, il n'y a pas de problème. Par contre, le nœud 4 ne peut être contacté directement. Le nœud 5 envoie une demande de connexion à ce nœud en exploitant le contenu du mot circulant. Lorsque la demande est reçue, le nœud 4 établit la connexion avec le nœud 5.*

Le mot circulant permet de maintenir la connaissance du réseau avec l'orientation des liens. Ainsi, en cas de déconnexion des nœuds, il peut être réutilisé pour la découverte de nouveaux voisins. La construction du réseau recouvrant peut être utilisé par d'autres composants au niveau de la couche 5. Comme il est non-orienté grâce à l'établissement des connexions, il peut facilement être exploité pour communiquer des informations entre les pairs. Pour le transfert de fichiers entre deux nœuds, il est cependant nécessaire d'établir une nouvelle connexion : le transfert peut surcharger le réseau de recouvrement. La solution proposée dans la section précédente peut être utilisée pour la recherche du fichier et la connexion est établie entre le demandeur et le possesseur du fichier suivant l'orientation des liens.

3.7 Conclusion

Nous avons proposé dans ce chapitre une adaptation du mot circulant aux réseaux orientés. Nous avons présenté un algorithme constitué de 3 phases différentes pour gérer son contenu.

Afin de réduire sa complexité, il se base sur la notion de cycle constructeur qui permet de construire un arbre couvrant enraciné en chaque nœud. En particulier, la taille du mot peut être réduite tout en tenant compte de l'orientation des liens.

À partir de simulations, nous avons observé le comportement de cette nouvelle gestion du mot circulant dans des réseaux aléatoires. La taille du mot est supérieure à celle du mot circulant non-orienté. En effet, la quantité d'information à maintenir est supérieure à cause de l'orientation des liens. Lorsque le nombre de nœuds augmente dans le réseau, la taille du mot augmente elle-aussi mais reste très inférieure à la borne maximale. Le taux de couverture du mot est quant à lui important quel que soit le nombre de pannes dans le système.

En exploitant les identités collectées dans le mot circulant, nous avons vu qu'il est possible de construire pour tout nœud un arbre couvrant malgré l'orientation des liens de communication. À partir de cet arbre, un nœud peut envoyer des requêtes et recevoir les réponses appropriées. Cette recherche de ressources est adaptée au modèle théorique que nous avons présenté dans le chapitre 2.2, ce qui signifie qu'elle prend en compte les communications orientées dues aux limitations des communications. De même, nous avons proposé une solution pour construire un réseau recouvrant dans le cadre des réseaux pair-à-pair en exploitant les informations contenues dans le mot.

Chapitre 4

Gestion décentralisée de tâches

Résumé : *Nous proposons dans ce chapitre une solution pour la gestion des tâches indépendantes, irrégulières et indivisibles adaptée aux réseaux dynamiques (couche 5 de notre modèle). Elle est basée sur la circulation d'une marche aléatoire et est complètement distribuée. En effet, l'assignation des tâches est laissée à la charge des nœuds. Cette politique locale évite le recours à un nœud dédié, ce qui permet une plus grande tolérance aux pannes. Nous proposons deux méthodes d'assignation que nous avons appelées méthode passive et méthode active. Après les avoir détaillées, nous comparons leur efficacité en fonction de différents paramètres physiques de la grille (nombre de nœuds, réseau) et des caractéristiques des tâches (longueur, nombre, irrégularité). Afin d'améliorer ces deux méthodes, nous proposons aussi une solution basée sur plusieurs jetons et une solution hybride entre ces deux méthodes qui consiste à passer de l'une à l'autre au cours de l'exécution. Nous proposons une autre solution basée sur l'utilisation du mot circulant qui consiste à diffuser l'état des tâches le long d'un arbre. Ces travaux ont fait l'objet d'une publication en 2007 [BFR07].*

4.1 Gestion des tâches à l'aide d'une marche aléatoire

La gestion des tâches est un composant essentiel dans les applications de calcul distribué. Elle est mise en relation avec la gestion des ressources et se trouve dans la couche 5 de notre modèle théorique. Lorsqu'une tâche est soumise dans la grille, elle doit être assignée à un nœud pour que son calcul puisse débuter. Cette assignation doit tout d'abord prendre en compte les considérations locales au nœud : les ressources physiques disponibles sur le nœud pour le calcul de la tâche (puissance de calcul, mémoire, nombre de processeurs) mais aussi les ressources logiques nécessaires (fichiers, applications ou bibliothèques). De plus, du point de vue de la grille, l'assignation des tâches doit permettre d'équilibrer la charge sur l'ensemble des nœuds pour exploiter équitablement les ressources.

Un ordonnancement des tâches a pour but de déterminer sur quels nœuds les tâches doivent être assignées. Nous distinguons deux types d'ordonnancement principaux : l'ordonnancement statique et l'ordonnancement dynamique. Dans le premier cas, le temps d'exécution de chaque tâche doit être connu lors de la soumission. À partir des ressources disponibles, un ordonnancement optimal est calculé afin de minimiser le temps d'exécution global ainsi que l'utilisation des ressources. Ce type d'ordonnancement est particulièrement adapté lorsque les tâches cor-

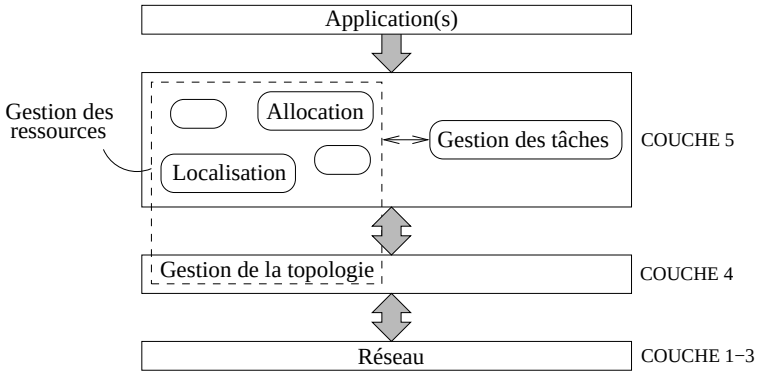


FIG. 4.1 – La gestion des tâches plongée dans le modèle théorique.

respondent à l'exécution d'une application dont le nombre d'opérations est connu ou si l'ensemble des ressources est dédié exclusivement à l'application. Dans les grappes de serveurs, cette méthode est particulièrement adaptée : les nœuds ont tous la même puissance et le réseau d'interconnexion est local.

Dans le cadre du vol de cycles, les applications ont pour but d'exploiter les ressources inutilisées d'une entreprise ou des particuliers (comme *SETI@home*). Dans ce cas, les ressources sont partagées avec d'autres applications. S'il est possible de connaître à un instant précis la charge des ordinateurs, son évolution est imprévisible car elle dépend de nombreux paramètres. De plus, la plupart des intergiciels actuels autorisent l'exécution de programmes qui sont fournis par l'émetteur de la tâche. Dans ce cas, le temps d'exécution n'est pas connu et des mécanismes supplémentaires doivent être mis en œuvre pour analyser ce programme. Pour toutes ces raisons, un ordonnancement dynamique est nécessaire. Les tâches sont attribuées au fur et à mesure sur les nœuds en fonction de leur disponibilité ou bien elles peuvent être déplacées d'un nœud à l'autre si la puissance de calcul devient insuffisante.

Dans les deux types d'ordonnancement (statique et dynamique), l'assignation des tâches est réalisée en fonction des ressources des nœuds. Les auteurs de *Nimrod* [BAG00] proposent de concentrer sur un serveur l'ensemble des informations sur les ressources. L'ordonnancement peut ensuite être réalisé, toutes les informations nécessaires étant connues en local. Dans *NetSolve*, les serveurs de calcul distants enregistrent sur un agent central, les différentes fonctions qu'ils sont en mesure de calculer et pour chacune, une évaluation de la complexité est précisée. Cette évaluation permet ainsi de prédire le temps de calcul des tâches et donc la charge des serveurs. Dans le cas de *Condor-G* [FTL+02], les auteurs proposent une solution basée sur *Globus*. Un ou plusieurs agents sont déployés sur des serveurs qui gèrent l'ordonnancement des tâches.

Toutes ces solutions nécessitent une capacité de traitement importante et une certaine robustesse des serveurs : s'ils tombent en panne ou ne peuvent plus répondre à cause d'une surcharge, l'assignation des tâches ne peut être réalisée. Une autre stratégie est de laisser les nœuds se charger de l'assignation. Lorsqu'ils terminent le calcul d'une tâche, ils décident par eux-même d'en calculer une nouvelle. Dans le cadre de *SETI@home*, ils interrogent un serveur via Internet pour obtenir les nouvelles données à analyser. Les auteurs de *CONFIIT* proposent quant à eux une approche pair-à-pair complètement distribuée : les paramètres des tâches sont diffusés à tous les nœuds pour éviter toute centralisation. Cette politique locale permet de répartir la charge

du service d'assignation sur tous les nœuds de la grille qui n'ont besoin que de la connaissance de leurs propres caractéristiques physiques. Aucun serveur dédié n'est nécessaire, ce qui est un avantage dans les réseaux dynamiques, tant au point de vue de la tolérance aux pannes qu'au passage à l'échelle de l'application. Cependant, cette approche pair-à-pair implique la mise en place de mécanismes pour être en mesure de synchroniser tous les nœuds. Dans *CONFIIT*, les nœuds sont placés dans une structure virtuelle dans laquelle un jeton circule pour mettre à jour l'état des tâches.

Nous proposons dans ce chapitre une solution pour la gestion de tâches indépendantes, irrégulières et indivisibles dédiée aux réseaux dynamiques. Elle est basée sur l'utilisation d'une marche aléatoire et elle est complètement distribuée. L'assignation des tâches est laissée à la charge des nœuds à l'aide d'une politique locale. Comme nous avons vu dans les chapitres précédents, une solution basée sur les marches aléatoires est peu coûteuse en terme de messages et est tolérante aux pannes. De plus, contrairement à *CONFIIT*, aucune structure virtuelle n'est maintenue, ce qui limite le nombre de messages de contrôle.

Dans la prochaine section, nous présentons deux méthodes d'assignation de tâches que nous avons appelées méthodes *passive* et *active*. Nous proposons dans la section 4.3 une comparaison entre ces deux méthodes en fonction de différents paramètres tels que le nombre de nœuds dans la grille ou les propriétés des tâches. Dans la section 4.4, nous proposons d'optimiser cette gestion des tâches à l'aide de plusieurs mécanismes. Nous proposons une méthode à base de plusieurs marches concurrentes, puis une méthode hybride entre les méthodes passive et active. En utilisant le mot circulant que nous avons introduit dans la section 1.5, nous montrons qu'il est possible d'utiliser un mécanisme basé sur la diffusion des états des tâches le long d'un arbre.

4.2 Méthodes d'assignation de tâches

Dans ce chapitre, nous introduisons les deux méthodes d'assignation que nous avons appelées passive et active. Nous présentons tout d'abord, les données nécessaires contenues sur les nœuds et dans le jeton et nous donnons différentes définitions relatives à la gestion des tâches. Nous présentons ensuite l'algorithme général de chaque méthode et pour terminer, nous détaillons les différents mécanismes communs relatifs à la gestion des pannes.

4.2.1 Quelques définitions

Tout calcul qui est soumis dans une grille peut être vu comme un ensemble de tâches. Dans le cas d'un calcul séquentiel, cet ensemble peut être éventuellement composé que d'une seule tâche.

Définition 4.1 (Tâche) *Une tâche est une entité élémentaire qui est caractérisée par une date de début et de fin dont la réalisation nécessite une durée. Une tâche est obligatoirement calculée sur un seul nœud.*

Nous représentons chaque tâche par un tuple $\{idT, idE, p, e, r\}$ où idT est un identifiant unique attribué à chaque tâche à la soumission. idE est l'identité de l'émetteur de la tâche. Les paramètres d'une tâche notés p sont les informations nécessaires pour son calcul. Il peut

s'agir du nom de l'application avec ses paramètres d'entrée ou du nom de la bibliothèque avec le nom de la fonction à exécuter et ses paramètres. Cependant, l'application ou la bibliothèque peuvent être sur un nœud particulier. Dans ce cas, p contient l'adresse de ce nœud. À tout moment, une tâche est caractérisée par un état e qui correspond à l'état d'avancement de son calcul. Cet état prend la valeur *non calculée* pour une tâche non calculée (état par défaut), *en cours* lorsqu'elle est en cours de calcul sur un ou plusieurs nœuds et *calculée* si elle est terminée. Lorsqu'une tâche est calculée, les identités des nœuds qui possèdent les résultats sont placées dans l'ensemble r. Ainsi, l'émetteur idE est en mesure de récupérer les résultats en contactant ces nœuds.

Dans notre cas, une tâche peut posséder des états différents suivant les nœuds de la grille. Le but de notre gestion des tâches est donc de diffuser et de maintenir les états des tâches sur les nœuds afin d'assurer une cohérence globale. Chaque nœud i possède un ensemble $\mathcal{E}_i$ qui contient toutes les tâches. Cet ensemble est mis à jour à l'aide de la circulation du jeton dans le réseau noté $J = \{\mathcal{E}_J\}$. L'ensemble des tâches $\mathcal{E}_J$ permet de mettre à jour celui des nœuds. Lorsqu'un nœud reçoit le jeton, il exécute l'algorithme 13 afin de mettre à jour son ensemble ainsi que celui du jeton.

Algorithme 13 Comparaison entre un ensemble de tâches local à un nœud $\mathcal{E}_i$ et celui du jeton $\mathcal{E}_J$.

Pour tout $(idT_i, idE_i, p_i, e_i, r_i) \in \mathcal{E}_i$ **Faire**
  Si $idT_i \notin \mathcal{E}_J$ **Alors**
    Ajouter la tâche dans $\mathcal{E}_J$
  Sinon
    Si $e_i < e_J$ **Alors**
      Mettre à jour l'état de la tâche dans $\mathcal{E}_i$
    Sinon
      Mettre à jour l'état de la tâche dans $\mathcal{E}_J$
      Ajouter i dans r_J si i possède les résultats de idT_i
    Fin Si
  Fin Si
Fin Pour
Pour tout $(idT_J, idE_J, p_J, e_J, r_J) \in \mathcal{E}_J$ **Faire**
  Si $idT_J \notin \mathcal{E}_i$ **Alors**
    Ajouter la tâche dans $\mathcal{E}_i$
  Fin Si
Fin Pour

Ce type de gestion est complètement décentralisé. En effet, les nœuds possèdent les paramètres des tâches ainsi que leur état courant. Ils sont donc en mesure de débuter le calcul d'une tâche à tout moment en fonction de leurs caractéristiques locales (puissance de calcul, mémoire, espace disque, ressources logiques). Cependant, l'état des tâches sur chaque nœud ne reflète par nécessairement leur état réel dans le système. Si un nœud modifie l'état d'une tâche, cette modification n'est pas répercutée immédiatement à tous les nœuds. Aussi, la sélection locale d'une tâche peut entraîner la réplication du calcul de certaines tâches appelées alors *tâches répliquées*.

Définition 4.2 (Tâche répliquée) *Une tâche est dite répliquée lorsqu'elle est calculée et terminée par un nœud alors que son résultat est déjà connu par un autre nœud ou localement.*

Remarque 4.1 *Lorsque deux nœuds sélectionnent simultanément la même tâche (ou plus généralement lorsqu'un nœud sélectionne une tâche qui est en cours de calcul sur un autre nœud), cela ne produit pas nécessairement une tâche répliquée. Si l'un des nœuds tombe en panne avant la fin de son calcul, la tâche ne sera calculée qu'une seule fois.*

Tout comme les tâches, chaque nœud est caractérisé par un état : *en attente* s'il attend pour calculer une tâche, *en cours de calcul* s'il calcule actuellement une tâche et *stoppé* s'il n'a plus de tâche à calculer. Lorsqu'un nœud i est stoppé, toutes les tâches de l'ensemble $\mathcal{E}_i$ sont marquées comme *en cours* ou *calculées*.

Définition 4.3 (Temps initial) *Nous appelons le temps initial, noté t_0, le temps à partir duquel tous les nœuds possèdent l'ensemble des paramètres des tâches et sont tous prêts à calculer.*

Dans notre étude, le temps de diffusion des paramètres des tâches n'est pas pris en compte. Nous nous basons sur t_0 afin de comparer uniquement l'efficacité des méthodes d'assignation des tâches. Quelle que soit la méthode que nous proposons dans la suite, nous supposons que cette diffusion est réalisée au préalable et qu'elle se termine à t_0.

Remarque 4.2 *Le jeton que nous utilisons pour diffuser l'état des tâches peut être utilisé pour diffuser les paramètres des nouvelles tâches. Dans ce cas, cela n'induit aucun mécanisme supplémentaire.*

Définition 4.4 (Temps d'exécution séquentiel) *Le temps d'exécution séquentiel d'un ensemble de tâches $\mathcal{T}$ est le temps nécessaire pour qu'un nœud unique calcule toutes les tâches de $\mathcal{T}$, les unes après les autres à partir de t_0. Nous notons ce temps t_{seq}.*

Lorsqu'un nœud possède un ensemble de tâches, il en sélectionne une et la calcule. Une fois le calcul terminé, il choisit la suivante et ainsi de suite jusqu'à ce que toutes les tâches soient calculées. Nous supposons ici que pour un nœud unique, le temps de début d'une tâche correspond exactement au temps de fin de la tâche précédente[1]. Nous considérons que le temps de sélection d'une tâche est largement inférieur au temps de calcul de la tâche : pour simplifier, nous l'incluons dans le temps de calcul. Le temps séquentiel nous permet donc de comparer une méthode distribuée par rapport à une méthode centralisée.

Définition 4.5 (Temps d'exécution distribué) *Le temps d'exécution distribué, noté $t_e(V)$, pour un ensemble de nœuds V et pour un ensemble de tâches $\mathcal{T}$ est le temps nécessaire pour que toutes les tâches de $\mathcal{T}$ soient calculées par l'ensemble des nœuds V. Pour simplifier, nous le notons t_e.*

Lorsque toutes les tâches ont été calculées, tous les nœuds ne sont pas nécessairement arrêtés. Par exemple, si la dernière tâche est en cours de calcul sur plusieurs nœuds, les derniers nœuds finissent leur calcul après t_e. Dans ce cas, les résultats du calcul sont ignorés ou comparés à ceux existants. Nous introduisons la notion de *temps total d'exécution*.

[1] Le temps de début de la première tâche correspond à t_0.

Définition 4.6 (Temps total d'exécution distribué) *Lorsque toutes les tâches sont supposées calculées par tous les nœuds, c'est-à-dire que pour toute tâche, son état est* calculé*, tous les nœuds sont dans l'état* arrêté*. Nous appelons ce temps, le temps total d'exécution noté* t_t*.*

Le temps total d'exécution distribué est égal ou supérieur au temps d'exécution distribué.

Définition 4.7 (Temps d'exécution d'un nœud) *Le temps d'exécution d'un nœud i noté* t_{e_i} *est le temps depuis* t_0 *jusqu'à ce que le nœud détecte la fin de l'exécution et qu'il se place alors en état* arrêté*. Durant le temps d'exécution, un nœud peut être en attente, calculer une tâche ou être en panne.*

Le maximum de tous les temps d'exécution des nœuds est égal au temps total d'exécution distribué.

Définition 4.8 (Temps de calcul) *Le temps de calcul du nœud i, noté* t_{c_i}*, est la somme de tous les temps durant lesquels il calcule une tâche.*

Nous verrons dans la suite que cette notion permet d'étudier l'utilisation de la puissance de calcul des nœuds au cours d'une exécution.

4.2.2 Méthode passive

Le jeton est utilisé pour mettre à jour l'état des tâches sur les nœuds (*non calculée*, *en cours* ou *calculée*). Les nœuds possèdent les paramètres nécessaires pour calculer les tâches mais ils n'ont qu'une connaissance locale de leur état d'avancement. Lorsqu'un nœud sélectionne une tâche, celle-ci est marquée localement comme étant en cours. C'est le jeton qui diffuse ce changement d'état aux autres nœuds. La répercussion de cet état sur les autres nœuds dépend du temps mis par le jeton pour atteindre ce nœud (temps de percussion de la marche) et pour atteindre tous les autres nœuds (temps de couverture de la marche).

Avec la méthode passive, les nœuds attendent de posséder le jeton avant de sélectionner une nouvelle tâche à calculer. Nous nous assurons ainsi qu'aucune tâche n'est répliquée. L'algorithme 14 présente l'algorithme exécuté à chaque réception du jeton. Cependant, l'attente du jeton implique une perte de la puissance de calcul qui n'est pas utilisée. L'efficacité de la méthode passive dépend donc du temps que met le jeton à atteindre les nœuds en attente d'une tâche à calculer. Plus le jeton est long, plus de la puissance de calcul est perdue. Nous appelons ce temps, le *temps d'attente*.

Définition 4.9 (Temps d'attente) *Le temps d'attente du nœud i noté* ta_i *correspond au temps durant lequel le nœud i attend sans calculer de tâche.*

Exemple 4.1 *Le diagramme de la figure 4.2 présente un exemple d'exécution de la méthode passive. Le temps d'exécution des différents nœuds est partagé entre le temps effectif de calcul (représenté par des rectangles, chacun représentant le calcul d'une tâche) et les différents temps d'attente. Pour le nœud 3, par exemple, nous notons un premier temps d'attente* $ta4$ *qui correspond au temps nécessaire pour calculer sa première tâche (*$t4$*) et* $ta5$ *qui correspond au temps nécessaire pour calculer la tâche suivante. À ce moment, les tâches sont toutes calculées ou en cours de calcul. Lorsque le jeton arrive sur le nœud 3, ce dernier passe dans l'état* arrêté*.*

Dans le cas particulier de cette exécution, nous observons que le nœud 4 s'arrête aussitôt après le calcul de sa tâche, temps qui correspond au temps t_e. Le nœud 5 qui est en attente du jeton, ne détecte qu'après t_e que toutes les tâches ont été assignées ou calculées. Lorsqu'il s'arrête, cela correspond au temps total d'exécution t_t.

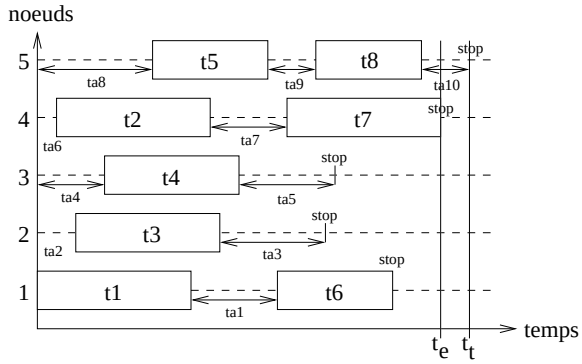


FIG. 4.2 – Exemple de diagramme d'exécution de la méthode passive : les rectangles représentent le calcul des tâches ($t1$ à $t8$) et les temps $ta1$ à $ta10$ représentent les temps d'attente.

Algorithme 14 Réception du jeton J par un nœud i avec la méthode passive.

Si J est valide **Alors**
  Mise-à-jour des ensembles $\mathcal{E}_J$ et $\mathcal{E}_i$ (algorithme 13)
  Si i est dans l'état *en attente* **Alors**
    Si il reste encore des tâches à calculer **Alors**
      Sélectionner une tâche non-calculée t dans $\mathcal{E}_J$
      Mise-à-jour de l'état de t (en cours) dans $\mathcal{E}_J$ et $\mathcal{E}_i$
      Passer i dans l'état *en cours de calcul*
      Débuter le calcul de t
    Sinon
      Passer dans l'état *stoppé*
    Fin Si
  Fin Si
  Envoyer J à un voisin de i
Fin Si

4.2.3 Méthode active

Nous avons vu que la méthode passive implique une perte de la puissance de calcul due à l'attente du jeton. Avec la méthode active, les nœuds sélectionnent une tâche avant la réception du jeton. Chaque nœud possède localement l'ensemble des paramètres des tâches et est donc en mesure d'en sélectionner une marquée comme *non calculée*. Cet état est une vision locale et peut donc être invalide : si un nœud débute le calcul d'une tâche, le temps nécessaire pour que le jeton arrive sur ce nœud et avertisse les autres nœuds du nouvel état de la tâche (*en cours*), cette tâche peut être calculée par un voire plusieurs autres nœuds. Nous obtenons

alors des tâches répliquées. Afin de limiter le nombre de celles-ci, les tâches à calculer sont choisies aléatoirement parmi l'ensemble des tâches non calculées. La probabilité que deux nœuds sélectionnent simultanément la même tâche est donc réduit.

Exemple 4.2 *La figure 4.3 représente un exemple d'exécution de la méthode active. Plusieurs tâches sont répliquées : les tâches 6 et 7. Lorsque le nœud 1 termine la tâche* $t1$*, il en sélectionne une aléatoirement. Or, le jeton n'a pas encore mis à jour sa connaissance locale. La tâche 6 est choisie alors qu'elle est en cours de calcul par le nœud 2. Nous remarquons que tous les nœuds s'arrêtent lorsqu'ils terminent leur dernière tâche.*

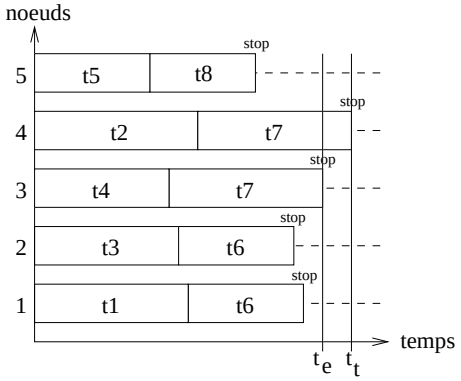


FIG. 4.3 – Exemple de diagramme d'exécution de la méthode active : les rectangles représentent le calcul des tâches ($t1$ à $t8$).

Comme nous pouvons le voir dans cet exemple, l'efficacité de la méthode active dépend directement du nombre de tâches répliquées. La production de tâches répliquées n'implique pas nécessairement l'augmentation de t_t, mais peut augmenter le temps de calcul des nœuds. Nous verrons dans la suite quels facteurs favorisent l'augmentation du nombre de tâches répliquées.

4.2.4 Tolérance aux pannes

La gestion des tâches que nous proposons est basée sur la circulation d'un jeton. Nous avons vu dans la section 1.4.5 les différents cas que nous devions gérer dans les applications à base de circulation aléatoire de jeton.

Perte de jeton. Le jeton contient l'ensemble des états des tâches. Cet ensemble est connu en local sur chaque nœud. Aussi, en utilisant le mécanisme des compte-à-rebours, un jeton est régénéré à partir de la vision locale du nœud. Pour la méthode passive, la duplication du jeton peut induire une duplication des tâches. La valeur d'initialisation des compte-à-rebours doit donc être suffisamment grande. Le jeton est redéfini par $J = \{idJ, \mathcal{E}_J\}$ où idJ est l'identifiant du jeton défini dans la section 1.4.5.

Duplication de jeton. Le mécanisme des compte-à-rebours peut induire une duplication des jetons tout comme le protocole de communication. Comme nous avons vu précédemment, la duplication du jeton est préjudiciable pour la méthode passive. L'exclusion mutuelle sur chaque

tâche n'est plus assurée et peut induire la duplication de tâches. Cependant, à l'aide de l'identité des jetons, les jetons superflus sont supprimés en un temps fini.

Corruption des données. Ces deux méthodes sont basées sur la circulation des états des tâches. Si ceux-ci sont corrompus, la sélection des tâches peut induire des tâches dupliquées mais aussi des tâches qui ne pourront être calculées. L'un des cas le plus couramment rencontré est du à la déconnexion des nœuds : si un nœud sélectionne une tâche, l'état de celle-ci dans le jeton est modifié et diffusé dans la grille. Or, si ce nœud tombe en panne avant la fin du calcul, la tâche est toujours marquée dans le jeton et sur les autres nœuds comme étant en cours de calcul : cette tâche devient une *tâche zombie* car elle ne peut plus être calculée. Il est donc essentiel de mettre en place un nouveau mécanisme. Nous proposons d'utiliser la *réplication volontaire*.

Définition 4.10 (Réplication volontaire) *La réplication d'une tâche est dite volontaire lorsque le calcul de cette tâche est connu comme étant en cours ou terminé et que la tâche est à nouveau assignée.*

Aussi, lorsque l'ensemble des tâches ne contient plus que des tâches notées comme terminées ou en cours, les nœuds sélectionnent une tâche parmi toutes celles notées en cours. Si cette tâche est une tâche zombie, son calcul est donc possible. Dans le cas contraire, cela entraîne une réplication de la tâche.

Si la réplication volontaire permet de gérer les tâches zombies, elle permet aussi d'accélérer le calcul ou de vérifier les résultats d'une tâche. Dans *BOINC*, chaque tâche est ainsi calculée plusieurs fois afin de croiser les résultats obtenus dans des configurations différentes (un mécanisme permet de s'assurer que le calcul d'une même tâche n'est pas réalisé sur un même nœud). Si les résultats sont différents, cela peut provenir d'un problème matériel (problème d'arrondis dans les calculs à nombres flottants) ou d'un acte de malveillance (un utilisateur peut volontairement retourner des résultats invalides). Dans *CONFIIT*, la réplication volontaire est utilisée pour accélérer le calcul : si une tâche est assignée sur un nœud lent, l'assigner à nouveau sur un autre nœud permet éventuellement de gagner du temps si ce nœud est plus rapide, au dépend d'une perte de la puissance de calcul.

Si l'état e d'une tâche est corrompu, il passe dans un état invalide e'. Si e' est moins avancé que e, l'état est corrigé en un temps fini grâce à l'algorithme 13. Dans le cas contraire, le jeton diffuse aux autres nœuds un mauvais état. Si cet état est *en cours*, la réplication volontaire corrige le problème et nous sommes assurés que la tâche sera calculée. Si l'état corrompu est *calculé*, la réplication volontaire n'est plus suffisante. Aussi, lorsque l'état d'une tâche est *calculé*, l'ensemble r doit posséder au moins une identité. Si ce n'est pas le cas, l'état de la tâche est corrigé en *non calculé*. Enfin, si un nœud détecte son identité dans l'ensemble r et qu'il ne possède pas les résultats de la tâche en local, il retire son identité de r. Si r est vide, l'état de la tâche est là-encore corrigé en *non calculé*. En dernier recours, si l'émetteur d'une tâche détecte que tous les nœuds placés dans l'ensemble r ne possèdent pas les résultats ou se sont déconnectés entre temps, la tâche peut être à nouveau soumise.

4.3 Comparaison entre la méthode active et la méthode passive

Comme nous avons vu précédemment, l'efficacité de ces deux méthodes repose sur la rapidité de circulation du jeton, c'est-à-dire son temps de couverture. Nous nous intéressons donc aux différents paramètres qui peuvent modifier le temps de couverture. Nous distinguons les paramètres suivants : le nombre de nœuds de la grille et l'aspect et les performances du réseau d'interconnexion des nœuds. Le temps de couverture doit aussi être mis en parallèle avec les paramètres associés aux tâches : le nombre de tâches, le temps moyen d'exécution des tâches et l'irrégularité des temps d'exécution de toutes les tâches. Nous présentons aussi des simulations dans des environnements dynamiques afin de montrer les impacts des pannes et de la réplication volontaire.

4.3.1 Notion d'efficacité

Une gestion des tâches efficace a plusieurs objectifs. Tout d'abord, elle doit répartir les calculs sur tous les nœuds de la grille afin d'utiliser équitablement les ressources. Ensuite, elle doit minimiser les ressources. Une solution qui exploite moins de ressources pour un même calcul est préférable et particulièrement dans le cas du vol de cycles. En effet, les applications de grille sont exécutées en même temps que des applications locales. Enfin, une gestion des tâches doit limiter le temps de calcul global. L'exécution de calculs sur une grille a pour but de réduire le temps d'exécution. D'autres facteurs, tels que celui économique, n'ont pas été pris en compte dans nos travaux.

En partant d'un ensemble donné de tâches, il est possible de comparer des solutions distribuées par rapport à une solution séquentielle. Nous utilisons la notion d'accélération.

Définition 4.11 (Accélération) *L'accélération d'une gestion des tâches distribuée, notée* Acc*, est le rapport entre le temps d'exécution séquentiel t_{seq} et le temps d'exécution distribué t_e :*

$$Acc = \frac{t_{seq}}{t_e}$$

L'accélération ne met pas en évidence le nombre de nœuds utilisés pour réaliser le calcul distribué. Ainsi, deux solutions ayant toutes les deux une efficacité de 0.8 pour un même ensemble de tâches, peuvent utiliser un nombre de nœuds différent. Dans ce cas, la solution qui utilise le moins de nœuds est plus efficace. Pour en tenir compte, nous parlons d'efficacité.

Définition 4.12 (Efficacité) *L'efficacité d'une gestion des tâches distribuée pour un nombre de nœuds* n*, exprimée en pourcentage, est égale à :*

$$e = \frac{t_{seq}}{t_e \times n} \times 100$$

Plus l'efficacité se rapproche de 100%, plus la gestion des tâches est optimale. Une baisse de l'efficacité peut être expliquée de plusieurs manières. Tout d'abord, s'il y a trop de nœuds par rapport au nombre de tâches, certains nœuds n'ont pas de travail. Or, l'efficacité est calculée

sur l'ensemble des nœuds. Ensuite, les temps d'attente pour la méthode passive sont trop nombreux. Les nœuds passent beaucoup de temps à attendre le jeton avant d'exécuter le calcul d'une nouvelle tâche. Enfin, s'il y a trop de tâches répliquées, la puissance de calcul des nœuds est dépensée.

4.3.2 Paramètres de simulation

Pour comparer les deux méthodes d'assignation, nous simulons le calcul de deux types d'ensembles finis de tâches. Dans un premier temps, nous considérons un ensemble fixé de temps mesurés lors d'une exécution en conditions réelles d'un problème donné. Dans un deuxième temps, nous considérons des ensembles aléatoires générés selon une loi de probabilité.

Dans [Kra99], Michaël Krajecki introduit les applications *FIIT*. Ce type d'application produit des ensembles finis de tâches qui ont les particularités suivantes : 1) aucune tâche ne dépend des résultats d'une autre tâche (il n'y a aucun ordre d'exécution), 2) le temps de calcul d'une tâche n'est pas prévisible et 3) elles sont indivisibles. Ces particularités empêchent de calculer un ordonnancement optimal au préalable et un ordonnancement dynamique est donc inévitable.

Le problème de Langford consiste à arranger des cubes de couleur suivant une règle déterminée. Pour simplifier, chaque couleur est numérotée de 1 à c (c est le nombre de couleurs) et pour chacune d'entre elles, nous avons un nombre identique de cubes, noté n avec $n \geq 2$. Un arrangement valide de tous les cubes doit respecter la condition : "*entre deux cubes d'une couleur j donnée, il ne peut y avoir que j cubes de couleurs différentes*". La résolution d'une instance du problème, notée $L = (n, c)$, consiste à énumérer tous les arrangements valides. La figure 4.4 présente un arrangement valide pour $L = (2, 3)$.

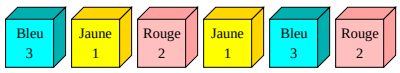


FIG. 4.4 – Une solution au problème de Langford avec 3 couleurs et 2 cubes de chaque couleur (instance $L = (2, 3)$).

L'explosion combinatoire de ce problème nécessite un calcul intensif et dans [KFJ+05], les auteurs proposent une méthode pour paralléliser le calcul. La résolution d'une instance revient alors à parcourir un arbre où chaque branche est un arrangement possible. En partant d'une profondeur donnée, nous obtenons un ensemble de tâches *FIIT*. Des exécutions ont été réalisées pour plusieurs instances et en particulier, nous utilisons dans ce chapitre, les temps des tâches obtenus lors d'une exécution avec l'intergiciel *CONFIIT* pour l'instance $L = (2, 16)$. L'ensemble, que nous notons dans la suite $\mathcal{T}_{Lang}$, contient 20490 tâches et la répartition des longueurs est représentée sur la figure 4.5. L'irrégularité des durées des tâches est assez importante et s'explique par l'algorithme de résolution du problème. Certaines branches doivent être développées jusqu'aux feuilles pour détecter si l'arrangement est valide, ce qui entraîne des tâches assez longues. D'autres branches sont rapidement élaguées si une incohérence est détectée, ce qui produit des temps très courts.

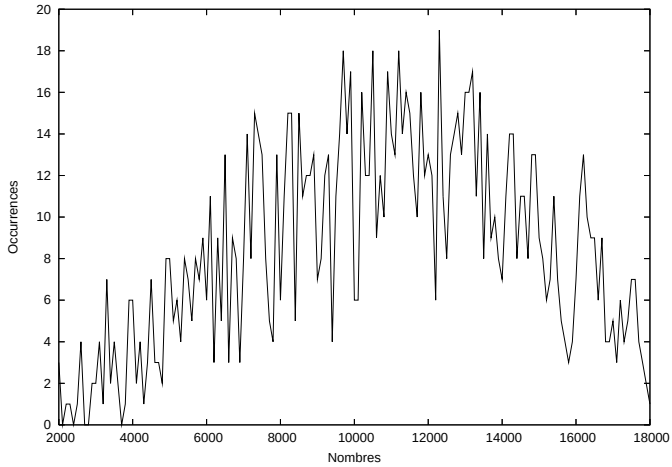


FIG. 4.5 – Répartition des longueurs des tâches de l'ensemble $\mathcal{T}_{Lang}$ (instance $L(2, 16)$).

Les autres ensembles de tâches sont générés aléatoirement à l'aide de la loi de probabilité log-normale. Une variable aléatoire X est une quantité dont ses valeurs sont aléatoires. X est caractérisée par sa fonction de répartition.

Définition 4.13 (Fonction de répartition) *La fonction de répartition d'une variable aléatoire X est l'application F de $\mathbb{R}$ dans $[0, 1]$ définie par :*

$$F(x) = P(X \leq x)$$

La répartition des valeurs que peut prendre une variable aléatoire X est décrite par sa loi de probabilité.

Définition 4.14 (Loi de probabilité) *On appelle loi de la variable aléatoire X, la loi de probabilité P_X qui à toute réunion dénombrable d'intervalles de $\mathbb{R}$ associe :*

$$P_X(I) = P(X \in I)$$

Définition 4.15 (Variable aléatoire absolument continue) *Une variable aléatoire X est absolument continue s'il existe une fonction f positive et intégrable telle que pour tout intervalle I de $\mathbb{R}$, on ait :*

$$P_X(I) = \int_I f(t)dt$$

On appelle f la densité de probabilité de la variable aléatoire X.

F est alors dérivable et admet f pour dérivée. On a :

$$P(a < X < b) = \int_a^b f(x)dx = F(b) - F(a)$$

Dans notre étude, nous avons utilisé en particulier la loi log-normale qui propose une répartition des nombres plus ou moins large en fonction de ses paramètres.

Définition 4.16 (Loi log-normale) *La loi log-normale est la loi d'une variable positive X telle que son logarithme népérien suive une loi de Laplace-Gauss. Sa densité est :*

$$f(x, \zeta, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} \exp^{-\frac{1}{2}\left(\frac{\ln x - \zeta}{\sigma}\right)^2}$$

Nous générons des temps de tâches en faisant varier les paramètres ζ et σ. La figure 4.6 représente la répartition d'un échantillon de valeurs calculées à l'aide de la loi log-normale avec comme paramètres $\zeta = 7$ et plusieurs valeurs de σ allant de 0,1 à 0,9. Plus les valeurs de σ sont faibles, plus la répartition des longueurs des tâches est proche de la moyenne. Afin d'obtenir des tâches suffisamment grandes, nous ajoutons 2000 aux valeurs et nous réalisons un seuillage par 20000 (ces valeurs ont été choisies arbitrairement à partir d'observations sur $\mathcal{T}_{Lang}$). Dans la suite, nous notons ces ensembles $\mathcal{T}_{\sigma=x}$ où x est la valeur de σ, ζ étant fixé à 7.

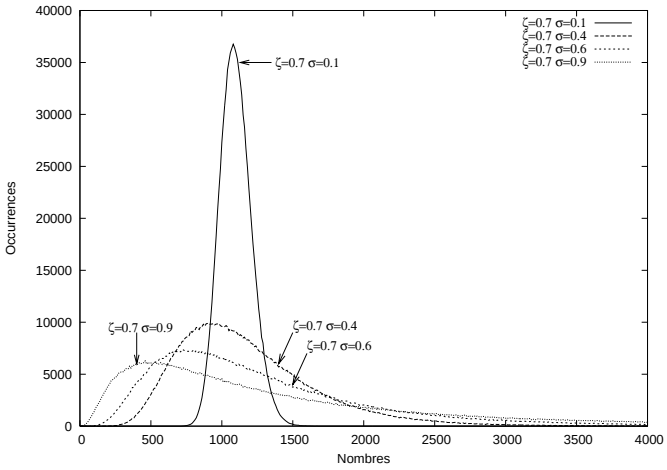


FIG. 4.6 – Répartition de valeurs générées à l'aide de la loi log-normale avec $\zeta = 7$ et différentes valeurs pour σ (c).

Dasor permet d'appliquer des capacités de calcul différentes pour chaque nœud de la grille (nombre de processeurs et vitesse des processeurs). Dans les simulations présentées dans ce chapitre, nous choisissons d'appliquer des capacités identiques pour tous les nœuds de la grille : un processeur par nœud et une vitesse identique pour tous. Cependant, à partir d'un ensemble donné de tâches, nous pouvons appliquer une vitesse plus grande aux nœuds pour simuler le calcul de tâches plus courtes tout en gardant la même irrégularité de l'ensemble.

4.3.3 Influences de la taille de la grille

Pour un nombre de tâches donné, l'augmentation du nombre de nœuds dans la grille doit permettre de diminuer le temps d'exécution. L'accélération est cependant limitée : les tâches sont indivisibles et s'il y a trop de nœuds, certains n'ont pas de tâche à calculer. Nous avons vu que le temps de couverture d'une marche aléatoire dépendait du nombre de nœuds dans le réseau (voir section 2.4.1). Nous cherchons donc à déterminer dans quelle mesure, l'augmentation de la taille du réseau influence l'efficacité des deux méthodes.

Nous avons réalisé des simulations sur des réseaux aléatoires dans lesquels nous avons augmenté le nombre de nœuds de 1000 à 5000. Les résultats sont représentés sur la figure 4.7 : la figure (a) représente des exécutions avec l'ensemble de tâches $\mathcal{T}_{Lang}$ et la figure (b) avec des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches.

Lorsque le nombre de nœuds est peu important (1000 nœuds), la méthode active propose une meilleure efficacité que la méthode passive. La différence est plus marquée avec l'ensemble $\mathcal{T}_{Lang}$ dont la répartition des longueurs des tâches est plus large. Le nombre de tâches répliquées avec la méthode active est faible. Cependant, l'attente du jeton avec la méthode passive ne permet pas de compenser cette perte de la puissance de calcul.

Si le nombre de nœuds dans la grille augmente, l'efficacité des deux méthodes diminue. Cela s'explique par l'augmentation du temps de couverture de la marche aléatoire. Pour la méthode passive, cela implique des temps d'attente du jeton plus important. Pour la méthode active, une mise-à-jour moins rapide des états des tâches entraîne une augmentation du nombre de tâches répliquées. Ainsi, nous observons que l'efficacité des deux méthodes se rapprochent. Avec l'ensemble $\mathcal{T}_{\sigma=0.1}$, nous pouvons remarquer que l'efficacité de la méthode passive devient meilleure que celle de la méthode active lorsque le nombre de nœuds dépasse 4000.

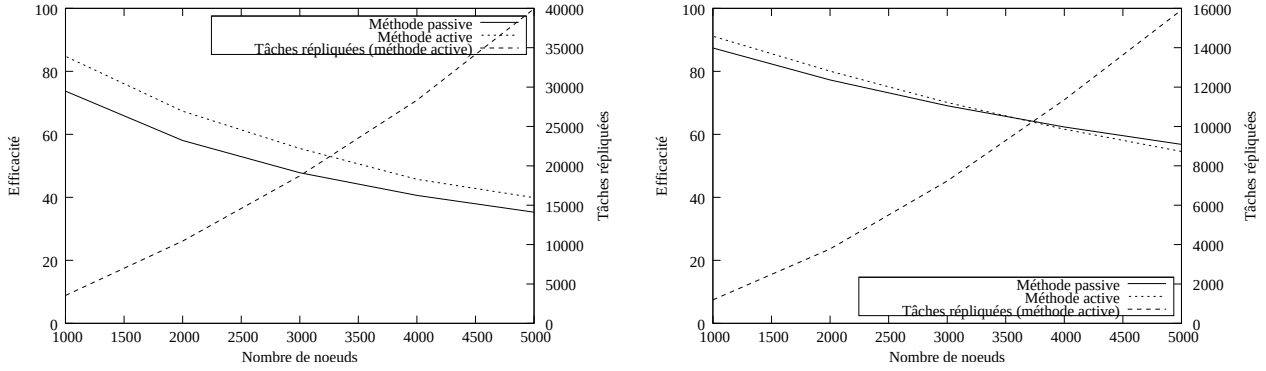


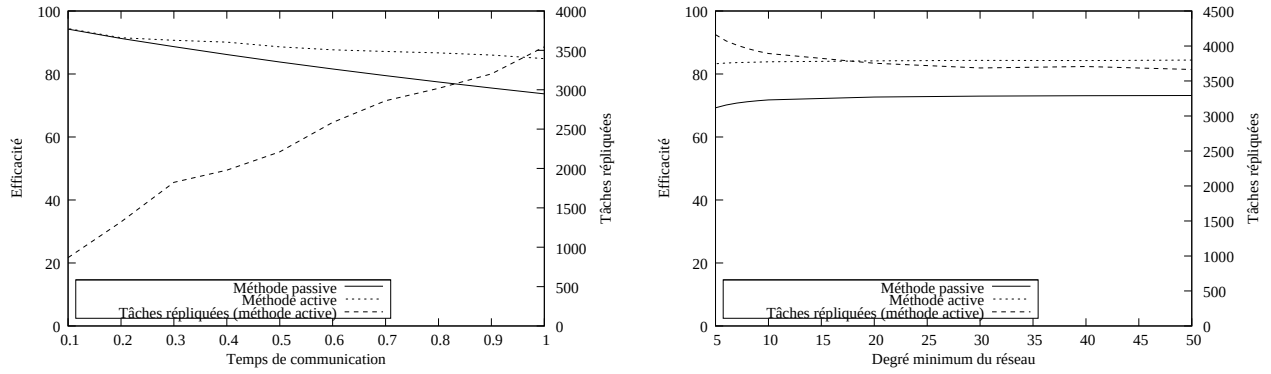
FIG. 4.7 – Efficacités des méthodes passive et active en fonction du nombre de nœuds dans la grille pour un ensemble de tâches donné. La figure (a) présente des résultats d'exécutions avec des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches et la figure (b) avec l'ensemble de tâches $\mathcal{T}_{Lang}$.

4.3.4 Les caractéristiques du réseau

Nous avons vu que le temps de couverture du jeton varie en fonction du débit du réseau d'interconnexion (voir section 1.4.4). Nous cherchons donc à observer l'influence du temps de communication sur l'efficacité des deux méthodes. En effet, plus il est élevé, plus le temps de couverture du jeton est important, entraînant un nombre de tâches répliquées plus important pour la méthode active et une augmentation du temps d'attente pour la méthode passive.

Nous avons réalisé des séries de simulations sur des réseaux de 1000 nœuds en faisant varier le temps de communication de 0.1 à 1 unité de temps. Les résultats sont représentés sur la figure 4.8 (a). Pour un temps de communication très court, le jeton circule très rapidement dans le réseau et les efficacités des deux méthodes sont identiques. Dans ce cas, les états des tâches sont mis à jour très rapidement et le nombre de tâches répliquées pour la méthode active est assez faible. De même, lorsqu'un nœud termine une tâche avec la méthode passive, il attend

le jeton durant un temps très court. La diminution du temps d'attente et du nombre de tâches répliquées implique une très bonne efficacité pour les deux méthodes.



(a) (b)

FIG. 4.8 – La figure (a) présente l'évolution de l'efficacité des deux méthodes en fonction du temps de communication et la figure (b) en fonction du degré minimum dans le réseau. Les deux courbes correspondent à des séries de simulations exécutées sur des réseaux de 1000 nœuds avec des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches.

Lorsque le temps de communication est plus important, le nombre de tâches répliquées augmente, impliquant une légère réduction de l'efficacité avec la méthode active. Celle de la méthode passive paraît plus sensible au temps de communication et nous observons une baisse plus nette (environ 10% d'écart avec la méthode active).

Dans la section 2.4.1, nous avons observé que le temps de couverture d'une marche aléatoire dépend aussi du degré du réseau. Pour un ensemble de tâches donné et un temps de communication fixé, nous avons réalisé des séries de simulations sur des réseaux aléatoires de degré minimum, le degré minimum variant de 5 à 50. La figure 4.8 (b) présente les résultats obtenus. Nous observons que l'influence du degré est très faible pour les deux méthodes. Pour des degrés faibles, c'est-à-dire inférieurs à 10, nous notons cependant une augmentation du nombre de tâches répliquées avec la méthode active sans conséquence notable sur l'efficacité (le temps de calcul des nœuds est cependant accru). Dans les mêmes conditions, pour la méthode passive, le temps d'attente est plus important, ce qui entraîne une réduction légère de l'efficacité (moins de 5%).

4.3.5 Les caractéristiques des tâches

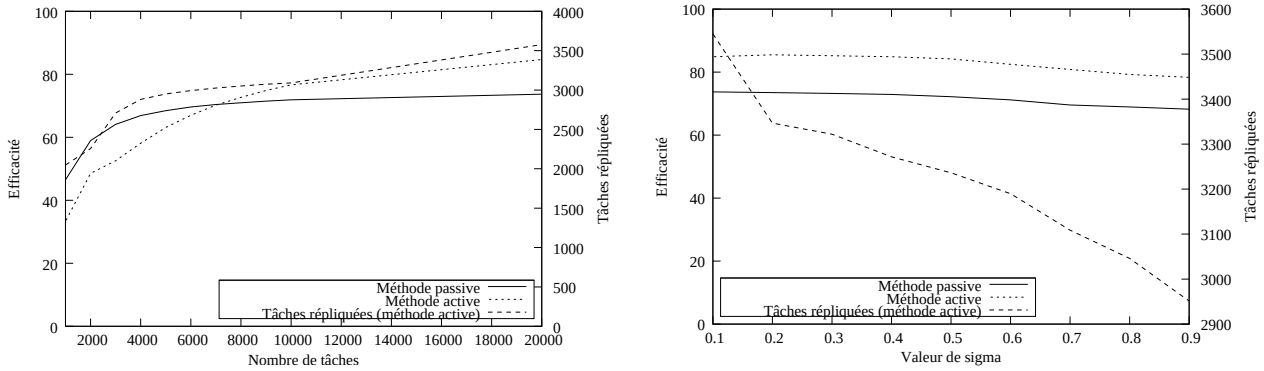
Le nombre de tâches produites dans une grille de calcul dépend des utilisateurs et des applications déployées au-dessus de la grille. Dans cette section, nous cherchons à déterminer l'influence des caractéristiques des tâches (nombre, irrégularité et longueur) sur les deux méthodes.

Plus le nombre de tâches est faible par rapport au nombre de nœuds, plus la méthode active entraîne un nombre important de tâches répliquées. En effet, lorsque le nombre de tâches non calculées est faible, la probabilité que plusieurs nœuds sélectionnent la même tâche est très importante. Nous avons donc réalisé des simulations sur des réseaux aléatoires de 1000 nœuds avec des ensembles $\mathcal{T}_{\sigma=0.1}$. Nous avons fait varier le nombre de tâches de 1000 (1 tâche par nœud)

jusqu'à 20000. Les résultats sont représentés sur la figure 4.9 (a). Nous remarquons que lorsque le nombre de tâches est très faible, les efficacités des deux méthodes sont faibles. Cependant, nous remarquons que pour un rapport nombre de tâches sur nombre de nœuds inférieur à 7, la méthode passive propose de meilleurs résultats. L'attente du jeton évite en effet les tâches répliquées qui dégradent l'efficacité de la méthode active.

Nous cherchons maintenant à déterminer l'influence de l'irrégularité des tâches. En effet, suivant l'application qui produit les tâches, celles-ci ont des temps d'exécution plus ou moins irréguliers. Les tâches que nous utilisons sont générées avec la loi log-normale. En faisant varier les paramètres σ, nous obtenons des ensembles dont la répartition des valeurs est plus ou moins proche de la moyenne, comme nous pouvons l'observer sur la figure 4.6. Lorsque σ est proche de 0,9, la répartition des temps d'exécution des tâches est très large alors que pour un σ qui se rapproche de 0,1, les tâches ont des temps d'exécution très proches de la moyenne.

Nous réalisons des simulations avec des ensembles de 20000 tâches et nous faisons varier σ de 0,1 à 0,9. Les résultats sont représentés sur la figure 4.9 (b). Nous observons que les deux méthodes conservent une efficacité presque constante quel que soit la valeur de σ. Cependant, la méthode active induit un nombre plus important de tâches répliquées lorsque les tâches ont des temps d'exécution très proches. Dans ce cas, les nœuds terminent plus ou moins simultanément leur tâche, ce qui augmente la probabilité qu'ils sélectionnent simultanément la même tâche, le jeton ne couvrant pas assez rapidement le réseau.



(a) (b)

FIG. 4.9 – La figure (a) présente l'évolution de l'efficacité des méthodes passive et active sur des ensembles $\mathcal{T}_{\sigma=0.1}$ en fonction du nombre de tâches et la figure (b) en fonction de l'irrégularité des longueurs des tâches ($0.1 \leq \sigma \leq 0.9$ et $\zeta = 7$).

À partir d'un ensemble donné de tâches, nous cherchons à déterminer l'influence de la longueur moyenne d'exécution. Pour cela, nous faisons varier la vitesse de calcul des nœuds du réseau. En l'augmentant, nous simulons le calcul de tâches courtes tout en gardant l'irrégularité de l'ensembles des tâches. La figure 4.10 représente les résultats de simulations sur l'ensemble $\mathcal{T}_{Lang}$ et sur des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches. Dans les deux cas, l'efficacité des deux méthodes diminue très nettement, celle de la méthode passive étant plus sensible. Des tâches très courtes ont des effets néfastes pour les deux méthodes, le temps de couverture du jeton est alors insuffisant. Sur les ensembles générés à l'aide de la loi log-normale, les efficacités sont

moins bonnes. Cela s'explique par la valeur de σ, fixée à 0,1, qui produit des ensembles de tâches ayant des temps d'exécution très proches de la moyenne. Plus la puissance de calcul des nœuds est importante, plus les effets observés sur la figure 4.9 (b) s'amplifient.

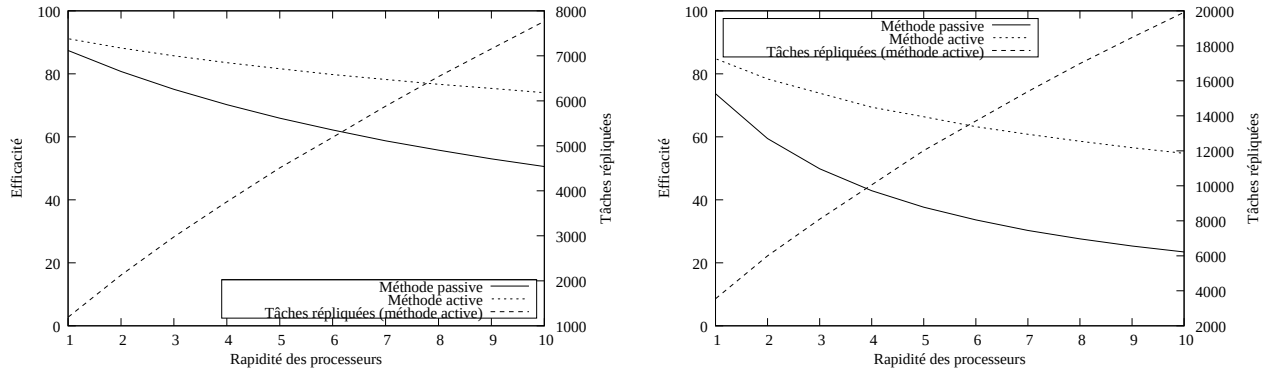


FIG. 4.10 – Impact des durées des tâches sur l'efficacité des méthodes active et passive. La figure (a) présente les résultats d'exécution avec l'ensemble $\mathcal{T}_{Lang}$ et la figure (b) avec des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches.

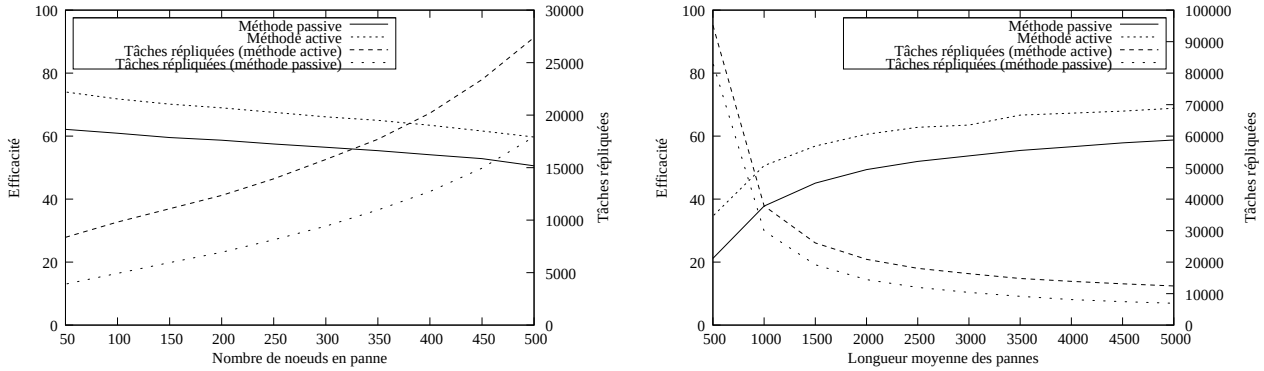
4.3.6 Les pannes dans la grille

Le dynamisme de la grille influence le temps d'exécution. Pour prendre en compte les pannes dans le système, nous introduisons une nouvelle méthode pour calculer l'efficacité. En partant du modèle de pannes fixe avec un nombre p de nœuds en panne, nous savons qu'à tout moment, il n'y a que $n - p$ nœuds qui sont en mesure de calculer. Nous calculons donc l'efficacité en fonction de $n-p$ au lieu de n. Nous pouvons comparer l'efficacité d'un réseau ayant 1000 nœuds et $p = 100$ nœuds en panne à un réseau de 900 nœuds.

Dans un premier temps, nous réalisons des simulations en faisant varier le nombre de nœuds en panne sur des réseaux de 1000 nœuds et nous donnons à p les valeurs de 50 à 500 (50% de nœuds en panne). La durée moyenne des pannes est de 5000 unités de temps. La figure 4.11 (a) présente les résultats obtenus avec des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches. Avec la méthode passive, des tâches répliquées apparaîssent dues à la réplication volontaire. Nous observons que l'efficacité des deux méthodes diminue légèrement lorsque le nombre de nœuds en panne augmente. Cela s'explique par le nombre de tâches répliquées engendrées par les pannes et par la réplication volontaire. En effet, lorsqu'un nœud sélectionne une tâche, celle-ci est marquée comme étant en cours de calcul. Lorsque le nombre de pannes est important, de plus en plus de tâches sont marquées comme en cours, produisant des tâches zombies. Nous remarquons d'autre part, que les deux méthodes évoluent sensiblement de la même manière, aussi bien en terme d'efficacité mais aussi en nombre de tâches répliquées.

Dans les séries précédentes, nous avons fixé la durée moyenne des pannes à environ 5000 unités de temps pour un temps de communication de 1. Nous réalisons une nouvelle série de simulations en faisant varier cette longueur moyenne, le nombre de nœuds en panne étant fixé à 200. Les résultats obtenus sont représentés sur la figure 4.11 (b). Nous observons que la

durée moyenne des pannes a beaucoup plus d'influence sur les efficacités. Le nombre de tâches répliquées est très important lorsque les pannes sont très courtes : le nombre de pannes étant important, le nombre de tâches zombies augmente jusqu'à ce toutes les tâches soient marquées comme en cours. Tout comme les simulations précédentes, la méthode passive évolue de manière identique à la méthode active.



(*a*) (*b*)

FIG. 4.11 – La figure (*a*) présente l'évolution de l'efficacité des deux méthodes en fonction du nombre de pannes dans le réseau et la figure (*b*) en fonction de la durée moyenne des pannes sur des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches dans des réseaux aléatoires de 1000 nœuds.

4.4 Optimisations des méthodes passive et active

Nous nous intéressons dans cette section à différentes optimisations des méthodes passive et active. Une première approche consiste à augmenter le nombre de jetons afin de diminuer le temps de visite des nœuds. D'autre part, nous avons vu dans la section 4.3.5 que lorsque le nombre de tâches est trop faible par rapport au nombre de nœuds, la méthode passive a une meilleure efficacité. Nous proposons donc une méthode hybride qui consiste à passer d'une méthode à l'autre tout au long de l'exécution afin de profiter des avantages de chacune d'entre elles. Nous proposons ensuite des méthodes basées sur une diffusion périodique des états des tâches dans le réseau. Pour cela, nous exploitons le contenu d'un mot circulant.

4.4.1 Multiple jetons

Le jeton a pour but de diffuser l'état des tâches entre tous les nœuds de la grille. Si nous augmentons le nombre de jetons, nous pouvons diminuer le temps de visite des nœuds. Le temps d'attente d'un jeton est ainsi diminué.

Pour la méthode active, le passage à plusieurs jetons est réalisé directement. En effet, chaque jeton peut contenir l'ensemble des états des tâches ce qui permet d'augmenter le taux de rafraîchissement et diminuer le nombre de tâches répliquées. Pour la méthode passive, le jeton assure une exclusion mutuelle sur les tâches. Si nous ajoutons un autre jeton qui contient un ensemble de tâches identique, des tâches répliquées sont produites. Nous devons donc répartir les tâches entre les jetons. Plusieurs stratégies sont possibles : 1) répartir les tâches en nombre égal dans chaque jeton, 2) en fonction des émetteurs des tâches ou 3) en fonction des ressources

nécessaires pour le calcul des tâches (application, bibliothèque ou fichiers). De même, il est possible de transférer dynamiquement les tâches d'un jeton à l'autre lorsqu'un nœud possède simultanément plusieurs jetons. Ces différentes stratégies sont aussi applicables avec la méthode active.

Nous avons réalisé des séries de simulations pour les méthodes passive et active en augmentant le nombre de jetons de 1 à 10. La figure 4.12 présente les résultats obtenus sur des réseaux aléatoires de 1000 nœuds avec des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches. Nous avons choisi la méthode 1) qui consiste à répartir les tâches en un nombre égal entre les jetons. Dans un premier temps, nous remarquons une diminution du nombre de tâches répliquées pour la méthode active et l'efficacité augmente sensiblement (environ 10%). Pour la méthode passive, nous observons une augmentation nette lors du passage de 1 à 2 jetons. Ensuite, l'efficacité n'évolue que légèrement et atteint un maximum avec plus de 8 jetons.

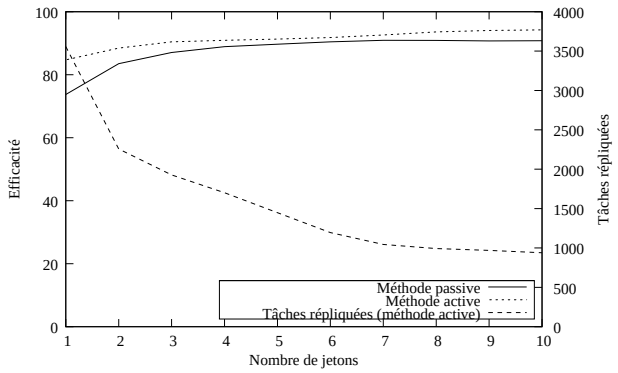


FIG. 4.12 – Évolution de l'efficacité en fonction du nombre de jetons dans le réseau sur des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches.

Si les efficacités sont améliorées pour les deux méthodes avec l'utilisation de plusieurs jetons, le choix du nombre de jetons doit être réalisé en fonction de la configuration courante du système (nombre de nœuds, réseau, tâches). Si cette configuration change, le nombre de jetons doit être ajusté pour garder une efficacité constante tout en limitant le nombre de messages produits. En effet, chaque jeton supplémentaire augmente le nombre de message émis et ce, indépendamment de la configuration. Il est donc nécessaire de mettre en place des mécanismes supplémentaires afin de modifier le nombre de jetons dynamiquement.

4.4.2 Méthode hybride passive/active

Dans la plupart des cas, la méthode active propose de meilleurs résultats que la méthode passive mais implique un nombre de tâches répliquées important qui limite son efficacité. La figure 4.13 présente le nombre de tâches répliquées au cours d'une exécution quelconque avec la méthode active. Nous observons que la majorité d'entre elles apparaît après 80% de l'exécution. À partir de ce moment, le nombre de tâches devient faible comparé au nombre de nœuds. Or, dans ce cas de figure, la méthode passive propose de meilleurs résultats comme nous pouvons l'observer sur la figure 4.9 (a). L'exclusion mutuelle sur chaque tâche empêche la réplication.

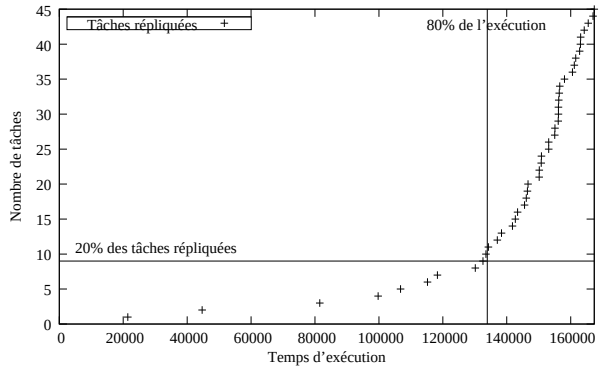


FIG. 4.13 – Évolution du nombre de tâches répliquées au cours d'une exécution avec la méthode active.

Nous proposons donc une méthode hybride qui consiste à passer de la méthode active vers la méthode passive et inversement en fonction du nombre de nœuds dans le réseau et du nombre de tâches restant à calculer. Pour cela, nous calculons le ratio suivant :

$$r = \frac{|\mathcal{T}|}{n}, \text{ où } n \text{ est le nombre de nœuds}$$

Lorsque r est en-dessous d'une borne minimale m, le nombre de tâches est insuffisant et les nœuds passent dans la méthode passive : s'ils terminent une tâche, ils attendent le jeton avant d'en sélectionner une nouvelle. Lorsque r dépasse la borne m, les nœuds passent dans la méthode active. Le ratio fluctue au cours de l'exécution en fonction du nombre de connexions ou déconnexions des nœuds, ainsi que du nombre de tâches soumises ou calculées dans le système. Ce calcul est réalisé localement à chaque fois qu'un nœud termine une tâche et, à un instant donné, les nœuds peuvent utiliser des méthodes différentes.

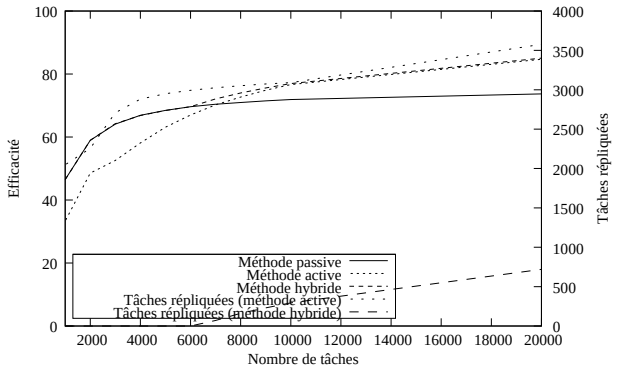


FIG. 4.14 – Évolution de l'efficacité de la méthode hybride en fonction du nombre de tâches avec $m = 7$ sur des ensembles $\mathcal{T}_{\sigma=0.1}$ de 20000 tâches.

Dans la section 4.3.5, nous avons observé que pour un réseau aléatoire de 1000 nœuds et pour un ensemble de tâches $\mathcal{T}_{\sigma=0.1}$, la méthode passive est meilleure que la méthode active lorsque le ratio est en dessous de 7. Nous avons réalisé des simulations de la méthode hybride avec $m = 7$ sur des ensembles $\mathcal{T}_{\sigma=0.1}$ de 1000 à 20000 tâches. Les résultats sont présentés sur

la figure 4.14. Lorsque le nombre de tâches initial est inférieur à 7000, avec la méthode hybride les nœuds ne passent que très rarement dans la méthode active. Le nombre de tâches répliquées est nul et donc l'efficacité est la même que celle de la méthode passive. Lorsque le nombre de tâches dépasse 7000, les nœuds passent plus souvent dans la méthode active. Nous observons alors une légère augmentation du nombre de tâches répliquées. Ce nombre reste très largement inférieur à celui de la méthode active qui est plus de 3 fois supérieur. Ainsi, l'efficacité de la méthode hybride est légèrement supérieure à celle de la méthode active.

Tout comme la méthode à base de multi-jetons, la borne minimale m dépend des caractéristiques des tâches et de la grille. Cependant, nous remarquons que la méthode hybride reste toujours supérieure à la méthode active car elle limite le nombre de tâches répliquées en fin de calcul. La puissance de calcul des nœuds est donc préservée et cette puissance peut donc être utilisée pour des applications locales.

4.4.3 Solutions basées sur la diffusion

Dans la section 1.5, nous avons introduit le mot circulant. Nous avons présenté une solution pour la gestion des ressources au niveau de la couche 4 de notre modèle théorique. Nous proposons d'exploiter le contenu du mot circulant pour construire des arbres couvrants. Ces arbres sont utilisés pour diffuser les états des tâches dans le réseau afin d'accélérer leur mise-à-jour. Nous distinguons plusieurs variantes : la diffusion simple le long d'un arbre notée $\mathcal{D}_s$, la diffusion et le retour à l'émetteur notée $\mathcal{D}_r$ et enfin, la diffusion, le retour suivi d'une nouvelle diffusion notée $\mathcal{D}_m$. Le principal avantage de ces solutions réside dans la réutilisation du mot circulant. Nous exploitons les structures existantes ce qui n'entraîne pas de surcoût de gestion.

Diffusions à l'aide d'un mot circulant ($\mathcal{D}_s$)

Nous définissons un jeton par le tuple $J = \{id_J, \mathcal{E}_J, W, C_J\}$, où id_J est l'identifiant du jeton, $\mathcal{E}_J$ est l'ensemble des états des tâches, W est un mot circulant[2] et C_J est un compteur de sauts. Lorsqu'un nœud i reçoit le jeton, il incrémente C_J et met à jour W. Si C_J est supérieur à une borne b, le nœud i construit un arbre de diffusion $\mathcal{A}_D$ enraciné en i à l'aide des données de W. Une diffusion de l'ensemble $\mathcal{E}_i$ (mis à jour à l'aide de $\mathcal{E}_J$) est ensuite amorcée le long de $\mathcal{A}_D$, comme représenté sur la figure 4.15. Nous définissons les messages de la diffusion par le tuple $M_D = \{id_J, \mathcal{E}_M, \mathcal{A}_D\}$ où id_J est l'identifiant du jeton (permettant de savoir si la diffusion doit être prise en compte par les nœuds), $\mathcal{E}_M$ qui est l'ensemble $\mathcal{E}_i$ mis à jour suivant les nœuds visités lors de la diffusion et $\mathcal{A}_D$ qui est l'arbre de diffusion. Lorsque ces messages sont reçus par un nœud, celui-ci met à jour son ensemble d'états de tâches et transmet le message mis à jour à tous ses voisins dans $\mathcal{A}_D$.

[2]Le mot circulant peut être orienté ou non, suivant le réseau de recouvrement.

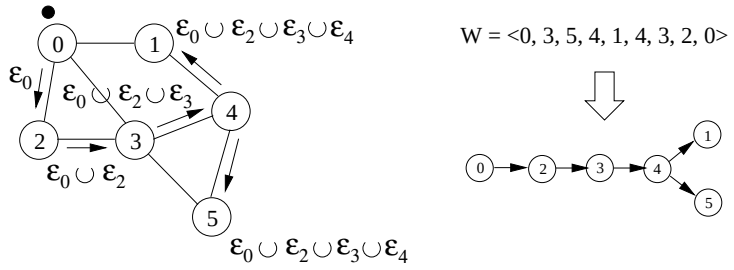


FIG. 4.15 – Exemple de diffusion des états des tâches à l'aide d'un mot circulant (ici, orienté).

Nous avons vu précédemment que le nombre de tâches répliquées est plus important vers la fin de l'exécution avec la méthode active, comme nous pouvons l'observer sur la figure 4.13. Pour éviter d'envoyer des diffusions inutiles et ainsi diminuer le nombre de messages émis dans le réseau, nous choisissons d'envoyer des diffusions en fonction du nombre de nœuds et du nombre de tâches restant à calculer. Les nœuds calculent donc le même ratio que pour la méthode hybride. Afin de régler la fréquence des diffusions, le ratio est calculé à un coefficient près. Nous définissons la borne b de la manière suivante :

$$b = min \left\{ \frac{|\mathcal{T}|}{n} * c_R, m_R \right\}$$

Où :
- $|\mathcal{T}|$ est le nombre de tâches restant à calculer ($\mathcal{T}$ étant l'ensemble des tâches) ;
- n est le nombre de nœuds dans le réseau ;
- c_R est le coefficient de rafraîchissement ;
- m_R est la valeur de rafraîchissement minimum.

Lorsque C_J est supérieur à b, une diffusion est émise dans le réseau et C_J est réinitialisé à 0. Lorsque le rapport entre le nombre de tâches et le nombre de nœuds devient trop petit, les diffusions deviennent de plus en plus fréquentes. Pour éviter la saturation du réseau, nous fixons un seuil minimum m_R. Le coefficient c_R permet de régler la fréquence de diffusion.

Enfin, pour optimiser la taille des messages diffusés, l'arbre $\mathcal{A}_D$ peut être élagué tout au long de la diffusion. Lorsqu'un nœud reçoit un message de la diffusion, pour chaque voisin j, l'arbre de diffusion transmis est le sous-arbre enraciné en j. La connaissance des autres sous-arbres est en effet inutile.

Dans la suite, nous exploitons la diffusion conjointement avec la méthode active. Cependant, elle peut être utilisée avec la méthode passive pour mettre à jour les ensembles locaux des nœuds et ainsi améliorer l'efficacité lorsque le nombre de pannes est important. Lorsque le jeton doit être régénéré, il est basé sur une connaissance plus récente. De même, la méthode hybride peut aussi être utilisée. Dans ce cas, le passage entre la méthode passive et la méthode active peut être plus rapide.

Diffusion avec retour ($\mathcal{D}_r$)

Lors de la diffusion le long de l'arbre $\mathcal{A}_D$, les ensembles des états des tâches contenus dans les messages de la diffusion sont mis à jour au fur et à mesure qu'ils traversent les nœuds. Cependant, si un nœud est à une profondeur importante dans l'arbre, son ensemble est mis à jour à partir d'un plus grand nombre de nœuds mais n'est diffusé qu'à un nombre limité de nœuds (à aucun si le nœud est une feuille de $\mathcal{A}_D$). Sur la figure 4.15, l'ensemble $\mathcal{E}_5$ est mis à jour à l'aide des ensembles $\mathcal{E}_0$, $\mathcal{E}_2$, $\mathcal{E}_3$ et ${\mathcal{E}_4}^3$, mais n'est diffusé à aucun nœud. De même, l'ensemble $\mathcal{E}_0$ n'est pas mis à jour.

Afin d'améliorer la mise-à-jour de l'ensemble des tâches des nœuds, nous proposons que les nœuds transmettent leur ensemble de tâches à leur père dans l'arbre. Ainsi, l'émetteur de la diffusion reçoit les ensembles de tous les nœuds de la grille et possède, après la diffusion et le retour, une vision globale des états des tâches. Pour cela, nous avons besoin d'un arbre de retour, noté $\mathcal{A}_R$, permettant à tous les nœuds de contacter l'initiateur de la diffusion. Cet arbre doit être communiqué en même temps que la diffusion, excepté si le graphe de communication est non-orienté. Dans ce cas, $\mathcal{A}_D$ est aussi utilisé pour le retour.

Dans le pire des cas, la diffusion produit $n-1$ messages. Par contre, le retour peut induire un nombre important de messages si chaque nœud doit transmettre l'ensemble de chacun de ces fils à son père. Nous proposons donc que chaque nœud attende les ensembles des états des tâches de chacun de ses fils avant de transmettre son ensemble mis à jour à son père. Nous limitons ainsi à $n-1$ le nombre de messages produits par le retour. Ce principe a été énoncé dans [FGL93] sous la forme de *vagues récursives distribuées*.

Plusieurs diffusions peuvent se croiser dans le réseau si le taux de rafraîchissement est important. Chacune doit donc être identifiée. Nous ajoutons un compteur de diffusions dans le jeton qui est initialisé à 0 à la création du jeton et incrémenté à chaque diffusion. L'identifiant, noté id_D, formé de l'identité du jeton id_J et de la valeur du compteur de diffusion, permet d'obtenir un identifiant unique pour chaque diffusion.

Nous ajoutons sur chaque nœud deux ensembles *attente* et *retour*. L'ensemble *attente* contient des couples (id_D, j) et permet de mettre en relation les identités des fils j desquels le nœud attend la réception de leur ensemble de tâches et l'identité de la diffusion id_D correspondante. L'ensemble *retour* contient des couples $(id_D, \text{père})$ et permet de déterminer à quel nœud le nœud courant doit envoyer son ensemble de tâches une fois celui de tous ses fils reçus.

Nous redéfinissons les messages de diffusion par le tuple $M_D = \{id_D, id_J, \mathcal{E}_M, \mathcal{A}_D, \mathcal{A}_R\}$ et les messages de retour par le tuple $M_R = \{id_D, id_J, \mathcal{E}_M, \mathcal{A}_R\}$. L'algorithme 15 montre les différentes actions réalisées lors de la réception d'un message de diffusion.

[3] Les différents ensembles correspondent à des temps d'exécution différents.

Algorithme 15 Réception d'un message $M_D = \{idD, idJ, \mathcal{E}_M, \mathcal{A}_D, \mathcal{A}_R\}$ sur un nœud i.

Si M_D est valide **Alors**
  Mise-à-jour de $\mathcal{E}_i$ et $\mathcal{E}_M$
  Si $\mathit{fils}(\mathcal{A}_D, i) \neq \emptyset$ **Alors**
    Pour tout $j \in \mathit{fils}(\mathcal{A}_D, i)$ **Faire**
      Envoyer message $\{id_D, id_J, \mathcal{E}_M, \mathcal{A}_D, \mathcal{A}_R\}$ à j
      $\mathit{attente} \leftarrow \mathit{attente} \cup \{(id_D, j)\}$
    Fin Pour
    $\mathit{retour} \leftarrow \mathit{retour} \cup \{(id_D, \mathit{père}(\mathcal{A}_D, i))\}$
  Sinon
    Envoyer message $M_R = \{id_D, id_J, \mathcal{E}_M, \mathcal{A}_R)\}$ à $\mathit{père}(\mathcal{A}_D, i)$
  Fin Si
Fin Si

Afin d'éviter l'attente de messages infinie de ses fils en cas de panne, nous plaçons un compte-à-rebours sur les nœuds pour chaque diffusion reçue. Lorsque le compte-à-rebours se termine, tous les couples correspondant à la diffusion sont supprimés de l'ensemble *attente* puis le nœud envoie le message de retour à son père. Si des messages invalides sont reçus de ses fils ultérieurement, ils sont ignorés. L'algorithme 16 est exécuté à chaque réception de message de retour.

Algorithme 16 Réception d'un message $M_R = \{id_D, id_J, \mathcal{E}_M, \mathcal{A}_R\}$ du nœud j sur un nœud i.

Si M_R est valide **Alors**
  Mise-à-jour de $\mathcal{E}_i$
  $\mathit{attente} \leftarrow \mathit{attente} \backslash \{(id_D, j)\}$
  Si *attente* ne contient plus de couple correspondant à id_D **Alors**
    Sélectionner le couple $(id, \mathit{père})$ dans *retour* tel que $id = id_D$
    $\mathit{retour} \leftarrow \mathit{retour} \backslash \{(id, \mathit{père})\}$
    Envoyer à *père* le message $\{id_D, \mathcal{E}_i, \mathcal{A}_R\}$
  Fin Si
Fin Si

L'utilisation de la méthode $\mathcal{D}_r$ peut aussi trouver une application dans le cadre de la gestion des pannes. Nous avons vu en effet, que lorsqu'un nœud est en cours de calcul d'une tâche et qu'il tombe en panne, cela peut produire une tâche zombie. Or, si les identités des nœuds qui calculent une tâche sont mémorisées en plus de l'état de la tâche, il est possible de déterminer si la tâche est une tâche zombie. La réplication volontaire devient alors inutile et la puissance de calcul des nœuds est sauvegardée.

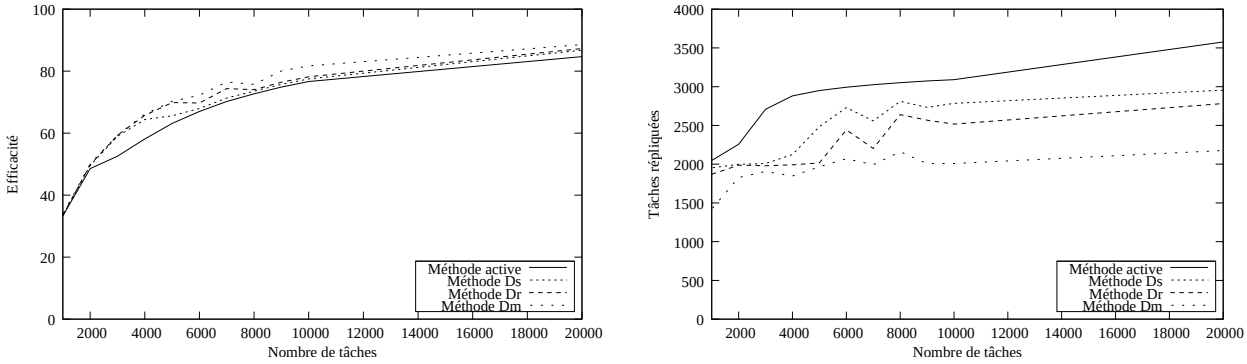
Diffusion avec retour et diffusion ($\mathcal{D}_m$)

La diffusion avec retour permet de mettre à jour l'ensemble des états des tâches de l'initiateur à partir de ceux de tous les nœuds de la grille. Cependant, les nœuds situés sur les feuilles de l'arbre de diffusion sont mis à jour uniquement par ceux de tous les nœuds situés de la racine jusqu'à eux. Plus la largeur de l'arbre est importante, moins il y a de nœuds qui sont mis à

jour. Aussi, nous proposons de diffuser à nouveau l'ensemble de l'initiateur une fois mis à jour. Il possède la connaissance la plus récente des états des tâches. Cette diffusion est réalisée de la même manière que pour la première diffusion : l'ensemble des tâches est diffusé le long du même arbre.

Simulations

Nous avons réalisé des simulations des différentes méthodes basées sur les diffusions sur des réseaux aléatoires de 1000 nœuds avec les paramètres $c_R = 1000$ et $m_R = 1500$ (ces valeurs ayant été choisies arbitrairement à partir des observations sur la méthode active). Pour voir l'impact des diffusions par rapport au nombre de tâches, nous utilisons des ensembles $\mathcal{T}_{\sigma=0.1}$ dans lesquels nous faisons varier le nombre de tâches de 1000 à 20000. Les résultats sont représentés sur la figure 4.16. La figure (a) présente l'évolution des efficacités des différentes solutions de vagues comparées à celle de la méthode active. Nous remarquons que les efficacités sont légèrement améliorées (près de 4%). La figure (b) présente l'évolution du nombre de tâches répliquées. Nous observons une large baisse avec près de 40% de tâches répliquées en moins pour la méthode $\mathcal{D}_m$ par rapport à la méthode active.

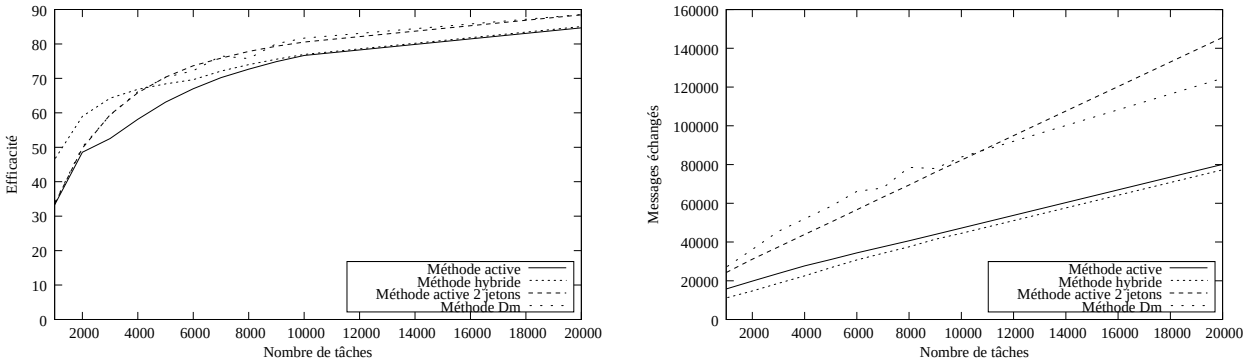


(a) (b)

FIG. 4.16 – Évolution de l'efficacité des méthodes basées sur les diffusions en fonction du nombre de tâches (a) et comparaison du nombre de tâches répliquées (b). Les simulations sont exécutées sur des réseaux aléatoires de 1000 nœuds avec des ensembles $\mathcal{T}_{\sigma=0.1}$, avec $c_R = 1000$ et $m_R = 1500$.

4.4.4 Comparaison entre les méthodes d'optimisation

La figure 4.17 (a) résume l'efficacité des différentes méthodes d'optimisation de la méthode active en fonction du nombre de tâches et la figure 4.17 (b) présente le nombre de messages générés (nombre de jetons émis et messages des différentes vagues).



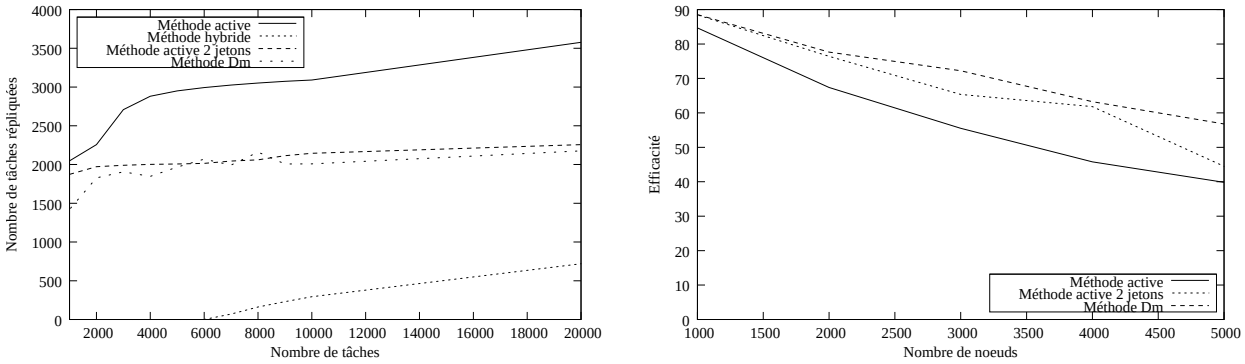
(*a*) (*b*)

FIG. 4.17 – Évolution de l'efficacité des méthodes d'optimisation (*a*) et comparaison du nombre de messages générés (*b*) en fonction du nombre de tâches à calculer. Les simulations sont exécutées sur des réseaux aléatoires de 1000 nœuds avec des ensembles $\mathcal{T}_{\sigma=0.1}$, avec $c_R = 1000$ et $m_R = 1500$.

La méthode hybride propose la plus faible amélioration de l'efficacité. Elle est très légèrement supérieure à celle de la méthode active, ce qui produit moins de messages échangés (en effet, le nombre de messages correspond à chaque envoi du jeton et ce nombre dépend du temps total d'exécution). Par contre, cette méthode génère moins de tâches répliquées que les autres méthodes comme observé sur la figure 4.18 (*a*). Elle est donc particulièrement adaptée pour réduire l'utilisation de la puissance de calcul des nœuds.

La méthode $\mathcal{D}_m$ et la méthode à base de 2 jetons ont sensiblement la même efficacité quel que soit le nombre de tâches et produisent sensiblement le même nombre de tâches répliquées. Nous observons cependant, que le nombre de messages produits est nettement supérieur avec la méthode avec 2 jetons. Il est possible d'améliorer encore l'efficacité en augmentant le nombre de jetons et en réduisant les paramètres c_R et m_R, mais cela produit beaucoup plus de messages. Le choix de l'une ou l'autre méthode dépend donc du réseau de communication. S'il est nécessaire de limiter le débit, la méthode $\mathcal{D}_m$ est préférable.

La figure 4.18 (*b*) présente les résultats des différentes méthodes d'optimisation en fonction du nombre de nœuds dans le réseau. Nous observons que la méthode à base de 2 jetons possède une efficacité identique à la méthode $\mathcal{D}_m$ lorsque le nombre de nœuds est plus faible. Lorsque nous augmentons le nombre de nœuds, l'efficacité diminue beaucoup plus rapidement. Nous avons vu dans le chapitre 1.4 que le temps de couverture des marches aléatoires augmente lorsque le réseau possède plus de nœuds. La méthode à base de 2 jetons est donc beaucoup plus sensible à l'augmentation du nombre de nœuds et supporte moins le passage à l'échelle que les méthodes à base de diffusions.



(*a*) (*b*)

FIG. 4.18 – Évolution du nombre de tâches répliquées en fonction du nombre de tâches (*a*) et évolution de l'efficacité en fonction du nombre de nœuds (*b*). Les simulations sont exécutées sur des réseaux aléatoires de 1000 nœuds avec des ensembles $\mathcal{T}_{\sigma=0.1}$, avec $c_R = 1000$ et $m_R = 1500$.

4.5 Conclusion

Nous avons proposé dans ce chapitre une solution pour la gestion des tâches basée sur une marche aléatoire destinée aux réseaux dynamiques. L'assignation des tâches est laissée à la charge des nœuds. En fonction de leurs caractéristiques et de leur état courant, les nœuds sont en mesure de prendre une décision sans recours à un serveur externe. Le jeton permet en effet de diffuser les paramètres des tâches ainsi que leur état courant d'avancement.

Nous avons détaillé deux méthodes d'assignation que nous avons appelées méthodes passive et active. La première consiste à attendre le jeton avant de sélectionner une nouvelle tâche alors que dans la seconde, les nœuds sélectionnent une tâche aléatoirement parmi celles restant à calculer. Nous avons observé à partir de plusieurs séries de simulations que la méthode passive propose de meilleurs résultats lorsque le nombre de tâches est trop faible par rapport au nombre de nœuds. Dans les autres cas, la méthode active est plus efficace.

Afin d'améliorer l'efficacité de ces deux méthodes, nous avons proposé plusieurs optimisations. La première est basée sur l'utilisation de plusieurs jetons. Nous avons vu que l'efficacité des deux méthodes est améliorée mais pour un nombre important de messages générés. La seconde optimisation est une méthode hybride entre les deux méthodes qui consiste à passer de l'une à l'autre en fonction du nombre de tâches et du nombre de nœuds. Elle permet un réglage plus fin que celle à base de plusieurs jetons et apporte une efficacité meilleure que celle des deux méthodes passive et active quel que soit le nombre de tâches. Enfin, nous avons proposé trois méthodes basées sur la diffusion des états des tâches dans le réseau : une diffusion simple, une diffusion suivie d'une remontée et une diffusion suivie d'une remontée suivie par une nouvelle diffusion. Elles exploitent des arbres aléatoires générés à partir d'un mot circulant, les états des tâches étant diffusés le long de ces arbres à intervalles variables. Ces méthodes induisent un nombre important de messages supplémentaires mais plus faible que la méthode à base de plusieurs jetons. De plus, elles apportent une meilleure efficacité et permettent de diminuer nettement le nombre de tâches répliquées. Si la gestion des ressources utilisée dans l'application est celle que nous avons proposée dans le chapitre 3, un seul mot circulant est nécessaire. Il est

alors utilisé à la fois pour récolter les identités dans le réseau et pour la construction des arbres de diffusion. Il n'y a donc pas de surcoût de gestion pour cette méthode de gestion des tâches et un seul jeton est nécessaire.

Chapitre 5

Optimisations d'applications à base de marches aléatoires et de mot circulant

Résumé : *Dans les chapitres précédents, nous avons proposé différentes solutions basées sur les marches aléatoires et le mot circulant. Elles supportent un dynamisme important du réseau tout en produisant un nombre très faible de messages de contrôle. Nous proposons dans ce chapitre différentes optimisations indépendantes des applications précédentes. Ces différents travaux sont actuellement en cours de réalisation.*

Le test de cohérence local détecte et corrige les incohérences topologiques contenues dans un mot circulant. Cependant, l'algorithme proposé dans [BBF04a] implique la suppression d'un nombre d'identités important. Nous proposons de l'optimiser en nous basant sur la structure de l'arbre couvrant. D'autre part, nous exhibons deux méthodes pour augmenter le taux de couverture du mot circulant en guidant la marche aléatoire vers les nœuds inexplorés ou en ajoutant systématiquement les voisins inexplorés. De même, l'arbre construit est aléatoire et nous avons observé que sa profondeur moyenne est importante. Nous proposons de la contrôler en modifiant la gestion du contenu du mot circulant. Cette partie est réalisée en collaboration avec le Professeur Felber de l'Université de Neuchâtel (Suisse).

Lorsque la taille du réseau devient trop importante, l'efficacité des applications à base de marches aléatoires diminue, ce qui s'explique par l'augmentation du temps de couverture du jeton. Nous proposons une méthode qui consiste à diviser le réseau en sous-parties et à cantonner des marches aléatoires dans chacune d'entre elles.

5.1 Introduction

Les différentes solutions proposées dans les chapitres précédents exploitent la circulation d'une marche aléatoire et le contenu d'un mot circulant. Nous proposons dans ce chapitre d'optimiser ces solutions en modifiant l'algorithme de circulation ainsi que les algorithmes utilisés pour gérer le contenu du mot.

La gestion du mot circulant permet de réduire sa taille et de corriger les incohérences topologiques. Lorsque le nombre de pannes est important dans le réseau, la méthode de réduction des incohérences topologiques a donc une importance cruciale sur le taux de couverture du mot circulant. Nous avons vu que la méthode proposée dans [BBF04a] peut induire un nombre

important d'identités supprimées lorsqu'une incohérence topologique est rencontrée. Nous cherchons à optimiser la réduction en nous basant sur l'arbre construit au sein du mot. Lorsque nous supprimons des identités, nous observons en parallèle l'arbre couvrant afin d'élaguer le moins de branches possible.

L'efficacité des solutions qui exploitent les marches aléatoires et le mot circulant, dépend de la rapidité de couverture de la marche aléatoire qui influence directement le taux de couverture du mot circulant. Or, le temps de couverture d'un réseau par une marche aléatoire est borné en $O(n^3)$. Lorsque le nombre de nœuds dans le réseau augmente, nous constatons donc une baisse de l'efficacité de nos solutions. Pour augmenter le taux de couverture du mot circulant, nous distinguons deux options principales : (1) influencer le déplacement de la marche ou (2) modifier la méthode d'ajout des identités dans le mot circulant. Pour l'option (1), il s'agit de diriger la marche aléatoire vers les nœuds inexplorés. Pour éviter de produire d'autres messages que ceux générés par la circulation du jeton, ce guidage doit être effectué uniquement avec les informations locales aux nœuds. Pour l'option (2), l'ajout des identités dans le mot n'est réalisé que lorsque le mot circulant arrive sur un nœud. Comme pour le cas précédent, nous pouvons compléter les informations dans le mot à l'aide des données locales aux nœuds.

Le mot circulant permet de créer des arbres couvrants adaptatifs et aléatoires, de manière totalement décentralisée. Ces arbres sont utilisés pour rechercher des ressources, par exemple, ou pour diffuser les états des tâches. La forme de l'arbre peut donc influencer l'efficacité des solutions qui l'exploitent. En particulier, une profondeur importante dans l'arbre induit une certaine latence dans la diffusion. Nous cherchons donc à limiter cette profondeur en modifiant la méthode d'ajout des identités. Nous n'ajoutons plus les identités en tête du mot mais à une place plus adaptée pour construire un arbre équilibré.

Le contenu du mot circulant est une image partielle du graphe de communication. La taille maximale de cette image dépend de la taille du réseau. Pour limiter les coûts de communication lors de la circulation du jeton, nous cherchons à réduire le nombre d'informations dans le mot circulant. Pour cela, nous choisissons de diviser le réseau en partitions. Le mot circulant ne contient alors que l'image de sa partition au lieu du réseau complet. Nous obtenons ainsi des partitions au sein desquelles une marche aléatoire circule. La taille des partitions étant bornée, le temps de couverture de la marche est donc lui aussi réduit.

Le plan de ce chapitre est le suivant. Dans la section 5.2, nous exposons des optimisations sur le test de cohérence local d'un mot circulant. Nous présentons deux méthodes basées sur l'exploitation de l'arbre couvrant. Dans la section 5.3, nous proposons deux méthodes pour augmenter le taux de couverture du mot circulant en guidant la circulation du jeton et en modifiant la méthode d'ajout des identités dans le mot. La section 5.4 détaille les algorithmes qui sont utilisés pour équilibrer les arbres construits dans le mot circulant. Enfin, la dernière section expose la méthode basée sur le partitionnement du réseau.

5.2 Optimisation du test de cohérence local

Si le mot circulant passe successivement sur un nœud j puis un nœud k, il contient le sous-mot $< W_i, W_{i+1} >$ avec $W_i = k$ et $W_{i+1} = j$. Or, si le nœud j tombe en panne ou que le lien (j, k) disparaît, le mot contient une incohérence topologique qui induit la construction d'arbres couvrants erronés. Le test de cohérence local proposé dans [BBF04a] corrige ces incohérences étape-par-étape, uniquement à l'aide des données locales des nœuds. Elles sont détectées en comparant le contenu du mot avec le voisinage du nœud et le sous-mot incriminé est supprimé. Ici, l'incohérence avec le lien $< W_i, W_{i+1} >$ est détectée sur le nœud W_i et le sous-mot $droite(W, i + 1)$ est supprimé. Cependant, cet algorithme entraîne un nombre important d'identités supprimées.

Exemple 5.1 *La figure 5.1 montre un exemple de circulation d'un mot circulant dans un graphe. Le nœud 0 reçoit le mot* $W =< 0, 3, 0, 2, 1, 5, 4 >$ *et l'envoie au nœud 1. Or, le lien* $(0, 1)$ *disparaît aussitôt après l'envoi. Le nœud 1 met à jour le mot, en ajoutant son identité en tête, qui devient* $W =< 1, 0, 3, 0, 2, 1, 5, 4 >$*. Il détecte ensuite une incohérence topologique et le sous-mot* $< 0, 3, 0, 2, 1, 5, 4 >$ *est supprimé conformément à l'algorithme proposé dans [BBF04a]. Le mot est alors réduit à* $< 1 >$ *et toutes les autres informations collectées précédemment sont perdues.*

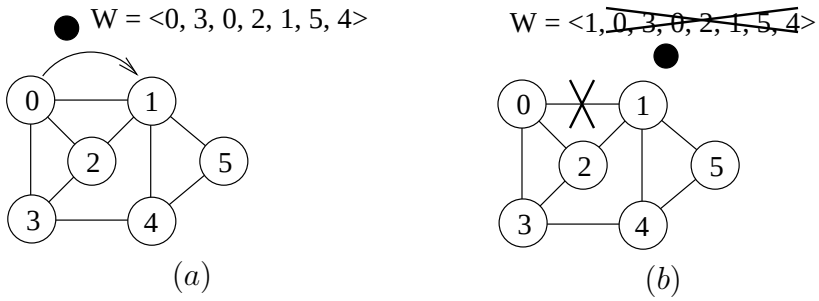


FIG. 5.1 – Exemple de correction d'incohérences topologiques basée sur [BBF04a].

Nous proposons deux méthodes originales qui diminuent le nombre d'identités supprimées lors de la détection d'incohérences topologiques, basées sur la structure de l'arbre contenu dans le mot.

5.2.1 Suppression de sous-arbres

La gestion du mot circulant telle qu'elle a été proposée dans [Fla01, BBF04a] a pour but de construire et de maintenir un arbre couvrant, enraciné sur le nœud courant après mise-à-jour. D'après l'exemple 5.1, nous remarquons que la suppression brutale du sous-mot peut être limitée à la partie concernée dans l'arbre correspondant. En effet, seul le sous-arbre enraciné en 0 pose un problème de cohérence. Ce nœud est considéré comme un voisin de 1 mais n'appartient plus à son voisinage qui est $Vois_1 = \{2, 4, 5\}$. Si nous supprimons uniquement ce sous-arbre, nous pouvons ainsi limiter le nombre d'identités supprimées et conserver des informations topologiques. Dans la suite, nous appelons cette méthode $\mathcal{M}_{\mathcal{S}}$.

Exemple 5.2 *La figure 5.2 présente un exemple de la méthode $\mathcal{M}_{\mathcal{S}}$ en reprenant le scénario de l'exemple 5.1. Dans la technique de réduction présentée dans [BBF04a], le mot devrait être, après mise-à-jour et réduction, $W =< 1 >$. Si nous ne supprimons que le sous-arbre enraciné en 0, le mot devient $W =< 1, 5, 4 >$.*

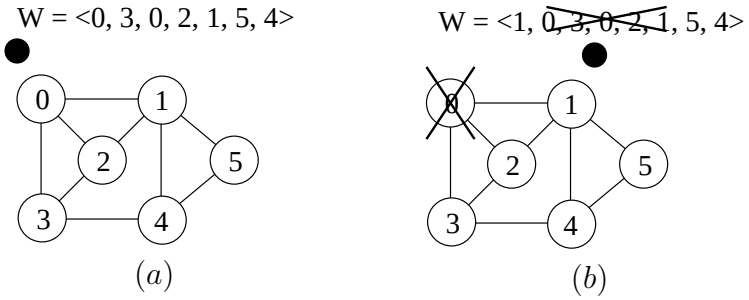


FIG. 5.2 – Exemple de correction d'incohérences topologiques avec la méthode $\mathcal{M}_{\mathcal{S}}$.

L'algorithme 17 décrit le test de cohérence local basé sur cette méthode. Le mot est parcouru de gauche à droite et pour chaque occurrence du nœud courant, l'identité située à droite doit être constructive (elle existe avant la position courante) ou doit appartenir au voisinage du nœud courant. Si ce n'est pas le cas, les identités à droite sont supprimées jusqu'à trouver une identité qui a déjà été rencontrée (*i.e.* qui ne fait pas partie du sous-arbre enraciné en 0). Un ensemble *visités* est donc mis à jour lors du parcourt du mot, chaque identité valide y est ajoutée au fur-et-à-mesure. Contrairement à l'algorithme décrit dans [BBF04a] qui s'arrête dès qu'une incohérence topologique est détectée, l'algorithme 17 doit parcourir l'ensemble du mot, c'est-à-dire au plus $2.n - 1$ identités, quel que soit le nombre d'incohérences détectées dans le mot.

Algorithme 17 Test de cohérence local du mot W basé sur la méthode $\mathcal{M}_{\mathcal{S}}$ sur le nœud i

$k \leftarrow 1$
visités $\leftarrow \{W_1\}$
Tant que $k < taille(W)$ **Faire**
  Si $(W_k = i) \wedge (k < taille(W)) \wedge (W_{k+1} \notin Vois_k) \wedge (W_{k+1} \notin visités)$ **Alors**
    Tant que $(k < taille(W)) \wedge ((W_{k+1} = W_k) \vee (W_{k+1} \notin visités))$ **Faire**
      $supprimer(W, k+1)$
    Fin Tant que
  Fin Si
  visités $\leftarrow$ *visités* $\cup \{W_k\}$
  $k \leftarrow k + 1$
Fin Tant que
Réduction du mot circulant (voir algorithme 3)

5.2.2 Reconstruction de l'arbre

Les seules informations ajoutées dans un mot circulant correspondent aux mouvements du jeton : si un nœud i reçoit le jeton de j, alors seul ce lien est éventuellement ajouté. Lors de la

réduction des incohérences topologiques, nous remarquons que certaines branches du sous-arbre supprimé peuvent être enracinées sur le nœud courant. Ainsi, des occurrences sont préservées. Cette optimisation ajoute des informations qui ne sont plus basées sur la circulation du mot circulant. Dans la suite, nous appelons cette méthode $\mathcal{M}_{\mathcal{R}}$.

Exemple 5.3 *À partir de l'exemple précédent, nous avons le mot* $W =< 1, 0, 3, 0, 2, 1, 5, 4 >$ *qui contient l'incohérence topologique* $< 1, 0 >$. *À l'aide du nouvel algorithme, il est réduit par* $W =< 1, 2, 1, 5, 4 >$ *comme le montre la figure 5.3. En effet,* $2 \in \text{Vois}_1$ *donc 2 devient le fils du nœud 1. Seule l'identité 3 disparaît, la connaissance locale du nœud 1 étant insuffisante pour la conserver dans le mot.*

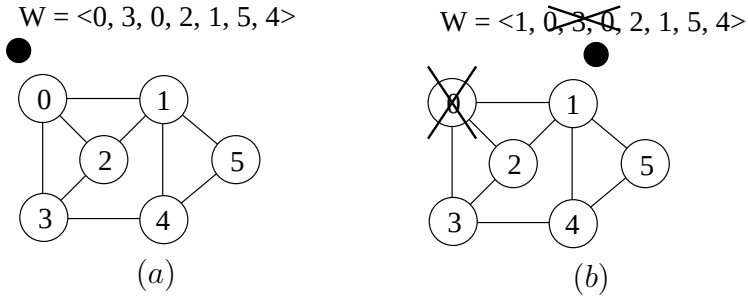


FIG. 5.3 – Illustration de la méthode $\mathcal{M}_{\mathcal{R}}$: l'identité 2 est rattachée au nœud 1 alors que le jeton n'est pas passé par le lien $(2, 1)$.

L'algorithme 18 décrit la méthode $\mathcal{M}_{\mathcal{R}}$. Nous remarquons que sa complexité est identique à celle de l'algorithme 17.

Algorithme 18 Test de cohérence local avec la méthode $\mathcal{M}_{\mathcal{R}}$ sur le nœud i

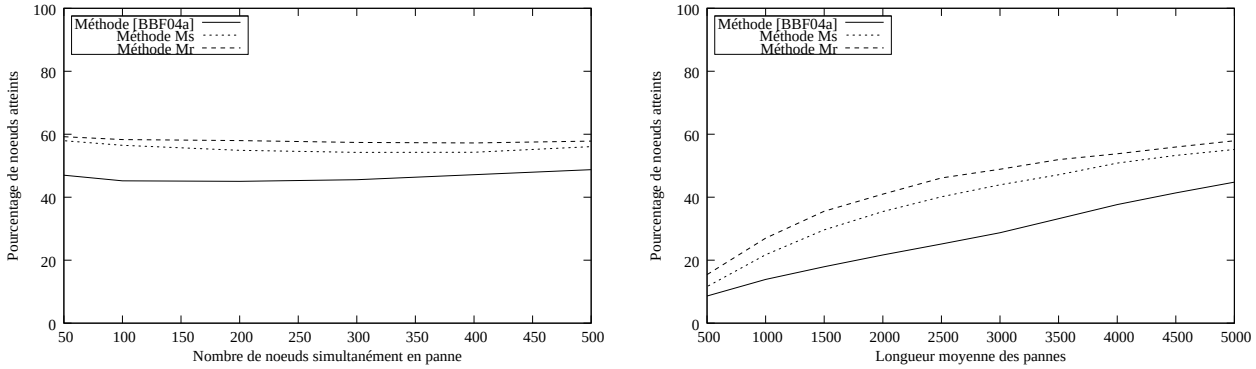
$k \leftarrow 1$
visités $\leftarrow \{W_1\}$
Tant que $k < taille(W)$ **Faire**
  Si $W_k = i$ **Alors**
    Tant que $(k < taille(W)) \wedge (((W_{k+1} = W_k) \vee ((W_{k+1} \notin visités) \wedge (W_{k+1} \notin Vois_k))))$
    Faire
      $supprimer(W, k + 1)$
    Fin Tant que
  Fin Si
  visités $\leftarrow$ *visités* $\cup \{W_k\}$
  $k \leftarrow k + 1$
Fin Tant que
Réduction du mot circulant (voir algorithme 3)

5.2.3 Simulations des méthodes $\mathcal{M}_{\mathcal{S}}$ et $\mathcal{M}_{\mathcal{R}}$

Nous réalisons deux séries de simulations pour observer le comportement des deux méthodes d'optimisation. Nous utilisons des graphes aléatoires de 1000 nœuds sur lesquels nous appliquons un modèle de pannes fixe. La première série consiste à modifier le nombre de nœuds

simultanément en panne (de 0 à 500) pour une durée moyenne des pannes fixée. Nous calculons le pourcentage de nœuds atteints à l'aide du contenu du mot circulant en utilisant la technique décrite dans la section 2.4. Nous obtenons les résultats représentés sur la figure 5.4 (a). Nous observons dans un premier temps que les deux méthodes apportent une nette amélioration de l'efficacité par rapport à la méthode [BBF04a] (plus de 10% de nœuds atteints en plus). Cependant, la différence entre la méthode $\mathcal{M}_{\mathcal{S}}$ et la méthode $\mathcal{M}_{\mathcal{R}}$ est très faible (un peu plus de 1%).

Pour la seconde série de simulations, nous fixons le nombre de nœuds simultanément en panne à 200 et nous modifions la durée moyenne des pannes (de 500 à 5000 étapes). Pour des pannes très fréquentes, l'efficacité est très faible pour les trois méthodes. Par contre, lorsque la durée moyenne augmente, nous observons une très nette amélioration pour les deux méthodes optimisées. Mais la méthode $\mathcal{M}_{\mathcal{S}}$ présente un taux de nœuds atteints légèrement inférieur (environ 5%), une différence qui est plus marquée lorsque les pannes sont fréquentes.



(a) (b)

FIG. 5.4 – Comparaison du taux de couverture du mot circulant en fonction de la méthode du test de cohérence local et du nombre de nœuds simultanément en panne dans le réseau (a) et en fonction de la longueur moyenne des pannes (b).

5.3 Augmentation du taux de couverture du mot circulant

Nous proposons d'accélérer la vitesse à laquelle un mot circulant collecte toutes les identités du réseau. Nous détaillons deux méthodes, l'une basée sur le guidage de la marche aléatoire en fonction des identités contenues dans le mot (appelée méthode $\mathcal{M}_{\mathcal{G}}$) et l'une basée sur l'ajout automatique des identités des voisins (appelée méthode $\mathcal{M}_{\mathcal{V}}$).

5.3.1 Guidage de la marche aléatoire

Le mot circulant est utilisé pour maintenir une image partielle du graphe de communication dans le but, notamment, de construire des arbres couvrants. Les identités rencontrées lors de la circulation sont ajoutées au fur et à mesure des déplacements du jeton. Nous proposons d'accélérer la récupération de nouvelles identités en guidant la marche aléatoire vers les nœuds

inexplorés du réseau. Ainsi, lorsqu'un nœud i reçoit le jeton, il compare son voisinage $Vois_i$ avec les identités contenues dans le mot (c'est-à-dire $identités(W)$) :

- $identités(W) \cap Vois_i = \emptyset$: tous les voisins sont contenus dans le mot circulant, le jeton est envoyé à un voisin choisi uniformément aléatoirement ;
- $identités(W) \cap Vois_i \neq \emptyset$: certains voisins sont inexplorés, le jeton est envoyé à l'un d'entre eux choisi uniformément aléatoirement.

Cet algorithme accélère la couverture du réseau en début d'exécution et les nœuds qui se connectent sont agglomérés plus rapidement. D'autre part, dans la section précédente, nous avons vu que des nœuds sont supprimés du mot à la suite de la réduction des incohérences topologiques alors qu'ils sont toujours présents dans le réseau. En guidant la marche, nous pouvons les réintégrer plus rapidement. Nous appelons cette méthode $\mathcal{M}_{\mathcal{G}}$.

En début d'exécution et particulièrement si le graphe est dense, la profondeur de l'arbre obtenu est importante. Le mot est guidé vers les nœuds inexplorés, ce qui crée un arbre proche d'une chaîne. Cependant, lorsque l'image du graphe de communication contient toutes les identités du réseau, le jeton se comporte comme une marche aléatoire. L'arbre évolue alors en fonction des déplacements de la marche et sa profondeur diminue.

5.3.2 Ajout du voisinage

Comme nous l'avons vu dans la section précédente, lorsqu'un nœud reçoit le jeton, il est en mesure de détecter les voisins qui ne sont pas dans le mot. Il peut donc choisir de les ajouter lui-même, sans attendre qu'ils soient visités. N'ayant qu'une connaissance locale du réseau limitée à son voisinage, il ne peut les ajouter qu'en tant que ses propres fils dans l'arbre. Pour chaque voisin, une nouvelle occurrence de l'identité du nœud courant est ajoutée à la fin du mot suivie de celle du voisin. L'algorithme 19 décrit cette procédure exécutée à chaque réception. La taille du mot croît plus rapidement qu'avec la méthode [BBF04a], mais les occurrences ajoutées sont constructives. Aussi la borne maximale donnée dans [Fla01] n'est pas modifiée. Nous appelons cette méthode $\mathcal{M}_{\mathcal{V}}$.

Algorithme 19 Méthode d'ajout $\mathcal{M}_{\mathcal{V}}$ dans un mot W reçu sur le nœud i

```
Ajout i dans W (i.e. W_1 = i)
Test de cohérence local
Pour tout k ∈ Vois_i \ identités(W) Faire
  Si W_{taille(W)} ≠ i Alors
    insérer(W, taille(W) + 1, i)
  Fin Si
  insérer(W, taille(W) + 1, k)
Fin Pour
Envoyer W à j choisi uniformément aléatoirement dans Vois_i
```

Le contenu du mot circulant est maintenu sous la forme d'un arbre. Or, avec cette méthode, les arbres obtenus en début d'exécution sont différents de ceux obtenus avec la méthode [BBF04a], en particulier si le graphe est dense. Dans ce cas, le degré moyen des nœuds dans l'arbre est assez élevé, le nombre de voisins dans le graphe étant important. Lorsque toutes les identités sont présentes dans le mot, les arbres évoluent suivant l'algorithme [BBF04a]. Ainsi,

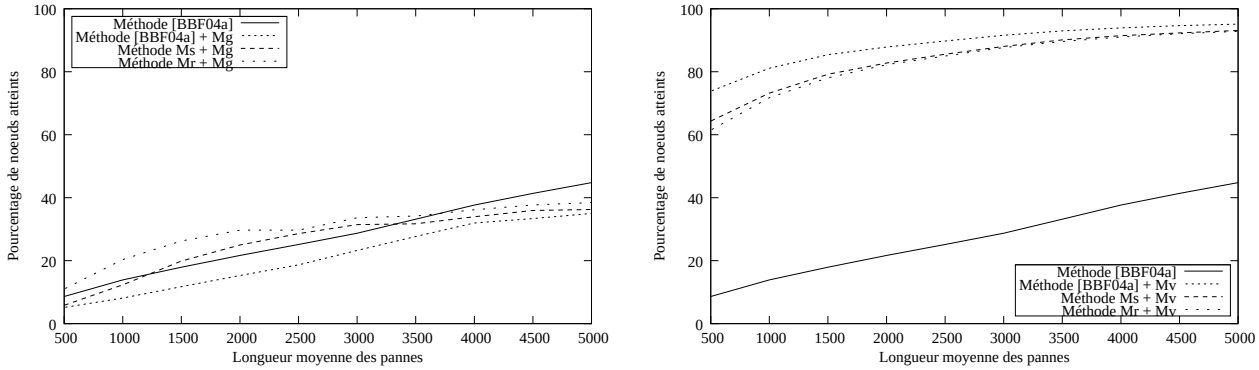
si un nœud possède un degré important dans l'arbre après l'ajout de ses voisins, son degré diminue progressivement grâce à la mise-à-jour du mot à chaque réception.

5.3.3 Comparaisons entre les différentes méthodes

Nous avons réalisé des séries de simulations avec les méthodes $\mathcal{M}_{\mathcal{G}}$ et $\mathcal{M}_{\mathcal{V}}$. Nous avons utilisé des réseaux aléatoires de 1000 nœuds sur lesquels nous avons appliqué un modèle de pannes fixe. Nous avons fait varier la longueur moyenne des pannes tout en fixant le nombre de nœuds en panne à 200. Les résultats obtenus sont représentés sur la figure 5.5. En plus de ces méthodes, nous avons aussi appliqué les différentes méthodes pour le test de cohérence local $\mathcal{M}_{\mathcal{S}}$ et $\mathcal{M}_{\mathcal{R}}$.

Avec le guidage de la marche aléatoire (figure 5.5 (a)), le pourcentage de nœuds atteints est légèrement supérieur à la méthode [BBF04a] lorsque les pannes sont fréquences. Cependant, dans la plupart des cas, la méthode $\mathcal{M}_{\mathcal{G}}$ apporte une efficacité plus faible, même avec les méthodes de réduction optimisées.

La méthode $\mathcal{M}_{\mathcal{V}}$ (figure 5.5 (b)) augmente considérablement le taux de couverture du mot circulant. Par rapport à la méthode [BBF04a], nous obtenons une couverture doublée, quel que soit le taux de pannes appliqué. Par contre, nous observons que la méthode $\mathcal{M}_{\mathcal{V}}$ détériore les performances des méthodes $\mathcal{M}_{\mathcal{S}}$ et $\mathcal{M}_{\mathcal{R}}$. Ainsi, les méthodes d'optimisation du test de cohérence local apportent une efficacité plus faible que celle proposée dans [BBF04a], contrairement à ce que nous observons dans la section 5.2.3.



(a) (b)

FIG. 5.5 – Taux de couverture du mot circulant en fonction de la longueur moyenne des pannes avec le guidage de la marche aléatoire (a) et avec l'ajout des voisins (b).

5.4 Optimisation de l'arbre couvrant

L'arbre couvrant construit dans le mot circulant est aléatoire et ses propriétés (profondeur, degré des nœuds) ne sont pas maîtrisées. Par simulation, nous obtenons une profondeur moyenne de 80 sur des réseaux de 1000 nœuds de degré minimum égal à 5 (voir section 2.4.2). Or, nous utilisons cet arbre pour plusieurs applications : la diffusion des états de tâches, la réinitialisation de compte-à-rebours ou la recherche de ressources. Dans tous les cas, une hauteur importante de

l'arbre entraîne une certaine latence entre le début de la diffusion et la réception des messages par les feuilles de l'arbre. D'autre part, certains nœuds peuvent avoir un degré dans l'arbre important. Lors de la diffusion d'informations, ils doivent émettre un nombre important de messages. Or, suivant la taille des données émises, une saturation du réseau est possible. Nous proposons donc de corriger la gestion du mot dans le but d'obtenir une profondeur plus faible tout en limitant le degré des nœuds dans l'arbre. Nous notons le degré maximum des nœuds d_{max}.

5.4.1 Insertion d'une nouvelle identité

Nous voulons limiter le degré des nœuds dans l'arbre. Or, d'après l'algorithme proposé dans [Fla01], sa racine est modifiée à chaque réception du jeton. Dans ce cas, le degré est difficile à contrôler : si un nœud a un degré adéquat, le placer en tant que racine de l'arbre entraîne une modification de son degré et de celui de son père. L'algorithme que nous proposons ne modifie pas la position des nœuds dans l'arbre sauf si leur degré ou celui de leur père n'est pas correct.

Lorsqu'une nouvelle identité doit être ajoutée, nous choisissons le meilleur père possible dans l'arbre. Ce nœud doit être dans son voisinage et avoir un degré dans l'arbre le plus faible possible. Comme nous n'avons aucune connaissance autre que le voisinage des nœuds, l'insertion de la nouvelle identité peut induire un degré supérieur à d_{max} pour son père. Cependant, l'équilibrage de l'arbre est réalisé étape par étape en fonction des mouvements du jeton. Les nœuds sont déplacés et les degrés incorrects sont corrigés.

Algorithme 20 Insertion d'une nouvelle identité i dans le mot W

Si $taille(W) = 0$ **Alors**
  $Insérer(W, 1, i)$
Sinon
  $degréMin \leftarrow degré(W, W_1)$
  $identitéMin \leftarrow W_1$
  /* *Recherche du meilleur père* */
  Pour tout $j \in identités(W)$ **Faire**
    Si $(degré(W, j) < degréMin) \wedge (j \in Vois_i)$ **Alors**
      $degréMin \leftarrow degré(W, j)$
      $identitéMin \leftarrow j$
    Fin Si
  Fin Pour
  /* *Ajout du lien* */
  Si $W_{taille(W)} \neq identitéMin$ **Alors**
    $insérer(W, taille(W) + 1, identitéMin)$
  Fin Si
  $insérer(W, taille(W) + 1, i)$
Fin Si

L'algorithme 20 décrit l'insertion d'une nouvelle identité i dans le mot. Si le mot est vide, i est simplement ajouté. Sinon, son père j doit répondre aux conditions suivantes :

– $j \in identités(W)$;
– $j \in Vois_i$;
– $\forall l \in identités(W), degré(j) \leq degré(l)$.

Une telle identité existe toujours dans le mot. En effet, le mot est envoyé depuis un voisin de i. Le graphe étant non-orienté, l'émetteur du jeton est l'un des pères possibles de i. Pour faciliter l'insertion de la nouvelle identité, nous l'ajoutons à la fin du mot, précédée de son père. Ainsi, la branche $(i \leftarrow j)$ est ajoutée.

5.4.2 Réduction du degré d'un nœud

Lorsque le jeton est reçu sur un nœud dont l'identité est déjà dans l'arbre, nous cherchons à équilibrer les degrés des nœuds en réduisant ceux supérieur à d_{max}. En effet, avec l'algorithme précédent, lorsqu'une nouvelle identité est ajoutée, elle peut impliquer l'augmentation du degré de son père au-delà de d_{max}. Avec sa connaissance locale, un nœud ne peut déplacer dans le mot que sa propre identité. En particulier, nous choisissons de déplacer une identité si le degré du père est supérieur à d_{max}.

Comme le montre la figure 5.6, lorsqu'un nœud reçoit le mot, il vérifie le degré de son père. Si celui-ci est supérieur à d_{max}, il cherche un nouveau père. Sur cet exemple, le nœud 4 décide de s'enraciner sur le nœud 3. Le nœud 1 voit son degré passer en dessous de d_{max}. Afin de limiter la profondeur de l'arbre, la recherche du nouveau père s'effectue de la racine aux feuilles. Lorsque ce nouveau père est trouvé, l'algorithme 21 est exécuté.

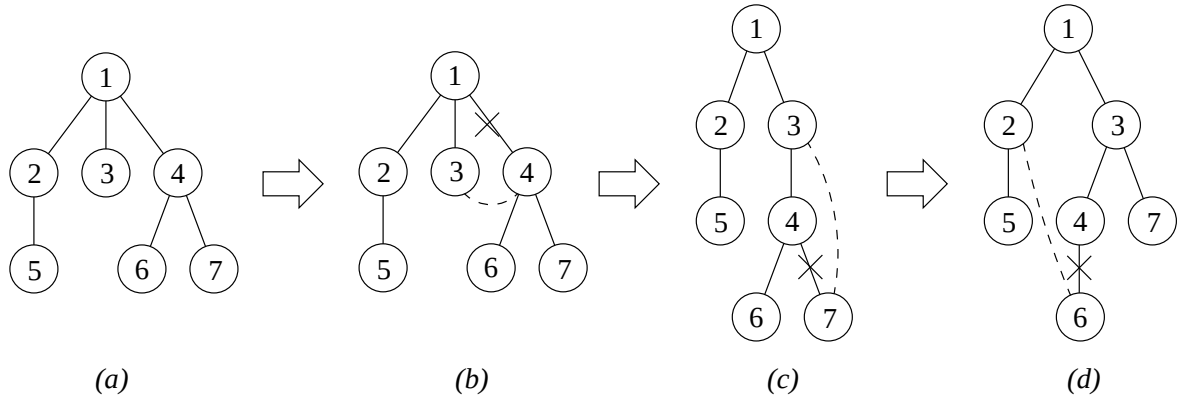


FIG. 5.6 – La figure (a) présente un arbre obtenu avec la circulation du jeton. À la réception sur le nœud 4 (b), celui-ci détecte que son père à un degré supérieur à $d_{max} = 2$. Il cherche un autre père et trouve le nœud 3. Dans le but de réduire la profondeur de l'arbre, le nœud 7 (c) puis le nœud 6 (d) remontent dans l'arbre.

Algorithme 21 Enracine le nœud i sur le nœud r dans le mot W

$enracinés \leftarrow \emptyset$
$visités \leftarrow \emptyset$
$j \leftarrow 1$
Tant que $W_j \neq i$ **Faire**
  $visités \leftarrow visités \cup \{W_j\}$
  $j \leftarrow j + 1$
Fin Tant que
Si $premièreOccurrence(W, W_{j-1}) < j - 1$ **Alors**
  $supprimer(W, j - 1)$
  $j \leftarrow j - 1$
Fin Si
Si $W_{taille(W)} \neq r$ **Alors**
  $insérer(W, taille(W) + 1, r)$
Fin Si
$fin \leftarrow taille(W)$
Tant que $W_j \notin visités$ **Faire**
  $enracinés \leftarrow enracinés \cup \{W_j\}$
  $déplacer(W, j, taille(W) + 1)$
  $fin \leftarrow fin - 1$
Fin Tant que
Tant que $j < fin$ **Faire**
  Si $W_j \in enracinés$ **Alors**
    $enracinés \leftarrow enracinés \cup \{W_j\}$
    $déplacer(W, j, taille(W) + 1)$
    $fin \leftarrow fin - 1$
  Sinon
    $visités \leftarrow visités \cup \{W_j\}$
    $j \leftarrow j + 1$
  Fin Si
Fin Tant que

Exemple 5.4 *La figure 5.7 montre un exemple de déplacement d'un nœud au sein d'un mot circulant. L'arbre obtenu avec le mot* $W =< 1, 2, 5, 2, 8, 1, 4, 6, 1, 9, 7, 4, 3 >$ *y est représenté, ainsi que les transformations successives du mot. Le nœud 4 reçoit le jeton et détecte que son père, le nœud 1, a un degré supérieur à* $d_{max} = 2$*. Or, le nœud 9 peut devenir son père (*i.e. $9 \in \text{Vois}_4$*) donc l'algorithme 21 ajoute dans un premier temps l'identité 9 à la fin du mot (étape 2). Comme l'identité qui précède la première occurrence de 4 existe avant cette position, elle est supprimée (étape 3). Ensuite, la branche enracinée en 4 est déplacée à la fin du mot (étape 4). Enfin, l'algorithme doit parcourir le reste du mot afin de déplacer toutes les branches correspondantes au sous-arbre enraciné en 4 (étape 5).*

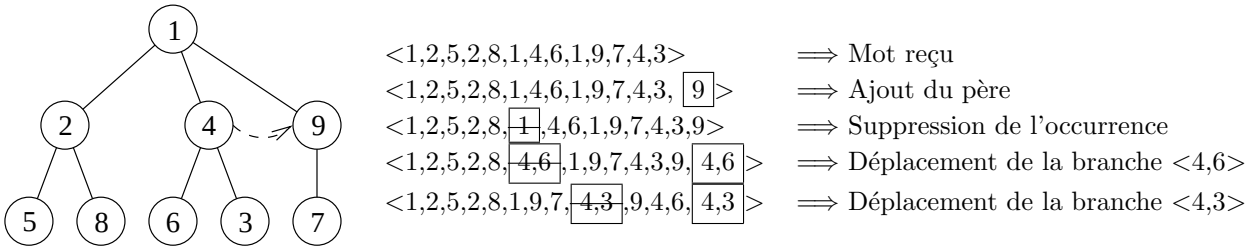


FIG. 5.7 – Étapes successives de déplacement d'un nœud au sein d'un mot circulant.

Remarque 5.1 *Lorsqu'une identité est déplacée dans le mot, le sous-arbre enraciné en elle est déplacé à la fin du mot afin d'éviter d'ajouter des mauvaises informations. L'algorithme 21 ne parcourt qu'une seule fois le mot pour réaliser cette opération.*

5.4.3 Réduction de la profondeur d'un nœud

Avec l'algorithme 21, le degré des nœuds est limité à d_{max}, mais la profondeur de l'arbre peut être grande. Pour la réduire, lorsqu'un nœud i reçoit le jeton, il cherche à remonter son identité dans l'arbre. Il recherche un nouveau père j qui répond aux conditions suivantes :

- $j \in \mathit{identités}(W)$;
- $j \in \mathit{Vois}_i$;
- $j \notin \mathit{branche}(\mathit{arbre}(W), i)$;
- $\mathit{profondeur}(W, j) < \mathit{profondeur}(W, \mathit{père}(i))$;
- $\mathit{degré}(W, j) < d_{max}$.

Si un tel nœud j existe, alors le nœud i y est enraciné. Ainsi, étape par étape, tous les nœuds sont enracinés sur des pères le plus haut possible dans l'arbre. Sur la figure 5.6 (c) et (d), les nœuds 7 et 6 diminuent leur profondeur. Lorsque le nœud 7 reçoit le jeton, il détecte que le nœud 3 est plus haut dans l'arbre que le nœud 4 et est enraciné dessus. De même, le nœud 6 est enraciné sur le nœud 2.

5.4.4 Gestion des pannes

Cette nouvelle méthode de gestion du mot s'appuie sur les algorithmes tels qu'ils ont été présentés dans la section 1.5. Les différents mécanismes pour gérer les pannes sont identiques. Les incohérences topologiques sont corrigées étape par étape en fonction des déplacements du jeton. Ces réductions peuvent induire l'augmentation du degré des nœuds ou de la profondeur de l'arbre. Mais après un certain nombre d'étapes qui dépend de la circulation ainsi que de la topologie du réseau (degré des nœuds), le mot est modifié.

Cependant, la correction des incohérences topologiques est basée sur un arbre enraciné sur le nœud courant. Ainsi, lorsque le mot est reçu sur un nœud i, nous avons obligatoirement $W_1 = i$. Seuls des sous-arbres enracinés en i peuvent être éventuellement supprimés, i n'ayant pas de père dans le mot. Avec la nouvelle gestion, ce n'est pas le cas : la première occurrence de i est située à une position quelconque dans le mot. Nous devons donc vérifier si son père est donc son voisinage. Si ce n'est pas le cas, le sous-arbre $\mathit{arbre}(W) \backslash \mathit{branche}(\mathit{arbre}(W, W_1), i)$ est invalide et doit être supprimé. Pour limiter la perte d'identités, le nœud i cherche à enraciner sur lui le plus possible de branches de cet arbre.

5.4.5 Applications

Chaque nœud de l'arbre couvrant obtenu au sein du mot circulant possède un degré au plus de k. Ce type d'arbre, appelé Tree^k (voir [BRF04]), a des propriétés intéressantes lorsqu'il est utilisé pour diffuser du contenu dans un réseau à partir d'une source donnée (qui est alors la racine de l'arbre). Si un nœud i doit diffuser un fichier à l'ensemble des nœuds du réseau, il peut utiliser l'arbre couvrant contenu dans un mot circulant. Il envoie ce fichier à un maximum de k voisins pour éviter de saturer sa bande passante de sortie. Lorsqu'un fils reçoit le fichier, il le transfert à ses propres voisins et ainsi de suite. L'arbre étant limité en profondeur, le temps de latence entre la diffusion par la racine et la réception sur les feuilles est ainsi limité.

Exemple 5.5 *Un nœud possède un fichier qu'il désire envoyer à l'ensemble du réseau. Il construit un arbre de diffusion à partir du mot circulant. Nous supposons ici que $d_{max} = 2$. Il envoie le fichier à ses voisins dans l'arbre qui le propagent à leur tour à leurs propres voisins. La figure 5.8 montre les différentes étapes de cette diffusion.*

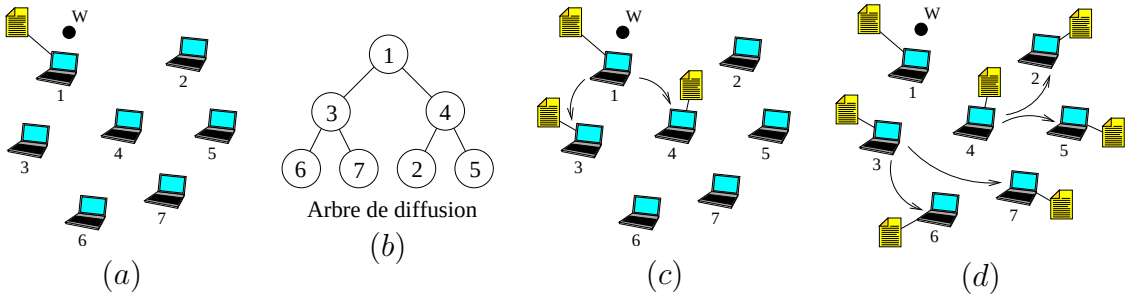


FIG. 5.8 – Le nœud 1 possède un fichier qu'il désire diffuser à l'ensemble des nœuds (a). À l'aide du mot circulant, un arbre de diffusion est construit (b). Le nœud 1 envoie le fichier à ses deux voisins (c) qui le diffuse ensuite à leurs propres voisins (d).

Pour améliorer encore la diffusion, les auteurs de [BCC+07] ont montré que les PTree^k apportent une efficacité accrue par rapport aux Tree^k. Au lieu d'utiliser un seul arbre, plusieurs arbres concurrents sont maintenus ayant tous la même racine. Cependant, la construction de plusieurs arbres différents à partir d'un mot circulant donné est impossible. Comme nous l'avons vu précédemment, le mot ne contient que des informations constructives et chaque nœud ne connaît que son voisinage. Nous avons donc deux solutions principales : (1) utiliser un seul mot circulant ou (2) utiliser autant de mots circulants que d'arbres souhaités. Dans le premier cas, le nœud racine doit attendre de recevoir autant de fois le mot circulant que le nombre d'arbres de diffusion souhaité. Chaque arbre reçu est différent car le contenu du mot circulant est modifié suivant les changements topologiques et à chaque réception sur un nœud. Cependant, le réseau étant dynamique, les arbres ont une durée de validité donnée. Si le nœud maintient j arbres, les arbres les plus anciens risquent d'être invalides. C'est pourquoi dans le cas (2), nous proposons d'utiliser j mot circulants qui se déplacent de manière indépendante. Comme les arbres construits dans les mots sont aléatoires, nous obtenons donc le nombre d'arbres différents voulus.

Ces différentes solutions pour diffuser des données dans un réseau exploitent le contenu du mot circulant. Elles sont donc totalement décentralisées, tolérantes aux pannes et économes en

terme d'échanges de messages. Ce dernier point est important dans ce type d'application. La mise-à-jour des arbres de diffusion ne doit pas utiliser trop de bande passante qui doit être réservée pour la diffusion proprement dite.

5.5 Passage à l'échelle d'applications à base de marches aléatoires

Le temps de couverture d'un graphe par une marche aléatoire est borné en $O(n^3)$. L'idée est de décomposer le réseau en plusieurs partitions dont la taille est comprise dans un intervalle $[m, M]$. Au sein de ces partitions, des marches aléatoires circulent, appelées *marches locales*. Leur zone de couverture est réduite, ce qui diminue leur temps de couverture tout en gardant les propriétés des marches aléatoires. Nous trouvons de nombreux exemples de partitionnement dans la littérature. Toutefois, nous proposons ici une solution originale qui exploite les informations récoltées par un mot circulant. Cette solution est complètement décentralisée, tolérante aux pannes et peut être adaptable aux différentes applications que nous avons proposées précédemment.

5.5.1 Marche aléatoire locale

Nous appelons une *marche aléatoire locale*, une marche qui circule au sein d'une partition donnée. Elle est définie comme un jeton $J = \{id_J, p_J, W_J\}$ où :

- id_J est l'identité du jeton qui permet de tester sa validité (voir section 1.4.5) ;
- p_J est l'identité de la partition à laquelle appartient le jeton ;
- W_J est un mot circulant qui permet de construire l'arbre couvrant $\mathcal{T}_{p_J}$ de la partition p_J.

Tout nœud doit être en mesure de recréer le jeton de sa partition ou éventuellement créer une nouvelle partition s'il n'en appartient à aucune. Dans ce cas, il crée une nouvelle marche et l'identité de la partition est le couple formé de l'identité du nœud et du temps courant. Un nœud ne pouvant pas créer plusieurs marches simultanément et chaque identité étant unique dans le réseau, les partitions ont toutes une identité unique. Les données sur un nœud i sont donc les suivantes :

- le compte-à-rebours T_i pour la régénération du jeton initialisé par t_{ini} qui dépend de M ;
- l'identité de la partition p_i à laquelle appartient le nœud ;
- la copie du dernier mot circulant notée W_i ;
- l'identité du dernier jeton id_i pour tester la validité des jetons reçus et utilisée pour générer l'identité des nouveaux jetons (voir section 1.4.5).

Le nombre de nœuds de chaque partition doit être connu sur chaque nœud pour créer ou fusionner des partitions de manière décentralisée. Cependant, en mémorisant le dernier mot circulant rencontré, cette connaissance globale de la partition peut être exploitée par d'autres applications (pour du routage ou de la recherche de ressources, par exemple). La trace du mot circulant peut aussi être utilisée pour régénérer des nouveaux jetons afin d'accélérer la mise-à-jour de l'arbre couvrant de la partition.

Chaque nœud n'a qu'une connaissance locale du réseau et n'est pas en mesure de déterminer si ses voisins appartiennent à sa partition. Lorsqu'il reçoit le jeton, il l'envoie aléatoirement à

l'un de ses voisins. Nous distinguons alors trois cas différents : 1) le jeton est reçu par un nœud i de sa partition (*i.e.* $p_i = p_J$), 2) le nœud destinataire n'appartient à aucune partition ou 3) le jeton atteint un nœud d'une autre partition.

Le cas 1) est le cas *normal*. La marche aléatoire reste dans sa partition, le mot circulant est donc mis à jour et le jeton est transféré à un voisin. Dans le cas 2), le nœud n'appartient à aucune partition et peut être absorbé dans p_J. Si le nombre de nœuds de p_J dépasse M, p_J peut être divisée en deux parties. Enfin, dans le cas 3), le nœud qui reçoit le jeton possède la connaissance de la partition p_J voisine ainsi que de sa propre partition. Si p_i et p_J sont stables, c'est-à-dire qu'elles possèdent toutes les deux un nombre de nœuds suffisant, le jeton est retourné dans sa partition. Dans le cas contraire, les deux partitions peuvent être fusionnées.

5.5.2 Division d'une partition

La marche aléatoire locale annexe les nœuds qui n'appartiennent à aucune partition. Chaque partition doit contenir un nombre de nœuds compris dans l'intervalle $[m, M]$. Lorsqu'il dépasse la borne maximale M, la partition doit être divisée en deux parties pas nécessairement identiques mais contenant au moins m nœuds (nous avons donc la relation $M > 2 \times m$). Cette condition est nécessaire pour obtenir deux partitions stables.

La division est réalisée à l'aide du contenu du mot circulant. Nous construisons un arbre couvrant $\mathcal{A}_{p_J}$ à partir de W_J, enraciné sur le nœud courant, que nous divisons en deux sous-arbres notés $\mathcal{A}_1$ et $\mathcal{A}_2$. Chacun correspond à l'arbre couvrant de deux nouvelles partitions. Pour avertir les nœuds de la division, nous diffusons l'information le long de $\mathcal{A}$. Pour mettre à jour la connaissance locale des nœuds, les mots circulants correspondant aux deux sous-arbres sont diffusés aussi. La figure 5.9 montre un exemple de décomposition d'un graphe avec comme paramètres $m = 4$ et $M = 8$ qui produit deux partitions de tailles identiques.

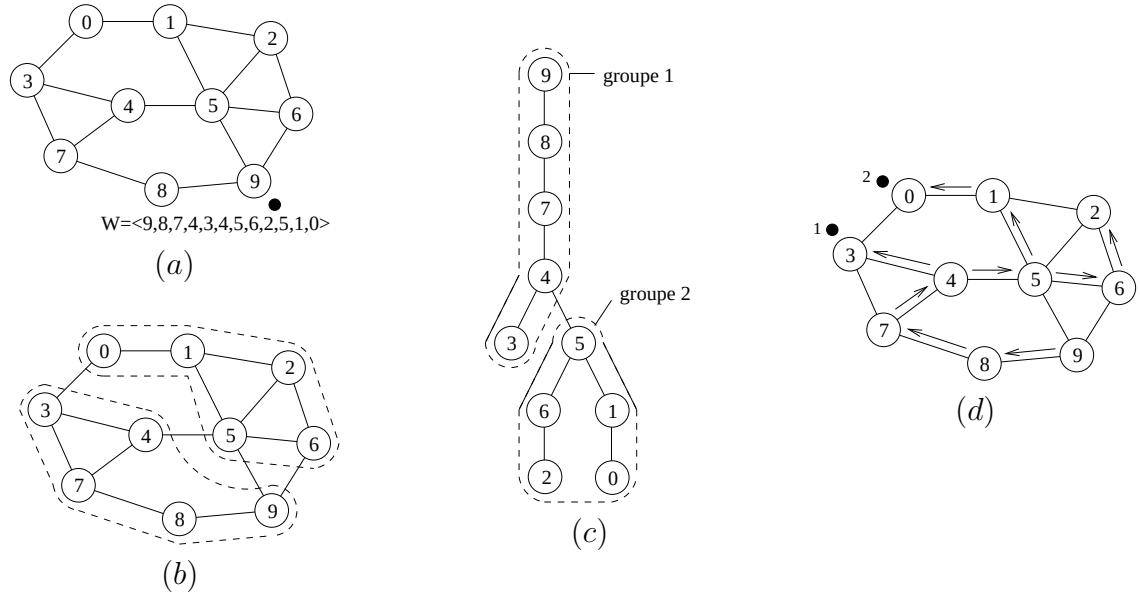


FIG. 5.9 – Le réseau quelconque (a) est décomposé en deux partitions distinctes (b) à l'aide d'un arbre couvrant (c) construit à partir d'un mot circulant. L'information est propagée le long de l'arbre (d) et deux nouvelles marches sont créées.

Cette division n'est pas toujours possible car elle dépend de la forme de l'arbre : l'étoile est un cas particulier d'arbre où tous les nœuds sont reliés à la racine. Un tel arbre ne peut être divisé en deux sous-arbres disjoints avec $m > 1$. Or, comme le mot circulant ne possède que des informations constructives, il n'est pas possible de modifier cet arbre avec les informations locales du nœud. Dans ce cas, la division est reportée et le mot circulant continue sa marche. Nous supposons que le graphe de communication est dense et en sachant que l'arbre maintenu dans un mot circulant est constamment modifié, nous obtenons en un temps fini un arbre divisible. L'algorithme 22 décrit le mécanisme de division d'un arbre en deux sous-arbres.

Une fois la division terminée, le jeton initial est supprimé. La création d'une marche aléatoire locale aux deux nouvelles partitions est automatique grâce au mécanisme des compte-à-rebours. Lors de la diffusion, le changement de partition est répercuté sur les nœuds qui connaissent ainsi l'identité de leur nouvelle partition. Une fois créées, les marches locales continuent ensuite à agréger les différents nœuds qui n'appartiennent à aucune partition, créer d'autres partitions si la borne maximale est de nouveau atteinte et ainsi de suite jusqu'à ce que chaque nœud du réseau appartienne à une partition.

Algorithme 22 Division d'un arbre $\mathcal{A}$ en deux sous-arbres $\mathcal{A}_1$ et $\mathcal{A}_2$

Pour i **allant de** 1 **à** n **Faire**
  $nbFils[i] \leftarrow 0$
Fin Pour
$empiler(\mathcal{P}, racine(\mathcal{A}))$
Tant que $est\_vide(\mathcal{P}) = faux$ **Faire**
  $courant \leftarrow sommet(\mathcal{P})$
  $dépiler(\mathcal{P})$
  Si $Vois_{courant} = \emptyset$ **Alors**
    $nbFils[courant] \leftarrow 1$
    $nbFils[père(\mathcal{A}, courant)] \leftarrow nbFils[père(\mathcal{A}, courant)] + 1$
  Sinon
    $nbFils[courant] \leftarrow 1$
    empiler dans $\mathcal{P}$ tous les fils de *courant*
  Fin Si
Fin Tant que
Sélectionner r tel que $nbFils[r]$ est le plus proche de $(M - m)/2$
Si $m \leq nbFils[r] \leq M$ **Alors**
  $\mathcal{A}_1 \leftarrow \mathcal{A} \setminus branche(\mathcal{A}, r)$
  $\mathcal{A}_2 \leftarrow branche(\mathcal{A}, r)$
Sinon
  La division est impossible
Fin Si

5.5.3 Fusion de partitions

Si une marche arrive sur un nœud i d'une autre partition, i possède alors les informations à la fois sur la partition du jeton p_J et de sa propre partition p_i. Si p_i et p_J sont stables,

c'est-à-dire qu'elles possèdent toutes les deux un nombre de nœuds compris dans l'intervalle $[m, M]$, le jeton est tout simplement retourné à l'émetteur et i n'est pas ajouté dans le mot circulant. Par contre, si les deux partitions ne sont pas stables, il est possible de réaliser une fusion.

La création des partitions est réalisée de manière décentralisée et ne dépend que de la circulation des marches locales. Le risque est d'obtenir des partitions dont le nombre d'identités est inférieur à m. Dans ce cas, elles doivent être absorbées par l'une des partitions adjacentes dans le but d'obtenir des partitions stables. Comme les marches locales circulent sur les nœuds adjacents à leur partition, il n'est pas nécessaire de mettre en place des mécanismes supplémentaires pour faire communiquer les partitions entre elles.

Si une fusion doit être réalisée, les seules informations connues sont celles contenues dans le jeton et éventuellement, les informations locales au nœud i. Cependant, le jeton de la partition p_i circule au sein de sa partition et peut être en cours d'annexion de nouveaux nœuds. Aussi, le nombre de nœuds de p_i n'est pas connu par le nœud i tant que la marche locale ne le visite pas de nouveau. Nous choisissons donc de laisser le jeton de la partition p_J décider lui-même si sa partition doit être absorbée. Ainsi, si le nombre d'identités de la partition p_J est inférieur à m, une diffusion le long de l'arbre $\mathcal{A}_{p_J}$ est amorcée pour avertir les nœuds de p_J qu'ils font partie dorénavant de la partition p_i. Le jeton est ensuite supprimé. Lors de la diffusion, le mot circulant contenu sur i est fusionné à celui contenu dans le jeton et est envoyé à tous les nœuds de p_J. La trace sur les nœuds est ainsi mise à jour.

Exemple 5.6 *La figure 5.10 montre un exemple de fusion entre deux partitions. Le jeton de la partition A arrive sur un nœud de la partition B. Le nœud 2 détecte que le nombre de nœuds contenus dans la partition A est insuffisant. Aussi, le nœud amorce la fusion de la partition A par la partition B.*

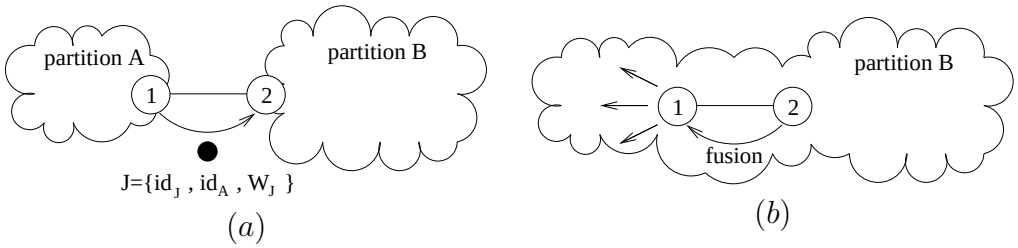


FIG. 5.10 – Le nœud 2 reçoit le jeton de la partition A (a). Le nombre de nœuds de la partition A est insuffisant, une fusion est donc exécutée (b).

5.5.4 Gestion des pannes

Le partitionnement et la fusion sont réalisés à l'aide d'une diffusion le long d'un arbre, c'est-à-dire en $n-1$ étapes maximum. Différentes pannes peuvent intervenir durant ces processus entraînant des incohérences. La figure 5.11 présente trois cas de conflit.

Le premier cas intervient lorsque deux marches essaient simultanément de fusionner leur partition. Dans ce cas, le résultat est une inversion totale des deux partitions et les deux marches sont détruites. Cependant, en un temps fini, une ou plusieurs autres marches sont recréées. La valeur d'initialisation des compte-à-rebours t_{ini} étant calculée aléatoirement dans un intervalle donné, en un temps fini, l'une des deux partitions est englobée par l'autre.

Plus généralement, plusieurs partitions peuvent se faire absorber en cascade, ce qui constitue le cas 2. Comme le montre la figure 5.11 (b), les partitions plus petites finissent par se faire absorber en un temps fini. Dans cet exemple, l'une des partitions finit par absorber les deux autres avec éventuellement la création de deux nouvelles partitions si son nombre de nœuds dépasse M.

Le dernier cas présenté intervient lorsqu'un jeton d'une partition donnée est dupliqué suite à la fin prématurée d'un compte-à-rebours. En un temps fini, l'ancien jeton finit par être détruit à l'aide des règles données dans la section 1.4.5 en comparant les identités des jetons. Cependant, il est possible que la duplication de jeton survienne en même temps qu'une agrégation de partitions ou la division d'une partition. Dans les deux cas, le jeton fautif se retrouve en un temps fini sur un nœud qui n'est plus dans sa partition et, comme présenté sur la figure, son nœud émetteur ne fait plus partie de sa partition. Sur cet exemple, le jeton est renvoyé par le nœud 4 au nœud 7 et lorsque le nœud 7 reçoit le jeton, il détecte une incohérence : son identité se retrouve en tête du mot circulant alors que le jeton ne fait pas partie de sa partition. Le jeton est considéré comme incorrect et est détruit. Cette vérification est obligatoire pour éviter la circulation infinie du jeton entre les nœuds 4 et 7.

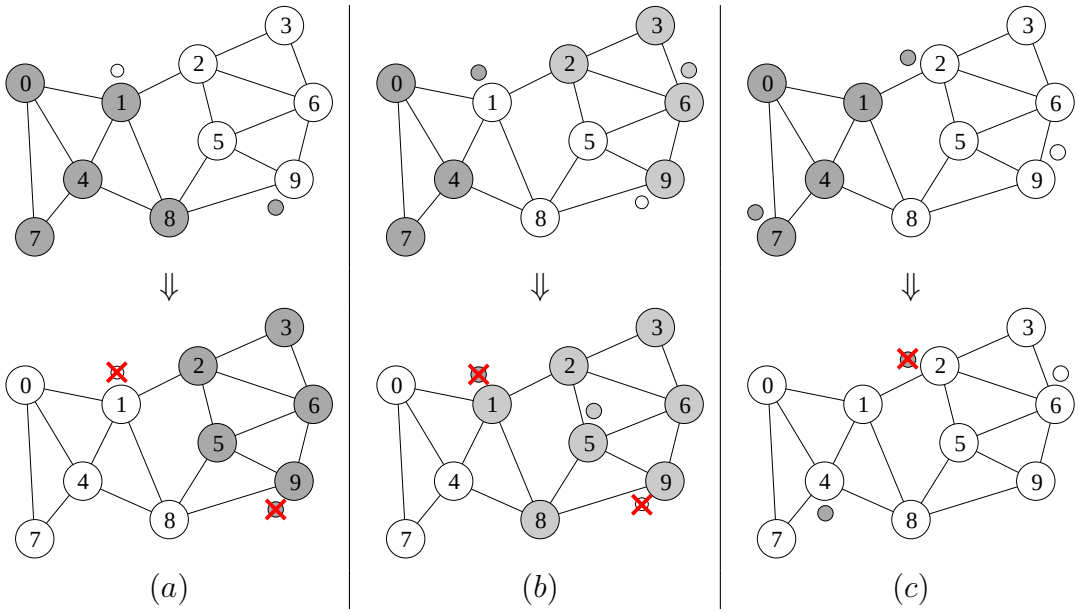


FIG. 5.11 – Différents cas de conflit que nous pouvons rencontrer lors de l'agglomération de partitions. Le cas (a) montre une absorption mutuelle entre deux partitions de dimensions inférieures à m, le cas (b) montre des absorptions en cascade et le dernier cas (c) provient d'une duplication de jeton.

Quel que soit le cas de figure, le temps nécessaire pour le rétablissement d'un fonctionnement stable dépend de la valeur d'initialisation des compte-à-rebours, celle-ci calculée aléatoirement dans un intervalle déterminé en fonction de M. Il est donc important de fixer une taille raisonnable pour les partitions.

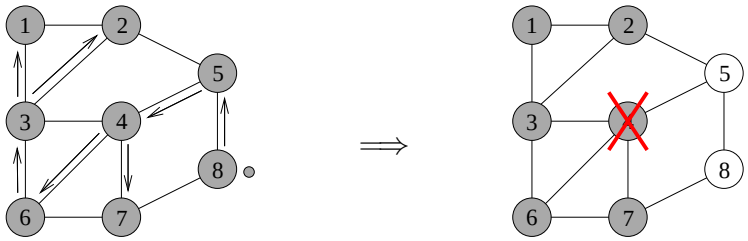
La panne d'un ou plusieurs nœuds peut avoir des répercussions sur les partitions. Nous distinguons les cas suivants :

1. la partition possède moins de m identités ;
2. la partition n'est plus connexe et produit deux partitions ayant la même identité ;
3. la panne survient pendant une diffusion (fusion ou partitionnement).

Si la partition ne possède plus assez de nœuds (cas 1.), elle est fusionnée en un temps fini avec l'une des partitions adjacentes. Par contre, cette absorption peut avoir des répercutions sur plus ou moins de partitions en fonction des bornes m et M choisies. L'instabilité de l'ensemble des partitions est plus grande si la différence entre m et M est petite.

Dans le deuxième cas, nous avons deux partitions contenant respectivement n_1 et n_2 nœuds. Si $n_1 > m$ et $n_2 > m$, les deux partitions sont stables et coexistent avec la même identité. Un jeton est recréé dans chaque d'entre elles. Si une connexion est rétablie entre les deux, l'un des deux jetons est détecté comme invalide et elles sont réabsorbées sans recours à une diffusion. Par contre, la séparation peut produire des partitions instables (*i.e.* si $n_1 < m$ ou $n_2 < m$). Elles peuvent être absorbées par des partitions adjacentes, dans quel cas leur identité est modifiée. Mais si elles absorbent des partitions adjacentes jusqu'à devenir stables, deux partitions coexistent dans le réseau avec une identité identique.

Le dernier cas 3. entraîne une incohérence au niveau des identités des partitions. La figure 5.12 montre le déroulement d'une diffusion qui est stoppée suite à la panne d'un nœud. Nous observons que le réseau se retrouve avec seulement quelques nœuds qui ont mis à jour l'identité de leur nouvelle partition et les autres qui ont gardé l'ancienne identité. Dans ce cas, la partition contenant les nœuds 8 et 5 finit par se faire absorber par l'ancienne partition ou par une partition adjacente. Pour le reste de l'ancienne partition, une nouvelle tentative de division survient si le nombre de nœuds est toujours supérieur à M.



(a) (b)

FIG. 5.12 – Conséquences de la panne d'un nœud lors d'une diffusion initiée par le nœud 8 (a) : le réseau doit être divisé en deux partitions mais le nœud 4 tombe en panne (b) et bloque la diffusion à tout le reste du réseau.

5.5.5 Algorithme principal

À l'initialisation, les nœuds du réseau n'appartiennent à aucune partition (*i.e.* $p_i = (0, 0)$) et chaque compte-à-rebours est initialisé par t_{ini}. Le comportement des nœuds dépend donc de deux événements particuliers : la réception d'un jeton ou la fin de son compte-à-rebours.

La réception d'un jeton est décrit par l'algorithme 23. Pour simplifier, le test de cohérence local et le test sur la validité du jeton ne sont pas décrits. Le comportement du nœud dépend du mot circulant reçu : si l'identité du nœud apparaît dans le mot, il doit vérifier si l'identité

de sa partition est identique à celle du jeton. Si c'est le cas, il s'agit du cas normal. Le mot circulant est alors mis-à-jour et le jeton est transféré à l'un des voisins. Dans le cas contraire, le mot contient l'identité du nœud alors que le jeton et le nœud appartiennent à des partitions différentes. Le jeton est donc invalide et est détruit. Enfin, si le jeton n'appartient pas à la même partition que le nœud, il est retourné à son émetteur sauf si sa partition ne contient pas assez de nœuds, auquel cas elle est fusionnée à celle du nœud courant.

Algorithme 23 Réception du jeton J sur le nœud i envoyé par le nœud k

```
Si (i ∈ identités(W_J)) ∨ (p_i = (0,0)) Alors
  Si (p_J = p_i) ∨ (p_i = (0,0)) Alors
    p_i ← p_J
    ajouter(W_J, i) et W_i ← W_J
    Si |identités(W_J)| > M Alors
      A ← arbre(W_J)
      Si la division de A en A_1 et A_2 est possible Alors
        /* La division de la partition peut être réalisée */
        Propager sur A les informations concernant la division
        Détruire J
      Sinon
        /* La division est impossible donc le jeton continue sa marche */
        Envoyer J à l choisi aléatoirement dans Vois_i
      Fin Si
    Sinon
      /* Cas normal : le jeton doit être envoyé à un voisin */
      Envoyer J à l choisi aléatoirement dans Vois_i
    Fin Si
  Sinon
    /* Le jeton a été retourné par k mais p_i ≠ p_J, le jeton est donc invalide */
    Détruire J
  Fin Si
Sinon
  /* Le jeton ne fait pas partie de la partition courante */
  Si |identités(W_J)| < m Alors
    /* La partition p_J doit être absorbée par p_i */
    Propager la fusion sur arbre(W_J)
    Détruire J
  Sinon
    Retourner J à k
  Fin Si
Fin Si
```

Lorsqu'un compte-à-rebours se termine, chaque nœud réagit suivant l'algorithme 24. Si le nœud n'appartient à aucune partition, il en crée une nouvelle : il crée une nouvelle marche et attribue une identité à sa partition (le couple formé de son identité et du temps courant).

Sinon, il considère que le jeton de sa partition a disparu et qu'il doit être régénéré.

Algorithme 24 Fin du compte-à-rebours T_i sur le nœud i

Si $p_i \neq 0$ **Alors**
  /* *Régénération du jeton de la partition* p_i */
  Création du jeton J avec $p_J \leftarrow p_i$ et $W_J \leftarrow W_i$
Sinon
  /* *Création d'une nouvelle partition* */
  $p_i \leftarrow (i, temps\ courant)$
  Création du jeton J avec $p_J \leftarrow p_i$ et $W_J \leftarrow < i >$
Fin Si
Envoi de J à k choisi aléatoirement dans $Vois_i$
$T_i \leftarrow t_{ini}$

5.6 Conclusion

Nous avons présenté dans ce chapitre différentes optimisations qui peuvent être apportées aux applications qui exploitent le couple d'outils formé du mot circulant et de la marche aléatoire. Ces différents mécanismes peuvent être adaptés aux solutions que nous avons présentées dans les chapitres précédents.

La gestion du mot circulant est améliorée sur plusieurs points. Premièrement, le test de cohérence local basé sur la structure de l'arbre couvrant permet de conserver plus d'informations constructives dans le mot lorsque des changements topologiques sont détectés. Deuxièmement, la construction de l'image partielle du graphe de communication est considérablement accélérée en ajoutant systématiquement les voisins des nœuds qui n'ont pas encore été visités. Troisièmement, en modifiant la gestion du contenu du mot, nous pouvons générer des arbres dont la hauteur et le degré moyen des nœuds sont limités.

Enfin, en utilisant uniquement les mécanismes propres au mot circulant et les propriétés des marches aléatoires, nous créons un partitionnement du réseau pour diminuer le temps de couverture de la marche aléatoire. Ce partitionnement est obtenu de manière totalement décentralisée et il est tolérant aux pannes.

Conclusion et Perspectives

Conclusion

La grille informatique est un outil dont le but est de regrouper des ressources partagées, distribuées et hétérogènes afin de réaliser des actions globales difficiles voire impossible à réaliser sur une machine unique. Le besoin toujours grandissant de puissance de calcul nous pousse à utiliser des processeurs qui sont disséminés dans les entreprises ou chez les particuliers. Outre le passage à l'échelle qui doit être géré, le dynamisme de telles ressources impliquent la mise en place de mécanismes particuliers. C'est pourquoi l'approche pair-à-pair a été appliquée aux applications de grille. La répartition de la charge des serveurs sur les nœuds permet d'éviter la distinction de nœuds particuliers, ce qui permet une plus grande tolérance aux pannes. Le principe des communications ad-hoc, quant à lui, permet un passage à l'échelle important.

En plus de cette approche pair-à-pair, nous avons choisi d'utiliser les marches aléatoires et le mot circulant qui sont particulièrement adaptés au dynamisme du réseau. Leur gestion est peu coûteuse, ce qui limite la charge des nœuds et les communications sont peu nombreuses, évitant ainsi la surcharge du réseau.

Dans un premier temps, nous nous sommes intéressés à la modélisation des applications de grille ou pair-à-pair, étape essentielle dans la conception de telles applications. Nous avons ainsi proposé un modèle théorique composé de 5 couches, chacune focalisée sur des mécanismes particuliers. Les trois premières couches s'appliquent à mettre en évidence les protocoles réseau sous-jacents à une grille, afin d'apporter des mécanismes de correction de pannes appropriés au niveau de l'application. Les deux couches supérieures représentent l'application en elle-même et les interactions entre ses différents composants. Ce modèle est générique et permet ainsi d'y plonger des solutions existantes afin de comparer leurs performances, par exemple, en fonction de stress particuliers (nombre de pannes, taille du réseau).

Nous avons développé une bibliothèque de simulation à événements discrets appelée *Dasor*, dédiée aux applications distribuées. Les simulateurs sont écrits indépendamment du réseau et des modèles de simulation. Cette indépendance entre les modèles et le simulateur permet à l'utilisateur de changer les modèles sans modifier le code du simulateur. En choisissant des modèles plus ou moins complexes, il est possible de simuler avec précision les protocoles sous-jacents ou au contraire, s'abstraire de ces protocoles pour réduire le temps de simulation. La bibliothèque propose plusieurs modèles comme des modèles de communication, de mobilité ou de pannes, ainsi que différents outils pour aider à la conception de simulateurs ou à l'analyse des résultats. Le modèle d'exécution des simulateurs écrits à l'aide de la bibliothèque sont calqués

sur le modèle théorique. Ainsi, le couple constitué du modèle théorique et de la bibliothèque de simulation forme un outil très complet pour la conception d'applications distribuées.

À partir du modèle, nous avons développé une solution pour la gestion des ressources qui exploite le contenu d'un mot circulant. Pour cela, nous avons proposé une nouvelle gestion pour le mot circulant pour prendre en compte l'orientation des liens de communication. Cette gestion prend ainsi en compte les limitations de communication dues aux pare-feux ou à des protocoles déployés dans le réseau. Au sein du mot circulant, une image partielle du graphe de communication est maintenue et peut être utilisée pour la construction de structures couvrantes. Les identités des ressources sont collectées et des chemins sont maintenus entre elles. Nous avons présenté différentes applications comme la localisation de ressources ou la construction d'un réseau de recouvrement pour les réseaux pair-à-pair.

Au-dessus de la couche de gestion des ressources, la gestion des tâches permet de réaliser des calculs dans une grille. Nous avons proposé une solution basée sur une marche aléatoire qui se charge de diffuser et de mettre à jour les paramètres des tâches ainsi que leur état de calcul courant. Cette solution, complètement distribuée, est basée sur deux méthodes d'assignation des tâches appelées *passive* et *active*. Nous avons comparé ces deux méthodes en fonction des paramètres physiques de la grille ainsi que des caractéristiques des tâches. Nous avons aussi proposé plusieurs optimisations possibles. La première consiste à exploiter la circulation de marches aléatoires concurrentes. Elle est plus efficace mais produit plus de messages, indépendamment de la configuration courante. D'autre part, nous nous sommes intéressés à une méthode hybride, capable de passer dynamiquement de la méthode *passive* à la méthode *active* et inversement, en fonction de la configuration de la grille et des tâches. Cette méthode réduit notablement le nombre de tâches répliquée et limite ainsi l'utilisation de la puissance de calcul des nœuds. Enfin, nous avons proposé d'utiliser un arbre couvrant, construit à partir d'un mot circulant, pour diffuser périodiquement les états des tâches. Cette méthode est plus flexible que les autres méthodes et propose une efficacité supérieure. Elle produit plus de messages que la solution hybride. Cependant, le mot circulant utilisé à la couche inférieure pour la gestion des ressources peut être exploité pour la construction des arbres couvrants. Ainsi, un seul jeton est utilisé pour les deux couches.

Enfin, nous avons proposé plusieurs optimisations pour les applications à base de marches aléatoires et de mot circulant. En particulier, nous avons présenté plusieurs méthodes pour améliorer la correction des incohérences topologiques dans le mot circulant. Elles limitent le nombre d'identités supprimées lors de la détection d'incohérences. D'autre part, nous avons étudié deux méthodes pour accélérer le déplacement du jeton dans le réseau en guidant la marche aléatoire à l'aide du contenu du mot. L'ajout systématique des voisins accélère considérablement la construction de l'image partielle du graphe de communication. De plus, nous avons proposé une méthode pour construire efficacement des arbres couvrants dont leur hauteur et le degré moyen de leurs nœuds sont contrôlés. Ces arbres sont particulièrement adaptés pour diffuser de l'information au sein d'un réseau dynamique. Enfin, nous avons proposé de faciliter le passage à l'échelle de solutions exploitant les marches aléatoires en partitionnant le réseau. Les partitions sont créées dynamiquement de manière totalement décentralisée, sont tolérantes aux pannes et sont maintenues à l'aide de marches aléatoires locales.

Perspectives

À partir des différents travaux réalisés au cours de cette thèse, nous envisageons des poursuites dans plusieurs directions.

Dasor. La bibliothèque propose pour le moment quelques modèles de simulation (panne, mobilité, communication, routage). Nous proposons de développer d'autres modèles, plus proches des contraintes des réseaux pair-à-pair ou des grilles. Ensuite, nous planifions d'optimiser encore le code. Actuellement, les réseaux simulés ne dépassent pas 10000 nœuds pour des raisons de mémoire. Or, les réseaux pair-à-pair ou les grilles peuvent comporter beaucoup plus de nœuds. Pour diminuer le temps d'exécution des simulations, une approche distribuée est envisagée sur deux niveaux : la parallélisation du moteur du simulateur et la parallélisation d'une série de simulations. Ce dernier niveau a été développé pour fonctionner sur la machine parallèle Roméo 2, du centre de calcul de Reims Champagne-Ardenne. Pour une plus grande portabilité, une intégration à l'intergiciel *CONFIIT* est prévue.

Les modèles de simulation renseignés via le fichier de description, sont appliqués à l'ensemble des nœuds du réseau lors de l'exécution du simulateur. Il serait intéressant de pouvoir appliquer des modèles à des sous-ensembles de nœuds afin d'étudier les interactions entre des composants sans fil et des composants filaires, par exemple, ou en proposant d'appliquer des protocoles de communication différents à des sous-parties du réseau.

Gestion des ressources. La gestion du mot circulant peut être lourde lorsque la taille du réseau devient trop importante. En effet, le mot circulant peut être vu comme une image partielle du graphe de communication et il contient l'identité de tous les nœuds ainsi que les chemins pour les atteindre. En exploitant les différents résultats du dernier chapitre, nous envisageons de proposer une nouvelle méthode pour hiérarchiser dynamiquement le réseau afin de réduire la taille du mot circulant et ainsi diminuer le temps de mise-à-jour de la connaissance de la topologie. La méthode de partitionnement doit être adaptée à l'orientation des liens de communication. De plus, les ressources doivent pouvoir être localisées dans le réseau complet. Il est donc nécessaire de déployer un protocole de routage entre ces partitions.

Gestion des tâches. La gestion des tâches que nous avons proposée ne prend pas en compte les liens entre les tâches et les autres ressources de la grille. Si des dépendances existent, notamment si de nombreuses tâches nécessitent le transfert d'un ensemble de données situé sur un nœud distant, le réseau de communication risque d'être saturé. Toujours de manière décentralisée, nous envisageons ainsi une assignation qui tient compte de la localité des données. Nous pouvons aussi prévoir d'assigner des tâches sur des nœuds voisins si des communications sont nécessaires.

Nous avons vu aussi que des protocoles comme *Gnutella* ont évolué vers une hiérarchisation dynamique des nœuds dans le but de réduire les communications. Nous projetons ainsi d'optimiser l'assignation des tâches en laissant un rôle plus important à des nœuds particuliers. Cette méthode doit cependant rester adaptative afin de garder une tolérance aux pannes importante.

Les différentes méthodes d'optimisation des méthodes passive et active dépendent de paramètres qui sont fixés par l'utilisateur. Les valeurs optimales de ces paramètres dépendent de la configuration courante (nombre de tâches, nombre de nœuds). Nous planifions d'automatiser

l'ajustement de ces paramètres afin qu'ils soient adaptés quelle que soit la configuration actuelle.

Méthodes d'optimisation. Les différentes solutions proposées dans le dernier chapitre ont des comportements différents suivant la topologie du réseau et le dynamisme des nœuds. Une étude approfondie permettrait de créer une solution plus générale, exploitant les atouts de chacune et s'adaptant automatiquement à l'évolution des paramètres d'exécution.

De même, le partitionnement est réalisé sans considération sur la topologie du réseau. Il pourrait être amélioré en le combinant aux différentes optimisations et ainsi contrôler l'aspect des partitions générées. Ainsi, celles-ci pourraient être utilisées pour d'autres applications comme le routage.

Annexes

Méthodes utilisées dans les différents algorithmes

Dans cette partie, nous décrivons les différentes méthodes utilisées dans les algorithmes de ce manuscrit. Les algorithmes de ces méthodes sont disponible dans la documentation technique de *Dasor* [Rab07a].

Les arbres. Pour la gestion d'un arbre $\mathcal{A}$, nous utilisons les différentes méthodes suivantes :
- $identités(\mathcal{A})$: retourne l'ensemble des identités contenues dans l'arbre, $|identités(\mathcal{A})|$ étant le nombre d'identités distinctes dans l'arbre ;
- $ajouter(\mathcal{A}, i, j)$: ajoute l'identité j dans l'arbre, enracinée sur le nœud i ;
- $racine(\mathcal{A})$: retourne l'identité de la racine de l'arbre ;
- $père(\mathcal{A}, i)$: retourne l'identité du père de i dans l'arbre (retourne i si i est la racine) ;
- $fils(\mathcal{A}, i)$: retourne l'ensemble des fils de i dans l'arbre ;
- $branche(\mathcal{A}, r)$: retourne la branche enracinée sur l'identité "r".

Nous supposons aussi l'existence des opérations suivantes :
- $\mathcal{A} \leftarrow r$: construit un arbre qui ne contient que l'identité r qui est aussi sa racine ;
- $\mathcal{A} \setminus branche(\mathcal{A}, r)$: représente l'arbre $\mathcal{T}$ privé de la branche enracinée en "r".

Les piles. Dans certains algorithmes, nous utilisons des piles, généralement d'entiers. Ce sont des structures qui ne permettent l'accès qu'au dernier élément ajouté. Les méthodes utilisées pour une pile $\mathcal{P}$ sont :
- $est\_vide(\mathcal{P})$: retourne "*vrai*" si la pile est vide ;
- $sommet(\mathcal{P})$: retourne la valeur qui est située au sommet de la pile sans la modifier ;
- $empiler(\mathcal{P}, e)$: place la valeur "e" au sommet de la pile ;
- $dépiler(\mathcal{P})$: supprime la valeur qui est au sommet de la pile.

Glossaire

Calcul global (*Global Computing*) : la problématique est identique à celle du *métacomputing*. Cependant, les dispositifs mis en réseau sont de natures différentes (machines parallèles, processeurs graphiques, consoles de jeu, ...). On trouve de nombreux exemples comme *SETI@home* ou *XtremWeb*.

Grappe (de serveurs) ou *ferme de calcul* (*Cluster*) : c'est un groupe de serveurs couplés dans le but de former un seul et unique ordinateur. Ils sont en général reliés par un réseau local très haut débit et partagent des capacités de disque communes.

Grille de PC (*Desktop grid*) : dans une telle grille, les ressources exploitées sont celles d'ordinateurs de bureau. On dit des applications qui exploitent ce type de grille pour le calcul, qu'elles réalisent du vol de cycles. Une grille de PC est caractérisée par le dynamisme de son environnement d'exécution. Les ressources sont très dynamiques et les performances du réseau sont très variables.

Grille RPC (*Grid RPC*) : c'est une grille basée sur le protocole *RPC* (pour *R*emote *P*rocedure *C*all). L'*API GridRPC* [SNM+02] fournit des mécanismes standardisés, portables ainsi qu'une programmation simplifiée pour implémenter le *RPC* dans les grilles. *DIET* est un environnement de *grille RPC*.

HPGC (pour *High Performance Grid Computing*). Contrairement aux *grilles de PC*, ces grilles impliquent des performances importantes. Les ressources sont fiables et en général sont de grosses capacités (machines parallèles, grappes de serveurs). Les différentes entités sont souvent reliées par un réseau haut-débit.

Intergiciel (ou *middleware*) : c'est un logiciel servant d'intermédiaire entre plusieurs logiciels, plusieurs protocoles ou les deux. Dans le cas d'un intergiciel de grille, il est placé aux couches 4 et 5 de notre modèle théorique et sert d'intermédiaire entre l'application de haut niveau et les ressources.

Métacomputeur : les auteurs de [SC92] définissent un *métacomputeur* comme un réseau de ressources de calcul ou de stockage hétérogènes reliées par un logiciel qui permet de les utiliser comme un seul ordinateur. *Globus* est un intergiciel de *métacomputing*.

Pull Model : dans ce modèle, les serveurs de calcul sont à l'initiative des demandes de tâches à calculer. Exemple : *XtremWeb*.

Push Model : contrairement au modèle *Pull*, les serveurs sont mis à disposition d'un agent externe qui est à l'initiative des demandes de tâches. Exemples : *DIET* et *NetSolve*.

Table de hachage distribuée (ou *DHT* pour *Distributed Hash Table*) : une table de hachage permet d'associer des ressources à un identifiant unique dans le but d'y accéder rapidement. Lorsqu'elle est distribuée, la table est répartie sur un ensemble de nœuds du réseau, chacun en possédant une partie. Le protocole *Chord* utilise les *DHT* pour la recherche de ressources dans des réseaux pair-à-pair.

Téraflops (ou *TFlops*) : un FLOPS (pour *FL*oating-point *O*perations *P*er *S*econd) correspond au nombre d'opérations en virgules flottantes par seconde d'un microprocesseur ou d'un ordinateur. Un TéraFlops (TFlops) correspond donc à 10^{12} opérations à virgule flottante par seconde.

Bibliographie

[AAB+02] Dorian Arnold, Sudesh Agrawal, Susan Blackford, Jack Dongarra, Michelle Miller, Kiran Seymour, Kiran Sagi, Zhiao Shi et Sathish Vadhiyar : Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, juin 2002.

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky et Dan Werthimer : SETI@home : an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, novembre 2002.

[And03] David P. Anderson : Public Computing : Reconnecting People to Science. *In Conference on Shared Knowledge and the Web*, novembre 2003.

[BAG00] Rajkumar Buyya, David Abramson et Jonathan Giddy : Nimrod/G : An Architecture of a Resource Management and Scheduling System in a Global Computational Grid. *CoRR*, cs.DC/0009021, 2000.

[BBF04a] Thibault Bernard, Alain Bui et Olivier Flauzac : Random Distributed Self-stabilizing Structures Maintenance. *In* Félix F. Ramos, Herwig Unger et Victor Larios, éditeurs : *Advanced Distributed Systems : Third International School and Symposium, ISSADS 2004*, volume 3061 de *Lecture Notes in Computer Science*, pages 231–240. Springer-Verlag, 2004.

[BBF04b] Thibault Bernard, Alain Bui et Olivier Flauzac : Topological Adaptability for the Distributed Token Circulation Paradigm in Faulty Environment. *In* Jiannong Cao, Laurence Tianruo Yang, Minyi Guo et Francis Chi-Moon Lau, éditeurs : *ISPA, International Symposium on Parallel and Distributed Processing and Applications*, volume 3358 de *Lecture Notes in Computer Science*, pages 146–155. Springer-Verlag, 2004.

[BBFN06] Thibault Bernard, Alain Bui, Olivier Flauzac et Florent Nolot : A multiple random walks based self-stabilizing k-exclusion algorithm in ad-hoc networks. *In Sixth International Symposium and School on Advanced Distributed Systems*, Lecture Notes in Computer Science. Springer, 2006.

[BBFR06a] Thibault Bernard, Alain Bui, Olivier Flauzac et Cyril Rabat : Decentralized Resources Management for Grid. *In RDDS'06 International Workshop on Reliability in Decentralized Distributed systems*, volume 4278 de *LNCS*, pages 1530–1539. Springer-Verlag, 2006.

[BBFR06b] Thibault Bernard, Alain Bui, Olivier Flauzac et Cyril Rabat : Gestion de la mobilité dans un réseau orienté à l'aide d'un mot circulant. *In ROADEF'06*, 2006.

[BBL02] Mark Baker, Rajkumar Buyya et Domenico Laforenza : Grids and grid technologies for wide-area distributed computing. *Softw. Pract. Exper.*, 32(15):1437–1466, décembre 2002.

[BCC+02] William H. Bell, David G. Cameron, Luigi Capozza, A. Paul Millar, Kurt Stockinger et Floriano Zini : Simulation of Dynamic Grid Replication Strategies in OptorSim. *In GRID '02 : Proceedings of the Third International Workshop on Grid Computing*, pages 46–57, London, UK, 2002. Springer-Verlag.

[BCC+07] Ernst W. Biersack, Damiano Carra, Renato Lo Cigno, Pablo Rodriguez et Pascal Felber : Overlay architectures for file distribution : Fundamental performance analysis for homogeneous and heterogeneous cases. *Computer Networks*, 51(3):901–917, 2007.

[BFR07] Alain Bui, Olivier Flauzac et Cyril Rabat : Fully Distributed Active and Passive Task Management for Grid Computing. *In ISPDC '07 : Proceedings of the Sixth International Symposium on Parallel and Distributed Computing*. IEEE Computer Society, juillet 2007.

[BIZ89] Judit Bar-Ilan et Dror Zernik : Random Leaders and Random Spanning Trees. *In Proceedings of the 3rd International Workshop on Distributed Algorithms (WDAG89)*, pages 1–12, London, UK, 1989. Springer-Verlag.

[BRF04] Ernst W. Biersack, Pablo Rodriguez et Pascal Felber : Performance Analysis of Peer-to-Peer Networks for File Distribution. *In Quality of Service in the Emerging Networking Panorama*, volume 3266 de *Lecture Notes in Computer Science*, pages 1–10. Springer, 2004.

[BS05] Alain Bui et Devan Sohier : On Time Analysis of Random Walk Based Token Circulation Algorithms. *In ISSADS'05 International School and Symposium on Advanced Distributed Systems*, volume 3563 de *Lecture Notes in Computer Science*, pages 63–71. Springer, 2005.

[BS07] Alain Bui et Devan Sohier : How to Compute Times of Random Walks based Distributed Algorithms. *Fundamenta Informaticae*, à paraître, 2007.

[BW90] Graham Brightwell et Peter Winkler : Maximum hitting time for random walks on graphs. *Random Structures and Algorithms*, 1:263–276, 1990.

[CDF+04] Franck Cappello, Samir Djilali, Gilles Fedak, Thomas Herault, Frédéric Magniette, Vincent Néri et Oleg Lodygensky : Computing on Large Scale Distributed Systems : XtremWeb Architecture, Programming Models, Security, Tests and Convergence with Grid. *In FGCS Future Generation Computer Science*, volume 21, pages 417–437, mars 2004.

[CDL+02] Eddy Caron, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson et Frédéric Suter : A Scalable Approach to Network Enabled Servers. *In* B. Monien et R. Feldmann, éditeurs : *Proceedings of the 8th International EuroPar Conference*, volume 2400 de *Lecture Notes in Computer Science*, pages 907–910, Paderborn, Germany, août 2002. Springer-Verlag.

[CFK+98] Karl Czajkowski, Ian Foster, Nick Karonis, Carl Kesselman, Stuart Martin, Warren Smith et Steven Tuecke : A Resource Management Architecture

for Metacomputing Systems. *In The 4th Workshop on Job Scheduling Strategies for Parallel Processing"*, pages 62–82. Springer-Verlag LNCS 1459, 1998.

[Coh03] Bram Cohen : Incentives Build Robustness in BitTorrent. *In Workshop on Economics of Peer-to-Peer Systems (P2PEcon'03)*, juin 2003.

[CRR+97] Ashok K. Chandra, Prabhakar Raghavan, Walter L. Ruzzo, Roman Smolensky et Prasoon Tiwari : The Electrical Resistance of a Graph Captures its Commute and Cover Times. *Computational Complexity*, 6(4), 1997.

[CSWH01] Ian Clarke, Oskar Sandberg, Brandon Wiley et Theodore W. Hong : Freenet : a distributed anonymous information storage and retrieval system. *In International workshop on Designing privacy enhancing technologies*, pages 46–66, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[DC05] Holly Dail et Eddy Caron : GoDIET : a tool for managing distributed hierarchies of DIET agents and servers. Rapport technique RR-2005-06, Laboratoire de l'Informatique du Parallélisme (LIP), février 2005. Also available as INRIA Research Report RR-5520.

[DQS01] Frédéric Desprez, Martin Quinson et Frédéric Suter : Dynamic Performance Forecasting for Network Enabled Servers in an heterogeneous Environment. *In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA 2001)*, volume 3, pages 1421–1427. CSREA Press, juin 2001.

[DSW02] Shlomi Dolev, Elad Schiller et Jennifer Welch : Random walk for self-stabilizing group communication in ad hoc networks. *In PODC '02 : Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 259–259, New York, NY, USA, 2002. ACM Press.

[ER59] Paul Erdös et Alfrèd Rényi : On Random Graphs. *Publicationes Mathematicae*, pages 290–297, 1959.

[Fei95a] Uriel Feige : A tight lower bound on the cover time for random walks on graphs. *Random Structures & Algorithms*, 6(4):433–438, 1995.

[Fei95b] Uriel Feige : A tight upper bound on the cover time for random walks on graphs. *Random Structures & Algorithms*, 6(1):51–54, 1995.

[FFK+97] Steven Fitzgerald, Ian Foster, Carl Kesselman, Gregor von Laszewski, Warren Smith et Steven Tuecke : A Directory Service for Configuring High-Performance Distributed Computations. *In Proc. 6th IEEE Symp. on High Performance Distributed Computing*, pages 365–375, Washington, DC, USA, 1997. IEEE Computer Society Press.

[FGL93] Gérard Florin, Roberto Gómez et Ivan Lavallée : Recursive distributed programming schemes. *In International Symposium on Autonomous Decentralized Systems (ISADS)*, pages 122–128, 1993.

[FI03] Ian T. Foster et Adriana Iamnitchi : On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing. *In* M. Frans Kaashoek et Ion Stoica, éditeurs : *IPTPS*, volume 2735 de *Lecture Notes in Computer Science*, pages 118–128. Springer, 2003.

[FK99] Ian Foster et Carl Kesselman, éditeurs. *The Grid : Blueprint for a Future Computing Infrastructure*. MORGAN-KAUFMANN, septembre 1999.

[FKF03] Olivier Flauzac, Michaël Krajecki et Jean Fugère : CONFIIT : a middleware for peer to peer computing. *In* C. Tan M. Gravilova et P. L'Ecuyer, éditeurs : *The 2003 International Conference on Computational Science and its Applications (ICCSA 2003)*, volume 2669 (III) of Lecture Notes in Computer Science, pages 69–78, Montréal, Québec, juin 2003. Springer-Verlag.

[Fla01] Olivier Flauzac : Random Circulating Word Information Management for Tree Construction and a Shortest Path Routing Tables Computation. *In* R. Gomez Cardenas, éditeur : *OPODIS*, Studia Informatica Universalis, pages 17–32. Suger, Saint-Denis, rue Catulienne, France, 2001.

[Fos06] Ian Foster : Globus Toolkit Version 4 : Software for Service-Oriented Systems. *J. Comput. Sci. Technol.*, 21(4):513–520, 2006.

[FTL+02] James Frey, Todd Tannenbaum, Miron Livny, Ian Foster et Steven Tuecke : Condor-G : A Computation Management Agent for Multi-Institutional Grids. *Cluster Computing*, 5(3):237–246, 2002.

[GB02] Thomas J. Giuli et Mary Baker : Narses : A Scalable Flow-Based Network Simulator. *CoRR*, cs.PF/0211024, 2002.

[GMS04] Christos Gkantsidis, Milena Mihail et Amin Saberi : Random Walks in Peer-to-Peer Networks. *In INFOCOM*, 2004.

[Gnu] Site officiel de Gnutella. http ://www.gnutella.com. Accédé en 2007.

[GRI] Grid'5000 Home. http ://www.grid5000.org.

[HTV07] William Hoarau, Sébastien Tixeuil et Fabien Vauchelles : FAIL-FCI : Versatile Fault-Injection. *Future Generation Computer Systems*, 2007.

[Jos03] Sam Joseph : An Extendible Open Source P2P Simulator. *P2PJournal*, 2003.

[KFJ+05] Michaël Krajecki, Olivier Flauzac, Christophe Jaillet, Pierre-Paul Mérel et Richard Tremblay : Solving an open Instance of the Langford Problem using CONFIIT : a Middleware for Peer-to-Peer Computing. *Parallel Processing Letters*, à paraître, 2005.

[Kra99] Michaël Krajecki : An object oriented environment to manage the parallelism of the FIIT applications. *In* V. Malyshkin, éditeur : *Parallel Computing Technologies, 5th International Conference, PaCT-99*, volume 1662 of Lecture Notes in Computer Science, pages 229–234, St. Petersburg, Russia, septembre 1999. Springer-Verlag.

[Lav86] Ivan Lavallée : *Contribution à l'algorithmique parallèle et distribuée, application à l'optimisation combinatoire*. Thèse d'état, Université de Paris XI, Orsay, 1986.

[LD03] Dong Lu et Peter A. Dinda : GridG : generating realistic computational grids. *SIGMETRICS Perform. Eval. Rev.*, 30(4):33–40, 2003.

[Li03] Sing Li : JXTA 2 : A high-performance, massively scalable p2p network. Technical report, IBM developerWorks, novembre 2003.

[LK99] Averill M. Law et W. David Kelton : *Simulation Modeling and Analysis*. McGraw-Hill, 3 édition, 1999.

[LMC03] Arnaud Legrand, Loris Marchal et Henri Casanova : Scheduling Distributed Applications : the SimGrid Simulation Framework. *In CCGRID '03 : Proceedings of the 3st International Symposium on Cluster Computing and the Grid*, page 138, Washington, DC, USA, 2003. IEEE Computer Society.

[Lov93] László Lovász : Random walks on graphs : A Survey. *In* T. Szonyi ed., D. Miklos et V. T. Sos, éditeurs : *Combinatorics : Paul Erdos is Eighty*, volume 2, pages 353–398. Janos Bolyai Mathematical Society, 1993.

[LPP04] Sebastien Lacour, Christian Perez et Thierry Priol : A Network Topology Description Model for Grid Application Deployment. *In GRID '04 : Proceedings of the Fifth IEEE/ACM International Workshop on Grid Computing (GRID'04)*, pages 61–68, Washington, DC, USA, 2004. IEEE Computer Society.

[LXC04] Xin Liu, Huaxia Xia et Andrew A. Chien : Validating and Scaling the MicroGrid : A Scientific Instrument for Grid Dynamics. *Journal of Grid Computing*, 2(2):141–161, 2004.

[Mat88] Peter Matthews : Covering Problems for Brownian Motion on Spheres. *The Annals of Probability*, 16(1):189–199, janvier 1988.

[Mil67] Stanley Milgram : The small world problem. *Psychology Today*, 1:62–67, 1967.

[Nap] Site officiel de Napster. http ://www.napster.com. Accédé en 2007.

[NEU] NeuroGrid. http ://www.neurogrid.net/.

[NS] The Network Simulator. http ://www.isi.edu/nsnam/ns/.

[PEE] PeerSim. http ://peersim.sourceforge.net/.

[Rab] Cyril Rabat : Dasor Home Page. http ://cosy.univ-reims.fr/∼crabat/DASOR/.

[Rab07a] Cyril Rabat : *Documentation technique de Dasor*, août 2007.

[Rab07b] Cyril Rabat : *Notice d'utilisation de Dasor*, août 2007.

[RBF06] Cyril Rabat, Alain Bui et Olivier Flauzac : A random walk topology management solution for GRID. *In Innovative Internet Community Systems, 5th International Workshop, IICS 2005*, volume 3908 de *Lecture Notes in Computed Science*, pages 91–104, Paris, juin 2006. Springer-Verlag.

[RD01] Antony Rowstron et Peter Druschel : Pastry : Scalable, decentralized object location and routing for large-scale peer-to-peer systems. *In IFIP/ACM Middleware 2001*, Heidelberg, Germany, 2001.

[RSUW05] Cyril Randriamaro, Olivier Soyez, Gil Utard et Fancis Wlazinski : Data Distribution in a Peer to Peer Storage System. *In CCGRID '05 : Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 1*, pages 284–291, Washington, DC, USA, mai 2005. IEEE Computer Society.

[SB04] Devan Sohier et Alain Bui : Hitting Times Computation for Theoretically Studying Peer-to-Peer Distributed Systems. *IPDPS*, 08:179a, 2004.

[SC92] Larry Smarr et Charles E. Catlett : Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[Seg83] Adrian Segall : Distributed Network Protocols. *Transaction on Information Theory*, 29:23–35, 1983.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek et Hari Balakrishnan : Chord : A Scalable Peer-to-peer Lookup Service for Internet Applications. *In Proceedings of the ACM SIGCOMM '01 Conference*, pages 149–160, San Diego, California, août 2001.

[SNM+02] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee et Henri Casanova : Overview of GridRPC : A Remote Procedure Call API for Grid Computing. *In GRID '02 : Proceedings of the Third International Workshop on Grid Computing*, pages 274–278, London, UK, 2002. Springer-Verlag.

[TD03] Nyik San Ting et Ralph Deters : 3LS - A Peer-to-Peer Network Simulator. *p2p*, 00:212, 2003.

[Tet91] Prasad Tetali : Random Walks and the Effective Resistance of Networks. *Journal of Theoretical Probability*, 4:101–109, 1991.

[TW93] Prasad Tetali et Peter Winkler : Simultaneous Reversible Markov Chains. *In Combinatorics : Paul Erdos is Eighty*, volume 1, pages 433–451, 1993.

[Var01] Andràs Vargas : The OMNeT++ Discrete Event Simulation System. *In The European Simulation Multiconference (ESM 2001)*, Prague, Czech Republic, 2001.

[Wat99] Duncan J. Watts : *Small Worlds*. Princeton University Press, Princeton, NJ, USA, 1999.

[WS98] Duncan J. Watts et Steven H. Strogatz : Collective dynamics of "small-world" networks. *Nature*, 393:440–442, juin 1998.

Étude et simulation de solutions pour les grilles et systèmes pair-à-pair : application à la gestion des ressources et des tâches

Résumé : Les grilles et les systèmes pair-à-pair sont caractérisés par le dynamisme de leurs ressources. Il est donc nécessaire de modéliser ces applications et d'apporter des outils appropriés pour structurer les ressources afin de gérer ce dynamisme et permettre l'exécution de services. Dans un premier temps, nous avons proposé un modèle théorique constitué de 5 couches dans le but de gérer les mécanismes indépendamment. Les trois premières couches se focalisent sur les mécanismes sous-jacents à l'application : la couche physique (ressources et réseau d'interconnexion), le routage et les communications. Les deux autres couches concernent l'application en elle-même : la couche de topologie et la couche de services et composants. Afin d'étendre la validité des algorithmes basés sur le modèle théorique, nous avons développé une bibliothèque qui permet d'écrire des simulateurs à événements discrets dont le modèle d'exécution est calqué sur notre modèle théorique. Ils sont écrits indépendamment du réseau et des modèles de simulation. Le choix de la granularité de la simulation est très large et peut être changé sans modifier le code du simulateur. Pour gérer le dynamisme des ressources, nous avons besoin d'outils adaptés. Nous avons choisi d'utiliser dans nos solutions les marches aléatoires et le mot circulant qui permettent de construire des applications tolérantes aux pannes et complètement distribuées. Le mot circulant est un outil utilisé pour récolter des informations topologiques dans un réseau. Conformément à notre modèle, nous proposons une nouvelle gestion de son contenu pour construire des structures couvrantes capables de gérer la volatilité des ressources. Enfin, nous nous sommes intéressés à la gestion des tâches indépendantes et irrégulières à l'aide d'une marche aléatoire. Cette solution est basée sur une politique de sélection locale aux nœuds. Elle est aussi complètement décentralisée, peu coûteuse en terme d'échanges de messages et tolérante aux pannes.

Mots-clés : grilles, systèmes pair-à-pair, marches aléatoires, mot circulant, modélisation, simulation, gestion de ressources, gestion de tâches, partitionnement

Study and simulation of solutions for grids and peer-to-peer systems: application to the resources and tasks management

Abstract : The grid or peer-to-peer applications gather a large number of resources that can be very dynamic. We need to model these applications and use appropriate tools to structure the ressources in order to manage the dynamism and allow execution of services. First, we focus on modelisation of such applications thanks to a five-layers model. The three lower layers focus on subjacent mechanisms of the grid : physical network, routing and communication. The grid middleware is composed of two layers : the topology layer and the services layer. To extend algorithm validity, we wrote a library to build discrete event simulators. Their execution model is based on our theoretical model. They are written independently of the network and of the simulation models (fault, mobility, communication) and the granularity of the simulation is very large. Thanks to these tools, we proposed a solution for the resources management in dynamical networks. We use a circulating word that moves randomly and collects topological informations. To manage directed communication depending on firewalls and subjacent protocols, we propose a new content management to adapt this tool to directed graphs. Finally, we focus on the tasks management with two assignement methods called passive and active. They use a local policy, so it limits the control message exchanges and increases its fault tolerance.

Keywords : grids, peer-to-peer systems, random walks, circulating word, modelisation, simulation, resources management, tasks management, clustering