Stockage décentralisé adaptatif : autonomie et mobilité des données dans les réseaux pair-à-pair

Benoît Romito

université de Caen Basse-Normandie

UNIVERSITÉ de CAEN BASSE-NORMANDIE

U.F.R. : Sciences

ÉCOLE DOCTORALE : SIMEM

THÈSE

présentée par

Benoît Romito

et soutenue

le 11 décembre 2012

en vue de l'obtention du

DOCTORAT de l'UNIVERSITÉ de CAEN

spécialité : Informatique et applications

(Arrêté du 7 août 2006)

Stockage décentralisé adaptatif : autonomie et mobilité des données dans les réseaux pair-à-pair

MEMBRES du JURY

Yves Demazeau	Directeur de Recherche	CNRS, Grenoble	(rapporteur)
Jean-Paul Arcangeli	Maitre de conférences	Université Toulouse III	(rapporteur)
Pierre Chevaillier	Professeur des Universités	École Nationale d'Ingénieurs de Brest	
Henri Roussel	Maitre de conférences	Université de Caen Basse-Normandie	
Bruno Zanuttini	Maitre de conférences	Université de Caen Basse-Normandie	
François Bourdon	Professeur des Universités	Université de Caen Basse-Normandie	(directeur)

Mis en page avec la classe thloria.

« La base empirique de la science objective ne comporte rien d'absolu. La science ne repose pas sur une base rocheuse. La structure audacieuse de ses théories s'édifie en quelque sorte sur un marécage. Elle est comme une construction bâtie sur pilotis. Les pilotis sont enfoncés dans le marécage mais pas jusqu'à la rencontre de quelque base naturelle ou "donnée" et, lorsque nous cessons d'essayer de les enfoncer davantage, ce n'est pas parce que nous avons atteint un terrain ferme. Nous nous arrêtons, tout simplement parce que nous sommes convaincus qu'ils sont assez solides pour supporter l'édifice, du moins provisoirement. »

— K. Popper, La Logique de la découverte scientifique [1934], Payot, 1973, p.111.

Remerciements

Ces trois années de recherche au sein de l'équipe MAD du laboratoire GREYC de l'université de Caen ont été très riches pour moi tant sur le plan scientifique qu'humain. Je tiens donc à remercier toutes les personnes qui ont, de près comme de loin, contribué à l'aboutissement de ces travaux. Je tiens tout d'abord à remercier chaleureusement François Bourdon, mon directeur de thèse qui s'est toujours montré à l'écoute de mes problèmes, me faisant part de ses conseils avisés. Son soutien et son amitié m'ont été d'une aide précieuse et je n'oublierai pas nos longues discussions qui m'ont bien souvent permis d'aller plus loin.

Je remercie mes rapporteurs Yves Demazeau et Jean-Paul Arcangeli de m'avoir fait l'honneur d'être relecteurs de cette thèse. Merci tout particulièrement à Yves, d'avoir apporté de l'attention à mes travaux et d'être venu à Caen pour en discuter avec nous.

Mes remerciements vont également à Pierre Chevaillier qui a accepté d'être mon examinateur. Merci à Bruno Zanuttini, qui a été un enseignant d'exception pendant mon cursus et qui a relu et corrigé ma thèse avec enthousiasme. Je remercie aussi Henri Roussel d'avoir accepté d'être examinateur de cette thèse et d'avoir apporté un éclairage d'éthologue sur mes travaux.

Merci à Roger Cozien et Laurent Jeanpierre pour m'avoir consacré un peu de leur temps précieux. Merci à toi Hugo de m'avoir aidé à amorcer ma thèse dans de bonnes conditions, en participant à nos réunions et en m'aidant à rédiger mes premiers articles. Je remercie également Abdel-Illah Mouaddib de m'avoir encouragé à postuler sur les sujets de thèse du laboratoire alors que j'y avais renoncé.

Mes remerciements vont maintenant à toute la petite équipe de thésards, ATER et MCF avec qui j'ai passé d'excellents moments aux pauses café, le midi au resto U, lors de notre voyage au ski ou pendant nos soirées JDR. Notre émulation et nos préoccupations similaires m'ont permit d'avancer dans les moments de doute. Je remercie tout particulièrement Arnaud de m'avoir fait profiter de sa bonne humeur quotidienne et d'avoir été un formidable « sparring partner » pendant nos entraînements de Karaté. *Osu* ! Merci à toi Mathieu pour ton amitié et ton enthousiasme ; je te remercie à nouveau de m'avoir appris le Snowboard alors que tu aurais pu perdre patience. Merci également à Nicolas (dit « prospect »), nos sessions golf et street golf resteront dans ma mémoire, elles m'ont permis de décompresser quand j'en avais besoin. Je n'oublie pas non plus de remercier Boris, Lamia, Guillaume, Grégory, Abir et Jean-Phillipe avec qui j'ai également partagé mon quotidien pendant ces trois ans.

Un grand merci à mes fidèles amis : Fabrice, Héloïse, Vincent, Stéphanie, Paul, Julie, Alice, Christophe, Max, Guillaume et Xavier. Vous avez contribué indirectement à ce travail de longue haleine en étant vous-mêmes et en me permettant d'échapper de temps en temps à mes éternels questionnements.

Je dédie cette thèse à mes parents et à mon frère. Je vous remercie d'avoir éveillé mon intérêt pour les études et de m'avoir soutenu à 100% dans mon cheminement. Pour cela je vous serai éternellement reconnaissant. Je sais d'où je viens et à qui je le dois.

Finalement, *last but not least*, merci à toi Laetitia de partager ma vie et de me combler par ton amour et ta gentillesse au quotidien. Ton soutien sans égal durant cette dernière année de thèse m'a été très précieux.

Lyon, le 9 septembre 2012.

Table des matières

Table des figures

Liste des tableaux

Introduction

Contexte

Le stockage de données décentralisé est au cœur des problématiques de l'informatique contemporaine. Cette dernière doit être en mesure de faire face à un flot colossal d'information à archiver, de manière sûre et sécurisée. Nous sommes actuellement en pleine ère de l'informatique dans les nuages, que l'on nomme également *Cloud Computing*. Ce concept, qui consiste à déporter un maximum de traitements sur des grilles de serveurs, pour offrir aux clients un accès universel à leurs ressources de n'importe où, est le résultat d'une évolution qui a débuté il y a plus de 20 ans. Le modèle centralisé, qui était jusqu'à la fin des années 1990 le modèle de référence pour le stockage de données, est devenu obsolète face à la croissance très rapide d'internet. L'augmentation conjointe du nombre d'utilisateurs et du volume de données à stocker au-dessus des capacités des serveurs, ont poussé les différents acteurs du stockage de données à considérer d'autres approches. Le modèle pair-à-pair s'est très rapidement distingué –par le biais de plateformes célèbres telles que Napster ou eMule– comme une alternative fonctionnelle, robuste et à faible coût pour le stockage et l'échange décentralisé de données. La philosophie derrière les architectures pair-à-pair consiste à répartir le service sur les clients (que l'on appelle pairs). Chaque pair peut alors occuper la fonction de client ou de serveur en fonction des requêtes qu'il reçoit. Dans les premiers types de réseaux pair-à-pair, seul le transfert de l'information était décentralisé alors que l'indexation était, quant à elle, centralisée pour des questions d'efficacité des requêtes de recherche. Les architectures pair-à-pair totalement décentralisées n'ont vu le jour qu'à l'issue d'un processus d'évolution –mené en grande partie par la recherche scientifique– avec l'arrivée des tables de hachage distribuées.

La disponibilité des données stockées dans les architectures décentralisées en pair-à-pair s'obtient par la mise en place de schémas de l'information efficaces. Ces systèmes distribués sont, en effet, constitués d'un très grand nombre de machines hétérogènes sur lesquelles il n'existe aucun contrôle et qui sont inévitablement victimes de fautes. Chaque système de stockage doit donc s'assurer que les données qu'il héberge sont suffisamment redondantes pour ne jamais être perdues. La redondance consiste à dupliquer de l'information identique sur différents pairs pour tolérer un certain nombre de fautes. Ce niveau de redondance est calculé en fonction de la disponibilité souhaitée dans le système ainsi que par le nombre de fautes observées. Néanmoins, la redondance ne saurait se suffire à elle-même pour garantir la disponibilité des données au cours

du temps. C'est pourquoi des politiques de supervision et de réparation des données sont mises en place. Ces politiques sont appliquées par les pairs sur les données et sont souvent spécifiques à chaque architecture.

Thèse

L'application des politiques de disponibilité est un processus rigide et propre à chaque architecture. La plupart d'entre-elles proposent un paramétrage *ad-hoc* pour le réseau considéré et fixé *a priori*. L'orientation que nous prenons dans cette thèse est totalement différente. Nous proposons de transférer la responsabilité de l'application de ces politiques des pairs aux documents. Dans cette nouvelle vision, les documents deviennent des entités autonomes qui décident d'appliquer leurs propres politiques de disponibilité en fonction de leurs propres contraintes. Ce basculement est réalisé par l'introduction des concepts d'entité autonome et de mobilité. L'autonomie et la mobilité sont obtenues à l'aide d'une modélisation du système sous forme de système multi-agents dans lequel un ensemble d'agents mobiles est déployé dans le réseau. Plus précisément, dans notre approche, une donnée est fragmentée en plusieurs éléments redondants. Chacun de ces éléments est ensuite encapsulé dans un agent mobile, autonome et capable de se déplacer dans le réseau. L'autonomie est vue ici comme la capacité que possède un agent à prendre des décisions et à effectuer des actions en fonction de la situation qu'il observe. Les déplacements des agents d'une donnée ne peuvent pas être totalement libres. Pour des questions de décentralisation des traitements, les agents d'une donnée doivent conserver un fort degré de localité entre-eux malgré leurs déplacements. C'est-à-dire qu'ils doivent se déplacer en groupe. L'obtention d'un tel comportement est le résultat d'algorithmes bio-inspirés qui transforment ces groupes d'agents mobiles en véritables nuées (par analogie avec les nuées d'étourneaux dans le vivant). L'émergence de ce comportement est le résultat de l'application locale et asynchrone de règles simples.

Cette thèse s'intéresse à la viabilité d'une telle approche et aux mécanismes à mettre en place pour assurer cette viabilité dans un contexte réel. Elle s'intéresse également au caractère adaptatif et flexible de ces nuées d'agents mobiles, dans une optique de stockage décentralisé. Nos contributions peuvent être regroupées en trois grands points :

1. l'étude de la viabilité du modèle de flocking d'agents mobiles et des paramètres influant sa dynamique : sous quelles conditions les nuées gardent-elles leur cohésion ? Quelle est la distribution des déplacements d'une nuée ? Quels facteurs extérieurs ont une influence sur sa dynamique ? Existe t-il des cas pour lesquels le flocking n'émerge pas ?
2. l'étude de l'adaptation d'une nuée à son environnement en recherchant ses paramètres de fragmentation optimaux par l'exploration. Nous cherchons à tirer partie des capacités exploratoires et adaptatives d'une nuée pour qu'elle arrive à trouver elle-même sa taille adéquate, en fonction de critères tels que son seuil de réparation ou encore des propriétés du réseau dans lequel elle évolue ;

3. l'étude du placement adaptatif et décentralisé appliqué à la problématique des fautes corrélées : nous proposons l'algorithme MinCor (*Minimal Correlations*) qui consiste à contrôler les nuées de manière décentralisée pour qu'elles trouvent un placement dans le réseau qui optimise un certain nombre de critères. Nous montrons le gain apporté par cette algorithme qui est facilement adaptable à d'autres types d'optimisation.

Organisation du document

Ce manuscrit est organisé en deux grandes parties. Dans la première partie consacrée à l'état de l'art, nous aborderons au premier chapitre, les différentes architectures pair-à-pair ainsi que leur évolution, du modèle centralisé au modèle totalement décentralisé. Nous aurons notamment l'occasion de présenter les différentes approches, leurs forces et leurs faiblesses. Le second chapitre traite de la disponibilité de l'information dans les plateformes de stockage en pair-à-pair. Nous verrons les différences qui existent entre les plateformes d'échange en pair-à-pair (telles que Napster, eMule ou Bittorrent) et les plateformes de stockage (telles que Wuala, TotalRecall et Glacier). C'est dans ce chapitre que nous décrirons l'ensemble des mécanismes nécessaires pour assurer la disponibilité et la durabilité des données stockées dans de tels systèmes dynamiques et sujets aux fautes. Ce sera l'occasion pour nous de faire un inventaire des plateformes de stockage en pair-à-pair existantes, en décrivant, autant que faire se peut, les mécanismes qui ont été retenus dans chacune d'entre-elles. Cet inventaire nous permettra de mieux situer notre propre approche par rapport à l'existant. Le dernier chapitre d'état de l'art fait une présentation des systèmes multi-agents et de la technologie des agents mobiles. Il décrit notamment l'approche collective que nous avons retenue pour modéliser notre système.

La seconde partie de ce document est entièrement consacrée à nos contributions. Nous présentons au chapitre 4 le modèle de nuées d'agents mobiles ainsi que les algorithmes qui régissent le comportement de nos agents. Nous menons également, dans ce chapitre, un certain nombre de simulations et d'expérimentations sur un prototype qui valident nos algorithmes dans un réseau réel. Le chapitre 5 s'attarde sur les mécanismes décentralisés qui permettent de rendre une nuée robuste et tolérante aux fautes. Il est notamment question, dans ce chapitre, de l'impact négatif des ruptures de la cohésion sur la disponibilité des nuées. Nous verrons que cet impact peut être réduit, voir supprimé, lorsque les nuées ont une taille suffisante par rapport à leur seuil de réparation. Nous verrons également qu'une nuée est capable de trouver cette valeur en explorant son environnement. Le chapitre 6 est consacré à l'impact des fautes corrélées sur les systèmes de stockage. Plus précisément, il présente MinCor, un algorithme de contrôle décentralisé de nuées. Cet algorithme de recuit-simulé distribué cherche un placement qui minimise les corrélations de fautes entre les agents. Ce placement repose sur un clustering préalable du réseau qui regroupe les pairs corrélés entre-eux. L'algorithme MinCor cherche ensuite à placer les agents d'une nuée sur des clusters disjoints. Cette optimisation est une nouvelle illustration convaincante des possibilités d'adaptation d'une nuée à son environnement. Enfin, nous clôturons nos travaux par le chapitre 7, qui est un chapitre de travaux inachevés et de perspectives

sur le passage à l'échelle de l'architecture. Cette réflexion nous amène à considérer de nouvelles architectures réseaux pour réduire le coût de fonctionnement des nuées, qui pourront déboucher sur de futures recherches.

Première partie

État de l'art

Chapitre 1

Les réseaux pair-à-pair

Sommaire

1.1 Du modèle client-serveur au pair-à-pair

Le modèle pair-à-pair émerge à la fin des années 1990 en réponse aux problématiques engendrées par l'explosion du nombre d'usagers d'internet (cf. Fig. 1.1). Il vient compléter l'approche client-serveur lorsque le nombre d'utilisateurs d'un service devient très grand et que ce service nécessite une utilisation intensive de ressources (temps CPU, espace de stockage, accès disques, bande passante, etc.) pouvant le mener vers un déni de service (DoS) quand la charge dépasse sa capacité.

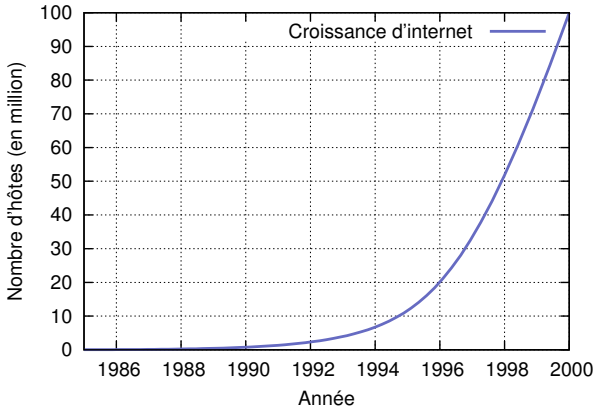
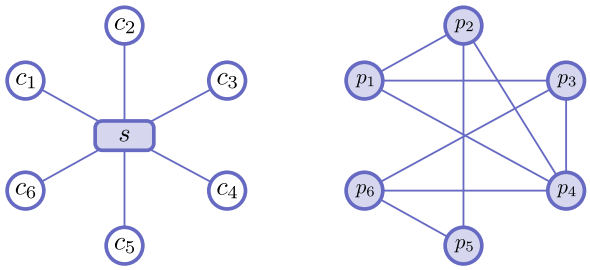


FIGURE 1.1 – Croissance d'internet entre 1985 et 2000 (sources : [Zakon, 1997] et [Zakon, 2011]).

Dans l'architecture client-serveur (cf. Fig. 1.2(a)), un serveur s fourni un service réseau à un ensemble de clients c_i par le biais d'une liaison point à point (unicast). La disponibilité du service est entièrement dépendante de s et cette architecture présente un certain nombre de vulnérabilités inhérentes à son aspect centralisé dont les plus critiques sont :

- la perte de s suite à une faute (panne de l'hôte, déconnexion réseau, attaque, etc.) rend indisponible le service à l'ensemble des clients ;
- la surcharge de s suite à un trop grand nombre de connexions peut entraîner un déni de service. i.e., s n'est plus en mesure de traiter les requêtes et le service devient indisponible ;
- la corruption de s par un tiers malveillant a des répercussions sur l'intégralité des clients du service.

Le modèle pair-à-pair (cf. Fig. 1.2(b)) propose une décentralisation du service en mettant à contribution les clients qui deviennent à leur tour chacun des fournisseurs d'une partie du service. Dans cette architecture, les pairs p_i sont à la fois clients et serveurs. Nous verrons dans la suite de ce chapitre qu'en fonction des approches qui sont proposées, les rôles des pairs sont plus ou moins homogènes et que la transition du modèle client-serveur pur au pair-à-pair totalement décentralisé s'est faite en plusieurs étapes. Le choix du type d'architecture est en réalité dépendant du type de service que l'on souhaite déployer. C'est dans ce contexte de mutualisation et de partage de ressources que se sont développées les applications pair-à-pair large échelle tels que le partage de contenu (Napster, BitTorrent, Gnutella, eMule, KaZaa, ...), la



(a) Architecture client-serveur.

(b) Architecture pair-à-pair.

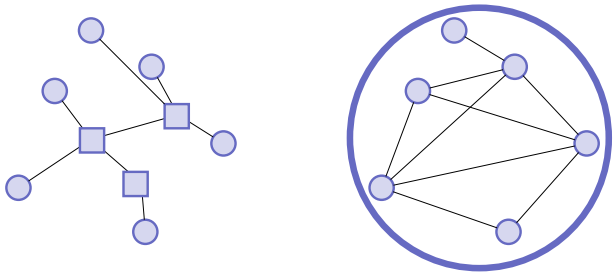
FIGURE 1.2 – Modèle client-serveur et modèle pair-à-pair.

diffusion de flux (PPTV, Joost, TVants, PPLive, SopCast, ...), le multicast applicatif (Scribe, Bayeux, NICE, OMNI, SCAMP, OverCast,...), la voix sur p2p (Skype), le stockage persistant (Oceanstore, Wuala, BitVault, Glacier, TotalRecall, ...) ou encore les tables de hachage distribuées (CAN, Chord, Tapestry, ...). De part sa décentralisation, le pair-à-pair a également favorisé le développement d'applications résistantes à la censure et proposant des niveaux d'anonymat à leurs utilisateurs. Plusieurs définitions du pair-à-pair ont été proposées dans la littérature. [Buford et Yu, 2010] en identifient deux qui couvrent les concepts de partage de ressources, d'auto-organisation, de décentralisation, de topologies et de réseaux logiques :

1. selon [Androutsellis-Theotokis et Spinellis, 2004] : « Les systèmes pair-à-pair sont des systèmes distribués qui consistent en une interconnexion de nœuds capables de s'auto-organiser en topologies réseau dont le but est de partager des ressources tel que du contenu, des cycles CPU, de l'espace de stockage et de la bande passante. Ces réseaux sont capables de s'adapter aux fautes et de tolérer les groupes de nœuds ayant un comportement instable tout en conservant un niveau de connectivité et de performances acceptable sans recourir au support d'un serveur global centralisé ni d'une quelconque autorité. » ;
2. selon [Buford *et al.*, 2008] : « Un réseau logique [1] est une couche réseau applicative virtuelle dans laquelle les nœuds sont adressables, qui fournit la connectivité, le routage ainsi que l'acheminement des messages entre les nœuds. Les réseaux logiques sont fréquemment utilisés comme support au déploiement de nouveaux services réseaux ou pour fournir une structure de routage qui n'est pas accessible à partir du réseau physique sous-jacent. Beaucoup de systèmes pair-à-pair sont des réseaux logiques qui fonctionnent au-dessus d'internet. ». Dans l'illustration de la Fig. 1.3, un réseau logique recouvre le réseau physique définissant de ce fait une nouvelle topologie virtuelle. Nous ajoutons à cette définition qu'un réseau logique définit une structure de voisinage entre les pairs.

Cependant, ce passage à la décentralisation, bien qu'apportant un certain nombre de solutions aux verrous posés par les architectures centralisées, soulève de nouvelles problématiques dont voici un aperçu :

1. On trouve également : *réseau de recouvrement* ou *overlay network*.



(a) Réseau physique d'hôtes (les cercles) et de routeurs (les carrés). Liaisons matérielles.

(b) Réseau logique interconnectant les pairs au dessus du réseau physique. Liaisons virtuelles.

FIGURE 1.3 – Réseau physique et réseau logique.

Problèmes de disponibilité de l'information : lorsqu'une donnée/ressource est distribuée/mise à disposition par des pairs, comment la trouver efficacement ? Comment s'assurer qu'elle n'est jamais perdue étant donné qu'il n'y a aucun contrôle possible sur les pairs ?

Problèmes de construction et de maintien du réseau : comment construire un réseau de manière décentralisée pour qu'il ait les propriétés voulues ? Comment gérer les connexions/déconnexions des pairs afin de conserver ces propriétés et notamment la propriété de connexité du réseau ?

Problèmes de passage à l'échelle et d'équilibrage de charge : quels mécanismes faut-il mettre en place pour que le fonctionnement du réseau passe à l'échelle ?

Problèmes de confidentialité et d'authenticité : lorsque des informations personnelles sont partagées dans de tels réseaux, comment s'assurer de leur confidentialité et de leur authenticité ?

Nous balayons dans ce chapitre d'état de l'art sur les réseaux pair-à-pair un ensemble d'architectures et de topologies pair-à-pair qui ont marqué la communauté de par leur innovation ou leur popularité. Ces architectures répondent à plus ou moins de problématiques parmi celles que nous avons énoncées ci-dessus en fonction des objectifs des applications qu'elles supportent. Nous verrons notamment que l'on peut classifier ces architectures en plusieurs catégories en fonction de leur degré de décentralisation. Ce degré de décentralisation est lié aux contraintes de la couche applicative ainsi qu'aux verrous technologiques qui existaient au moment de la création de ces architectures. Dans ce chapitre, nous nous intéressons plus particulièrement à l'aspect topologique des solutions proposées plus qu'au stockage robuste et persistant de l'information en pair-à-pair qui fera l'objet du chapitre 2.

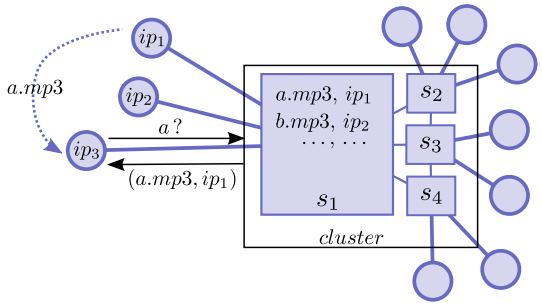


FIGURE 1.4 – Organisation de Napster.

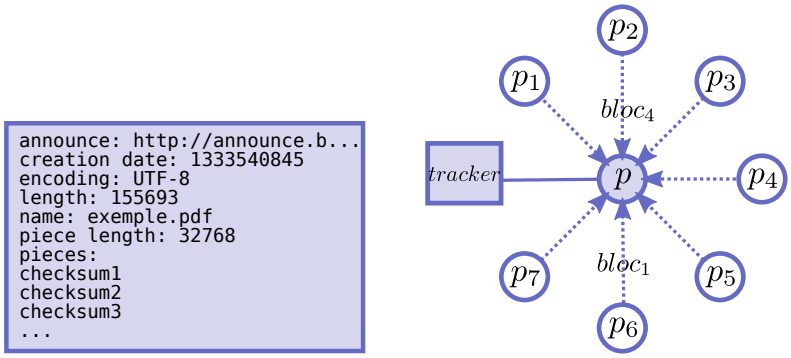
1.2 Pair-à-pair centralisé

La première génération de réseaux pair-à-pair voit le jour en 1999 avec l'arrivée de Napster [Napster, 1999] puis en 2000 avec BitTorrent [Cohen, 2008]. Elle est dite centralisée parce que le fonctionnement de ces protocoles est dépendant de serveurs. En réalité, les protocoles de première génération présentent un début de décentralisation puisque l'échange d'information est effectivement fait de pair à pair. Par contre des serveurs jouent le rôle d'index pour chercher les données et mettre en relation les pairs.

1.2.1 Napster

Napster [Napster, 1999] était une plateforme d'échange de fichiers musicaux de type mp3 crée par Shawn Fanning et Sean Parker dans laquelle chaque pair pouvait mettre à disposition le contenu de sa bibliothèque musicale. Dans cette architecture, chaque pair envoie la liste de ses fichiers à un serveur qui stocke l'association *adresse du pair* $\leftrightarrow$ *données + métadonnées*. C'est en réalité un cluster d'environ 160 serveurs selon [Saroiu *et al.*, 2003]. Lorsqu'un pair recherche un nom de fichier, il envoie une requête sous forme de mots clés au serveur d'index auquel il est connecté. Ce dernier lui retourne une liste de n-uplets *(ip, fichier, metadonnées)*. Une fois que l'utilisateur a choisi la donnée $fichier_i$ qui l'intéresse, l'échange se fait directement entre le pair demandeur et l'hébergeur ip_i. Quand le fichier est téléchargé, le pair demandeur devient à son tour un hébergeur qui est indexé dans le serveur central pour cette donnée. Cette procédure est schématisée à la Fig. 1.4 dans laquelle le pair ip_3 envoie au serveur s_1 une requête de recherche contenant le mot clé "a". s_1 envoie la réponse (ip_1, $a.mp3$, ...) à ip_3 qui peut ensuite contacter ip_1 et initier le transfert pair-à-pair de $a.mp3$.

Cette architecture fournit un mécanisme de recherche efficace puisqu'il consiste en la simple interrogation d'un index. Elle est également tolérante au *churn* (les perturbations engendrées par les connexions/déconnexions de pairs) et allie expressivité des requêtes à l'exhaustivité des réponses. Cependant, l'aspect centralisé, bien que relativement résistant au déni de service grâce à l'utilisation de cluster reste vulnérable à la censure. A titre d'exemple, la version gratuite et publique de Napster a été fermée en 2001 suite à une décision de justice et a signé l'arrêt total du service puisque les serveurs le faisant fonctionner ont été déconnectés.



(a) Exemple de contenu d'un fichier de méta-données torrent.

(b) Swarm : le pair p se connecte au tracker et récupère les pairs p_i d'un torrent. Ces pairs lui envoient ensuite en parallèle les blocs qu'ils possèdent.

FIGURE 1.5 – Organisation de BitTorrent.

1.2.2 BitTorrent

BitTorrent [Cohen, 2008] est un protocole créé par Bram Cohen mais également un client pair-à-pair [BitTorrent, 2012]. Son objectif est de fournir la possibilité à n'importe qui de déployer une infrastructure pair-à-pair sur ses propres serveurs. Dans BitTorrent, un serveur (le *tracker*) répertorie un certain nombre de ressources à partager (les *torrents*). Les pairs connectés au tracker s'abonnent aux torrents qui encapsulent les ressources qu'ils souhaitent récupérer. Un torrent est en fait un fichier contenant les méta-données de la ressource ainsi que l'adresse du tracker qui l'héberge. Il est donc en général mis à la disposition des pairs par son propriétaire au moyen d'un hyperlien sur un site web connu. Dans le protocole BitTorrent les données sont découpées en blocs de tailles fixes choisies entre 64Ko et 4Mo. Le fichier torrent contient la liste des blocs ainsi que leur somme de contrôle de type SHA-1 [SHA, 1995]. Lorsqu'un pair p souhaite télécharger une donnée d, il récupère le torrent de d et se connecte au tracker correspondant. Ce tracker lui retourne un sous ensemble p_i de pairs parmi tous ceux déjà abonnés au torrent de d (le *swarm*). p peut ensuite récupérer le document en se connectant aux pairs p_i et en téléchargeant les blocs de la donnée qu'ils possèdent tout en s'assurant que les sommes de contrôle sont valides. La Fig. 1.5(a) donne un exemple de fichier torrent encapsulant les méta-données d'un fichier *exemple.pdf*. La Fig. 1.5(b) quand à elle décrit l'organisation de l'overlay dans BitTorrent ; p a récupéré les pairs p_i après une requête au tracker et peut récupérer les blocs de la donnée en parallèle.

Comme dans Napster, l'approche BitTorrent est centralisée. Elle permet un accès rapide et efficace aux ressources par interrogation du tracker. De plus, l'ajout du téléchargement multi-source a grandement accéléré les temps de téléchargement. Par contre, le tracker étant un point central, BitTorrent souffrait dans ses versions initiales des mêmes inconvénients que Napster dans le sens où l'indisponibilité d'un tracker ne permettait plus de mettre en relation les pairs inscrits

à ses torrents. Cependant, dans les dernières versions du protocole, les pairs se connectent également à des tables de hachage distribuées (cf. Sec. 1.5) qui jouent le rôle de trackers décentralisés. Ces tables sont utilisées lorsque les trackers centraux deviennent indisponibles pour éviter que le service ne soit interrompu. BitTorrent est un réseau pair-à-pair populaire et toujours utilisé actuellement sur internet. Son succès et sa résistance à la censure sont notamment dus au fait que n'importe qui peut déployer un tracker robuste mais également que sa spécification ouverte a favorisé plusieurs implémentations (dont certaines sont open-source) compatibles entre-elles.

1.3 Pair-à-pair semi-centralisé et hybride

La première génération de réseaux pair-à-pair que nous venons de présenter a permis d'alléger les serveurs en déportant la charge du transfert des données sur les pairs. Mais face aux vulnérabilités des serveurs centraux une autre génération de réseaux pair-à-pair est apparue avec les protocoles eDonkey [Heckmann et Bock, 2002] et Fastrack [Liang *et al.*, 2006] qui proposent deux mécanismes différents pour diminuer ce degré de centralisation. Dans le premier, l'idée est de déployer plusieurs clusters de type Napster distincts. Le second consiste à créer une topologie sans serveurs basée sur une organisation hiérarchique des pairs.

1.3.1 eMule et eDonkey

Le protocole eDonkey [Heckmann et Bock, 2002, Heckmann *et al.*, 2004] voit le jour en l'an 2000 implémenté dans le client eDonkey2000 puis remplacé par le client eMule [Kulbak et Bickson, 2004] en 2002. Dans cette architecture de partage de documents, le contenu mis à disposition par l'ensemble des pairs est indexé par des clusters de serveurs distincts bien connus [2]. Le schéma de la Fig. 1.6 présente un exemple d'architecture eDonkey à deux clusters dans laquelle la recherche du fichier *a.pdf* par le pair p_1 est restreinte au réseau 2. Dans ce protocole les fichiers sont hachés [3] et découpés en parties, elles-mêmes découpées en blocs de taille fixe. Cette taille est dépendante du payload TCP [Postel, 1981] de sorte qu'un bloc de donnée tienne dans un paquet TCP pour éviter la fragmentation. Par conséquent, des fichiers avec des noms différents mais avec le même contenu seront considérés comme identiques et *vice versa*. Ce découpage en parties permet aux pairs de télécharger des parties en parallèle et d'accélérer ainsi les transferts. La recherche de données dans eDonkey est d'abord effectuée sur le serveur auquel est connecté le pair. Si cette recherche est infructueuse, elle peut être étendue aux autres serveurs du cluster jusqu'à trouver des sources. Dans l'exemple de la Fig. 1.6, le pair p_1 qui recherche le fichier *a.pdf* interroge d'abord le serveur s_1 qui n'a pas l'information et la propage à ses voisins. Le fonctionnement d'eDonkey est relativement proche de celui de BitTorrent à la différence que le protocole n'est pas ouvert et que le nombre de serveurs n'est pas aussi important. Dans cette architecture semi-centralisée, la perte d'un serveur ne pénalise pas tout le système. Dans le cas

2. Citons par exemple les serveurs Razorback, DonkeyServer1 et DonkeyServer2. Razorback2 indexait à lui seul le contenu de plus d'un million de pairs.

3. La fonction de hachage utilisée, le hash ed2k, se base sur la fonction de hachage MD4 [Rivest, 1992].

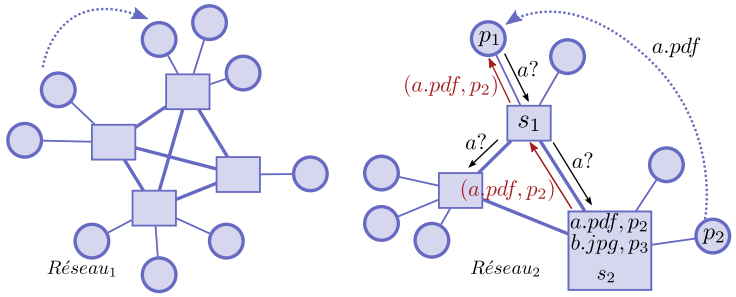


FIGURE 1.6 – Organisation d'eDonkey en plusieurs réseaux disjoints.

où c'est tout un cluster qui est mis hors ligne, les pairs peuvent toujours se connecter à un autre cluster bien connu. Même si le protocole n'est pas ouvert, le code du serveur a néanmoins été distribué sur internet par reverse engeneering sous le nom de serveur Lugdunum. C'est pourquoi, même si la société MetaMachine, propriétaire d'eDonkey, a du cesser toute activité en 2005 suite à une décision de justice [4], des serveurs eDonkey non-officiels sont toujours actifs et le réseau continue d'exister.

1.3.2 FastTrack et KaZaA

Le second mécanisme que l'on caractérise d'hybride n'a plus recours à des serveurs publics pour fonctionner et introduit une hiérarchie de rôles entre les pairs. Dans FastTrack [Liang *et al.*, 2006], protocole propriétaire créé en 2001 et implémenté dans les logiciels KaZaA [Liang *et al.*, 2004], iMesh [iMesh, 2012] et Grokster [Grokster, 2005], certains pairs (les *super nodes* ou SN) supportent la charge d'indexation des autres pairs (les *ordinary node* ou ON). Dans cette approche, les pairs les plus puissants en terme de bande passante, d'espace, de mémoire, d'uptime, etc. peuvent s'auto-élire SN et prendre en charge les requêtes de recherche et d'index d'un certain nombre d'ON. Lorsqu'un pair rejoint un réseau FastTrack, il contacte un SN et lui envoie son contenu à indexer. Par conséquent, lorsqu'un ON s'auto-élit SN, il est déjà connecté à un SN. La structure de FastTrack est donc composée d'une interconnexion de SN auxquels sont attachés des feuilles de type ON. La Fig. 1.7 présente un exemple de réseau fonctionnant avec FastTrack. Dans cet exemple, le pair ON_1, connecté au SN_5 met à disposition la donnée *a.pdf* qu'il indexe avec ses métadonnées sur SN_5. Lorsqu'un pair ON_2 recherche cette donnée, il adresse une requête de recherche à son SN (SN_1 ici) qui est propagée par inondation dans l'interconnexion de SN jusqu'à ce que SN_5 soit trouvé. Une fois trouvé, la réponse remonte et le transfert pair-à-pair entre les deux ON peut commencer. Les méta-données contiennent notamment le haché du fichier pour que la localisation de ses blocs soit possible une fois que l'ON a choisi le fichier qu'il souhaite récupérer. Dans KaZaA, les SN conservent une liste d'autres SN et reconfigurent leurs voisinages SN-SN toutes les dix minutes pour permettre une plus grande

4. Ce n'est pas le protocole en tant que tel qui est mis en cause mais son usage. En effet, il a été décidé que les serveurs d'eDonkey enfreignaient la loi en indexant du contenu protégé par le droit d'auteur et ont été mis en demeure.

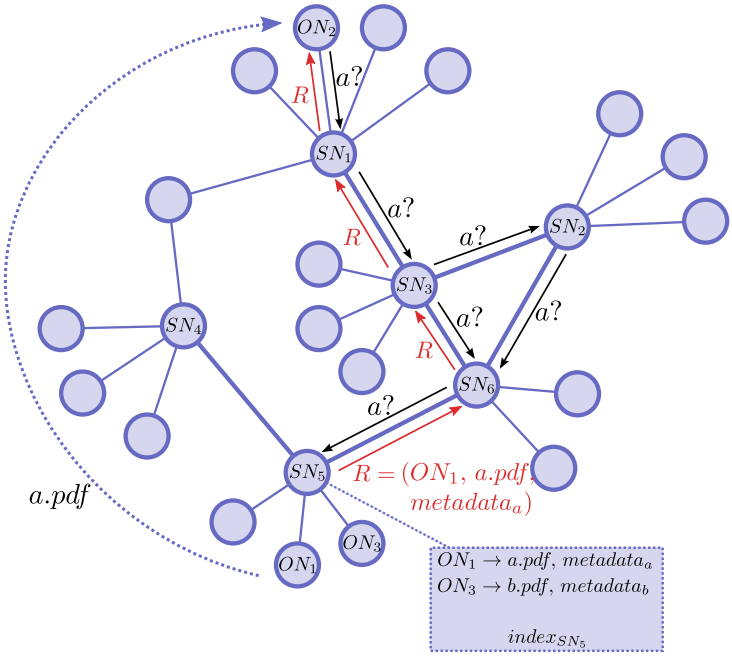


FIGURE 1.7 – Organisation de FastTrack.

couverture du réseau tout en conservant des temps de recherche courts. FastTrack a su exploiter l'hétérogénéité des pairs pour fournir un niveau de décentralisation supérieur à celui d'eDonkey puisque tout pair peut potentiellement devenir un organe d'indexation. De plus, contrairement à une approche totalement décentralisée comme nous le verrons avec Gnutella (cf. Sec. 1.4.1), la recherche se fait de manière efficace sur un sous-ensemble du réseau. Ce type d'architecture hybride a su allier robustesse et efficacité, ce qui lui a valu en 2004 d'être l'architecture pair-à-pair la plus utilisée dans le monde [Gauron, 2006]. Il faut cependant noter que l'auto-élection peut poser problème si trop (ou à l'inverse pas assez) de SN sont élus. En 2005, certains usagers du réseau FastTrack/KaZaA font l'objet de poursuites judiciaires aux États-Unis et en Australie provoquant l'écroulement du réseau suite au départ des ses utilisateurs. Le modèle hybride reste néanmoins un modèle performant qui passe à l'échelle. C'est pourquoi il est actuellement toujours utilisé dans d'autres applications comme c'est le cas de Skype, le réseau de téléphonie sur IP, qui a atteint les 40 millions d'utilisateurs connectés simultanément le 10 avril 2012 [5].

1.4 Pair-à-pair décentralisé non-structuré

La robustesse des réseaux pair-à-pair hybride est obtenue par une certaine décentralisation et par un nombre suffisant de SN qui permettent au service de continuer à fonctionner même lorsque certains des SN sont déconnectés. Nous avons cependant évoqué le problème du nombre de SN qui doit être régulé pour un fonctionnement optimal du système. Les réseaux décentralisés non-structurés vont plus loin : tous les pairs ont le même rôle que les autres. Cette conception soulève

5. `http://blogs.skype.com/en/2012/04/40_million_people_how_far_weve.html`

de nouvelles problématiques dont la plus délicate est celle de l'efficacité de la recherche (en terme de rapidité mais aussi en quantité de messages). Nous décrivons cette approche à travers deux réseaux pair-à-pair célèbres : Gnutella v0.4 [Gnu, 2003] et Freenet $\leq$ v0.7 [Clarke *et al.*, 2001]. Ces architectures sont dites non-structurées parce que la topologie du réseau résultant de ces protocoles n'est pas régie par des modèles structurels choisis *a priori*.

1.4.1 Gnutella v0.4

Gnutella est créé en 2000 par Justin Frankel et Tom Pepper. Le protocole est initialement fermé et accessible par le client propriétaire du même nom. Cependant, plusieurs clients alternatifs ont rapidement vu le jour (Clip2, Gnucleus, LimeWire, Morpheus, giFT, etc.) et ces implémentations, réalisées par reverse engeneering, ont abouti à une grande incompatibilité des clients [Gauron, 2006]. Face à ce constat, une spécification ouverte a été créée par un certain nombre d'acteurs industriels [Gnu, 2003] pour uniformiser les traitements. Gnutella v0.4 est le premier réseau pair-à-pair décentralisé non-structuré dans lequel les pairs ont le même rôle et indexent leur propre contenu. Lorsqu'un pair souhaite rejoindre le réseau, il trouve un ensemble de pairs déjà connectés, s'y connecte puis envoie une requête de connexion à chacun. Ces requêtes de connexion sont propagées par inondation à tous les voisins. Les inondations dans Gnutella v0.4 sont bornées à 7 sauts (TTL = 7). Chaque pair recevant une requête de connexion choisit d'ajouter le pair initiateur à ses voisins en fonction d'un critère de localité [Lua *et al.*, 2005]. La taille des voisinages n'est ni spécifiée ni bornée et le réseau logique résultant ne présente aucune structure caractéristique. La recherche d'informations dans Gnutella v0.4 s'effectue également par une inondation bornée par le TTL. Lorsqu'une donnée est trouvée, le résultat peut être directement adressé à l'initiateur ou, si ce n'est pas possible (lorsque l'initiateur se trouve derrière un pare-feu notamment) remonter le chemin emprunté par la requête de recherche. Un exemple de réseau Gnutella v0.4 est présenté à la Fig. 1.8. Ce type de recherche est inefficace lors du passage à l'échelle. D'une part, la charge sur chaque nœud croît linéairement avec le nombre total de requêtes et ce nombre de requêtes croît à son tour avec la taille du système [Chawathe *et al.*, 2003]. D'autre part, le fait de fixer un TTL ne permet pas d'atteindre la totalité du réseau lorsque celui-ci a un diamètre important, ce qui a pour conséquence de générer des faux négatifs. On voit donc clairement l'avantage d'un tel protocole en terme de robustesse mais le manque de structure le contraint à utiliser un algorithme de recherche couteux. Pourtant, des améliorations du mécanisme de recherche ont été proposées tels que les expanding rings et les marches aléatoires [Lv *et al.*, 2002] ainsi que des améliorations plus structurelles [Chawathe *et al.*, 2003] telles que le contrôle de flux, l'adaptation dynamique à la topologie réseau ou encore l'indexation des pairs voisins. Cependant, l'arrivée en 2003 de Gnutella v0.6 (i.e., Guntella2) basé sur une architecture hybride a contribué à éclipser petit à petit Gnutella v0.4.

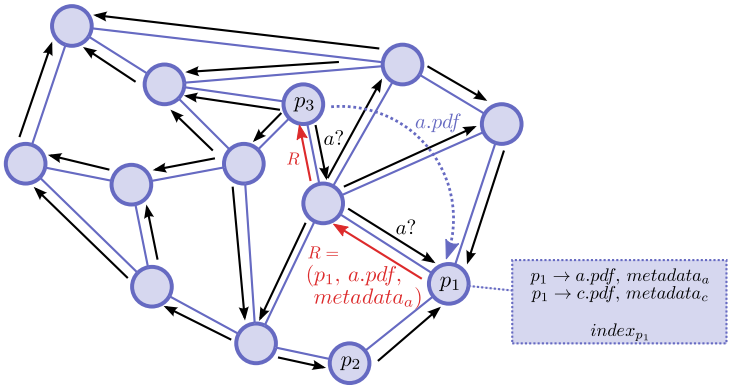


FIGURE 1.8 – Organisation de Gnutella v0.4.

1.4.2 Freenet <v0.7

Freenet [Clarke *et al.*, 2001] dans ses versions inférieures à la 0.7 était un réseau pair-à-pair décentralisé non structuré. Son objectif premier est de fournir une infrastructure offrant un meilleur anonymat aux utilisateurs ainsi qu'une meilleure résistance à la censure par rapport aux architectures existantes. Dans Freenet, lorsqu'un pair souhaite rejoindre le réseau, il choisit un ensemble de pairs auxquels se connecter ainsi qu'un identifiant aléatoire sur un espace d'adressage. Les ressources sont identifiées par une clé définie sur le même espace d'adressage que les pairs, ce qui permet, à l'aide d'une mesure de distance entre les clés et les pairs, de publier une clé sur le pair le plus proche d'elle sur cet espace logique. Ces clés peuvent être de 3 types différents :

1. KSK (Keyword-signed key) : ce sont des clés qui sont générées de manière déterministe à partir d'une description textuelle de la donnée d. Tout pair peut donc recalculer la clé associée à d. `videos,sport,snowboard,the.art.of.flight` est un exemple de description textuelle. On remarque qu'il est tout à fait possible que deux utilisateurs génèrent la même KSK pour deux contenus différents ;
2. SSK (Signed-subspace key) : ce sont des clés qui permettent à un utilisateur d'ajouter son identité à une KSK pour d'une part solutionner le problème de la pollution de clés mais également pour ajouter une authentification au contenu. Avec ce mécanisme, il n'est pas possible de publier une clé sous l'identité d'un autre utilisateur (les utilisateurs ont une identité virtuelle qui correspond à un trousseau de clés asymétriques qui leur est propre) ;
3. CHK (Content-hash key) : c'est une clé qui identifie le contenu d'une donnée et non sa description. Dans Freenet, les documents sont stockés par leur contenu. Un pair qui souhaite mettre à disposition une ressource d_1 associée à une description $SSK(d_1)$ va donc d'abord publier le couple $\langle d_1, CHK(d_1)\rangle$ dans le réseau puis créer une autre ressource d_2 qui contient juste $CHK(d_1)$ et publier le couple $\langle d_2, SSK(d_1)\rangle$. Ce mécanisme permet non seulement de pouvoir gérer plusieurs version d'une donnée mais également de fragmenter une donnée en blocs et permettre le téléchargement multi-sources. Dans ce cas, d_2 contient

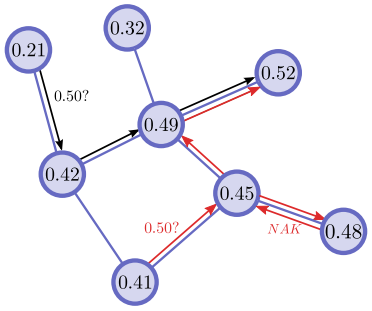


FIGURE 1.9 – Recherche de clé dans Freenet.

la liste de CHK des blocs.

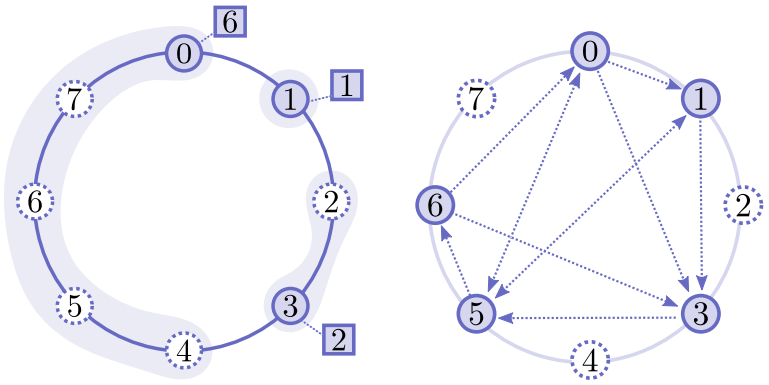
Les ressources sont ensuite signées et chiffrées avec la clé générée. Lorsqu'un pair p_1 souhaite publier une ressource r_1 ayant une clé k_1, il envoie $\langle r_1,k_1 \rangle$ à son voisin p_i qui a l'identifiant le plus proche de k_1 selon la distance XOR (ou exclusif bit-à-bit). Ce choix est appliqué récursivement par chaque pair jusqu'à ce qu'aucun voisin ne soit plus proche de la clé à insérer que le pair courant. Lorsqu'une boucle dans le routage est détectée ou qu'il n'y a plus de successeurs disponibles sur une branche de l'arbre, l'algorithme effectue un backtrack et repart d'un autre voisin. Lors de ce parcours en profondeur dirigé par la métrique XOR vers le pair le plus proche de la clé, le couple $\langle r_1,k_1 \rangle$ est répliqué sur chacun des pairs intermédiaires entre p_1 et le pair cible. Cette réplication transparente permet non seulement de trouver plus rapidement une donnée mais également d'assurer sa persistance en fonction de sa popularité. En effet, dans les autres plateformes de partage que nous avons présentées, les ressources restent stockées sur le pair qui les propose. La persistance des données est donc liée *de facto* à leur popularité au sein de la communauté. Dans Freenet, les ressources sont au contraire insérées quelque part dans le réseau. Sans ce mécanisme de réplication, la popularité d'une donnée ne serait pas prise en compte. Les caches de répliques gérés par chaque pair sont soumis à une politique LRU (*least recently used* : lorsque le cache est plein, la donnée la moins récemment utilisée est supprimée) ce qui permet de définir une taille de cache fixe. Le processus de recherche d'une donnée est analogue et consiste à chercher sa clé en se rapprochant de son pair hébergeur à l'aide de la métrique XOR. La Fig. 1.9 montre un exemple de recherche de la clé d'identifiant 0.50 stockée sur le pair d'identifiant 0.52 à partir de deux pairs distincts 0.21 et 0.41. Une fois la donnée trouvée, elle emprunte le chemin inverse de la recherche et elle est répliquée de manière transparente sur chacun des pairs de ce chemin. Le routage de Freenet est inspiré du routage en oignon [Goldschlag *et al.*, 1999]. En effet les pairs intermédiaires pendant ce routage n'ont aucune information sur l'émetteur, le récepteur ou la donnée elle-même puisque celle-ci est chiffrée. De plus, les pairs ne connaissent pas le contenu des données qu'ils hébergent puisque ces dernières sont chiffrées avec leur clé KSK/SSK qui est stockée ailleurs dans le réseau. *In fine*, le réseau Freenet est un bon exemple de réseau pair-à-pair totalement décentralisé, donc robuste, possédant un mécanisme de recherche plus efficace que l'inondation. Le routage best-effort est effectivement un procédé plus efficace ; cependant, la réplication paresseuse qui s'effectue sur le chemin de retour des re-

quêtes dégrade énormément ses performances. Cette plateforme pair-à-pair est l'une des seules à fournir un certain anonymat, le déni plausible [6] ainsi que l'authenticité des données qui y sont insérées. Elle souffre néanmoins du fait qu'il est difficile de chercher une donnée par mots-clés et que l'utilisation de clés SSK nécessite que les clés soient publiées à un endroit qui en facilite le partage. Pour se faire, Freenet met à disposition de ses utilisateurs des Freesites (des pages web hébergées dans Freenet) qui peuvent servir à partager ces clés. Mais ces usages fastidieux ainsi que les temps de recherche pouvant être longs contraignent Freenet à n'être utilisé que par une communauté restreinte d'utilisateurs pour des usages spécifiques orientés autour de l'anonymat.

1.5 Pair-à-Pair décentralisé structuré

Les réseaux pair-à-pair décentralisés non-structurés sont des réseaux qui sont tolérants aux fautes et à la censure. Contrairement aux modèles centralisés, semi-centralisés et hybrides, ils offrent une bonne répartition des rôles qui leur permet de gagner une très grande robustesse. Mais cette répartition, comme nous l'avons vu avec Gnutella (cf. Sec. 1.4.1) et Freenet (cf. Sec. 1.4.2), fait appel à des mécanismes de recherche coûteux et peu efficaces qui posent des problèmes de passage à l'échelle. C'est en réponse à ces problèmes que les réseaux pair-à-pair décentralisés structurés, qui englobent notamment les tables de hachage distribuées (DHT), ont vu le jour. Ce sont des réseaux dans lesquels les pairs ont les mêmes rôles et qui sont structurés par une topologie connue *a priori* qui contraint les relations de voisinage. Ces contraintes sont choisies pour alléger et accélérer les processus d'insertion et de recherche de données par un routage efficace qui limite le nombre de messages ainsi que la profondeur de la recherche. L'idée maîtresse qui est reprise dans toutes les topologies que nous allons présenter est de construire le réseau logique comme un espace d'adressage muni d'une métrique, dans lequel il est possible d'accéder rapidement à une coordonnée (i.e., le pair occupant cette partie de l'espace d'adressage). Les tables de hachage distribuées sont une catégorie de réseaux décentralisés structurés qui fournissent deux primitives `put` et `get`. Soit une donnée v possédant une clé k définie sur le même espace d'adressage que les pairs, la primitive `put(k,v)` stocke la donnée v sur le pair en charge du sous-espace contenant k. Symétriquement, la primitive `get(k)` récupère la donnée associée à la clé k en interrogeant le pair responsable du sous-espace contenant k. Nous présentons dans cette section quatre DHT –parmi les plus illustres– construites à partir de structures topologiques différentes : Chord [Stoica *et al.*, 2001], CAN [Ratnasamy *et al.*, 2001], Kademlia [Maymounkov et Mazières, 2002] et Tapestry [Zhao *et al.*, 2004]. Nous présentons également la version actuelle de Freenet [Clarke *et al.*, 2010], un exemple de réseau décentralisé structuré aux propriétés de petits mondes.

6. Le déni plausible dans Freenet se matérialise par le fait qu'un pair ne sait pas quelles sont les données qu'il héberge ni l'identité de leur propriétaire. Il ne peut donc pas être considéré comme responsable du contenu qu'il héberge.



(a) Structure d'anneau de Chord.

(b) Tables de voisinages de chaque pair (*finger tables*).

FIGURE 1.10 – Organisation de Chord avec $m = 3$ (les disques pleins représentent les pairs connectés et les carrés représentent les clés hébergées sur les pairs).

1.5.1 Chord

Chord [Stoica *et al.*, 2001] est une DHT et fournit donc, par conséquent, les deux primitives `put` et `get` décrites ci-dessus. Elles requièrent notamment que chaque pair ait une adresse mais également qu'il soit possible de calculer une clé pour chaque document. L'espace d'adressage du réseau Chord est défini sur un anneau des entiers naturels modulo 2^m, où m désigne le nombre de bits sur lequel sont codées les clés. Le contenu des ressources est haché avec l'algorithme SHA-1 ($m = 160$ bits). Le haché sert de clé à la ressource et nous rappelons que les clés et les adresses des pairs partagent le même espace d'adressage. On a donc $key_{data} = \text{SHA-1}(data)$ et $key_{pair} = \text{SHA-1}(IP_{pair})$. Les pairs sont donc projetés sur l'anneau en fonction de leurs identifiants. L'exemple de réseau de la Fig. 1.10(a) présente une telle organisation, avec $m = 3$, dans laquelle seuls les pairs d'identifiant 0, 1 et 3 sont connectés. Chord définit comme premier niveau de relations de voisinage les successeurs dans l'anneau. Dans l'exemple de la Fig. 1.10(a), le successeur du pair 3 est le pair 0. Chaque pair p est responsable du sous-espace d'adressage $]predecessor(key_p), key_p]$. C'est-à-dire qu'il est responsable des clés dont la valeur se trouve dans cet intervalle. Une clé key_{data} est donc assignée au premier pair p qui satisfait $key_p \geq key_{data}$ et un tel pair est noté $successor(key_{data})$. Dans l'exemple, la clé 6 est assignée au pair 0 car il est responsable du sous-espace d'adressage]3,0].

Dans cette version minimale de Chord, insérer ou rechercher une clé nécessite de parcourir l'anneau en visitant les successeurs de chaque pair. Dans l'exemple, si le pair 1 cherche la clé 6 il doit interroger le pair 3 qui lui retournera l'adresse du pair 0. 1 doit ensuite interroger le pair 0 pour récupérer la valeur associée à la clé. Pour des réseaux de grande taille, typiquement lorsque l'espace d'adressage est égal à 2^{160}, cette recherche est lente et inefficace puisqu'elle parcourt au pire cas tous les pairs. L'idée proposée pour accélérer ce traitement est de tirer parti de cette structure d'anneau et de construire des liens supplémentaires qui permettent de faire des sauts

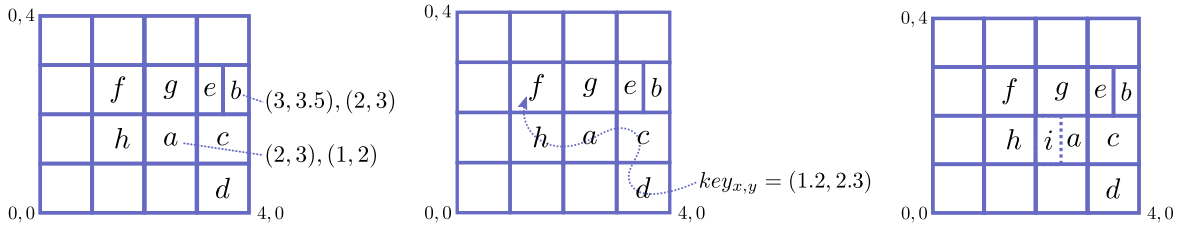
dans l'espace de recherche guidés par la distance. Chaque pair p maintient une table de voisinage (*finger table*) d'au plus m entrées. La i^{e} entrée de cette table pointe vers le premier pair s à une distance d'au moins 2^{i-1} sur l'anneau. Chaque pair p possède donc un ensemble de successeurs s_i tel que $s_i = successor(key_p + 2^{i-1})$, $1 \leq i \leq m$. On remarque que la première entrée de cette table est le successeur direct du pair sur l'anneau. Dans la Fig. 1.10(b), les pairs 5 et 6 ont été connectés au réseau et on a fait apparaître avec les flèches pointillées les tables de voisinages de chaque pair. Il est alors possible d'accélérer la recherche en empruntant ces liens « longue distance ». Le principe de cet algorithme est le suivant : lorsqu'un pair p veut récupérer une clé k, il cherche le pair j qui précède immédiatement k dans sa table de voisinage. p interroge ensuite j et lui demande le prédécesseur immédiat de k dans sa table de voisinages. Ce procédé est itéré jusqu'à trouver j^*, le prédécesseur absolu de k. p peut alors récupérer $successor(j^*)$, le pair qui héberge k. Dans l'exemple de la Fig. 1.10(b), lors que le pair 1 cherche la clé 7, il interroge le pair 5 qui est le prédécesseur de 7 le plus proche dans sa table de voisinage. Le pair 5 lui retourne l'adresse du pair 6 qui est son prédécesseur de 7 le plus proche. Enfin, le pair 1 demande au pair 6 son successeur et trouve la clé.

Lorsqu'un pair rejoint (respectivement quitte) le réseau, il prend en charge (respectivement libère) une partie de l'espace d'adressage, ce qui correspond à une migration de clés. Lorsqu'un pair p rejoint le réseau, il demande à son successeur dans l'anneau l'ensemble des clés k_i telles que $successor(k_i) = key_p$ et en prend la charge. Lorsqu'au contraire un pair p quitte le réseau, il transfert l'intégralité de ses clés à son successeur.

Chord fait appel au hachage consistant [Karger *et al.*, 1997] pour assigner les clés aux pairs. Ce schéma décentralisé assure une bonne répartition des clés sur les pairs ainsi qu'un déplacement limité de celles-ci lors de départs ou d'arrivées de pairs. Les pairs maintiennent une table de voisinage de taille $\log(N)$, avec N le nombre total de pairs du réseau, ce qui permet non seulement d'obtenir une recherche dont la complexité est de l'ordre de $O(\log N)$ mais également une mise à jour des tables de voisinages dont la complexité reste logarithmique, de l'ordre de $O(\log N^2)$.

1.5.2 CAN

CAN [Ratnasamy *et al.*, 2001] est une DHT dont l'espace d'adressage logique est un espace torique à d dimensions en coordonnées cartésiennes. Comme pour Chord, les pairs dans CAN sont responsables d'un sous-espace d'adressage. La Fig. 1.11(a) illustre un CAN défini sur l'espace [0,4] × [0,4]. Dans cet exemple, le pair a est responsable de la zone délimitée par le carré [(2,3),(1,2)] et le pair b de la zone délimitée par le rectangle [(3,3.5),(2,3)]. Pour stocker une donnée dans un CAN, on calcule d'abord sa clé key à l'aide d'un algorithme de hachage puis cette clé est affectée, de manière unique et déterministe, à un point $key_{x,y,...}$ dans l'espace d'adressage du CAN. C'est le pair responsable de la zone logique dans laquelle se trouve $key_{x,y,...}$ qui est responsable de la donnée. Deux zones sont voisines si et seulement si elles ont exactement une seule frontière commune en $d-1$ dimensions. Par exemple à la Fig. 1.11(b), les pairs a et c sont voisins car ils ont une seule frontière commune. Les pairs a et f ne sont pas voisins car ils n'ont



(a) Les pair sont responsables d'une zone de l'espace logique.

(b) Structure des voisinages et insertion de clé.

(c) Ajout d'un pair i au CAN.

FIGURE 1.11 – Organisation de CAN.

pas de frontière commune.

L'insertion et la recherche d'une clé sont basées sur un algorithme de routage glouton. Le pair source va chercher un voisin dont la coordonnée se rapproche le plus de la coordonnée de la cible selon la distance euclidienne. Ce procédé est répété jusqu'à arriver dans la zone cible (i.e., sur le pair qui en est responsable). Par exemple dans la Fig. 1.11(b), le pair d veut insérer la clé key qui a pour coordonnées le point $key_{x,y} = (1.2,2.3)$. Il choisi donc le voisin qui le rapproche le plus de la zone cible en l'occurrence c. c choisi ensuite a puis a choisi h qui choisi à son tour f qui est le propriétaire de la zone et qui doit stocker key.

Le protocole de connexion d'un pair p au réseau est réalisé en plusieurs étapes :

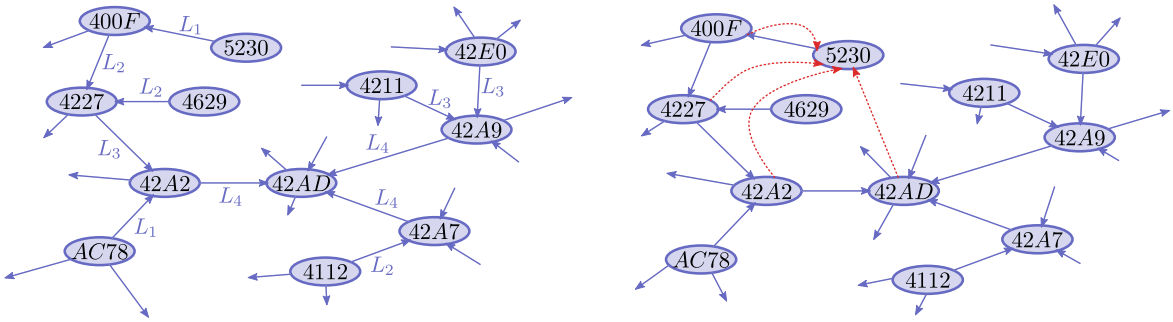
1. trouver un pair α déjà connecté au réseau et lui demander les propriétés de l'espace d'adressage. p choisi ensuite aléatoirement une coordonnée dans cet espace ;
2. p cherche ensuite, à partir du pair α, la zone du CAN qui contient sa coordonnée avec le même mécanisme de recherche que celui de l'insertion de clé. L'occupant p' actuel de cette zone la sépare en deux et attribue la moitié correspondante à p ainsi que les clés dont il est responsable ;
3. enfin, pour que p puisse participer au routage, les voisinages sont mis à jour à partir des voisins de p'.

Par exemple dans la Fig. 1.11(c), le pair i se connecte au CAN dans la zone initialement affectée au pair a. a envoie à i les données qu'il doit prendre en charge. Lors du départ d'un pair, les opérations inverses sont effectuées.

En termes de performances, dans CAN, chaque pair maintient une table de voisinages de taille $2d$ et la complexité de la rechercher/insertion est de l'ordre de $O(dN^{1/d})$ avec N le nombre total de pairs. La mise à jour des voisinages lors du départ et de l'arrivée d'un pair impliquent $2d$ pairs voisins.

1.5.3 Tapestry

Tapestry [Zhao *et al.*, 2004] est une DHT basée sur un réseau logique appelé DOLR [Hildrum *et al.*, 2002], un maillage inspiré par le maillage de Plaxton [Plaxton *et al.*, 1997]. Cette structure permet notamment d'obtenir une publication ainsi qu'une recherche efficace de l'information.



(a) Structure du réseau logique

(b) *Location mapping* dans Tapestry.

FIGURE 1.12 – Organisation de Tapestry.

Les adresses dans Tapestry prennent leurs valeurs dans un espace de 160 bits et sont codées dans une base $\beta = 16$ (hexadécimale) ce qui génère des identifiants de $b = 40$ digits. Tapestry fait l'hypothèse que les clés (et les identifiants de pairs) sont réparties uniformément sur cet espace en suggérant d'utiliser la fonction de hachage SHA-1. Les tables de voisinages des pairs sont organisées en b niveaux. Chaque niveau i contient β pairs voisins dont l'identifiant a un préfixe commun de taille $i - 1$. Par exemple dans un espace d'adressage où $b = 5$, un pair d'identifiant $58B69$ pourrait avoir dans son 3^{e} niveau les pairs d'identifiant $58A23$ et $58C59$. La Fig. 1.12(a) donne un exemple de maillage Tapestry partiel en tables de voisinages multi-niveaux. Le routage d'une clé k est préfixe. C'est-à-dire que l'identifiant p_n du n^{e} pair sur le chemin de la source à la cible partage un préfixe de taille supérieure ou égale à n avec la clé. Lorsqu'il doit router une clé, un pair p_n cherche dans le $n + 1^{\text{e}}$ niveau de sa table de voisinage un pair p_{n+1} tel que $p_{n+1}[n] = k[n]$. Dans l'exemple de la Fig. 1.12(a), le pair d'identifiant $p_0 = 5230$ qui souhaite insérer la clé d'identifiant $k = 42AF$ va d'abord chercher dans son niveau L_1 un pair p_1 qui vérifie $k[0] = p_1[0]$. Il trouve $p_1 = 400F$. La requête est transmise à p_1 qui cherche dans sa table de voisinage L_2 un pair p_2 qui vérifie $k[1] = p_2[1]$. Il trouve $p_2 = 4227$. La requête est transmise à p_2 qui cherche dans sa table de voisinage L_3 un pair p_3 qui vérifie $k[2] = p_3[2]$. Il trouve $p_3 = 42A2$. La requête est transmise à p_3 qui cherche dans sa table de voisinage L_4 un pair p_4 qui vérifie $k[3] = p_4[3]$ et n'en trouve pas. Lorsque Tapestry ne trouve pas de pair qui vérifie exactement l'égalité des digits, il cherche le pair qui possède le digit le plus proche et trouve $p_4 = 42AD$. Dans cet exemple, le pair $42AD$ devient la racine de la clé k. Plutôt que de stocker la donnée, les pairs racines dans Tapestry stockent un pointeur (*location mapping*) vers la source et un mécanisme d'indirection est ajouté pour accélérer les recherches : une fois que la clé a été publiée, une requête de notification emprunte le chemin inverse au routage initial et crée sur chaque pair un pointeur vers le pair source pour cette clé. Dans l'exemple de la Fig. 1.12(b), un pointeur inverse a été créé pour chaque pair traversé lors de la publication de la clé (les liens pointillés). Maintenant, si le pair $p = 4629$ cherche la clé k, il commence le routage par 4227 qui possède un pointeur vers la source. p peut donc directement récupérer la donnée associée à la clé sans passer par les pairs $42A2$ puis $42AD$. Une même donnée peut être insérée à

partir de plusieurs sources distinctes. Dans un tel cas de figure, s'il existe une portion de chemin commune entre les sources et la cible, les pairs de ce chemin possèdent un pointeur d'indirection pour chaque source.

Lorsqu'un pair p souhaite rejoindre le réseau, il choisit un identifiant key_p au hasard et trouve le pair s qui est le responsable actuel de key_p dans le réseau. s déduit n, le plus grand préfixe commun entre key_s et key_p et envoie un message qui traverse le réseau à direction de tous les pairs qui partagent le préfixe n. Lorsque ces pairs reçoivent un tel message, ils ajoutent p dans leurs tables de voisinage. p est ensuite contacté par chacun des pairs qui le connaissent et il les ajoute dans le $|n+1|^{\text{e}}$ niveau de sa table de voisinage. p commence ensuite la construction des autres niveaux de sa table de voisinage et demande à chaque pair du niveau $|n+1|$ de lui envoyer ses prédécesseurs (*backpointers*[7]). Cet ensemble de prédécesseurs sert à remplir le niveau $|n|$ de la table des voisinages de p. Ce procédé est répété jusqu'à ce que la table soit entièrement construite. Lorsqu'un pair p se déconnecte il contacte un ensemble de ses prédécesseurs en leur fournissant, pour chaque niveau de leur table de voisinage, un pair de remplacement pris dans sa propre table de voisinage. Pour finir, les prédécesseurs notifiés publient de nouveau les clés qui passaient par p. De même, p publie à nouveau les clés qu'il hébergeait avant de se déconnecter.

Les voisinages dans Tapestry sont de taille fixe égale à $\beta \log_\beta N$, avec N la taille de l'espace d'adressage. En effet, les tables de voisinage ont $b = \log_\beta N$ niveaux et β voisins par niveau. La recherche s'effectue alors au maximum en $O(\log_\beta N)$ sauts.

1.5.4 Kademlia

Kademlia [Maymounkov et Mazières, 2002] est une table de hashage distribuée utilisée dans les dernières versions d'eMule (cf. Sec. 1.3.1). L'espace d'adressage est défini sur les nombres entiers de 160 bits et muni de la métrique XOR pour calculer des distances entre les identifiants. Les identifiants et les clés sont générés à l'aide de la fonction de hachage SHA-1. Chaque pair dans Kademlia se représente l'intégralité de l'espace d'adressage sous la forme d'un arbre binaire dans lequel les pairs occupent la place de feuilles. Chaque pair p va diviser l'arbre en une série de sous-arbres ne le contenant pas. Ces subdivisions de l'arbre sont appelées des *buckets*. La Fig. 1.13 montre un exemple de représentation en arbre qu'un pair d'identifiant 0011 se fait du réseau. Les zones en pointillés représentent les *buckets*. Ces zones sont les tables de voisinage d'un pair qui maintient pour chaque bucket un ensemble de pairs connectés dans cette zone. Les buckets ont une taille fixe k soumise à la politique de cache LRU et sont mises à jour de manière transparente à chaque fois qu'un pair reçoit un message d'un autre pair. Les voisins contenus dans la n^{e} bucket d'un pair p ont un préfixe commun de taille n avec p. Plus précisément, un pair p d'identifiant $key(p)$ maintient un ensemble de b_i k-bucket avec $0 \leq i < 160$. La i^{e} k-bucket contient k pairs $p_j \neq p$ tels que $key(p) \oplus key(p_j) \in [2^i, 2^{i+1}[$. Dans l'exemple de la Fig. 1.13, les k-buckets de droite à gauche représentent respectivement les intervalles $[2^1,2^2[$, $[2^2,2^3[$, $[2^3,2^4[$ et $[2^4,2^5[$.

7. Dans Tapestry, chaque pair maintient la liste des pairs qui les possèdent dans leurs tables de voisinages.

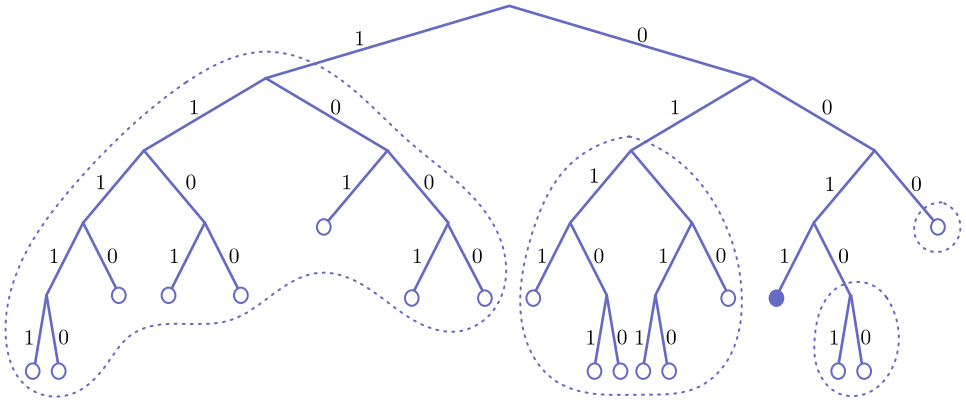


FIGURE 1.13 – Organisation de Kademlia.

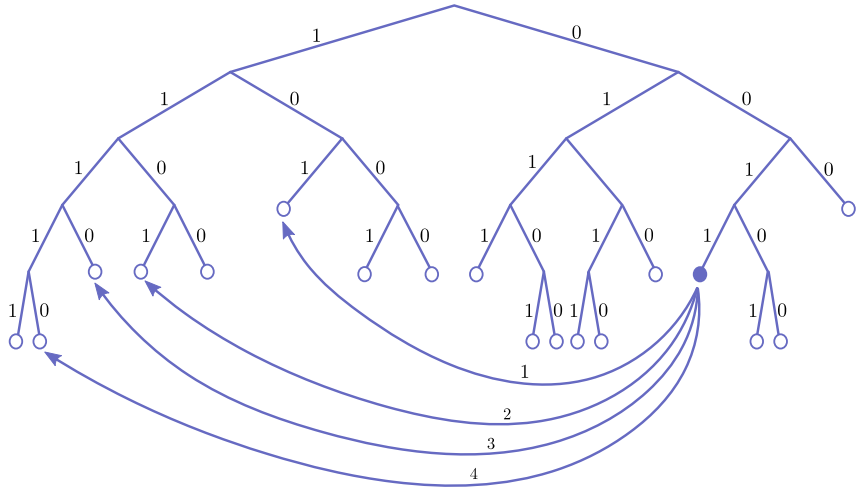


FIGURE 1.14 – Recherche d'un pair dans Kademlia. Le pair d'identifiant 0011 cherche le pair d'identifiant 11110.

Dans cette structure, la recherche d'une clé α à partir d'un pair p se fait par routage. Le pair p cherche la k-bucket b_i dans laquelle α se trouve. p envoie ensuite la requête au pair de b_i le plus proche de α qui effectue la même opération et retourne à p, son plus proche voisin. Cette procédure est répétée jusqu'à trouver le pair responsable de α. La Fig. 1.14 montre un exemple de recherche du pair 11110 à partir du pair 0011. Le pair 0011 situe 11110 dans la k-bucket $[2^4,2^5[$ puisque $00011 \oplus 11110 = 29$. Le pair 0011 commence la recherche en interrogeant un de ses voisins dans cette k-bucket, en l'occurrence le pair 101. 101 cherche alors son voisin le plus proche de la cible et le retourne à 0011. Le pair 0011 met alors à jour sa bucket avec 1101 et reprend la recherche à partir de 1101 et ainsi de suite jusqu'à trouver le pair cible. La publication d'une clé se fait sur le même mécanisme à la différence qu'elle est publiée sur les γ pairs les plus proches de α, avec γ un paramètre du système.

Lorsqu'un pair p rejoint le réseau, il ajoute un pair passerelle p' dans une de ses buckets. Ensuite, il effectue plusieurs recherches de clés ayant pour rôle de remplir ses buckets. Le mécanisme

de transfert de clés, suite au départ ou à l'arrivée d'un pair, est géré de manière transparente par des republications périodiques de clés par l'ensemble de pairs.

La complexité de la recherche (après une optimisation qui consiste à changer de base b) est de l'ordre de $O(\log_{2^b} N)$ avec N, le nombre total de pairs dans le réseau et b, la base sur laquelle est codé l'espace d'adressage.

1.5.5 Freenet >v0.7

Nous revenons brièvement sur Freenet dans sa version actuelle [Clarke *et al.*, 2010] (la description de Freenet <v0.7 est accessible à la Sec. 1.4.2) puisque son organisation passe de non-structurée à structurée. Dans cette nouvelle mouture de Freenet, les pairs se connectent uniquement à des pairs de confiance formant ainsi un réseau *Darknet* qui peut être apparenté à un sous-réseau du réseau social mondial [Sandberg, 2006, Evans *et al.*, 2007]. L'hypothèse émise dans Freenet est que ce sous-réseau est un petit monde [Milgram, 1967, Watts et Strogatz, 1998] dont la propriété des six degrés de séparation peut être exploitée pour obtenir un routage efficace.

Lorsqu'un pair p se connecte, il choisit un identifiant aléatoirement, comme pour Freenet <v0.7, et se connecte à un certain nombre de pairs de confiance à l'aide d'un protocole sécurisé. Dans le cas où p ne connaît aucun pair de confiance, il a toujours la possibilité de se connecter sur la partie publique de Freenet comme dans la version initiale. Freenet est donc composée d'un *Darknet* et d'un *Opennet* et les pairs ont le choix de fonctionner en mode privé pur pour plus de sécurité ou en mode hybride. Une procédure de recuit simulé [Sandberg, 2006] est exécutée en arrière-plan et échange périodiquement des identifiants de sorte que des pairs proches dans la topologie aient également un identifiant proche. Le routage étant le même que dans la version initiale de Freenet, c'est-à-dire glouton, l'idéal serait d'avoir une structure d'identifiants qui permette de se rapprocher un peu plus à chaque saut de la cible. En pratique, une telle propriété n'est pas toujours obtenue, c'est pourquoi un TTL est affecté aux messages de recherche.

1.6 Les réseaux aléatoires

Les topologies des architectures décentralisées non-structurées que nous avons vu à la Sec. 1.4 présentent des caractères aléatoires qui ne sont pas bien maitrisés. À l'inverse, il existe d'autres topologies aléatoires qui sont issues de modèles de graphes aléatoires étudiés en théorie des graphes et qui possèdent des propriétés intéressantes pour répondre à certaines problématiques de robustesse. Ces graphes sont notamment bien adaptés au support de la diffusion épidémique de l'information qui est, elle-même, proposée comme une alternative à la diffusion multicast fiable. Nous présentons dans cette section les enjeux de tels graphes et nous détaillons un protocole (SCAMP) qui est utilisé dans nos travaux.

1.6.1 Théorie des graphes aléatoires : le modèle de Erdös-Rényi

Les recherches effectuées en théorie des graphes aléatoires s'intéressent aux structures probables qu'un graphe aléatoire peut présenter en fonction de certains paramètres. Nous nous intéressons ici au modèle de Paul Erdös et de Alfréd Rényi [Erdős et Rényi, 1959]. Un graphe peut-être défini de deux manières dans ce modèle.

Définition 1 *$\Gamma_{n,N}$ est un graphe aléatoire non orienté ayant n sommets étiquetés P_1,P_2,...,P_n et N arêtes. Le choix des N arêtes est aléatoire parmi les $\binom{n}{2}$ arêtes possibles tel que les $\binom{\binom{n}{2}}{N} = C_{n,N}$ choix possibles sont supposés équiprobables. De fait, si $G_{n,N}$ est l'un des graphes $C_{n,N}$, la probabilité que le graphe $\Gamma_{n,N}$ soit identique à $G_{n,N}$ est égale à $\frac{1}{C_{n,N}}$.*

Définition 2 *On trouve également dans la littérature une autre définition. On définit un graphe aléatoire comme un graphe de n sommets tel qu'il existe une arête entre toute paire de sommets avec une certaine probabilité p indépendamment des autres arêtes. Le nombre d'arêtes est alors une variable aléatoire dont l'espérance est égale à $\binom{n}{2}p$. Si l'on souhaite obtenir un graphe de N arêtes en moyenne, il faut choisir $p = \frac{N}{\binom{n}{2}}$. On note un tel graphe $\Gamma^{*\ *}_{n,N}$. Les résultats que nous présentons par la suite sont les mêmes pour les graphes aléatoires $\Gamma_{n,N}$ et $\Gamma^{*\ *}_{n,N}$.*

Définition 3 *Soit A une propriété qu'un graphe peut ou ne peut pas posséder, on note $\mathbb{P}_{n,N}(A)$ la probabilité que le graphe aléatoire $\Gamma_{n,N}$ possède la propriété A. On a*

$$\mathbb{P}_{n,N}(A) = \frac{A_{n,N}}{C_{n,N}}$$

avec $A_{n,N}$ le nombre de $G_{n,N}$ possédant la propriété A.

Définition 4 *Soit la propriété $\mathbb{P}_0(n,N)$: le graphe aléatoire de n nœuds et N arêtes est connexe.*

Alors, [Erdős et Rényi, 1960] énoncent le théorème qui suit.

Théorème 5 *Soit c un nombre réel fixé, et*

$$N_c = \left[\frac{1}{2}\, n\, ln(n) + c\, n\right] \tag{1.1}$$

alors, si $\mathbb{P}_0(n,N_c)$ est la probabilité que $\Gamma_{n,N}$ soit connexe. On a :

$$\lim_{n\to+\infty} \mathbb{P}_0(n,N_c) = \exp^{-\exp^{-c}} \tag{1.2}$$

soit

$$d = \frac{2N}{n} = \ln(n) + 2c \tag{1.3}$$

avec d, le degré moyen du graphe aléatoire.

Seuil de connexité

$\frac{1}{2} n\,ln(n)$ est appelé seuil de connexité. Ceci signifie qu'avec une forte probabilité, tous les graphes aléatoires ayant un nombre d'arêtes $N > N_c$ sont connexes. Réciproquement, tous les graphes aléatoires ayant un nombre d'arêtes $N < N_c$ possèdent au moins 2 composantes connexes. Ce résultat est particulièrement intéressant pour la construction de réseaux connexes et tolérants aux fautes.

1.6.2 Diffusion épidémique de l'information

La diffusion épidémique (i.e., *gossip protocols*) est une approche probabiliste qui consiste à diffuser d'une certaine manière une information de proche en proche dans une population de sorte que tout individu de cette population ait reçu l'information au bout d'un certain temps. Dans ce type de diffusion, on considère que les individus ont leur libre arbitre et peuvent décider de ne pas propager l'information à certaines de leurs connaissances. L'intérêt d'une telle approche dans des protocoles de diffusion réseau est double :

1. elle passe à l'échelle puisque les participants n'ont pas besoin de connaître tout le système ;
2. elle tolère un certain degré de fautes puisque des participants au protocole peuvent décider de ne pas propager la rumeur.

[Kermarrec *et al.*, 2003] donnent une définition plus formelle de la structure de la diffusion épidémique dans une population :

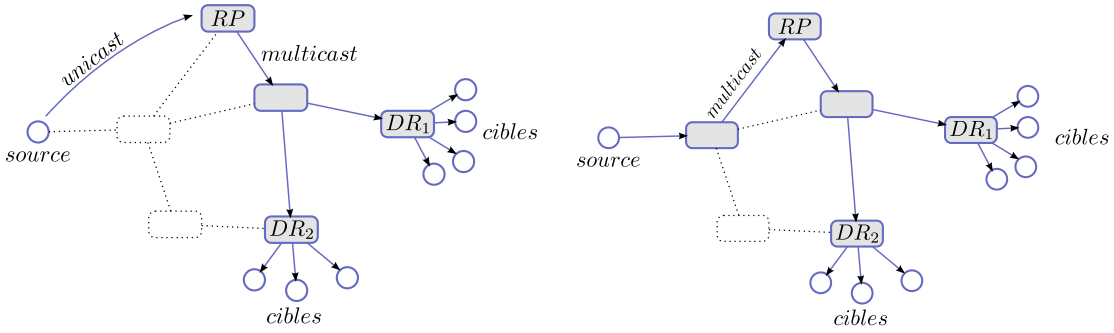
Définition 6 *Soit le système constitué de n nœuds. Chaque nœud n_i possède une vue aléatoire et partielle $d_i \subseteq n$ de l'ensemble du système.*

1. *lorsqu'un nœud n_i veut propager une rumeur à l'ensemble du système, il initie un nouvel identifiant de rumeur et envoie cette information à $k \subseteq d_i$ éléments tirés aléatoirement ;*
2. *lorsqu'un nœud n_i reçoit une rumeur, s'il ne l'a pas déjà traitée, il la traite et la propage à $k \subseteq d_i$ éléments tirés aléatoirement.*

La succession des k propagations par les n nœuds construit un réseau logique que l'on peut modéliser par un graphe orienté. À la fin de la diffusion, la rumeur aura atteint l'ensemble des participants si et seulement si le graphe résultant de ces interactions est connexe. On remarque que le protocole de diffusion comme il vient d'être défini construit un graphe aléatoire suivant le modèle Erdös-Rényi. Nous avons vu à la section précédente qu'il existe un seuil de connexité dans ces graphes qui est conditionné par le degré moyen des nœuds. Dans le contexte de la diffusion épidémique, c'est donc la taille des vues partielles des participants qui conditionnera l'issue de la diffusion.

1.6.3 Application à la diffusion fiable en multicast

Cette propriété de connexité est exploitée notamment dans le domaine de la diffusion multicast tolérante aux fautes. En effet, avec l'arrivée d'IPv6 et du multicast en natif, de nouvelles



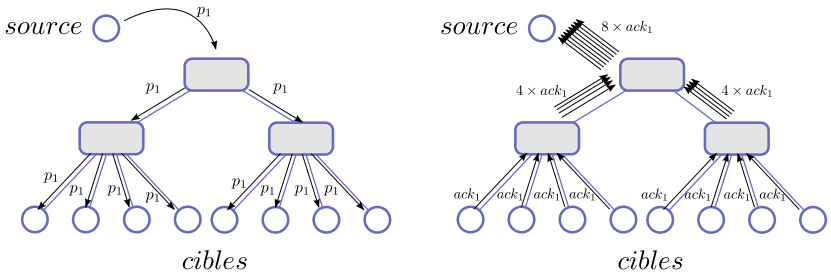
(a) La *source* encapsule ses paquets vers la racine *RP* de l'arbre multicast qui les diffuse ensuite dans l'arbre PIM à destination des *cibles*.

(b) Une fois que la source est identifiée, un chemin multicast de *source* à *RP* peut être créé pour supprimer le coût de encapsulations/décapsulations.

FIGURE 1.15 – Diffusion multicast dans un arbre PIM.

questions ont été soulevées et notamment celle de la diffusion fiable de l'information dans un groupe multicast. Cette diffusion fiable consiste à tolérer les pertes de paquets et de nœuds dans l'arbre multicast pendant la diffusion. Dans une diffusion multicast classique [Cizault, 2002], chaque paquet multicast à destination d'un groupe est émis une seule fois par l'émetteur. La responsabilité de propagation du paquet au sein du groupe est alors répartie sur l'ensemble des routeurs multicasts se trouvant entre l'émetteur et chacun des récepteurs. La construction d'un groupe multicast dans le standard de l'IETF requiert l'utilisation de deux mécanismes. Le premier consiste à recenser les participants des différents groupes multicast aux extrémités du réseau. Ce rôle est joué par les routeurs de bordure qui implémentent le protocole MLD (Multicast Listener Discovery) [Deering *et al.*, 1999] pour IPv6 et IGMP [Holbrook *et al.*, 2006] pour IPv4. Le second, crée un arbre multicast partagé PIM (Protocol Independent Multicast) [Fenner *et al.*, 2006] dont l'objectif est de créer une structure de données hiérarchique supportant la diffusion efficace entre les différents membres d'un groupe. Dans un arbre PIM, les récepteurs du groupe sont des feuilles et leurs noeuds parents sont des routeurs multicasts relais appelés DR (*designated routers*). Un DR est un routeur situé dans le même domaine qu'un récepteur. La racine de l'arbre est appelée RP (rendez-vous point). La diffusion par la (ou les) source(s) est alors faite en encapsulant le paquet multicast dans un paquet unicast à destination du RP. Sur réception de ce paquet, le RP décapsule le paquet multicast et le diffuse à ses fils dans l'arbre PIM correspondant au bon groupe. De fils en fils, le message est diffusé à tout le groupe (cf. Fig. 1.15(a)). Cette figure illustre bien le fait que le message multicast n'est envoyé qu'une seule fois par le RP au premier routeur alors que dans une approche unicast il aurait été dupliqué 7 fois (un pour chaque destinataire). Cependant les opérations d'encapsulation/décapsulation sont coûteuses pour le RP. C'est pourquoi, après les premières diffusions vers les RP, un chemin similaire à l'arbre PIM peut être créé de la source au RP permettant à la source de propager directement le paquet multicast au RP sans encapsulation (cf. Fig. 1.15(b)). Le RP devient alors un simple routeur multicast relais.

Le multicast permet de réduire le coût de la diffusion en mutualisant le transfert des paquets



(a) La *source* émet le paquet p_1 à destination du groupe.

(b) L'ensemble des cibles acquittent par ack_1 le paquet p_1.

FIGURE 1.16 – Illustration du phénomène des tempêtes d'acquittements.

à destination de participants qui partagent le même sous-arbre. Dans une approche unicast, le même paquet est envoyé une fois à destination de chaque participant même si ceux-ci partagent le même sous-arbre. Le protocole PIM définit une structure pour le routage mais n'assure pas que les paquets seront bien reçus ni qu'ils seront reçus dans le bon ordre. La structure arborescente accentue le problème puisque la perte d'une liaison déconnectera tout un sous-arbre. Dans les approches unicast, la robustesse est obtenue au niveau réseau en utilisant des mécanismes d'acquittement comme c'est le cas dans le protocole TCP. L'application directe de ces techniques à acquittement dans le contexte de la diffusion multicast pose le problème des tempêtes d'acquittements. Si chaque destinataire envoie des acquittements, le gain du multicast est grandement réduit. Ce phénomène est illustré à la Fig. 1.16 dans laquelle la source émet un paquet p_1 à destination du groupe (cf. Fig. 1.16(a)) qui est ensuite acquitté par toutes les cibles (cf. Fig. 1.16(b)). Plusieurs travaux proposent des algorithmes visant à réduire le nombre d'acquittements émis. Nous décidons de présenter deux protocoles fiabilisant le multicast avec des techniques d'acquittements :

1. [Paul *et al.*, 1997] proposent dans RMTP (Reliable Multicast Transport Protocol) de désigner des routeurs faisant office de proxy/cache pour un ensemble de cibles dans leur sous-arbre. Un exemple illustratif est présenté à la Fig. 1.17. Dans cette approche, chaque routeur désigné (DR) possède un cache des paquets multicasts qu'il a reçu. Les cibles acquittent les paquets qu'elles ont reçu à leur DR. Un DR peut donc retransmettre les paquets n'ayant pas été acquittés ou qui sont explicitement demandés. Lorsqu'une telle demande de retransmission est reçue par un DR, il place les paquets à retransmettre dans une file d'attente et arme un timer T_{retx}. Ce mécanisme permet d'attendre que d'autres cibles du sous-arbre n'ayant pas reçu, elles non plus, les paquets en file se manifestent. Lorsque T_{retx} expire, si le nombre de cibles ayant demandé la retransmission des paquets en file dépasse un certain seuil, alors ces paquets sont diffusés en multicast. Sinon, ils sont diffusés en unicast à chacun des destinataires. Un routeur désigné, au même titre que les cibles, acquitte ses paquets à son DR. Dans l'exemple de la Fig. 1.17, dans le cas où un paquet est envoyé et n'est pas perdu, seuls 2 acquittements remontent à la *source* au lieu de 14 pour une approche triviale. Avec ce mécanisme de routeurs désignés, RMTP fiabilise

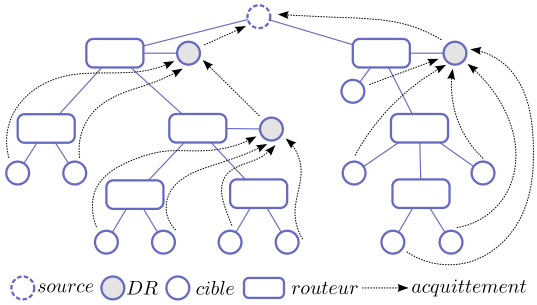
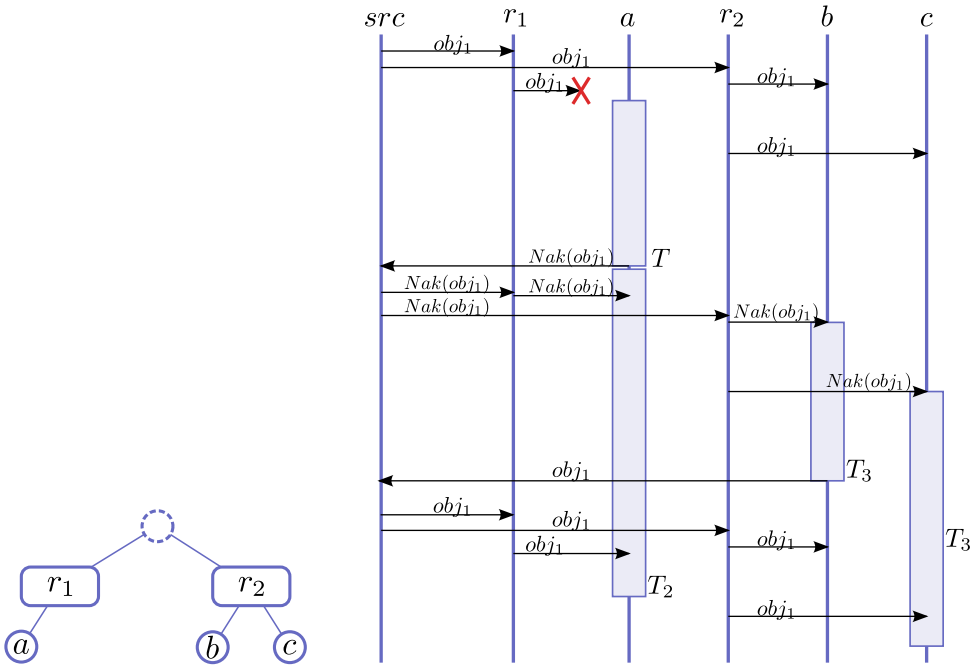


FIGURE 1.17 – Remontées d'acquittements vers des routeurs dédiés dans RMTP.

la diffusion multicast tout en limitant l'explosion du nombre d'acquittements ;

2. [Floyd *et al.*, 1995] proposent dans SRM (Scalable Reliable Multicast) de séparer la donnée diffusée dans le groupe en pages. Chaque page contient plusieurs objets. Dans SRM, un véritable schéma de nommage est utilisé pour identifier les objets dans les différentes pages. Contrairement à RMTP qui centralise la gestion des acquittements, SRM la décentralise et la probabilise. Ce sont les hôtes qui sont responsables de la gestion des erreurs. Toute demande de réparation ou toute réparation est diffusée à tout le groupe. Pour éviter les tempêtes d'acquittements, des temporisations pour chaque demande de réparation sont armées. Concrètement, lorsqu'un hôte A détecte une perte (par une rupture de séquence par exemple), il planifie, à l'aide d'un timer T, l'envoi d'un paquet de demande de réparation à un instant futur choisi aléatoirement. Quand T expire, A diffuse cette requête en multicast à tout le groupe et arme une temporisation $T_2 = 2T$ pour l'attente de la réparation. Si A reçoit la même requête de réparation (un autre hôte n'a pas reçu le même objet) avant l'expiration de T, alors il étend la date d'expiration de T. Lorsqu'un hôte B reçoit une demande de réparation et qu'il est en possession du paquet à réparer, il arme une temporisation T_3 pour la réémission de cet objet. Si B reçoit le paquet de réparation avant l'expiration de T_3, il annule la procédure. Sinon, il diffuse la réparation en multicast à tout le groupe quand T_3 expire. Ce principe est expliqué sur l'exemple de la Fig. 1.18. Contrairement à RMTP, tout hôte du groupe peut réémettre des paquets perdus et le contrôle du nombre d'acquittements est géré par des temporisations précises.

Les deux protocoles que nous avons présentés, et de manière plus générale la famille de protocoles les englobant, se trouvent au niveau réseau (au sens du modèle OSI). Hors, face à la lenteur du déploiement du multicast IP et à la demande croissante en terme de diffusion de contenu sur le web, la recherche s'est tournée vers la diffusion multicast de niveau application [Chawathe *et al.*, 2000, Zhuang *et al.*, 2001, Rowstron *et al.*, 2001, Hosseini *et al.*, 2007]. Dans une approche située au niveau application, le relais de l'information à diffuser n'est plus à la charge des routeurs intermédiaires mais elle passe à la charge des membres du groupe et met ces membres au même niveau, nous plaçant clairement dans un cadre pair-à-pair. Une des solutions proposées [Ganesh *et al.*, 2001, Ganesh *et al.*, 2002, Ganesh *et al.*, 2003, Kermarrec *et al.*, 2003] sur laquelle nous allons mettre l'emphase repose sur la diffusion épidémique de l'in-



(a) Structure de l'arbre multicast.

(b) Échanges de messages suite à la perte du paquet obj_1 entre r_1 et a.

FIGURE 1.18 – Exemple de fonctionnement simplifié du protocole SRM.

formation. Le protocole SCAMP, que nous décrivons en détail dans la section suivante, construit un réseau logique aléatoire de type Erdös-Rényi ayant un degré moyen au-dessus du seuil de connexité. Nous avons vu que lorsqu'une telle structure est effective, la probabilité que l'ensemble des membres du groupe reçoivent l'information tend vers 1. La tolérance aux fautes en terme de pertes de liens et de pairs est obtenue en créant de la redondance d'arcs au-dessus du seuil de connexité (la k-connexité) de sorte que le graphe supporte un certain nombre de fautes avant de se déconnecter.

1.6.4 SCAMP : Scalable Membership Protocol

Le protocole SCAMP [Ganesh *et al.*, 2001] est une procédure totalement décentralisée qui construit un réseau logique, modélisé sous la forme d'un graphe aléatoire orienté, suivant le modèle de Erdös-Rény et ayant un degré sortant moyen convergeant vers $(c+1)\log(n)$; où c est une constante de tolérance aux fautes et n le nombre de nœuds du graphe. Cette constante permet de moduler le nombre d'arcs du graphe au dessus du seuil de connexité (cf. eq.(1.3)). [Kermarrec *et al.*, 2003] montrent à cette occasion que le seuil de connexité d'un graphe aléatoire orienté est le même que celui d'un graphe aléatoire non-orienté.

Définition 7 *On note (n,c)-SCAMP un réseau aléatoire construit avec le protocole SCAMP et ayant cette structure.*

Propriété 8 *En présence de fautes, un (n,c)-SCAMP reste connexe si la proportion d'arcs perdus est inférieure à* $\frac{c}{c+1}$.

La structure de SCAMP est donc tout à fait propice à la diffusion multicast fiable et possède les propriétés suivantes :

- Passage à l'échelle et décentralisation : dans la définition 6 sur la diffusion épidémique (cf. Sec. 1.6.2), les participants ont une vue partielle d_i de la population n qui est tirée aléatoirement et uniformément. Cette contrainte pose un problème de passage à l'échelle dans les réseaux pair-à-pair puisqu'elle nécessite que tous les pairs se connaissent ou aient recours à une entité centrale pouvant les mettre en relation. Or, construire un groupe multicast large échelle tolérant aux fautes n'est pas possible s'il dépend d'une entité centrale. SCAMP propose une construction du graphe de manière totalement décentralisée. Dans cette construction, le réseau croît petit à petit en s'assurant que la propriété de degré moyen soit respectée. Ce protocole peut être déployé à large échelle étant donné que son degré croît logarithmiquement ;
- Robustesse : ce réseau logique est tolérant aux fautes puisque son degré moyen est au-dessus du seuil de connexité. Cette tolérance est quantifiable au moyen de la constante de tolérance aux fautes c. Il est donc possible de choisir les propriétés du réseau en fonction du taux de pannes attendu ;
- Légèreté : SCAMP met en place un ensemble de mécanismes qui sont simples et peu couteux en comparaison de techniques reposant sur des acquittements.

Définition 9 *Soit un (n,c)-SCAMP que l'on modélise sous la forme d'un graphe aléatoire orienté* $G = (V,A)$. *Pour tout sommet* $s \in V$, *on note* $PartialView_s$, PV_s *ou encore* $SUCC_s$, *l'ensemble des successeurs de* s.

Définition 10 *Soit un (n,c)-SCAMP que l'on modélise sous la forme d'un graphe aléatoire orienté* $G = (V,A)$. *Pour tout somment* $s \in V$, *on note* $InView_s$, IV_s *ou encore* $PRED_s$ *l'ensemble des prédécesseurs de* s.

Nous décrivons maintenant dans le détail le protocole SCAMP. Cela nous paraît nécessaire dans la mesure où nous en faisons un usage intensif dans nos contributions.

Construction du réseau

Lorsqu'un pair s rejoint un réseau SCAMP, il envoie un message `subscribe` à un pair b (appelé nœud de contact) qui est déjà connecté à ce réseau. b joue le rôle de passerelle pour s. Nous faisons l'hypothèse, pour le moment, que b est choisi par s aléatoirement parmi tous les membres déjà présents dans le réseau (nous décrivons plus loin dans cette section l'algorithme permettant de trouver une telle passerelle). s décide de se connecter :

1. s ajoute b à sa $PartialView$;
2. s envoie un message `subscribe` à b ;

3. Lorsqu'un pair reçoit un message `subscribe`, il exécute l'algorithme 1 qui dissémine la demande de connexion dans le réseau. Les messages `forward` ont pour but de créer le nombre d'arcs nécessaires pour que la propriété de degré de SCAMP soit vérifiée. Les messages `forward` ne sont donc jamais perdus et sont propagés jusqu'à atteindre leur place dans le réseau. L'algorithme 1 diffuse deux vagues de messages `forward` :
   (a) Une première diffusion est faite à tous les voisins permettant d'atteindre une convergence moyenne du degré vers $\log(n)$;
   (b) Une seconde diffusion est faite à c voisins pris aléatoirement parmi PV_b pour atteindre la convergence de $(c+1)\log(n)$.
4. Sur réception d'un message `forward` par un pair x du graphe, x exécute l'algorithme 2 qui ajoute s à PV_x avec une probabilité $p = \frac{1}{|PV_x|+1}$, ou propage le message `forward` à un de ses successeurs sinon.

Algorithme 1 : Algorithme de gestion des souscriptions

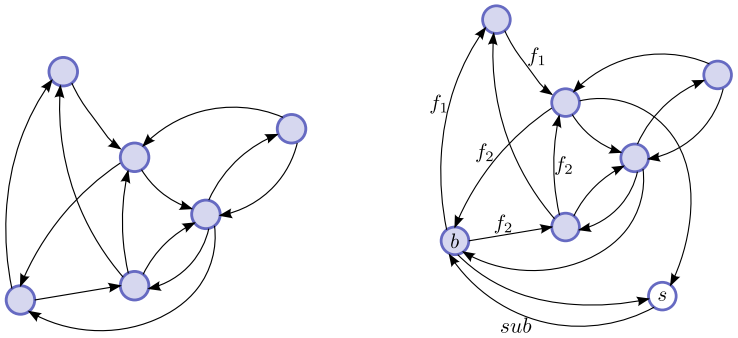
Entrées : s le nœud souhaitant rejoindre le réseau, b le pair exécutant l'algorithme, c la constante de tolérance aux fautes

// Propager la requête à tout voisin;
pour $v \in PV_b$ **faire**
    EnvoyerMessage(v, `forward(s)`);
// Envoyer c Forward supplémentaires à des voisins choisis aléatoirement;
pour $(j = 0;\ j < c;\ j++)$ **faire**
    $v_{aleat} \leftarrow ChoixPairAleatoire(PV_b)$;
    EnvoyerMessage(v_{aleat}, `forward(s)`);

Algorithme 2 : Algorithme de gestion de Forward

Entrées : x le pair exécutant l'algorithme, `forward(s)` le message reçu

$tirage \leftarrow FloatAleatoire \in [0,1]$;
$p \leftarrow \frac{1}{|PV_x|+1}$;
si $s \notin PV_x$ **alors**
    // ajouter le pair s aux successeurs de x avec la probabilité p;
    si $tirage \leq p$ **alors**
        $PV_x \leftarrow PV_x \cup \{s\}$;
        return;
// Les requêtes forward ne sont jamais perdues;
$v_{aleat} \leftarrow ChoixPairAleatoire(PV_x)$;
EnvoyerMessage(v_{aleat}, `forward(s)`);



(a) Exemple de réseau SCAMP.

(b) Le pair s rejoint le réseau par un nœud de contact b.

FIGURE 1.19 – Réseau SCAMP avant et après la connexion du pair s.

La Fig. 1.19 illustre l'ajout d'un pair s à un réseau SCAMP. La structure du réseau avant l'ajout est présentée à la Fig. 1.19(a). Dans la procédure de connexion illustrée à la Fig. 1.19(b), le pair s choisis le pair b comme nœud de contact et lui envoie un message `subscribe` puis s ajoute b à PV_s. Lorsque b reçoit la souscription de s, il applique l'algorithme 1 et envoie un message `forward` à chacun de ses successeurs. Les deux messages `forward` f_1 et f_2 sont propagés de pair en pair suivant l'algorithme 2 jusqu'à ce que les deux arcs attendus soient créés. Dans cet exemple $c = 0$, c'est pourquoi b n'envoie pas de messages `forward` supplémentaires à ses successeurs.

Décentralisation de la construction par indirection

Nous avons fait l'hypothèse que le pair s choisit son contact aléatoirement parmi l'ensemble des pairs déjà connectés au réseau. Cette hypothèse dans un cadre totalement décentralisé n'est pas réaliste. Pourtant, le nœud de contact doit bel et bien être choisi aléatoirement pour que le réseau soit aléatoire. [Ganesh *et al.*, 2003] proposent un mécanisme d'indirection qui permet à une requête `subscribe` d'effectuer une marche aléatoire à partir du nœud de contact b jusqu'à un pair cible j à partir duquel les messages `forward` seront propagés. La marche aléatoire repose sur une chaîne de Markov distribuée dans laquelle on fait correspondre l'ensemble des pairs aux états et les relations de voisinage aux transitions entre ces états. Les messages `subscribe` vont alors effectuer un nombre de sauts bornés dans cette chaîne. À chaque saut, le message est propagé d'un pair i à un pair j avec une certaine probabilité w_{ij} avec $j \in PV_i$. On conditionne la matrice des transitions comme suit :

$$\sum_{j \in PV_i} w_{ij} = 1 \tag{1.4}$$

et

$$\sum_{i \in IV_j} w_{ij} = 1 \tag{1.5}$$

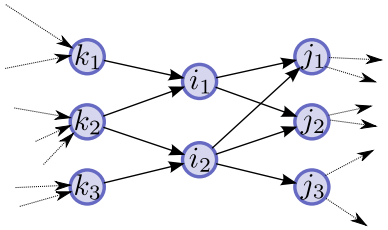


FIGURE 1.20 – Structure de la chaîne de Markov supportant le mécanisme d'indirection.

La matrice W de poids $w_{ij} \geq 0$ satisfaisant (1.4) et (1.5) est irréductible et doublement stochastique et sa distribution stationnaire est la loi uniforme. Cette matrice est répartie sur les pairs qui associent pour chacun de leurs successeurs et prédécesseurs la valeur de la transition correspondante dans W. Dans l'exemple de la Fig. 1.20, le pair i_1 affecte donc respectivement $w_{i_1j_1}$, $w_{i_1j_2}$, $w_{k_1i_1}$ et $w_{k_2i_1}$ aux arcs (i_1j_1), (i_1j_2), (k_1i_1) et (k_2i_1). Lorsqu'un nouveau nœud j arrive dans la PV (respectivement IV) d'un nœud i, i initie la valeur de w_{ij} (respectivement w_{ji}) comme étant égale à la moyenne des poids de cette vue. Deux problèmes se posent : (1) la matrice doit être symétrique et, par conséquent, les poids des deux extrémités doivent être égaux. (2) La perturbation introduite par l'ajout d'un nouveau nœud fait que la matrice W peut violer (1.4) et (1.5). Ce phénomène est visible à la Fig. 1.20 dans laquelle les deux équations suivantes doivent être vérifiées :

$$w_{k_1i_1} + w_{k_2i_1} = 1 \tag{1.6}$$

$$w_{k_2i_1} + w_{k_2i_2} = 1 \tag{1.7}$$

On constate que la modification d'un des poids d'une des équations entraînera une répercussion sur l'autre. Pour solutionner ces deux problèmes, chaque pair va périodiquement normaliser les poids contenus dans ses vues puis propager cette mise à jour à l'autre extrémité de l'arc jusqu'à ce que le système trouve un équilibre. Cette boucle de normalisation périodique est détaillée dans l'algorithme 3. On remarque que, pour mettre en place cette chaîne de Markov, il est

Algorithme 3 : Mise à jour des arcs pour un nœud i

$w_{in} \leftarrow \sum_{j \in IV_i} w_{ji}$;
$w_{out} \leftarrow \sum_{j \in PV_i} w_{ij}$;
pour $j \in IV_i$ **faire**
    $w_{ji} \leftarrow \frac{w_{ji}}{w_{in}}$;
    EnvoyerMessage(j, `updateweight`(w_{ji}));
pour $j \in PV_i$ **faire**
    $w_{ij} \leftarrow \frac{w_{ij}}{w_{out}}$;
    EnvoyerMessage(j, `updateweight`(w_{ij}));

nécessaire que les pairs aient également l'information de leur IV. Pour ce faire, lorsqu'un pair i ajoute un pair j dans PV_j, il envoie un message de notification à j. Sur réception de cette notification, le pair j ajoute i à IV_j. Lorsqu'un pair reçoit un message `updateweight`(w) il

Algorithme 4 : Sur réception de `updateweight`(w) par un nœud i

Entrées : j le pair d'où provient les message `updateweight`, w le nouveau poids

si $j \in PV_i$ **alors**
  $w_{ij} \leftarrow w$;

sinon
  $w_{ji} \leftarrow w$;

exécute l'algorithme 4 qui met à jour le poids de l'arc. [Ganesh *et al.*, 2003] montrent qu'après un nombre suffisant d'itérations de l'algorithme 3, la matrice W approxime une matrice W^0 qui vérifierait (1.4) et (1.5).

Indirection des messages `subscribe`

Il est maintenant possible d'effectuer la marche aléatoire. Lorsqu'un pair b reçoit un message `subscribe` d'un pair s, il envoie un message `subscription`(s, $2|PV_b|$) à un de ses successeurs choisi en fonction de la matrice W (cf. algorithme 5). Ce message contient l'adresse de s et un compteur qui représente le nombre de sauts de la marche aléatoire. Ce compteur est décrémenté à chaque fois que le message `subscription` est propagé dans la chaîne. Lorsqu'un pair i reçoit un message `subscription`(s, $compt_s$) il exécute l'algorithme 6 qui choisit le pair vers lequel propager la requête. Lorsque $compt_s = 0$, la marche s'achève et le pair qui exécute l'algorithme devient le nouveau nœud de contact et exécute l'algorithme 1.

Algorithme 5 : Initiation de la marche aléatoire par un nœud b pour un nœud s

$compt_s \leftarrow 2|PV_b|$;
$w_{out} \leftarrow \sum_{j \in PV_b} w_{bj}$;
pour $j \in PV_b$ **faire**
  $w_{bj} \leftarrow \frac{w_{bj}}{w_{out}}$;
Choisir $j \in PV_b$ avec la probabilité w_{bj};
EnvoyerMessage(j, `subscription`(s, $compt_s$));

Maintien du réseau

À la suite de fautes, un nœud i peut se retrouver isolé. C'est-à-dire que tous les éléments de IV_i sont déconnectés. En conséquence, i ne recevra jamais de messages. La détection de l'isolation est faite au moyen d'un mécanisme de battement de cœur. Périodiquement, tout nœud va envoyer un message `hello` à ses successeurs. Lorsqu'un nœud ne reçoit plus les messages

Algorithme 6 : Sur réception de `subscription`(s, $compt_s$) par un pair i

$w_{out} \leftarrow \sum_{j \in PV_i} w_{ij}$;
si $compt_s \neq 0$ **alors**
  pour $j \in PV_i$ **faire**
    $w_{ij} \leftarrow \frac{w_{ij}}{w_{out}}$;
  Choisir $j \in PV_i$ avec la probabilité w_{ij};
  $compt_s \leftarrow compt_s - 1$;
  EnvoyerMessage(j, `subscription`(s,$compt_s$));
sinon
  // La marche aléatoire a atteint son but;
  i devient le nœud de contact et applique le protocole SCAMP (cf. algorithme 1);

`hello` et s'aperçoit qu'il est isolé, il se déconnecte de sa PV et réinitialise une souscription au graphe SCAMP (cf. algorithme 5) depuis une porte d'entrée bien connue ou d'un élément de son ancienne PV.

Départ de pairs

Lorsqu'un pair i souhaite se déconnecter, il exécute l'algorithme 7 qui choisit $|IV_i| - c - 1$ pairs dans IV_i à reconnecter avec autant d'éléments de PV_i. Lorsqu'un pair j reçoit un message `inform`(e) il ajoute e à PV_j. Le déroulement de l'algorithme est illustré à la Fig. 1.21. Cet

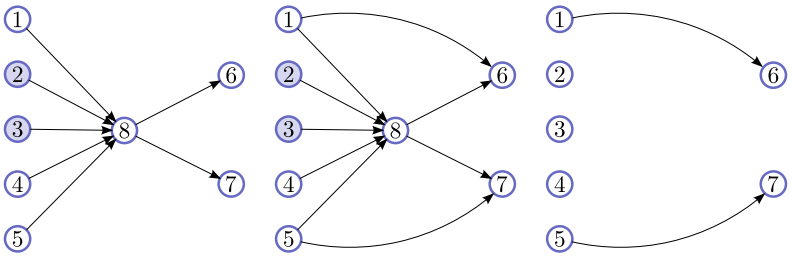
Algorithme 7 : Déconnexion d'un pair i

si $|IV_i| \leq c + 1$ **alors**
  return ;
$t \leftarrow \min(|IV_i| - c - 1, |PV_i|)$;
$Q_1 \leftarrow$ Choix de t éléments dans IV_i;
$Q_2 \leftarrow$ Choix de t éléments dans PV_i;
pour $e \in Q_1$ **faire**
  $j \leftarrow$ Choix dans Q_2;
  $Q_2 \leftarrow Q_2 - j$;
  EnvoyerMessage(e, `inform`(j));

algorithme garantit que la déconnexion de pairs préserve la propriété de degré du réseau. En effet, si on se place dans un (n,c)-SCAMP avec M_n le nombre d'arcs avant la déconnexion du pair i, on a :

$$E[M_n] \approx (c+1)\, n \log(n) \tag{1.8}$$

où $E[M_n]$ est le nombre d'arcs espérés du réseau. Après l'exécution de l'algorithme 7 par i, le nombre d'arcs perdus est égal au nombre d'arcs de PV_i et de $c + 1$ arcs supplémentaires. Or,



(a) Le pair 8 choisi $c+1$ voisins de IV à déconnecter (ici 2 et 3).

(b) Un message `inform(6)` est envoyé au pair 1 et un message `inform(7)` est envoyé au pair 5 résultant en une création de deux arcs.

(c) Le pair 8 se déconnecte.

FIGURE 1.21 – Déconnexion du pair 8 (c=1).

$|PV_i| = \frac{E[M_n]}{n}$. On obtient en conséquence :

$$E[M_{n-1}] = E[M_n] - \frac{E[M_n]}{n} - (c+1) \tag{1.9}$$

$$\approx (c+1)(n-1)\left(\log(n) - \frac{1}{n-1}\right) \tag{1.10}$$

$$\approx (c+1)(n-1)\log(n-1) \tag{1.11}$$

1.6.5 HiScamp : Hiérarchisation de SCAMP

De part leur construction, les réseaux logiques aléatoires ne tiennent pas compte de la topologie sous-jacente du réseau et leur utilisation peut être relativement coûteuse pour les routeurs se situant dans le cœur du réseau. C'est par exemple le cas lorsque les successeurs d'un pair sont situés à l'autre extrémité du réseau physique. Une solution à ce problème est apportée par les réseaux aléatoires hiérarchiques et notamment HiScamp [Ganesh *et al.*, 2002], une extension de SCAMP. HiScamp est un réseau logique dans lequel plusieurs réseaux SCAMP sont interconnectés par le biais d'un second réseau logique. Un exemple est présenté à la Fig. 1.22 dans laquelle le réseau logique étiqueté L_2 interconnecte les réseaux logiques SCAMP étiquetés L_1. Dans cette approche, les pairs sont clusterisés en fonction d'une mesure de distance D sur le réseau physique. Cette mesure de distance est définie comme étant le temps d'aller-retour d'un paquet entre deux pairs (*round trip time*). Par conséquent, les pairs proches sur le réseau physique seront dans le même cluster logique. Les pairs ont une vue partielle des pairs de leur cluster et une vue partielle des pairs d'autres clusters. La question est de trouver la taille de chacune des deux vues pour que la diffusion épidémique soit possible dans l'intégralité du réseau et tolérante aux fautes.

Définition 11 *Dans un réseau HiScamp, chaque pair x possède une vue partielle notée $hview_x$ à deux niveaux. Le premier niveau $hview_{x,1}$ contient l'ensemble des successeurs de x dans le même cluster que x. Le second niveau $hview_{x,2}$ contient l'ensemble des successeurs de x dans des clusters différents du cluster de x.*

Définition 12 *Dans un réseau HiScamp, chaque pair x possède une vue partielle notée $iview_x$ qui contient l'ensemble des liens sortant du cluster de x (les liens inter-clusters). On a :*

$$iview_x = \bigcup_{y \in clus(x)} hview_{y,2}$$

avec $clus(x)$ la fonction qui retourne l'ensemble des pairs dans le même cluster que x. Tous les pairs du même cluster partagent la même iview.

Soit un réseau HiScamp ayant M clusters disjoints, avec n pairs dans chaque cluster et $N = Mn$ le nombre de pairs total. Pour tout pair x on a :

$$k = \frac{\sum_x |hview_{x,1}|}{N} \tag{1.12}$$

la taille moyenne des vues intra-cluster et

$$f = \frac{\sum_x |iview_x|}{N} \tag{1.13}$$

la taille moyenne des vues inter-clusters. Alors [Kermarrec *et al.*, 2003] montrent que si

$$f = \log(M) - \log(\beta/2) \tag{1.14}$$

et

$$k = \log(N) - \log(\beta/2) \tag{1.15}$$

avec $\beta > 0$ un paramètre du système, alors la probabilité que le graphe soit connexe est au moins $e^{-\beta}$. Par conséquent, le degré sortant inter-clusters doit être logarithmique en le nombre de clusters et le degré sortant intra-clusters doit être logarithmique en le nombre de pairs du réseau (résultat similaire à l'eq. (1.3)).

Construction du réseau

Lorsqu'un pair s souhaite rejoindre un réseau HiScamp, il trouve le pair j déjà connecté au réseau tel que $D(s,j)$ soit minimale. L'article sur HiScamp [Ganesh *et al.*, 2002] ne fournit pas l'algorithme de recherche de j. Une possibilité, bien que lente et coûteuse, serait de lancer une recherche exhaustive dans le réseau à partir d'un nœud de contact quelconque. Une autre possibilité, nettement moins coûteuse, serait d'effectuer une marche aléatoire à nombre de sauts bornés qui retournerait le meilleur candidat trouvé. Cependant, cette dernière devrait être suffisamment bien calibrée pour retourner un pair du cluster dans lequel s devrait se trouver. Si l'on fait l'hypothèse qu'un tel pair j est trouvé, s envoie un message `subscribe(s)` à j. Lorsqu'un

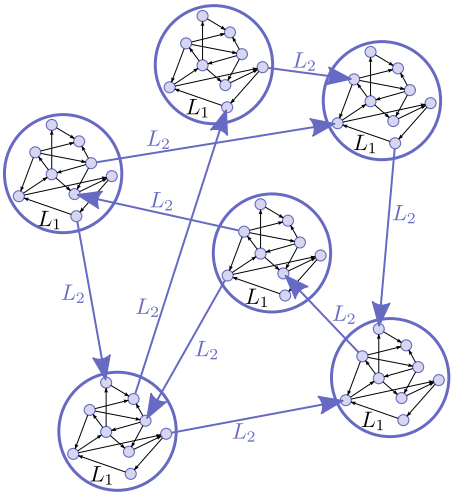


FIGURE 1.22 – Structure d'un réseau HiScamp à deux niveaux L_1 et L_2.

pair j reçoit ce type de message, il exécute l'algorithme 8. Dans cet algorithme, si la distance $D(s,j)$ est inférieure à un certain seuil t, alors le pair s est intégré au cluster de j et la diffusion des messages `intraforward` de SCAMP se fait à l'intérieur du cluster de j. Si $D(s,j) \geq t$ alors s crée son propre cluster et j le connecte aux autres en diffusant des messages `interforward` à destination de $iview_j$. Lorsqu'un pair j reçoit un message `intraforward(s)` (cf. algorithme 9),

Algorithme 8 : Sur réception de `subscribe(s)` par un pair j

```
d ← D(j,s);
si d < t alors
    pour v ∈ hview_{j,1} faire
        EnvoyerMessage(v, intraforward(s));
    pour (i = 0; i < |iview_j| + c − 1; i++) faire
        v_aleat ← ChoixPairAleatoire(hview_{j,1});
        EnvoyerMessage(v_aleat, intraforward(s));
sinon
    pour v ∈ iview_j faire
        EnvoyerMessage(v, interforward(s));
    pour (i = 0; i < c − 1; i++) faire
        v_aleat ← ChoixPairAleatoire(iview_j);
        EnvoyerMessage(v_aleat, interforward(s));
```

il intègre s dans $hview_{j,1}$ avec une probabilité dépendante de la taille de $hview_{j,1}$. Sinon, il propage le message `intraforward(s)` à un de ses voisins dans le cluster. Lorsqu'un pair j reçoit un message `interforward(s)` (cf. algorithme 10), il décide d'ajouter un lien inter-cluster avec une probabilité dépendante de la taille de sa vue inter-clusters. Sinon, il propage la demande à ses clusters voisins. Dans le cas où le pair j a ajouté le cluster de s à ses voisins, il propage

Algorithme 9 : Sur réception de `intraforward(s)` par un pair j

```
tirage ← FloatAleatoire ∈ [0,1];
p ← 1/(|hview_{j,1}|+1);
si s ∉ hview_{j,1} alors
    si tirage ≤ p alors
        hview_{j,1} ← hview_{j,1} ∪ {s};
        return ;
v_aleat ← ChoixPairAleatoire(hview_{j,1});
EnvoyerMessage(v_aleat, intraforward(s));
```

sa nouvelle $iview_j$ à ses voisins du même cluster à l'aide d'un message `gossip`. Un pair j qui reçoit un message `gossip(`$inview_s$`)` (cf. algorithme 11), s'il ne l'a pas déjà traité, met à jour sa propre $inview_j$ puis propage le message à ses voisins du même cluster. Cette procédure garantit que les pairs d'un cluster partagent la même *iview*. L'initialisation des tables de s dépend de

Algorithme 10 : Sur réception de `interforward(s)` par un pair j

```
tirage ← FloatAleatoire ∈ [0,1];
p ← 1/(|iview_j|+1);
si s ∉ hview_{j,2} alors
    si t ≤ p alors
        hview_{j,2} ← hview_{j,2} ∪ {s};
        iview_j ← iview_j ∪ {s};
        pour v ∈ hview_{j,1} faire
            EnvoyerMessage(v, gossip(iview_j));
        return ;
v_aleat ← ChoixPairAleatoire(hview_{j,2});
EnvoyerMessage(v_aleat, interforward(s));
```

la situation :

- s est intégré au cluster de j : $hview_{s,1} = \{j\}$, $hview_{s,2} = \emptyset$ et $iview_s = iview_j$;
- s crée son propre cluster : $hview_{s,1} = \emptyset$ et $hview_{s,2} = iview_s = \{j\}$.

Algorithme 11 : Sur réception de `gossip(`$inview_s$`)` par un pair j

```
si iview_j ≠ iview_s alors
    iview_j ← iview_s;
    for v ∈ hview_{j,1} do
        EnvoyerMessage(v, gossip(iview_s));
```

Équilibrage en arrière plan

Dans l'état actuel, les liens inter-clusters sont uniquement établis entre les pairs qui sont à l'origine des clusters. Cette construction est très mal adaptée aux réseaux dynamiques pour lesquels elle est pourtant destinée puisque les pairs à l'origine des clusters sont des points de vulnérabilité. Pour pallier cette faiblesse, l'algorithme 12 est exécuté périodiquement pour équilibrer les arcs interclusters.

Algorithme 12 : Procédure périodique d'équilibrage des liens inter-clusters par un pair s

```
si hview_{s,1} ≠ ∅ alors
    si |hview_{s,2}| > 1 alors
        p_1 ← ChoixPairAleatoire(hview_{s,2});
        hview_{s,2} ← hview_{s,2} − {p_1};
        p_2 ← ChoixPairAleatoire(hview_{s,1});
        EnvoyerMessage(p_2, interforward(p_1));
```

1.7 Conclusion

Nous avons présenté, dans ce premier chapitre d'état de l'art, un ensemble de réseaux logiques pair-à-pair parmi les plus remarquables. La structure de ces réseaux a suivi une évolution qui part des réseaux ayant un fort degré de centralisation tels que Napster et qui s'achève par l'arrivée des réseaux totalement décentralisés tels que Chord. À travers plusieurs exemples d'architectures, nous avons vu que ce passage à la décentralisation soulève un certain nombre de problématiques qui sont liées à l'inefficacité des algorithmes utilisés pour localiser et récupérer les données dans des réseaux de très grande taille. L'inefficacité de ces algorithmes vient du fait qu'ils supportent à eux seuls l'intégralité de la charge générée par la décentralisation. La solution que la communauté a retenue pour pallier ce manque d'efficacité consiste à modifier la structure des réseaux pour qu'ils supportent à leur tour une partie de cette charge. C'est typiquement le cas des approches de type DHT, dans lesquelles le réseau est structuré en zones d'adressages, chacune responsable d'un certain nombre de données. Cette structure d'adressage garantit que l'accès à chacune des ressources par son adresse est réalisée dans un temps acceptable.

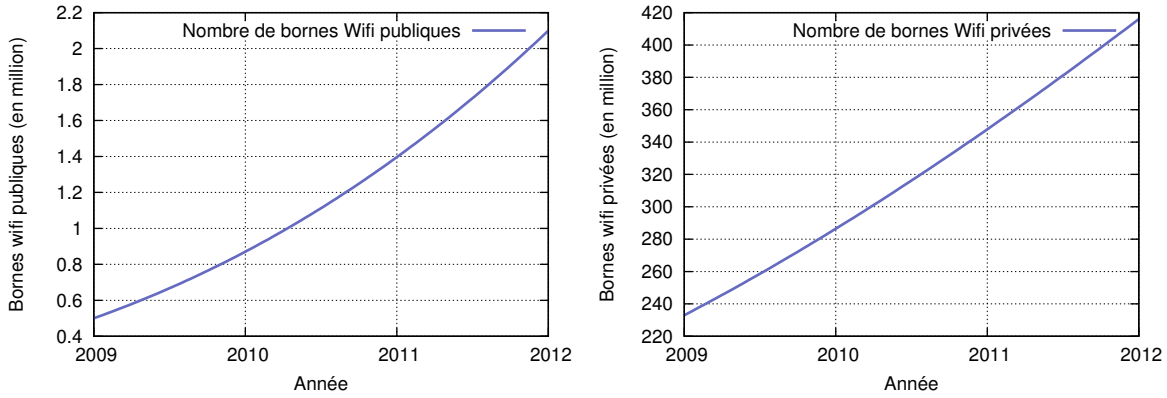
Mais cette structuration des réseaux n'est pas nécessairement une réponse universelle au support des applications pair-à-pair. En effet, les réseaux structurés doivent mettre en place des mécanismes pour préserver leur structure malgré des taux de *churn* qui peuvent être très élevés. Vient s'ajouter à cela le fait que le moindre dysfonctionnement des pairs peut avoir des conséquences terribles sur le résultat du routage. C'est pourquoi, bien que fournissant une recherche plus efficace, les réseaux décentralisés structurés peuvent présenter une moins bonne robustesse aux milieux ouverts (tels que l'internet) que leurs homologues non-structurés. Nous avons justement présenté le cas des applications de diffusion dans les réseaux pair-à-pair qui

peuvent préférer se baser sur des réseaux décentralisés non-structurés dont la robustesse est bien maîtrisée. C'est par exemple le cas de la diffusion multicast avec SCAMP. Ce choix de la robustesse est également celui que nous faisons dans l'application de stockage de données mobiles et décentralisée que nous décrivons dans nos contributions.

Chapitre 2

Disponibilité de l'information dans les réseaux pair-à-pair

Sommaire



(a) Bornes wifi publiques.

(b) Bornes wifi privées.

FIGURE 2.1 – Croissance mondiale des antennes wifi entre 2009 et 2011. (sources : [Wireless Broadband Alliance Ltd., 2011]).

2.1 Problématiques et enjeux

Nous nous sommes intéressés au chapitre 1 à différentes topologies pair-à-pair à travers deux applications : le partage de ressources et la diffusion robuste multicast de niveau applicatif. Lorsque nous avons traité du partage de ressources, nous avons eu l'occasion d'évoquer la notion de popularité d'une donnée. Elle correspond au nombre de pairs connectés au réseau qui peuvent fournir cette donnée. Dans ces applications pair-à-pair en milieu ouvert (Napster, BitTorrent, Gnutella, eDonkey, ...), les pairs peuvent se connecter, se déconnecter ou même stopper de partager la donnée à tout moment. C'est pourquoi, une donnée peut perdre en popularité et disparaître petit à petit du réseau jusqu'à éventuellement se retrouver indisponible. De fait, aucune procédure distribuée n'est mise en place pour assurer la persistance des données au cours du temps. Ce mécanisme d'oubli passif est, en fait, très bien adapté aux échanges de contenus d'actualité puisqu'une donnée populaire pourra être récupérée très rapidement alors qu'une donnée périmée tombera dans l'oubli. Cependant, il reste inadapté pour d'autres usages tels que le stockage persistant.

Les applications de stockage persistant en pair-à-pair se sont développées en parallèle des applications d'échange en réponse à des problématiques plus chères au milieu industriel : la disponibilité et la robustesse des archives numériques large échelle. La problématique n'est plus d'échanger du contenu entre un maximum de participants mais d'offrir une structure décentralisée, robuste, ayant un faible coût de maintenance, qui puisse traiter un très grand nombre de requêtes tout en garantissant que les données qui y sont insérées ne seront jamais perdues et qu'il sera toujours possible d'y accéder. Cette problématique s'est ensuite rapidement étendue au grand public avec l'arrivée massive des terminaux mobiles connectés à internet (*netbooks*, *smartphones*, tablettes, ...) couplée à la croissance des points d'accès sans fil wifi et 3G (cf. Fig. 2.1 et Fig. 2.2). En effet, cette avancée technologique a créé de nouveaux usages tendant

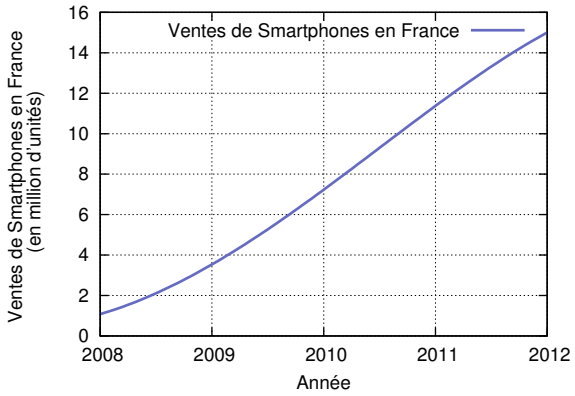


FIGURE 2.2 – Ventes de Smartphones en France. (sources : [GfK Retail and Technology France, 2011]).

vers l'ubiquité. L'utilisateur actuel est quasiment toujours connecté, mobile et possède plusieurs terminaux. Il a donc besoin d'une parfaite synchronisation entre ses terminaux, il souhaite pouvoir accéder à ses données (e-mail, photos, documents textuels,...) de n'importe où, et que cet accès soit entièrement transparent. On regroupe à l'heure actuelle cette tendance sous le terme générique de *Cloud Computing* [Vaquero *et al.*, 2009] dont le *storage as a service* (STaaS) est une composante à part entière. Il faut noter cependant la différence entre les architectures de cloud computing telles que Google Drive, Amazon Cloud Drive, Microsoft Skydrive, DropBox ou encore Apple iCloud qui sont en réalité des clusters privés de serveurs flexibles et les architectures de stockage en pair-à-pair telles que Wuala (dans ses versions initiales), Ubistorage ou encore Allmydata dont la charge de fonctionnement est répartie sur les utilisateurs.

Dans ce chapitre, nous traitons du stockage persistant de documents dans les réseaux pair-à-pair et, plus particulièrement, de la problématique de la disponibilité de l'information dans de telles infrastructures. Dans les applications de partage, le système s'auto-organise de telle sorte que la disponibilité d'une donnée est fonction de sa popularité. Dans le stockage, il faut au contraire contrôler le système pour que les fautes qui surviennent (déconnexions de pairs, pannes de disques, données corrompues, etc.), qu'elles soient définitives ou non, ne risquent pas d'engendrer la perte des données. D'une manière générale, la disponibilité est obtenue par des schémas de l'information qui reposent sur la mise en place de 4 mécanismes :

1. introduire de la redondance dans les données pour tolérer un certain nombre de fautes ponctuelles ;
2. mettre en place une supervision pour connaître l'état de chaque donnée au cours du temps. Nous verrons que l'efficacité de cette supervision est dépendante du placement de la redondance ;
3. réparer les données périodiquement en fonction du résultat de la supervision pour assurer le niveau de redondance dans le temps et par conséquence assurer la durabilité du système ;
4. mettre en place une recherche efficace qui garantisse que toute donnée puisse être récupérée

à n'importe quel moment.

La suite du chapitre est donc consacrée à ces mécanismes ainsi qu'à la présentation d'un certain nombre d'architectures de stockage pair-à-pair persistant parmi les plus connues.

2.2 Redondance des données et tolérance aux fautes

Deux stratégies de redondance sont couramment utilisées pour tolérer les fautes dans les systèmes de stockage persistant en pair-à-pair : la réplication et les codes d'effacement (*erasure coding*). Nous présentons dans cette section les avantages et les inconvénients des deux méthodes et nous commençons par donner une définition de ce que nous considérons comme étant une faute dans un système de stockage.

Définition 13 *On appelle faute d'un pair le fait que le contenu qu'il héberge soit inaccessible au reste du réseau à un certain instant. Si ce pair redevient accessible après une certaine période alors on parle de faute temporaire. Sinon on parle de faute permanente.*

Définition 14 *Si un événement provoque simultanément la faute de plusieurs pairs, alors on parle de fautes corrélées (par opposition aux fautes indépendantes).*

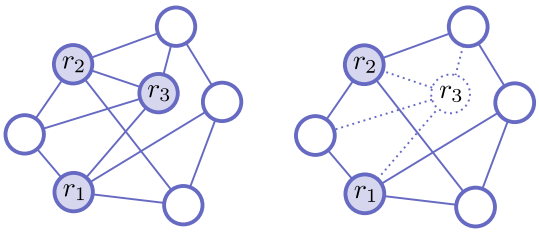
Définition 15 *La disponibilité A d'une donnée est la probabilité que cette donnée soit disponible sur la période de temps observée.*

La réplication comme les codes d'effacement proposent de dupliquer les données et de répartir ces répliques sur des pairs distincts. Avec ce mécanisme, lorsqu'une faute se produit et touche un des pairs hébergeant les répliques, alors seule la réplique touchée se retrouve indisponible et la donnée initiale est toujours accessible à partir des autres répliques (cf. Fig. 2.3). Un système est donc en mesure de tolérer un certain nombre de fautes qui est dépendant de la quantité de la redondance. La difficulté consiste à trouver le bon niveau de redondance en fonction de la disponibilité souhaitée sans pour autant surcharger l'espace de stockage du système. Pour analyser la performance des différentes stratégies de redondance, nous reprenons les définitions de [Duminuco, 2009].

Définition 16 *Le niveau de robustesse d'un schéma de redondance est donné par le nombre de fautes successives qu'il peut tolérer. On note $P(dataloss|l)$ la probabilité de perdre la donnée (dataloss) sachant qu'elle a subi l fautes.*

Définition 17 *Soit $|data|$ l'espace occupé par la donnée initiale et soit $|data_{red}|$ l'espace occupé par la donnée une fois que le schéma de redondance lui a été appliqué (i.e., donnée initiale + redondance). Alors, on appelle facteur de redondance :*

$$\omega = \frac{|data_{red}|}{|data|} \tag{2.1}$$



(a) Réseau pair-à-pair contenant une donnée ayant 3 éléments dupliqués r_1 r_2 et r_3.

(b) Lorsqu'une faute rend indisponible le pair hébergeant r_3, la donnée est toujours accessible à partir de r_1 et r_2.

FIGURE 2.3 – Mécanisme de redondance de données.

Nous verrons à la Sec. 2.3 qu'une donnée est réparée périodiquement pour maintenir le niveau de redondance souhaité au cours du temps. Cette réparation consiste à réintroduire les répliques perdues à l'aide de celles qui sont toujours disponibles sur les autres pairs.

Définition 18 *On appelle degré de réparation, que l'on note d, le nombre de pairs impliqués dans le processus de réparation.*

L'opération de lecture sur un pair a un coût qui se matérialise par du trafic réseau mais également d'autres facteurs tels que la charge sur le pair ou encore des opérations de synchronisation entre pairs. On cherche donc à minimiser d et ω.

2.2.1 Réplication

La réplication est le mécanisme de redondance de base. Il consiste à prendre une donnée et à la dupliquer à l'identique un certain nombre de fois. Si l'on suppose qu'on a N_{rep} répliques dans le réseau alors un tel schéma possède les propriétés suivantes :

un facteur de redondance $\omega = N_{rep}$;

un degré de réparation $d = 1$;

une robustesse

$$P(dataloss|l) = \begin{cases} 0 & \text{si } l < N_{rep} \\ 1 & \text{sinon.} \end{cases} \tag{2.2}$$

C'est-à-dire qu'il est possible de reconstruire les répliques manquantes si au moins une réplique est toujours disponible dans le réseau. Ou encore que le schéma est capable de tolérer $N_{rep} - 1$ fautes.

2.2.2 Codes d'effacement

Les codes d'effacement [Rizzo, 1997, Plank, 1997, Plank et Ding, 2005] ont été proposés pour le stockage décentralisé [Kubiatowicz *et al.*, 2000, Adya *et al.*, 2002] dans le but de diminuer le

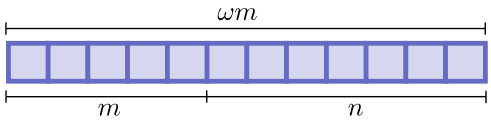


FIGURE 2.4 – Découpage d'une donnée par un (m,n)-codes d'effacement.

coût de stockage engendré par la réplication au détriment du degré de réparation. En effet, les systèmes reposant sur des codes d'effacement nécessitent un espace de stockage beaucoup moins important tout en fournissant la même tolérance aux fautes que les systèmes basés sur de la réplication [Weatherspoon et Kubiatowicz, 2002].

Un (m,n)-code d'effacement est un code qui découpe une donnée (data) en m blocs. Après une phase d'encodage, $m+n$ fragments d'information sont générés. Les codes d'effacement nous garantissent que la donnée peut être reconstruite à partir de n'importe quel sous-ensemble de m fragments parmi les $m+n$. Ce code permet donc de tolérer n fautes. Le coût de stockage supplémentaire induit par un (m,n)-code d'effacement est donné par

$$C_{ec} = n \cdot \frac{|data|}{m} \tag{2.3}$$

et celui de la réplication pour la même tolérance aux fautes est donné par

$$C_r = n \cdot |data| \tag{2.4}$$

avec $|data|$ la taille de la donnée. On note que les valeurs de m et n sont un compromis à choisir entre le niveau de robustesse souhaité et les contraintes sur l'espace de stockage utilisé. Nous donnons, à ce propos, à la Sec. 2.5 un éclairage sur le choix de ces valeurs en fonction de la disponibilité requise dans le système et du taux fautes observé.

Un tel schéma de code d'effacement (cf. Fig. 2.4) possède donc les propriétés suivantes :

un facteur de redondance $\omega = \frac{n}{m} + 1$;

un degré de réparation $d = m$;

une robustesse

$$P(dataloss|l) = \begin{cases} 0 & \text{si } l \leq n \\ 1 & \text{sinon.} \end{cases} \tag{2.5}$$

Par conséquent, un code d'effacement est théoriquement beaucoup moins coûteux en terme d'espace de stockage qu'un mécanisme de réplication tout en fournissant le même niveau de disponibilité. Cependant, il est à noter que les codes d'effacement complexifient et alourdissent la conception du système puisqu'ils empêchent l'accès en mode bloc à la donnée et impliquent obligatoirement m pairs différents dans le processus de reconstruction là où la réplication n'en implique qu'un seul. Dans leurs travaux [Blake et Rodrigues, 2003, Rodrigues et Liskov, 2005] font une comparaison de la réplication et des codes d'effacement en prenant en compte la bande passante requise pour supporter les deux schémas. Ils montrent que les codes d'effacement sont intéressants en terme de gain de stockage lorsque la disponibilité des pairs est faible. Cependant,

le coût de maintenance engendré peut être trop important pour des utilisateurs classiques (la bande passante moyenne des pairs dans leurs évaluations est de 100 kbp/s). C'est pourquoi ils proposent une stratégie hybride dans laquelle une donnée possède une réplique ainsi que des fragments codés par un code d'effacement. Lorsque des blocs sont perdus, il peuvent être réparés à partir de la réplique ce qui supprime le problème du degré de réparation élevé. Cependant, lorsque la réplique est perdue et qu'elle est en train d'être reconstruite, aucun bloc ne peut être régénéré temporairement.

2.2.3 Codes linéaires

Les codes linéaires sont une famille de codes d'effacement. Ils consistent à générer les $m+n$ fragments comme des combinaisons linéaires des m blocs initiaux. Les opérations d'encodage et de décodage sont effectuées sur les corps finis puisque ces derniers sont munis d'opérations internes qui garantissent que tous les calculs effectués restent dans le corps considéré. De manière générale, les données sont encodées sur q bits et prennent leurs valeurs dans un corps fini de cardinal 2^q que l'on note $\mathbb{K}_{2^q}$.

Chaque bloc est codé sur $\mathbb{K}_{2^q}$ de sorte à obtenir le vecteur B_m des m blocs. Un vecteur F_{m+n} de fragments est ensuite généré à partir d'une matrice d'encodage $E_{m+n,m}$ ayant $m+n$ lignes et m colonnes comme suit :

$$F_{m+n} = E_{m+n,m} B_m \tag{2.6}$$

Les $m+n$ fragments, associés à leur numéro de ligne, peuvent ensuite être disséminés dans le réseau. Lorsque n fragments on été perdus, on construit F^*_m la matrice des fragments restants. Il est possible de retrouver B_m en calculant :

$$B_m = E^{*-1}_{m,m} F^*_m \tag{2.7}$$

avec $E^*_{m,m}$ la matrice carrée dont les lignes sont une sélection des m lignes de $E_{m+n,m}$ qui correspondent aux m lignes de F^*_m. On remarque que pour que le décodage soit possible la matrice $E^*_{m,m}$ doit être inversible.

2.2.4 Codes de Reed-Solomon

Les codes de Reed-Solomon [Reed et Solomon, 1960, Plank, 1997, Plank et Ding, 2005] sont une implémentation célèbre des codes linéaires, notamment utilisée de manière intensive dans la correction des erreurs de transmissions réseaux telles que la transmission satellite, la transmission ADSL ou encore lors de la lecture de CD/DVD. Un code de Reed-Solomon, noté $RS(m,n)$, utilise une matrice d'encodage $E_{m+n,m}$ construite comme suit :

$$E_{m+n,m} = \begin{pmatrix} I_{m,m} \\ V_{n,m} \end{pmatrix} \tag{2.8}$$

avec $I_{m,m}$ la matrice identité et $V_{n,m}$ une matrice de Vandermonde :

$$v_{i,j} = j^{i-1} \tag{2.9}$$

on a donc :

$$E_{m+n,m} = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & m \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 2^{n-1} & 3^{n-1} & \dots & m^{n-1} \end{pmatrix} \tag{2.10}$$

La matrice de l'Eq. 2.9 lorsqu'elle est carrée a tous ses vecteurs indépendants. Par conséquent, toute sous-matrice $E'_{m,m}$ de $E_{m+n,m}$ est inversible rendant ainsi le décodage possible.

Exemple 19 *On souhaite encoder la donnée* 000101|001000|000110|000010 *découpée en blocs de 6 bits avec un RS(4,2). On construit le vecteur des blocs et la matrice d'encodage dans* $\mathbb{K}_{2^6}$ *:*

$$E = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 5 \\ 8 \\ 6 \\ 2 \end{pmatrix}$$

puis on calcule les fragments :

$$F = EB = \begin{pmatrix} 5 \\ 8 \\ 6 \\ 2 \\ 21 \\ 47 \end{pmatrix}$$

À la suite de deux fautes, les fragments de valeur 5 et 6 sont perdus. Il reste donc 4 fragments et la donnée peut être reconstruite. Pour reconstruire les blocs initiaux on construit F^* *le vecteur des fragments toujours disponibles,* E^* *la matrice d'encodage filtrée avec les lignes de* F^* *et* E^{*-1} *son inverse :*

$$F^* = \begin{pmatrix} 8 \\ 2 \\ 21 \\ 47 \end{pmatrix} \quad E^* = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 2 & 3 & 4 \end{pmatrix} \quad E^{*-1} = \begin{pmatrix} -0.5 & 0.5 & 1.5 & -0.5 \\ 1 & 0 & 0 & 0 \\ -0.5 & -1.5 & -0.5 & 0.5 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

On peut donc finalement décoder les fragments et retrouver B :

$$B = \begin{pmatrix} -0.5 & 0.5 & 1.5 & -0.5 \\ 1 & 0 & 0 & 0 \\ -0.5 & -1.5 & -0.5 & 0.5 \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 8 \\ 2 \\ 21 \\ 47 \end{pmatrix} = \begin{pmatrix} 5 \\ 8 \\ 6 \\ 2 \end{pmatrix}$$

2.2.5 Codes linéaires aléatoires

Les codes linéaires aléatoires [Koetter et Médard, 2003, Li et Yeung, 2003, Lin *et al.*, 2009] sont des codes basés sur une matrice d'encodage à coefficients aléatoires. [Szymonacedaski *et al.*, 2005] ont montré que si le corps fini est suffisamment grand (i.e., $q \geq 16$) la probabilité que toute sous-matrice $m \times m$ de la matrice d'encodage soit inversible est égale à 1. L'algorithme de reconstruction des blocs initiaux est alors identique à celui que nous avons décrit pour les codes de Reed-Solomon. Avec ce type de code, il est possible de regénérer des fragments perdus sans reconstruire la donnée [Li *et al.*, 2010]. Lorsqu'un nouveau pair (*newcomer*) souhaite regénérer un fragment, il crée le vecteur F^*_m à partir de m fragments encore disponibles et filtre la matrice d'encodage pour obtenir la sous-matrice $E^*_{m,m}$ de coefficients correspondant aux vecteurs de F^*. Tout comme l'encodage et le décodage, la regénération est basée sur des combinaisons linéaires de fragments. Soit un nouveau vecteur de coefficients $(\sigma_1, \ldots, \sigma_m)^T$, $\sigma_j \in \mathbb{K}_{2^q}$ on pose :

$$(\sigma_1, \ldots, \sigma_m)B_m = (r_1, \ldots, r_m)F^*_m \tag{2.11}$$

avec $r_j \in \mathbb{K}_{2^q}$. On remplace B_m de l'eq. 2.7 dans l'eq. 2.11 et on obtient :

$$(r_1, \ldots, r_m) = (\sigma_1, \ldots, \sigma_m)E^{*-1}_{m,m} \tag{2.12}$$

Le nouveau pair peut encoder les fragments F^*_m avec le vecteur $(r_1, \ldots, r_m)^T$ et mettre la matrice d'encodage $E_{m+n,n}$ à jour avec le nouveau vecteur de coefficients $(\sigma_1, \ldots, \sigma_m)$. Ces nouveaux coefficients sont tirés aléatoirement dans $\mathbb{K}_{2^q}$ quand $q \geq 16$ pour garantir que la nouvelle matrice d'encodage soit toujours inversible. Ce type de code permet d'éviter l'inversion d'une très grosse matrice à chaque réparation.

Exemple 20 *Soit :*

$$B = \begin{pmatrix} 16 \\ 5 \end{pmatrix} \quad E = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \\ 1 & 2 \end{pmatrix} \quad F = EB = \begin{pmatrix} 16 \\ 5 \\ 21 \\ 26 \end{pmatrix}$$

À la suite de deux fautes, les fragments de valeur 5 et 21 sont perdus. Un nouvel arrivant souhaite regénérer un fragment. Il récupère le vecteur de fragments restants et calcule la matrice filtrée E^* :

$$F^* = \begin{pmatrix} 16 \\ 26 \end{pmatrix} \quad E^* = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}$$

puis choisi un nouveau vecteur de coefficients $r_3 = (-3 \;\; 4)$ *aléatoirement dans* $\mathbb{K}_{2^q}$ *et calcule le nouveau fragment* F_3 *par combinaisons linéaires (cf. Eq. 2.11) :*

$$F_3 = (-3 \;\; 4) \begin{pmatrix} 16 \\ 26 \end{pmatrix} = 56$$

Le vecteur de fragments disponibles devient alors :

$$F = \begin{pmatrix} 16 \\ \emptyset \\ 56 \\ 26 \end{pmatrix}$$

La matrice d'encodage E *est mise à jour par une matrice* E_1 *(cf. Eq. 2.12) :*

$$\sigma_3 = r_3 E^* = (-3 \;\; 4) \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} = (1 \;\; 8) \qquad E_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 8 \\ 1 & 2 \end{pmatrix}$$

Si maintenant un pair souhaite reconstruire la donnée initiale B *à partir des fragments 1 et 3 :*

$$F' = \begin{pmatrix} 16 \\ 56 \end{pmatrix}$$

il inverse la matrice E_1 *filtrée sur les lignes de* F'

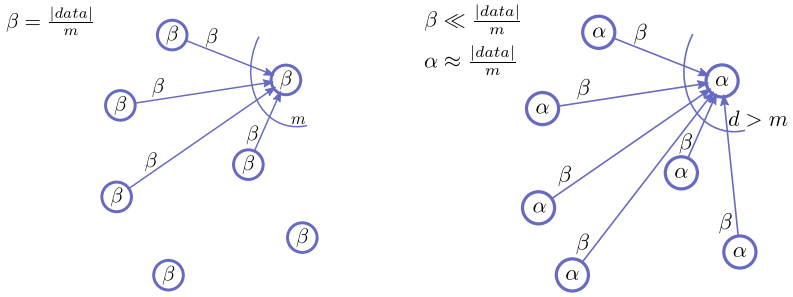
$${E'_1}^{-1} = \begin{pmatrix} 1 & 0 \\ 1 & 8 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 0 \\ -0.125 & 0.125 \end{pmatrix}$$

et peut enfin calculer les blocs initiaux (cf. Eq. 2.7) :

$$B = {E'_1}^{-1} F' = \begin{pmatrix} 16 \\ 5 \end{pmatrix}$$

2.2.6 Codes régénérant

Le degré $d = m$ des codes d'effacement classiques a un coût important en bande passante. [Rodrigues et Liskov, 2005] mettent en évidence le fait que les codes d'effacement, bien que plus légers en terme de stockage, peuvent avoir des performances en terme de disponibilité de l'information qui sont inférieures à la réplication lorsque la bande passante est faible. Face à ce constat, un nouveau type de codes est proposé dans la littérature : les codes régénérant (*regenerating codes*) [Wu *et al.*, 2007, Dimakis *et al.*, 2010, Suh et Ramchandran, 2010, Dimakis *et al.*, 2011, Kermarrec *et al.*, 2011], qui sont un sous-type de codage réseau (*network coding*) [Jaggi *et al.*, 2005]. L'idée développée dans ces travaux est d'augmenter le degré de réparation $d > m$ mais de diminuer la quantité d'information $\beta \ll \frac{|data|}{m}$ récupérée sur chacun des d pairs et



(a) Code d'effacement : pour reconstruire un fragment il est nécessaire de récupérer m fragments de taille β.

(b) Code régénérant : pour reconstruire un fragment il est nécessaire de récupérer $d > m$ parties de fragments de taille β.

FIGURE 2.5 – Différence entre codes d'effacement et codes régénérant.

donc d'économiser de la bande passante. Dans la Fig. 2.5, qui illustre ce principe, les fragments d'un code régénérant sont de taille $\alpha \approx \frac{|data|}{m}$. Lorsqu'une regénération a lieu, le nouveau pair récupère $d > m$ parties de fragments de taille $\beta \ll \frac{|data|}{m}$ et les recombinent entre elles pour créer un nouveau fragment de taille α.

Les codes régénérant permettent d'obtenir un compromis entre coût de stockage α et coût de refragmentation $\gamma = d\beta$. Ce compromis a été analysé dans le détail dans les travaux de [Wu *et al.*, 2007, Dimakis *et al.*, 2010] qui définissent la fonction :

$$\alpha^*(k,m,d,\gamma) = \begin{cases} \frac{\mathcal{M}}{m}, & \gamma \in [f(0), +\infty) \\ \frac{\mathcal{M}-g(i)\gamma}{m-i}, & \gamma \in [f(i), f(i-1)), i = 1, \ldots, m-1 \end{cases} \tag{2.13}$$

avec

$$k = m + n \tag{2.14}$$

$$\mathcal{M} = |data| \tag{2.15}$$

$$f(i) = \frac{2\mathcal{M}d}{(2m-i-1)i + 2m(d-m+1)} \tag{2.16}$$

$$g(i) = \frac{(2d-2m+i+1)i}{2d} \tag{2.17}$$

Pour tout $\alpha \geq \alpha^*(n,k,d,\gamma)$, il est possible d'obtenir un codage valide aux points (n,k,d,α,γ) par un codage réseau linéaire. La fonction α^* est appelée fonction de seuil et la Fig. 2.6 présente son graphe pour les paramètres $\mathcal{M} = 1$, $m = 5$, $k = 10$, $d = k - 1$. Le point MSR (minimum-storage regenerating) représente le code permettant d'obtenir un coût de stockage optimal pour un faible coût en bande passante. Respectivement, le point MBR (minimum-bandwidth regenerating) offre un coût en bande passante optimal pour un coût de stockage réduit. Le point EC correspond à un code d'effacement classique de coordonnées $(1, \mathcal{M}/m)$. Les codes régénérant sont, de fait, une généralisation des codes d'effacement. La bande passante de réparation minimale γ est donnée

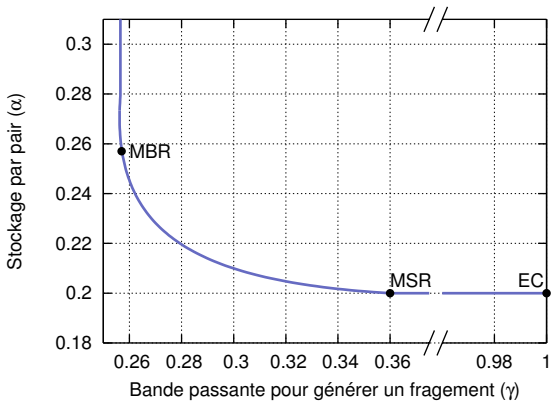


FIGURE 2.6 – Fonction de seuil α^* de paramètres $\mathcal{M} = 1$, $m = 5$, $k = 10$, $d = k - 1$.

	m	d	α	γ
Code d'effacement	32	N/A	1Mo	32 Mo
MSR	32	36	1Mo	7.2Mo
MBR	32	36	1.8Mo	1.8Mo

TABLE 2.1 – Différences entre codes d'effacement et codes régénérant pour $\mathcal{M} = 32$ Mo.

par :

$$\gamma_{min} = f(m-1) = \frac{2\mathcal{M}d}{2md - m^2 + m} \tag{2.18}$$

[Wu *et al.*, 2007] montrent que :

$$MSR = \left(\frac{\mathcal{M}d}{m(d-m+1)}, \frac{\mathcal{M}}{m} \right) \tag{2.19}$$

et

$$MBR = \left(\frac{2\mathcal{M}d}{2md - m^2 + m}, \frac{2\mathcal{M}d}{2md - m^2 + m} \right) \tag{2.20}$$

[Kermarrec *et al.*, 2011] donnent un tableau comparatif de ces trois types de codage sur une donnée de 32 Mo séparée en 32 blocs initiaux (cf. Tab. 2.1). On constate le compromis obtenu entre MSR et MBR par rapport aux codes d'effacement de type Reed-Solomon.

2.2.7 Codes hiérarchiques

Dans sa thèse, Alessandro Duminuco [Duminuco, 2009] propose une implémentation des codes régénérant et fait une analyse de leurs performances. Il montre que bien que fournissant un compromis coût de stockage/coût réseau, les codes régénérant ont des taux de codage/décodage beaucoup plus bas que les codes d'effacement classiques. C'est face à ce problème qu'il propose les codes hiérarchiques [Duminuco et Biersack, 2008, Duminuco et Biersack, 2009], un type de code permettant de réduire le degré de réparation d par rapport aux codes d'effacement linéaires

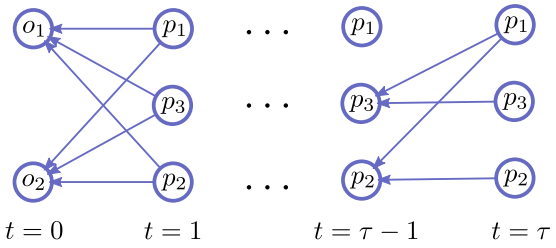


FIGURE 2.7 – Exemple de graphe de flot d'information pour un (2,1)-code d'effacement.

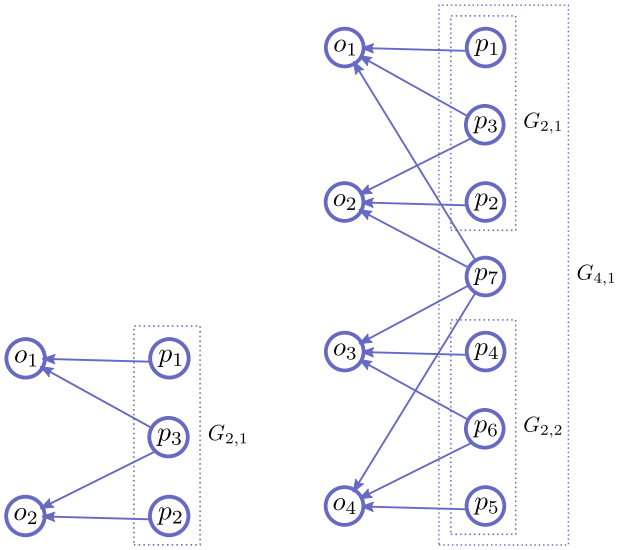
classiques tout en conservant un niveau de disponibilité similaire à ces derniers et des taux de codage/décodage supérieurs aux codes régénérant.

Les codes hiérarchiques sont une hybridation dans laquelle on cherche un compromis entre la réplication de fragments et les codes d'effacement pour réduire le degré de réparation. Ces codes reposent sur un outil théorique appelé graphe de flot d'information (*information flow graph*) qui permet de représenter le statut des fragments et leurs combinaisons linéaires au cours du temps. Un exemple [Duminuco, 2009] de graphe de flot d'information d'un (2,1)-code d'effacement est présenté à la Fig. 2.7. Dans un tel graphe, les sommets représentent les fragments et les arcs représentent la manière dont ils sont combinés. Dans cette figure, chaque fragment p_i à $t = 1$ est une combinaison linéaire des deux blocs initiaux o_1 et o_2. Le compteur discret t est incrémenté à chaque fois qu'un bloc est perdu ou qu'une regénération survient. Par exemple, lorsque le compteur $t = \tau - 1$, le bloc p_1 est perdu car il n'a aucun successeur au temps $t = \tau$. Au temps $t = \tau$, le fragment p_1 est regénéré par combinaison linéaire des deux blocs restant. p_2 et p_3 quand à eux ont un unique prédécesseur à $t = \tau - 1$, signifiant qu'ils n'ont pas été perdus ni recombinés. Cet outil permet de connaître le degré de réparation de n'importe quel code grâce au théorème suivant [Dimakis *et al.*, 2010] :

Théorème 21 *Soit un* (m,n)*-code et son graphe de flot d'information* G*. Une sélection* P^m *de* m *fragments au temps* t *est suffisante pour reconstruire la donnée initiale s'il est possible de trouver dans* G*,* m *chemins disjoints des sommets sources aux* P^m *sommets cibles.*

Un code hiérarchique se construit comme suit (un exemple de (4,3)-code hiérarchique est donné à la Fig. 2.8) :

1. construire un (m_0,n_0)-code linéaire aléatoire. On note l'ensemble des fragments générés G_{d_0}, avec $d_0 = m_0$;
2. choisir deux paramètres g_1 et h_1. Répliquer G_{d_0} g_1 fois de sorte à obtenir g_1 groupes $G_{d_0,1}, \ldots, G_{d_0,g_1}$ et créer h_1 fragments supplémentaires à partir des $g_1 m_0$ blocs initiaux. On obtient l'ensemble de fragments $G_{d_1,1}$ correspondant à un (d_1,H_1)-code hiérarchique avec $H_1 = g_1 n_0 + h_1$ et $d_1 = g_1 m_0 = g_1 d_0$;
3. répéter l'étape précédente autant de fois que nécessaire. À l'étape s, on choisit g_s et h_s et on réplique la structure $G_{d_{s-1},1}$, G_s fois. On crée ensuite h_s fragments supplémentaires à partir des blocs initiaux et correspondant à un degré $d_s = g_s d_{s-1}$. L'ensemble des fragments



(a) Première étape de construction pour $m_0 = 2$ et $n_0 = 1$.

(b) Seconde étape avec $g_1 = 2$ et $h_1 = 1$.

FIGURE 2.8 – Exemple de graphe d'information d'un (4,3)-code hiérarchique.

constitue le groupe $G_{d_s,1}$ correspondant au (d_s,H_s)-code hiérarchique avec $H_s = g_s H_{s-1} + h_s$.

[Duminuco, 2009] fait la proposition suivante :

Proposition 22 *c'est Soit un graphe de flot d'information d'un (m,n)-code hiérarchique au temps t et soit p un fragment réparé au temps t. On note G(p) la hiérarchie des groupes qui contiennent p et on note R(p) l'ensemble des fragments au temps t − 1 qui ont été recombinés pour reconstruire p.*

Si *$\forall t$ et $\forall p$, $R(p)$ vérifie :*

1. *$|G_{d,i} \cap R(p)| \leq d$ $\forall G_{d,i}$ appartenant au code ;*
2. *$\exists G_{d,i} \in G(p) : R(p) \subseteq G_{d,i}, |R(p)| = d$.*

La condition (1) signifie que parmi l'ensemble des R(p) fragments recombinés, il ne peut y avoir qu'un maximum de d fragments choisis dans n'importe quel niveau $G_{d,i}$. La condition (2) signifie qu'il doit exister un groupe dans la hiérarchie G(b) qui contienne tous les fragments recombinés et que leur nombre doit être égal au degré de combinaison utilisé dans ce groupe.

Alors *le code ne se dégrade pas. i.e., il vérifie le Th. 21.*

Avec ce type de code, la regénération de fragments n'est plus dépendante d'une seule valeur de degré de réparation mais elle dépend du fragment qui a été perdu. En effet, dans l'exemple de la Fig 2.8, le fragment p_1 peut être réparé de deux manières :

1. en utilisant 2 fragments de son groupe i.e., p_2 et p_3 (groupe $G_{2,1}$) ;

	Fautes(l)		
	1	2	3
$P(d=2\|l)$	0.86	0.42	0
$P(d=4\|l)$	0.14	0.58	0.77
$P(dataloss\|l)$	0	0	0.23

TABLE 2.2 – Table de $P(d|l)$ et de $P(dataloss|l)$ pour un (4,3)-code hiérarchique.

2. à partir de n'importe quelle combinaison de 4 fragments parmi les autres (groupe $G_{4,1}$). Par exemple p_3, p_7, p_4, p_6.

Le degré de réparation est, par conséquent, dépendant des fragments qui sont disponibles au moment de la réparation. Duminuco propose une table (cf. Tab. 2.2) qui donne la probabilité $P(d|l)$ qu'une refragmentation nécessite, au pire cas, un degré d sachant l fautes successives pour un (4,3)-code hiérarchique. On constate donc qu'avec ce code il est possible dans 86% des cas de reconstruire un fragment avec un degré deux fois plus faible qu'un code d'effacement classique lorsqu'une seule faute a été subie. Ce gain en bande passante est obtenu au détriment du niveau de tolérance aux fautes qui est diminué comme nous pouvons le constater à la dernière ligne de ce tableau. En effet, dans un (4,3)-code d'effacement classique, $P(dataloss|l=3)=0$ alors que pour un (4,3)-code hiérarchiques $P(dataloss|l=3)=0.23$. Cet inconvénient peut être néanmoins évité par des regénérations plus fréquentes et moins coûteuses.

2.3 Durabilité du système

Nous avons vu à la section précédente les techniques de redondance qui permettent d'obtenir de la tolérance aux fautes ponctuelles dans les systèmes de stockage distribués. i.e., les données sont capables de tolérer un certain nombre de fautes successives avant d'être perdues définitivement. Pour maintenir ce niveau de tolérance au cours du temps, il est nécessaire de réparer périodiquement les répliques (ou les fragments) qui sont perdues. Ces réparations sont déclenchées à la suite d'une phase de supervision qui compte le nombre de répliques disponibles dans le système. Cette section présente les mécanismes de supervision et de réparation et traite également du problème du placement des répliques lorsque ces réparations sont effectuées de manière décentralisée.

2.3.1 Politiques de réparation

Une politique de réparation définit le moment à partir duquel une donnée doit être réparée (i.e., lorsque des éléments redondants doivent être regénérés [8]). Cette décision de réparation est

8. Ces éléments redondants sont des fragments pour les approches à bases de codes et des répliques pour la réplication. Dans la suite nous employons le terme fragment pour désigner ces éléments redondants mais les mécanismes décrits sont également applicables à la réplication.

prise en fonction du nombre de fragments disponibles dans le réseau à l'issue de la phase de supervision. On distingue dans la littérature deux politiques de réparation.

Politique de réparation immédiate

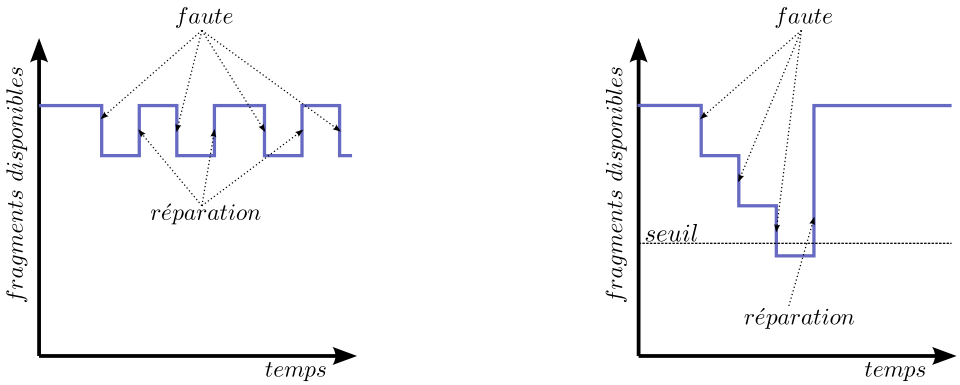
Cette politique réactive, également appelée politique agressive (*eager repair*), consiste à reconstruire l'intégralité des fragments perdus à chaque fois que des fautes ont été détectées. Plus formellement, on considère une donnée fragmentée selon un (m,n)-code d'effacement. On insère donc $k = m + n$ fragments dans le réseau. Après un certain temps, le processus de supervision s'exécute et compte $k' < k$ fragments disponibles. Dans le cas où $k' < m$, aucune réparation n'est possible. Par contre, lorsque $m \leq k' < k$, cette politique de réparation reconstruit $k - k'$ fragments. Le procédé de réparation n'est évidemment pas le même en fonction du schéma de redondance choisi.

- si le schéma de redondance est basé sur un modèle de réplication, le superviseur peut récupérer une réplique et l'envoyer à $k - k'$ nouveaux pairs. Dans un souci d'équilibrage de charge, une variante consiste à contacter $k - k'$ pairs et à leur demander d'envoyer une copie de leur réplique à d'autres pairs ;
- si le schéma de redondance est basé sur un code d'effacement, alors, en fonction de son degré de réparation d, le superviseur contacte d pairs et récupère d fragments. Il insère ensuite les $k - k'$ fragments manquants dans le réseau.

Cette approche ne permet pas de prendre en compte les fautes temporaires (cf. Def. 13) et n'est pas réellement appropriée pour l'utilisation de codes d'effacement. En effet, dans le cas où une donnée fragmentée subit une faute entre chaque supervision, il est nécessaire à chaque supervision de lancer un processus de réparation impliquant d pairs ; processus bien plus lourd que pour la réplication. Cette politique est utilisée dans [Dabek *et al.*, 2001, Adya *et al.*, 2002, Haeberlen *et al.*, 2005, Zhang *et al.*, 2007].

Politique de réparation différée

La politique précédente permet au système d'être réactif au détriment d'une consommation réseau plus importante dans le cas où des codes sont utilisés. Cependant, cette surconsommation supplémentaire peut être évitée si le nombre moyen de fautes survenant entre plusieurs supervisions ne descend pas en-dessous du nombre de fautes tolérables par le schéma de redondance utilisé. La politique différée, également appelée politique paresseuse (*lazy repair*) ou politique à seuil (*threshold repair*), est basée sur ce principe. Elle consiste à différer la réparation tant qu'un certain seuil de fragments perdus n'a pas été atteint. Plus formellement, on considère une donnée fragmentée selon un (m,n)-code d'effacement. On insère donc $k = m + n$ fragments dans le réseau. On définit ensuite un seuil de réparation r tel que $m \leq r < k$. Après un certain temps, le processus de supervision s'exécute et compte $k' < k$ fragments disponibles. Si $k' < m$ alors la donnée ne peut pas être réparée. Si $k' > r$ alors la réparation est différée à une prochaine supervision. Par contre, lorsque $m \leq k' \leq r$, les $k - k'$ fragments manquants sont regénérés. Toute la



(a) Politique de réparation immédiate.

(b) Politique de réparation différée.

FIGURE 2.9 – Différences entre politique de réparation immédiate et politique de réparation différée.

difficulté d'une telle approche consiste à trouver un modèle de disponibilité suffisamment précis pour que le choix de r ne diminue pas la robustesse du système. Cette politique est plus adaptée aux fautes temporaires puisqu'elle laisse la possibilité à un pair temporairement indisponible de revenir avec ses fragments avant que le seuil ne soit atteint. Elle est également conseillée si le schéma de redondance est un code d'effacement car elle évite de lancer trop de processus de réparations. La politique différée est décrite notamment dans [Duminuco, 2009, Dalle *et al.*, 2009, Dandoush, 2010] et utilisée dans [Ghemawat *et al.*, 2003, Bhagwan *et al.*, 2004]. [Kermarrec *et al.*, 2011] proposent le schéma de la Fig. 2.9 pour illustrer la différence entre les deux politiques de réparation. En réalité, la politique immédiate est un cas particulier de politique différée ayant un seuil de réparation $r = 1$.

2.3.2 Politiques de placement et de supervision

Les procédures qui permettent de superviser, d'insérer ou de récupérer des fragments sont dépendantes du placement de ces fragments dans le réseau. C'est en effet la manière dont sont placées les répliques qui détermine la façon d'y accéder et ce placement a un impact sur les performances du système [Lian *et al.*, 2005, Giroire *et al.*, 2009a, Giroire *et al.*, 2009b, Caron *et al.*, 2010]. Nous proposons de détailler dans cette section les deux politiques de placement et de supervision qui sont majoritairement utilisées dans les systèmes de stockage distribués.

Politique de placement global

Dans la politique de placement global, chaque donnée est supervisée par un pair qui est choisi aléatoirement parmi l'ensemble des pairs du réseau. Les k fragments de redondance de cette donnée sont disséminés sur k pairs également choisis aléatoirement. Ce schéma de supervision est illustré à la Fig. 2.10(a). Dans cette politique, lorsqu'une réparation doit être déclenchée, c'est

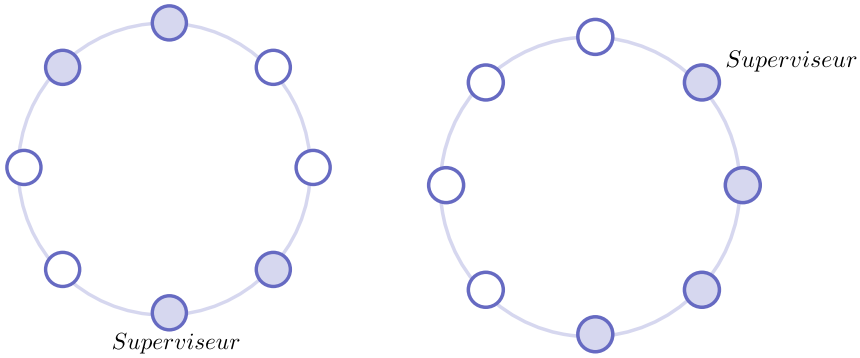
le superviseur qui récupère les d fragments et regénère puis dissémine les nouveaux fragments. Cette politique a l'avantage de répartir uniformément les répliques sur l'intégralité du réseau et d'éviter, par conséquent, les goulots d'étranglement lorsqu'une faute survient. Cet équilibrage permet notamment d'obtenir les meilleurs délais de réparation. En contre partie, il est nécessaire d'avoir un mécanisme qui permette d'insérer aléatoirement des fragments. Dans une architecture décentralisée, si le réseau pair-à-pair sous-jacent est une DHT alors il suffit de tirer un nombre aléatoire sur l'espace d'adressage correspondant et de prendre le pair responsable de ce nombre. Par contre, si le réseau est non-structuré, des méthodes à bases de marches aléatoires peuvent être utilisées mais s'avèrent être beaucoup plus lentes et plus difficiles à calibrer. Ce placement peut également souffrir d'un problème de centralisation au niveau du superviseur comme c'est typiquement le cas dans l'implémentation de GFS [Ghemawat *et al.*, 2003] dans laquelle une entité centrale est responsable de disséminer les fragments et de les superviser. Cette politique est également utilisée dans [Kubiatowicz *et al.*, 2000, Zhang *et al.*, 2007] au dessus d'une DHT.

Politique de placement local

Dans la politique de placement local (cf. Fig. 2.10(b)), les répliques d'une donnée sont stockées sur les k successeurs d'un pair qui peuvent participer au processus de réparation. Le superviseur est désigné comme étant l'élément de la chaîne ayant l'identifiant le plus faible. Comme pour le placement global, lorsqu'une réparation est déclenchée, le superviseur courant récupère d fragments toujours disponibles et recrée de la redondance. Cette politique permet une réparation ainsi qu'une supervision totalement décentralisée. Son principal désavantage vient du phénomène mis en évidence à la Fig. 2.11. Dans cet exemple, deux données g et f sont codées en 4 fragments chacune et sont placées selon une politique locale. Lorsque le pair 1 hébergeant les fragments f_2 et g_3 subit une faute et que la phase de réparation est déclenchée, alors, cette réparation implique les pairs 0 et 2 dans deux réparations (la réparation de f_2 et celle de g_3). Plus généralement, avec le placement local, lorsqu'une réparation a lieu, elle implique les m pairs qui sont de chaque côté du pair indisponible. Ce phénomène peut engendrer un goulot d'étranglement en saturant la bande passante de la zone sur laquelle le chevauchement a lieu, augmentant, en conséquence, le délai de réparation. Cette augmentation du délai augmente à son tour la probabilité de perdre la donnée par rapport à un placement global (dans le cas où la donnée subit trop de fautes avant que la réparation ne soit achevée). Ce choix de placement est fait par [Dabek *et al.*, 2001, Druschel et Rowstron, 2001, Adya *et al.*, 2002, Bhagwan *et al.*, 2004, Haeberlen *et al.*, 2005].

À l'issue de leur étude sur l'impact des stratégies de placement de répliques sur la disponibilité des données dans les réseaux pair-à-pair, [Giroire *et al.*, 2009a] concluent notamment que :

- le temps de réparation d'une donnée avec la politique de placement local est beaucoup plus long qu'avec une politique de placement global (à cause du goulot d'étranglement) ;
- pour différentes valeurs de bande passante, la probabilité de perdre une donnée avec la politique de placement local est supérieure à la probabilité de perdre une donnée avec la



(a) Placement global. (b) Placement local.

FIGURE 2.10 – Différences entre une politique de placement local et une politique de placement global.

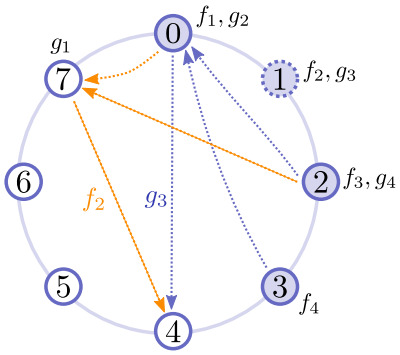


FIGURE 2.11 – Réparation dans un placement local à 4 répliques. Mise en évidence du phénomène de goulot d'étranglement.

politique de placement global.

2.4 Plateformes de stockage persistant

Après avoir présenté les mécanismes de redondance, de réparation et de placement, nous présentons dans cette section un ensemble de plateformes de stockage persistant en pair-à-pair parmi les plus illustres en nous efforçant de mettre en évidence les choix de conception qui ont été retenus pour chacune d'entre elles. Nous proposons également à l'issue de cette section, un tableau récapitulatif (cf. Tab. 2.3) de ces différentes plateformes.

2.4.1 Oceanstore/POND

Le projet Oceanstore [Kubiatowicz *et al.*, 2000] est l'un des premiers travaux décrivant une architecture complète de stockage persistant à l'échelle d'internet [9]. Les travaux fondateurs ne

9. On note qu'avant les années 2000, il existe déjà dans la littérature des plateformes de stockage distribuées mais qui sont destinées à une échelle plus restreinte. La plateforme MAFTIA [Deswarte *et al.*, 1991] décrit

parlent pas explicitement de pair-à-pair, cependant l'architecture repose sur un grand nombre de serveurs hétérogènes et non fiables et peut donc être transposée à un cas de pair-à-pair. Cette architecture a été implémentée par le prototype POND [Rhea *et al.*, 2003] et a été conçue pour fonctionner au-dessus de Tapestry (cf. Sec. 1.5.3). Dans POND, les données possèdent un identifiant unique (GUID) et sont non modifiables une fois qu'elles ont été insérées dans le réseau. Cependant, un fichier peut posséder plusieurs versions successives. Ces versions sont chaînées de la plus ancienne à la plus récente et cette chaîne est identifiée par un AGUID. Il est donc possible grâce à ce chaînage d'insérer ou de supprimer des versions d'un fichier. Un système d'ACL est d'ailleurs mis en place pour gérer les permissions sur les données. Un fichier est donc identifié par un AGUID qui contient le GUID de la version la plus récente (la tête de la chaîne) et ces méta-fichiers sont répliqués. Dans Oceanstore, les données sont coupées en blocs qui sont identifiés par un BGUID. Chaque bloc est affecté au pair ayant l'identifiant le plus proche de son BGUID dans Tapestry. Ces blocs sont ensuite encodés à l'aide d'un (16,16)-code de Reed-Solomon (cf. Sec. 2.2.3) et les fragments générés sont disséminés dans le réseau en suivant une politique de placement global. Le pair responsable d'un bloc est le superviseur *de facto* des fragments de ce bloc. Les fragments sont supervisés de manière active périodiquement et sont réparés dès qu'ils sont perdus. L'ensemble de ces fragments constitue une archive du bloc. Comme le décodage des codes d'effacement est un processus coûteux, des répliques de chaque bloc peuvent également être insérées dans le réseau. Lorsqu'un pair souhaite récupérer une donnée, il cherche en premier lieu l'existence d'une réplique et la récupère si elle existe. Dans le cas contraire, il récupère 16 fragments, reconstruit le bloc et publie le fait qu'il possède une réplique de ce bloc. Dans Oceanstore, seuls les fragments issus du code d'effacement sont supervisés et réparés. Les répliques sont quand à elles construites à la demande (i.e., si un fichier n'a pas été consulté depuis longtemps, il y a de fortes chances que seulement sa version archivée soit toujours disponible) et peuvent également être supprimées si la place disponible sur un pair devient critique (POND utilise une politique de cache LRU). Le système est doté d'un mécanisme d'introspection, une boucle observation–optimisation, qui tourne en arrière-plan et qui permet notamment au système d'ajuster le nombre des répliques disponibles en fonction de leur utilisation.

2.4.2 PAST

PAST [Druschel et Rowstron, 2001, Rowstron et Druschel, 2001b] est une architecture de stockage persistant en pair-à-pair à l'échelle d'internet reposant sur la DHT Pastry [Rowstron et Druschel, 2001a]. Dans PAST, les utilisateurs et les pairs possèdent un trousseau de clés asymétriques leur servant à se forger un identifiant logique sur 160 bits et à émettre des certificats. Lorsqu'une donnée doit être insérée dans PAST, son identifiant *fileID*, est d'abord créé à partir de son nom de fichier et de la clé publique de son propriétaire. Le propriétaire émet également un

notamment une architecture distribuée de stockage robuste et sécurisé basée sur de la réplication de blocs dans les serveurs privés d'une entreprise et dans laquelle toutes les décisions sont prises suite à un vote à la majorité entre les serveurs.

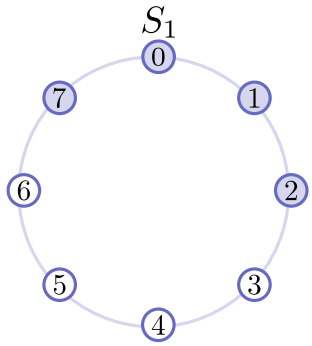
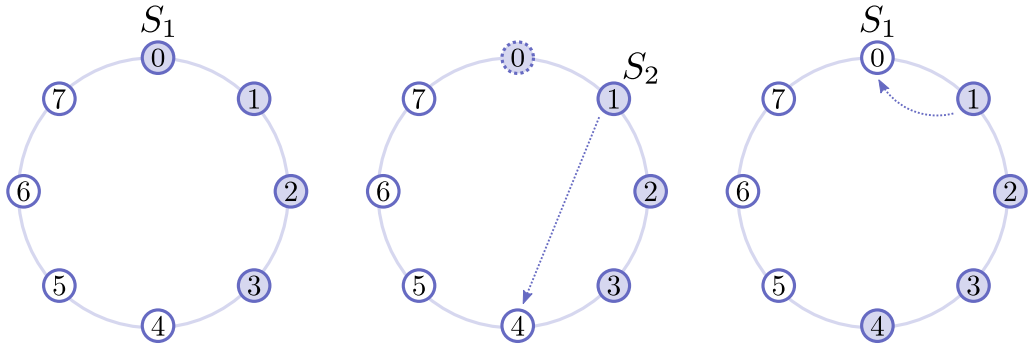


FIGURE 2.12 – Placement et supervision dans PAST. Exemple avec un bloc d'identifiant 0 ayant un facteur de réplication $k = 4$. Les répliques sont placées sur les k plus proches voisins du superviseur.

certificat signé contenant la signature du document ainsi que des métadonnées. Ce certificat lui permettra de s'authentifier lorsqu'il voudra récupérer cette donnée. La donnée et son certificat sont ensuite répliqués k fois puis insérés dans Pastry en suivant une politique de placement local qui consiste à placer les k répliques sur les k plus proches voisins du pair ayant l'identifiant le plus proche du fileID dans la DHT. Les données insérées dans PAST sont non modifiables et ne peuvent pas être supprimées. Néanmoins, le propriétaire d'une donnée dont l'authentification est valide peut réclamer l'espace de stockage associé à cette donnée (i.e., elle ne sera plus supervisée ni réparée). Le superviseur d'une donnée est le pair qui a l'identifiant le plus proche du fileID de cette donnée sur la DHT. Comme les répliques sont stockées sur les k plus proches pairs du superviseur, ce dernier doit superviser ses $k/2$ prédécesseurs et ses $k/2$ successeurs (cf. Fig. 2.12). Dés qu'il détecte l'indisponibilité d'un de ces pairs, il restaure la redondance et la politique de réparation est donc immédiate. Les données peuvent également être chiffrées par leur propriétaire pour plus de sécurité et PAST fait usage de cartes à puce émises par l'entité fournissant le service pour, d'une part, stocker les trousseaux de clés et générer les certificats, et, d'autre part, gérer les éventuels quotas affectés aux utilisateurs.

2.4.3 CFS

CFS (*Collaborative Filesystem*) [Dabek *et al.*, 2001] est une architecture de stockage persistant en pair-à-pair destinée à être déployée sur internet et reposant sur la DHT Chord (cf. Sec. 1.5.1). Cette approche est assez similaire à PAST mais fournit en plus un véritable système de fichiers arborescent qui peut être monté par les pairs. CFS fait une distinction entre les pairs pouvant modifier le système de fichiers (les *publishers*) et les autres pairs qui n'ont qu'un accès en lecture. Les données sont découpées en blocs, identifiés par leur haché, et ces blocs sont répliqués sur k pairs. Ce choix de conception est motivé par le fait que le stockage de petits blocs permet un meilleur équilibrage de charge que la réplication d'un fichier entier. La gestion de l'insertion, de la recherche, du placement et de la supervision de ces répliques est effectuée par DHash, une couche applicative se trouvant entre Chord et le système de fichiers. DHash insère les k répliques



(a) Placement initial sur les 4 successeurs du pair 0.

(b) La perte du superviseur S_1 entraîne son remplacement par S_2 et par la copie d'une réplique sur le pair 4.

(c) Le retour de S_1 avec ses disques vides déclenche également une réparation.

FIGURE 2.13 – Placement et supervision dans CFS. Exemple avec un bloc d'identifiant 0 ayant un facteur de réplication $k = 4$.

d'un bloc en suivant une politique de placement local sur les k successeurs du pair responsable de ce bloc dans la DHT. La supervision de ces répliques est effectuée de manière décentralisée par le pair responsable du bloc qui s'assure périodiquement que ses $k-1$ successeurs sont toujours disponibles (cf. Fig. 2.13(a)). La politique de réparation mise en place dans DHash est la même que dans PAST, à savoir immédiate. C'est-à-dire que dès qu'une réplique est perdue, elle est immédiatement réparée. Si le superviseur devient indisponible à la suite d'une faute, les propriétés de Chord font que c'est son successeur direct dans l'anneau qui devient le nouveau superviseur du bloc et il insère une nouvelle réplique sur son $k-1^e$ successeur (cf. Fig. 2.13(b)). Dans le cas inverse, où un pair se connecte et devient le nouveau responsable d'un bloc, il crée une nouvelle réplique dans son espace de stockage à partir de son successeur (cf. Fig. 2.13(c)). Pour éviter de surcharger les pairs qui hébergent des données populaires, CFS est muni d'un système de caches de répliques (soumis à la politique LRU) assez similaire à celui utilisé pour la réplication paresseuse dans Freenet (cf. Sec. 1.4.2). C'est-à-dire que les blocs sont copiés de manière transparente sur les pairs se trouvant sur la remontée du résultat d'une requête de recherche.

2.4.4 Farsite

Farsite [Adya *et al.*, 2002] est une architecture issue de la recherche privée qui vise à créer un système de fichiers réseau qui mutualise les ressources inutilisées des machines de bureau d'une entreprise ou d'une université. L'environnement d'exécution de l'application étant restreint, un certain nombre d'hypothèses sont posées : le nombre de pairs est de l'ordre de 10^5 et ces machines sont interconnectées par un réseau de faible latence qui possède une large bande passante. L'idée est de fournir aux employés de l'entreprise un montage réseau sécurisé, en apparence centralisé, mais qui, en réalité, est totalement décentralisé. Les permissions et l'authentification

sont gérées à l'aide de certificats qui sont émis par une entité administrative de l'entreprise. Un certificat est émis pour chaque pair, chaque utilisateur et chaque espace de nommage du système. Les documents sont systématiquement chiffrés avec une clé symétrique générée par leur propriétaire respectif. Pour autoriser d'autres utilisateurs $u_1, \ldots, u_n$ à accéder à la donnée, cette clé symétrique est transmise à chaque u_i dans une forme chiffrée avec la clé publique de u_i. Le système de fichier est organisé en espaces de nommages disjoints. Chaque espace de nommage est un arbre distribué dont les nœuds sont des répertoires qui sont administrés par un ensemble de pairs que l'on appelle groupe du répertoire. Le rôle du groupe d'un répertoire est de stocker les entrées de ce répertoire, qui sont répliquées sur chacun de ses membres, et de gérer, par des accords Byzantins [Castro et Liskov, 1999], les permissions de toutes les requêtes d'utilisateurs à destination de ce répertoire. Les membres du groupe d'un répertoire authentifient notamment les utilisateurs et vérifient que les données retournées par les pairs sont intègres (à l'aide du haché de la donnée par exemple). Les données sont répliquées aléatoirement sur des pairs de stockage et sont supervisées par le groupe du répertoire qui les contient. Dans Farsite, la réparation des répliques est faite immédiatement lorsqu'une faute est détectée. On note également que les pairs peuvent occuper à la fois les différents rôles (stockage, client et membre du groupe d'un répertoire).

2.4.5 Total Recall

Les auteurs de Total Recall [Bhagwan *et al.*, 2004] sont les premiers à porter un véritable regard sur le paramétrage des systèmes de stockage persistant en pair-à-pair large échelle. Ils constatent, au moment de leur publication, que dans les plateformes existantes le paramétrage du système est soit fixé arbitrairement dans la spécification, soit laissé à la discrétion de son administrateur. Or, le niveau de sûreté attendu est entièrement dépendant de la disponibilité des pairs qui varie au cours du temps. Face à ce constat, Bagwan *et al.*, proposent une approche différente dans laquelle le système s'observe afin d'adapter ses politiques en fonction de la situation actuelle dans laquelle il se trouve. Dans Total Recall, les données peuvent être répliquées ou codées avec un code d'effacement et peuvent être réparées de manière immédiate ou différée. Le nombre de répliques et la politique de réparation sont calculés en fonction de la situation observée. Plus concrètement, à partir de l'observation de ses voisins, chaque pair va maintenir deux modèles de disponibilité :

1. un modèle à court terme qui calcule le minimum de disponibilité obtenu sur les dernières 24h (par exemple : le minimum de pairs disponibles a été observé à 4h avec 40% de pairs connectés) ;
2. un modèle à long terme qui calcule le nombre de fautes permanentes observées sur plusieurs semaines.

Le modèle à court terme permet d'ajuster le nombre de répliques/fragments disponibles dans le système en réponse à des fautes temporaires. Cette adaptation est réalisée à l'aide d'une fonction qui retourne le nombre de répliques/fragments à introduire en fonction du niveau actuel de fautes

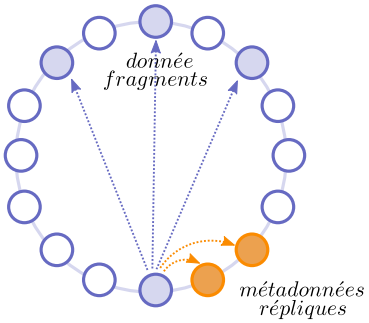


FIGURE 2.14 – Stratégies de supervision hybride dans TotalRecall. Une donnée est fragmentée et suit un placement global alors que ses métadonnées sont répliquées et suivent un placement local.

pour un niveau de disponibilité fixé lors de la création du système (cf. Eq. 2.22 et Eq. 2.23 que nous décrivons plus loin). Le modèle à long terme, quand à lui, sert à définir le seuil de réparation en fonction du nombre de fautes permanentes. Le choix de la politique de redondance a été étudié à l'aide de simulations et dépend de la taille de la donnée. Cette étude conclut que les données de grande taille doivent être codées et réparées avec une politique à seuil alors que les petits fichiers doivent être répliqués et réparés avec une politique immédiate. À partir de cette constatation, les données insérées par les utilisateurs dans Total Recall sont systématiquement codées à l'aide d'un code d'effacement et disséminées en suivant un placement global. Par contre, les métadonnées de ces fichiers, qui contiennent notamment la liste des fragments ainsi que les pairs qui les hébergent, sont répliquées et insérées sur les successeurs du superviseur de la donnée en suivant un placement local (cf. Fig. 2.14).

2.4.6 Glacier

Glacier [Haeberlen *et al.*, 2005] est une plateforme de stockage décentralisée en pair-à-pair qui utilise le placement global et les codes d'effacement pour obtenir de la disponibilité. Elle est destinée à une utilisation interne dans une entreprise dans la mesure où un cluster de serveurs sûrs est nécessaire à son efficacité. L'apport de cette plateforme par rapport à OceanStore ou Past vient du fait qu'elle a été pensée pour tolérer les fautes corrélées (cf. Def. 14) en augmentant massivement le nombre de fragments tout en conservant de bonnes performances. Cette architecture est structurée en deux parties :

- un cluster de serveurs (*primary store*) classique qui stocke des répliques de chaque bloc de données pour permettre de tolérer des fautes à court terme tout en gardant de bonnes performances ;
- un réseau pair-à-pair structuré par une DHT qui archive les fragments des blocs codés pour tolérer les fautes à long terme et reconstruire des répliques et des fragments.

Les opérations d'insertion sont d'abord effectuées sur le cluster puis elles sont agrégées. Un agrégat est une concaténation de données qui, lorsqu'elle a atteint une taille suffisamment im-

portante, est archivée par un code d'effacement dans le réseau. Ceci permet de limiter le nombre de fichiers dans le réseau pair-à-pair. Chaque donnée est associée à un bail qui doit être renouvelé par son propriétaire et un mécanisme de ramasse-miettes s'exécute périodiquement pour supprimer les blocs de données dont le bail a expiré. La politique de réparation est immédiate mais Glacier impose néanmoins un nombre maximum R_{max} de réparations simultanées afin d'éviter de saturer la bande passante du système pour pouvoir continuer à récupérer et à insérer des données pendant une phase de réparation. Lorsque le système détecte un nombre de fautes corrélées supérieur à un certain seuil critique, il passe dans un mode de survie (*large-scale failure*) dans lequel seules les communications de supervision et de réparation sont tolérées. Lorsque la source de la corrélation a été détectée et que les pairs vulnérables ont été mis à jour ou retirés du système par l'administrateur, le système passe dans un mode de réparation (*recovery mode*) dans lequel les fragments sont reconstruits en totalité ainsi que la *primary store* et les agrégats. Lorsque le système est réparé, il reprend son mode normal dans lequel les données peuvent être à nouveau consultées et insérées.

2.4.7 BitVault

BitVault [Zhang *et al.*, 2007] est une plateforme de stockage persistant destinée au monde de l'entreprise qui utilise la technologie pair-à-pair. Elle est principalement destinée à l'archivage de données administratives telles que les emails, les factures, la documentation, etc. c'est-à-dire à un grand nombre de documents de petite taille. Par conséquent, cette architecture accorde plus d'importance à la vitesse de réparation qu'au coût de stockage et fait donc le choix de la réplication comme mécanisme de redondance. L'architecture est supportée par une DHT de type Chord (cf. Sec. 1.5.1) et le placement des répliques est global. Le réseau physique étant celui d'une entreprise, on s'attend à une faible latence associée à une large bande passante. C'est pourquoi, BitVault fixe le délai entre deux supervisions de l'ordre de la minute, lui offrant une forte réactivité face aux fautes. Les réparations sont faites dès qu'une faute est détectée.

2.4.8 Wuala

Wuala [Grolimund, 2007, Wuala, 2012] est une plateforme propriétaire de stockage persistant en pair-à-pair déployée sur internet. Elle est issue des travaux de Dominik Grolimund et de Luzius Meisser menés à l'école polytechnique fédérale de Zurich puis commercialisée par la société LaCie Inc. L'architecture réseau est découpée en deux parties distinctes : un cluster de serveurs appartenant à LaCie et un réseau pair-à-pair décentralisé structuré constitué des utilisateurs du service (ceci n'est pas sans rappeler l'architecture de Glacier). Dans Wuala, les données à insérer sont chiffrées avec le standard AES 128 bits [AES, 2001] puis fragmentées avec un (m,n)-code d'effacement de type Reed-Solomon. Les $m + n$ fragments sont ensuite insérés dans le réseau pair-à-pair de manière aléatoire et une copie des m fragments initiaux est également insérée dans le cluster de serveurs privés. Le placement des fragments est global et c'est le pair qui a inséré les données qui en est le superviseur. S'il n'existe pas assez de fragments dans le réseau

Plateforme	Redondance	Supervision	Réparation
BitVault	réplication	global	immédiate
CFS	réplication	local	immédiate
Farsite	réplication	global	immédiate
Glacier	codes d'effacement	local	immédiate
Oceanstore	hybride	global	immédiate
Past	réplication	local	immédiate
Total Recall	hybride	hybride	hybride
Wuala	codes d'effacement	global	–

TABLE 2.3 – Choix de conception dans des plateformes célèbres de stockage persistant en pair-à-pair.

pair-à-pair, alors le superviseur peut interroger les serveurs de Wuala pour obtenir les fragments manquants et *vice versa*.

Afin d'éviter les comportements égoïstes (*free-riding*), Wuala est muni du système de réputation distribué Havelaar [Grolimund *et al.*, 2006]. Dans ce système, chaque pair envoie périodiquement à d'autres pairs un rapport sur le comportement des pairs avec lesquels il a communiqué (bande passante et espace de stockage mis à disposition). À partir de ces observations, chaque pair calcule localement une réputation pour les autres pairs et ajuste la quantité de ressources qu'il leur partage en conséquence. Dans des versions antérieures, les utilisateurs avaient la possibilité d'échanger de la disponibilité et de la bande passante contre de l'espace de stockage (i.e., plus un pair était connecté longtemps avec un bon débit et plus il était récompensé par de l'espace de stockage supplémentaire). Dans la version actuelle de l'application, la composante pair-à-pair a été supprimée et sa fonction a été remplacée par le *Wuala Cloud*, une interconnexion de serveurs de la société disséminés dans des data centers européens. L'espace alloué par Wuala n'est, de fait, plus variable et un compte gratuit offre 5Go d'espace de stockage.

2.5 Modèles de disponibilité et modèles de fautes

Les mécanismes de redondance, de supervision et de réparation permettent d'obtenir de la tolérance aux fautes et de la durabilité dans les systèmes de stockage en pair-à-pair. Leur paramétrage a une influence directe sur la disponibilité des données. C'est pourquoi, non seulement ce paramétrage est loin d'être une tâche triviale, mais il doit être effectué avec précision. D'une manière générale, si les fautes sont indépendantes, il est possible de déduire les paramètres de redondance pour un niveau de disponibilité voulu à condition que le modèle de fautes des pairs soit connu. L'élaboration de ce modèle de fautes peut être déduit de différentes traces de réseaux bien connus [Godfrey, 2006], il peut également être déduit à partir des résultats d'une version beta de l'application ou bien être adaptatif par une boucle de rétro-action sur le système [Bhagwan, 2004]. Cependant, l'hypothèse d'indépendance des fautes est restrictive et certains pairs

partagent des traits communs les faisant co-varier [Weatherspoon *et al.*, 2002]. Par exemple, les pairs peuvent être situés dans le même bâtiment, ils peuvent partager les mêmes routeurs ou présenter les mêmes vulnérabilités (même systèmes d'exploitation, même services réseau). Sous hypothèse d'indépendance, héberger plusieurs répliques d'un même document sur des pairs effectivement corrélés expose le document à la perte simultanée de plusieurs répliques alors que son modèle de disponibilité ne le prend pas en compte. La disponibilité espérée s'en trouve alors faussée, pouvant conduire dans le pire des cas à la perte de la donnée. Mais face à la difficulté de modéliser ces fautes corrélées, la plupart des applications font l'hypothèse d'indépendance. Nous présentons ces deux approches dans la suite de la section.

2.5.1 Indépendance des fautes

Si l'on fait l'hypothèse que les fautes apparaissent dans le système de manière indépendante, alors il est possible de déduire les paramètres de réplication et de fragmentation à appliquer à une donnée pour un niveau de disponibilité requis [Bhagwan *et al.*, 2002, Bhagwan, 2004]. La disponibilité d'un pair est mesurée sur une période d'observation plus ou moins longue qui peut correspondre à la période entre deux supervisions. On note P_{disp} la probabilité qu'un pair soit disponible sur la fenêtre d'observation considérée.

Réplication

Lorsque la politique de redondance fait usage de la réplication, alors la disponibilité A_{rep} d'une donnée (cf. Def. 15) sur une certaine fenêtre d'observation est donnée par :

$$A_{rep} = 1 - (1 - P_{disp})^{N_{rep}} \tag{2.21}$$

qui correspond à la probabilité d'avoir au moins toujours une réplique disponible à la fin de la fenêtre d'observation, étant donné que N_{rep} répliques étaient disponibles au début de cette fenêtre. On en déduit N_{rep} pour A_{rep} et P_{disp} fixés :

$$N_{rep} = \frac{\log(1 - A_{rep})}{\log(1 - P_{disp})} \tag{2.22}$$

Exemple 23 *Sous indépendance des fautes et sur une période d'observation donnée, si un pair est disponible 40% du temps, et que le niveau de disponibilité voulu par le concepteur du système est de 0.999 alors le nombre de répliques à introduire dans le système est* $N_{rep} = 14$.

Codes d'effacement

Lorsque la politique de redondance fait usage d'un (m,n)-code d'effacement, la disponibilité d'une donnée A_{eff} correspond à la probabilité qu'à la fin de la période d'observation, n pairs hébergeant des fragments soient déconnectés et m pairs soient toujours connectés ou que $n - 1$

pairs soient déconnectés et que $m+1$ soient connectés, etc. Ce qui donne lieu à la formulation suivante :

$$A_{\mathit{eff}} = P(Y \geq m) = \sum_{j=m}^{\omega m} \binom{\omega m}{j} P_{disp}^{j} (1 - P_{disp})^{\omega m - j} \tag{2.23}$$

avec $\omega = (n/m) + 1$ le facteur de redondance (cf. Def. 17) et Y une variable aléatoire comptant le nombre de pairs disponibles qui stockent des fragments de la donnée à la fin de la période d'observation. Avec cette formulation, l'extraction de ω pour une valeur de disponibilité A_{eff} fixée est un calcul non trivial. C'est pourquoi, on a recours à l'approximation normale de la loi binomiale pour trouver ω. C'est-à-dire que pour des valeurs de ωm suffisamment grandes la variable aléatoire :

$$W = \frac{Y}{\omega m} \tag{2.24}$$

représentant la fraction de pairs hébergeant des fragments toujours disponibles à la fin de la période d'observation, suit une loi normale de paramètres :

$$\mu = \mathbb{E}(W) = \frac{1}{\omega m}\mathbb{E}(Y) = P_{disp} \tag{2.25}$$

et

$$\begin{aligned}
\sigma^2 &= \mathrm{Var}(W) = \mathbb{E}\left[\left(\frac{Y}{\omega m} - \mathbb{E}\left[\frac{Y}{\omega m}\right]\right)^2\right] && (2.26)\\
&= \mathbb{E}\left[\frac{1}{\omega m^2}\Big(Y - \mathbb{E}(Y)\Big)^2\right] && (2.27)\\
&= \frac{1}{\omega m^2}\mathbb{E}\Big[\Big(Y - \mathbb{E}(Y)\Big)^2\Big] && (2.28)\\
&= \frac{1}{\omega m^2}\,\mathrm{Var}(Y) = \frac{P_{disp}(1 - P_{disp})}{\omega m} && (2.29)
\end{aligned}$$

En utilisant cette approximation on peut écrire :

$$A_{\mathit{eff}} = P(Y \geq m) = P(W \geq \frac{1}{\omega}) \tag{2.30}$$

et par définition de la densité de probabilités de la loi normale :

$$A_{\mathit{eff}} = \frac{1}{\sigma\sqrt{2\pi}} \int_{\frac{1}{c}}^{1} \mathrm{e}^{-(x - P_{disp})^2/(2\sigma^2)} \mathrm{d}x \tag{2.31}$$

La Tab. 2.4 donne les valeurs de x suivant une loi normale pour lesquelles $P(W \geq x)$ correspond au niveau de disponibilité voulu. Si x est égal à $1/\omega$ alors on peut exprimer ω en fonction du niveau de disponibilité voulu :

$$\frac{1}{\omega} = \mu - k\sigma \tag{2.32}$$

avec k la valeur lue dans la table pour le niveau de disponibilité voulu. En remplaçant μ et σ dans l'Eq. 2.32 on obtient l'équation quadratique en $\sqrt{\omega}$:

$$\begin{aligned}
&\frac{1}{\omega} = P_{disp} - k\sqrt{\frac{P_{disp}(1 - P_{disp})}{\omega m}} && (2.33)\\
\Leftrightarrow\quad &-P_{disp}\,\omega + k\sqrt{\omega}\sqrt{\frac{P_{disp}(1 - P_{disp})}{m}} + 1 = 0 && (2.34)
\end{aligned}$$

Disponibilité voulue	x
0.800	$\mu - 1.29\sigma$
0.900	$\mu - 1.65\sigma$
0.950	$\mu - 1.96\sigma$
0.990	$\mu - 2.58\sigma$
0.995	$\mu - 2.81\sigma$
0.999	$\mu - 3.30\sigma$

TABLE 2.4 – Valeurs d'une variable aléatoire suivant une loi normale pour le niveau de disponibilité voulu.

que l'on résout en éliminant les racines impossibles :

$$\omega = \left(\frac{k\sqrt{\frac{P_{disp}(1-P_{disp})}{m}} + \sqrt{\frac{i^2 P_{disp}(1-P_{disp})}{m} + 4P_{disp}}}{2P_{disp}} \right)^2 \tag{2.35}$$

Exemple 24 *Sous indépendance des fautes et sur une période d'observation donnée, si un pair est disponible 40% du temps, que le nombre de blocs initiaux est égal à* 20 *et que le niveau de disponibilité voulu par le concepteur du système est de 0.999 alors on lit* $k = 3.30$ *dans la table et on calcule le facteur de redondance* $\omega = 4.11$ *soit* $n = 63$*. Pour le niveau de disponibilité requis et le taux de fautes observé, les données doivent donc être codées avec un* (20,63)*-code d'effacement.*

2.5.2 Fautes corrélées

L'hypothèse d'indépendance permet de calculer les paramètres de redondance adéquats par rapport aux propriétés du système. Cependant en pratique, nous l'avons évoqué plus haut, les fautes sont corrélées dans les réseaux venant biaiser les résultats de l'analyse ci-dessus. Une analyse [Kondo *et al.*, 2008] des traces de BOINC [10] sur une année a permis d'établir plusieurs statistiques sur la disponibilité des pairs. On s'aperçoit notamment que le nombre de connexions et déconnexions est dépendant de l'heure de la journée et du jour de la semaine, mais également que ces motifs ne sont pas les mêmes dans des zones géographiques différentes. Une étude [Bhagwan *et al.*, 2003] précédente du réseau Overnet sur 7 jours avait déjà montré que la période de la journée et de la semaine avaient un impact sur le flux des usagers. Néanmoins, dans cet article, les auteurs préfèrent évoquer des tendances dans les comportements que des corrélations à proprement parler. Partant de ce constat, deux grandes idées sont proposées dans la littérature pour prendre en compte ce problème des corrélations : l'approche introspective et l'approche analytique.

10. Berkeley Open Infrastructure for Network Computing.

Approche introspective

La première consiste à décrire le système finement pour effectuer un clustering de machines similaires. Une fois qu'un ensemble de clusters est calculé, la répartition des répliques (ou fragments) d'un même document est faite sur des clusters distincts. [Weatherspoon *et al.*, 2002] proposent un algorithme de clustering. La première étape de leur algorithme consiste à construire un modèle du réseau (*Model builder*) en découvrant des ensembles de noeuds tombant en panne avec une faible corrélation. Ce modèle est construit à partir d'informations renseignées par les administrateurs et par l'observation de l'évolution des machines dans le temps (évolution de l'*uptime* notamment). Chaque pair est alors représenté sous la forme d'un vecteur d'attributs. Une fois le modèle construit (i.e., un ensemble de clusters de machines qui sont faiblement corrélées), chaque fragment est disséminé par un serveur central (Disseminator) dans un cluster de l'ensemble. Dans [Kondo *et al.*, 2008], le clustering est calculé *a priori* avec l'algorithme k-means à partir des traces collectées. Dans l'approche par introspection, le clustering peut-être relancé périodiquement pour prendre en compte les changements dans le réseau.

Approche analytique

La seconde idée part du principe qu'un clustering exhaustif réaliste est trop difficile à obtenir et qu'il faut plutôt apporter de nouveaux modèles et moduler en adéquation avec ces modèles la redondance dans le réseau. [Bakkaloglu *et al.*, 2002] proposent deux distributions de probabilités permettant de modéliser la disponibilité d'une donnée en présence de fautes corrélées. Ces distributions sont dépendantes des mêmes paramètres que la distribution classique de l'Eq. 2.23 et d'un paramètre supplémentaire représentant le niveau de corrélation dans le réseau. Dans un premier modèle basé sur des probabilités conditionnelles, le niveau de corrélation est calculé par la probabilité qu'un pair soit indisponible sachant que $k-1$ autres pairs sont également indisponibles. Pour $k=1$ on a $P_1 = 1 - P_{disp}$. Pour $k=2$ cette probabilité s'exprime comme suit :

$$P_2 = \sum_{(X,Y)\in n\times n} P(X=\overline{D}|Y=\overline{D})/|n\times n| \tag{2.36}$$

avec n l'ensemble des pairs du réseau et $P(X=\overline{D})$ la probabilité que le pair X soit indisponible. Puis, pour tout k, les auteurs donnent une expression récursive de P_k :

$$P_k = P_{k-1} + \frac{P_{k-1}+P_{k-2}}{2} \tag{2.37}$$

Le second modèle proposé dans [Bakkaloglu *et al.*, 2002] est une distribution bêta-binomiale qui part de la formulation classique et qui y introduit une fonction d'intensité f_p qui donne la distribution de probabilités qu'une fraction p des n pairs soit indisponible. Les résultats d'expériences montrent que les deux distributions font une erreur beaucoup moins importante que la distribution classique sur les jeux de données analysés et que, parmi ces deux distributions, c'est la distribution conditionnelle qui fait l'erreur la plus faible.

Suivant le même principe, [Nath *et al.*, 2006] proposent un modèle de corrélation de fautes paramétrable et inspiré des tendances observées sur plusieurs traces de différents systèmes pair-à-pair. La particularité de leur étude vient du fait qu'ils énoncent également quatre principes de conception qui sont sensés permettre aux systèmes de mieux tolérer les fautes corrélées en pratique :

1. les fautes corrélées de grande taille (par exemple un déni de service distribué) ont un effet dominant sur la disponibilité des systèmes et des mécanismes devraient être mis en place pour réagir en conséquence ;
2. faire l'hypothèse que la taille des corrélations est identique dans le système, et, donc, modéliser une faute comme la perte d'un nombre de machines fixé, est une approximation pouvant conduire à des inexactitudes dramatiques en pratique. Par conséquent, les fautes corrélées devraient plutôt être modélisées par une distribution ;
3. dans les systèmes de redondance basés sur des codes d'effacement, en présence de fautes corrélées, l'augmentation de n n'apporte qu'un gain marginal en terme de disponibilité. Sur leurs traces, tripler la valeur de n n'apporte même pas un *neuf* supplémentaire de disponibilité ;
4. une disponibilité plus élevée en présence de fautes indépendantes ne permet absolument pas, une fois transposée dans un cas de fautes corrélées, d'avoir une disponibilité plus élevée ; en particulier, les corrélations ont un impact plus fort sur des systèmes ayant un m élevé que sur ceux ayant un m faible. Par conséquent, la robustesse du système devrait être systématiquement mise à l'épreuve sur des modèles de fautes corrélées.

2.6 Conclusion

Dans ce second chapitre d'état de l'art, nous avons abordé la problématique de la disponibilité des données dans les applications de stockage décentralisé en pair-à-pair. Cette disponibilité est obtenue par la mise en place de trois mécanismes : redondance, supervision et réparation. Un schéma de redondance est d'abord appliqué sur les données avant de les insérer dans le système pour leur permettre de tolérer ponctuellement un certain nombre de fautes. Ensuite, ce niveau de redondance est préservé au cours du temps à l'aide de supervisions et de réparations périodiques pour s'assurer de la persistance des données. La construction et le calibrage de ces trois politiques sont dépendants de plusieurs paramètres tels que le niveau de disponibilité attendu, le taux de fautes observé ou encore les ressources dont disposent les pairs. La plupart des plateformes de stockage décentralisé ont recours à des modèles de disponibilité pour choisir ces paramètres *a priori*. Un tel choix impose une nouvelle étude de paramètres et une mise à jour de l'infrastructure à chaque fois que la dynamique du réseau pair-à-pair vient à changer. *a contrario*, d'autres approches, comme TotalRecall, partent du principe qu'un réseau pair-à-pair est en perpétuelle évolution et qu'il est indispensable de réévaluer les paramètres fonctionnels du système pendant son exécution pour s'assurer que le niveau de disponibilité attendu est bien

préservé au cours du temps. Cette seconde approche, que nous pouvons caractériser de « réactive-autonome », nous semble être la plus adaptée à la dynamique des réseaux pair-à-pair. Ce besoin de réactivité et d'autonomie des applications en milieu pair-à-pair est l'une des motivations à la base de nos travaux sur la mobilité et la flexibilité des données. Cette motivation nous a conduit à utiliser des approches autonomes issues de l'intelligence artificielle comme notamment une modélisation sous la forme de systèmes multi-agents.

Chapitre 3

Agents mobiles, systèmes multi-agents et applications

Sommaire

3.1 Intelligence artificielle, agent et systèmes multi-agents

Nos travaux sur la mobilité des données dans les architectures de stockage décentralisé sont à l'intersection des réseaux pair-à-pair et de l'intelligence artificielle distribuée. Nous nous sommes intéressés dans les deux chapitres précédents aux spécificités des architectures pair-à-pair ainsi qu'aux mécanismes indispensables qui permettent la construction de systèmes de stockage décentralisés, robustes et durables. L'objet du présent chapitre est multiple. Après une brève

introduction de l'intelligence artificielle, nous présentons les concepts d'agent et de systèmes multi-agents qui sont centraux en intelligence artificielle. Puis, dans un second temps, nous axons notre exposé sur la migration de code dans les architectures distribuées et, notamment, sur la technologie des agents mobiles qui est le paradigme de programmation répartie dont nous faisons usage dans nos travaux.

3.1.1 Intelligence artificielle

L'intelligence artificielle (IA) est un domaine vaste qui regroupe plusieurs sous-domaines tels que la reconnaissance de formes, la représentation des connaissances, l'apprentissage, la robotique, la vision, la théorie des jeux, le raisonnement logique, la planification, la vie artificielle, la simulation multi-agents, etc. Il n'y a pas de définition universelle de l'IA. Historiquement ce domaine, né peu de temps après l'apparition des premiers ordinateurs, a d'abord cherché à concevoir des machines aussi intelligentes que l'homme. Après un certain enthousiasme initial, la communauté s'est retrouvée confrontée à des verrous scientifiques et a évolué vers un nouveau but plus réalisable : la conception « d'entités intelligentes » où l'intelligence est étroitement liée au concept de rationalité. On trouve dans la littérature différentes définitions de l'IA qui ont évolué au cours du temps [Russell et Norvig, 2010] :

concevoir des systèmes qui pensent comme des humains

Dans cette approche issue des sciences cognitives, on cherche à rendre automatique les activités associées à la pensée humaine :

> « [...] l'automatisation des activités associées à la pensée humaine telles que la prise de décision, la résolution de problèmes, l'apprentissage [...] » –[Bellman, 1978]

et à les comparer avec des modèles qui sont issus de recherches en psychologie cognitive. Cette approche ne s'intéresse pas tant à la performance des résultats obtenus qu'à la manière dont ils sont obtenus puisque ce domaine cherche à transposer de la manière la plus fidèle possible les processus mentaux d'un humain en programmes informatiques.

concevoir des systèmes qui pensent rationnellement

Dans cette approche, l'intelligence artificielle est :

> « L'étude des facultés mentales à travers l'utilisation de modèles de calcul. » – [Charniak et McDermott, 1985]

Elle découle directement des travaux en logique et en raisonnement logique dans laquelle on cherche à résoudre des problèmes à l'aide de formalismes de logique qui permettent de faire un certain nombre de déductions à partir d'un ensemble de faits. Les prises de décision sont donc l'aboutissement d'un processus de raisonnement.

concevoir des systèmes qui agissent aussi bien que des humains

> « L'étude des mécanismes permettant à un ordinateur de réaliser des tâches qui sont, pour le moment, mieux exécutées par l'humain.» –[Rich et Knight, 1991]

Alan Turing a proposé un test permettant de déterminer si un système peut être caractérisé d'intelligent (au sens de l'intelligence humaine). Son principe consiste à confronter le système à un expérimentateur humain qui, à l'aide d'un certain nombre de questions et de challenges, doit déterminer s'il a à faire à un humain ou à une machine. La difficulté à laquelle cette approche de l'IA se confronte vient notamment du fait que l'humain a un domaine de raisonnement ainsi qu'une possibilité d'imagination sans limites. C'est pourquoi, la recherche s'est intéressée à l'étude des principes sous-jacents à l'intelligence, plutôt qu'à concevoir des systèmes passant le test de Turing en imitant l'humain sur un ensemble de cas exhaustifs, qui semblent de toute façon voués à l'échec.

concevoir des systèmes qui agissent rationnellement

> « L'Intelligence Artificielle est l'étude et la conception d'agents intelligents. » –[Poole *et al.*, 1998]

C'est cette définition qui caractérise le mieux l'intelligence artificielle contemporaine. Un comportement rationnel consiste à prendre des décisions en adéquation avec un ensemble de buts. Cette notion de rationalité est utilisée dans les approches récentes pour caractériser l'intelligence d'un système et semble plus réaliste que la précédente basée sur le test de Turing. Derrière cette notion de rationalité se cache non seulement le raisonnement mais également une notion d'utilité. Plus une action est utile (i.e., elle sert les buts du système) et plus elle devrait être effectuée. L'intérêt d'une telle approche vient du fait que si l'on dispose d'une fonction d'évaluation de l'utilité, alors, on est en mesure de déterminer si le comportement artificiel résultant est optimal par rapport à ce critère.

C'est au cours des années 1990 que la communauté s'accorde sur le concept d'agent. Une entité autonome intelligente (que nous détaillons par la suite) dont le degré d'intelligence se mesure en fonction de sa capacité à être rationnelle. Ce paradigme agent a notamment permis d'accorder la communauté sur un ensemble de concepts régissant la construction d'intelligences artificielles. Cependant, cette vision mono-agent dans laquelle une IA est une entité rationnelle qui raisonne pour accomplir ses buts est incomplète puisqu'elle occulte tout un pan de l'intelligence artificielle : les systèmes multi-agents (SMA). Ce domaine, historiquement issu de l'intelligence artificielle distribuée (IAD) et inspiré par l'intelligence collective, met en interaction un ensemble d'agents autonomes qui peuvent chercher à atteindre collectivement un but. Les SMA ont notamment été introduits à partir d'une volonté de formaliser les interactions, les échanges et l'organisation de systèmes distribués basés sur des entités autonomes. L'approche *Voyelles* [Demazeau, 1995, Ricordel, 2001] propose notamment d'analyser les systèmes selon quatre points de vue : **A**gents, **E**nvironnement, **I**nteractions et **O**rganisation qui forment les briques de base du système.

3.1.2 Le concept d'agent

Le concept d'agent est assez vaste lui aussi et plusieurs définitions en ont été données [Wooldridge et Jennings, 1995, Ferber, 1995, Briot et Demazeau, 2001, Franklin et Graesser, 1996, Beynier, 2006, Russell et Norvig, 2010, Canu, 2011]. Cependant, la communauté s'accorde sur une définition basique d'un agent, qui peut être ensuite raffinée en fonction des besoins. Un agent est une entité autonome, située dans un environnement, qui perçoit cet environnement à l'aide d'un ensemble de capteurs, qui agit sur cet environnement au moyen d'un ensemble d'effecteurs et qui peut chercher à atteindre des buts.

Autonomie

La notion d'autonomie est centrale dans le concept d'agent et matérialise le passage d'un paradigme de programmation dans lequel c'est l'humain qui appelle les procédures et déclenche des effets sur des objets passifs, à une conception dans laquelle les objets sont actifs et où un cycle à trois temps « perception–décision–action » s'exécute perpétuellement. Les agents sont donc des entités qui agissent et dont le comportement est intimement lié aux perceptions. Pour Jacques Tisseau, tout modèle censé représenter un être vivant (au sens « agir à la manière de ...») doit être doté d'une telle interface sensorimotrice :

> « L'autonomisation d'un modèle consiste à le doter de moyens de perception et d'action au sein de son environnement, ainsi que d'un module de décision lui permettant d'adapter ses réactions aux stimuli tant externes qu'internes [...] La notion d'animat, par exemple, concerne les animaux artificiels dont les lois de fonctionnement s'inspirent de celles des animaux [...] un animat est situé dans un environnement ; il possède des capteurs pour acquérir des informations sur son environnement et des effecteurs pour agir au sein de cet environnement. A la différence d'un avatar dont le contrôle est assuré par un utilisateur humain, l'animat doit assurer lui-même ce contrôle pour coordonner ses perceptions et ses actions [...] » –[Tisseau, 2001]

Mais les agents ne se cantonnent pas à modéliser des êtres vivants et peuvent être de natures très diverses. Cette définition d'un agent est volontairement large pour ne pas contraindre l'environnement et le cadre d'exécution d'un agent. En effet, on peut distinguer deux grandes familles d'agents : les agents physiques réels qui évoluent dans des environnements plus ou moins ouverts (applications en robotique, contrôleurs industriels, etc.) et les agents logiciels (des processus informatiques) s'exécutant sur un système ou une interconnexion de systèmes. Ces agents peuvent être inspirés du vivant, comme c'est le cas pour les robots humanoïdes, les colonies de fourmis ou les bancs de poissons, ou être virtuels, comme c'est le cas des agents de diagnostic médical ou de contrôleurs de chaînes de montage.

Environnement

De même, agents et environnements étant très liés, l'environnement d'exécution d'un agent peut être physique et tangible ou bien virtuel et logique. Un environnement peut dès lors désigner tout et n'importe quoi. C'est pourquoi, la communauté de l'intelligence artificielle s'est penchée sur la classification des différents types d'environnements en fonction de leurs propriétés [Russell et Norvig, 2010] :

- observabilité : elle décrit la portion de l'environnement que l'agent est à même de percevoir. Dans la plupart des cas, cette observabilité est partielle (par opposition à totale) et limitée par la précision et la portée des capteurs de l'agent ;
- dynamique : un environnement est dit dynamique (par opposition à statique) s'il est susceptible d'évoluer pendant la phase de décision de l'agent. Un environnement dynamique est plus délicat à gérer puisque la perception de l'agent peut changer alors que l'agent n'a pas encore pris sa décision remettant ainsi en cause ses choix ;
- déterminisme : si l'état suivant de l'environnement est complètement déterminé par son état courant et par l'action que l'agent exécute, alors cet environnement est déterministe. Si au contraire, le résultat de l'action d'un agent ne donne pas systématiquement le même résultat alors l'environnement est stochastique. Ce résultat peut par exemple être régi par une distribution de probabilités ;
- continuité : un environnement peut être discret (par opposition à continu) et présenter un nombre fini de perceptions et d'actions possibles. Par exemple, un agent pion qui se déplace sur un échiquier est dans un environnement discret ; Un agent voiture qui se déplace dans une rue se trouve dans un environnement continu ;
- connaissance : c'est la connaissance qu'a un agent des lois qui régissent l'environnement dans lequel il évolue. Si un agent peut connaître *a priori* le résultat de ses actions alors il est dans un environnement connu. Sinon, il devra apprendre ces lois par une exploration de l'espace d'états.

Nous avons mentionné qu'un agent peut être limité dans sa perception de l'environnement par la fiabilité et la portée de ses capteurs. Cette notion est capitale puisqu'il est rare de trouver des environnements totalement observables et déterministes. Si l'on prend l'exemple d'un robot autonome d'exploration chargé de cartographier l'étage d'un immeuble, il doit prendre en compte dans ses décisions : l'incertitude de ses mesures (par exemple l'erreur entre sa position réelle et celle qu'il observe), l'incertitude sur le résultat de ses actions (si l'agent est bloqué et qu'il ne le sait pas, l'action d'avancer n'aura pas le résultat escompté) et la dynamique de l'environnement (une porte qui se ferme alors que l'agent a décidé de rentrer dans la pièce).

Comportements

Ces agents autonomes en évolution dans des environnements ne sont pas des coquilles vides. Ils sont régis par un ensemble de comportements qui les caractérisent. Encore une fois, plusieurs classifications des agents ont été proposées en fonction de leurs caractéristiques. Certains font la

distinction entre les agents cognitifs/délibératifs et les agents réactifs [Ferber, 1995, Wooldridge et Jennings, 1995].

Dans la première école de pensée (cognitive), les agents possèdent un modèle de leur environnement leur permettant de déclencher un ensemble de raisonnements suite à leurs perceptions. Ils sont intentionnels et dirigés par des buts qu'ils cherchent à accomplir en établissant un certain nombre de plans. La phase de raisonnement (ou de délibération) peut être plus ou moins complexe en fonction du niveau d'intelligence de l'agent. Les agents purement cognitifs basent leurs raisonnements sur une représentation purement symbolique. Mais il existe d'autres types d'agents cognitifs qui raisonnent à partir de données numériques. Ceci nous amène de nouveau à considérer la notion de rationalité qui est intimement liée en IA à une fonction d'utilité. Une fonction d'utilité est définie sur l'ensemble des états et donne à un agent, pour chaque état s, une valeur numérique de satisfaction qu'il aura à se trouver dans s. Un comportement rationnel devrait choisir d'exécuter les actions qui maximisent son utilité espérée, c'est-à-dire, celles qui le rapprochent le plus de son but tout en considérant les contraintes de l'environnement mais également le coût occasionné par les actions qu'il entreprend.

Dans la seconde école de pensée (réactive), les agents prennent des décisions rapidement dans une courte fenêtre de temps en associant pour chaque perception l'action à effectuer. Il faut néanmoins distinguer dans les architectures réactives :

- les agents réflexes/tropiques qui ne possèdent pas de modèle de leur environnement et qui associent des réponses immédiates à leurs perceptions, comparables aux stimuli-réponses observés chez les êtres vivants ; c'est par exemple le cas des agents fourmis [Drogoul, 1993] ;
- les agents ayant une bonne réactivité mais possédant un modèle de leur environnement. Dans ces approches, l'agent pré-calcule une politique (se matérialisant par une table donnant, pour chaque état dans lequel l'agent peut se trouver, l'action qu'il doit effectuer) en fonction de ses buts à partir de son modèle du monde. Ensuite, lors de l'exécution il se contente d'appliquer cette politique en fonction de ses observations. C'est le cas par exemple de certaines approches utilisant les processus de décision markoviens [Puterman, 1994, Sigaud et Buffet, 2010].

3.1.3 Les systèmes multi-agents

En nous plaçant dans un cadre mono-agent, nous n'avons pas encore abordé les capacités sociales des agents qui leurs permettent d'évoluer à plusieurs dans le même environnement en ayant la possibilité d'œuvrer à des buts communs (ou non), connus ou inconnus. Ce domaine : les systèmes multi-agent [Ferber, 1995, Briot et Demazeau, 2001] fait suite à certains travaux de l'intelligence artificielle distribuée telle que les langages d'acteurs [Agha, 1986] ou à d'autres travaux issus de la vie artificielle tels que les phénomènes collectifs [Deneubourg *et al.*, 1990, Drogoul, 1993] et l'évolution de populations à l'aide d'algorithmes génétiques [Goldberg, 1989]. Les SMA se structurent en deux approches dominantes : l'approche collective et l'approche par coordination explicite.

Approche collective

La première s'inscrit dans la lignée de la vie artificielle et de l'intelligence collective dans laquelle plusieurs entités effectuent des tâches relativement simples et d'où émerge, via leurs interactions, un comportement global complexe.

> « Ces travaux sont motivés par la constatation suivante : il existe dans la nature des systèmes capables d'accomplir des tâches collectives complexes dans des environnements dynamiques, sans contrôle externe ni coordination centrale, comme par exemple les colonies d'insectes ou encore le système immunitaire. Les recherches sur les systèmes multi-agents poursuivent ainsi deux objectifs majeurs. Le premier objectif s'intéresse à la réalisation de systèmes distribués capables d'accomplir des tâches complexes par coopération et interaction. Le second objectif concerne la compréhension et l'expérimentation des mécanismes d'auto-organisation collective qui apparaissent lorsque de nombreuses entités autonomes interagissent. Dans tous les cas, ces modèles privilégient une approche locale : les décisions ne sont pas prises par un coordinateur central connaissant chaque entité, mais par chacune des entités individuellement. » –[Tisseau, 2001]

C'est la composante massivement distribuée qui prime dans cette approche collective, clairement inspirée par le vivant, dans laquelle la coopération est bien souvent réalisée par une coordination implicite et indirecte via l'environnement. Ces systèmes, inspirés des capacités auto-organisationnelles et de sélection naturelle des systèmes biologiques, présentent une très forte robustesse aux perturbations aussi bien externes qu'internes. Cette robustesse vient notamment du fait que les rôles des agents sont redondants : aucun agent n'est indispensable individuellement à l'accomplissement du but global. L'exemple le plus connu et le plus manifeste est sans doute celui des colonies de fourmis [Corbara *et al.*, 1993].

> « [...] alors que toutes les fourmis se situent sur un plan d'égalité et qu'aucune d'entre elles ne possède de pouvoir d'autorité stricte sur les autres, les actions des fourmis se coordonnent de manière que la colonie survive [...] » –[Ferber, 1995]

Tout un pan de recherche s'est ouvert à partir de ce constat sur les systèmes bio-inspirés : l'intelligence en essaim [Bonabeau *et al.*, 1999]. Un des objectifs de ce domaine vise à résoudre des problèmes ayant une très forte complexité, que des approches complètes en algorithmique classique peinent à résoudre. Citons notamment les travaux sur l'optimisation à l'aide de fourmis virtuelles tels que la résolution du problème du voyageur de commerce [Colorni *et al.*, 1991, Bianchi *et al.*, 2002], celle du problème de la reconnaissance des graphes hamiltoniens [Wagner et Bruckstein, 1999], celle du problème du *bin packing* [Bilchev et Parmee, 1996] et celle du problème de la coloration de graphe [Costa et Hertz, 1997]. [Monmarché, 2000] propose notamment un état de l'art intéressant sur ces méthodes d'optimisation. L'intelligence en essaim trouve également des applications en robotique collective avec, par exemple, les applications de robots nettoyants [Wagner *et al.*, 2008] ou de robots patrouilleurs [Glad, 2011]. D'autres approches

cherchent plutôt à simuler le plus fidèlement possible les systèmes du vivant pour, d'une part, comprendre les mécanismes intrinsèques qui les régissent (créant, de fait, un pont entre la biologie et les SMA) mais également, pour être en mesure de reproduire ces comportements de manière crédible dans diverses applications (cinéma, jeu vidéo, etc.). Nous pouvons citer notamment les travaux sur le déplacement en nuée des oiseaux [Reynolds, 1987, Reynolds, 2006] et sur les bancs de poissons [Ferber, 1995]. Ces comportements de groupe sont bien évidemment obtenus par la mutualisation des savoirs faire de chacun des agents mais ne sauraient conduire à une solution organisée et cohérente sans des mécanismes d'interaction entre les agents. Ces interactions peuvent revêtir différentes formes :

- dépôt sur l'environnement : c'est une communication indirecte par l'environnement. Dans ce type de communication, chaque agent a la possibilité de déposer des marqueurs dans l'environnement que les autres agents sont capables de capter lorsqu'ils se trouvent à proximité. Ces marqueurs peuvent être de formes diverses qu'ils soient qualitatifs ou quantitatifs. C'est par exemple le cas des dépôts de phéromones dans les colonies de fourmis. Lorsque les interactions ont lieu par des modifications de l'environnement, on parle de *stigmergie* [Grassé, 1959] ;
- répulsion / attirance : c'est une communication par influences mutuelles inspirée de la physique ; les agents émettent des champs d'attraction ou de répulsion qui ont une certaine portée ; lorsque plusieurs agents sont à portée, leurs décisions sont guidées par la résultante de ces influences ; le modèle de flocking de Reynolds fonctionne notamment sur ce principe d'anti-collisions ; cette coordination n'est pas sans rappeler les déplacements par champ de potentiels [Latombe, 1991] dans lesquels l'environnement émet des champs d'attraction permettant aux agents de se déplacer vers leurs buts ;
- éco-résolution : l'éco-résolution met en interaction des éco-agents qui ont un but sous la forme d'un état de satisfaction ; tant qu'un agent est insatisfait, il cherche à se satisfaire ; si dans sa quête de satisfaction il se retrouve gêné par un autre agent, il peut l'agresser ; lorsqu'un agent est agressé et qu'il est insatisfait, il recherche un moyen de s'enfuir.

Jacques Ferber regroupe ces interactions sous le terme de « coopération réactive ».

Approche par coordination explicite

La seconde approche en SMA, que nous avons caractérisée de coordination explicite, met en interaction des agents intelligents qui communiquent de manière directe et explicite pour coopérer et collaborer. L'agent, pour calculer ses plans, peut alors prendre en compte les intentions, les engagements, les demandes et les buts des autres agents. La communication est directe dans le sens où les agents communiquent par messages au travers d'un canal de communication. Le modèle BDI [Rao et Georgeff, 1995] (*Belief Desire Intentions*) est un exemple célèbre de modèle dans lequel les agents raisonnent à partir de croyances, de désirs et d'intentions. Dans ces approches, la solution globale n'émerge pas des interactions locales, mais résulte d'un certain nombre de phases de délibérations et de synchronisations qui visent à maximiser l'efficacité glo-

bale lorsque les agents sont coopératifs. *A contrario*, lorsque les agents sont en compétition, ils cherchent plutôt à maximiser leur efficacité individuelle. Cette complexification du processus de décision nécessite bien souvent l'établissement de rôles et d'une hiérarchie dans le système. On distingue deux approches dans l'établissement de ces organisations [Bonnet, 2008] :

- l'approche *bottom-up* dans laquelle les agents n'ont pas de but commun *a priori*. Ces agents négocient en fonction de leurs propres buts et une organisation qui limite les conflits découle de ces phases de négociations. Le Contract Net [Smith, 1980] est un exemple de protocole permettant d'établir des contrats sur lesquels les agents s'engagent au moyen d'appels d'offres. Dans le cas d'agents en compétition, la mise aux enchères est une pratique courante en SMA et peut nécessiter un agent central qui joue le rôle de commissaire priseur. Les travaux de Gregory Bonnet [Bonnet et Tessier, 2008, Bonnet, 2008] sur la coopération dans les constellations satellitaire, via une logique d'engagement, est un exemple d'approche bottom-up dans laquelle les agents ne communiquent que lorsqu'ils sont à portée d'émission ;
- l'approche *top-down* dans laquelle les buts communs sont connus. Dans ce cas, il faut organiser les agents pour faciliter les échanges et leur coopération. On distingue [Horling et Lesser, 2004, Bonnet, 2008] trois formes d'organisation dans ce type de SMA : les groupes (les équipes, les coalitions, les congrégations), les hiérarchies (simples, uniformes, multidivisionnelles et holarchies) et les marchés. Les travaux de [Matignon *et al.*, 2012] sur la cartographie multi-robots d'un environnement inconnu fournissent un exemple d'approche top-down dans laquelle une équipe de robots se coordonne pour explorer l'espace le plus rapidement possible. Chaque agent calcule sa politique d'exploration sur un modèle commun de l'environnement qui est mis à jour périodiquement à l'aide des informations des capteurs des autres agents. Les travaux de [Lozenguez *et al.*, 2011] suivent également une approche top-down dans laquelle une flotte de robots possédant une carte de l'environnement doit se coordonner pour effectuer plusieurs tâches. L'architecture présente une hiérarchie à deux niveaux dans laquelle un agent *leader* calcule une allocation de tâches au début de la mission et la distribue aux autres robots qui en déduisent leur propre politique.

La communication directe est donc centrale dans cette approche par coordination explicite et la recherche en SMA s'est donc penchée sur la communication entre agents, en cherchant à donner aux agents la possibilité d'exprimer leurs états mentaux (comme c'est le cas des agents BDI par exemple). Ce domaine de recherche s'inspire de la théorie des actes de langages [Searle, 1969] dans laquelle un acte de langage désigne l'ensemble des actions intentionnelles effectuées au cours d'une communication entre humains.

> « [...] l'énonciation [...] est un acte qui sert avant tout à produire des effets sur son destinataire. Lorsqu'on dit "passe moi le sel" à table, on ne s'intéresse pas directement à la vérité ou la fausseté de la phrase, mais surtout à l'effet que cette demande peut produire sur son interlocuteur et donc à ce que ce dernier passe effectivement le sel. Plus exactement cette phrase n'est ni vraie ni fausse, mais elle renvoie à une action qui peut réussir ou échouer.» –[Ferber, 1995]

Il émane de cette théorie trois actes élémentaires : *locutoire* (production de phrases à l'aide d'une grammaire et d'un lexique), *illocutoire* (réalisation de l'acte effectué par le locuteur sur le destinataire : affirmer, questionner, demander de faire, promettre, prévenir, etc.) et *perlocutoire* (les effets que les actes illocutoires peuvent avoir sur les états du destinataire, ses actions, ses croyances et ses jugements). De fait, le succès et la satisfaction d'un acte sont exprimés par des conditions qui participent à la définition de la sémantique de l'acte. ACL-FIPA [FIP, 1997] est un langage de communication agent qui définit un ensemble d'actes de communication de base, associés à la sémantique de chacun. De façon opérationnelle, il définit un ensemble de messages que tout agent doit être capable de traiter. Cet effort de standardisation a ouvert la porte à une interopérabilité des communications entre agents tout en fournissant un langage ayant une bonne expressivité. ACL-FIPA présente néanmoins des limites dans la définition claire et partagée d'une sémantique commune.

3.1.4 Systèmes complexes et simulation multi-agents

La science des systèmes complexes s'est construite avec pour objectif la théorisation des phénomènes d'auto-organisation. Nous nous référons à la définition que donne Rodolphe Charrier dans sa thèse pour caractériser de tels systèmes :

> « [...] un système complexe est constitué d'un grand nombre d'entités en interaction mutuelle, et éventuellement en interaction avec un environnement, dont le comportement global émerge de façon non linéaire du niveau local. » –[Charrier, 2009]

Ces systèmes échappent majoritairement, de par leur complexité, à l'analyse mathématique et il est bien souvent nécessaire de recourir à la simulation pour les étudier [Boccara, 2010]. Le cadre multi-agents se prête tout à fait à la modélisation et à la simulation de tels systèmes en répartissant leur contrôle au niveau des composants autonomes. Nous pouvons citer à titre d'exemple la modélisation multi-agents de la colonne à distiller dans laquelle chaque plateau est un agent autonome qui communique avec ses plateaux voisins [El Falou, 2006] ou encore celle de l'étude de la coagulation par la mise en interaction d'agents cellules [Tisseau, 2001]. Toutefois, l'étude des systèmes complexes sous l'angle de la simulation multi-agents soulève la question des éventuels biais introduits par le simulateur. En effet, cette exécution en parallèle de plusieurs modèles doit être contrôlée pour vérifier qu'il existe bien une équité entre les agents. Cette équité doit être vérifiée à plusieurs niveaux :

- équité logique : les agents doivent tous être désignés autant de fois que les autres sur un cycle de simulation ;
- équité de désignation : le simulateur ne doit instaurer aucune priorité de désignation entre les agents ;
- équité temporelle : les agents doivent disposer du même *quantum* de temps pour s'exécuter (ceci évite les distorsions temporelles).

Ces principes d'équité ne sont pas garantis par les modèles multi-tâches proposés par la plupart des environnements de programmation et l'utilisation de simulateurs n'introduisant pas de biais

est indispensable. Ces principes d'équité sont notamment mis en place dans le simulateur multi-agents oRis [Harrouet, 2000, Harrouet *et al.*, 2002] et ont été vérifiés et éprouvés au moyen de plusieurs expériences dans [Cozien, 2002]. oRis est le simulateur que nous utilisons intensivement dans nos travaux pour simuler nos environnements multi-agents et nous le décrivons plus en détail à la Sec. 4.3 avec l'architecture de notre système.

3.2 Agents mobiles

Cette courte introduction aux agents et aux systèmes multi-agents nous amène finalement aux agents mobiles, le paradigme de programmation répartie que nous avons retenu pour supporter notre architecture. Dans les sections précédentes, nous avons fait implicitement l'hypothèse que les agents, étant situés, pouvaient se déplacer. C'est par exemple le cas des robots explorateurs. Alors pourquoi parler d'agents mobiles subitement ? En réalité, un agent mobile désigne un type d'agents logiciels déployés dans une architecture réseau, qui a la possibilité de se déplacer sur différents sites au cours de son exécution. Ce type d'agent est en fait un raffinement du modèle de code mobile auquel une couche agent (au sens de la Sec. 3.1.2) est ajoutée. Un agent mobile est donc un code mobile aux propriétés d'agent. Il convient dans un premier temps de définir les concepts de mobilité de code avant d'arriver au concept des agents mobiles.

3.2.1 Vers une mobilité du code

Il existe plusieurs paradigmes de programmation pour réaliser des applications réparties et le choix du paradigme est dépendant de l'application à laquelle il est destiné. [Picco, 1998] et [Cubat Dit Cros, 2005] proposent un inventaire détaillé de ces technologies dans lequel :

- un composant représente un processus ;
- un savoir-faire représente un appel de procédure ou de méthode sur un objet ;
- une ressource représente une donnée ou un périphérique utilisé lors des calculs ;
- un site est une machine du réseau pouvant exécuter des composants.

Nous rappelons à présent brièvement ces différents paradigmes.

Le paradigme « classique » client/serveur

C'est le paradigme de programmation réseau classique point-à-point entre deux machines. L'une jouant le rôle de serveur et l'autre de client. Dans ce paradigme, un composant B (le serveur) offre un service et s'exécute sur le site S_B. Les ressources et le savoir-faire nécessaires à l'exécution du service sont également localisés sur S_B. Un composant client A s'exécutant sur un site S_A peut demander l'exécution du service en adressant une requête à B. Le composant B déclenche le service en exécutant le savoir-faire correspondant sur le site S_B. Un résultat peut éventuellement être retourné au composant A. Il ne faut pas confondre ce type d'interaction avec les moyens de l'obtenir. En effet, il existe trois moyens pour mettre en place une architecture client/serveur :

1. envoi de message : c'est la première méthode d'échanges réseaux qui a été mise à disposition dans les systèmes d'exploitation via l'interface socket sur les systèmes UNIX par exemple [Blaess, 2011]. Elle fournit un premier niveau d'abstraction pour manipuler les couches réseau, mais elle laisse à la charge du concepteur la mise en place du protocole de communication et du format des messages ;

2. appel de procédure distante : cette technologie part du constat qu'envoyer un message à un serveur et recevoir une réponse est similaire à un appel de fonction dans n'importe quel langage de programmation. L'appel de procédure à distance (RPC) est une interface de programmation qui permet l'invocation directe et transparente dans un code exécuté sur un site S_A de procédures exécutées sur un autre site S_B [Birrell et Nelson, 1984]. Cette transparence est obtenue par un appel à une librairie tierce (le *stub*) qui transforme les appels de procédure à distance en échanges réseaux par sockets et qui donne l'illusion au programme que son appel est local. Le codage systématique des données échangées (paramètres, valeurs de retour) est obtenu par sérialisation (i.e., transformer la représentation des données en un flux d'octets transférable par le réseau).

3. appel de méthode à distance : l'appel de méthode à distance est l'extension du principe des RPC au modèle de la programmation orientée objets. Dans cette approche, des instances d'objets sont enregistrées dans des annuaires et sont accessibles par une référence réseau sur laquelle il est possible de faire des invocations. Comme pour les RPC, l'illusion de l'appel local est réalisée au moyen d'un *stub*. Le standard CORBA [Geib *et al.*, 1997] définit un ensemble de spécifications qui ont permis de rendre la technologie de l'appel de méthode à distance interopérable et disponible dans la plupart des langages de programmation.

Envoi du savoir-faire

L'envoi du savoir-faire (*Remote Evaluation*) est un paradigme dans lequel un composant A situé sur un site S_A possède le savoir-faire mais ne dispose pas des données qui sont situées sur un site distant S_B. A envoie son savoir-faire à un composant B situé sur le site S_B. B exécute le code qui lui a été envoyé par A et peut éventuellement retourner à A un résultat. Ce paradigme de programmation répartie est notamment utilisé dans les architectures de bases de données de type SQL ou dans les Shells distants.

Récupération du savoir-faire

La récupération du savoir-faire (*Code on Demand*) est le dual du précédent. Un composant A sur un site S_A possède des données sur S_A mais ne dispose pas du code pour manipuler ces données. A doit donc interagir avec un compostant B situé sur un site distant S_B qui possède le savoir-faire. B envoie le savoir-faire à A qui l'exécute sur ses données. C'est le paradigme qui est utilisé par la technologie des Applets Java.

Migration de processus

La migration de processus (le lecteur pourra se référer à [Fuggetta *et al.*, 1998], [Milojičić *et al.*, 1999] et [Milojíčić *et al.*, 2000] pour plus de précisions) est un prolongement de l'envoi de savoir-faire dans lequel le code envoyé d'un site S_A à un site S_B n'est plus simplement une procédure sous la forme d'un ensemble d'instructions mais tout un processus : le code et son unité d'exécution (son espace d'adressage, sa pile, son compteur de programme, etc.). Cette migration a lieu pendant que le processus s'exécute. Elle rend par conséquent la tâche beaucoup plus complexe mais permet au processus de ne pas interrompre son flot d'exécution. En réalité, on distingue la mobilité forte de la mobilité faible. Dans la mobilité forte, le processus peut effectivement être transféré dans son état d'exécution complet (i.e., avec son unité d'exécution) et repris là où il avait été arrêté avant son déplacement. Dans la mobilité faible, le code est transféré sans son unité d'exécution mais peut être accompagné de données d'initialisation. La mobilité forte présente un confort de programmation dans lequel un processus peut être migré de manière transparente à tout moment de l'exécution puisqu'il passe simplement d'un état inactif à un état actif. Dans la mobilité faible, le programmeur doit explicitement définir les situations de migration et capturer l'état d'exécution de son processus (les variables, structures de données qui sont importantes) pour qu'il puisse reprendre là où il était arrêté.

```
Fonction main
  pour i = 0 ; i < 100 ; i++ faire
    affiche(i) ;

Fonction faible_main(j)
  faible_reprendre(j)

Fonction faible_reprendre(j)
  pour i = j ; i < 100 ; i++ faire
    si doisSeDeplacer() alors
      deplacement(i) ;
    affiche(i) ;
```

Par exemple, la fonction `main()` ci-dessus est un code qui affiche les entiers de 0 à 99. Dans un contexte de mobilité forte, le processus exécutant cette fonction `main()` pourra être déplacé autant de fois que souhaité et ceci de manière transparente pour le code puisqu'on constate qu'aucun dispositif spécial n'a été mis en place. Dans un contexte de mobilité faible, le programmeur devra mettre en place, lorsque c'est possible, des procédures pour reprendre l'exécution là où elle a été interrompue. Le même programme transposé dans un contexte de mobilité faible est donné par la fonction `faible_main(j)`, elle-même dépendante de la fonction `faible_reprendre(j)`. On constate que si le processus doit se déplacer, alors il transfert l'incrément j de la boucle en même temps que son code. Arrivé à destination, le nouveau site crée un nouveau processus et exécute la fonction `faible_main(j)` pour reprendre l'exécution en j. Le processus est initialisé lors de sa création avec l'appel `faible_main(0)`. Comme cet exemple le laisse sous-entendre,

il existe deux types de migration. Si la migration est à l'initiative du code alors, elle est dite pro-active, sinon elle est déclenchée par le système et elle est dite réactive.

3.2.2 Du code mobile aux agents mobiles

Comme nous l'avons déjà mentionné, un agent mobile est un code mobile aux propriétés d'agent. C'est-à-dire qu'il est autonome, qu'il évolue dans un environnement à l'aide de son interface sensorimotrice et qu'il dispose d'une composante sociale lui permettant de communiquer avec d'autres agents. Il convient donc d'expliciter ces propriétés d'agent dans un contexte d'agents mobiles. Nous donnons une vision globale issue notamment de [Cubat Dit Cros, 2005] et de [Pommier, 2010] tout en donnant notre propre cadre. On note que d'une manière générale, cette technologie est déployée au moyen d'un intergiciel (*middleware*) qui s'exécute sur le système d'exploitation de chacun des sites d'exécution et qui fournit, en plus du support de la mobilité, un certain nombre d'appels.

Autonomie

Chaque agent s'exécute continuellement dans un processus ou dans un thread. Lorsqu'un agent mobile se déplace, sa boucle d'exécution est relancée après sa migration.

Environnement

L'environnement dans le contexte des agents mobiles correspond à une interconnexion de sites d'exécution. Dans le cadre de nos travaux, cet environnement est un réseau pair-à-pair.

Perception

La perception est variable et dépendante de l'intergiciel agents mobiles. Dans le cadre de nos travaux, la perception d'un agent est limitée au pair qui l'héberge ainsi qu'au voisinage de ce pair. C'est-à-dire qu'un agent est capable de percevoir d'autres agents s'exécutant sur son pair et sur ses pairs voisins. Il est également capable d'accéder à certaines informations sur les pairs de son environnement qui sont mises à sa disposition (uptime, capacité des pairs, propriétaire, service disponibles, etc.). Ce qu'un agent est à même de percevoir est donc entièrement déterminé par l'intergiciel.

Actions

Les actions d'un agent mobile sont les mêmes que celles d'un agent logiciel classique auxquelles s'ajoutent la mobilité. Dans le cadre de nos travaux, ainsi que dans une majeure partie des cas, le déplacement est pro-actif et résulte de la prise de décision de l'agent. Il ne peut en être autrement pour obtenir de l'autonomie.

Communication

La communication entre agents est encore une fois gérée par l'intergiciel et peut revêtir plusieurs formes :

- elle peut être déléguée (par l'utilisation d'une boite aux lettres par exemple) ou directe (par l'établissement d'une liaison point-à-point) entre des agents ;
- elle peut être locale à un site d'exécution (seuls les agents situés sur le même site peuvent communiquer entre-eux) ou distante (il est possible de contacter des agents en exécution sur des sites distants) ;
- elle peut être synchrone ou asynchrone.

Certains choix sont plus difficiles que d'autres à mettre en place. Par exemple, lorsque des communications distantes et directes sont autorisées, il est non seulement nécessaire de pouvoir localiser un agent après plusieurs déplacements (au moyen d'un annuaire centralisé ou d'une mise à jour explicite de toutes ses références) mais il est également nécessaire de recréer des connexions après chaque déplacement. Cette méthode de communication est notamment plus difficile à mettre en place qu'une communication asynchrone indirecte. De même, la mobilité faible est plus simple à mettre en place que la mobilité forte et elle est bien souvent retenue dans les intergiciels agents mobiles. Elle n'en est pas moins bien adaptée à cette technologie puisque la mobilité est à l'initiative de l'agent qui peut prendre les mesures nécessaires à cette occasion pour transférer son état d'exécution. Nous verrons notamment (cf. Sec. 3.2.4) que la plateforme JavAct [Arcangeli *et al.*, 2001] utilise la mobilité faible en encapsulant l'état de l'agent dans des comportements d'acteurs [Agha, 1986].

3.2.3 Discussion autour de la technologie des agents mobiles

La technologie des agents mobiles a vu le jour au milieu des années 1990 et a rapidement suscité l'intérêt de la recherche comme une alternative à la programmation classique client/serveur [Harrison *et al.*, 1994]. Elle a d'ailleurs donné naissance à la conférence *Mobile Agents* [Picco, 2002] qui s'est tenue entre 1997 et 2002. L'utilisation de ce paradigme présente un certain nombre d'avantages (que nous détaillons ci-dessous), mais l'expansion de cette technologie a été grandement freinée par les problèmes de sécurité posés par la migration de code tant du côté agent que du côté site d'exécution [Leriche, 2006]. Il faut cependant relativiser ces propos et noter qu'il existe des plateformes récentes et maintenues pour déployer des agents mobiles. Nous pouvons notamment citer PIAX [PIAX, 2009] et Mobile-C [Chen *et al.*, 2006]. Il semble également qu'il y ait actuellement un engouement pour cette technologie dans le domaine des réseaux de capteurs avec la plateforme MAPS [Aiello *et al.*, 2011] notamment. Cette technologie fait donc toujours l'objet de recherches comme nous le verrons à la Sec. 3.3. Nous présentons maintenant la synthèse d'un ensemble d'avantages et d'inconvénients relatifs à la technologie des agents mobiles qui est issue notamment de [Cubat Dit Cros, 2005], [Leriche, 2006] et [Pommier, 2010].

Avantages

Réduction de la charge réseau. La mise en place d'agents mobiles permet de réduire le trafic réseau étant donné qu'une majorité des traitements sont réalisés localement. C'est notamment le cas dans [Sahai et Morin, 1998] sur une application de supervision et de gestion de réseaux dans laquelle des agents mobiles ont une tâche à accomplir sur un ensemble de sites et la réalisent de manière autonome avant de retourner sur le site de leur initiateur. Dans [Gray *et al.*, 2002], l'évaluation de la réduction de charge est effectuée sur une application militaire de collecte d'information de terrain dans laquelle les agents mobiles font le lien entre les soldats sur le terrain et le poste de commandement.

Diminution du temps de latence. Dans les échanges client/serveur un gain en temps de réponse des applications est obtenu s'il est possible de rapprocher physiquement le client du serveur sur le réseau. On retrouve notamment ce problème dans le placement des caches dans les réseaux de distribution de contenu (*Content Delivery Network*). Une mise en pratique au moyen d'agents mobiles a été réalisée dans [Johansen, 1998] sur une application de serveur d'images satellites de grande résolution et sur une application de serveur de vidéos de grande taille. Dans ces deux applications, le client est un agent mobile qui se déplace sur le serveur pour effectuer les recherches dans ces collections de grande taille. Sur cette application, le gain en temps de réponse de l'application agents mobiles par rapport à l'application client/serveur est très significatif. Dans [El Falou et Bourdon, 2005, El Falou, 2006], le gain en temps de réponse des approches par agents mobiles est illustré sur des problématiques de recherche d'information. Dans ce contexte applicatif, le résultat de la requête d'un client est obtenu suite à l'interrogation en séquence d'un ensemble de serveurs. Dans cette approche, un agent mobile calcule sa politique de visite et d'interaction avec les serveurs à l'aide d'un processus de décision markovien, qui cherche à optimiser le temps de réponse de l'application. Cette politique est notamment calculée en fonction du débit sur les liens et de la taille de l'agent. Cette approche est hybride dans le sens où c'est la politique de l'agent qui détermine si tel ou tel échange se fera par une migration de l'agent ou par une communication client/serveur entre l'agent et le serveur concerné. Les résultats expérimentaux montrent que cette approche hybride par MDP apporte de meilleurs temps de réponse que des approches 100% client/serveur ou 100% mobiles.

Tolérance aux fautes. L'utilisation d'agents mobiles permet de réduire la durée des connexions réseau leur conférant ainsi une meilleure tolérance aux perturbations sur les liens. Une connexion n'est effectivement nécessaire que lorsque l'agent se déplace.

Conception et génie logiciel. Cette nouvelle technologie ouvre la porte à un nouveau paradigme de programmation. Dans JavAct [Arcangeli *et al.*, 2001], par exemple, les agents sont implémentés au moyen d'objets (au sens de la programmation orientée objets) actifs s'exécutant dans un thread et possédant les propriétés d'agents. La porte est alors ouverte à un nouveau paradigme que l'on pourrait caractériser de *Programmation par Agents Mobiles*.

> « Le principal apport des agents mobiles se situe sur un plan du génie logiciel : les agents mobiles sont des unités de structuration des applications ; ils unifient en un modèle unique différents paradigmes d'interaction entre entités réparties (client-serveur, communication par messages, code mobile) [...] le concept d'agent mobile offre un cadre fédérateur pour traiter des problèmes différents et se substituer à diverses solutions *ad hoc* ; c'est l'argument génie logiciel. Mais pour l'instant, cet argument a eu peu de poids face au saut technologique demandé aux développeurs. » –[Leriche, 2006]

Ces agents objets fournissent un niveau d'abstraction supplémentaire et facilitent la programmation des applications distribuées en permettant au programmeur d'implémenter directement les comportements de ses entités communicantes sans programmation réseau.

Adaptabilité. Ce changement dans la manière de programmer les applications réparties ouvre la porte à l'utilisation de technologies et de recherches issues de l'intelligence artificielle et des SMA. L'adaptabilité est une caractéristique notable des SMA dont nous avons déjà traité à la Sec. 3.1.3.

Limites

Comme nous l'avons mentionné plus haut, la technologie des agents mobiles vient avec son lot d'inconvénients et de verrous dont les principaux sont :

Le manque d'interopérabilité. Un très grand nombre de plateformes agents mobiles ont vu le jour (nous en présentons un certain nombre à la section suivante) et ont pour la plupart été développées pour des problématiques bien précises en se basant sur des définitions d'agents multiples. Il en résulte une très grande incompatibilité entre les différentes plateformes qui ne facilite pas la mise en commun d'agents issus de différents organismes. La norme MASIF (*Mobile Agent System Interoperability Facility*) [MAS, 1997, Milojičić *et al.*, 1999] tente néanmoins d'apporter un cadre à ce que doit être une plateforme agents mobiles. Elle donne un certain nombre de définitions et d'interfaces censées faciliter l'interopérabilité entre les différentes plateformes agents mobiles. MASIF normalise notamment la gestion des agents, leur transfert et les espaces de noms. Cependant, malgré cet effort de normalisation, MASIF reste incomplet et ne décrit notamment pas la structure des communications entre les agents ni les protocoles de sécurité à appliquer. On note d'ailleurs que la majorité des plateformes actuelles n'implémentent pas MASIF (cf. Tab. 3.1).

La difficulté de sécuriser les architectures agents mobiles. C'est sans doute le plus gros frein actuel à l'utilisation de cette technologie [Roth, 2004]. On peut distinguer les attaques sur les agents et les attaques sur les plateformes. Le second type d'attaque est facilement neutralisé par l'utilisation de machines virtuelles [Topaloglu et Bayrak, 2008]. Cependant, la protection d'un

agent mobile contre les attaques de plateformes malveillantes est encore un domaine ouvert de recherche. Les attaques sur les agents peuvent revêtir différentes formes. Elles vont de l'altération du comportement de l'agent à la modification de ses données en passant par l'usurpation de son identité. Nous proposons de présenter succinctement deux travaux sur la sécurisation des agents mobiles pour souligner le fait que la communauté est toujours intéressée par ce problème et qu'elle cherche activement à trouver des solutions pour favoriser l'adoption de cette technologie.

[Maggi et Sisto, 2003] s'intéressent uniquement à sécuriser les données d'un agent sous hypothèse que la séquence des sites qu'il va visiter est connue *a priori*. Pour ces auteurs, il n'est pas possible d'empêcher un attaquant malicieux d'altérer un agent mobile. Par contre, ils souhaitent être en mesure de savoir si (et sur quel site le cas échéant) les données collectées ont été altérées ou supprimées une fois que l'agent a terminé sa mission. Pour ce faire les données collectées sont liées entre elles avec des signatures cryptographiques pour que l'initiateur soit en mesure de détecter les incohérences. Pour prévenir les coalitions de pairs malveillants pouvant influencer le chaînage, à chaque saut, l'état de l'historique des pairs restant à visiter est ajouté aux données secrètes pour détecter d'éventuelles manipulations du chemin de l'agent une fois le parcours terminé.

[Ametller *et al.*, 2004] proposent un schéma permettant de sécuriser le code des agents mobiles si leur chemin est connu *a priori*. Cette sécurisation se veut efficace contre les tentatives de modification du code par des utilisateurs malicieux. Dans cette approche un agent est structuré sous la forme d'une paire (C,D), avec C un code d'amorçage en clair permettant de charger le code chiffré D de l'agent. Dans ce schéma, tout pair j possède un trousseau de clés de chiffrement asymétrique (p_j,s_j). Soit $E_{p_j}(d)$, la fonction qui chiffre une donnée d avec la clé publique p_j du site j et soit $E_{s_j}(d)$, la fonction qui signe une donnée d avec la clé privée du site j. Dans cette approche, l'initiateur de l'agent connaît à l'avance l'ensemble des k site que l'agent doit visiter. Il va créer, en conséquence, l'ensemble $\{d_1,\ldots,d_k\}$, avec d_i le code en clair que l'agent devra exécuter sur le site i. Pour tout d_i, l'initiateur peut calculer D_i tel que :

$$D_i = E_{p_i}(d_i,H(C))$$

avec H une fonction de hachage robuste de type SHA-1 et sous hypothèse qu'il est possible de récupérer l'ensemble des clés publiques des k sites. Avec ce schéma, seul le site concerné sera en mesure de déchiffrer le code à exécuter. L'initiateur crée donc un agent (C,D) avec $D = D_1,\ldots,D_k$. Lorsque cet agent arrive sur un site i, le code d'amorçage C est exécuté et consiste à demander à la plateforme s'exécutant sur i le déchiffrement de D_i. La plateforme récupère alors le couple $(d_i,H(C))$ et vérifie que le code d'amorçage C qui s'exécute a le même haché que le $H(C)$ déchiffré. Si cette vérification est bonne, la plateforme autorise l'exécution de d_i. Dans le cas contraire l'agent est corrompu et par conséquent il est détruit. Ce schéma permet une robustesse face à des tentatives d'altération de l'agent (C ou D). Cependant, une plateforme malicieuse connaissant la clé publique du site d'arrivée k peut très bien créer et chiffrer un code D_k qui soit totalement valide mais pourtant malicieux. Pour pallier cette faiblesse, les auteurs proposent de signer le code avec la clé privée s_a de l'initiateur. Pour ce faire, la clé publique p_a

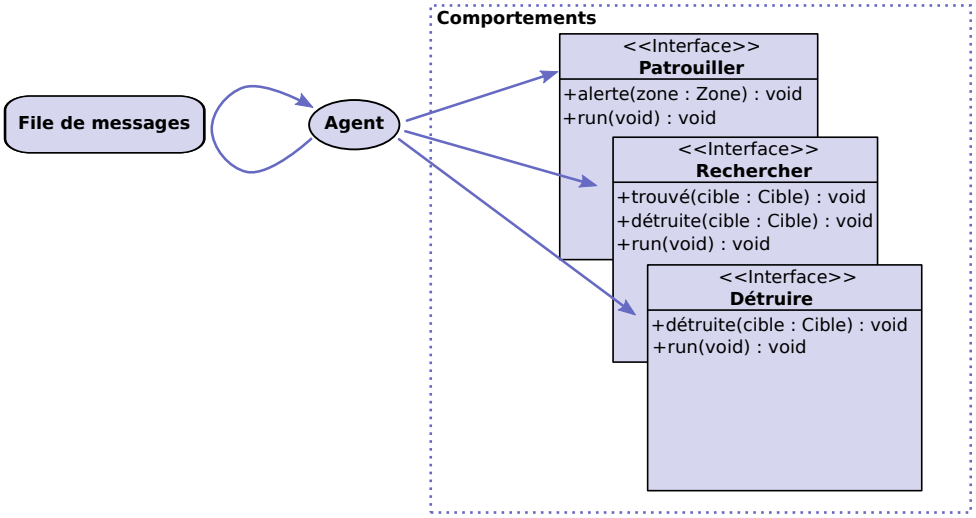


FIGURE 3.1 – Vision schématique d'un agent acteur de surveillance disposant de trois comportements.

de l'initiateur est incluse en clair dans un champ de C et la définition de D_i est ensuite modifiée comme suit :

$$D_i = E_{p_i}((d_i,t_i),H(C))$$

avec

$$t_i = E_{s_a}(H(d_i))$$

t_i est la signature par s_a du code devant être exécuté sur le site cible. Avec ce nouveau mécanisme, la signature du code est également vérifiée à l'aide de p_a. Ce schéma de sécurité reste néanmoins vulnérable à la suppression pure et simple par un attaquant d'un ou de plusieurs D_i. Ce genre d'attaque pouvant, par exemple, être utilisé en cas d'une concurrence déloyale.

3.2.4 Les plateformes agents mobiles et la plateforme JavAct

Il existe un grand nombre de plateformes agents mobiles ayant déjà été inventoriées dans [Picco, 1998, Gray *et al.*, 2002] et [Cubat Dit Cros, 2005]. [Pommier, 2010] en donne une version actualisée dans sa thèse. Ces plateformes ont déjà été étudiées et décrites dans le détail dans ces travaux ; c'est pourquoi, nous nous contentons du tableau récapitulatif et actualisé de la Tab. 3.1 et nous portons une attention plus particulière à la plateforme JavAct [Arcangeli *et al.*, 2001] dont nous faisons l'usage dans nos travaux pour héberger nos agents.

Agents acteurs

JavAct [Arcangeli *et al.*, 2001, Arcangeli *et al.*, 2004a, Arcangeli *et al.*, 2004b, Leriche, 2006, JavAct, 2012] est une plateforme agents mobiles développée à l'IRIT de Toulouse et basée sur la machine virtuelle de Java. Elle offre la possibilité de développer des agents mobiles objets (les agents mobiles JavAct sont des objets qui s'exécutent dans un thread) dont la migration

Plateforme	Langage	Mobilité	MASIF	Dernière version
Aglets [Lange *et al.*, 1997, Jones, 2002]	Java	Faible	**Oui**	2012 *
Ara [Peine et Stolpmann, 1997, Peine, 1998]	Tcl/C/C++	Forte	Non	1997
Concordia [Wong *et al.*, 1997, Walsh *et al.*, 1998]	Java	Faible	Non	2003
D'Agents [Gray *et al.*, 1998, Gray *et al.*, 2002]	Java, Tcl, Scheme	Forte	Non	2002
Grasshopper [Bäumer et Magedanz, 1999]	Java	Faible	**Oui**	2003
JADE [Bellifemine *et al.*, 1999, JADE, 2012]	Java	Faible	Non	2011 *
JavAct [Arcangeli *et al.*, 2001, Arcangeli *et al.*, 2004b]	Java	Faible	Non	2008 *
LIME [Picco *et al.*, 1999]	Java	Les deux	Non	2006 *
PLANGENT [Ohsuga *et al.*, 1997]	Java	Faible	Non	1997
MAPS [Aiello *et al.*, 2011]	Java	Forte	Non	2010 *
Mobile-C [Chen *et al.*, 2006]	C/C++	Faible	Non	2011 *
Mole [Baumann *et al.*, 1998]	Java	Faible	Non	1998
Moorea [Dillenseger *et al.*, 2002]	Rhum/Java	Forte	**Oui**	2002
μCode [Picco, 1999, Picco, 1998]	Java	Faible	Non	2002
NOMADS [Suri *et al.*, 2000]	Java (JVM modifiée)	Forte	Non	2000
PIAX [Teranishi, 2009, PIAX, 2009]	Java	Faible	Non	2011 *
SPRINGS [Ilarri *et al.*, 2006]	Java	Faible	Non	2006 *
Tacoma [Johansen *et al.*, 2002]	C	Faible	Non	2002
Telescript [White, 1994]	Telescript	Forte	Non	1997
Voyager [Voyager, 2012]	Java	Forte	Non	2012 *

TABLE 3.1 – Plateformes agents mobiles les plus connues. (*) Plateformes toujours maintenues.

faible et les communications sont gérées par un langage d'acteurs [Agha, 1986] et reposant sur Java RMI (*Remote Method Invocation*). Nous présentons maintenant le modèle acteur de JavAct et le concept d'agent acteur indépendamment de la mobilité. Un acteur est une entité qui communique par messages asynchrones et qui est identifiée par une référence unique au sein de sa communauté d'agents. Cette référence permet d'envoyer des messages à destination d'une file de messages que possède chaque agent. Cette file de message est traitée en série et ce traitement est intimement lié à la notion de comportement. Dans ce modèle de calcul, un comportement définit une interface que possède l'acteur et à laquelle les messages qu'il reçoit sont adressés. Un comportement fait correspondre chaque message, qu'il doit être en mesure de traiter, à une méthode pour le traiter. Un agent, suivant ce modèle de calcul, peut changer de comportement en cours d'exécution et donc reconfigurer dynamiquement son interface. Programmer un acteur revient alors à programmer ses comportements et leurs enchaînements. La Fig. 3.1 donne un exemple d'agent acteur de surveillance disposant de trois comportements et dont la mission est d'éliminer des menaces. On remarque que chaque comportement est muni d'une méthode `run`. Cette méthode est exécutée dans un thread lorsque son comportement est activé. C'est dans cette méthode que s'exécute la boucle perception–décision–action que nous avons mentionnée à la Sec. 3.1.2. C'est cette fonctionnalité, supportée dans JavAct, qui fait que nous avons à faire à des agents acteurs et non à de simples acteurs. L'exemple de l'agent de surveillance est simpliste ; on suppose que plusieurs agents acteurs de surveillance évoluent dans un environnement découpé en zones. Ils coopèrent pour éliminer des cibles qu'il aperçoivent lors de leurs patrouilles. Dans cet exemple, un agent peut revêtir trois comportements différents :

1. Patrouiller : l'agent explore son environnement à la recherche de menaces. S'il aperçoit une cible, il peut contacter les autres agents pour leur demander de l'aide en leur envoyant le message `alerte(zone)` contenant la zone dans laquelle il se trouve. Dans ce comportement, un agent en patrouille peut également recevoir un message d'alerte d'autres agents, qui déclenchera la méthode `alerte(zone:Zone):void` et qui le fera passer dans le comportement Rechercher ;

2. Rechercher : lorsque l'agent exécute ce comportement, il recherche activement la menace dans une zone qui lui a été notifiée par un message d'alerte. Il peut recevoir les messages `trouvé(cible)`, lui donnant la position de la cible, et `détruite(cible)`, lui indiquant que la cible a été détruite. Sur réception du premier message l'agent passe dans le comportement Détruire. Sur réception du second message, l'agent passe dans le comportement Patrouiller. Si l'agent ne reçoit pas ces messages et si la recherche est infructueuse, alors il repasse dans le comportement Patrouiller. De même si l'agent trouve la cible il passe dans le comportement Détruire ;

3. Détruire : la position de la cible est connue, soit par message des autres agents, soit par la perception de l'agent. L'agent se met en quête de la détruire. Si la cible est détruite alors l'agent repasse dans le comportement Patrouiller et si elle n'est plus visible, l'agent repasse dans le comportement Rechercher.

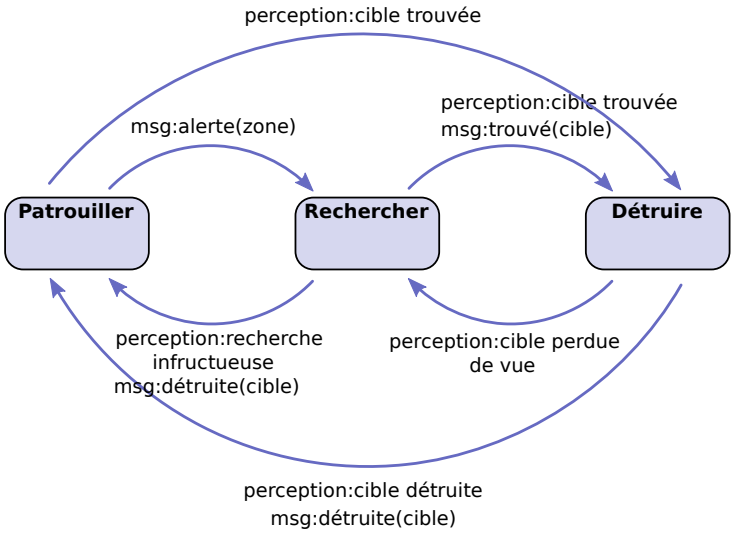


FIGURE 3.2 – Diagramme de transition entre comportements.

L'enchaînement entre ces trois comportements est présenté à la Fig. 3.2. On note qu'un acteur peut être dans l'incapacité de traiter un message qu'il reçoit s'il n'est pas en train d'exécuter le comportement adéquat. Dans l'exemple, c'est le cas si l'agent exécute le comportement Détruire et qu'il reçoit le message `alerte(zone)`. Les messages ne pouvant pas être traités sont ignorés et sont appelés messages orphelins.

Mobilité

L'exemple précédent illustre le concept d'acteurs et de comportement dans un cadre vaste. Nous nous replaçons maintenant dans le cas d'agents acteurs purement logiciels. Un comportement encapsule ses données et ses méthodes propres ; c'est pourquoi, lors d'un changement de comportement, l'acteur se réduit, d'une part, au contenu de sa file de messages entrants et, d'autre part, au nouveau comportement qu'il doit exécuter. La migration d'un agent dans JavAct est donc réalisée lors d'un changement de comportement et consiste en la création à distance d'un acteur exécutant le nouveau comportement et en un transfert de sa file de messages vers le nouveau site. Une fois que la création du nouvel agent est effective sur le nouveau site, l'ancien agent est transformé en proxy dont le but est de relayer les messages à destination du nouvel agent. En théorie, cet ensemble de déplacements crée donc une chaîne de proxies qui permet d'adresser à l'agent les messages qui proviennent de sa référence initiale. Cependant, en pratique, cette chaîne de proxies est non seulement vulnérable, puisque la perte d'un pair de la chaîne entraîne l'incapacité d'envoyer des messages à l'agent, mais elle est également coûteuse et ralentit l'acheminement des messages. Dans JavAct, plutôt que de construire une chaîne, c'est le site créateur de l'agent qui fait office d'unique proxy en étant notifié de la nouvelle position de l'agent pour chacun de ses déplacements. Contacter un agent revient donc à adresser un message à sa référence sur le site de son créateur qui se chargera à son tour de propager le message. Le

mécanisme de mobilité est donc, de fait, totalement transparent pour les communications. Dans JavAct, tout agent a accès à la méthode `go(site)` lui permettant de notifier l'intergiciel de sa volonté de se déplacer sur un certain site. Le déplacement de l'agent est différé à la lecture du prochain message (i.e., le prochain message sera exécuté sur le nouveau site). C'est pourquoi l'appel de `go` est pris en considération mais n'interrompt pas la méthode qui est en train de s'exécuter.

Pour l'implémentation de nos agents, nous avons adapté la librairie JavAct à nos besoins. Cette adaptation se situe au niveau du modèle de communication. Dans une architecture pair-à-pair, les déconnexions sont trop fréquentes pour qu'un pair soit responsable à lui seul de la référence d'un agent. Dans notre modification, les communications entre agents ne sont possibles que localement à un pair (i.e., deux agents doivent être sur le même pair pour pouvoir communiquer). Pour ce faire, chaque agent s'inscrit sur un annuaire local qui peut être consulté par les autres agents et dans lequel il inscrit sa référence. Lorsque l'agent migre vers un autre pair, il se désinscrit de l'annuaire courant et s'inscrit dans l'annuaire du nouveau pair. Avec cette modification, la référence d'un agent n'est perdue que si l'agent est également perdu. Avec ce mécanisme, les agents sont totalement autonomes et ne sont pas dépendants de leur site de création.

3.3 Agents mobiles et applications

Le paradigme agents mobiles est utilisé dans un certain nombre d'applications dont nous donnons un aperçu dans cette section. Nous verrons notamment que les agents mobiles peuvent être utilisés en informatique ubiquitaire [Satoh, 2002, Urra *et al.*, 2009], en diagnostic et réparation [Watanabe *et al.*, 2004], en recherche d'information [Arcangeli *et al.*, 2004b, Papadakis *et al.*, 2008] et en détection d'intrusions [Deeter *et al.*, 2004].

3.3.1 Informatique ubiquitaire

Dans [Satoh, 2002], une architecture agents mobiles est utilisée pour que des applications suivent leurs utilisateurs au cours de leurs déplacements. Par exemple, si un utilisateur se lève de son bureau, prend ses clés de voiture et rentre chez lui, l'application devrait pouvoir se transférer sur sa machine personnelle lorsqu'il rentre chez lui (i.e., lorsqu'il arrive dans la zone d'influence d'un périphérique pouvant héberger l'application) et le tout de manière transparente. Dans cette approche, l'espace est découpé en zones d'influence qui sont dotées de capteurs. Ces capteurs peuvent détecter des tags RFID (*Radio-frequency identification*) qui servent à identifier diverses entités physiques. Plusieurs agents mobiles peuvent être affectés à un tag RFID de sorte que lorsque ce tag est détecté dans une zone d'influence qui contient une machine pouvant héberger des agents, l'ensemble des agents affectés à ce tag migrent sur la machine libre considérée.

Les travaux de [Urra *et al.*, 2009] s'inscrivent dans la problématique de la surveillance de zones par des capteurs en utilisant des réseaux *ad hoc* véhiculaires (VANET). L'idée est de réduire le

nombre de capteurs nécessaires à la couverture d'un espace en exploitant la mobilité de véhicules évoluant dans cet espace. En effet, la couverture d'un espace avec des capteurs fixes implique un nombre de capteurs proportionnel à la superficie de cet espace. Dans l'approche proposée, un ensemble de véhicules évoluent dans l'espace à surveiller et forment un réseau *ad hoc*. Chaque véhicule est muni d'un ensemble de capteurs, d'un GPS, ainsi que d'une plateforme agents mobiles. Lorsqu'une zone de l'espace doit être analysée, des agents sont envoyés en direction de cette zone en migrant de véhicule en véhicule dans le but de collecter un maximum d'informations sur cette zone. La politique de déplacement est effectuée à l'aide d'une heuristique sur une vision du voisinage de l'agent. Un des résultats de cette étude stipule que l'augmentation du nombre de véhicules dans la carte contribue à stabiliser les agents de surveillance sur les zones, résultant en un plus grand nombre de relevés. Ce problème de la recherche des paramètres suffisants pour qu'un système arrive à l'équilibre nous est chère puisque nous y sommes confrontés également dans notre travaux (cf. chapitre 5).

3.3.2 Diagnostic et réparation

Dans [Watanabe *et al.*, 2004], les auteurs proposent une architecture multi-agents mobiles pour le diagnostic et la réparation dans les réseaux. Les algorithmes qui y sont proposés s'inspirent de principes issus du système immunitaire humain. Par analogie, un certain nombre d'anticorps et d'antigènes virtuels modélisés sous la forme d'agents mobiles se déplacent dans le réseau. Ces agents portent les informations sur les données de leur site d'origine et se diagnostiquent mutuellement lorsqu'ils se rencontrent à la recherche de corruption dans leurs données. Le réseau est organisé en réseau aléatoire dans lequel chaque hôte possède une chaîne de bits qui définit son état. Il peut donc y avoir des états valides et des états corrompus. La difficulté vient du fait qu'un hôte ne peut déterminer si son état est valide ou corrompu qu'en comparaison avec celui des autres. Les agents encapsulant le statut de leur site émetteur effectuent une marche aléatoire dans le réseau et se testent mutuellement lorsqu'ils se rencontrent. Lors de cette confrontation, les agents comparent leur état et prennent en compte leur crédibilité (les agents on une mesure de confiance sur le fait qu'ils sont plus ou moins corrompus). La difficulté, une fois cette architecture en place, est de prendre les bonnes décisions de diagnostic. En effet, un agent corrompu ne sait pas qu'il est corrompu (ou tout du moins ne sait pas jusqu'à quel point) et si une trop grande valeur de crédibilité lui est affectée alors il peut arriver qu'un agent totalement sain modifie son état. Cet article donne des éléments de réponse en évaluant un ensemble de tests permettant de prendre une décision de réparation.

3.3.3 Recherche d'information

Les travaux de [Papadakis *et al.*, 2008] proposent une architecture de moteur de recherche qui implémente un robot d'indexation (*Web Crawler*) basé sur des agents mobiles. Ces travaux font l'hypothèse que chaque site à indexer héberge une plateforme agents mobiles. Dans un moteur de recherche classique, l'indexation passe par une phase de web crawl qui rapatrie dans des dépôts

de données l'ensemble du corpus à indexer et qui ensuite insère ce contenu dans un index (c'est le cas de Google [Brin et Page, 1998]). Ce procédé nécessite une large bande passante associée à une espace de stockage important. *A contrario*, dans l'approche proposée par [Papadakis *et al.*, 2008], l'indexation est déportée sur les sites à indexer de sorte que les agents ne rapatrient plus l'intégralité du corpus dans des dépôts mais reviennent simplement avec l'index correspondant à ce qu'ils ont crawlé.

3.3.4 Détection d'intrusions

Le projet APHIDS [Deeter *et al.*, 2004] est une architecture multi-agents mobiles pour la création d'un IDS (*Intrusion Detection System*) décentralisé. Dans cette application, chaque site critique du réseau exécute un ou plusieurs moteurs IDS et une plateforme agents mobiles. APHIDS se situe au-dessus des moteurs IDS dans un modèle en couches. L'administrateur du système définit un ensemble de déclencheurs sous forme d'agents sondes qui sont déployés sur les sites du réseau. Lorsqu'une sonde est activée, un certain nombre d'agents mobiles sont disséminés pour collecter de l'information sur d'autres sites du réseau et être en mesure d'analyser l'attaque à partir d'une vue moins locale. Une fois que la collecte est terminée, l'information est recoupée pour éventuellement faire remonter une alerte. Dans cette approche, les agents mobiles permettent une exploitation rapide des journaux présents sur l'ensemble des sites du système sans pour autant augmenter la charge des liens réseaux puisque l'analyse de ces journaux est faite à distance.

3.4 Conclusion

Les systèmes multi-agents peuvent être vus comme une formalisation des interactions, des échanges et de l'organisation des systèmes distribués constitués d'entités autonomes. Cette formalisation a permis d'accorder la communauté de l'intelligence artificielle – sans cesse confrontée à ce type de systèmes – sur une modélisation commune. Il est possible de classer les SMA en deux grandes familles. Celle que nous avons caractérisée de coordination explicite et l'autre que nous avons caractérisée d'approche collective. Dans la première, les organisations, les interactions, etc. sont explicitement spécifiées *a priori* pour diriger le système vers l'objectif recherché. Ce type de fonctionnement tolère difficilement les cas de figure n'ayant pas été spécifiés lors de la conception du système. Dans la seconde approche, l'objectif global du système est rarement connu à l'avance et émerge de l'interaction des comportements individuels de chacune des entités. Cette approche se caractérise par le très grand nombre d'entités qui sont mises en jeu et par la redondance de leurs comportements. Ceci lui confère une grande robustesse ainsi qu'une faculté d'adaptation accrue aux changements survenant dans son environnement. Ce sont ces deux propriétés de robustesse et d'adaptabilité que nous recherchons pour notre application de stockage décentralisé de données mobiles en pair-à-pair et la modélisation que nous proposons repose sur ce type de SMA. L'adaptabilité sera obtenue par la combinaison conjointe d'un SMA

flexible et par la mobilité des données. Cette mobilité repose elle-même sur la technologie des agents mobiles, nous permettant de construire un SMA dans un contexte de réseau physique réel. Le choix d'une plateforme d'agents mobiles adéquate pour nos besoins a longtemps été un sujet de réflexion. Notre choix s'est finalement porté sur l'intergiciel JavAct pour sa légèreté et le fait qu'il est facilement adaptable à nos besoins. Cependant, même cette version de JavAct modifiée ne solutionne pas les problèmes du passage à l'échelle des plateformes agents mobiles. La totalité des intergiciels d'agents mobiles que nous avons présentés dans ce chapitre implémentent un agent mobile sous la forme d'un objet actif s'exécutant dans un thread qui est ordonnancé par le système d'exploitation qui héberge la plateforme. Ce modèle a l'avantage de laisser au système la charge de l'ordonnancement et de permettre aux entités de communiquer à n'importe quel instant de l'exécution. Cependant lorsque le nombre d'entités hébergées par la plateforme devient très important (notre application se destine à héberger des millions d'agents), ce modèle s'écroule. Il n'existe pas à notre connaissance d'intergiciel d'agents mobiles qui permette de gérer autant d'entités. La première idée qui vient à l'esprit immédiatement pour solutionner ce problème serait de considérer des clusters de pairs comme un seul pair (en mettant en place les mécanismes nécessaires) pour augmenter la capacité d'un nœud. Une autre idée pourrait être de placer les agents dans deux modes. Un mode faible consommation de ressources qui détermine simplement si un agent a besoin d'effectuer des traitements et, le cas échéant, le placer dans une file de candidats pour des traitements plus long. Un mode normal dans lequel l'agent effectue des traitements plus longs. Il se pose alors la question de la communication entre agents qui ne semblerait être réalisable qu'au moyen de boîtes de messages par agent. Malgré la limitation actuelle de cette technologie, l'intergiciel JavAct s'est avéré être suffisant pour les expérimentations que nous avons souhaité mettre en place dans nos travaux.

Deuxième partie

Contributions

Chapitre 4

Mobilité et autonomie des données stockées : un modèle de flocking bio-inspiré

Sommaire

4.1 Autonomie et mobilité des données

Dans les architectures « classiques » pair-à-pair que nous avons présenté au chapitre 2, ce sont les pairs qui assurent le niveau de disponibilité requis par le système en appliquant les politiques de supervision et de réparation sur les documents dont ils sont responsables. Dans cette approche, le document est un matériau inerte dont la pérennité est entièrement dépendante du bon fonctionnement des pairs. Cette organisation des rôles rend difficile, voire impossible, l'application de politiques propres à chaque document puisqu'elles devraient être connues de l'ensemble des pairs. Par conséquent, un document n'a aucun moyen d'ajuster son niveau de robustesse en fonction de l'état actuel du réseau. De même, il n'a aucun moyen de changer son placement en fonction de critères que l'on souhaiterait optimiser.

Nous prenons, dans cette thèse, une orientation différente des approches pair-à-pair classiques en transférant la responsabilité des pairs aux documents. Dans cette nouvelle conception du stockage réparti de l'information, un document applique ses propres règles pour assurer sa disponibilité. Ce nouveau paradigme ouvre la porte à une hétérogénéité des traitements. Par exemple, un document très sensible (un e-mail contenant des informations confidentielles, la comptabilité d'une entreprise, etc.) pourra décider d'utiliser plus de redondance pour sa disponibilité qu'un fichier vidéo ayant moins d'importance. Mais la disponibilité d'un document, comme nous avons pu le constater au chapitre 2, est également impactée par le placement de sa redondance (*churn*, politique de placement, fautes corrélées, etc.). C'est pourquoi, dans cette nouvelle approche, en plus de prendre ses propres décisions, chaque donnée a la possibilité de se déplacer dans le réseau pour se repositionner lorsque sa politique de disponibilité le requiert. Cette mobilité peut donc permettre à une donnée d'explorer le réseau dans lequel elle évolue à la recherche d'un positionnement adéquat (nous abordons ce point au chapitre 6).

Dans ce chapitre, nous décrivons et nous analysons l'architecture qui supporte cette approche dans laquelle les documents sont autonomes et mobiles. Nous traitons également de la recherche d'une telle donnée : comment recherche-t-on une donnée mobile ? Quel est l'impact de la mobilité sur la recherche de l'information ?

4.2 Un modèle de nuées d'agents mobiles

Le modèle de l'information que nous allons présenter est basé sur ces deux grandes idées d'autonomie et de mobilité des données. Le chapitre 1 a mis en évidence la faiblesse des approches centralisées et la force des approches décentralisées. Nous nous interdisons donc, dans notre conception, l'utilisation d'éléments centralisés. C'est pour cette raison que nous avons retenu, pour nos travaux, une modélisation sous la forme d'un système multi-agents composé d'agents mobiles (cf. chapitre 3). Nous sommes en effet en présence d'un système totalement décentralisé dans lequel un grand nombre d'entités autonomes (les données) sont en évolution et

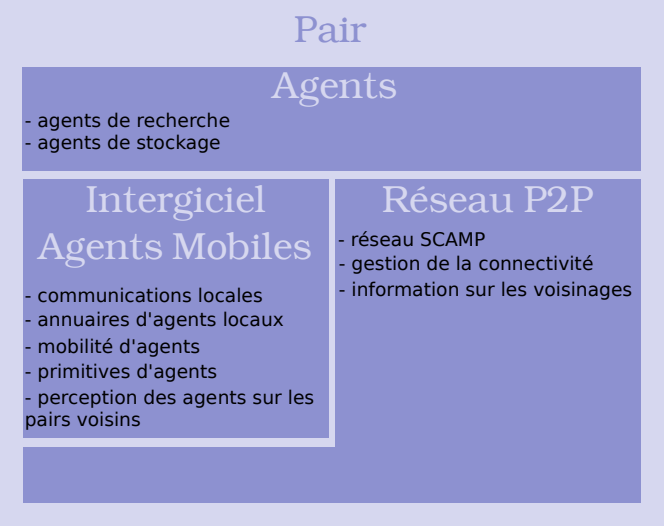


FIGURE 4.1 – Structuration des pairs en couches dans notre architecture pair-à-pair et multi-agents.

en interaction dans un environnement (le réseau pair-à-pair). Le réseau pair-à-pair devra donc également satisfaire les contraintes de décentralisation.

4.2.1 Schéma de l'information mobile et architecture de l'application

Le modèle de l'information multi-agents mobiles que nous considérons dans cette thèse a d'abord été introduit dans [Pommier et Bourdon, 2009] et nous l'avons ensuite adapté dans un cadre applicatif réel dans [Pommier *et al.*, 2010]. Des éléments présentés dans ce chapitre sont également détaillés dans la thèse d'Hugo Pommier [Pommier, 2010] avec lequel nous avons grandement collaboré entre 2009 et 2011. Nos contributions, dans ce chapitre, se situent au niveau de la validation de la simulation par des expérimentations sur un réseau pair-à-pair physique réel (cf. Sec. 4.3), au niveau de l'évaluation des algorithmes de flocking (cf. Sec. 4.4) et au niveau de la recherche de nuées (cf. Sec. 4.5). Ce modèle de l'information multi-agents mobiles se structure en trois couches illustrées à la Fig. 4.1.

La couche réseau pair-à-pair

C'est la couche qui supporte l'architecture. Elle définit un réseau logique qui sert d'environnement aux agents. Obtenir de la robustesse au niveau multi-agents n'est pas possible si le réseau pair-à-pair sur lequel l'architecture repose n'est pas robuste. Nous avons présenté à ce propos à la Sec. 1.6.4 le protocole SCAMP, un réseau pair-à-pair aléatoire au fonctionnement totalement décentralisé. Nous l'avons retenu dans ces travaux pour sa tolérance aux fautes modulable, sa décentralisation, sa flexibilité et sa légèreté. Cette tolérance aux fautes s'obtient en augmentant le nombre d'arcs du graphe aléatoire au-dessus de son seuil de connexité par le biais de la constante de connexité c. Pour rappel, un (n,c)-SCAMP a un degré moyen qui converge

vers $(c+1)\log(n)$ avec n le nombre de pairs du réseau. Plus le degré augmente au-dessus du seuil de connexité et moins la perte d'un pair risque de déconnecter le graphe. C'est sur ce type de réseau que repose notre architecture. La couche réseau, en plus de sa fonction de maintien de la connectivité entre les pairs, met à la disposition des couches supérieures l'information sur les voisinages pair-à-pair. C'est-à-dire que sur chaque pair x il est possible (via un appel de méthode) d'interroger la couche réseau pour qu'elle retourne les pairs voisins $v(x)$ de x. SCAMP est un graphe aléatoire orienté, néanmoins la vue que la couche réseau donne à ses couches supérieures est celle d'un graphe non orienté. Le voisinage d'un pair est donc défini comme étant l'union de ses successeurs et des ses prédécesseurs. On obtient donc la définition suivante :

Définition 25 *Soit un (n,c)-SCAMP modélisé sous la forme d'un graphe orienté $G = (V,A)$. On définit $v : V \mapsto 2^V$ la fonction qui retourne le voisinage d'un pair $x \in V$ telle que $v(x) = PV(x) \cup IV(x)$. On note $V_x = v(x)$, avec $IV(x)$ l'ensemble des prédécesseurs du pair x (cf. Def. 10) et $PV(x)$ l'ensemble des successeurs du pair x (cf. Def. 9) dans SCAMP.*

La couche intergiciel d'agents mobiles

Cette couche exécute une plateforme d'agents mobiles. Pour les besoins de notre application, cet intergiciel doit fournir, *a minima*, un certain nombre de fonctionnalités dont voici la liste :

- gestion de la mobilité des agents (suspension, transfert et reprise) ;
- gestion des primitives d'agents (communications agent-agent, création d'agents, destruction d'agents, etc.) ;
- gestion des communications entre agents (communications directes locales, communications par l'environnement) ;
- gestion de la perception des agents (les agents perçoivent les agents s'exécutant sur le même pair qu'eux ainsi que ceux s'exécutant sur les pairs voisins).

Dans la suite de nos travaux, nous faisons abstraction de la plateforme concrètement utilisée pour réaliser cette couche, l'important étant qu'elle fournisse les fonctionnalités ci-dessus. Cependant, lors de nos expérimentations sur un réseau physique réel, nous avons utilisé l'intergiciel JavAct. JavAct (cf. Sec. 3.2.4) fournit dans sa version de base : le support de la mobilité, la gestion des communications par messages entre agents et des primitives pour la gestion des agents. Au regard des points que nous avons listés ci-dessus, nous avons dû modifier JavAct pour la réalisation de notre prototype :

- communications locales et directes : un mécanisme d'annuaire s'exécute dans l'intergiciel de chaque pair. L'annuaire d'un site référence les agents en exécution sur ce site ainsi qu'un ensemble de données publiques que l'agent souhaite communiquer (son créateur, son identifiant, sa signature, etc.). Ceci permet aux agents d'obtenir la référence des agents s'exécutant sur leur pair pour leur adresser des messages ;
- communications locales et indirectes : notre JavAct modifié met à disposition une table de hachage accessible en lecture/écriture par tous les agents et fournit, de ce fait, une communication indirecte par l'environnement ;

– facilité de perception des agents : l'intergiciel permet aux agents, grâce à l'annuaire, de percevoir les agents s'exécutant sur le même pair qu'eux. Cet annuaire permet également à un agent (en passant par une primitive de l'intergiciel) d'interroger ses pairs voisins pour récupérer leur annuaire et, par conséquent, les données publiques des agents voisins.

La couche agents

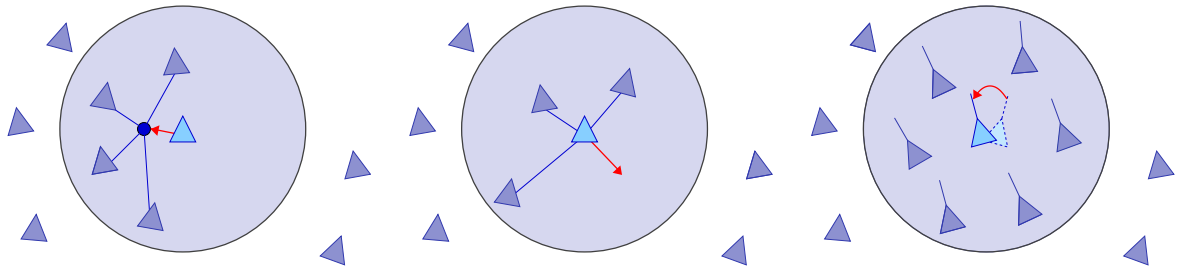
Cette couche est le lieu d'exécution des agents mobiles. Elle dépend des deux couches précédentes et fournit à chaque agent la possibilité de percevoir, de raisonner et d'agir sur son environnement. Dans le cadre de notre application de stockage décentralisé de documents mobiles, cette couche héberge majoritairement [11] deux types d'agents mobiles : les agents mobiles de recherche et les agents mobiles de stockage. Les agents de recherche feront l'objet de la Sec. 4.5.

Les agents de stockage vont, quand à eux, encapsuler les données à insérer dans le système. Dans notre approche, une donnée qui doit être stockée est fragmentée à l'aide d'un (m,n)-code d'effacement (cf. Sec. 2.2.2) pour assurer sa redondance. Le choix des codes d'effacement par rapport à la réplication vient pleinement du fait que ces codes sont moins coûteux en terme d'espace de stockage (comme nous avons pu le voir au Chap. 2). À l'issue de la phase d'encodage, chaque fragment est encapsulé dans un agent mobile distinct. Les fragments d'un document deviennent donc autonomes et mobiles dans le réseau.

4.2.2 Contrôle de la mobilité : le déplacement en nuées

La mobilité de fragments doit cependant être contrôlée. Nous avons vu au Chap. 2 que pour assurer la robustesse d'une donnée dans le temps, elle doit être réparée périodiquement. Nous avons également vu que dans ce contexte, un placement local permet la mise en place d'un processus de supervision et de réparation décentralisé. C'est pourquoi, pour superviser et réparer une donnée mobile, il est nécessaire de préserver un fort degré de localité entre ses fragments malgré leurs déplacements. Cette contrainte de localité entre fragments est également importante lors de la recherche des données. Étant donné que les fragments sont mobiles, il n'est pas possible de recourir à une architecture pair-à-pair décentralisée structurée (cf. Sec. 1.5) pour accélérer les recherches. En effet, le principe d'une DHT étant de stocker une donnée sur le pair ayant l'identifiant logique le plus proche d'elle, ceci impose que la donnée reste immobile une fois qu'elle a trouvé sa place. La recherche d'une donnée mobile va donc inévitablement se baser sur des mécanismes moins efficaces (et notamment sur les marches aléatoires comme nous le verrons à la Sec. 4.5) qui doivent localiser chacun des fragments. Cette localisation peut être accélérée si les fragments d'une donnée suivent un placement local, puisque dans ce cas, il suffit de contacter un des fragments pour contacter l'ensemble des fragments du groupe (par propagation). Les agents fragment d'une donnée doivent donc se déplacer en groupe pour permettre une réparation décentralisée.

11. Nous pourrions imaginer d'autres types d'agents, en particulier dans une perspective de sécurisation du dispositif, avec des agents de surveillance, de faux agents malveillants, etc.



(a) Règle de cohésion. (b) Règle de séparation. (c) Règle d'alignement.

FIGURE 4.2 – Schéma illustrant l'application des trois règles de Reynolds par un agent (au centre) sur une vision locale de son voisinage.

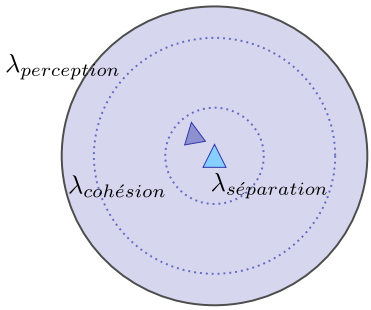
Cette problématique de déplacement de groupe sans contrôle central a fait l'objet d'études en biologie avec, notamment, l'étude des comportements en essaim (ruches, nuées d'oiseaux, bancs de poissons, etc.). Certains travaux se sont ensuite penchés sur la reproduction de ces comportements en simulation ou en robotique. C'est, notamment, le cas des travaux de Craig Reynolds [Reynolds, 1987, Reynolds, 1999] sur l'étude de règles régissant le déplacement en nuées des oiseaux (le *flocking*). C'est ce modèle de déplacement sur lequel nous nous inspirons pour que nos agents mobiles se déplacent en groupe. Nous l'avons retenu parce qu'il fait appel à des règles simples qui sont appliquées par chaque agent sur une vision limitée de la nuée à son voisinage.

4.2.3 Modèle de Reynolds

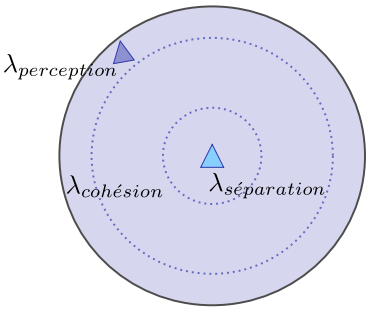
Dans ses travaux, Reynolds souhaitait trouver des règles simples, facilement applicables par chaque agent, permettant de reproduire le vol des oiseaux en nuées en faisant émerger le comportement global du groupe par l'application asynchrone de règles locales. Reynolds a identifié trois règles qu'un agent doit appliquer pour qu'un comportement de flocking émerge :

1. cohésion : chaque agent veille à ne pas être trop éloigné de ses agents voisins (cf. Fig. 4.2(a)). Chaque agent va calculer le centre de gravité des agents qu'il perçoit à partir de leurs positions. Il en déduit ensuite un vecteur de cohésion en fonction de sa propre position ;
2. séparation : chaque agent veille à ne pas être trop prés de ses agents voisins, afin d'éviter les collisions (cf. Fig. 4.2(b)). La séparation est obtenue par un bilan de forces en calculant, pour chaque agent voisin, une force de répulsion obtenue en fonction de la distance qui les sépare (i.e., un agent proche est très répulsif) ;
3. alignement : chaque agent adapte sa vitesse et sa direction en fonction de celle de ses agents voisins (cf. Fig. 4.2(c)). Cette adaptation est faite en prenant la moyenne des valeurs du voisinage. Un vecteur de déplacement est ensuite déduit en soustrayant la valeur de vitesse moyenne des voisins à la valeur actuelle de l'agent.

La décision de déplacement de l'agent peut ensuite être prise en normalisant les vecteurs issus de ces trois règles et en prenant leur somme pondérée. Les travaux de Reynolds se placent



(a) Violation de la règle de séparation.



(b) Violation de la règle de cohésion.

FIGURE 4.3 – Illustration de la violation des règles de séparation et de cohésion par rapport à $\lambda_{séparation}$ et $\lambda_{cohésion}$.

dans un espace euclidien et mettent en jeu les notions de distance entre agents, de rayon de perception et de vitesse de déplacement. Ces notions n'ont pas d'équivalent trivial dans un réseau et l'applicabilité de telles règles dans notre contexte requiert leur adaptation.

4.2.4 Nuées d'agents mobiles

Hugo Pommier [Pommier, 2010] propose dans sa thèse une adaptation des règles de flocking de Reynolds pour qu'elles soient applicables par des agents mobiles dans un réseau pair-à-pair. Cette adaptation consiste à définir les notions de rayon de perception, de distance entre agents et à modifier les règles de Reynolds en adéquation avec ces définitions.

Définition 26 *(Perception) Un agent exécutant l'algorithme de flocking perçoit les agents s'exécutant sur le même pair que lui et sur ses voisins directs.*

Définition 27 *(Distance) La distance $d(a,b)$ entre deux agents mobiles a et b est donnée par la latence entre leurs pairs hébergeurs, se mesurant à l'aide du RTT (Round Trip Time).*

Il définit ensuite trois paramètres $\lambda_{perception}$, $\lambda_{séparation}$ et $\lambda_{cohésion}$ qui statuent sur la violation des règles de cohésion et de séparation. La valeur $\lambda_{perception}$ définit simplement le rayon de perception de l'agent. Le paramètre $\lambda_{séparation}$ définit la distance en-dessous de laquelle la séparation entre deux agents est violée (cf. Fig. 4.3(a)) et le paramètre $\lambda_{cohésion}$ définit la distance à partir de laquelle la cohésion entre deux agents est rompue (cf. Fig. 4.3(b)). C'est-à-dire que si un agent détecte qu'un autre agent viole la règle de séparation, alors il va chercher à s'en éloigner. Par contre, si un agent détecte qu'un autre agent viole la règle de cohésion, alors il va chercher à s'en rapprocher.

Les règles de flocking de Reynolds sont ensuite transposées dans un contexte d'agents mobiles évoluant dans un réseau pair-à-pair comme suit :

1. cohésion : un agent mobile se déplace vers l'agent le plus éloigné (au sens du RTT) de son voisinage pair-à-pair ;

2. séparation : un pair ne peut pas héberger deux agents mobiles appartenant à la même nuée (cette règle garantit que deux fragments d'un même document ne seront jamais stockés sur le même pair, pour éviter que la disponibilité requise ne soit diminuée) ;
3. alignement : cette règle n'a pas d'équivalent dans ce modèle de flocking. Il faudrait définir la notion de direction et de vitesse dans un réseau. On note néanmoins que le mécanisme de dépôts de phéromones que nous décrivons plus bas (cf. Sec. 4.2.5) peut faire office de règle d'alignement, en dirigeant la nuée vers des zones peu explorées.

Dans la version de Reynolds, les règles fournissent à l'agent un vecteur de déplacement. Dans la version de Pommier, ces règles sont appliquées en même temps et dépendent des distances. Les règles énoncées ci-dessus fixent alors les variables λ à : $\lambda_{séparation} = 0$ et $\lambda_{cohésion} = \text{MaxRtt}$, avec MaxRtt la latence maximale entre le pair qui exécute l'algorithme et ses voisins. Pommier propose ensuite l'algorithme 16 de flocking exécuté par un agent mobile f sur un pair x. Cet agent f est issu du processus de fragmentation par code d'effacement d'une donnée *data*. La nuée N_{data} est définie comme étant l'ensemble des agents fragment issus de l'encodage de *data*.

Algorithme 16 : flocking exécuté par un agent f

Entrées : x un pair, f un agent en exécution sur x, N_{data} la nuée chargée du stockage de *data*

Sorties : p un pair sur lequel f doit se déplacer

$Busy \leftarrow$ pairs $\in V_x$ hébergeant des fragments $\in N_{data}$;
pour chaque $y \in Busy$ **faire**
  si $d(x,y) < \lambda_{séparation}$ **alors**
    $Busy \leftarrow Busy \setminus \{y\}$;

$Free \leftarrow$ pairs $\in V_x$ n'hébergeant pas de fragments $\in N_{data}$;
pour chaque $y \in Busy$ ***and*** $z \in Free$ ***and*** $y \in V_z$ **faire**
  si $d(y,z) > \lambda_{cohésion}$ **alors**
    $Free \leftarrow Free \setminus \{z\}$;

retourner *un pair* $p \in Free$ *choisi aléatoirement*

Dans cette première version de l'algorithme [Pommier *et al.*, 2010], l'agent fragment f construit d'abord l'ensemble des pairs occupés de son voisinage (ceux qui hébergent des membres de sa nuée) et l'ensemble des pairs libres de son voisinage (ceux qui sont des candidats potentiels au déplacement). Ensuite, les candidats au déplacement qui sont trop éloignés de la nuée au sens du $\lambda_{cohésion}$ sont supprimés des choix puis une cible potentielle pour le déplacement est désignée.

L'algorithme de Pommier présente une faiblesse en terme de maintien de la cohésion. En effet, il est possible d'avoir dans l'ensemble $Free$ des pairs n'ayant pour seul voisin dans la nuée, que le pair qui héberge le fragment qui s'apprête à se déplacer. Si de tels pairs sont choisis pour être la cible du déplacement, alors l'agent va rompre volontairement la cohésion avec la nuée et se retrouver isolé. Ce phénomène non souhaité est illustré dans la situation de la Fig. 4.4.

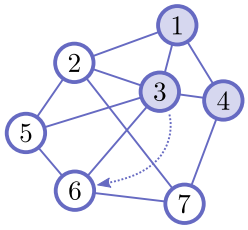
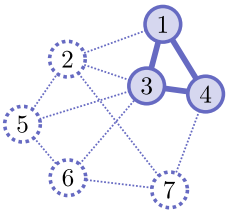
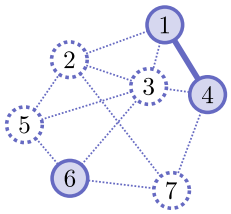


FIGURE 4.4 – Illustration d'une rupture de cohésion si le choix dans *Free* n'est pas restreint.



(a) Nuée de cohésion maximale égale à 3 ; pas de rupture de cohésion.



(b) Nuée présentant deux composantes connexes. Sa cohésion est donc égale à 2 ; rupture de cohésion.

FIGURE 4.5 – Une nuée peut être vue comme un sous-graphe du réseau. La valeur de cohésion est donnée par la taille de la composante connexe la plus grande.

Dans cette figure, une nuée de trois fragments (représentés par les disques pleins) est hébergée sur les pairs d'identifiant 1, 3 et 4. L'agent hébergé sur le pair 3 a la variable $Free = \{2, 5, 6\}$ (on fait l'hypothèse que ces pairs sont tous valides par rapport à la condition sur le $\lambda_{cohésion}$). Si finalement l'agent en 3 choisit de se déplacer sur le pair 6, alors, il se retrouvera isolé à l'issue de son déplacement (i.e., il ne percevra plus aucun agent de sa nuée). Or, les ruptures de cohésion doivent être évitées pour des raisons de supervision et de réparation (cf. Chap. 5). Une nuée peut être vue comme un sous-graphe du réseau dans lequel les sommets correspondent aux agents fragment et les arêtes correspondent aux adjacences entre ces agents. Par exemple, à la Fig. 4.5(a), la nuée des trois agents s'exécutant sur les pairs 1,3 et 4 peut être représentée par le sous-graphe du réseau ayant pour sommets $V = \{1,3,4\}$ et pour arêtes $E = \{(1,3),(3,4),(4,1)\}$.

Définition 28 *(Cohésion) Soit une nuée de fragments et son sous-graphe réseau correspondant. Sa cohésion est donnée par la taille de sa composante connexe la plus grande.*

Ainsi, par définition, dès lors qu'un fragment est isolé ou que la nuée est séparée en sous-nuées, elle fait l'objet d'une rupture de cohésion. Par exemple, à la Fig. 4.5(a), la nuée n'est pas victime de rupture de cohésion et sa cohésion est égale à 3. Par contre, dans la Fig. 4.5(b), la même nuée a subi une rupture de cohésion et présente deux composantes connexes. Sa valeur de cohésion est donc égale à 2. Étant donné que la perception d'un agent se limite aux voisins du pair qui l'héberge, il nous semble que la contrainte de cohésion comme elle a été énoncée à l'algorithme 16, sur une valeur de RTT uniquement, est insuffisante et doit être renforcée. Nous proposons donc en conséquence l'algorithme 17. Il prend en compte ce phénomène en contraignant les candidats

dans $Free$ à posséder au moins un agent de la nuée comme voisin et en s'assurant que cet agent soit différent de celui qui effectue le déplacement. Si nous reprenons l'exemple de la Fig. 4.4, nous aurions la variable $Candidates = \{2\}$ et le déplacement de l'agent hébergé sur le pair 3 ne provoquerait pas de rupture de cohésion. Les pairs de $Busy$ qui sont des voisins de pairs de $Free$ sont appelés *pairs pivots* pour l'agent qui se déplace. Dans ce même exemple de la Fig. 4.4, le pair 2 est un candidat pour le pair 3 parce qu'il est voisin du pair 1, qui est lui-même un pair pivot pour l'agent s'exécutant en 3.

Algorithme 17 : flocking modifié et exécuté par un agent f

Entrées : x un pair, f un agent en exécution sur x, N_{data} la nuée chargée du stockage de $data$

Sorties : p un pair sur lequel f doit se déplacer

$Busy \leftarrow$ pairs $\in V_x$ hébergeant des fragments $\in N_{data}$;
pour chaque $y \in Busy$ **faire**
  si $d(x,y) < \lambda_{séparation}$ **alors**
    $Busy \leftarrow Busy \setminus \{y\}$;
$Free \leftarrow$ pairs $\in V_x$ n'hébergeant pas de fragments $\in N_{data}$;
pour chaque $y \in Busy$ ***and*** $z \in Free$ ***and*** $y \in V_z$ **faire**
  si $d(y,z) > \lambda_{cohésion}$ **alors**
    $Free \leftarrow Free \setminus \{z\}$;
$Candidates \leftarrow \{z \in Free \mid \exists\, y \in Busy, y \in V_z\}$;
retourner *un pair* $p \in Candidates$ *choisi aléatoirement*

4.2.5 Dépôts de phéromones

Ce modèle de flocking d'agents mobiles vient avec un mécanisme de stigmergie (cf. Sec. 3.1.3), introduit pour permettre notamment [12] d'équilibrer les déplacements d'une nuée sur son environnement. Ce mécanisme bio-inspiré consiste à déposer une certaine quantité de phéromones, ϕ, à chaque déplacement d'un agent sur un lien. Un lien logique n'ayant pas d'existence propre, ce sont les pairs qui stockent, pour chacun de leurs voisins, une valeur entière représentant la quantité de phéromones du lien vers ce voisin.

Définition 29 *Soit deux pairs a et b, avec $a \in V_b$ et $b \in V_a$. $\phi_{a,b}$ est le niveau de phéromones de l'arête (a,b), stocké en a. De même, $\phi_{b,a}$ est le niveau de phéromones de cette même arête (a,b), stocké en b. On a $\phi_{a,b} = \phi_{b,a}$.*

Dans la Fig. 4.6(a), un agent mobile situé sur le pair d'identifiant 6 s'apprête à se déplacer sur le pair d'identifiant 3. Avant son déplacement le lien n'a jamais été parcouru : son niveau de

12. Ce mécanisme sert également de support à un algorithme de calcul de confiance entre les pairs qui ne sera pas présenté ici, étant donné qu'il n'intervient pas dans nos travaux ; cf. [Pommier, 2010].

(a) Avant le déplacement de l'agent s'exécutant sur le pair 6, le niveau de phéromones de l'arête (6,3) est égal à 0.

(b) Une fois que l'agent s'est déplacé, sur le pair 3, le niveau de phéromones a été incrémenté (ici d'une unité).

FIGURE 4.6 – Dépôts de phéromones lors du déplacement d'un agent.

phéromones est nul. Une fois que l'agent s'est déplacé, à la Fig. 4.6(b), le niveau de phéromones de l'arrête (6,3) a été incrémenté d'une certaine constante (ici 1). Ce mécanisme simple va donc affecter une plus grande valeur de phéromones à une arête qui est plus visitée qu'une autre. Une fois que ce mécanisme est en place, il est utilisé dans la prise de décision de chaque agent d'une nuée : dans les algorithmes de flocking ci-dessus, l'agent choisit le pair candidat au déplacement qui possède le plus petit niveau de phéromones. La dernière ligne de l'algorithme 17 se réécrit donc en :

$$\textbf{retourner} \quad \underset{p \in candidates}{\arg\min} \quad \phi_{x,p}$$

Ce mécanisme va pousser les nuées (comme nous le verrons à la Sec. 4.4.2) à explorer des zones qui ont été moins explorées que d'autres. Les valeurs de phéromones peuvent être partagées par toutes les nuées, auquel cas, l'algorithme de flocking va chercher à éviter les chevauchements entre elles. Sinon, les phéromones peuvent être propres à chaque nuée si l'objectif pour ces nuées est de parcourir rapidement l'espace. Dans les deux cas, il nous semble que ce mécanisme de phéromones peut faire office de règle d'alignement dans cette transposition des règles de Reynolds. En effet, l'objectif de ces phéromones est bien de diriger les nuées vers des zones peu explorées.

4.3 Cadre expérimental

Ce modèle de stockage de données mobiles et bio-inspiré a fait l'objet d'un certain nombre de simulations mais également d'expérimentations réelles ; l'objectif étant de s'assurer que les nuées suivant ce modèle ont le comportement qui est attendu et de mettre en évidence, puis d'étudier, les paramètres pouvant avoir un impact sur ce comportement. Ces deux approches expérimentales n'ont pas du tout la même vocation. Dans l'approche par simulation, le système peut être étudié intensivement, en faisant varier plusieurs paramètres. La simulation permet, notamment, une étude du modèle qui soit rapide, à moindre coût, sur un temps simulé pouvant être très long, sur une vision globale du système et sur différents facteurs d'échelle. Dans une approche par

expérimentation réelle, un prototype de l'application est confronté à son environnement réel et n'est pas soumis aux éventuelles hypothèses simplificatrices introduites dans le simulateur. L'expérimentation réelle permet bien évidemment une validation comportementale du système, mais elle permet également de valider le cadre de simulation dans le cas où les résultats obtenus sont similaires. L'expérimentation permet également de mettre en évidence des détails techniques qui ont pu être omis lors de l'établissement du modèle. Cependant, ces expérimentations sont plus fastidieuses à mettre en place et sont vite limitantes en terme de passage à l'échelle et en terme de durée de simulation. Pour un système de stockage décentralisé, il faudrait en effet pouvoir monopoliser un réseau pair-à-pair large échelle sur plusieurs semaines. Partant de ce constat, l'approche expérimentale que nous avons suivie dans nos travaux fait appel majoritairement à la simulation. Nous avons cependant pris la peine de créer un prototype afin de valider le simulateur en comparant les résultats obtenus par les deux approches. Un ensemble d'évaluations du système vont être proposées dans le reste de ces contributions ; il nous semble donc important de décrire préalablement le cadre de ces évaluations ainsi que les choix de conception que nous avons opérés.

4.3.1 Environnement de simulation

L'environnement de simulation est développé sous oRis [Harrouet, 2000, Harrouet *et al.*, 2002], un simulateur multi-agents muni de son propre langage pour implémenter les différentes entités autonomes. C'est un simulateur qui ordonnance des agents sur une période de temps discrétisée sous la forme de cycles de simulation, dans lequel l'équité entre les entités est garantie à plusieurs niveaux (cf. Sec. 3.1.4). L'une des particularités d'oRis vient du fait qu'il offre la possibilité à l'expérimentateur d'interagir avec la simulation alors même que celle-ci s'exécute. oRis offre notamment la possibilité de modifier le comportement d'agent entre deux cycles de simulation.

oRis vient nativement avec une fonction d'envoi de messages qui permet à chaque agent d'envoyer des objets à destination de la file de messages entrant des autres agents. Cependant, ce mécanisme de communication est insuffisant pour notre application puisqu'il ne fournit aucune gestion de la capacité des liens, qui est pourtant primordiale dans notre simulateur, puisqu'il a pour vocation de simuler un réseau. Nous avons donc commencé par implémenter une couche réseau (n,c)-SCAMP paramétrée par n, le nombre de pairs du réseau, et c, la constante de tolérance aux fautes (cf. Sec. 1.6.4). Dans cette couche, chaque pair a un débit qui lui est propre et ce débit est partagé équitablement entre les différents agents en exécution sur la couche agents. C'est-à-dire que si un pair héberge plusieurs agents en déplacement, chacun de ces agents se voit affecter la même quantité de bande passante par cycle de simulation. Les agents sont hébergés par des pairs SCAMP qui sont responsables de leur transfert. Lorsqu'un pair tombe en panne, les agents qu'il héberge sont perdus. De fait, nous ne considérons pas les fautes temporaires dans notre implémentation. Ce choix vient principalement du fait que les données étant mobiles, relancer des agents perdus après un certain temps n'a pas de sens puisque ces

derniers ont une très forte chance d'être isolés de leur nuée. Le simulateur ne gère donc que des fautes permanentes. L'exécution des agents est régie par l'ordonnanceur d'oRis qui choisit quel agent activer. Chaque agent, lorsqu'il est activé décide, avec une certaine probabilité, d'exécuter l'algorithme de flocking. Par conséquent, les agents ne décident pas forcément de se déplacer à chaque tour. Cette variable sert à paramétrer la réactivité d'un agent et la vitesse d'une nuée. On note que, durant son déplacement, un agent ne peut pas être activé et ne peut donc pas prendre de décisions.

Paramètres par défaut

Étant donné que la simulation du modèle de flocking se déroule dans le temps, il est impératif de spécifier un certain nombre de variables telles que la durée d'une simulation (le nombre de cycles d'une simulation) ou encore la durée d'un cycle en temps simulé. Par défaut, notre simulateur est paramétré comme suit :

- taille d'un agent (fragment inclus) : 64 Mo ;
- durée simulée dans un cycle : 60 secondes ;
- nombre de cycles d'une simulation : 30000 ($\approx$ 20 jours de temps simulé) ;
- bande passante moyenne : entre 64kb/s et 1Mb/s ;
- probabilité de prise de décision d'un agent : 0.1 ;
- nombre de répétitions de chaque expérience : 70.

Certaines des ces valeurs ont été choisies pour être plausibles dans un cadre où les utilisateurs du système seraient des usagers d'internet en utilisant les débits de l'ADSL, ainsi que des tailles d'agents relativement raisonnables par rapport à l'ordre de grandeur des fichiers qui transitent actuellement sur internet. Les autres valeurs ont, quant à elles, été choisies plus arbitrairement dans un soucis de trouver un compromis entre une simulation ayant un temps simulé suffisamment long tout en gardant un temps d'exécution qui reste raisonnable.

4.3.2 Environnement d'expérimentation en milieu réel

L'expérimentation d'un prototype est notre second moyen pour vérifier que le comportement en flocking émerge de l'interaction de nos agents. Ces résultats vont donc nous permettre de valider expérimentalement nos simulations et par la même occasion la façon dont nous utilisons le simulateur. En effet, nous verrons que dans les deux cas, les résultats concordent. Notre prototype repose sur la couche réseau pair-à-pair (n,c)-SCAMP et sur la plateforme JavAct, que nous avons modifiée pour nos besoins (cf. Sec. 4.2.1). Cette architecture a ensuite été déployée sur un réseau pair-à-pair réel de 100 machines. Dans ce premier prototype, les agents se déplacent perpétuellement dans le réseau en appliquant l'algorithme 17 (cf. Sec. 4.2.4).

4.4 Évaluation du déplacement en nuées

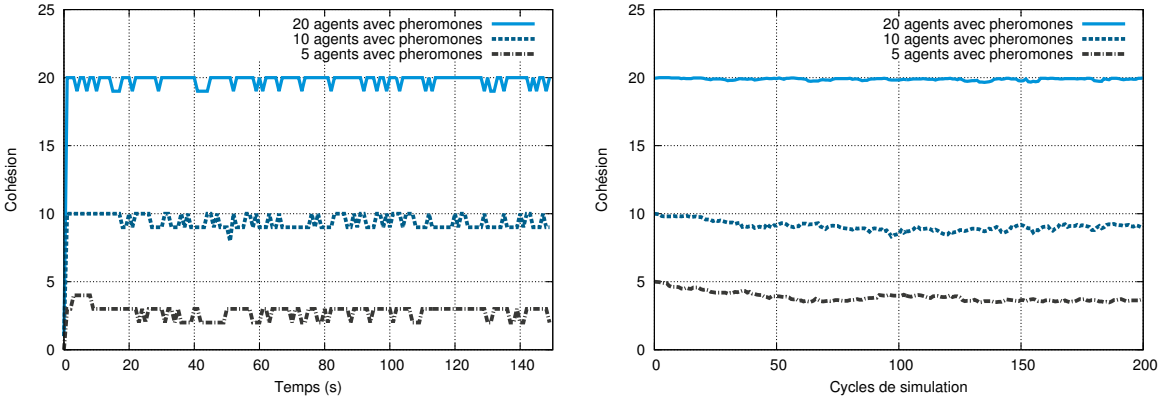
Les premières évaluations du modèle que nous avons réalisées, ont porté sur la mesure de la cohésion et sur la mesure de la couverture du réseau par une nuée. La première mesure consiste à calculer périodiquement la valeur de cohésion (cf. Def. 28, Sec. 4.2.4), extraite à partir du graphe de la nuée. Cette mesure permet d'évaluer si les éléments des nuées se déplacent bel et bien en groupe. La seconde mesure, la couverture, compte le nombre de pairs que la nuée a visités et en quelles proportions. Cette évaluation a pour objectif de donner des éléments de réponse sur la nature de la mobilité de la nuée. La nuée couvre-t-elle bien tout le réseau ? Ou, au contraire, fait-elle du sur place ? Quel est l'impact du mécanisme de phéromones sur les déplacements de la nuée ? Dans un souci de montrer la convergence des résultats entre simulation et implémentation réelle, nous présentons conjointement, dans la suite de cette section, les mesures obtenues à l'issue des simulations ainsi que celles obtenues sur l'exécution du prototype.

4.4.1 Mesure de la cohésion

La cohésion d'une nuée est donnée par la taille de sa composante connexe la plus grande. Par conséquent, plus la mesure de cohésion d'une nuée est proche de son nombre de fragments, plus sa cohésion est préservée. Si nous effectuons un parallèle avec le déplacement en nuée des oiseaux, il arrive que certains oiseaux décrochent temporairement de la nuée ou que la nuée se sépare en sous-nuées. Ce phénomène est intrinsèque au flocking et il est donc normal de l'observer au niveau de notre modélisation multi-agents. Cependant, au regard de notre problématique de déplacement de groupe pour une supervision décentralisée, un bon comportement de flocking, est un comportement qui minimise les ruptures de cohésion.

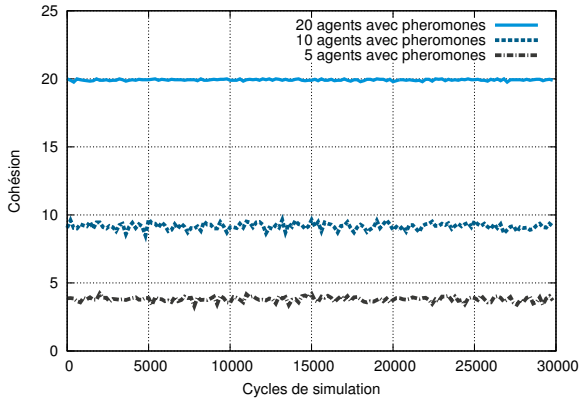
Validation comportementale

La première expérience que nous proposons consiste à faire évoluer des nuées de 5, 10 et 20 agents dans un réseau (100,3)-SCAMP et à mesurer leur cohésion dans un environnement réel ainsi que dans un environnement simulé. Les résultats de cette expérience sont présentés à la Fig. 4.7. Dans ces graphiques, chaque courbe représente l'évolution de la cohésion d'un type de nuée au cours du temps. Dans l'évaluation du prototype (cf. Fig. 4.7(a)), la cohésion, pour les nuées de taille 10 et 20, est quasiment préservée pendant toute la durée de l'expérience. En effet, ces nuées ne perdent en moyenne qu'un seul fragment durant leurs déplacements et ces pertes sont temporaires. Par contre, la nuée de taille 5 a du mal à préserver sa cohésion puisque sa valeur de cohésion moyenne est approximativement égale à 3. Ces résultats montrent qu'il existe des nuées pour lesquelles la cohésion a été préservée durant les déplacements et que par conséquent notre algorithme de flocking est valide pour ces configurations. Ce constat soulève néanmoins la question de la taille minimale d'une nuée nécessaire et suffisante pour que le flocking émerge. Nous donnons par la suite un premier élément de réponse à cette question, en montrant que la cohésion d'une nuée est fortement impactée par les paramètres du réseau dans lequel elle évolue.



(a) Cohésion de nuées expérimentées sur le prototype.

(b) Cohésion de nuées simulées dans les mêmes conditions que le prototype (1 cycle = 1s).



(c) Cohésion de nuées simulées sur les paramètres par défaut (1cycle = 60s) et sur une période plus longue.

FIGURE 4.7 – Mesures de cohésions sur des nuées réelles et simulées dans un réseau (100,3)-SCAMP.

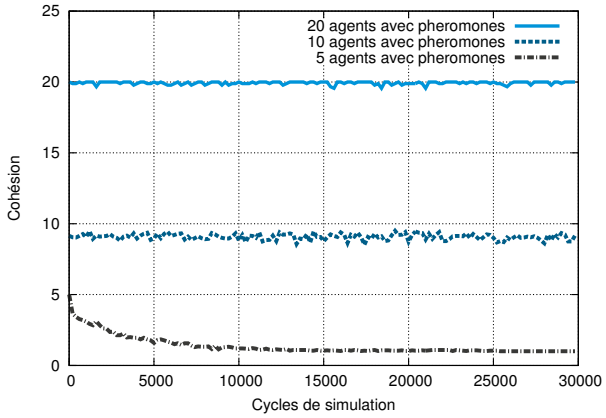


FIGURE 4.8 – Mesure de la cohésion de nuées simulées dans un réseau (100,3)-SCAMP. Les agents de ces nuées restent immobiles tant qu'ils sont isolés. On constate qu'avec cette stratégie, les nuées de taille insuffisante se retrouvent éclatées.

La Fig. 4.7(b), présente les résultats pour la même expérience, mais cette fois-ci réalisée dans notre simulateur. Sur ces simulations, les paramètres sont exactement les mêmes que ceux du prototype. C'est à dire que les agents ont une prise de décision de l'ordre de la seconde (i.e., 1 cycle de simulation = 1s) et la taille d'un agent simulé est égale à la taille d'un agent JavAct. On constate que les valeurs obtenues sont sensiblement les mêmes [13], sauf peut-être pour la nuée de 5 agents pour laquelle la valeur moyenne de cohésion n'est plus 3 mais 4. Ces résultats similaires pour les deux modes d'expérimentation valident, dans une certaine mesure, notre simulateur sur ces paramètres. La Fig. 4.7(c) présente, quant à elle, les résultats de simulations effectuées avec les paramètres par défaut énoncés plus haut. C'est-à-dire que la période simulée est maintenant de 30000 cycles et que le délai entre deux décisions pour un agent varie entre 1min et 10min. L'instance de réseau et la taille des nuées déployées restant, par ailleurs, les mêmes que précédemment. Nous constatons que les valeurs de cohésions obtenues sont identiques à celles des deux graphiques précédents. Ceci montre que le comportement des nuées est stable dans le temps et qu'il n'est pas impacté par la réactivité des agents. Les résultats présentés à la Fig. 4.8 donnent un élément de réponse sur la stratégie à utiliser lorsqu'un fragment est isolé. En effet, quel comportement un agent fragment doit-t-il adopter lorsqu'il est perdu ? (i.e., lorsqu'il ne perçoit aucun autre agent de sa nuée dans ses voisinages). Dans cette expérience, les agents isolés arrêtent leur déplacement tant qu'ils restent isolés (dans l'expérience précédente, ces agents effectuaient un déplacement aléatoire sur un pair libre de leur voisinage jusqu'à retrouver leur nuée). On constate que, pour les nuées de taille 10 et 20, les résultats sont les mêmes que ceux de l'expérience précédente. Le comportement des agents isolés dans ce cas n'a pas d'influence significative sur le comportement global du système. Par contre, pour la nuée de taille 5, la différence est notable puisque dans cette stratégie, où les agents restent immobiles quand ils

13. Les simulations sont reproduites 70 fois et c'est la valeur moyenne, pour chaque cycle, qui est présentée dans les graphiques.

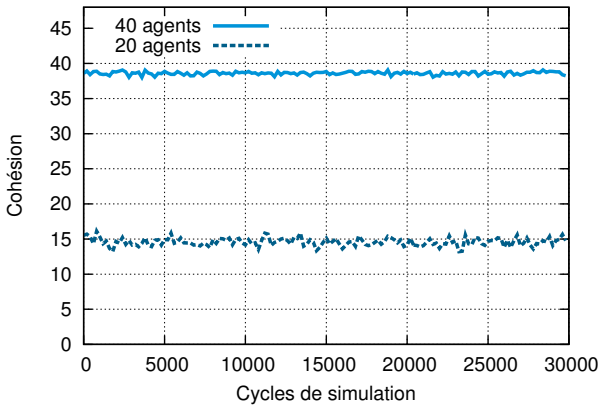


FIGURE 4.9 – Mise en évidence de l'impact de la taille d'une nuée sur sa cohésion. Mesure de la cohésion de deux nuées de taille 20 et 40, simulées dans un réseau (600,4)-SCAMP.

sont isolés, la nuée se retrouve complètement éclatée (i.e., sa cohésion est égale à 1) au bout d'un certain nombre de cycles. On constate donc qu'une nuée de taille suffisante présente une certaine robustesse aux perturbations ainsi qu'aux comportements non souhaités de certains de ses agents.

Paramètres ayant un impact sur la cohésion

Il semble, au regard de ces premières mesures de cohésion, que plus la nuée présente une faible taille et plus le comportement de flocking se dégrade, à taille de réseau constant. Pour confirmer cette intuition, nous proposons de changer les paramètres de la simulation en nous plaçant dans un (600,4)-SCAMP et en comparant la valeur de cohésion d'une nuée de 20 fragments avec celle d'une nuée de 40 fragments. Les résultats de cette expérience sont présentés à la Fig. 4.9. On constate que la nuée de 40 fragments a une cohésion moyenne de 39 fragments alors que la nuée de 20 fragments a une cohésion moyenne de 15. Par conséquent cette expérience confirme notre hypothèse, qu'à taille de réseau constant, plus une nuée possède un nombre de fragments important et plus sa cohésion est préservée. Ces expériences mettent en évidence le fait qu'il semble exister une taille de nuée pour laquelle la cohésion est toujours préservée à 100% (ce point sera détaillé dans le chapitre 5).

Si l'on compare les courbes des nuées de 20 fragments des Fig. 4.7(c) et Fig. 4.9, on constate également qu'à taille de nuée constante, la structure du réseau joue un rôle sur la cohésion. En effet, il semble que plus le nombre de pairs du réseau augmente et plus la cohésion est difficile à préserver. La taille du réseau jouant un rôle manifeste sur la valeur de cohésion d'une nuée, il est tout à fait légitime de se demander si le degré moyen du réseau a également un impact sur cette cohésion. Nous proposons donc de mesurer la cohésion d'une nuée évoluant dans deux réseaux de même taille mais ayant chacun un degré différent. La Fig. 4.10 présente les résultats de cette expérience avec une nuée de 45 fragments qui évolue dans un réseau (4000,4)-SCAMP puis dans un réseau (4000,20)-SCAMP. Premièrement, on constate que les deux nuées n'ont

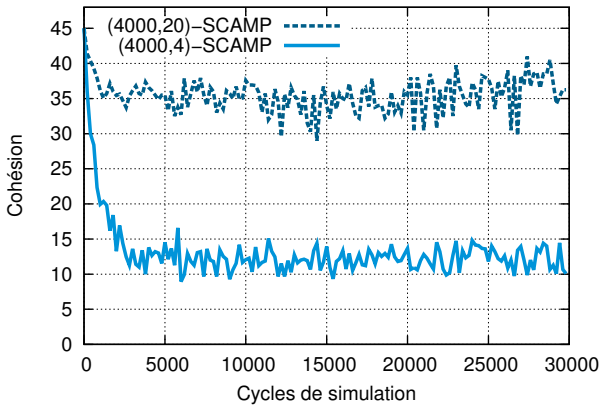


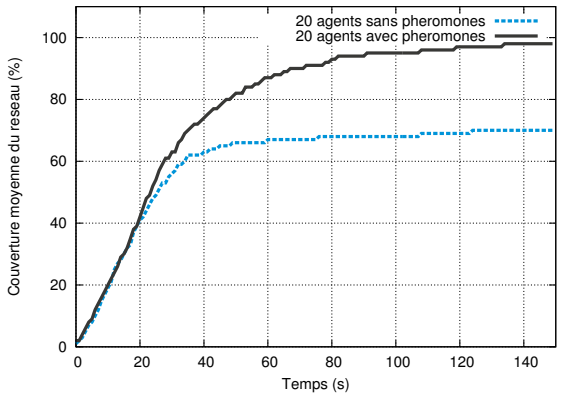
FIGURE 4.10 – Mise en évidence de l'impact de la structure du réseau sur la cohésion d'une nuée de 45 fragments.

pas une cohésion maximale. Ceci montre, en accord avec ce que nous venons de dire plus haut, qu'une taille de nuée de 45 fragments ne semble pas avoir une taille adaptée à ces paramètres réseaux. Deuxièmement, on remarque que la cohésion dans le réseau (4000,20)-SCAMP est bien mieux préservée que dans le réseau (4000,4)-SCAMP, nous permettant de formuler l'hypothèse que plus le degré du réseau augmente et moins la cohésion d'une nuée est impactée, à taille de réseau constante.

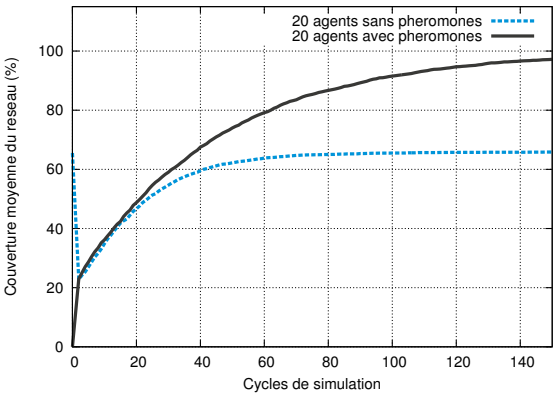
4.4.2 Mesure de la couverture

L'étude de la cohésion nous a permis de montrer que l'algorithme de flocking préserve la cohésion sous certaines conditions. Mais cette étude de la cohésion n'apporte aucune information sur la nature de la mobilité d'une nuée. Effectivement, elle ne permet pas de dire si une nuée fait du sur place ou si, au contraire, elle parcourt l'intégralité du réseau. Pour mesurer cette propriété de mobilité d'une nuée, nous proposons la mesure de couverture réseau. Cette mesure consiste à compter, à chaque instant, la proportion du réseau qui a été visitée par la nuée. Elle permet notamment de mettre en évidence et d'évaluer l'utilité du mécanisme de dépôt de phéromones (cf. Sec. 4.2.5) sur les capacités exploratoires d'une nuée.

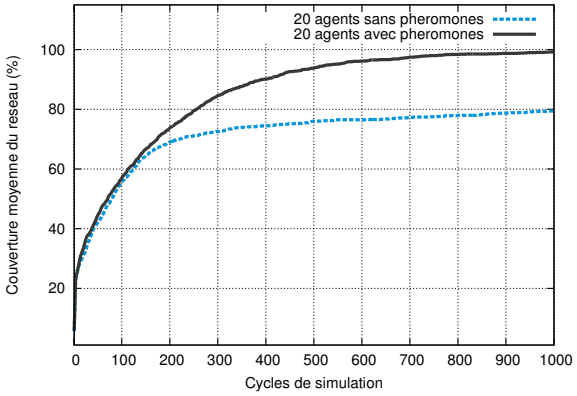
Nous mesurons la couverture réseau au moyen de trois expériences faisant évoluer des nuées de 20 fragments dans un (100,3)-SCAMP. La première est réalisée dans un réseau réel, sur le prototype et les deux autres sont réalisées sur le simulateur. Le résultat de ces expériences est présenté à la Fig. 4.11. Dans chacune d'elles, la couverture est mesurée sur une nuée utilisant le dépôt de phéromones et sur une nuée n'en tenant pas compte. On constate, dans un premier temps, que, tout comme pour l'expérience sur la cohésion, les résultats de simulation (cf. Fig. 4.11(b)) sont proches des résultats obtenus sur les expérimentations réelles du prototype (cf. Fig. 4.11(a)), venant valider notre simulateur à un second niveau. Ensuite, on constate que l'utilisation de phéromones permet à une nuée de parcourir son espace plus rapidement et en intégralité. *a contrario*, sans mécanisme de phéromones, environ 20% des pairs n'ont pas été



(a) Couverture réseau de nuées expérimentées sur le prototype.



(b) Couverture réseau de nuées simulées dans les mêmes conditions que le prototype (1 cycle = 1s).



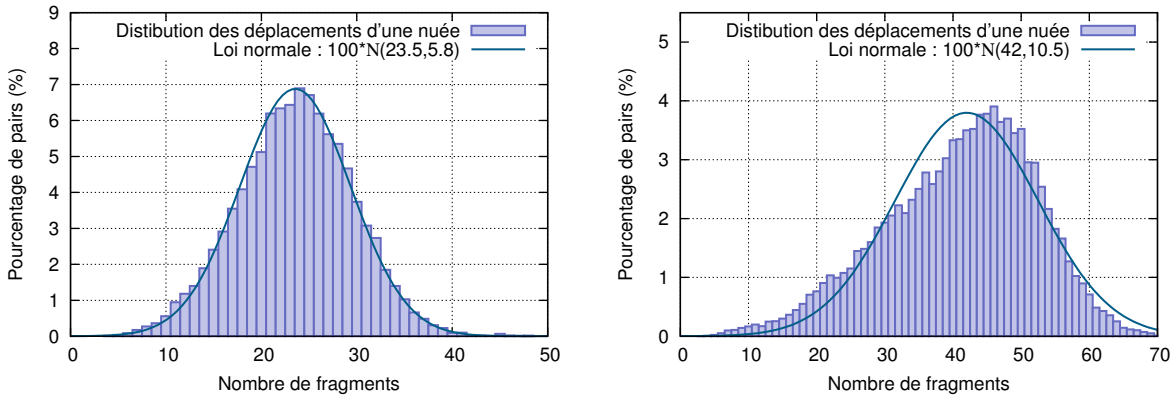
(c) Couverture réseau de nuées simulées avec les paramètres par défaut (1 cycle = 60s) et sur une période plus longue.

FIGURE 4.11 – Premières mesures de la couverture réseau sur des nuées réelles et simulées dans un réseau (100,3)-SCAMP.

visités par les nuées à la fin de l'expérience. Par conséquent, dans le reste de ce document, les nuées utiliseront implicitement des phéromones pour effectuer leurs déplacements à moins qu'il ne soit explicitement mentionné le contraire. Finalement, on constate à la Fig. 4.11(c), que, lorsque la réactivité des agents est diminuée (i.e., l'écart entre deux décisions est compris entre 1min et 10min), le temps nécessaire à une nuée pour couvrir l'espace s'en retrouve logiquement augmenté en conséquence.

4.4.3 Distribution des déplacements

La couverture réseau donne de l'information sur la proportion du réseau qui a été parcourue par une nuée mais ne donne pas d'information sur la répartition des ses déplacements. La distribution des déplacements d'une nuée permet de savoir si sa charge est bien répartie sur



(a) Évolution dans un réseau (1000,10)-SCAMP. Interpolation de cette distribution par la loi normale : $100 \cdot \mathcal{N}(23.5,5.8)$.

(b) Évolution dans un réseau (600,20)-SCAMP. Interpolation de cette distribution par la loi normale : $100 \cdot \mathcal{N}(42,10.5)$.

FIGURE 4.12 – Distribution des déplacements d'une nuée de 45 fragments dans deux instances de réseaux.

les pairs ou si, au contraire, seuls quelques pairs supportent toute la charge à eux seuls. Cette distribution s'obtient d'abord en comptant, pour chaque pair x, le nombre de fragments f qu'il a hébergés pendant la simulation. On regroupe ensuite, pour chaque valeur de f, le nombre de pairs x ayant hébergés f fragments. Ces valeurs sont ensuite normalisées pour donner une distribution de probabilité. La Fig. 4.12(a) présente le résultat de simulations effectuées sur un réseau (1000,10)-SCAMP avec une nuée de 45 fragments. On y observe la distribution des déplacements de la nuée. Ce graphique se lit comme suit : « a% de pairs ont hébergés b fragments », avec a se lisant en ordonnées et b en abscisses. Cette distribution présente un critère de normalité et peut être interpolée par la loi normale $\mathcal{N}(23.5,5.8)$. Étant donné la normalité de cette distribution, nous pouvons conclure qu'une majorité de pairs ont hébergé la nuée. À titre d'exemple, 90% des pairs ont hébergés entre 16 et 31 fragments. Le caractère normal de cette distribution exhibe le fait que le mode exploratoire d'une nuée est progressif et qu'elle fait preuve d'une certaine rigidité : elle possède un noyau d'agents qui se déplace progressivement. Ce déplacement n'est pas sans rappeler le flocking des oiseaux, dans lequel une nuée peut s'étirer ou bien se rétrécir, mais dont les déplacements semblent déterminés par une conjonction majoritaire de choix abondant dans le même sens. En effet, quelques oiseaux (et nos agents) en périphérie de la nuée n'ont pas le pouvoir à eux seuls d'attirer la nuée vers eux. Par analogie, on trouve donc bien des zones dans le réseau qui sont moins visitées que d'autres par une nuée.

La Fig. 4.12(b) présente les résultats d'une expérience similaire mais effectuée dans un réseau plus petit : (600,20)-SCAMP. La normalité de la distribution est moins nette puisque son interpolation par $\mathcal{N}(42,10.5)$ génère une erreur beaucoup plus importante que sur l'expérience précédente. À la lecture de ce graphique nous constatons que la distribution est décalée vers la droite et plus aplatie. C'est-à-dire que, d'une manière générale, les pairs ont vu passer plus d'agents et que les écarts de valeurs entre ces pairs ont augmentés (distribution plus large).

L'équilibrage de charge dans ce réseau est par conséquent différent de celui de l'expérience précédente mais reste gaussien.

Nous clôturons cette section d'évaluation de l'algorithme de flocking en rappelant que notre transposition des règles de Reynolds dans un réseau préserve non seulement la cohésion des nuées durant leurs déplacements (à condition que certaines conditions, mises en évidence, soient réunies) mais également que ces déplacements de groupe se font en parcourant l'intégralité du réseau (lorsque des phéromones sont utilisées) et que la distribution de ces déplacements suit approximativement une loi normale.

4.5 Recherche d'une nuée

Nous abordons maintenant [14] la question de la recherche de nuées, que nous avons déjà évoquée plus haut. Dans l'état actuel des choses, nous sommes capables de fragmenter une donnée avec un code d'effacement et de plonger ces fragments dans un environnement de telle sorte qu'un déplacement de groupe en nuée émerge. Ces nuées doivent pouvoir être trouvées et récupérées par leur propriétaire. Il convient donc d'étudier le comportement des algorithmes de recherche lorsqu'ils sont confrontés à des données mobiles. Pour simplifier les traitements, nous faisons l'hypothèse que la cohésion d'une nuée est préservée de sorte que trouver un fragment d'une nuée revient à trouver l'intégralité de cette nuée (i.e., lorsqu'un fragment est trouvé, la requête de recherche est transférée par inondation à ses agents voisins). Pour trouver une nuée, nous proposons deux algorithmes décentralisés de recherche qui sont exécutés par un agent mobile. Le premier algorithme est une marche aléatoire et le second est une marche aléatoire bio-inspirée qui repose sur des dépôts de phéromones. Dans cette dernière, chaque agent de recherche a la possibilité de déposer dans le réseau des phéromones qui lui sont propres [15]. La recherche par inondation n'a pas été retenue étant donnée qu'elle est un frein au passage à l'échelle des systèmes (cf. Sec. 1.4.1).

4.5.1 Recherche par marche aléatoire

La marche aléatoire exécutée par l'agent repose sur la création préalable d'une chaîne de Markov qu'il utilisera dans ses décisions de déplacement. La construction de cette chaîne est déduite du réseau et supportée par ce dernier. Dans cette modélisation, nous faisons correspondre les pairs aux états S de la chaîne et les voisinages pair-à-pair aux transitions P_{xy} entre ces états. On suppose, en accord avec les propriétés de SCAMP, que le réseau est connexe et que son degré moyen est égal à $(c+1)\log(n)$. Un agent de recherche peut donc potentiellement atteindre l'ensemble des pairs à partir de son état courant. Soit $dep_x = V_x \cup \{x\}$, l'ensemble des déplacements possibles pour un agent de recherche s'exécutant sur le pair x. La probabilité

14. Cette section a fait l'objet des contributions de la publication [Pommier *et al.*, 2011].

15. Attention, ces phéromones sont différentes de celles utilisées pour le déplacement des nuées et il n'y a aucune influence entre elles.

de transition P_{xy} entre deux états x et y est donnée par :

$$P_{xy} = \begin{cases} \dfrac{1}{|dep_x|} = \dfrac{1}{(c+1)\log(n)+1} & \text{si } y \in dep_x \\ 0 & \text{Sinon} \end{cases} \tag{4.1}$$

avec

$$\forall x \in S, \sum_{y \in dep_x} P_{xy} = 1 \tag{4.2}$$

La distribution est uniforme et chaque état peut être atteint avec la même probabilité. SCAMP garantit que la propriété de degré $(c+1)\log(n)$ est toujours vérifiée. Sous cette hypothèse, les transitions de la chaîne sont symétriques et la distribution est stationnaire. Les déplacements d'un agent hébergé sur un pair x, utilisant cette marche aléatoire pour trouver une nuée *doc*, sont décrits, çi-après, dans l'algorithme 18. La première étape de cet algorithme vérifie si un fragment de la nuée recherchée est présent sur le pair courant. Si un tel fragment f_{doc} est trouvé, alors, l'agent de recherche transmet à f_{doc} l'adresse de l'initiateur de la recherche. La propagation de la requête de recherche dans la nuée est laissée à la charge de f_{doc}. Si par contre le pair courant n'héberge pas de fragments de *doc* alors l'agent de recherche choisit son déplacement en fonction de la probabilité de transition P_{xy} avec $y \in dep_x$, et ainsi de suite.

Algorithme 18 : Agent de recherche hébergé sur un pair x et suivant une marche aléatoire.

Entrées : *doc* : identifiant de la nuée recherchée, *émetteur* : l'adresse de l'initiateur de la requête, V_x : voisinage du pair x

Sorties : y : un pair candidat au déplacement suivant

```
// Vérifier si un fragment de doc est présent
```

si *trouvé*(*doc*) **alors**
  Remettre *émetteur* à f_{doc};
sinon
  $dep_x \leftarrow V_x + \{x\}$;
  pour chaque $y \in dep_x$ **faire**
    $P_{xy} = \dfrac{1}{|dep_x|}$;
  Choisir $y \in dep_x$ avec la pobabilité P_{xy};
  Se déplacer sur y;

4.5.2 Recherche guidée par des phéromones

La marche aléatoire est un processus sans mémoire. Il est donc possible, en suivant cette méthode, qu'un agent de recherche visite à plusieurs reprises le même pair dans un laps de temps très court. Ce phénomène peut ralentir la couverture réseau de l'agent et ralentir la recherche. Par conséquent, nous proposons un second algorithme, basé sur la stigmergie, dans lequel les agents de recherche déposent une certaine quantité de phéromones sur les pairs pour

marquer leurs déplacements. Le niveau de phéromones de recherche pour une nuée *doc* sur un pair x est noté $\rho_{doc,x}$. Ce mécanisme de phéromones peut supporter la recherche d'une même nuée par plusieurs agents différents, partageant les phéromones propres à cette recherche et non des phéromones propres à l'agent. Avec cet algorithme, nous souhaitons conserver le caractère aléatoire de la recherche tout en marquant les zones du réseau qui ont déjà été visitées. Pour se diriger, un agent de recherche sélectionne un pair de son voisinage ayant le plus faible niveau de phéromones. La nuée est un système mobile ; par conséquent, il faut donner à un agent la possibilité de revisiter des zones qu'il a déjà exploré. Cette possibilité est obtenue en évaporant périodiquement les phéromones sur chaque pair. Pour ce faire, chaque pair x fixe son taux d'évaporation de phéromones comme suit :

$$evapo_{doc,x} = \frac{\rho_{doc,x}}{n} \tag{4.3}$$

avec n le nombre de pairs du réseau que x déduit de son degré moyen : $|V_x| = (c+1)\log(n)$. x va ensuite périodiquement effectuer l'opération $\rho_{doc,x} := \rho_{doc,x} - evapo_x$. La justification d'un tel niveau d'évaporation qui est donnée dans [Pommier, 2010] et [Pommier *et al.*, 2011], stipule qu'au bout de n itérations de l'évaporation, on a $evapo_{doc,x} = 0$. Cette propriété tient, sous l'hypothèse que la valeur ρ_x du pair considéré n'a pas augmenté après son premier dépôt. Or, il peut arriver qu'un agent doive rebrousser chemin et, par conséquent, modifier la quantité initialement déposée sur un pair. De plus, dans un cadre multi-agents, il est tout à fait possible qu'un agent n'ait pas d'autre choix que d'emprunter un lien déjà emprunté par un autre agent. La propriété $evapo_{doc,x} = 0$ au bout de n itérations n'est donc pas garantie. Les expériences que nous menons par la suite, sur la recherche de nuée, sont réalisées pour montrer la différence entre une approche avec marquage et une approche sans marquage. Nous n'avons, en revanche, mené aucune expérience sur le paramètre d'évaporation à proprement parler ; son impact reste donc à évaluer. Dans cet algorithme de recherche (cf. algorithme 19), lorsque plusieurs pairs voisins de l'agent de recherche ont un niveau de phéromones minimum, le choix du pair candidat au déplacement est effectué de manière aléatoire parmi ces pairs. La fonction `demander(`ρ_y`,y)` demande la valeur ρ_y au pair y et la retourne.

4.5.3 Évaluations des algorithmes de recherche

L'évaluation de ces deux algorithmes a été réalisée à l'aide de simulations sur un (400,4)-SCAMP. Elle repose sur deux expériences : la première consiste à comparer les deux algorithmes en terme de taux de réussite et de rapidité des recherches. La seconde s'intéresse à la question de l'impact de la vitesse de déplacement d'une nuée ainsi que de sa taille sur la rapidité des recherches. Dans chacun des cas, les deux algorithmes ont été comparés sur des valeurs moyennes issues de la reproduction de plusieurs simulations.

Algorithme 19 : Agent de recherche déposant des phéromones sur un pair x

Entrées : *doc* : identifiant de la nuée recherchée, *émetteur* : l'adresse de l'initiateur de la requête, V_x : voisinage du pair x

Sorties : Un pair y candidat pour le déplacement suivant

incrémenter($\rho_{doc,x}$);

`// Vérifier si un fragment de` *doc* `est présent`

si *trouvé*(*doc*) **alors**

  Remettre *émetteur* à f_{doc};

sinon

  $dep_x \leftarrow V_x + \{x\}$;

  $pheros \leftarrow \emptyset$;

  pour chaque $p \in dep_x$ **faire**

    $pheros \leftarrow pheros \cup demander(\rho_{doc,p}, p)$;

  $\rho_{doc,y} \leftarrow \min(pheros)$;

  Se déplacer sur y;

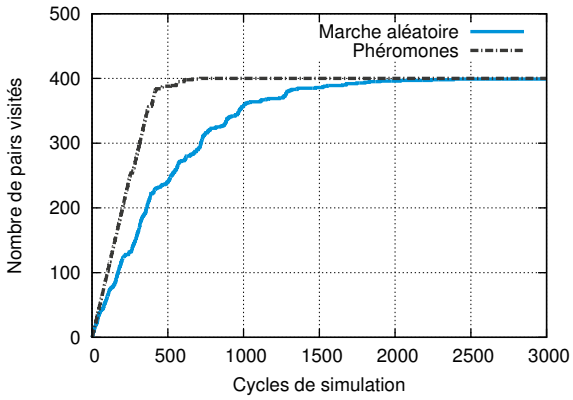
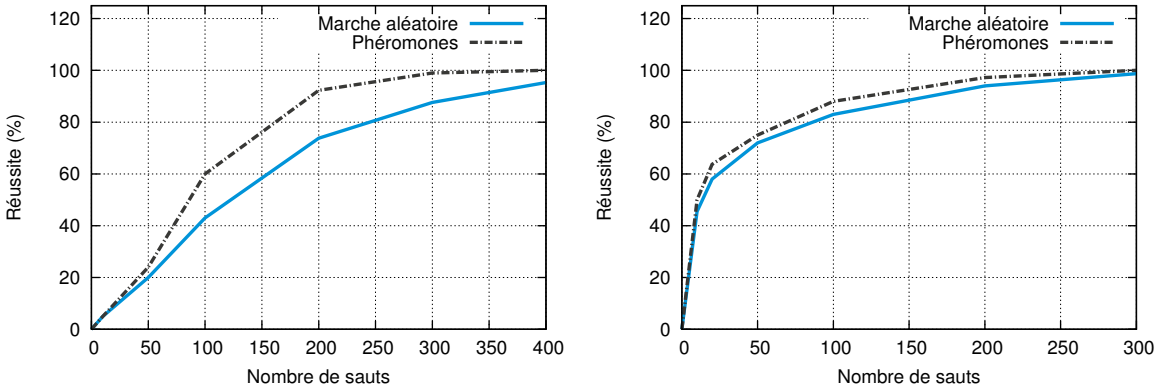


FIGURE 4.13 – Couverture du réseau par les agents de recherche.

Couverture réseau et mesure des succès des recherches

La première expérience que nous proposons consiste à mesurer la couverture réseau des deux types d'agents de recherche. Comme pour la mesure de couverture réseau d'une nuée (cf. Sec. 4.4.2), cette mesure consiste à retourner, à chaque instant, le nombre de pairs distincts qu'un agent a visité. La Fig. 4.13 présente les résultats de couverture pour les deux types de recherche. Ces résultats confirment que l'utilisation de phéromones accélère bien la vitesse exploratoire d'un agent de recherche. En effet, l'agent qui effectue une marche aléatoire a couvert l'intégralité du réseau en ≈ 2000 cycles alors que l'agent bio-inspiré l'a couvert en seulement ≈ 600 cycles.

Cette mesure de la couverture ne nous permet pas de statuer sur le résultat des recherches. Plusieurs questions se posent. Est-ce que ces agents de recherche trouvent bien les nuées ? Y a t'il une recherche plus efficace que l'autre ? Peut-on borner le nombre de sauts nécessaires à



(a) Nuée de taille 15.

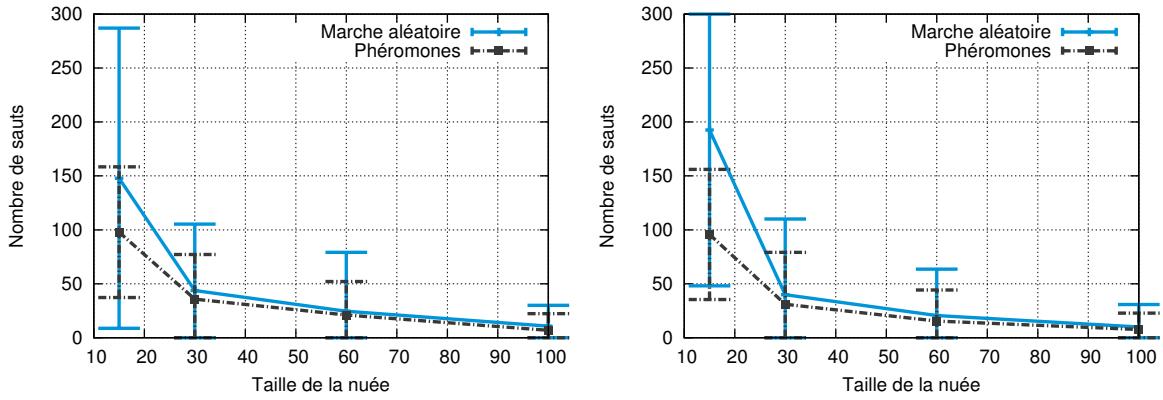
(b) Nuée de taille 30.

FIGURE 4.14 – Performances de la recherche.

un agent de recherche pour trouver une nuée ? Pour répondre à ces questions, nous proposons de fixer le nombre de déplacements maximum alloués aux agents de recherche et de mesurer le pourcentage de réussite de la recherche pour ce nombre de déplacements. Ces simulations on été réalisées pour les deux types de recherche, sur des nuées de taille 15 et 30, en faisant varier le nombre de déplacements maximum alloué à chaque agent. Les résultats de ces expériences sont présentés à la Fig. 4.14. Ces deux graphiques affichent des courbes donnant le taux de réussite des deux types de recherche en fonction d'un nombre de sauts fixé. On constate tout d'abord que le taux de réussite est égal à 100%, pour les deux tailles de nuées et les deux types de recherche, lorsque les agents ont droit à un nombre suffisant de déplacements. Ce qui veut dire que si on laisse suffisamment de temps au marcheur, il est capable de retrouver une nuée. On remarque que, pour la nuée de taille 15, la recherche par phéromones a un meilleur taux de réussite que la recherche par marche aléatoire pour un même nombre de sauts donné. Cet écart se réduit pour la nuée de taille 30 et la différence entre les deux algorithmes de recherche est marginale dans ce cas. Nous pouvons conclure que, lorsque la nuée est de petite taille, une recherche par phéromones pourra être préférée puisqu'elle nécessitera moins de sauts qu'une marche aléatoire pour trouver une nuée. On remarque également que pour un nombre de sauts bornés à 150 (correspondant à 37.5% de la taille du réseau), la recherche par phéromones a un taux de réussite compris entre 80% et 95%. Par conséquent, il est possible de réduire significativement le nombre de sauts d'un marcheur si quelques échecs de recherche peuvent être tolérés.

Impact de la taille et de la vitesse d'une nuée sur la recherche

Nous observons, sur les graphiques de la Fig. 4.15, le nombre de sauts nécessaires à un agent de recherche pour localiser une nuée en fonction de la vitesse et de la taille de cette dernière. Chaque courbe affiche la valeur moyenne et l'écart type du nombre de sauts nécessaires à un agent de recherche pour retrouver des nuées de tailles différentes. Ces simulations ont été réalisées pour des nuées de taille 15, 30, 60 et 100, se déplaçant aux vitesses 1, $\frac{1}{2}$, $\frac{1}{5}$, et $\frac{1}{10}$. Une vitesse de



(a) Vitesse 1.

(b) Vitesse $\frac{1}{2}$.

(c) Vitesse $\frac{1}{5}$.

(d) Vitesse $\frac{1}{10}$.

FIGURE 4.15 – Impact de la taille d'une nuée sur le nombre de sauts nécessaires pour la localiser, pour différentes vitesses de nuée.

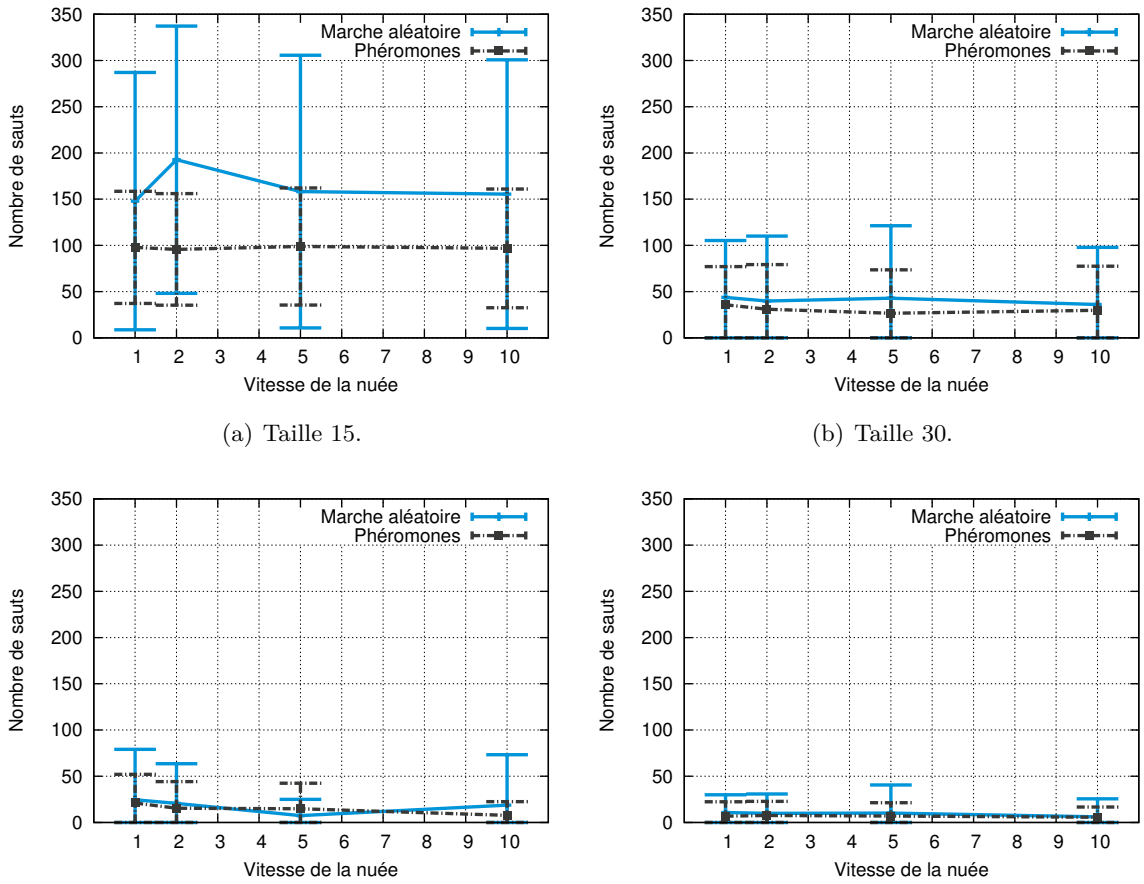
$\frac{1}{2}$ signifie simplement que chaque agent de la nuée se déplace tous les 2 cycles de simulation. Les agents de recherche, quand à eux, se déplacent systématiquement à chaque cycle de simulation. Ces graphiques sont similaires pour les différentes vitesses. Ceci montre que la vitesse d'une nuée n'a pas d'impact sur le résultat des recherches. Ce nouveau jeu de données confirme également les phénomènes que nous avons mis en évidence à l'expérience précédente :

1. plus la taille d'une nuée augmente et moins le nombre de sauts nécessaires à sa localisation est important ;
2. la recherche par phéromones est plus efficace sur des nuées de petite taille qu'une simple marche aléatoire.

Ces tendances sont également vérifiées à la Fig. 4.16. Sur ces graphiques, on affiche la valeur moyenne et l'écart-type du nombre de sauts nécessaires à un agent de recherche pour localiser des nuées de 15, 30, 60 et 100 fragments en fonction de différentes vitesses de déplacement. On constate à nouveau que la vitesse d'une nuée n'a pas d'incidence significative sur le résultat de la recherche et que la recherche à base de phéromones est plus efficace sur les nuées de petite taille.

4.6 Conclusion

Nous avons détaillé, dans ce chapitre, notre application de stockage décentralisé, dans laquelle la donnée est une entité autonome et mobile. Ce nouveau paradigme, qui rend chaque donnée responsable de son placement, ouvre non seulement la porte à des traitements hétérogènes mais également à une forte adaptabilité des données à l'environnement dans lequel elles évoluent. Cette approche, émergente et fortement bio-inspirée, s'inspire du déplacement des oiseaux en nuées pour permettre aux fragments d'une donnée de conserver une proximité de voisinage malgré leurs déplacements. Nous nous sommes notamment intéressés aux algorithmes qui permettent de transposer cette formulation bio-inspirée dans un réseau pair-à-pair en conservant un contrôle décentralisé et asynchrone. Les résultats des expériences que nous avons menées montrent que, si un certain nombre de paramètres sont réunis, les fragments d'un document forment bien une nuée qui se déplace à travers le réseau en conservant sa cohésion. La dynamique du flocking fait intrinsèquement des ruptures de cohésion. Cet effet est indésirable pour deux raisons : il fausse la supervision des données (nous détaillons ce point au chapitre suivant) et il ralentit la procédure de recherche. En d'autres termes, les ruptures de cohésion perturbent la disponibilité des données qui suivent ce modèle et sont donc des phénomènes néfastes. Nous avons mis en évidence expérimentalement le fait que, le nombre de fragments d'une nuée ainsi que les propriétés du réseau dans lequel cette nuée évolue jouent un rôle sur la valeur de cohésion observée et qu'il semble possible de minimiser cette valeur. La loi régissant le choix de ces paramètres optimaux (i.e., les paramètres qui minimisent les ruptures de cohésion) n'a pas encore été clairement identifiée. Il n'est donc pas possible de les connaître *a priori*. Néanmoins, il semble qu'une possibilité pour qu'une nuée trouve ses paramètres optimaux soit qu'elle explore son environnement et qu'elle



(a) Taille 15.

(b) Taille 30.

(c) Taille 60.

(d) Taille 100.

FIGURE 4.16 – Impact de la vitesse d'une nuée sur le nombre de sauts nécessaires pour la localiser, pour différentes tailles de nuée.

ajuste sa taille en fonction de ses observations. C'est cette approche, reposant sur la capacité d'adaptation d'une nuée, que nous avons retenu dans la suite de cette thèse et qui fait l'objet du prochain chapitre.

L'algorithme 17 a mis en évidence le fait que le flocking peut émerger dans un réseau par un simple raisonnement sur la structure logique qui relie les agents entre eux plutôt que sur une mesure de distance (comme cela pouvait être le cas avec les règles de Reynolds). Cette constatation, lorsqu'elle est mise en perspective avec le flocking réel, soulève un certain nombre de questions : existe t'il un équivalent de cette propriété dans le monde animal ? Les oiseaux évoluant en nuées sont ils dotés de dispositifs cognitifs leur permettant d'établir des relations logiques avec d'autres oiseaux de leur voisinage ? Cette question qui dépasse la portée de ce travail reste néanmoins très intéressante puisqu'il semble qu'il y ait dans le comportement des nuées des repliements de cette dernière sur elle-même, qui semblent impossible à reproduire avec une simple information de distance. Les cris caractéristiques que poussent ces oiseaux ne serviraient ils pas à maintenir un simili de réseau logique ? Contre toute attente, [Ballerini *et al.*, 2008] ont réalisés une étude éthologique sur cette problématique qui semble abonder dans notre sens en réfutant l'hypothèse de relations métriques entre les oiseaux. Ces travaux qui ont été réalisés à l'aide de clichés 3D pris chez plusieurs nuées d'étourneaux (*Sturnus vulgaris*) concluent que les interactions entre les oiseaux d'une nuée sont d'ordre topologiques et que chaque oiseau considère en moyenne 6.5 ± 0.9 voisins dans ses prises de décisions. Cette étude qui réfute donc le modèle de Reynolds considère que l'attraction entre deux oiseaux voisins topologiquement est la même, qu'ils soient éloignés d'1m ou de 5m. Sur la question de l'établissement et du maintien de cette topologie, les auteurs sont convaincus qu'elle est réalisée par la vision ; le nombre limité de voisins permettant au cerveau de chaque oiseau de prendre une décision en une centaine de millisecondes.

Chapitre 5

Robustesse du modèle de flocking

Sommaire

5.1 Ruptures de cohésion et pannes de pairs

Nous avons décrit au Chap. 4 l'architecture de notre application de stockage décentralisé et mobile. Dans cette architecture, chaque document est fragmenté et transformé en une nuée d'agents mobiles se déplaçant dans le réseau. Nous avons constaté, au moyen de plusieurs expériences, que ce déplacement a le comportement attendu et que ces nuées sont par conséquent fonctionnelles. L'objet du présent chapitre est maintenant de traiter de la robustesse de ces nuées, en étudiant les mécanismes à mettre en place pour qu'elles puissent tolérer les fautes [16] .

16. Les éléments majeurs de ce chapitre ont été publiés dans [Romito *et al.*, 2011].

(a) Dans cette nuée de 4 fragments sans rupture de cohésion, il n'y a qu'un seul superviseur dans la composante connexe {1,3,4,9}.

(b) Dans cette nuée de 4 fragments avec rupture de la cohésion, il y a un superviseur dans la composante connexe {1,9} et un superviseur dans la composante connexe {6,7}.

FIGURE 5.1 – Rupture de cohésion d'une nuée et impact sur le mécanisme de supervision.

5.1.1 Impact des ruptures de la cohésion sur la supervision d'une nuée

Les résultats du chapitre précédent ont montré qu'une nuée d'agents mobiles, implémentée selon notre transposition des règles de Reynolds, a une dynamique qui génère des ruptures de cohésion lorsque sa taille n'est pas adaptée au réseau dans lequel elle évolue. Ces ruptures de cohésion posent des problèmes dès lors que la supervision des nuées est décentralisée. En effet, la supervision locale (cf. Sec. 2.3.2) fait l'hypothèse que les fragments d'une donnée sont connexes pour pouvoir les compter périodiquement, de manière décentralisée. Ensuite, à l'issue de ce comptage, la politique de réparation reconstruit si besoin les fragments qui ont été perdus. Dans le cas d'une donnée immobile, cette connexité entre fragments est toujours préservée. Par contre, dans le cas d'une donnée mobile se déplaçant en nuée, cette connexité peut être perdue lors d'une rupture de cohésion. Dans le cas où une nuée se sépare en sous-nuées et rompt la connexité entre ses fragments, le processus de supervision compte une valeur de cohésion par sous-nuée ; chacune de ces valeurs étant inévitablement inférieure au nombre de fragments toujours présents dans le réseau. Chaque superviseur va donc avoir une vision de la nuée qui est partielle et limitée à la composante connexe à laquelle il appartient. Il prendra alors des décisions de réparation qui sont incorrectes si la cohésion est rompue. Nous illustrons ce phénomène à la Fig. 5.1. Dans la Fig. 5.1(a), la nuée {1,3,4,9} a une cohésion qui est totalement préservée. Le mécanisme de supervision local va donc désigner un seul superviseur dans cette composante connexe qui va compter le nombre de fragments disponibles. Dans cet exemple, le superviseur compte 4 fragments disponibles ce qui correspond effectivement à la taille totale de la nuée. Par conséquent, il prendra la bonne décision et ne déclenchera pas de réparation. En revanche, dans l'exemple de la Fig. 5.1(b), la nuée est séparée en deux composantes connexes. Le mécanisme de supervision local doit donc désigner deux superviseurs : un dans chacune des deux composantes. Ces deux superviseurs vont ensuite compter le nombre de fragments de leur composante : ici 2. Si la politique de réparation (cf. Sec 2.3.1) est immédiate ou à seuil avec $r = 2$, alors chacun des deux superviseurs va lancer une réparation en considérant que 2 fragments ont été perdus. La réparation va donc générer deux nuées de 4 fragments chacune alors que pourtant, les 4

fragments initiaux sont toujours disponibles dans le réseau. La disponibilité de la donnée reste préservée mais son coût d'hébergement a doublé inutilement.

5.1.2 Choix du mécanisme de réparation

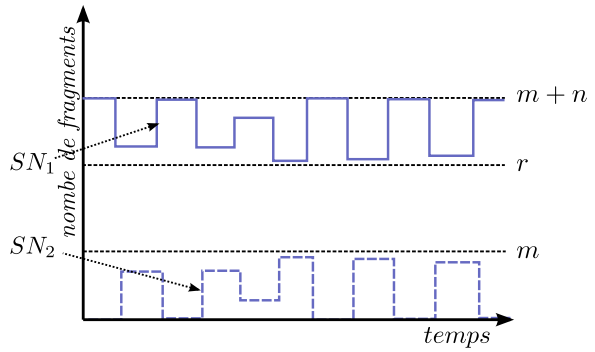
La fluctuation de la cohésion d'une nuée est, d'une manière générale, un phénomène néfaste pour une supervision décentralisée fiable. À la vue des résultats sur la cohésion que nous avons présentés à la Sec. 4.4, il s'avère que la cohésion d'une nuée oscille autour de sa valeur moyenne. Si nous considérons ces oscillations comme étant des fautes temporaires et s'il est possible de déterminer l'amplitude de ces oscillations, alors, à condition que cette amplitude soit suffisamment faible, il y a un avantage à mettre en place une politique de réparation à seuil dont le seuil dépend de cette valeur d'amplitude. Il convient donc, tout d'abord, de caractériser ce que nous entendons par faible et forte amplitude.

Oscillation de faible amplitude

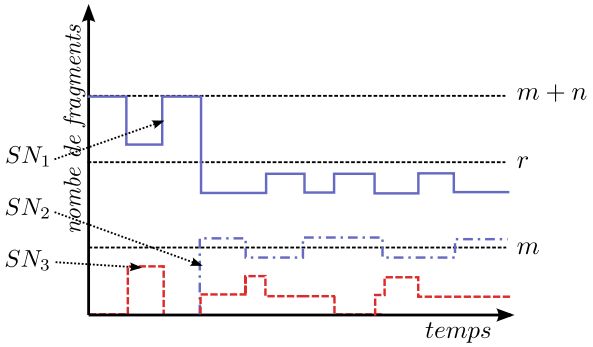
Soit une nuée N_1 fragmentée avec un (m_1,n_1)-code d'effacement. L'ensemble N_1 contient l'ensemble des fragments de la nuée et on a $|N_1| = m_1 + n_1$. On définit ensuite un seuil de réparation r_1 tel que $m_1 \leq r_1 < m_1 + n_1$. Cette nuée va être inévitablement victime de ruptures de cohésion durant son déplacement. Lorsqu'une rupture de cohésion survient, N_1 est séparée en un ensemble de sous-nuées SN_1,SN_2,...,SN_k. S'il existe une et une seule sous-nuée SN_i tel que $r_1 < |SN_i|$ et si pour toute autre sous-nuée SN_j, $i \neq j$, on a $|SN_j| < m_1$, alors la nuée SN_i ne déclenche pas de réparation car le seuil de réparation n'est pas atteint et les autres sous-nuées SN_j ne déclenchent pas non plus de réparation puisqu'elles ne disposent pas de suffisamment de fragments pour se réparer. Nous sommes dans une situation dans laquelle il existe une nuée principale qui est viable autour de laquelle gravitent des sous-nuées non-viables et de petite taille. C'est cette dynamique de nuée que nous caractérisons comme étant de « faible » amplitude. En effet, si une nuée conserve cette dynamique au cours du temps, alors, ses variations de cohésion ne déclenchent pas de réparations inutiles. Ce phénomène de faible oscillation est illustré sur le graphique de la Fig. 5.2(a) dans lequel une nuée se sépare à plusieurs reprises en une sous-nuée principale SN_1 et en une sous-nuée non-viable SN_2. Ces deux sous-nuées ne déclenchent jamais le mécanisme de réparation.

Oscillation de forte amplitude

Si, par contre, la nuée N_1 se sépare en sous-nuées SN_1,SN_2,...,SN_k à la suite d'une rupture de cohésion et qu'il existe au moins une nuée SN_i telle que $m_1 \leq |SN_i| \leq r_1$, alors l'amplitude de la rupture de cohésion a été trop forte pour le seuil donné. Dans ce cas, un certain nombre de réparations va être déclenché et il dépend du nombre de sous-nuées qui ont dépassé r_1. Ce phénomène de forte oscillation de la cohésion est illustré sur le graphique de la Fig. 5.2(b) dans lequel une nuée se sépare à plusieurs reprises en trois sous-nuées SN_1, SN_2 et SN_3. Les sous-nuées SN_1 et SN_2 dépassent toutes les deux le seuil de réparation et sont susceptibles de



(a) Rupture de cohésion de faible amplitude. La sous-nuée SN_1 ne dépasse pas le seuil de réparation r et la sous-nuée SN_2 n'a pas suffisamment de fragments pour se reconstruire. Aucune réparation n'est donc déclenchée.



(b) Rupture de cohésion de forte amplitude. Les sous-nuées SN_1 et SN_2 ont toutes les deux une dynamique qui est susceptible de déclencher des réparations étant donné que le seuil r est dépassé. Par contre, la sous-nuée SN_3 n'a pas suffisamment de fragments pour se réparer.

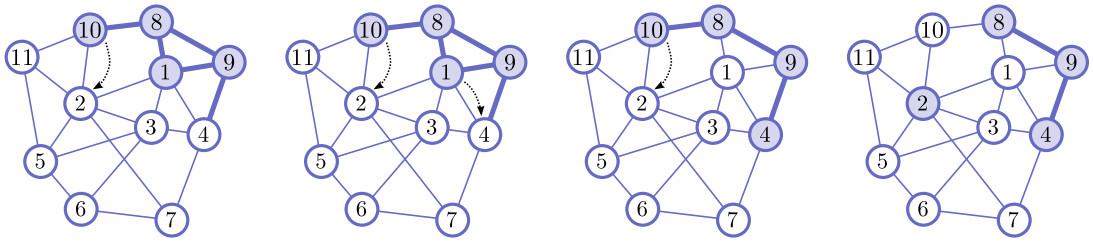
FIGURE 5.2 – Faible et forte amplitude des oscillations de la cohésion d'une nuée.

déclencher chacune une réparation. La sous-nuée SN_3 est, quand à elle, non-viable étant donné qu'elle ne possède pas suffisamment de fragments pour être réparée.

Par conséquent, la réparation à seuil s'avère être bien adaptée à la réparation de nuées puisqu'elle permet de tolérer les oscillations de cohésion qui sont de faible amplitude. La mise en place d'une telle politique passe par la détermination du seuil de réparation. Ce seuil doit être choisi en fonction des variations de la cohésion d'une nuée ainsi que du niveau de réactivité face aux fautes qui est attendu. Nous avons vu à ce propos que la cohésion est elle-même dépendante du nombre de fragments de la nuée ainsi que des propriétés du réseau. C'est-à-dire que pour une instance de réseau donné, il faut trouver une taille de nuée qui génère des ruptures de cohésion de faible amplitude pour le seuil de réparation choisi. Une étude sur le choix de ces paramètres de fragmentation et de réparation est réalisée un peu plus loin dans ce chapitre (cf. Sec.5.3) et nous verrons qu'une nuée est capable de trouver ces paramètres de manière autonome et décentralisée par l'exploration de son espace d'états.

5.1.3 Facteurs ayant un impact sur la cohésion d'une nuée

Un premier mécanisme permettant de limiter l'apparition des ruptures de cohésion en limitant le choix dans les candidats de flocking a été introduit dans l'algorithme 17 (cf. Sec. 4.2.4). Cet algorithme a recours à des agents pivots pour éliminer des choix de déplacement qui engendreraient des ruptures de cohésion. Cependant, ces ruptures sont inévitables par le simple fait que les fragments pivots sont également des agents mobiles. Un agent peut donc prendre un autre agent de sa nuée comme pivot pour choisir son déplacement et s'apercevoir une fois qu'il s'est déplacé que ce pivot a lui même bougé. Ce phénomène est illustré sur l'exemple de la Fig. 5.3 dans lequel une nuée de 4 fragments rompt sa cohésion malgré l'utilisation de pivots.



(a) $t = 1$,*cohésion* $= 4$ (b) $t = 2$,*cohésion* $= 4$ (c) $t = 3$,*cohésion* $= 4$ (d) $t = 4$,*cohésion* $= 3$

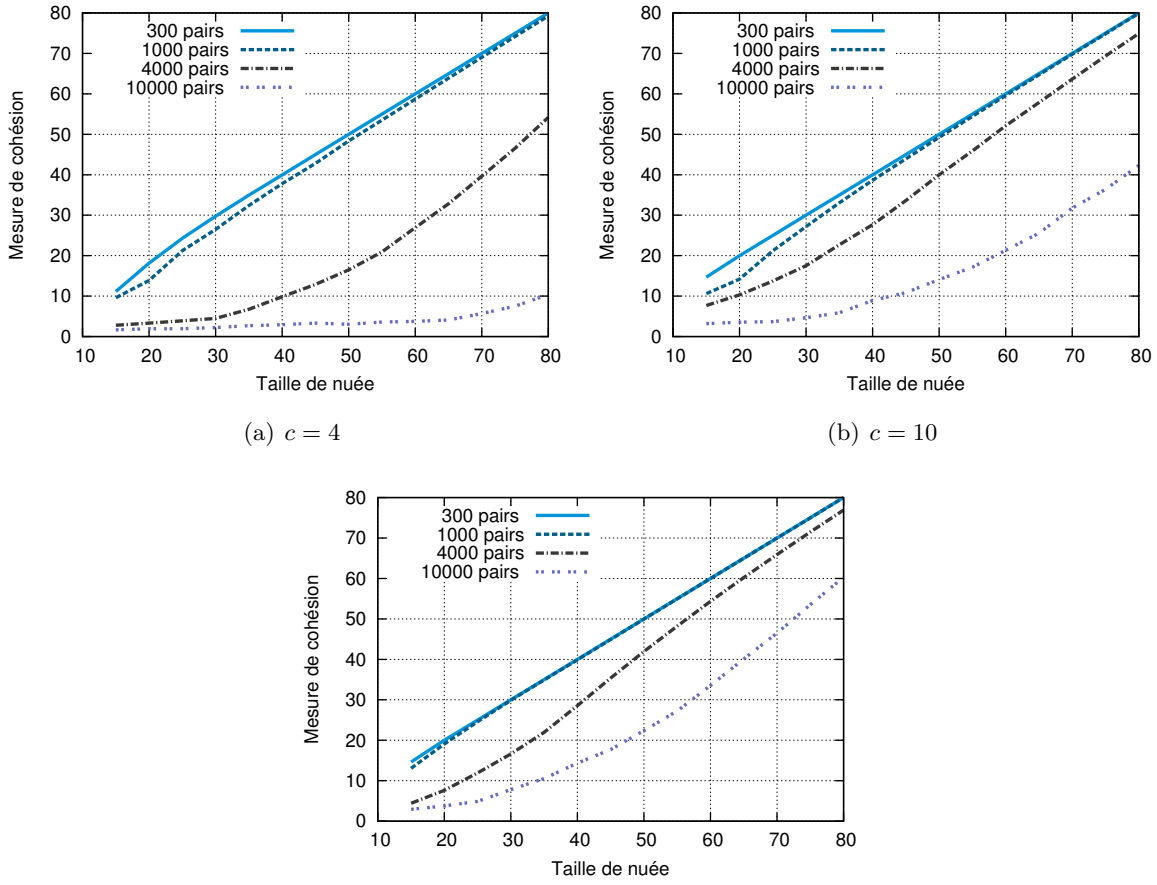
FIGURE 5.3 – Illustration d'une rupture de cohésion provoquée par un pivot mobile.

Dans cet exemple, à $t = 1$, l'agent hébergé sur le pair 10 décide de se déplacer sur le pair 2 et choisit le fragment hébergé sur le pair 1 comme pivot. Seulement, à $t = 2$, l'agent pivot décide également de se déplacer sur le pair d'identifiant 4. À $t = 3$, l'agent pivot a terminé son déplacement plus rapidement que l'agent hébergé sur le pair 10 (à cause d'un lien réseau plus rapide par exemple). À $t = 4$, l'agent hébergé initialement sur le pair 10 a terminé son déplacement et se retrouve isolé du fait que son pivot s'est déplacé entre temps. La valeur de cohésion est donc égale à 3 alors qu'elle devrait être égale à 4.

Les ruptures de cohésion sont inéluctables. Cependant, à la vue des résultats présentés à la fin du chapitre précédent, il apparaît que la taille et le degré moyen du réseau ainsi que le nombre de fragments d'une nuée ont un impact sur la valeur de cohésion observée. Nous proposons, par conséquent, de vérifier ces hypothèses par des expériences reposant sur une variation de paramètres beaucoup plus importante que précédemment. Ces nouvelles expériences vont notamment nous permettre de donner des valeurs ainsi que des ordres de grandeurs sur ces paramètres. Elles sont effectuées sur le simulateur et consistent à faire évoluer une nuée de taille fixe, sans mécanismes de supervision/réparation, dans une instance de réseau et à mesurer sa cohésion moyenne. Les paramètres que nous faisons varier sont au nombre de trois :

1. la taille de la nuée qui varie entre 15 et 80 agents ;
2. la taille du réseau qui varie entre 300 et 10000 pairs ;
3. le degré moyen du réseau pour une taille de réseau donnée, qui est paramétré par les constantes $c = 4$, $c = 10$ et $c = 20$.

Les résultats de ces expériences sont présentés à la Fig. 5.4. Ces graphiques affichent pour chaque triplet de paramètres, les valeurs moyennes de cohésion obtenu à l'issue de la reproduction de 70 simulations. Par exemple, sur le graphique de la Fig. 5.4(a), on lit sur la courbe pleine, les valeurs de cohésions de nuées évoluant dans un (300,4)-SCAMP. Dans cette instance de réseau, les nuées ayant un nombre de fragments égal à 40 ont obtenues une valeur de cohésion moyenne égale à 40 ; c'est-à-dire que sur ce triplet de paramètres, il n'y a pas eu de ruptures de cohésion. On note à ce propos que plus une courbe se rapproche de la fonction affine et moins les nuées correspondantes sont victimes de ruptures de cohésion pour l'instance de réseau considérée. Si nous nous attardons un peu plus sur cette Fig. 5.4(a), nous constatons que dans un (300,4)-SCAMP et dans un (1000,4)-SCAMP, les nuées sont victimes de ruptures de cohésion de faible



(a) $c = 4$

(b) $c = 10$

(c) $c = 20$

FIGURE 5.4 – Analyse des facteurs ayant un impact sur la cohésion de nuées.

amplitude et ce, quelle que soit leur taille. On remarque également, que plus la taille d'une nuée augmente et moins l'amplitude de ces ruptures est importante. En revanche, pour les réseaux (4000,4)-SCAMP et (10000,4)-SCAMP, les ruptures de cohésion sont de forte amplitude pour les tailles de nuées simulées. En particulier, sur l'instance réseau de 10000 pairs, le flocking est quasiment inexistant pour les tailles de nuées considérées. Cette constatation n'est pas sans nous rappeler l'une des questions soulevées au chapitre précédent : existe t-il une taille de nuée minimale au dessus de laquelle le flocking émerge ? À la vue de ces résultats, il semble que, non seulement une taille de nuée minimale est requise pour que le flocking émerge, mais également que cette taille est dépendante de l'instance du réseau considérée et plus particulièrement de la taille de ce réseau.

Si nous passons maintenant aux résultats du graphique de la Fig. 5.4(b), dans lequel le degré moyen des instances réseau a augmenté via la constante $c = 10$, nous ne constatons pas de changements significatifs pour les réseaux de 300 et 1000 pairs. Par contre, pour les instances de 4000 et 10000 pairs, les valeurs de cohésion mesurées sont significativement plus importantes. En effet, dans l'instance (4000,10)-SCAMP, l'amplitude moyenne des ruptures de cohésion est de l'ordre de 10 fragments contre 30 dans l'expérience précédente. De même, pour l'instance (10000,10)-SCAMP, à partir d'un nombre de fragments égal à 40, des nuées viables apparaissent, marquant un gain significatif avec les résultats de l'expérience précédente. La comparaison entre la Fig. 5.4(a) et la Fig. 5.4(b) met en évidence le fait qu'à taille de réseau identique, l'augmentation du degré moyen permet de diminuer l'amplitude des ruptures de cohésion d'une nuée. Cette tendance est manifestement confirmée par les résultats de la Fig. 5.4(c) dans lesquels l'amplitude des ruptures de cohésion a encore diminuée par rapport aux expériences précédentes. Dans cette expérience, les nuées évoluant dans des réseaux de 300 et 1000 pairs ne subissent plus de ruptures de cohésion. Ceci met en évidence le fait qu'il existe manifestement un seuil sur les propriétés du réseau à partir duquel le phénomène de ruptures de cohésion n'apparaît plus. Nous pouvons extraire de ces observations un certain nombre de conclusions :

- la cohésion d'une nuée est dépendante de la taille du réseau dans lequel elle évolue ;
- pour une instance de réseau donnée, plus une nuée est grande et plus sa cohésion est préservée ;
- pour une instance de réseau donnée, il existe une taille de nuée à partir de laquelle la cohésion est toujours préservée ;
- pour une instance de réseau donnée, il existe une taille de nuée en-dessous de laquelle le flocking n'émerge pas.
- pour une taille de réseau donnée, l'augmentation de son degré moyen réduit l'amplitude des ruptures de cohésion que subissent les nuées évoluant en son sein.

Si nous reprenons la métaphore du vol en nuée des oiseaux, nous pouvons dire que l'augmentation du nombre d'agents d'une nuée de fragments peut être comparée à une densification de la nuée dans l'espace alors que l'augmentation du degré du réseau logique peut être, quant à elle, comparée à une augmentation de la perception de chaque agent.

Cette étude approfondie sur le comportement d'une nuée d'agents mobiles soulève également

la question du compromis à réaliser lors du choix des paramètres. Ce compromis se situe au niveau de la taille des nuées, du degré moyen du réseau et de l'amplitude des ruptures de cohésion. D'un côté, l'explosion inutile du nombre de fragments est clairement néfaste pour le système puisqu'elle augmente la charge réseau et gaspille des ressources de stockage inutilement. D'un autre côté, l'augmentation trop importante du degré moyen du réseau pour réduire l'explosion du nombre de fragments a pour conséquence d'alourdir la charge réseau puisque la quantité de trafic de contrôle inter-pairs s'en trouve augmentée. Ce problème de passage à l'échelle du système sera abordé au Chap. 7. Nous laissons cette problématique de côté pour le moment et nous nous concentrons sur la supervision et la réparation de nuées.

5.2 Supervision et réparation d'une nuée

Dans un système où la cohésion est totalement maîtrisée [17], seules les pannes de pairs, entraînant des pertes de fragments, sont susceptibles de déclencher des réparations. Nous avons vu que la politique de réparation à seuil est bien adaptée pour tolérer les variations de la cohésion d'une nuée ; cependant, nous n'avons pas décrit l'algorithme qui permet d'effectuer ces réparations. Nous proposons donc dans cette section de décrire la procédure que nous utilisons pour réaliser la supervision et la réparation d'une nuée de manière décentralisée.

5.2.1 Élection de leader

Le placement local et mobile d'une nuée a été retenu dans un souci de décentralisation des mécanismes de supervision et de réparation, conformément aux hypothèses formulées à la base de ce travail. Cette décentralisation est réalisée au moyen d'une élection périodique d'un leader choisi parmi les agents d'une nuée. Le rôle de l'agent leader est de compter le nombre de fragments de sa nuée au moyen d'un arbre couvrant distribué et de déclencher une réparation si le seuil de réparation r a été atteint. Soit Δ_t, la période entre deux supervisions ; cette valeur représente la réactivité du système face aux fautes et doit être choisie en fonction du nombre moyen de fautes par seconde qui est observé dans le système, ainsi que du temps nécessaire à une réparation. Chaque agent va donc armer une temporisation $t = \Delta_t + \epsilon$, avec ϵ un nombre tiré aléatoirement dans un intervalle cohérent avec Δ_t (i.e., inférieure à Δ_t). Lorsque cette temporisation expire, l'agent correspondant lance une procédure d'élection de leader qui est basée sur le calcul d'un arbre couvrant du réseau, limitée au graphe de sa nuée. Cette procédure découle directement de l'algorithme *Minimum-Weight Spanning Tree* [Gallager *et al.*, 1983] (MST). Une fois que l'arbre couvrant est trouvé, la racine de l'arbre correspond au pair leader de la nuée. Cet algorithme a la particularité d'être décentralisé, d'informer l'ensemble de ses participants de sa terminaison et de supporter plusieurs initiateurs concurrents.

17. i.e., les paramètres de fragmentation et de réparation sont choisis en adéquation avec les propriétés du réseau de sorte que les ruptures de cohésion soient de faible amplitude.

Principe de l'algorithme MST

L'algorithme MST construit l'arbre couvrant de poids minimal d'un réseau modélisé sous la forme d'un graphe non-orienté $G = (V,E)$. Cet algorithme a la particularité de pouvoir être initié par chacun des nœuds de manière asynchrone. On appelle sous-MST ou S-MST, un sous-arbre de l'arbre couvrant de poids minimal du graphe. L'algorithme débute avec chacun des nœuds constituant à lui seul un S-MST et se termine lorsqu'un unique S-MST isomorphe au MST est construit. Soit $\mathcal{S}$, un S-MST, on appelle arête sortante de $\mathcal{S}$ une arête qui relie deux nœuds a et b tel que $a \in \mathcal{S}$ et $b \notin \mathcal{S}$. Gallager *et al.* basent la construction de leur algorithme sur les deux propriétés suivantes :

Propriété 30 *Soit $\mathcal{S}$ un S-MST d'un MST $\mathcal{M}$ et soit $e = (a,b)$ une arête sortante de $\mathcal{S}$ de poids minimal tel que $a \in \mathcal{S}$. Alors, intégrer le nœud b à $\mathcal{S}$ via l'arête e génère un nouvel arbre qui est également un S-MST de $\mathcal{M}$.*

Propriété 31 *Si toutes les arêtes d'un graphe connexe ont des poids distincts, alors, le MST qui en résulte est unique.*

Ces deux propriétés nous donnent une intuition sur la construction du MST. L'algorithme débute avec chaque nœud constituant un S-MST à lui seul. La Prop. 30 va nous permettre d'élargir ces S-MST dans n'importe quel ordre. Ensuite, sous hypothèse que les arêtes ont un poids distinct [18], lorsque deux S-MST ont une arête commune la Prop. 31 assure que l'union de ces deux S-MST est également un S-MST. Le squelette de l'algorithme est le suivant : chaque S-MST cherche de manière asynchrone une arête sortante de poids minimal. Lorsque cette arête est trouvée, le S-MST va chercher à fusionner avec le S-MST qui est à l'autre extrémité de l'arête. La manière dont cette fusion est réalisée dépend du *niveau* de ces deux S-MST, qui dépend, lui même, des précédentes fusions. Plus spécifiquement, un S-MST ne contenant qu'un seul nœud est au niveau 0. Supposons maintenant qu'un S-MST $\mathcal{S}$ donné a un niveau $\mathcal{L} \geq 0$ et que le S-MST $\mathcal{S}'$ à l'autre extrémité de l'arête sortante de poids minimal de $\mathcal{S}$ a un niveau $\mathcal{L}'$.

- Si $\mathcal{L} < \mathcal{L}'$, alors $\mathcal{S}$ est immédiatement fusionné dans $\mathcal{S}'$ et ce nouveau S-MST reste au niveau $\mathcal{L}'$.
- Si $\mathcal{L} = \mathcal{L}'$ alors $\mathcal{S}$ et $\mathcal{S}'$ ont la même arête sortante de poids minimal et les deux S-MST fusionnent immédiatement en un nouveau S-MST de niveau $\mathcal{L} + 1$. L'arête ayant donné lieu à la fusion est appelé *cœur* du nouvel S-MST.
- Dans les autres cas, le S-MST $\mathcal{S}$ attend simplement que $\mathcal{S}'$ atteigne un niveau suffisamment élevé pour pouvoir fusionner avec.

Exemple 32 *Nous reprenons l'exemple, donné par Gallager et al., de la Fig. 5.5 pour illustrer ces règles. $\mathcal{S}$ est un S-MST de niveau 1 né de la fusion entre les nœuds d'identifiant 1 et 2 sur l'arête de poids minimal* (1,2)*. Le nœud 3 a ensuite été fusionné avec son arête de poids minimal*

18. On note qu'il est possible d'obtenir des arêtes de poids distincts en hachant, pour chaque arête, la concaténation des adresses des pairs se trouvant à ses extrémités.

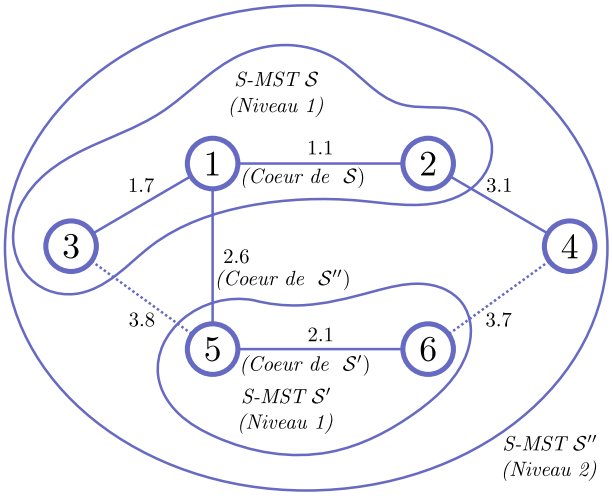


FIGURE 5.5 – S-MST et cœurs

car son niveau était inférieur à celui de $\mathcal{S}$. $\mathcal{S}'$ *s'est construit en parallèle et va fusionner avec* $\mathcal{S}$ *sur l'arête de poids minimal* (1,5). *Cette fusion forme* $\mathcal{S}''$ *de niveau 2 et termine l'algorithme. Le cœur de* $\mathcal{S}''$ *est l'arête* (1,5) *et, soit* 1 *soit* 5 *peuvent être choisis comme racine de l'arbre.*

5.2.2 Détails de l'algorithme d'élection

Nous présentons maintenant plus en détail l'algorithme MST [19] qui peut être séparé en quatre phases :

1. la recherche par chaque nœud de son arête sortante de poids minimal ;
2. la coopération inter-nœuds pour trouver l'arête sortante de tout un S-MST ;
3. la fusion de deux S-MST ;
4. la détection de la terminaison de l'algorithme.

Dans cet algorithme, les nœuds peuvent se trouver dans les états :

- *sleeping* : le nœud ne participe pas encore à l'algorithme ;
- *find* : le nœud participe au choix de l'arête sortante de poids minimal de son S-MST ;
- *found* : l'arête sortante du S-MST a été trouvée.

Les arêtes peuvent se trouver dans les états :

- *branch* : l'arête est désignée comme étant une branche du MST ;
- *rejected* : l'arête relie deux nœuds du S-MST mais n'est pas une branche. Exemple de l'arête (3,5) pour le S-MST $\mathcal{S}''$ sur la Fig. 5.5 ;
- *basic* : l'arête n'est ni une branche ni une arête rejetée.

19. Le pseudo-code de l'algorithme peut être trouvé dans [Gallager *et al.*, 1983].

Trouver l'arête sortante de poids minimal d'un nœud

Tout nœud se trouve initialement dans l'état *sleeping*. Nous commençons par décrire la procédure qui permet aux nœuds de trouver leur arête sortante de poids minimal et nous considérons, dans un premier temps, le cas dans lequel le S-MST est réduit à un seul nœud (niveau $\mathcal{L} = 0$). Dans ce cas, un nœud se réveillant spontanément à la suite de l'expiration de la temporisation $\Delta_t + \epsilon$ ou se réveillant par la réception de n'importe quel message du protocole, va choisir son arête incidente e qui est de poids minimal. Une fois que e a été trouvée, le nœud marque cette arête comme étant une *branche* du MST, envoie un message *Connect* au nœud se trouvant à l'autre extrémité de l'arête et passe dans l'état *found* en attente d'une réponse au message *Connect*.

Nous nous intéressons maintenant au cas des nœuds de niveau $\mathcal{L} > 0$. Supposons qu'un S-MST $\mathcal{S}''$ de niveau $\mathcal{L}$ vient de se créer suite à la fusion de deux S-MST $\mathcal{S}$ et $\mathcal{S}'$ de niveau $\mathcal{L} - 1$ sur leur arête sortante de poids minimale e (cf. Fig. 5.6). L'arête e devient le cœur de $\mathcal{S}''$ et son poids l'identifie (on a fait l'hypothèse que les poids des arêtes sont distincts). À l'issue de la fusion, les deux nœuds aux extrémités du cœur e vont envoyer sur leurs branches un message *Initiate*. Ce message, qui contient l'identité ainsi que le niveau de $\mathcal{S}''$, est propagé sur les branches de $\mathcal{S}$ et $\mathcal{S}'$ jusqu'aux feuilles pour informer les membres de cette nouvelle fusion. Ce message est également propagé aux S-MST de niveau $\mathcal{L} - 1$ qui seraient en attente de fusion avec des nœuds de $\mathcal{S}''$. Lorsqu'un nœud reçoit le message *Initiate*, il passe dans l'état *find* et cherche son arête incidente qui sort du S-MST et qui est de poids minimal. Afin d'être sûr qu'il trouve bien une arête sortante de son S-MST, le nœud ne considère que les arêtes dans l'état *basic* pour ses candidats. Une fois qu'un tel candidat est trouvé, le nœud lui envoie un message *Test* qui encapsule l'identité de son S-MST. Sur réception d'un message *Test* envoyé par un nœud a à un nœud b, b va comparer l'identité de son S-MST avec celle contenue dans le message :

- si les deux identités sont égales (l'arête n'est pas sortante), b répond à a avec un message *Rejected* qui a pour conséquence de passer l'état du lien concerné à *rejected* pour ses deux extrémités ;
- dans le cas où les deux identités diffèrent :
  - si le niveau du S-MST de b est supérieur ou égal à celui du S-MST de a, alors, le message *Accept* est envoyé en réponse et certifie à a que l'arc est bien un arc sortant,
  - si le niveau du S-MST de b est inférieur à celui du S-MST de a, b se met en attente et diffère sa réponse jusqu'à ce que le niveau de son S-MST soit supérieur ou égal à celui de a.

Trouver l'arête sortante de poids minimal d'un S-MST

Nous venons de détailler la procédure qui permet à un nœud de trouver une arête incidente de poids minimal qui sort de son S-MST. Mais cette arête sortante n'est peut être pas l'arête sortante de poids minimal de tout le S-MST. Les nœuds doivent donc ensuite coopérer pour désigner l'unique arête sortante de poids minimale de tout leur S-MST. Pour ce faire, chaque

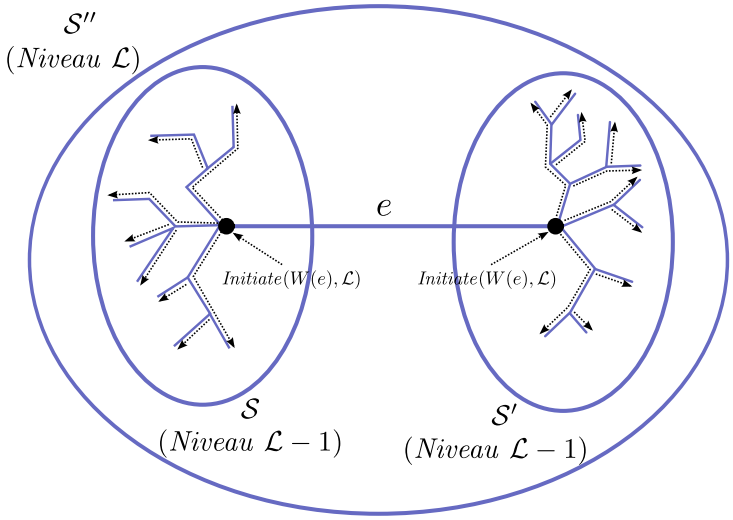


FIGURE 5.6 – Propagation du message *Initiate* dans les deux branches du nouveau S-MST $\mathcal{S}''$.

feuille du S-MST (i.e., les nœuds qui ne sont adjacents qu'à une seule branche) va envoyer le message *Report(W)* sur sa branche parente, avec W, le poids de son arête incidente de poids minimal qui sort du S-MST ($W = \infty$ s'il n'existe pas de telle arête). De manière similaire, tout nœud interne du S-MST cherche son arête incidente et sortante de poids minimal. Il se met ensuite en attente tant que tous ses fils ne lui ont pas communiqué leur meilleure arête sortante. Une fois que tous les fils d'un nœud interne lui ont communiqué leur choix, le nœud désigne comme meilleur arête sortante celle qui a le poids le plus faible et propage à son tour le message *Report(W)* sur sa branche parente. Lorsqu'un nœud envoie un message *Report*, il passe dans l'état *found* pour signifier le fait qu'il a achevé la phase de recherche de l'arête sortante du S-MST. Lorsque le message *Report(W)* atteint un des nœuds adjacents au cœur du S-MST, ce nœud le propage à l'autre extrémité du cœur pour que la meilleure arête soit choisie. Sur l'exemple de la Fig. 5.7, le S-MST $\mathcal{S}$ a une arête sortante vers le S-MST $\mathcal{S}'$ de poids 2 et une arête sortante vers le S-MST $\mathcal{S}''$ de poids 1. Le poids de ces arêtes remonte au cœur de $\mathcal{S}$ par des messages *Report*. La meilleure arête est celle qui sort vers $\mathcal{S}''$.

Fusionner deux S-MST

Lorsque l'arête sortante de poids minimal du S-MST est choisie par le cœur du S-MST à l'issue de la phase de recherche (tous les nœuds du S-MST sont dans l'état *found*), un message *Change-core* est envoyé dans le sous-arbre qui contient cette arête sortante. Ce message *Change-core* est envoyé par l'un des nœuds du cœur et emprunte le chemin inverse des messages *Report* jusqu'à arriver sur le nœud connecté à l'arête sortante du S-MST qui a été désignée comme étant de poids minimal. Lorsque le message *Change-core* arrive à cette arête sortante, le message *Connect($\mathcal{L}$)*, contenant le niveau du S-MST, est envoyé vers le S-MST voisin pour le fusionner avec le S-MST actuel. La Fig. 5.8 illustre la redescente du message *Change-core* sur le même cas d'exemple que celui précédemment proposé à la Fig. 5.7. Dans cet exemple, l'arête sortante vers

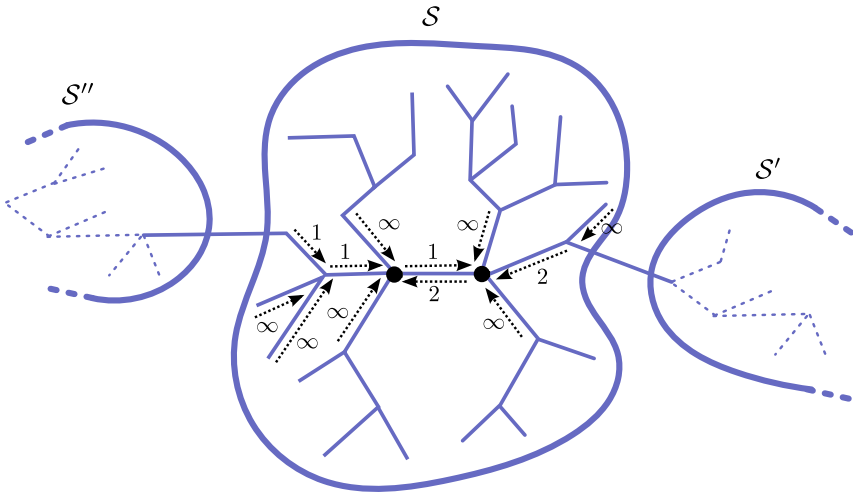


FIGURE 5.7 – Illustration de la remontée de messages *Report* lors de la phase de recherche de l'arête sortante de poids minimal d'un S-MST.

$\mathcal{S}''$ a été choisie pour devenir le nouveau cœur du S-MST résultant de la fusion entre $\mathcal{S}$ et $\mathcal{S}''$.

Si $\mathcal{S}$ et $\mathcal{S}''$ ont le même niveau $\mathcal{L}$, alors ils fusionnent sur cette arête et donnent naissance à un nouvel S-MST $\mathcal{S}'''$ de niveau $\mathcal{L}+1$. Un message *Initiate*, encapsulant la nouvelle identité du S-MST, est ensuite propagé dans chaque sous-arbre (cf. Fig. 5.6) qui a pour conséquence de passer les nœuds de $\mathcal{S}'''$ dans l'état *find*. Si, en revanche, $\mathcal{S}$ est au niveau $\mathcal{L}$ et $\mathcal{S}''$ au niveau $\mathcal{L}''$ et que $\mathcal{L} < \mathcal{L}''$, alors, $\mathcal{S}$ est absorbé par $\mathcal{S}''$ et le nouvel S-MST reste au niveau $\mathcal{L}''$. Un message *Initiate*, encapsulant l'identité de $\mathcal{S}''$ et $\mathcal{L}''$, est également propagé dans chaque sous-arbre pour notifier de cette fusion.

Détecter la terminaison de l'algorithme

Cet algorithme est une succession de phases de recherche de l'arête sortante et de phases de fusion. Une phase de recherche s'achève lorsque tous les nœuds d'un S-MST ont envoyé leur message *Report*. Il s'ensuit la phase de fusion avec le S-MST voisin, relié par la meilleure arête sortante du S-MST actuel. Cette fusion diffuse un message *Initiate* dans le nouvel arbre et passe les nœuds dans l'état *find*, relançant ainsi une nouvelle phase de recherche. Cette boucle *recherche–fusion* s'exécute jusqu'à ce que le message *Report*(∞) remonte au cœur du S-MST et qu'il soit échangé par les deux extrémités de ce cœur. Lorsque cet échange se produit, cela signifie qu'il n'existe plus d'arête sortante du S-MST et que cet arbre est donc le MST du graphe. C'est de cette manière que l'algorithme détecte sa terminaison. L'un des deux nœuds adjacents au cœur sera donc désigné comme étant la racine de l'arbre (cette désignation peut par exemple utiliser l'ordre qui existe entre les identifiants des nœuds) et pourra diffuser un message *Leader*(a) à travers l'arbre, avec a l'identifiant du pair racine. Ce message *Leader* clôture l'élection et a pour effet de notifier chaque nœud de l'identité du leader qui a été choisi à l'issue de l'algorithme.

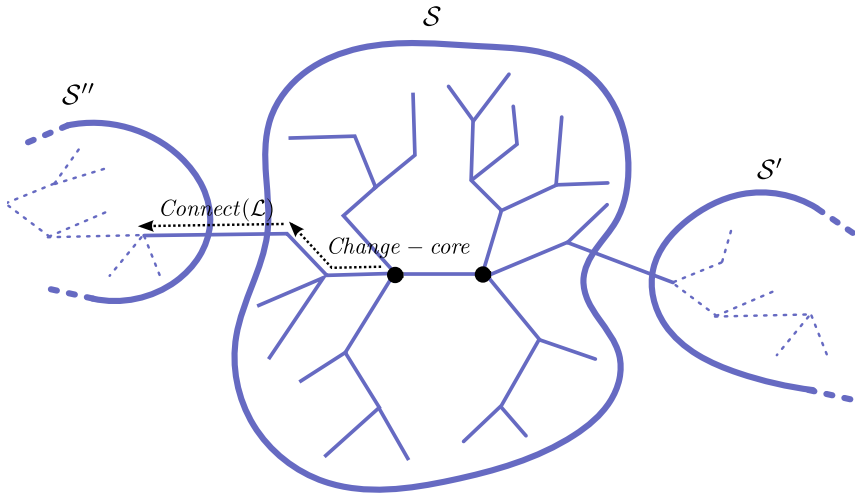


FIGURE 5.8 – Lorsque l'arête sortante du S-MST est trouvée, une fusion est initiée au moyen du message *Change-core*.

5.2.3 Supervision

Le protocole MST est implémenté par la couche réseau (cf. 4.2.1) qui fournit le service d'élection de leader aux agents. C'est-à-dire que ce sont les pairs qui initient et participent au protocole et, non les agents. De manière similaire à l'information sur les voisinages, les agents adressent des requêtes à la couche réseau pour initier le protocole et récupérer l'information sur leur leader et sur l'arbre couvrant actuel. Étant donné que plusieurs agents de nuées différentes peuvent être hébergés sur le même pair, un pair peut donc être amené à participer à plusieurs élections simultanément. C'est pourquoi l'identifiant des nuées est ajouté à chaque message du protocole et permet à un pair de gérer un arbre couvrant par nuée. Ensuite, lorsqu'un pair exécute le protocole pour une nuée et qu'il cherche son arête sortante de poids minimal, il ne considère comme des nœuds du graphe que les pairs voisins qui sont occupés par un agent de cette même nuée. Lorsque le protocole se termine, la couche réseau notifie la couche multi-agents mobiles que le protocole s'est terminé. Chaque agent peut ainsi consulter localement ses informations sur le MST et notamment l'identité du leader.

Lorsqu'un agent d'une nuée α envoie une requête à sa couche réseau pour initier une élection de leader (suite à l'expiration de sa temporisation Δ_t), cet appel a pour effet préliminaire de propager à toute la nuée par inondation un message $Stop(\alpha)$. Sur réception d'un tel message, tout agent de la nuée α arête de se déplacer, propage le message à ses voisins et arme une temporisation τ choisie telle que $\tau \leq \Delta_t$. L'initiateur du protocole arme également cette temporisation qui va servir à reprendre les déplacements de la nuée si la supervision échoue. Si la temporisation τ arrive à expiration, un message $Start(\alpha)$ est propagé par inondation à tous les agents de la nuée. Sur réception de ce message *Start*, tout agent reprend ses déplacements, propage le message à ses voisins, désamorce la temporisation τ et considère la supervision comme étant achevée. Ce mécanisme permet à une nuée d'éviter de rester bloquée si une erreur survient durant l'exécution

du protocole.

Une fois tous ces éléments mis en place, la supervision peut avoir lieu. Un agent d'une nuée α dont la temporisation $\Delta_t + \epsilon$ expire lance une élection de leader. Si cette élection est un succès, l'agent leader l est notifié de son statut par la couche réseau et effectue un parcours en profondeur de l'arbre MST pour se construire une vision de sa nuée. Il peut alors compter N_α le nombre d'agent disponibles dans le réseau.

5.2.4 Réparation

Si $m \leq N_\alpha \leq r$ alors le leader l choisit m agents de sa nuée et les contacte pour récupérer les fragments qu'ils encapsulent. Lorsque l a les m fragments en sa possession, il décode la donnée originale et génère les agents fragments manquants. Le leader peut ensuite envoyer, par propagation aux agents de la nuée, le message $Start(\alpha)$, qui a pour conséquence de les notifier que la réparation est terminée et qu'ils peuvent reprendre leurs déplacements. L'envoi de ce message a également pour conséquence d'initier le déplacement du leader et des nouveaux agents. La règle de séparation aura alors pour effet de les faire se déplacer immédiatement dès leur prochaine prise de décision. Plusieurs fautes peuvent survenir pendant le processus de réparation :

- le superviseur subit une faute : dans ce cas, la réparation échoue et il faudra attendre que la temporisation τ expire sur tous les agents de la nuée pour que les déplacements reprennent. Ensuite, à l'expiration de la prochaine temporisation $\Delta_t + \epsilon$ un nouveau processus de supervision et de réparation s'initiera ;
- l'un des m agents choisis pour réparer la nuée subit une faute avant d'avoir été envoyé au superviseur : dans ce cas, l demande à un autre agent de la nuée toujours disponible de lui envoyer son fragment. Si ce n'est pas possible, cela signifie que la nuée n'est plus viable et qu'elle ne peut pas être réparée.

Nous avons fait l'hypothèse au début de cette section que la cohésion est totalement maîtrisée et ne génère pas de ruptures. Cependant, si on relâche cette contrainte et que des codes d'effacement classiques (Reed-Solomon par exemple) sont utilisés, il se peut que deux fragments identiques se retrouvent dans la même nuée. Ce phénomène est provoqué par le fait que les codes d'effacement linéaires génèrent toujours les mêmes fragments de manière déterministe. Hors, si une nuée est séparée en deux à la suite d'une rupture de cohésion et qu'une des deux partitions se répare, alors les fragments réparés seront identiques à ceux de la première partition. Lorsque les deux partitions de la nuée fusionneront à nouveau, la nuée résultante aura des fragments dupliqués. Ce phénomène perturbe les supervisions basées sur un simple comptage des agents puisqu'elles donnent au superviseur l'illusion que le niveau de disponibilité est supérieur au niveau réel. Une contre-mesure permettant d'éviter cette perturbation du superviseur consiste à modifier la supervision pour qu'elle ne compte les agents identiques qu'une seule fois (au moyen du haché du fragment par exemple) et que les m agents sélectionnés pour la réparation soient distincts.

5.3 Recherche des paramètres de fragmentation d'une nuée

5.3.1 Adaptation d'une nuée à son environnement

Il est primordial d'avoir une fluctuation de la cohésion qui soit maîtrisée pour que les mécanismes de supervision et de réparation soient efficaces et que le niveau de disponibilité attendu pour la nuée soit atteint. Nous avons montré à la Fig. 5.4 que l'amplitude des ruptures de cohésion peut être contrôlée en adaptant la taille de la nuée au réseau dans lequel elle évolue. Cependant, plutôt que de donner dans l'exhaustivité en calculant une table qui contiendrait les valeurs de cohésion pour toutes les combinaisons de paramètres possibles, et qui, de toute évidence, ne serait pas réaliste, nous souhaitons plutôt que chaque nuée soit capable de trouver ses propres paramètres pendant son exécution. En d'autres termes, qu'elle s'auto-adapte à l'environnement dans lequel elle évolue. Cette recherche d'un état d'équilibre, dans lequel les ruptures de cohésion sont tolérables pour un seuil de réparation donné, va s'obtenir grâce au mécanisme de réparation qui va déclencher une succession de réparations jusqu'à obtenir cet équilibre. C'est-à-dire que chaque nuée va se déplacer et lancer des réparations à chaque fois que le seuil r est dépassé suite à des ruptures de cohésion. Les nouveaux fragments introduits dans le système vont s'additionner aux fragments existants et par conséquent augmenter la taille de la nuée qui va alors croître petit à petit jusqu'à trouver un équilibre. Une fois cet équilibre trouvé, les nuées pourront décider de réajuster leur schéma de fragmentation pour qu'il soit en adéquation avec leur nouvelle taille.

5.3.2 Protocole expérimental

Cette faculté d'adaptation d'une nuée à son environnement a été analysée au moyen de plusieurs simulations. Ces simulations ont été menées sur des réseaux pair-à-pair (p,c)-SCAMP de taille $p = 100$, $p = 300$, $p = 600$, $p = 1000$, $p = 4000$ et $p = 10000$ et pour des constantes $c = 4$, $c = 10$ et $c = 20$. Une nuée fragmentée avec un (10,5)-code d'effacement et possédant un seuil de réparation fixé à $r = 13$ est ensuite déployée dans chacune de ces instances de réseau. Ceci signifie que chaque nuée commence initialement avec 15 fragments et se répare chaque fois que sa cohésion est comprise entre 10 et 13. Étant donné que l'objectif de ces expérimentations est de laisser chaque nuée déduire le schéma de fragmentation à appliquer pour le seuil de réparation donné, aucun modèle de fautes n'a été activé puisqu'il aurait biaisé les résultats. De plus, le Δ_t de la politique de supervision est fixé à 1 cycle (i.e., une supervision a lieu toutes les minutes) pour permettre une convergence plus rapide vers la valeur d'équilibre [20].

5.3.3 Résultats

Les tableaux 5.1, 5.2 et 5.3 ainsi que la Fig. 5.9 présentent les résultats de ces simulations. Chaque tableau regroupe les résultats pour différentes tailles de réseau. On peut y lire la mesure

20. Étant donné qu'aucun modèle de fautes n'est actif, la supervision et la réparation ne peuvent pas échouer et le mécanisme associé à la temporisation τ est inutile dans ce cas et a donc été désactivé.

Propriétés réseau		Cohésion x			Nombre de fragments y		
p	$deg(p)$	$\bar{x}$	σ_x	Var_x	$\bar{y}$	σ_y	Var_y
100	16	18.7	0.7	0.5	18.8	0.7	0.5
300	21	25.6	1.1	1.1	26.1	1.8	3.4
600	25	31	1.3	1.7	33.5	1.15	1.3
1000	27	36.6	1.7	3.07	41.9	1.5	2.3
4000	36	46.4	17.6	**310**	70.3	21.4	**457**
10000	39	37.1	29.6	**879**	84.8	55.7	**3113**

TABLE 5.1 – Mesure de la cohésion et du nombre de fragments d'une nuée initialement de taille 15 avec $r = 13$ et $c = 4$.

Propriétés réseau		Cohésion x			Nombre de fragments y		
p	$deg(p)$	$\bar{x}$	σ_x	Var_x	$\bar{y}$	σ_y	Var_y
100	25	16.1	0.7	0.5	16.1	0.7	0.5
300	35	22.6	1.3	1.6	23	1.2	1.5
600	45	26.8	1.2	1.4	27.7	1.1	1.3
1000	54	31.6	1.3	1.9	33.8	1.3	1.6
4000	70	44.9	5.9	34.1	55.8	7	49
10000	72	49.6	5.84	76.2	76.7	13.5	183.9

TABLE 5.2 – Mesure de la cohésion et du nombre de fragments d'une nuée initialement de taille 15 avec $r = 13$ et $c = 10$.

de la cohésion moyenne et la mesure du nombre de fragments moyens observés durant chaque simulation. On y trouve également la valeur du degré moyen du réseau. Par exemple, dans la Tab. 5.1, la 3^{e} ligne signifie que dans un réseau (100,4)-SCAMP, la nuée après réparations successives a obtenu une cohésion moyenne de 18,7 fragments, d'écart-type 0,7 et de variance 0,5. Le nombre de fragments moyens de cette nuée est de 18.8 ce qui signifie qu'elle a eu besoin d'introduire quatre nouveaux agents pour que ses déplacements n'influent plus sur la politique de supervision.

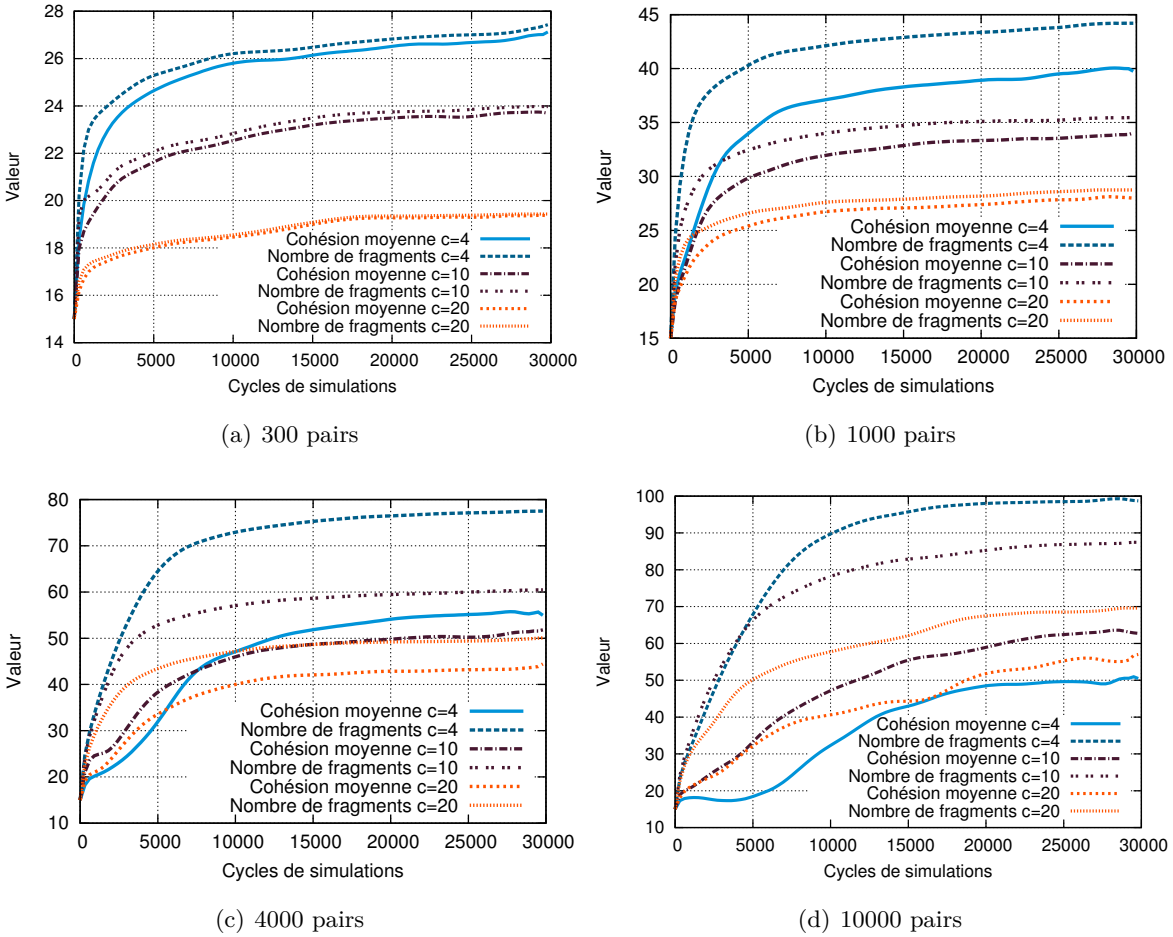
On constate à la vue des résultats de la Tab. 5.1 que, d'une manière générale, l'augmentation de la taille du réseau force la nuée à créer un plus grand nombre de fragments pour conserver une cohésion adéquate. On constate également que la cohésion et le nombre de fragments de la nuée restent très proches pour des tailles de réseau inférieures ou égales à 1000 pairs. Ceci signifie que ces nuées subissent des ruptures de cohésion de très faible amplitude. Par contre, pour les réseaux de taille 4000 et 10000, on constate que le nombre de fragments générés est quasiment le double de la valeur de cohésion et que la dispersion de cette distribution est très importante. Ceci signifie que dans les grandes instances de réseau à degré relativement faible, la nuée est souvent victime de ruptures de cohésion de forte amplitude et qu'il est donc nécessaire d'augmenter énormément le nombre de fragments pour s'éloigner du seuil r. Les valeurs de dispersion élevées

Propriétés réseau		Cohésion x			Nombre de fragments y		
p	$deg(p)$	$\bar{x}$	σ_x	Var_x	$\bar{y}$	σ_y	Var_y
100	29	15.2	0.5	0.2	15.2	0.5	0.2
300	45	18.8	0.7	0.5	18.8	0.8	0.5
600	58	23.6	1.1	1.2	24	1.1	1.2
1000	67	26.5	1.1	1.2	27.5	1	1.1
4000	87	39	2.5	6.7	46.3	2.7	7.5
10000	101	43.8	5.2	27.5	59.3	6.7	45

TABLE 5.3 – Mesure de la cohésion et du nombre de fragments d'une nuée initialement de taille 15 avec $r = 13$ et $c = 20$.

pour $p = 4000$ et $p = 10000$ indiquent que, dans ces instances, l'équilibre trouvé est très variable entre plusieurs simulations. La Tab. 5.2 présente des résultats de simulations analogues mais cette fois en augmentant le degré moyen du réseau à l'aide de la constante $c = 10$. Dans les petites instances de réseaux, les résultats sont analogues en terme d'adéquation entre cohésion et nombre de fragments. Pour les plus grandes instances, nous remarquons que la dispersion de la distribution a grandement diminuée et que la valeur de la cohésion s'est rapprochée du nombre de fragments observés. La nuée a donc tendance à moins se séparer en sous-nuées. Il est également à noter que l'augmentation du degré du réseau a permis à la nuée de générer entre 10 et 20% de fragments de moins par rapport à $c = 4$. Les résultats de la Tab. 5.3 effectués sur des réseaux de constante $c = 20$ confirment la tendance observée dans la Tab. 5.2. En effet, avec un degré encore plus important, les mesures de cohésion et la taille de la nuée se confondent et ceci signifie que la nuée n'est plus victime de ruptures de cohésion de forte amplitude. On remarque également que la taille moyenne des nuées est également plus basse avec un gain en nombre de fragments variant entre 19 et 35% par rapport à $c = 4$. Ensemble de tendances que nous avions déjà observées à la Fig. 5.4.

D'une manière générale, nous constatons à l'aide de ces résultats que le nombre de fragments moyen observé est supérieur au nombre de fragments initiaux. Ce constat montre que chaque nuée a effectivement cherché à augmenter sa taille pour que l'amplitude de ses ruptures de cohésion s'éloigne du seuil $r = 13$ donné. Mais ces résultats ne sauraient suffire à déterminer si oui ou non les nuées sont parvenues à trouver un équilibre. C'est pourquoi nous présentons à la Fig. 5.9 les résultats de ces expériences sous une forme différente. Dans ces graphiques, on affiche, pour chaque cycle de simulation, les valeurs moyennes de cohésion et le nombre de fragments de ces nuées. Cette présentation des données nous permet d'analyser l'évolution d'une nuée au cours du temps. La Fig. 5.9(a) montre cette évolution pour les différentes valeurs de c lorsque p est fixé à 300. On constate qu'après une période de réparations, les nuées se stabilisent autour des valeurs données dans les tables précédentes. On note encore une fois le gain de réparations engendré par l'augmentation de la constante c. La Fig. 5.9(b) présente la même tendance pour $p = 1000$ à la différence que lorsque $c = 4$ la nuée, bien qu'ayant convergé vers un état d'équilibre, subit



(a) 300 pairs

(b) 1000 pairs

(c) 4000 pairs

(d) 10000 pairs

FIGURE 5.9 – Évolution de la cohésion et du nombre de fragments de nuées en recherche d'équilibre dans différentes instances réseau.

des ruptures de cohésion de faible amplitude pour le seuil r. Dans les résultats de la Fig. 5.9(c) pour $p = 4000$ et de la Fig. 5.9(d) pour $p = 10000$, on se rend bien compte qu'un grand réseau de faible degré sépare la nuée en sous-nuées. Ce comportement est très accentué dans le réseau (10000,4)-SCAMP où l'on voit clairement que le nombre de fragments a explosé et que sa valeur de cohésion est deux fois moins importante. Toutes les courbes de ces graphiques présentent une phase de réparation (partie croissante) durant laquelle la taille de la nuée augmente jusqu'à converger vers un équilibre. Les nuées ont donc été capables de s'adapter à leur environnement en trouvant la taille adéquate pour le seuil de réparation qui a été fixé.

5.4 Conclusion

Les ruptures de cohésion sont inévitables dans ce modèle de l'information. Cependant, leur amplitude peut être maîtrisée par un paramétrage fin du nombre de fragments d'une nuée. Lorsque ce paramétrage est « bon », c'est-à-dire, lorsque l'amplitude des ruptures de cohésion

est faible par rapport au seuil de réparation qui a été fixé, les ruptures temporaires que subit une nuée ne déclenchent pas inutilement un lourd processus de réparation. Par conséquent, une nuée bien paramétrée ne déclenchera des réparations que lorsque les pairs qui l'hébergent sont victimes de fautes. Nous avons montré par un ensemble d'expériences que, pour toute instance de réseau logique de type SCAMP, il existe un seuil en nombre de fragments à partir duquel l'amplitude des ruptures de cohésion est faible pour le seuil de réparation fixé. Ces expériences ont mis en évidence le fait que, par un mécanisme très simple de réparations successives, les nuées sont capables de trouver ce nombre de fragments idéal par l'exploration de leur espace d'états. Cette faculté d'adaptation est très intéressante notamment dans le fait que si les propriétés du réseau sont amenées à changer au cours du temps, les nuées vont pouvoir adapter leur taille en conséquence. La nature de ces résultats expérimentaux laisse envisager l'existence d'une loi régissant le nombre de fragments idéal pour une nuée par rapport aux paramètres de son environnement et au seuil de réparation. Une telle loi permettrait d'éviter la phase initiale de recherche de l'équilibre (entre 0 et 1500 cycles) durant laquelle les nuées sont plus vulnérables aux ruptures de cohésion que lorsque cette recherche est terminée. Les tables de résultats présentés dans ce chapitre nous donnent néanmoins une première estimation des paramètres vers lesquels une nuée va ensuite tendre. Enfin, nos observations tendent à montrer que plus la taille du réseau augmente et plus le nombre de fragments – nécessaires pour conserver des ruptures de cohésion de faible amplitude – augmente en proportion. Ce phénomène peut potentiellement poser des problèmes de passage à l'échelle du système si l'explosion du nombre de fragments n'est pas maîtrisée. Cette question sera abordée en partie dans le Chap. 7, consacré à la réduction de cette explosion du nombre de fragments d'une nuée.

Chapitre 6

Placement dynamique et décentralisé de nuées d'agents mobiles

Sommaire

6.1 Adaptabilité des nuées d'agents mobiles

6.1.1 Contexte

Maintenant qu'il est possible de déployer des nuées d'agents mobiles dans des réseaux pour stocker des données de manière décentralisée (cf. Chap. 4) et que ces nuées sont dotées de mécanismes pour assurer la disponibilité qui est attendue (cf. Chap. 5), nous nous intéressons à leur capacité d'adaptation à l'environnement. Nous avons déjà constaté qu'une nuée peut se servir de sa dynamique pour trouver les paramètres de fragmentation adéquats au réseau dans lequel elle évolue. Nous souhaitons aller plus loin dans ce chapitre en montrant que la mobilité et la flexibilité des nuées ouvre réellement la porte à des optimisations distribuées

résultant de l'exploration de l'environnement (à condition que cet environnement fournisse les informations nécessaires). L'adaptabilité est vue ici comme la faculté que possède une nuée à ajuster son placement, son comportement, sa taille, sa vitesse, etc., en réponse à la structure de l'environnement qu'elle explore et en fonction de critères qu'elle doit optimiser.

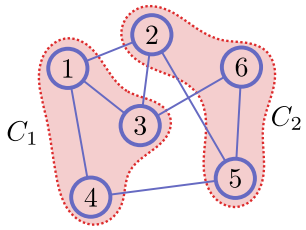
6.1.2 Application à la problématique des fautes corrélées

Ce chapitre se situe au niveau de la problématique des fautes corrélées que nous avons déjà abordée à la Sec. 2.5.2. Pour rappel, une faute corrélée est une faute qui a pour conséquence de déconnecter plusieurs pairs en même temps. Si plusieurs fragments d'une donnée sont hébergés sur des pairs corrélés, alors, une seule faute suffira pour engendrer la perte simultanée de plusieurs fragments. Nous avons vu que la plupart des plateformes de stockage décentralisé existantes font l'hypothèse d'indépendance des fautes pour des questions de simplicité de modélisation. Or, les fautes sont bel et bien corrélées dans les réseaux résultant en une dégradation de la disponibilité attendue sous hypothèse d'indépendance. Les corrélations devraient donc être prises en compte systématiquement. Nous avons vu qu'il existe deux approches pour traiter le problème des corrélations : l'approche introspective et l'approche analytique. Dans la première, le système est décrit finement pour effectuer un clustering de machines corrélées (i.e., chaque cluster contient des pairs corrélés). Une fois qu'un ensemble de clusters est calculé, la répartition des fragments d'un même document est faite sur des clusters distincts par un serveur central : le *Disseminator*. Dans la seconde, des modèles probabilistes du taux de corrélation sont donnés *a priori* pour adapter les schémas de redondance utilisés.

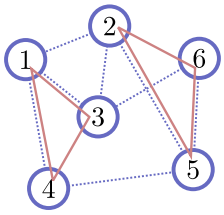
Nous faisons le choix, dans ce chapitre, de nous placer dans le cadre de l'approche par clustering. Notre proposition consiste à supprimer l'entité centrale *Disseminator* de [Weatherspoon *et al.*, 2002], qui est un point de vulnérabilité, et à tirer partie de la mobilité du modèle de flocking pour permettre à chaque nuée de trouver la meilleure place par une exploration des clusters ; le tout de manière décentralisée. Nous supposons, pour cela, qu'à l'issue du processus de clustering [Weatherspoon *et al.*, 2002, Kondo *et al.*, 2008], un ensemble de clusters de pairs corrélés est construit et que les pairs d'un même cluster se connaissent entre eux, formant ainsi un second réseau logique au-dessus du réseau pair-à-pair (cf. Fig. 6.1). Notre contribution consiste à proposer un algorithme décentralisé de déplacement de nuée qui :

1. minimise le nombre de fragments d'une nuée sur les mêmes clusters pour amortir l'impact des fautes correllées ;
2. répartisse le coût de stockage des nuées sur l'ensemble du réseau ;
3. conserve les propriétés du flocking.

Nous réalisons ce placement multi-critère à l'aide d'une procédure de recuit simulé distribuée selon la dynamique de Metropolis [Metropolis *et al.*, 1953], qui requiert la définition d'une fonction donnant l'énergie d'une nuée dans une certaine configuration. Cette fonction d'énergie doit contenir les différents critères que nous venons d'énoncer. La force de cette approche vient du fait que la fonction d'énergie globale peut être transformée pour être calculée localement



(a) Les cluster C_1 et C_2 matérialisent respectivement le fait que les pairs {1,4,3} et les pairs {2,6,5} sont corrélés.



(b) Le clustering peut être vu comme un second niveau de réseau logique (la modélisation sous forme d'hypergraphe de cette nouvelle couche de réseau logique semble plus pertinente).

FIGURE 6.1 – Exemple de clustering de pairs corrélés.

par les pairs nous permettant ensuite de construire un algorithme totalement décentralisé. Le principe de l'algorithme consiste à ajouter une couche de prise de décision au dessus du flocking pour conserver ses propriétés. C'est-à-dire que les possibilités de déplacements d'un agent sont toujours celles résultantes de l'algorithme de flocking dans *candidates*. Mais le choix dans cet ensemble est influencé pour que les critères énoncés plus haut soient optimisés.

6.2 La dynamique du recuit simulé et sa distribution

La mécanique du recuit simulé selon la dynamique de Metropolis [Metropolis *et al.*, 1953, Kirkpatrick *et al.*, 1983, Lamotte, 1992, Bertsimas et Tsitsiklis, 1993] est une procédure itérative qui consiste à trouver une configuration minimisant une fonction d'énergie $\mathbf{E}$, définie sur un espace de configurations S. La nature d'une configuration est déterminée par le problème et par la modélisation du système[21]. La fonction d'énergie $\mathbf{E}$ permet de donner, pour chaque configuration, une mesure de sa « proximité » avec l'optimal ; l'optimal étant la configuration qui minimise $\mathbf{E}$. Ensuite, en partant d'une configuration initiale quelconque $d_0 \in S$, une configuration candidate $d_1 \in S$ choisie dans le voisinage $v(d_0)$ de d_0 est acceptée avec une certaine probabilité qui décroît en fonction de $\mathbf{E}(d_1) - \mathbf{E}(d_0)$. Cette étape est répétée à partir de la configuration choisie jusqu'à ce qu'un minimum global de $\mathbf{E}$ soit trouvé. La possibilité d'accepter une configuration candidate ayant une énergie plus forte, avec une certaine probabilité, permet à l'algorithme d'éviter de rester bloqué dans des minima locaux. Cet algorithme stochastique est en rupture avec les algorithmes de descente déterministes qui peuvent se retrouver bloqués dans un minimum local en choisissant systématiquement la meilleure configuration dans leur voisinage. Ce problème du minimum local est illustré sur l'exemple de la Fig. 6.2. Dans cet exemple, quand le système est dans la configuration d_1, une descente déterministe retournera le minimum de la fonction $\mathbf{E}$ dans le voisinage $v(d_1)$. Ce minimum est alors un minimum local de

21. Par exemple si l'on cherche à trouver un positionnement optimal d'un ensemble de particules dans un espace, il est possible de prendre comme configuration l'ensemble des coordonnées de chacune des particules à un instant donné.

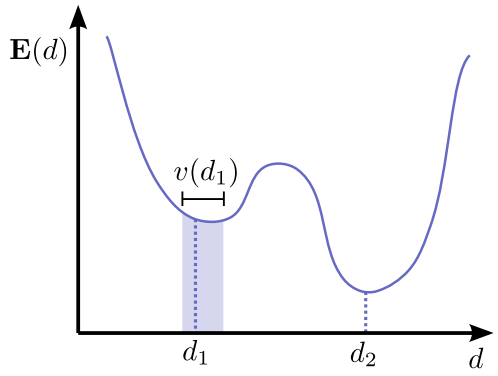


FIGURE 6.2 – Configuration dans laquelle le voisinage de d_1 est trop restreint pour pouvoir sortir du minimum local par une recherche déterministe.

la fonction $\mathbf{E}$ et la solution retournée est incorrecte puisque $\mathbf{E}$ prend son minimum en d_2.

Pour trouver le minimum de la fonction $\mathbf{E}$, l'algorithme de Metropolis utilise la distribution de Gibbs, que l'on note $G_T(s)$. Cette loi est issue de la thermodynamique statistique et probabilise un espace S en prenant comme densité de probabilité la fonction :

$$G_T(s) = \frac{1}{Z_T} \exp\left(\frac{-\mathbf{E}(s)}{T}\right) \tag{6.1}$$

avec Z_T la constante de normalisation :

$$Z_T = \sum_{s_i \in S} \exp\left(\frac{-\mathbf{E}(s_i)}{T}\right) \tag{6.2}$$

et avec le paramètre $T > 0$, appelé température du système. Lorsque T tend vers 0, la distribution G_T a la propriété de se concentrer uniformément sur l'ensemble de ses minima globaux.

L'algorithme de Metropolis repose sur une chaîne de Markov $x_T(\tau)$ définie sur l'ensemble des configurations S dans laquelle la température $T(\tau)$ est constante. L'ensemble des états de la chaîne correspond à l'ensemble des configurations possibles. Les états s'_i accessibles à partir d'un état s sont propres au problème et nécessitent la définition d'une fonction de voisinage sur les configurations. Si l'état courant de la chaîne $x(\tau)$ est égal à i, le choix d'un voisin candidat (i.e., une configuration candidate) j de i est régi par une certaine probabilité q_{ij}. La probabilité de transition entre deux états i,$j \in S$ dans la chaîne est donnée par :

$$\mathbb{P}[x(\tau+1) = j | x(\tau) = i] = q_{ij}\, min\left(\exp\Big(-\frac{\mathbf{E}(j) - \mathbf{E}(i))}{T}\Big), 1\right) \tag{6.3}$$

si $i \neq j$. Autrement dit, $\mathbb{P}[x(\tau+1) = j | x(\tau) = i]$ est la probabilité de désigner une certaine configuration candidate j (avec q_{ij}) et d'accepter cette configuration candidate (le second facteur de l'Eq. 6.3). La chaîne $x_T(\tau)$ est réversible, irréductible et apériodique et sa distribution de probabilité est donnée par l'Eq. 6.1. Cette chaîne possède alors la propriété suivante :

$$\lim_{\substack{\tau \to +\infty \\ T \text{fixé}}} \mathbb{P}_T(x(\tau) = s) = G_T(s), \quad s \in S \tag{6.4}$$

C'est-à-dire que la chaîne converge vers la distribution de Gibbs et que l'algorithme régi par cette chaîne converge, lui-même, vers une configuration ayant une énergie minimale. Pour plus de détails se référer à [Bertsimas et Tsitsiklis, 1993].

6.3 MinCor : minimiser le nombre de fragments sur des pairs corrélés

Cette section est consacrée à MinCor [Romito et Bourdon, 2012a, Romito et Bourdon, 2012b], l'algorithme décentralisé que nous proposons pour réaliser le déplacement adaptatif des nuées sur les clusters de corrélations (i.e., minimiser le nombre de fragments d'une nuée sur les même clusters de corrélation). La construction de cet algorithme de recuit simulé distribué passe par plusieurs étapes :

1. la modélisation du système que nous considérons ;
2. la définition d'une fonction d'énergie de nuée qui intègre nos critères à optimiser ;
3. le découpage de cette fonction d'énergie pour qu'elle puisse être calculée localement et de manière asynchrone par chaque agent ;
4. le choix du meilleur candidat parmi les candidats retournés par l'algorithme de flocking.

6.3.1 Définition du modèle

Le modèle que nous définissons est constitué du graphe non-orienté $\Gamma = (V,E)$ modélisant le réseau pair-à-pair avec V l'ensemble des pairs [22] et E l'ensemble des arrêtes. Un ensemble d'agents mobiles fragments F sont en évolution dans Γ sous la forme d'un ensemble de nuées N tel que :

$$\bigcap_{n_i \in N} n_i = \emptyset \quad \text{et} \quad \bigcup_{n_i \in N} n_i = F \tag{6.5}$$

Il ne peut donc y avoir ni de fragments orphelins ni de fragments appartenant à plusieurs nuées en même temps. A l'issue du clustering, un ensemble G de clusters de corrélations est généré. Les éléments $g \in G$ sont des sous-ensembles de V tel que pour tout $u,w \in g$, u est corrélé avec w. C'est-à-dire que lorsqu'un pair subit une faute, tous les pairs de son cluster sont déconnectés. Nous considérons dans la suite que les clusters sont disjoints. Nous définissons un certain nombre de fonctions pour manipuler ce modèle :

Définition 33 *Soit $a : F \to V$ la fonction qui retourne le pair hébergeant un fragment. $u = a(f)$ ssi f est hébergé par u. Soit $\{f_1,\ldots,f_m\} \in N$ une nuée de m fragments. On note par f_j^u que f est le fragment d'identifiant j et qu'il est hébergé sur le pair u. On a donc $a(f_j^u) = u$. On appelle configuration α d'une nuée $n \in N$ une certaine affectation des fragments de n sur les pairs de V que l'on note $n_\alpha = \{(f,u) \in F \times V \mid u = a(f), \forall f \in n\}$.*

22. À ne pas confondre avec v la fonction retournant le voisinage d'un pair cf. Def. 25, Sec. 4.2.1.

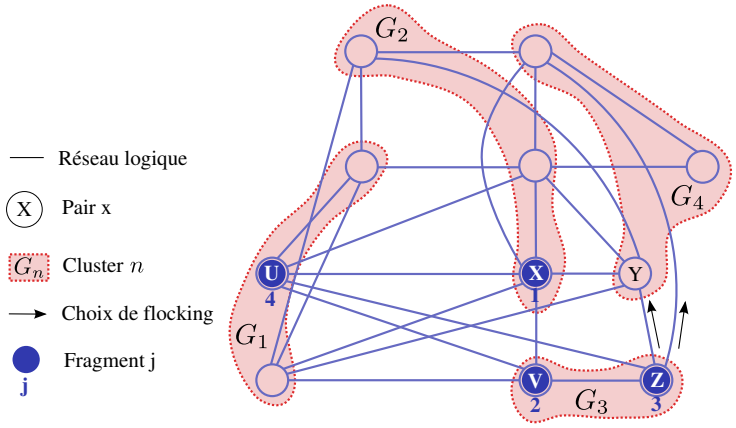


FIGURE 6.3 – Nuée de 4 fragments dans un réseau de 11 pairs avec 4 clusters de corrélations.

Définition 34 *Soit* $t : G \times N \to 2^V$, $g,n \mapsto g \cap \{a(f), \forall f \in n\}$. *La fonction qui retourne les pairs du cluster* g *qui hébergent des fragments de la nuée* n.

Définition 35 *Soit* $clus : V \to G$. *La fonction qui retourne le cluster* g *dans lequel se trouve un pair* u. *On a* $g = clus(u)$ *ssi* $u \in g$.

Définition 36 *Soit* C *un ensemble de fonctions de coût définies de* $F \to \mathbb{R}$ *représentant le coût d'hébergement d'un fragment. On pourrait, par exemple, définir le coût de stockage* $\mathcal{V} \in C$ *d'un fragment* f^u *comme suit :*

$$\mathcal{V}(f^u) = \frac{\mathcal{O}(u) + \mathcal{T}(f^u)}{\mathcal{E}(u)}$$

avec $\mathcal{O}$, $\mathcal{E}$ *et* $\mathcal{T}$ *des fonctions qui retournent respectivement : l'espace occupé d'un pair, l'espace total disponible sur un pair et la taille d'un fragment.*

Exemple 37 *La Fig. 6.3 présente un exemple d'une nuée de 4 fragments, répartie sur un réseau de 11 pairs, eux-mêmes répartis sur 4 clusters de corrélation. La nuée est dans la configuration* $\alpha = \{(1,x),(2,v),(3,z),(4,u)\}$ *(i.e., le fragment d'identifiant 1 est sur le pair d'identifiant* x, *le fragment d'identifiant 2 est sur le pair d'identifiant* v, *etc.).*

6.3.2 Fonction d'énergie d'une nuée

Nous définissons, à présent, la fonction d'énergie globale d'une nuée $n \in N$ dans un réseau pair-à-pair suivante :

$$\mathbf{E}(n_\alpha) = \sum_{f \in n} \sum_{c \in C} c(f) + w \sum_{g \in G} |t(g,n)|^\gamma \tag{6.6}$$

Cette fonction affecte à chaque configuration de nuée une valeur numérique permettant de mesurer la qualité de la configuration. L'énergie d'une nuée dans une configuration α est une quantité qui est égale au coût d'hébergement de ses fragments ajoutée à la proportion de ses fragments sur des pairs corrélés. Plus le nombre de fragments d'une même nuée sur le même cluster (le

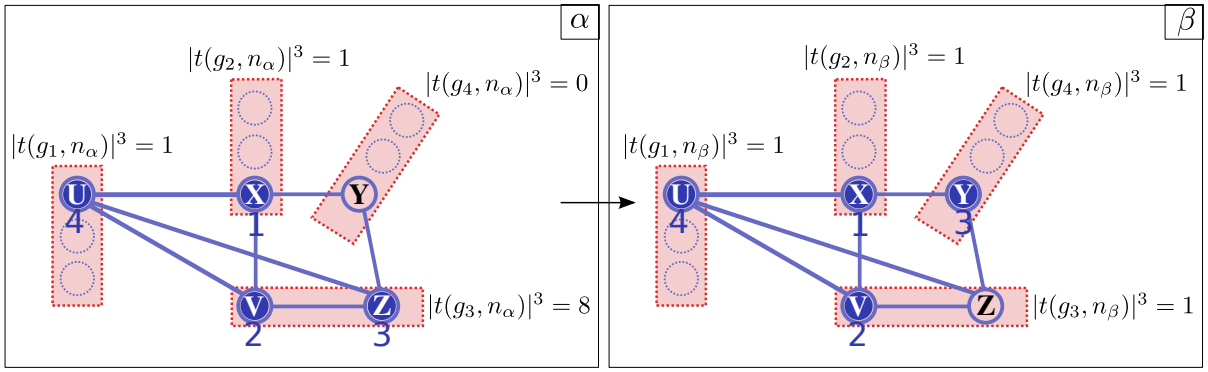


FIGURE 6.4 – Transition de la configuration α à la configuration β par déplacement de f_3. Le graphe présenté est le même que celui de la Fig. 6.3 mais dans une forme filtrée.

terme $|t(g,n)|^\gamma$) est élevé et plus l'énergie résultante est élevée. La constante $\gamma \in \mathbb{R}^+$ permet d'affecter une pénalité aux clusters contenant beaucoup de fragments de la même nuée. $w \in \mathbb{R}^+$ est une constante permettant au concepteur du système de moduler l'importance du critère de répartition sur des clusters disjoints par rapport au critère d'équilibrage de charge.

Le principe du changement de configuration d'une nuée nécessaire à l'algorithme de recuit simulé est présenté à la Fig 6.4. Le passage de la configuration α à la configuration β est réalisé en déplaçant le fragment 3 du pair z au pair y. Les candidats possibles pour les changements de configurations sont ceux retournés par l'algorithme de flocking (cf. Alg. 17, Sec. 4.2.4). La décentralisation de l'algorithme de recuit simulé est possible puisque le changement de configuration a un impact local sur le système. Cet impact est matérialisé par le déplacement d'un seul agent de la nuée à la fois, qui entraîne une perturbation limitée aux voisinages du déplacement. La fonction $\mathbf{E}$ peut donc être calculée de manière asynchrone par chaque agent sur une vision locale de sa nuée qui est limitée au voisinage du pair qui l'héberge. Ce principe est illustré à la Fig. 6.4 dans laquelle, le fragment f_3 hébergé par le pair z peut calculer localement en z :

$$\Delta(\mathbf{E}_z) = \mathbf{E}_z(n_\beta, f_3^y) - \mathbf{E}_z(n_\alpha, f_3^z) \tag{6.7}$$

C'est-à-dire, la différence d'énergie induite par le déplacement calculée localement sur le pair z. De manière plus générale, on exprime $\mathbf{E}_z(n_\alpha, f_i^z)$ et $\mathbf{E}_z(n_\beta, f_i^y)$ calculés par un fragment quelconque f_i comme suit :

$$\mathbf{E}_z(n_\alpha, f_i^z) = \sum_{f \in \mathcal{F}} \sum_{c \in C} c(f) + \sum_{c \in C} c(f_i^z) + w \sum_{g \in \mathcal{G}} |t(g, n_\alpha)|^\gamma + w|t(clus(y), n_\alpha)|^\gamma + w|t(clus(z), n_\alpha)|^\gamma \tag{6.8}$$

$$\mathbf{E}_z(n_\beta, f_i^y) = \sum_{f \in \mathcal{F}} \sum_{c \in C} c(f) + \sum_{c \in C} c(f_i^y) + w \sum_{g \in \mathcal{G}} |t(g, n_\beta)|^\gamma + w|t(clus(y), n_\beta)|^\gamma + w|t(clus(z), n_\beta)|^\gamma \tag{6.9}$$

avec

$$\mathcal{F} = \{f \in n \,|\, a(f) \in v(z) \cup v(y)\} \setminus f_i \tag{6.10}$$

l'ensemble des fragments de n sur les voisinages de z et y qui restent immobiles. Et

$$\mathcal{G} = \{clus(a(f)) \,|\, a(f) \in v(z) \cup v(y), \forall f \in n\} \backslash \{clus(z), clus(y)\} \tag{6.11}$$

l'ensemble des clusters qui possèdent des fragments de n dans le voisinage de y et z mais qui ne sont ni la cible ni la source du déplacement. On élimine les termes constants et on obtient :

$$\Delta(\mathbf{E}_z) = \sum_{c \in C} c(f_i^y) + w|t(clus(y), n_\beta)|^\gamma + w|t(clus(z), n_\beta)|^\gamma \\ - \sum_{c \in C} c(f_i^z) - w|t(clus(y), n_\alpha)|^\gamma - w|t(clus(z), n_\alpha)|^\gamma \tag{6.12}$$

Exemple 38 *Sur la Fig. 6.4 si l'on considère $C = \emptyset$, $w = 1$, $\gamma = 3$, et que f_3^z est le fragment souhaitant se déplacer on a : $\mathcal{F} = \{f_1, f_2, f_4\}$, $\mathcal{G} = \{g_1, g_2\}$, $\Delta(\mathbf{E}_z) = 4 - 10 = -6$. Donc un gain d'énergie à déplacer f_3 en y.*

6.3.3 Algorithme de placement

Tous les éléments sont en place pour construire l'algorithme 20 : MinCor. Cet algorithme décrit la procédure de déplacement par recuit simulé exécutée par chaque agent sur un pair z quelconque de manière asynchrone. L'appel de la fonction RandomFlockingCandidates(z) exécute l'algorithme de flocking (cf. Alg. 17, Sec. 4.2.4) et retourne un pair tiré aléatoirement dans la liste des pairs candidats pour le déplacement. L'appel de PeersOfCluster(y) retourne la liste des pairs du cluster de y. Cette fonction nécessite l'envoi de messages par le réseau entre le pair z et le pair y. Le fragment f_i^z en z va commencer par évaluer son coût de stockage dans la configuration α (la variable $cost_z$) et son coût de stockage dans la configuration β (la variable $cost_y$). Ensuite, ce même agent en z compte les fragments de sa nuée dans les clusters α (la variable t_z) et β (la variable t_y). La décision de changer de configuration est prise en suivant la dynamique de Metropolis à température constante T de l'Eq. 6.3 et utilise la fonction d'énergie $\mathbf{E}_z$ définie dans l'Eq. 6.12. Les variables t_y et t_z comptent les fragments de $\{n\} \backslash f_i$ qui sont respectivement dans le cluster de y et dans le cluster de z. C'est pourquoi $|t(clus(y), n_\alpha| = t_y$ et $|t(clus(y), n_\beta| = t_y + 1$ parce que dans la configuration β, f_i est passé dans le cluster de y. De même, $|t(clus(z), n_\beta| = t_z$ et $|t(clus(z), n_\alpha| = t_z + 1$ parce que dans la configuration α, f_i est dans le cluster de z. Si la configuration candidate est acceptée, alors, l'agent se déplace effectivement sur le pair y (AgentMove(y)) sinon il reste sur le pair z.

6.4 Évaluation de l'approche

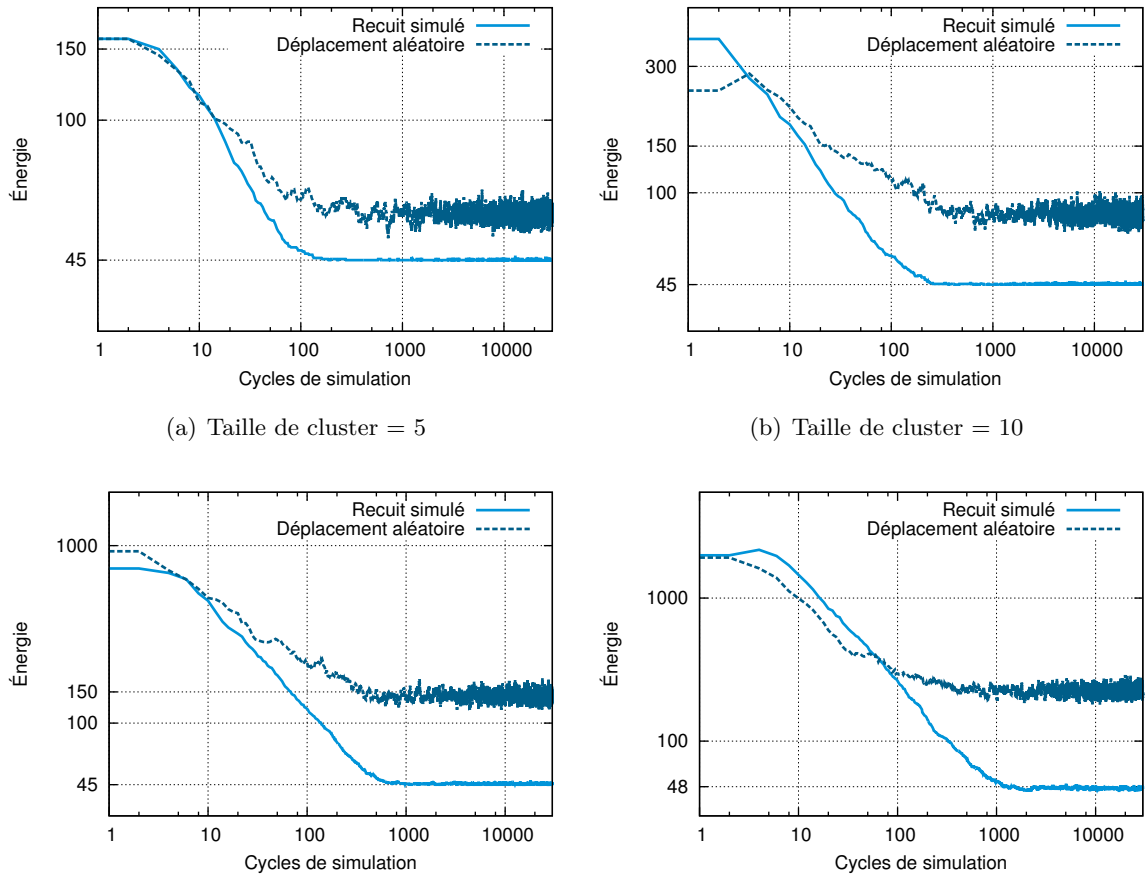
Nous proposons maintenant d'évaluer l'algorithme MinCor au moyen de trois expériences. Dans un premier temps, nous évaluons la qualité du placement par recuit simulé sur les clusters

Algorithme 20 : MinCor

Entrées : f_i^z l'agent fragment en z exécutant l'algorithme, n la nuée de f_i, C un ensemble de fonctions de coût, γ la constante de pénalité, w le facteur de gain

$y \longleftarrow$ RandomFlockingCandidates(z)
$cluster_z \longleftarrow$ PeersOfCluster(z)
$cluster_y \longleftarrow$ PeersOfCluster(y)
$cost_z \longleftarrow 0$
$cost_y \longleftarrow 0$
pour chaque $c \in C$ **faire**
  $cost_z \longleftarrow cost_z + c(f,z)$
  $cost_y \longleftarrow cost_y + c(f,y)$
$t_z \longleftarrow 0$
$t_y \longleftarrow 0$
pour chaque *peer* $\in cluster_z$ **faire**
  si *peer héberge un fragment* f_j *de* n **alors**
    si $j \neq i$ **alors**
      $t_z \longleftarrow t_z + 1$
pour chaque *peer* $\in cluster_y$ **faire**
  si *peer héberge un fragment* f_j *de* n **alors**
    si $j \neq i$ **alors**
      $t_y \longleftarrow t_y + 1$
$\Delta(\mathbf{E}_z) \longleftarrow cost_y + w\big((t_y+1)^\gamma + t_z^\gamma\big) - cost_z - w\big(t_y^\gamma - (t_z+1)^\gamma\big)$
$p \leftarrow \min\left(\exp\left(\frac{-\Delta(\mathbf{E}_z)}{T}\right),1\right)$
$tirage \leftarrow$ Random float choice in [0,1]
si $tirage \leq p$ **alors**
  AgentMove(y)

de corrélation par rapport au placement aléatoire de l'algorithme de flocking classique. Cette évaluation est faite en collectant les valeurs d'énergie de nuées exécutant ces deux algorithmes et en les comparant. Dans un second temps, nous comparons la distribution des déplacements d'une nuée utilisant l'algorithme MinCor avec celle d'une nuée aléatoire, nous permettant de donner des éléments de réponse sur l'impact d'un tel changement de politique sur l'occupation du réseau par les nuées. Enfin, dans un troisième temps, nous mesurons le nombre de fautes concurrentes obtenues avec MinCor en comparaison avec une approche mobile aléatoire et une approche classique immobile, lorsque le système subit des fautes corrélées. Cette expérience permet de quantifier le gain à utiliser l'algorithme MinCor par rapport aux autres approches. Ces expérimentations sont réalisées sur notre simulateur multi-agents et la durée d'une simulation reste fixée à 30000 cycles, où chaque cycle correspond à 60 secondes en temps réel, soit une du-



(a) Taille de cluster = 5

(b) Taille de cluster = 10

(c) Taille de cluster = 20

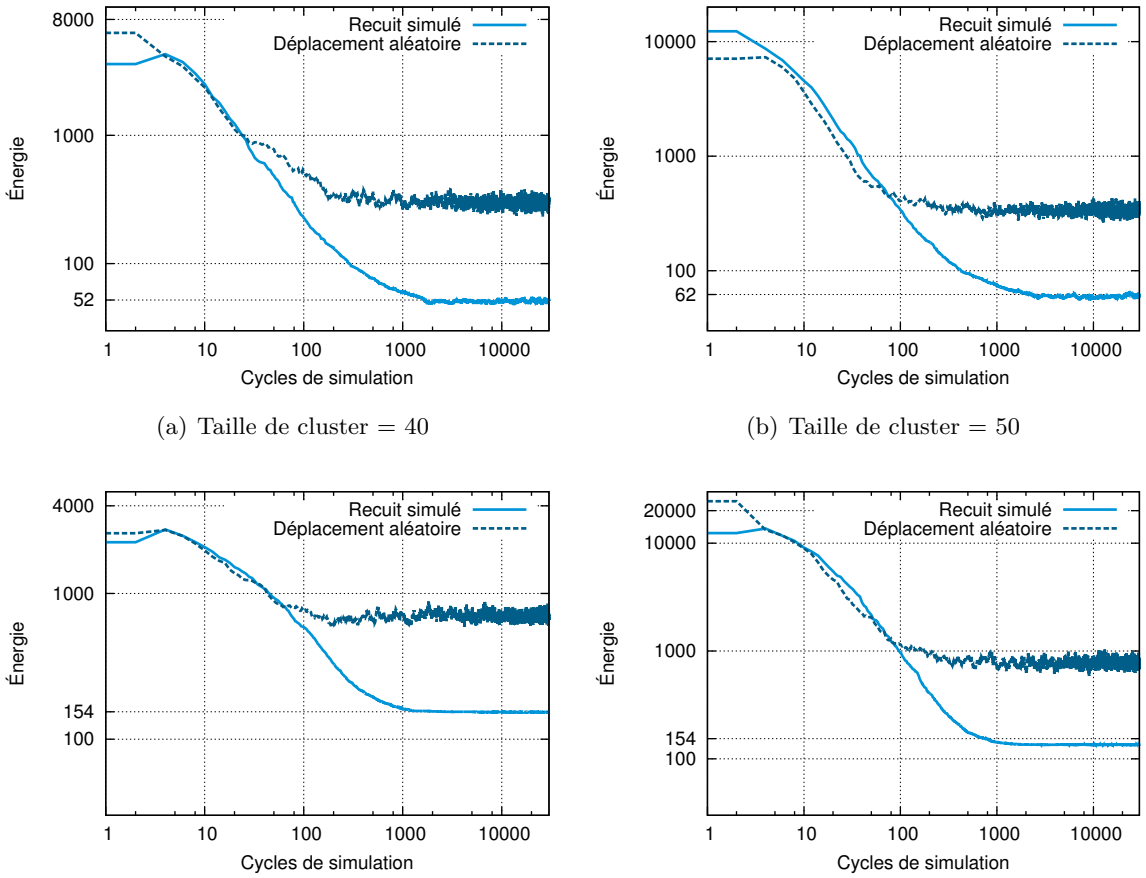
(d) Taille de cluster = 30

FIGURE 6.5 – Évolution de l'énergie moyenne des nuées pour le placement par flocking aléatoire et pour le flocking par MINCOR.

rée par simulation d'environ 21 jours. Chaque expérience a été reproduite 70 fois et ce sont les valeurs moyennes qui sont présentées. Le réseau pair-à-pair supportant l'architecture est de type (1000,20)-SCAMP (i.e., chaque pair a en moyenne 15% des pairs du réseau dans son voisinage). Les taux de transfert réseau restent contenus entre 64kb/s et 640kb/s et la taille d'un fragment reste également à 64Mo.

6.4.1 Qualité du placement

Dans cette première expérience, deux nuées de 45 fragments sont déployées dans le réseau. Cette taille de nuée a été choisie en adéquation avec les résultats que nous avons observés à la Sec. 5.3 et garantit que, pour l'instance (1000,20)-SCAMP, la cohésion de la nuée ne sera perturbée que par les fautes qui surviennent et non par la structure du réseau. Les décisions de mouvement sont prises par chacun des agents à chaque cycle de simulation avec une probabilité de 0.1. La première nuée se déplace avec l'algorithme classique (l'algorithme 17 de déplacement



(a) Taille de cluster = 40

(b) Taille de cluster = 50

(c) Taille de cluster = 80

(d) Taille de cluster = 90

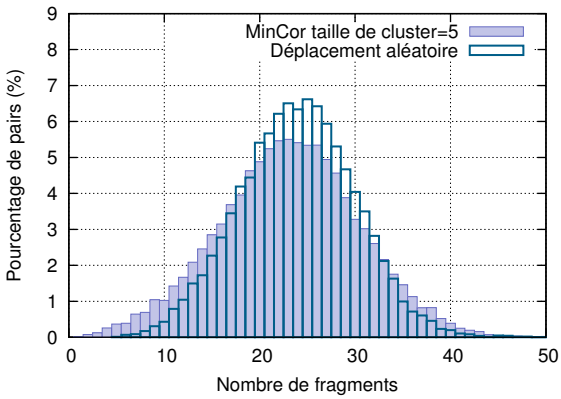
FIGURE 6.6 – Évolution de l'énergie moyenne des nuées pour le placement par flocking aléatoire et pour le flocking par MINCOR (suite).

aléatoire) et l'autre utilise l'algorithme MinCor avec les paramètres $w = 1$, $\gamma = 3$, $C = \emptyset$ et $T = 1$. Cette première expérience s'intéresse à la qualité du placement sur les clusters de corrélation. C'est pour cette raison que la composante de coût d'hébergement dans la fonction d'énergie est ignorée. Nous mesurons l'énergie des deux nuées au cours de la simulation pour des clusters de tailles 5, 10, 20, 30, 40, 50, 80, 90 et 100. Une taille de cluster de 5 dans un réseau de 1000 pairs signifie qu'il y a un total de 200 clusters. Les résultats des Fig. 6.5 et 6.6, en échelle logarithmique, montrent l'évolution moyenne de l'énergie des deux nuées au cours des simulations. Nous observons d'une manière générale sur l'ensemble des graphiques, que l'énergie des nuées évoluant avec MinCor est plus faible que celles évoluant de manière aléatoire avec l'algorithme de flocking classique. Nous remarquons également que lorsque le nombre de clusters est supérieur ou égal au nombre de fragments Fig. 6.5(a), Fig. 6.5(b) et Fig. 6.5(c), l'énergie du placement par recuit est égale au nombre de fragments de la nuée. Ceci montre que la nuée a été capable de placer chacun de ses fragments dans des clusters distincts tout en conservant sa localité (i.e., trouver sa configuration d'énergie minimale). On note, à ce propos, que l'énergie de ces nuées ne pourra jamais être inférieure à 45. On constate ensuite à la Fig. 6.5(d) et à la Fig. 6.6(a) que lorsque le nombre de clusters devient inférieur à 45, le minimum d'énergie atteint avec MinCor devient inévitablement supérieur à 45. Dans ces cas, la nuée n'a pas réussi à placer la totalité de ses fragments sur des clusters distincts. D'une manière générale, plus la taille des clusters augmente et plus le minimum d'énergie atteint par la nuée est élevé. Finalement, on remarque les fortes valeurs d'énergie initiales. Ces fortes valeurs proviennent du fait que dans notre implémentation, le déploiement de chaque nuée est effectué à partir d'un pair initiateur. De ce fait, tous les fragments au démarrage se retrouvent dans le même cluster avant de commencer à se déplacer, générant ainsi une valeur d'énergie anormalement élevée.

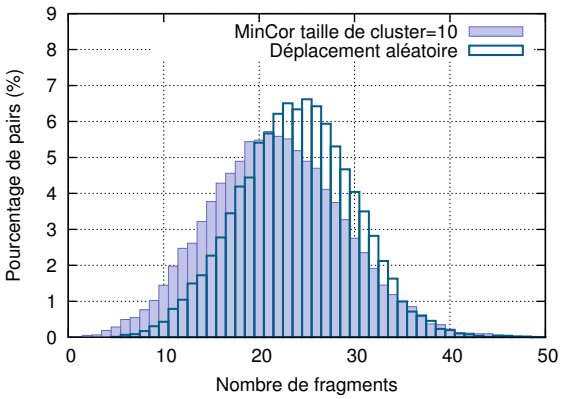
En conclusion, nous montrons avec cet ensemble d'expérimentations qu'une nuée exécutant l'algorithme MinCor est capable de trouver une répartition qui minimise le nombre de fragments sur des clusters de pairs corrélés de manière asynchrone et décentralisée. Dans ces expériences, les clusters ont la même taille. Dans le cas où les clusters seraient de tailles variables, les résultats n'ont aucune raison de différer ; i.e., seul le nombre de clusters a une influence sur le placement.

6.4.2 Répartition des déplacements

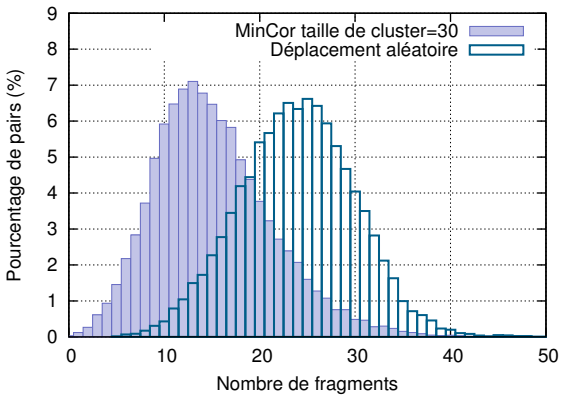
Maintenant que nous savons que l'algorithme MinCor réalise le placement attendu pour la fonction d'énergie que nous avons énoncée, nous proposons de mesurer l'impact d'une telle politique de flocking sur la distribution des déplacements d'une nuée. Chaque graphique de la Fig. 6.7 présente, pour une taille de cluster donnée, la distribution des déplacements d'une nuée évoluant avec l'algorithme MinCor et d'une nuée évoluant avec l'algorithme aléatoire. Nous constatons, tout d'abord que, pour n'importe quelle taille de cluster, la distribution des déplacements d'une nuée utilisant MinCor est différente de celle d'une nuée classique. Plus précisément, une nuée MinCor a tendance à effectuer moins de déplacements qu'une nuée aléatoire. Par exemple, à la Fig. 6.7(c), 65% des pairs ont hébergés 10 fragments pour MinCor



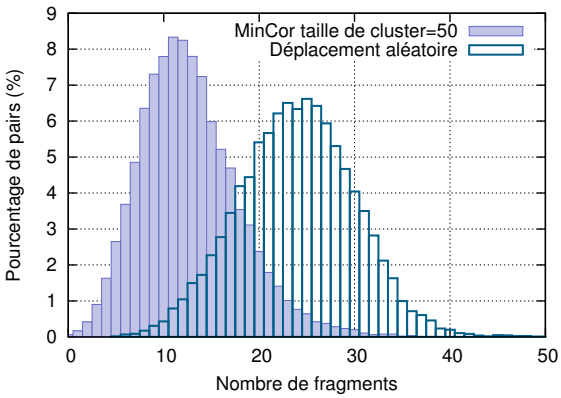
(a) Taille de cluster = 5



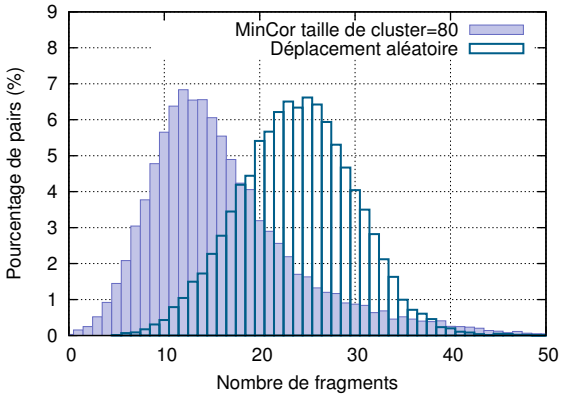
(b) Taille de cluster = 10



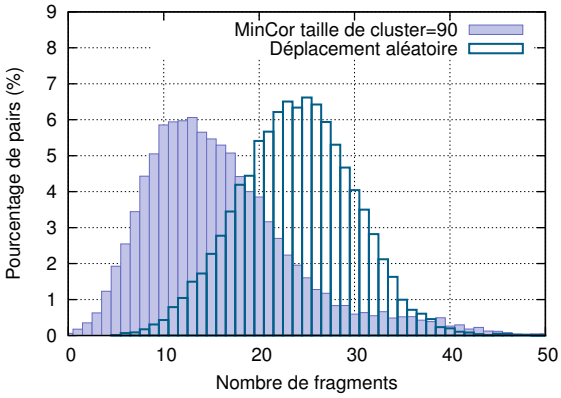
(c) Taille de cluster = 30



(d) Taille de cluster = 50



(e) Taille de cluster = 80



(f) Taille de cluster = 90

FIGURE 6.7 – Couverture réseau de nuées se déplaçant avec l'algorithme MINCOR.

contre 22 pour un déplacement aléatoire. Ceci vient du fait que plus une nuée MinCor arrive proche de son équilibre (i.e., sur son minimum) et moins elle se déplace. On remarque néanmoins que l'écart entre les deux distributions est très faible lorsque le nombre de clusters est suffisant pour que la nuée place un fragment par cluster (cf. Fig. 6.7(a) et 6.7(b). Il semblerait, dans ce cas, qu'étant donné la multitude de minima possibles pour **E**, la nuée conserve la même *vélocité* que lors d'un déplacement aléatoire.

6.4.3 Mesure des fautes concurrentes

Nous proposons, à présent, de mesurer le nombre de pertes de fragments simultanées obtenues pour des nuées de 45 fragments lorsque les fautes sont corrélées. La première nuée se déplace en suivant l'algorithme de flocking classique, la seconde cherche un placement initial et ne se déplace plus ensuite. La troisième utilise MinCor pour ses déplacements. Une nuée immobile représente une approche classique que nous utilisons pour comparer les performances de nos algorithmes avec l'existant. Nous introduisons ensuite un modèle de fautes qui déconnecte un pair p tous les $faultstep$ cycles. La faute a pour conséquence la déconnexion de tous les pairs du cluster de p et de tous les fragments qui y étaient hébergés. La première faute survient après le 1000^e cycle pour laisser le temps à MinCor de converger. Le réseau est ensuite placé dans un état stationnaire : pour chaque pair déconnecté au cycle τ, un nouveau pair est reconnecté au cycle $\tau + 1$. Le nombre de clusters est préservé par un clustering périodique du réseau lorsque suffisamment de pairs se sont reconnectés. Chaque nuée est fragmentée avec un (10,35)-code d'effacement et un seuil de réparation $r = 13$ est défini. Les mécanismes de supervision et de réparation décrits au Chap. 5 sont activés. En d'autres termes, lorsque la taille d'une nuée atteint r à la suite de plusieurs fautes, une procédure de réparation est lancée pour restaurer les 45 fragments. Nous mesurons dans les graphiques de la Fig. 6.8 le nombre moyen de pertes simultanées de fragments pour chaque type de nuée. i.e., lorsqu'une nuée subit une faute, on compte le nombre de fragments qu'elle a perdus suite à cette faute. Plus le nombre de fautes concurrentes est faible et plus la nuée correspondante est tolérante aux fautes corrélées. Ces résultats sont présentés pour plusieurs variations du $faultstep$ et de la taille des clusters. On constate, dans un premier temps, que MinCor est le meilleur algorithme de placement et qu'une nuée prenant ses décisions avec cet algorithme perd moins de fragments lorsqu'elle subit une faute que les autres nuées dans quasiment toutes les situations. Le comportement erratique, dans certains cas, de la courbe représentant le placement immobile s'explique par le fait qu'une nuée immobile ayant un grand nombre de ses fragments placés dans le même cluster a de plus grandes chances de perdre plus de fragments d'un coup. Le résultat est donc dépendant du choix de placement initial. Ce phénomène est naturellement amplifié pour les clusters de grande taille. On remarque également que les courbes sont décroissantes en fonction du $faultstep$. L'explication provient du fait que les nuées sont beaucoup plus perturbées durant leur phase de réparation lorsqu'il y a beaucoup de fautes. En effet, le processus de réparation duplique m agents (provenant du (m,n)-code d'effacement) et les envoie sur un pair *cible* pour reconstruire les fragments manquants. Mais,

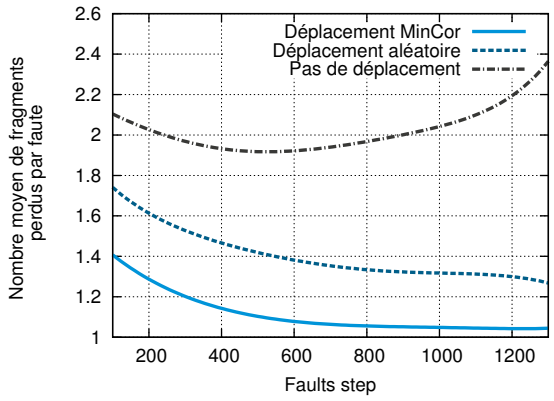
si le pair *cible* est perdu pendant la réparation, les agents dupliqués qui sont déjà arrivés se retrouvent également perdus, résultant en un nombre de pertes simultanées plus important. Finalement, on observe qu'une nuée qui se déplace aléatoirement selon l'algorithme classique fournit de meilleurs résultats qu'un placement immobile dans tous les cas, à l'exception d'une taille de cluster de 80. Ce résultat, assez surprenant, montre que sa mobilité à elle seule semble suffisante pour éviter certaines corrélations. Enfin, on note qu'il est normal d'avoir un nombre de fautes décroissant en fonction du $faultstep$ puisque la durée de la simulation est fixée.

6.5 Conclusion

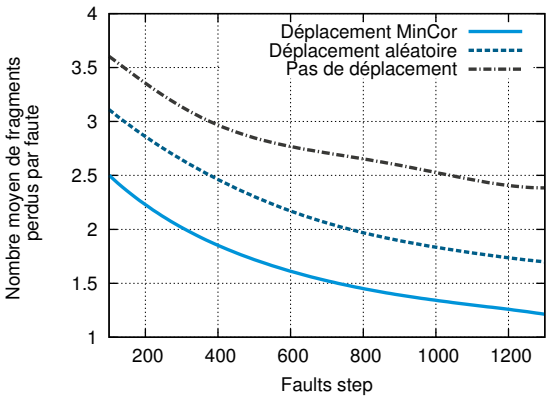
Ce chapitre illustre un second mécanisme d'adaptation d'une nuée d'agents mobiles à son environnement [23]. Cette adaptation réalisée à l'aide de l'algorithme MinCor, permet à une nuée de chercher, de manière asynchrone et décentralisée, un placement qui minimise le nombre de ses fragments sur des pairs corrélés. L'ensemble des expériences que nous avons menées montrent que ces nuées sont capables de trouver un placement optimal qui leur confère une meilleure résistance aux fautes corrélées. L'algorithme de recuit-simulé, suivant la dynamique de Metropolis, lorsqu'il est arrivé dans un des minimums globaux de la fonction d'énergie, est conçu pour accepter de changer d'état vers un autre minimum. Cette propriété de la politique de déplacement permet à une nuée de rester mobile même si elle a convergé vers un des minimums de sa fonction d'énergie. En pratique, il existe une grande quantité de minimums puisque, pour une configuration α d'énergie minimale, toute permutation des fragments de la nuée sur les pairs de α est également une configuration d'énergie minimale. Une nuée reste donc perpétuellement en mouvement. Néanmoins, sa vélocité s'en trouve légèrement réduite, et les expériences sur la distribution des déplacements l'ont confirmé. Ce résultat était prévisible puisque les nuées aléatoires sont, de fait, moins contraintes dans leurs déplacements.

Au delà de la problématique des fautes corrélées, l'algorithme de recuit-simulé distribué que nous avons présenté dans ce chapitre permet de contrôler les déplacements d'une nuée de manière décentralisée, à condition que les critères encapsulés dans cette fonction soient calculables localement sur chaque pair. L'idée derrière ce type d'optimisation consiste à faire de l'exécution décentralisée à partir d'informations qui sont réparties à travers le réseau. C'est également le cas avec la composante de coût d'hébergement que nous avons laissée de côté dans nos expérimentations. Cette information est répartie à travers le réseau et c'est en se déplaçant que la nuée résout le problème d'équilibrage de charge de manière décentralisée. Si nous revenons, à présent, sur la problématique de la minimisation des corrélations, nous avons fait l'hypothèse que les clusters sont disjoints. Il serait intéressant de considérer le cas de clusters non disjoints faisant intervenir une notion de probabilité de co-occurrence des fautes. Cette modélisation, permettrait de modéliser les différentes sources de corrélations. Par exemple, un pair A peut être corrélé avec un pair B sur un type de fautes T_1. Mais le pair A peut également être corrélé avec un pair C

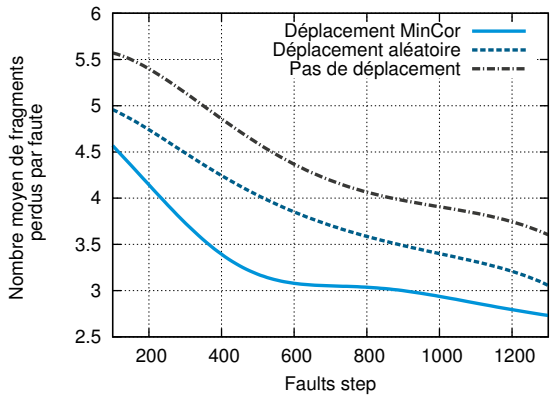
23. Le premier mécanisme étant celui permettant à une nuée de trouver les paramètres de fragmentation à appliquer en fonction de l'instance de réseau considérée (cf. Chap. 5).



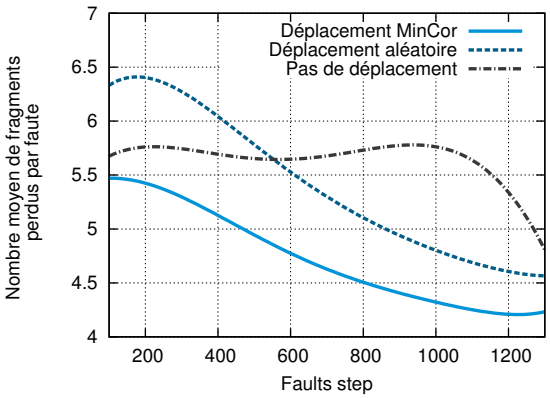
(a) Taille de cluster = 10



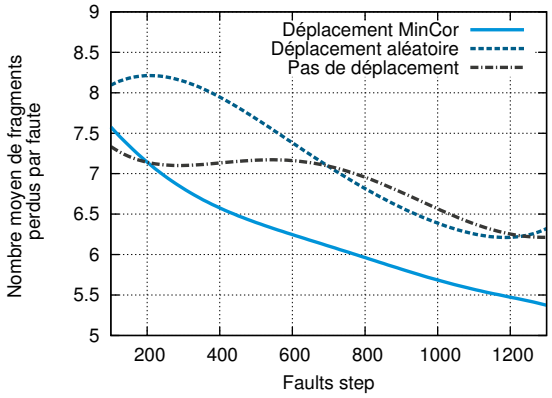
(b) Taille de cluster = 20



(c) Taille de cluster = 40



(d) Taille de cluster = 60



(e) Taille de cluster = 80

FIGURE 6.8 – Nombre moyen de fragments perdus à chaque fois qu'un nuée subit une faute.

sur un type de faute T_2 alors que le pair B n'est pas corrélé avec le pair C. Maintenant, si A est victime d'une faute, B va également être victime de cette faute si elle est de type T_1 mais pas si elle est de type T_2. La corrélation de A avec B est alors probabiliste et semble plus réaliste (par exemple A peut-être dans le même bâtiment que B tout en partageant une vulnérabilité avec le pair C situé dans un autre lieu géographique). Quelle serait donc la stratégie de placement à adopter avec une telle modélisation ?

Chapitre 7

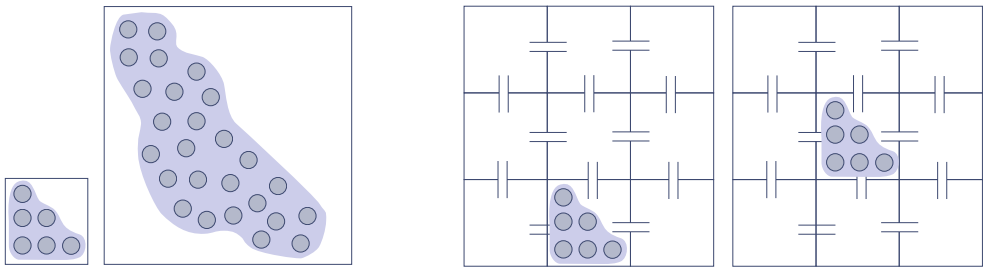
Passage à l'échelle

Sommaire

7.1 Explosion du nombre de fragments

Nous avons vu au Chap. 5 qu'une nuée est capable de trouver ses paramètres de fonctionnement par l'exploration de son environnement. L'ensemble des expériences que nous avons réalisées ont montré que plus la taille du réseau augmente, plus la taille des nuées (i.e., leur nombre d'agents) augmente en conséquence pour éviter les ruptures de cohésion de forte amplitude. Cette croissance peut être limitée, pour une taille de réseau donnée, en augmentant le degré du réseau. Cependant, pour des questions de passage à l'échelle du réseau logique, le degré moyen ne devrait pas croître de manière trop importante. De même, au niveau multi-agents, le nombre d'agents par nuée devrait être borné à une certaine valeur qui dépend des capacités de l'infrastructure. Nous nous retrouvons alors confrontés à un verrou technologique puisque, dans la situation actuelle, la croissance du nombre d'agents ne peut pas être maîtrisée lorsque la taille du réseau augmente.

De la même manière que la structuration des réseaux totalement décentralisés a permis de solutionner les problèmes de performance posés par les algorithmes de recherche dans les réseaux pair-à-pair totalement décentralisés, la proposition que nous faisons pour permettre de limiter cette explosion du nombre d'agents, consiste à modifier la structure logique du réseau.



(a) (b)

FIGURE 7.1 – Pour une densité fixée et si aucune rupture de cohésion n'est tolérée, le nombre d'agents nécessaires pour relier de bout en bout les deux extrémités d'un espace est proportionnel à la taille de cet espace. Il est clair que dans la Fig. 7.1(a), la petite nuée de gauche n'a pas l'élasticité suffisante pour relier les deux extrémités de l'espace dans lequel évolue la grande nuée de droite. En revanche, dans un réseau organisé en sous-réseaux de petite taille, comme à la Fig. 7.1(b), la petite nuée peut évoluer sur l'intégralité du réseau à condition qu'il y ait une interconnexion entre ceux-ci.

Contrairement aux chapitres précédents, ce chapitre est un chapitre de « travaux inachevés » et de perspectives qui n'ont pas été publiés ; il n'est donc pas à prendre comme un travail totalement abouti. L'idée majeure que nous explorons dans la suite consiste à hiérarchiser les réseaux (cf. Sec. 1.6.5). C'est-à-dire que, plutôt que de considérer un réseau logique comme un tout, nous le considérons comme une interconnexion de petits réseaux à l'intérieur desquels l'explosion du nombre de fragments peut être contenue. Cette idée vient du fait que, si l'on fait un parallèle avec un espace euclidien, l'organisation en réseau aléatoire revient à donner la possibilité à un pair situé à un endroit du réseau d'être voisin avec un pair situé à l'opposé. Ainsi, schématiquement, une configuration dans laquelle on ne tolérerait aucune rupture de cohésion et dont deux agents d'une même nuée pourraient être situés aux extrémités opposées, doit pouvoir s'étirer sur tout l'espace (i.e., trouver un chemin entre ces deux pairs). Cette idée est illustrée de manière très schématique à la Fig. 7.1(a). Le diamètre du réseau pourrait donner le nombre minimum d'agents requis, à condition qu'une nuée ait l'élasticité suffisante pour se configurer sous la forme d'une chaîne sans rompre sa cohésion. En pratique, une nuée se configure rarement en chaîne pour des questions de robustesse et possède une élasticité/résistance qui lui est propre (le nombre de ses fragments se retrouve donc supérieur au diamètre du réseau). Pour limiter l'explosion du nombre d'agents en fonction du nombre de pairs, il semble intéressant de rétablir la hiérarchie dans l'organisation de l'espace qui a été perdue lors du passage à la topologie logique. Cette hiérarchie consiste à découper l'espace en plusieurs sous-espaces, chacun possédant les propriétés structurelles limitant l'explosion du nombre d'agents, et à définir les liens qui permettent aux nuées de passer d'un espace à un autre. La Fig. 7.1(b) illustre de manière schématique le fait que, à condition que les liaisons entre les espaces soient correctes, la nuée n'a pas besoin d'augmenter sa taille, étant donné qu'elle s'exécute successivement dans

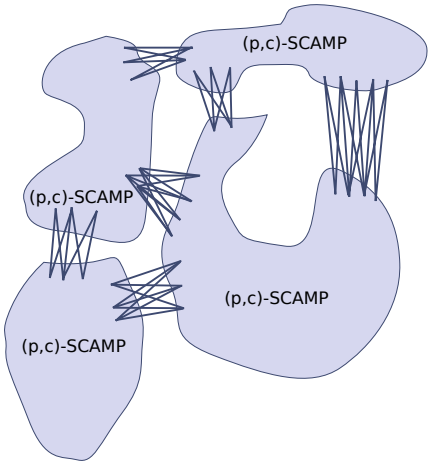


FIGURE 7.2 – Deux niveaux d'overlay. Sous-espaces (p,c)-SCAMP et définition de zones de passage entre ces sous-espaces.

des sous-espaces de mêmes propriétés. Cette approche, lorsqu'elle est transposée aux réseaux logiques, prend une forme de réseau hiérarchique à la manière de l'interconnexion des réseaux SCAMP présentés à la Fig. 7.2. Toute la difficulté réside dans l'interconnexion de ces sous-réseaux. Cette interconnexion doit être tolérante aux fautes et permettre le passage de toute nuée d'un sous-réseau à un autre, sans dégrader les propriétés de la nuée ou engendrer une croissance de ses fragments.

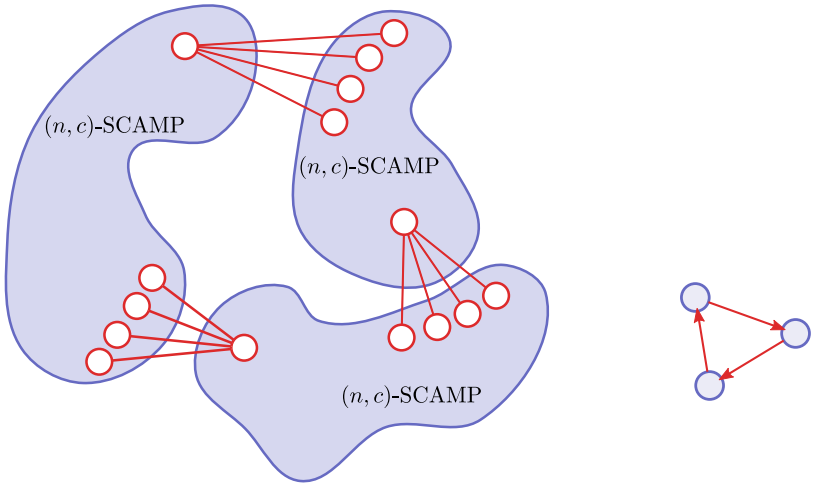
Nous présentons, dans la suite de ce chapitre, les interconnexions que nous avons explorées. Elles sont au nombre de quatre et les deux premières se sont révélées inefficaces :

1. interconnexion par *surjection* ;
2. interconnexion par *produit cartésien* ;
3. interconnexion par *inter-réseaux fortement connectés et par suppression de sous-nuées non-viables* ;
4. interconnexion par *passerelles*.

7.2 Interconnexion de sous-réseaux

Nous proposons donc, dans cette section, les résultats que nous avons obtenus sur ces différents types d'interconnexions. Ces réseaux hiérarchiques sont tous construits sur le même principe ; c'est ensuite la manière de les interconnecter qui diffère. Soit m, le nombre de sous-réseaux SCAMP désirés, n, le nombre de pairs par sous-réseau et c, la constante de tolérance aux fautes de chacun de ces sous-réseaux. Le réseau hiérarchique souhaité se construit alors comme suit :

1. construire les m (n,c)-SCAMP ;
2. construire le «meta» graphe d'interconnexion $\mathcal{G}$ (i.e., le modèle d'interconnexion des sous-réseaux). Ce graphe est de type Erdös-Rényi, paramétré avec un nombre de nœuds égal à



(a) Réseau hiérarchique.

(b) Graphe d'interconnection $\mathcal{G}$ correspondant.

FIGURE 7.3 – Interconnexion surjective : pour chaque arc de $\mathcal{G}$ reliant deux sous-réseaux i et j, connecter un pair de i avec un certain nombre de pairs de j.

m et un degré moyen convergeant vers $(\Phi + 1)\log(m)$. Avec Φ une constante de tolérance aux fautes permettant de moduler la tolérance aux fautes du niveau d'interconnexion [24] ;

3. ensuite, appliquer pour chaque arc de $\mathcal{G}$ le schéma d'interconnexion souhaité.

7.2.1 Interconnexion surjective

Le premier type d'interconnexion que nous considérons est illustré sur le schéma de la Fig. 7.3. Il consiste à connecter, pour chaque arc (i,j) du graphe d'interconnexion $\mathcal{G}$, un pair du sous-réseau i avec un certain nombre de pairs λ du sous-réseau j. Ce nombre est appelé taille de l'interconnexion ($\lambda = 4$ dans l'exemple). Dans notre implémentation, ces pairs sont choisis aléatoirement et il n'est pas permis qu'un pair puisse interconnecter plus de deux sous-réseaux à la fois. Nous avons ensuite mené un certain nombre de simulations pour évaluer le gain à utiliser cette structure logique. Ces simulations consistent à laisser évoluer une nuée de 15 fragments configurée avec un seuil de réparation fixé à 11 dans un réseau hiérarchique de 4000 pairs, constitué de 20 sous-réseaux de type (200,20)-SCAMP qui sont interconnectés par la méthode surjective (on note un tel réseau (20×200,20)-S-SCAMP). Cette nuée va rechercher, à l'aide de réparations successives, un état d'équilibre dans lequel les ruptures de cohésion sont de faible amplitude. Nous suivons cette croissance en mesurant le nombre de fragments ainsi que la cohésion de cette nuée au cours des simulations. Les résultats de ces expériences sont présentées dans le Tab. 7.1. Dans ce tableau, chaque ligne présente les résultats pour une taille d'interconnexion donnée. La dernière ligne affiche les résultats que nous avions obtenus à la Sec. 5.3 sur la même instance de réseau, mais qui était non-hiérarchique. Nous constatons que les nuées

24. Nous avons fixé arbitrairement $\Phi = 3$ dans toutes les expériences présentées dans la suite de ce chapitre.

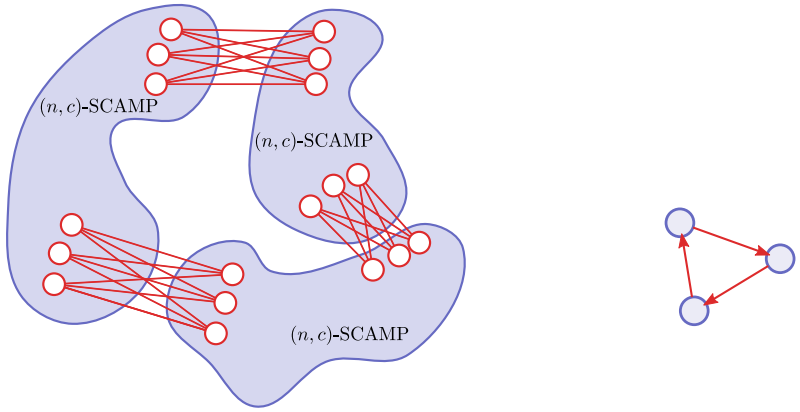
Taille de l'interconnexion (λ)	Cohésion	Nombre de fragments	Nombre de pairs visités
10	17(0.4)	17(0.5)	200(3.1)
20	17(0.1)	17(0.2)	197(5.6)
30	17(0.8)	17(0.8)	197(5.1)
40	18(1.5)	22(6.3)	303(146.3)
50	17(0.4)	17(0.4)	199(3.4)
70	18(0.8)	27(13.6)	374(255.6)
Pas d'évaporation des phéromones			
10	17(0.0)	17(0.0)	202(0.2)
20	17(0.3)	17(0.3)	201(0.1)
30	17(0.3)	17(0.3)	200(0.1)
40	17(0.1)	18(1.5)	213(12.8)
50	16(0.8)	17(0.0)	121(79.8)
70	17(0.6)	18(0.6)	199(0.1)
Val. Ref. non-hiérarchique	**39 (2.5)**	**46.3 (2.7)**	**4000(0)**

TABLE 7.1 – Mesures obtenues pour des nuées évoluant dans un $(20 \times 200{,}20)$-S-SCAMP hiérarchique et interconnecté par surjection. Les valeurs entre parenthèses sont les valeurs d'écart-type. La dernière ligne du tableau donne, pour rappel, les valeurs obtenues pour une instance de mêmes paramètres (4000,20)-SCAMP non-hiérarchique (cf. Sec. 5.3.3).

n'arrivent pas à sortir de leur sous-réseau d'origine pour toutes les valeurs de λ testées. Nous avons d'abord pensé que l'évaporation des phéromones était peut-être trop rapide et nous avons décidé de refaire ces expériences sans évaporation ; les résultats sont identiques. Nous concluons donc que cette instance de réseau hiérarchique est totalement inefficace et ne solutionne pas le problème de l'explosion du nombre de fragments. En revanche, elle met en évidence le fait qu'il existe des topologies réseau qui ne permettent pas à une nuée de couvrir l'intégralité de son environnement.

7.2.2 Interconnexion par produit cartésien

Le second type d'interconnexion que nous considérons est illustré sur le schéma de la Fig. 7.4. Il consiste à choisir, pour tout arc (i,j) du graphe d'interconnexion $\mathcal{G}$, un nombre de pairs λ de i et à les connecter à λ pairs de j ($\lambda = 3$ dans l'exemple). Cette idée d'augmenter le maillage vient comme une tentative pour résoudre le problème de couverture réseau observée dans l'interconnexion précédente. Nous effectuons le même type d'expérience que précédemment à part que l'instance de réseau étudiée change. Nous nous plaçons maintenant dans un réseau hiérarchique de 4000 pairs, constitué de 4 sous-réseaux (1000,20)-SCAMP qui sont interconnectés par produit cartésien (on note un tel réseau $(4 \times 1000{,}20)$-C-SCAMP). Les résultats de ces simulations sont affichés dans le Tab. 7.2. La première chose que nous constatons est que ce



(a) Réseau hiérarchique.

(b) Graphe d'interconnection $\mathcal{G}$ correspondant.

FIGURE 7.4 – Interconnexion par produit cartésien : pour chaque arc de $\mathcal{G}$ reliant deux sous-réseaux i et j, choisir un certain nombre de pairs dans i et les interconnecter avec autant de pairs de j.

Taille de l'interconnexion (λ)	Cohésion	Nombre de fragments	Nombre de pairs visités
5	26 (1.4)	41 (7.0)	3445 (994.0)
7	26 (1.5)	44 (8.0)	3437 (1046.0)
10	26 (1.3)	47 (9.8)	3703 (465.0)
20	28 (2.8)	62 (3.7)	3953 (93.0)
30	29 (1.7)	63 (6.2)	3863 (192.0)
40	33 (2.0	64 (1.3)	3928 (100.0)
Val. Ref. non-hiérarchique	**39 (2.5)**	**46.3 (2.7)**	**4000(0)**

TABLE 7.2 – Mesures obtenues pour des nuées évoluant dans un $(4 \times 1000{,}20)$-SCAMP Hiérarchique et interconnecté par produit cartésien. Les valeurs entre parenthèses sont les valeurs d'écart-type. La dernière ligne du tableau donne, pour rappel, les valeurs obtenues pour une instance de mêmes paramètres (4000,20)-SCAMP non-hiérarchique (cf. Sec. 5.3.3).

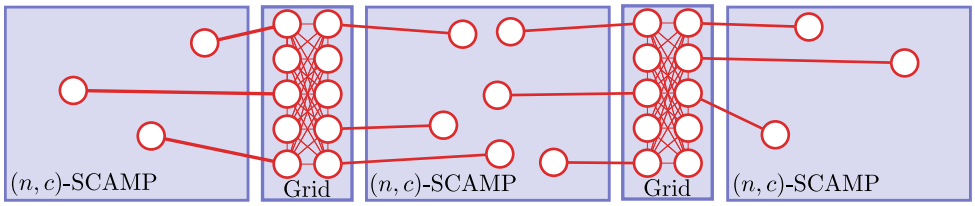


FIGURE 7.5 – Inter-réseaux fortement connectés : chaîner les sous-réseaux en les interconnectant à l'aide de graphes complets ($\lambda = 4$).

nouveau type d'interconnexion permet à une nuée de parcourir l'intégralité du réseau. Ensuite, pour $\lambda < 10$, le nombre de fragments obtenus avec ce réseau hiérarchique est similaire à celui que nous avions obtenu dans le réseau non-hierarchique correspondant. Par contre, lorsque $\lambda > 10$, ce nombre de fragments devient supérieur à celui obtenu dans le réseau non-hiérarchique. Nous en concluons que, bien que fonctionnelle, cette approche n'apporte aucun gain. En effet, elle ne réduit pas l'explosion du nombre de fragments et dégrade légèrement les performances du flocking, comme en témoignent les valeurs de cohésion plus faibles que dans une approche non-hiérarchique.

7.2.3 Inter-réseaux fortement connectés et suppression de sous-nuées non-viables

Les deux stratégies précédentes sont probablement non-viables car elles ne créent pas assez de ruptures entre les sous-réseaux. C'est de ce postulat que naît l'idée que nous présentons à présent. La nouvelle interconnexion consiste à relier les sous-réseaux à l'aide de graphes complets. Ce principe, illustré sur le schéma de la Fig. 7.5, crée, pour chaque arc (i,j) de $\mathcal{G}$, un graphe complet GRID de η pairs qui sert d'interconnexion entre i et j. L'interconnexion entre i et GRID est réalisée de manière aléatoire en connectant λ pairs de i à λ pairs de GRID (dans l'exemple, $\lambda = 3$ et $\eta = 10$). Chaque GRID est alors un espace transitoire dans lequel les nuées passent systématiquement avant de changer de sous-réseau. Dans des évaluations préliminaires, nous nous sommes rendus compte qu'avec cette topologie réseau, lorsqu'une nuée traverse un GRID pour rejoindre un autre sous-réseau, elle a tendance à laisser dans le réseau d'origine quelques fragments qui forment une ou plusieurs sous-nuées non-viables. Pour éviter cette pollution, nous avons modifié le processus de supervision et de réparation pour détecter et supprimer ces sous-nuées non-viables et empêcher, ainsi, qu'une nuée ne laisse trop de fragments derrière elle. Ce mécanisme de *garbage collector* fonctionne en mettant en sursis, pour un certain temps, toute nuée non-viable (i.e., qui ne contient pas assez de fragments pour se réparer). L'évaluation de cette approche est réalisée de la même manière que précédemment, sur plusieurs types de réseaux avec $\eta = 50$. Ces résultats sont présentés dans le Tab. 7.3 et les colonnes intitulées « Ref » affichent les valeurs que nous avions obtenues pour les mêmes instances de réseaux non-hiérarchiques. Nous constatons, d'une part, qu'avec ce type d'interconnexions les nuées

Instance	λ	Cohésion		Nombre de fragments		Nombre de pairs visités
		GRID	Ref	GRID	Ref	
(5x200,20)	5	17(1.1)	26.5(1.1)	25(3.0)	27.5(1.0)	526
(5x200,20)	10	19(0.8)	26.5(1.1)	29(2.3)	27.5(1.0)	1200
(5x200,20)	20	20(0.2)	26.5(1.1)	32(2.8)	27.5(1.0)	1200
(5x200,20)	30	21(0.5)	26.5(1.1)	28(3.2)	27.5(1.0)	1200
(5x200,20)	45	24(0.8)	26.5(1.1)	39(7.6)	27.5(1.0)	1200
(20x200,c20)	45	23(0.4)	39(2.5)	**40.9(3.5)**	**46.3(2.7)**	4950
(50x200,c20)	45	23(0.4)	43.8(5.2)	**36(0.8)**	**59.3(6.7)**	12450

TABLE 7.3 – Mesures obtenues pour des nuées évoluant dans différentes instances de réseaux interconnectés par des graphes complets. Les valeurs entre parenthèses sont les valeurs d'écart-type. La colonne Ref donne, pour rappel, les valeurs obtenues pour une instance non-hiérarchique de mêmes paramètres (cf. Sec. 5.3.3).

couvrent totalement le réseau[25], à l'exception du réseau (5x200,20,$\lambda = 5$). D'autre part, pour les grandes instances réseau (4000 et 10000 pairs), le gain en nombre de fragments générés est significatif et compris entre 12% et 39%. Ces deux observations valident donc le fait que cette approche réduit l'explosion du nombre de fragments. De plus, ces résultats donnent un éclairage sur l'impact que peut avoir la taille de l'interconnexion sur le comportement des nuées. En effet, on constate que pour une taille de réseau fixée, l'augmentation de l'interconnexion réduit le gain de l'approche (i.e., plus l'interconnexion est grande et plus la nuée génère des fragments), mais qu'il est quand même nécessaire d'avoir une interconnexion suffisante pour pourvoir parcourir l'intégralité du réseau. Par exemple, l'interconnexion de taille 5 dans le réseau (5x200,20) est insuffisante pour permettre à la nuée de parcourir tout son environnement. Il y a donc ici, une valeur d'interconnexion minimale à trouver pour que l'approche soit efficace.

7.2.4 Utilisation de pairs passerelles

L'approche précédente, bien que donnant des résultats intéressants, introduit une structure logique beaucoup plus rigide et difficile à maintenir lorsque le réseau subit des fautes. La dernière idée que nous proposons, et sans doute la plus simple à mettre en place, consiste à utiliser des pairs passerelles entre les sous-réseaux. Le rôle de ces passerelles est de capter l'intégralité des agents d'une nuée et de les envoyer un par un vers l'autre extrémité du lien. La principale faiblesse de ce type de fonctionnement est qu'il est centralisé. En effet, lors de son transit, une nuée est entièrement dépendante de la passerelle qu'elle emprunte. Cependant, il est possible de rendre les nuées insensibles aux fautes que peuvent subir les passerelles. La procédure est relativement proche de la supervision. Lorsque le superviseur d'une nuée décide d'emprunter

25. Si le nombre de pairs visités se trouve être supérieur à la taille théorique du réseau, c'est simplement que le nombre de pairs des GRID n'a pas été inclu dans la notation. Par exemple, dans l'instance de réseau (5×200,20), le réseau est constitué de 1000 pairs et de 4 GRID de 50 pairs ; d'où, un nombre de pairs visités égal à 1200.

une passerelle, il clone chacun des fragments de la nuée et les envoie à l'autre extrémité de la passerelle. Lorsque tous les fragments ont été transférés, l'ancienne nuée peut être détruite. Par contre, si une faute survient sur le pair de destination et que les fragments déjà transférés sont perdus, la nuée d'origine reprend ses déplacements jusqu'à trouver une autre passerelle pour transiter.

7.3 Conclusion

Nous avons proposé de hiérarchiser les réseaux pour réduire l'explosion du nombre de fragments d'une nuée. Parmi les stratégies que nous avons testées, la stratégie d'interconnexion par réseaux fortement connectés a donné des résultats intéressants en terme de gain par rapport à une approche non-hiérarchique. Ces résultats montrent que les nuées ont la possibilité de passer à l'échelle si nécessaire. Néanmoins, cette introduction de structure dans le réseau logique rend plus difficile sa construction et son maintien en présence de fautes. D'autant plus que le type d'interconnexion ayant donné les seuls résultats viables nécessite un algorithme pour la construction et le maintien des graphes complets (GRID) mais également le maintien de la taille de l'interconnexion (λ) à la valeur désirée. Ce maintien devant être réalisé de manière décentralisée, les algorithmes résultants sont loin d'être triviaux. Rappelons cependant que ce travail est inachevé et qu'il existe, sans doute, d'autres types d'interconnexions plus simples à mettre en place et donnant de bons résultats. L'idée de la hiérarchisation restant, selon nous, une idée intéressante à explorer. Dans l'immédiat, nous avons évoqué la perspective de mettre en place des passerelles entre les sous-réseaux qui consisteraient à utiliser un mécanisme d'élection de leader similaire à la supervision pour gérer les transferts. Cette approche, bien que plus simple à déployer, a le désavantage de recourir à nouveau à une entité centrale, plus vulnérable aux fautes.

Conclusion et perspectives

Bilan

Les travaux présentés dans ce mémoire de thèse portent sur le stockage de données décentralisé dans les architectures pair-à-pair. L'approche que nous avons présentés transforme les documents en nuées d'agents mobiles autonomes qui ont la possibilité de se déplacer dans le réseau. Cette mobilité et cette autonomie offrent aux nuées une certaine adaptabilité vis-à-vis de leur environnement, que nous avons eu l'occasion d'étudier à travers plusieurs expériences. Rappelons à présent les éléments majeurs de ce travail.

Du flocking dans un réseau

Dans le Chap. 4, nous nous sommes intéressés aux différents algorithmes permettant de mettre en nuées un ensemble d'agents mobiles. Ces algorithmes asynchrones et décentralisés ont été adaptés dans un réseau à partir des règles de Reynolds. Nous avons, tout d'abord, mis en évidence le fait que l'algorithme de flocking peut se passer de la notion de distance pour être adapté dans un réseau. Seules les relations topologiques existant entre les pairs ont de l'importance. Les premières simulations que nous avons effectuées sur ce modèle ont montré que les agents mobiles forment bien des nuées connexes. Pour valider notre simulateur, nous avons développé en parallèle un prototype de l'application que nous avons déployé dans un réseau physique réel d'une centaine de pairs. En reproduisant les mêmes expériences sur ce prototype et en obtenant les mêmes résultats, nous avons été en mesure de valider notre simulateur pour la suite de nos travaux. Ces expériences ont notamment mis en évidence le fait que les nuées peuvent subir des ruptures de cohésion lors de leurs déplacements.

Recherche des paramètres de fragmentation optimaux : auto-adaptation des nuées

Dans le Chap. 5, nous avons mis en place des politiques de supervision et de réparation décentralisées pour assurer la disponibilité et la durabilité des nuées. Nous nous sommes rendus compte que les fluctuations de la cohésion d'une nuée peuvent perturber dangereusement ces politiques. En fait, lorsque l'amplitude des ruptures de cohésion est trop forte par rapport au seuil de réparation fixé, les nuées lancent des réparation inutiles et coûteuses. Nous nous sommes rendus compte que l'amplitude de ces ruptures est dépendante de la taille de la nuée ainsi que

des propriétés du réseau dans lequel elle évolue. Nous avons également constaté que, dans un réseau aléatoire de type SCAMP, pour une taille de nuée fixée, plus la taille du réseau augmente et plus les ruptures de cohésion sont de forte amplitude. Face à ce constat, nous avons cherché un moyen de réduire l'amplitude de ces ruptures en cherchant la taille adéquate d'une nuée en fonction du réseau dans lequel elle évolue. Il s'avère que par un mécanisme simple de réparations successives, les nuées arrivent à trouver par auto-adaptation leur taille idéale. Grâce à cette adaptation, une nuée est non seulement capable de trouver ses paramètres de fragmentation à l'issue d'une phase d'initialisation mais également de pouvoir réagir si la structure du réseau vient à évoluer dans le temps.

Placement adaptatif et application aux fautes corrélées

Dans le Chap. 6, nous présentons les résultats de l'algorithme MinCor. Nous montrons que si un clustering de pairs corrélés peut être calculé, alors, il est possible de répartir les nuées sur ces clusters de sorte à minimiser les corrélations. Le contrôle de ce placement est réalisé à l'aide de la mécanique du recuit-simulé qui cherche à minimiser certains critères. Cette dynamique préserve le déplacement en nuée étant donné que le choix des déplacements est effectué sur un filtrage des candidats renvoyés par l'algorithme de flocking. À la suite d'un ensemble d'expériences, nous montrons qu'en présence de fautes corrélées, MinCor apporte un gain significatif en terme de disponibilité par rapport aux autres approches. Ces expériences ont été l'occasion de mettre en évidence le fait que la mobilité, à elle seule, apporte un gain en robustesse par rapport à un placement immobile.

Passage à l'échelle de l'architecture

Dans le dernier chapitre, nous avons abordé le problème du passage à l'échelle de l'architecture. Ce problème vient du fait que, dans SCAMP, pour conserver des ruptures de cohésion de faible amplitude, les nuées doivent croître conjointement avec le réseau. Or, les plateformes agents mobiles ayant une capacité bornée, la taille des nuées doit être réduite au minimum. Notre proposition, pour réduire cette explosion, consiste à hiérarchiser les réseaux. Cette hiérarchisation est obtenue en interconnectant plusieurs sous-réseaux de petite taille pour que les nuées gardent une taille constante et raisonnable. Après plusieurs tentatives, nous avons été en mesure de proposer un réseau hiérarchique apportant un gain significatif. Ce réseau est construit en interconnectant les sous-réseaux à l'aide de graphes complets et en nettoyant périodiquement les sous-nuées non-viables qui sont générées. Bien qu'apportant un gain évident, cette structure réseau rigide reste relativement fastidieuse à construire et à maintenir. C'est pourquoi nous avons suggéré une autre approche consistant à utiliser des passerelles centralisées pour l'interconnexion. Ces travaux ont un caractère inachevé mais il nous semble quand même important de souligner qu'ils ont une potentialité certaine.

Perspectives

Le prolongement de ces travaux peut être envisagé dans plusieurs directions. Premièrement, nous avons exhibé, à l'aide de nos simulations, que l'utilisation des relations topologiques entre les agents dans les prises de décisions est suffisante pour faire émerger le flocking dans un réseau. Ce constat nous a amenés à considérer la transposition de ce modèle dans le vivant, en nous interrogeant sur les véritables liens d'adjacences entre les étourneaux en nuées. Il s'avère que [Ballerini *et al.*, 2008] ont identifié des propriétés similaires aux notres chez *Sturnus vulgaris*. Un prolongement de ce *crossover* entre l'informatique et la biologie pourrait consister à modéliser des nuées d'oiseaux avec ces nouvelles relations, puis observer si cette modélisation est meilleure que les précédentes basées sur des relations métriques.

Il nous semble également important d'étudier plus en détail les questions de passage à l'échelle des nuées. Tester de nouvelles interconnexions et proposer des algorithmes pour construire et maintenir ces réseaux hiérarchiques pourrait être un bon point de départ. L'utilisation des passerelles centralisées n'est qu'une solution temporaire, qui n'est pas réellement satisfaisante dans ce type d'environnement.

Sur cette même thématique du passage à l'échelle, nous avons eu l'occasion d'évoquer le fait que les plateformes agents mobiles actuelles fonctionnent sur un modèle qui encapsule chaque agent dans un fil d'exécution. Ce choix de conception est inadapté si chaque pair doit exécuter plusieurs milliers d'agents en même temps (ce qui est le cas dans notre application de stockage). Parmi les pistes que nous avons retenues pour solutionner ce problème, nous avons proposé l'idée d'une plateforme ordonnancée dans laquelle les agents peuvent se trouver dans l'état actif ou inactif. Un agent actif est un agent ayant des messages en attente à traiter ou bien devant prendre une décision. Les agents inactifs sont, quand à eux, des agents qui ne sont pas dans la boucle de l'ordonnanceur et qui sont, de temps en temps, réveillés pour savoir s'ils doivent effectuer des traitements. Ce réveil doit avoir une fenêtre temporelle très faible. Dans notre cas, un agent de flocking serait inactif la plupart du temps et ne deviendrait actif que s'il doit décider de se déplacer.

L'aspect sécurité n'a pas été abordé dans cette thèse et nous avons préféré nous attarder sur les mécanismes permettant d'assurer la disponibilité des nuées. Néanmoins, la sécurité dans ce type d'application est également primordiale. Elle peut être abordée sur deux plans : la sécurisation de l'application elle-même et l'apport de la mobilité des nuées à la sécurité du système.

1. Le premier plan consiste à mettre en place des mécanismes d'authentification pour s'assurer : que seuls les agents d'une même nuée puissent interagir entre-eux, que l'authenticité du code de chaque agent soit vérifiée avant son exécution et qu'aucun agent forgé ne puisse être introduit dans le système. Ces mécanismes d'authentification pourraient être suffisants pour sécuriser le SMA à condition que les plateformes soient intègres. Or, cette hypothèse n'est pas réaliste dans un cadre pair-à-pair. Il faudra donc ajouter des mécanismes pour s'assurer que les plateformes n'altèrent pas le code des agents mais également

que ces dernières puissent détecter, à l'aide de mécanismes de confiance par exemple, les comportements anormaux de leurs plateformes voisines.

2. Le second plan consiste à évaluer le gain de la mobilité des données sur la sécurité. Par exemple, le fait qu'une nuée est toujours en mouvement rend il plus délicat la mise en place d'attaques, en comparaison avec une architecture statique ?

Finalement, un autre prolongement de ce travail pourrait consister à évaluer le coût de l'approche en terme de ressources consommées. Mais également, d'évaluer comment la variation des différents paramètres du système tels que : la taille des nuées, la réactivité des prises de décision ou encore le schéma de fragmentation, conditionnent ces performances.

Bibliographie

[SHA, 1995] (1995). *Secure hash standard*. National Institute of Standards and Technology, Washington. Note : Federal Information Processing Standard Publication 180-1. 12

[FIP, 1997] (1997). *FIPA 97 Part 2 Version 2.0 : Agent Communication Language Specification*. Foundation for Intelligent Physical Agents. 86

[MAS, 1997] (1997). *Mobile Agent System Interoperability Facilities. TC document orbos/97-10-05*. Object Management Group. 93

[AES, 2001] (2001). *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology. Note : Federal Information Processing Standard Publication 197. 69

[Gnu, 2003] (2003). *The Annotated Gnutella Protocol Specification v0.4*. The Gnutella Developer Forum. Annotated Standard (Revision 1.6). 16

[Adya *et al.*, 2002] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M. et Wattenhofer, R. P. (2002). Farsite : federated, available, and reliable storage for an incompletely trusted environment. *In Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 1–14, New York, NY, USA. ACM. 49, 60, 62, 66

[Agha, 1986] Agha, G. (1986). *Actors : a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA. 82, 91, 97

[Aiello *et al.*, 2011] Aiello, F., Fortino, G., Gravina, R. et Guerrieri, A. (2011). A Java-Based Agent Platform for Programming Wireless Sensor Networks. *The Computer Journal*, 54(3):439–454. 91, 96

[Ametller *et al.*, 2004] Ametller, J., Robles, S. et Ortega-Ruiz, J. A. (2004). Self-Protected Mobile Agents. *In Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, pages 362–367. 94

[Androutsellis-Theotokis et Spinellis, 2004] Androutsellis-Theotokis, S. et Spinellis, D. (2004). A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4):335–371. 9

[Arcangeli *et al.*, 2001] Arcangeli, J., Maurel, C. et Migeon, F. (2001). An API for High-Level Software Engineering of Distributed and Mobil Applications. *In Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, FTDCS '01, pages 155–. 91, 92, 95, 96

[Arcangeli *et al.*, 2004a] Arcangeli, J.-P., Hennebert, V., Sébastien, L., Migeon, F. et Pantel, M. (2004a). JavAct 0.5.0 : principes, installation, utilisation et développement d'applications. Rapport technique IRIT/2004-5-R, IRIT. 95

[Arcangeli *et al.*, 2004b] Arcangeli, J.-P., Leriche, S. et Pantel, M. (2004b). Development of Flexible Peer-To-Peer Information Systems Using Adaptable Mobile Agents. *In Proceedings of the Database and Expert Systems Applications, 15th International Workshop*, DEXA '04, pages 549–553. 95, 96, 99

[Bakkaloglu *et al.*, 2002] Bakkaloglu, M., Wylie, J. J., Wang, C. et Ganger, G. R. (2002). On Correlated Failures in Survivable Storage Systems. Rapport Technique CMU-CS-02-129, Carnegie Mellon University. 74

[Ballerini *et al.*, 2008] Ballerini, M., Cabibbo, N., Candelier, R., Cavagna, A., Cisbani, E., Giardina, I., Lecomte, V., Orlandi, A., Parisi, G., Procaccini, A., Viale, M. et Zdravkovic, V. (2008). Interaction ruling animal collective behavior depends on topological rather than metric distance : Evidence from a field study. *Proceedings of the National Academy of Sciences*, 105(4):1232. 133, 185

[Baumann *et al.*, 1998] Baumann, J., Hohl, F., Rothermel, K. et Strasser, M. (1998). Mole – Concepts of a mobile agent system. *World Wide Web*, 1(3):123–137. 96

[Bäumer et Magedanz, 1999] Bäumer, C. et Magedanz, T. (1999). Grasshopper - a mobile agent platform for active telecommunication. *In Proceedings of the Third International Workshop on Intelligent Agents for Telecommunication Applications*, IATA '99, pages 19–32. 96

[Bellifemine *et al.*, 1999] Bellifemine, F., Poggi, A. et Rimassa, G. (1999). JADE — A FIPA- compliant agent framework. *In Proceedings of the 4th International Conference on the Practical Applications of Agents and Multi-Agent Systems*, PAAM-99, pages 97–108. 96

[Bellman, 1978] Bellman, R. (1978). *An introduction to artificial intelligence : can computers think ?* Boyd & Fraser Pub. Co. 78

[Bertsimas et Tsitsiklis, 1993] Bertsimas, D. et Tsitsiklis, J. (1993). Simulated annealing. *Statistical Science*, 8(1):10–15. 157, 159

[Beynier, 2006] Beynier, A. (2006). *Une contribution la résolution des Processus Décisionnels de Markov Décentralisés avec contraintes temporelles.* Thèse de doctorat, Université de Caen Basse-Normandie. 80

[Bhagwan, 2004] Bhagwan, R. (2004). *Automated Availability Management in Large-scale Storage Systems.* Thèse de doctorat, University Of California, San Diego. 70, 71

[Bhagwan *et al.*, 2002] Bhagwan, R., Savage, S. et Voelker, G. M. (2002). Replication Strategies for Highly Available Peer-to-Peer Storage Systems. Rapport Technique CS2002-0726, UCSD. 71

[Bhagwan *et al.*, 2003] Bhagwan, R., Savage, S. et Voelker, G. M. (2003). Understanding availability. *In IPTPS*, pages 256–267. 73

[Bhagwan *et al.*, 2004] Bhagwan, R., Tati, K., Cheng, Y.-C., Savage, S. et Voelker, G. M. (2004). Total Recall : System Support for Automated Availability Management. *In NSDI*, pages 337–350. 61, 62, 67

[Bianchi *et al.*, 2002] Bianchi, L., Gambardella, L. et Dorigo, M. (2002). An Ant Colony Optimization Approach to the Probabilistic Traveling Salesman Problem. *In Proceedings of the Seventh International Conference on Parallel Problem Solving from Nature (PPSN VII)*, Lecture Notes in Computer Science. Springer Verlag. 83

[Bilchev et Parmee, 1996] Bilchev, G. et Parmee, I. (1996). Evolutionary Metaphors for the Bin Packing Problem. *In Fifth Annual Conference on Evolutionary Programming*. 83

[Birrell et Nelson, 1984] Birrell, A. D. et Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59. 88

[BitTorrent, 2012] BitTorrent (2012). `http://www.bittorrent.com`. 12

[Blaess, 2011] Blaess, C. (2011). *Programmation système en C sous Linux : Signaux, processus, threads, IPC et sockets*. Eyrolles. 88

[Blake et Rodrigues, 2003] Blake, C. et Rodrigues, R. (2003). High availability, scalable storage, dynamic peer networks : Pick two. *In HotOS*, pages 1–6. 50

[Boccara, 2010] Boccara, N. (2010). *Modeling Complex Systems*. Graduate Texts in Contemporary Physics. Springer. 86

[Bonabeau *et al.*, 1999] Bonabeau, E., Dorigo, M. et Theraulaz, G. (1999). *Swarm Intelligence : From Natural to Artificial Systems*. Santa Fe Institute Studies in the Sciences of Complexity. Oxford University Press. 83

[Bonnet, 2008] Bonnet, G. (2008). *Coopération au sein d'une constellation de satellites*. Thèse de doctorat, Université de Toulouse. 85

[Bonnet et Tessier, 2008] Bonnet, G. et Tessier, C. (2008). An incremental adaptive organization for a satellite constellation. *In AAMAS-OAMAS*, pages 108–125. 85

[Brin et Page, 1998] Brin, S. et Page, L. (1998). The anatomy of a large-scale hypertextual Web search engine. *In Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 107–117. 101

[Briot et Demazeau, 2001] Briot, J. et Demazeau, Y. (2001). *Principes et architecture des systèmes multi-agents*. IC2 : Série Informatique et systèmes d'information. Hermes Science Publications. 80, 82

[Buford *et al.*, 2008] Buford, J., Yu, H. et Lua, E. K. (2008). *P2P Networking and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. 9

[Buford et Yu, 2010] Buford, J. F. et Yu, H. (2010). Peer-to-peer networking and applications : Synopsis and research directions. *In* Shen, X., Yu, H., Buford, J. et Akon, M., éditeurs : *Handbook of Peer-to-Peer Networking*, pages 3–45. Springer US. 9

[Canu, 2011] Canu, A. (2011). *Planification multiagent sous incertitude orientée interactions : modèle et algorithmes*. Thèse de doctorat, Université de Caen Basse-Normandie. 80

[Caron *et al.*, 2010] Caron, S., Giroire, F., Mazauric, D., Monteiro, J. et Pérennes, S. (2010). Data life time for different placement policies in p2p storage systems. *In Globe*, pages 75–88. 61

[Castro et Liskov, 1999] Castro, M. et Liskov, B. (1999). Practical byzantine fault tolerance. *In Proceedings of the third symposium on Operating systems design and implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA. USENIX Association. 67

[Charniak et McDermott, 1985] Charniak, E. et McDermott, D. (1985). *Introduction to artificial intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 78

[Charrier, 2009] Charrier, R. (2009). *L'intelligence en essaim sous l'angle des systèmes complexes : étude d'un système multi-agent réactif à base d'itérations logistiques couplées*. Thèse de doctorat, Université de Nancy 2. 86

[Chawathe *et al.*, 2000] Chawathe, Y., McCanne, S. et Brewer, E. A. (2000). Rmx : Reliable multicast for heterogeneous networks. *In IN PROC. IEEE INFOCOM*, pages 795–804. 31

[Chawathe *et al.*, 2003] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N. et Shenker, S. (2003). Making gnutella-like p2p systems scalable. *In Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 407–418. 16

[Chen *et al.*, 2006] Chen, B., Cheng, H. H. et Palen, J. (2006). Mobile-C : a mobile agent platform for mobile C-C++ agents. *Software—Practice & Experience*, 36(15):1711–1733. 91, 96

[Cizault, 2002] Cizault, G. (2002). *IPv6, théorie et pratique*. O'Reilly, 3ème édition. 29

[Clarke *et al.*, 2010] Clarke, I., Sandberg, O., Toseland, M. et Verendel, V. (2010). Private Communication Through a Network of Trusted Connections : The Dark Freenet. `https://freenetproject.org/papers/freenet-0.7.5-paper.pdf`. 19, 26

[Clarke *et al.*, 2001] Clarke, I., Sandberg, O., Wiley, B. et Hong, T. W. (2001). Freenet : a distributed anonymous information storage and retrieval system. *In International workshop on Designing privacy enhancing technologies : design issues in anonymity and unobservability*, pages 46–66. 16, 17

[Cohen, 2008] Cohen, B. (2008). The BitTorrent Protocol Specification. BEP 3 (standard). 11, 12

[Colorni *et al.*, 1991] Colorni, A., Dorigo, M. et Maniezzo, V. (1991). Distributed Optimization by Ant Colonies. *In* Varela, F. et Bourgine, P., éditeurs : *Proceedings of the First European Conference on Artificial Life (ECAL)*, pages 134–142. MIT Press, Cambridge, Massachusetts. 83

[Corbara *et al.*, 1993] Corbara, B., Drogoul, A., Fresneau, D. et Lalande, S. (1993). *Self-organization and life. From the simple rules to global complexity*, chapitre Simulating the sociogenesis process in ant colonies with MANTA, pages 12–24. MIT press. 83

[Costa et Hertz, 1997] Costa, D. et Hertz, A. (1997). Ants Can Colour Graphs. *Journal of the Operational Research Society*, 48:295–305. 83

[Cozien, 2002] Cozien, R. (2002). *Premiers Éléments de la Théorie du Calcul Singulier*. Thèse de doctorat, Université de Reims Champagne-Ardenne. 87

[Cubat Dit Cros, 2005] Cubat Dit Cros, C. (2005). *Agents Mobiles Coopérants pour les Environnements Dynamiques*. Thèse de doctorat, Institut National Polytechnique de Toulouse. 87, 90, 91, 95

[Dabek *et al.*, 2001] Dabek, F., Kaashoek, M. F., Karger, D. R., Morris, R. et Stoica, I. (2001). Wide-Area Cooperative Storage with CFS. *In SOSP*, pages 202–215. 60, 62, 65

[Dalle *et al.*, 2009] Dalle, O., Giroire, F., Monteiro, J. et Perennes, S. (2009). Analysis of Failure Correlation Impact on Peer-to-Peer Storage Systems. *In Peer-to-Peer Computing*, pages 184–193. 61

[Dandoush, 2010] Dandoush, A. (2010). *L'Analyse et l'Optimisation des Systèmes de Stockage de Données dans les Réseaux Pair-à-Pair*. Thèse de doctorat, Université de Nice Sophia-Antipolis. 61

[Deering *et al.*, 1999] Deering, S., Fenner, W. et Haberman, B. (1999). Multicast Listener Discovery (MLD) for IPv6. RFC 2710 (Proposed Standard). Updated by RFCs 3590, 3810. 29

[Deeter *et al.*, 2004] Deeter, K., Singh, K., Wilson, S., Filipozzi, L. et Vuong, S. (2004). APHIDS : A Mobile Agent-Based Programmable Hybrid Intrusion Detection System. *In Mobility Aware Technologies and Applications*, volume 3284 de *Lecture Notes in Computer Science*, pages 244–253. 99, 101

[Demazeau, 1995] Demazeau, Y. (1995). From Interactions to Collective Behaviour in Agent-Based Systems. *In First European Conference on Cognitive Science*, pages 117–132. 79

[Deneubourg *et al.*, 1990] Deneubourg, J. L., Goss, S., Franks, N., Sendova-Franks, A., Detrain, C. et Chrétien, L. (1990). The dynamics of collective sorting robot-like ants and ant-like robots. *In From Animals to Animats*, pages 356–365. 82

[Deswarte *et al.*, 1991] Deswarte, Y., Blain, L. et charles Fabre, J. (1991). Intrusion Tolerance in Distributed Computing Systems. *In Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 110–121. 63

[Dillenseger *et al.*, 2002] Dillenseger, B., Tagant, A.-M. et Hazard, L. (2002). Programming and Executing Telecommunication Service Logic with Moorea Reactive Mobile Agents. *In Proceedings of the 4th International Workshop on Mobile Agents for Telecommunication Applications*, MATA '02, pages 48–57. 96

[Dimakis *et al.*, 2011] Dimakis, A., Ramchandran, K. et Wu, Y. (2011). A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489. 54

[Dimakis *et al.*, 2010] Dimakis, A. G., Godfrey, B., Wu, Y., Wainwright, M. J. et Ramchandran, K. (2010). Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9). 54, 55, 57

[Drogoul, 1993] Drogoul, A. (1993). *De La Simulation Multi-Agent A La Résolution Collective de Problèmes. Une Étude De l'Émergence De Structures D'Organisation Dans Les Systèmes Multi-Agents*. Thèse de doctorat, Université de Paris VI. 82

[Druschel et Rowstron, 2001] Druschel, P. et Rowstron, A. I. T. (2001). Past : A large-scale, persistent peer-to-peer storage utility. *In HotOS*, pages 75–80. 62, 64

[Duminuco, 2009] Duminuco, A. (2009). *Redondance et maintenance des données dans les systèmes de sauvegarde de fichiers pair-à-pair*. Thèse de doctorat, TELECOM ParisTech. 48, 56, 57, 58, 61

[Duminuco et Biersack, 2008] Duminuco, A. et Biersack, E. (2008). Hierarchical codes : How to make erasure codes attractive for peer-to-peer storage systems. *In Proceedings of the 2008 Eighth International Conference on Peer-to-Peer Computing*, P2P '08, pages 89–98, Washington, DC, USA. IEEE Computer Society. 56

[Duminuco et Biersack, 2009] Duminuco, A. et Biersack, E. (2009). A practical study of regenerating codes for peer-to-peer backup systems. *In Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 376–384. 56

[El Falou, 2006] El Falou, S. (2006). *Programmation répartie, optimisation par agent mobile*. Thèse de doctorat, Université de Caen Basse-Normandie. 86, 92

[El Falou et Bourdon, 2005] El Falou, S. et Bourdon, F. (2005). A MDP Solution for Mobile Agents Displacement. *In Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '05, pages 29–33. 92

[Erdős et Rényi, 1959] Erdős, P. et Rényi, A. (1959). On random graphs. I. *Publicationes Mathematicae Debrecen*, 6:290–297. 27

[Erdős et Rényi, 1960] Erdős, P. et Rényi, A. (1960). On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 5:17–61. 27

[Evans *et al.*, 2007] Evans, N. S., GauthierDickey, C. et Grothoff, C. (2007). Routing in the dark : Pitch black. *In ACSAC*, pages 305–314. 26

[Fenner *et al.*, 2006] Fenner, B., Handley, M., Holbrook, H. et Kouvelas, I. (2006). Protocol Independent Multicast - Sparse Mode (PIM-SM) : Protocol Specification (Revised). RFC 4601 (Proposed Standard). Updated by RFC 5059. 29

[Ferber, 1995] Ferber, J. (1995). *Les systèmes multi-agents : Vers une intelligence collective*. I.I.A. Informatique intelligence artificielle. InterEditions. 80, 82, 83, 84, 85

[Floyd *et al.*, 1995] Floyd, S., Jacobson, V., McCanne, S., Liu, C.-G. et Zhang, L. (1995). A Reliable Multicast Framework for Light-Weight Sessions and Application Level Framing. *In SIGCOMM*, pages 342–356. 31

[Franklin et Graesser, 1996] Franklin, S. et Graesser, A. C. (1996). Is it an agent, or just a program ? : A taxonomy for autonomous agents. *In ATAL*, pages 21–35. 80

[Fuggetta *et al.*, 1998] Fuggetta, A., Picco, G. P. et Vigna, G. (1998). Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361. 89

[Gallager *et al.*, 1983] Gallager, R. G., Humblet, P. A. et Spira, P. M. (1983). A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77. 142, 144

[Ganesh *et al.*, 2001] Ganesh, A. J., Kermarrec, A.-M. et Massoulié, L. (2001). SCAMP : Peer-to-Peer Lightweight Membership Service for Large-Scale Group Communication. *In Networked Group Communication*, pages 44–55. 31, 32

[Ganesh *et al.*, 2002] Ganesh, A. J., Kermarrec, A.-M. et Massoulié, L. (2002). HiScamp : self-organizing hierarchical membership protocol. *In ACM SIGOPS European Workshop*, pages 133–139. 31, 39, 40

[Ganesh *et al.*, 2003] Ganesh, A. J., Kermarrec, A.-M. et Massoulié, L. (2003). Peer-to-peer membership management for gossip-based protocols. *IEEE Trans. Computers*, 52(2):139–149. 31, 35, 37

[Gauron, 2006] Gauron, P. (2006). *Interconnexion et routage efficace pour des procédures de recherche décentralisées dans les systèmes pair-à-pair*. Thèse de doctorat, Université de Paris-Sud XI. 15, 16

[Geib *et al.*, 1997] Geib, J., Gransart, C. et Merle, P. (1997). *CORBA : des concepts à la pratique*. I.I.A. Informatique intelligence artificielle. InterÉditions. 88

[GfK Retail and Technology France, 2011] GfK Retail and Technology France (2011). Le marché de la téléphonie mobile selon GfK Focus sur les smartphones. `http://www.gfkrt.com/imperia/md/content/rt-france/cp_gfk_march___des_smartphones_2011_et_perspectives_d_ici_2015.pdf`. xi, 47

[Ghemawat *et al.*, 2003] Ghemawat, S., Gobioff, H. et Leung, S.-T. (2003). The google file system. *In SOSP*, pages 29–43. 61, 62

[Giroire *et al.*, 2009a] Giroire, F., Monteiro, J. et Pérennes, S. (2009a). P2P Storage Systems : How Much Locality Can They Tolerate ? *In LCN*, pages 320–323. 61, 62

[Giroire *et al.*, 2009b] Giroire, F., Monteiro, J. et Pérennes, S. (2009b). P2P Storage Systems : How Much Locality Can They Tolerate ? Rapport de recherche RR-7006, INRIA. 61

[Glad, 2011] Glad, A. (2011). *Etude de l'auto-organisation dans les algorithmes de patrouille multi-agent fondés sur les phéromones digitales*. Thèse de doctorat, Université de Nancy 2. 83

[Godfrey, 2006] Godfrey, B. (2006). Repository of Availability Traces. `http://www.cs.illinois.edu/~pbg/availability/`. 70

[Goldberg, 1989] Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st édition. 82

[Goldschlag *et al.*, 1999] Goldschlag, D., Reed, M. et Syverson, P. (1999). Onion routing for anonymous and private internet connections. *Communications of the ACM*, 42:39–41. 18

[Grassé, 1959] Grassé, P.-P. (1959). La reconstruction du nid et les coordinations interindividuelles chez *Bellicositermes natalensis* et *Cubitermes sp.* la théorie de la stigmergie : Essai d'interprétation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–80. 84

[Gray *et al.*, 2002] Gray, R. S., Cybenko, G., Kotz, D., Peterson, R. A. et Rus, D. (2002). D'agents : applications and performance of a mobile-agent system. *Software—Practice & Experience - Special issue : Mobile agent systems*, 32(6):543–573. 92, 95, 96

[Gray *et al.*, 1998] Gray, R. S., Kotz, D., Cybenko, G. et Rus, D. (1998). D'Agents : Security in a multiple-language, mobile-agent system. *In* Vigna, G., éditeur : *Mobile Agents and Security*, volume 1419 de *Lecture Notes in Computer Science*, pages 154–187. 96

[Grokster, 2005] Grokster (2005). `http://www.grokster.com/`. 14

[Grolimund, 2007] Grolimund, D. (2007). Wuala - A Distributed File System. Oral Talk at Google Tech Talks, 30 Octobre 2007. `http://www.youtube.com/watch?v=3xKZ4KGkQY8`. 69

[Grolimund *et al.*, 2006] Grolimund, D., Meisser, L., Schmid, S. et Wattenhofer, R. (2006). Havelaar : A robust and efficient reputation system for active peer-to-peer systems. *In 1st Workshop on the Economics of Networked Systems (NetEcon)*. 70

[Haeberlen *et al.*, 2005] Haeberlen, A., Mislove, A. et Druschel, P. (2005). Glacier : Highly Durable, Decentralized Storage Despite Massive Correlated Failures. *In NSDI*. 60, 62, 68

[Harrison *et al.*, 1994] Harrison, C., Chess, D. et Kershenbaum, A. (1994). Mobile agents : Are they a good idea ? Rapport technique, IBM Research. 91

[Harrouet, 2000] Harrouet, F. (2000). *oRis : s'immerger par le langage pour le prototypage d'univers virtuels à base d'entités autonomes*. Thèse de doctorat, Université de Bretagne Occidentale. 87, 116

[Harrouet *et al.*, 2002] Harrouet, F., Tisseau, J., Reignier, P. et Chevaillier, P. (2002). oRis : un environnement de simulation interactive multi-agents. *Technique et Science Informatiques*, 21(4):499–524. 87, 116

[Heckmann et Bock, 2002] Heckmann, O. et Bock, A. (2002). The eDonkey 2000 Protocol. Rapport technique, Multimedia Communications Lab, Darmstadt University of Technology. 13

[Heckmann *et al.*, 2004] Heckmann, O., Bock, A., Mauthe, A. et Steinmetz, R. (2004). The eDonkey File-Sharing Network. *In GI Jahrestagung (2)*, pages 224–228. 13

[Hildrum *et al.*, 2002] Hildrum, K., Kubiatowicz, J. D., Rao, S. et Zhao, B. Y. (2002). Distributed object location in a dynamic network. *In Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 41–52. 22

[Holbrook *et al.*, 2006] Holbrook, H., Cain, B. et Haberman, B. (2006). Using Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Protocol Version 2 (MLDv2) for Source-Specific Multicast. RFC 4604 (Proposed Standard). 29

[Horling et Lesser, 2004] Horling, B. et Lesser, V. (2004). A survey of multi-agent organizational paradigms. *The Knowledge Engineering Review*, 19(4):281–316. 85

[Hosseini *et al.*, 2007] Hosseini, M., Ahmed, D. T., Shirmohammadi, S. et Georganas, N. D. (2007). A survey of application-layer multicast protocols. *IEEE Communications Surveys and Tutorials*, 9(1-4):58–74. 31

[Ilarri *et al.*, 2006] Ilarri, S., Trillo, R. et Mena, E. (2006). Scalable PlatfoRm for movINg Software (SPRINGS). `http://sid.cps.unizar.es/SPRINGS/`. 96

[iMesh, 2012] iMesh (2012). `http://www.imesh.com/`. 14

[JADE, 2012] JADE (2012). Java Agent DEvelopment Framework. `http://jade.tilab.com/`. 96

[Jaggi *et al.*, 2005] Jaggi, S., Sanders, P., Chou, P. A., Effros, M., Egner, S., Jain, K. et Tolhuizen, L. M. G. M. (2005). Polynomial time algorithms for multicast network code construction. *IEEE Transactions on Information Theory*, 51(6):1973–1982. 54

[JavAct, 2012] JavAct (2012). JAVACT : un intergiciel Java pour les agents mobiles adaptatifs. `http://www.javact.org/`. 95

[Johansen, 1998] Johansen, D. (1998). Mobile agent applicability. *In Mobile Agents*, pages 80–98. 92

[Johansen *et al.*, 2002] Johansen, D., Lauvset, K. J., van Renesse, R., Schneider, F. B., Sudmann, N. P. et Jacobsen, K. (2002). A TACOMA retrospective. *Software—Practice & Experience*, 32(6):605–619. 96

[Jones, 2002] Jones, M. T. (2002). Java mobile agents & the aglets SDK. *Dr. Dobb's Journal*, 27(1):42–48. 96

[Karger *et al.*, 1997] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M. et Lewin, D. (1997). Consistent hashing and random trees : distributed caching protocols for relieving hot spots on the world wide web. *In Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663. 21

[Kermarrec *et al.*, 2003] Kermarrec, A.-M., Massoulié, L. et Ganesh, A. J. (2003). Probabilistic reliable dissemination in large-scale systems. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):248–258. 28, 31, 32, 40

[Kermarrec *et al.*, 2011] Kermarrec, A.-M., Straub, G. et Scouarnec, N. L. (2011). Repairing multiple failures with coordinated and adaptive regenerating codes. *CoRR*, abs/1102.0204. 54, 56, 61

[Kirkpatrick *et al.*, 1983] Kirkpatrick, S., Gelatt, C. D., Jr. et Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220:671–680. 157

[Koetter et Médard, 2003] Koetter, R. et Médard, M. (2003). An algebraic approach to network coding. *IEEE/ACM Transactions on Networking*, 11(5):782–795. 53

[Kondo *et al.*, 2008] Kondo, D., Andrzejak, A. et Anderson, D. P. (2008). On correlated availability in internet-distributed systems. *In GRID*, pages 276–283. 73, 74, 156

[Kubiatowicz *et al.*, 2000] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C. et Zhao, B. (2000). Oceanstore : an architecture for global-scale persistent storage. *SIGPLAN Notices*, 35(11):190–201. 49, 62, 63

[Kulbak et Bickson, 2004] Kulbak, Y. et Bickson, D. (2004). The eMule Protocol Specification. Rapport technique, School of Computer Science and Engineering of The Hebrew University of Jerusalem. 13

[Lamotte, 1992] Lamotte, J.-L. (1992). *Restauration d'images par la méthode du recuit simulé, implantation sur une machine parallèle à base de transputers.* Thèse de doctorat, Université de Caen Basse-Normandie. 157

[Lange *et al.*, 1997] Lange, D. B., Oshima, M., Karjoth, G. et Kosaka, K. (1997). Aglets : Programming Mobile Agents in Java. *In Proceedings of the International Conference on Worldwide Computing and Its Applications*, WWCA '97, pages 253–266. 96

[Latombe, 1991] Latombe, J.-C. (1991). *Robot Motion Planning :.* Kluwer international series in engineering and computer science : Robotics. Kluwer Academic Publishers. 84

[Leriche, 2006] Leriche, S. (2006). *Architectures à composants et agents pour la conception d'applications réparties adaptables.* Thèse de doctorat, Université de Toulouse II. 91, 93, 95

[Li *et al.*, 2010] Li, J., Yang, S., Wang, X. et Li, B. (2010). Tree-structured data regeneration in distributed storage systems with regenerating codes. *In Proceedings of the 29th conference on Information communications*, INFOCOM'10, pages 2892–2900. 53

[Li et Yeung, 2003] Li, S. et Yeung, R. (2003). Linear network coding. *IEEE Transactions On Information Theory*, 49(2):371–381. 53

[Lian *et al.*, 2005] Lian, Q., Chen, W. et Zhang, Z. (2005). On the impact of replica placement to the reliability of distributed brick storage systems. *In Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, ICDCS '05, pages 187–196. 61

[Liang *et al.*, 2004] Liang, J., Kumar, R. et Ross, K. (2004). Understanding KaZaA. 14

[Liang *et al.*, 2006] Liang, J., Kumar, R. et Ross, K. W. (2006). The FastTrack overlay : a measurement study. *Computer Networks*, 50(6):842–858. 13, 14

[Lin *et al.*, 2009] Lin, Y., Liang, B. et Li, B. (2009). Priority random linear codes in distributed storage systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(11):1653–1667. 53

[Lozenguez *et al.*, 2011] Lozenguez, G., Adouane, L., Beynier, A., Martinet, P. et Mouaddib, A.-I. (2011). Map partitioning to approximate an exploration strategy in mobile robotics. *In Advances on Practical Applications of Agents and Multiagent Systems*, volume 88 de *Advances in Intelligent and Soft Computing*, pages 63–72. Springer Berlin / Heidelberg. 85

[Lua *et al.*, 2005] Lua, E. K., Crowcroft, J., Pias, M., Sharma, R. et Lim, S. (2005). A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93. 16

[Lv *et al.*, 2002] Lv, Q., Cao, P., Cohen, E., Li, K. et Shenker, S. (2002). Search and replication in unstructured peer-to-peer networks. *In Proceedings of the 16th international conference on Supercomputing*, ICS '02, pages 84–95. 16

[Maggi et Sisto, 2003] Maggi, P. et Sisto, R. (2003). A configurable mobile agent data protection protocol. *In Proceedings of the second international joint conference on Autonomous agents and multiagent systems*, AAMAS '03, pages 851–858. 94

[Matignon *et al.*, 2012] Matignon, L., Jeanpierre, L. et Mouaddib, A.-I. (2012). Coordinated Multi-Robot Exploration under Communication Constraints using Decentralized Markov Decision Processes. *In Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI-12)*. 85

[Maymounkov et Mazières, 2002] Maymounkov, P. et Mazières, D. (2002). Kademlia : A Peer-to-Peer Information System Based on the XOR Metric. *In IPTPS*, pages 53–65. 19, 24

[Metropolis *et al.*, 1953] Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A. et Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. *Journal of Chemical Physics*, 21:1087. 156, 157

[Milgram, 1967] Milgram, S. (1967). The small-world problem. *Psychology Today*, 1(1):61–67. 26

[Milojičić *et al.*, 1999] Milojičić, D., Douglis, F. et Wheeler, R. (1999). *Mobility : processes, computers, and agents*. ACM Press Series. Addison-Wesley. 89

[Milojičić *et al.*, 1999] Milojičić, D., Breugst, M., Busse, I., Campbell, J., Covaci, S., Friedman, B., Kosaka, K., Lange, D., Ono, K., Oshima, M., Tham, C., Virdhagriswaran, S. et White, J. (1999). Mobility. chapitre MASIF, the OMG Mobile Agent System Interoperability Facility, pages 628–641. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. 93

[Milojíčić *et al.*, 2000] Milojíčić, D. S., Douglis, F., Paindaveine, Y., Wheeler, R. et Zhou, S. (2000). Process migration. *ACM Computing Surveys*, 32(3):241–299. 89

[Monmarché, 2000] Monmarché, N. (2000). *Algorithmes de fourmis artificielles : application à la classification et à l'optimisation*. Thèse de doctorat, Université de Tours. 83

[Napster, 1999] Napster (1999). `http://www.napster.com`. 11

[Nath *et al.*, 2006] Nath, S., Yu, H., Gibbons, P. B. et Seshan, S. (2006). Subtleties in tolerating correlated failures in wide-area storage systems. *In Proceedings of the 3rd conference on*

Networked Systems Design & Implementation - Volume 3, NSDI'06, pages 17–17, Berkeley, CA, USA. USENIX Association. 75

[Ohsuga *et al.*, 1997] Ohsuga, A., Nagai, Y., Irie, Y., Hattori, M. et Honiden, S. (1997). Plangent : An Approach To Making Mobile Agents Intelligent. *IEEE Internet Computing*, 1(4):50–57. 96

[Papadakis *et al.*, 2008] Papadakis, N., Doulamis, A., Litke, A., Doulamis, N., Skoutas, D. et Varvarigou, T. (2008). MI-MERCURY : A mobile agent architecture for ubiquitous retrieval and delivery of multimedia information. *Multimedia Tools and Applications*, 38(1): 147–184. 99, 100, 101

[Paul *et al.*, 1997] Paul, S., Sabnani, K. K., Lin, J. C.-H. et Bhattacharyya, S. (1997). Reliable multicast transport protocol (rmtp). *IEEE Journal on Selected Areas in Communications*, 15(3):407–421. 30

[Peine, 1998] Peine, H. (1998). Security concepts and implementation in the ara mobile agent system. *In Proceedings of the 7th Workshop on Enabling Technologies : Infrastructure for Collaborative Enterprises*, WETICE '98, pages 236–242. 96

[Peine et Stolpmann, 1997] Peine, H. et Stolpmann, T. (1997). The Architecture of the Ara Platform for Mobile Agents. *In Proceedings of the First International Workshop on Mobile Agents*, MA '97, pages 50–61. 96

[PIAX, 2009] PIAX (2009). P2P Interactive Agent eXtensions. `http://www.piax.org/en/`. 91, 96

[Picco, 1998] Picco, G. P. (1998). *Understanding, evaluating, formalizing, and exploiting code mobility*. Thèse de doctorat, Politecnico Di Torino. 87, 95, 96

[Picco, 1999] Picco, G. P. (1999). uCODE : A Lightweight and Flexible Mobile Code Toolkit. *In Proceedings of the Second International Workshop on Mobile Agents*, MA '98, pages 160–171. 96

[Picco, 2002] Picco, G. P., éditeur (2002). *Mobile Agents, 5th International Conference, MA 2001 Atlanta, GA, USA, December 2-4, 2001, Proceedings*, volume 2240 de *Lecture Notes in Computer Science*. Springer. 91

[Picco *et al.*, 1999] Picco, G. P., Murphy, A. L. et Roman, G.-C. (1999). LIME : Linda meets mobility. *In Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 368–377. 96

[Plank, 1997] Plank, J. S. (1997). A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Software – Practice & Experience*, 27(9):995–1012. 49, 51

[Plank et Ding, 2005] Plank, J. S. et Ding, Y. (2005). Note : Correction to the 1997 tutorial on reed-solomon coding. *Software – Practice & Experience*, 35(2):189–194. 49, 51

[Plaxton *et al.*, 1997] Plaxton, C. G., Rajaraman, R. et Richa, A. W. (1997). Accessing nearby copies of replicated objects in a distributed environment. *In Proceedings of the ninth*

annual ACM symposium on Parallel algorithms and architectures, SPAA '97, pages 311–320. 22

[Pommier, 2010] Pommier, H. (2010). *Placement et stockage de l'information bio-inspiré : une approche orientée agent mobile*. Thèse de doctorat, Université de Caen Basse-Normandie. 90, 91, 95, 107, 111, 114, 127

[Pommier et Bourdon, 2009] Pommier, H. et Bourdon, F. (2009). Agents mobiles et réseaux pair-à-pair vers une gestion sécurisée de l'information répartie. *Revue d'Intelligence Artificielle*, 23(5-6):697–718. 107

[Pommier *et al.*, 2010] Pommier, H., Romito, B. et Bourdon, F. (2010). Bio-inspired Data Placement in Peer-to-Peer Networks - Benefits of using Multi-agents Systems. *In Proceedings of the 6th International Conference on Web Information Systems and Technologie*, pages 319–324. 107, 112

[Pommier *et al.*, 2011] Pommier, H., Romito, B. et Bourdon, F. (2011). Searching flocks in peer-to-peer networks. *In Advances on Practical Applications of Agents and Multiagent Systems*, volume 88 de *Advances in Intelligent and Soft Computing*, pages 103–108. Springer Berlin / Heidelberg. 125, 127

[Poole *et al.*, 1998] Poole, D., Mackworth, A. et Goebel, R. (1998). *Computational Intelligence : A Logical Approach*. Oxford University Press. 79

[Postel, 1981] Postel, J. (1981). Transmission Control Protocol. RFC 793 (Standard). Updated by RFCs 1122, 3168, 6093, 6528. 13

[Puterman, 1994] Puterman, M. (1994). *Markov decision processes : discrete stochastic dynamic programming*. Wiley series in probability and statistics. Wiley-Interscience. 82

[Rao et Georgeff, 1995] Rao, A. S. et Georgeff, M. P. (1995). Bdi agents : From theory to practice. *In ICMAS*, pages 312–319. 84

[Ratnasamy *et al.*, 2001] Ratnasamy, S., Francis, P., Handley, M., Karp, R. et Shenker, S. (2001). A scalable content-addressable network. *In Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172. 19, 21

[Reed et Solomon, 1960] Reed, I. et Solomon, G. (1960). Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304. 51

[Reynolds, 2006] Reynolds, C. (2006). Big fast crowds on ps3. *In Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*, pages 113–121. 84

[Reynolds, 1987] Reynolds, C. W. (1987). Flocks, herds and schools : A distributed behavioral model. *In Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '87, pages 25–34. 84, 110

[Reynolds, 1999] Reynolds, C. W. (1999). Steering Behaviors For Autonomous Characters. *In Game Developers Conference*, pages 763–782. 110

[Rhea *et al.*, 2003] Rhea, S. C., Eaton, P. R., Geels, D., Weatherspoon, H., Zhao, B. Y. et Kubiatowicz, J. (2003). Pond : The oceanstore prototype. *In FAST*. 64

[Rich et Knight, 1991] Rich, E. et Knight, K. (1991). *Artificial intelligence*. Artificial Intelligence Series. McGraw-Hill. 79

[Ricordel, 2001] Ricordel, P.-M. (2001). *Programmation Orientée Multi-Agents : Développement et Déploiement de Systèmes Multi-Agents Voyelles*. Thèse de doctorat, Institut National Polytechnique de Grenoble. 79

[Rivest, 1992] Rivest, R. (1992). The MD4 Message-Digest Algorithm. RFC 1320 (Historic). Obsoleted by RFC 6150. 13

[Rizzo, 1997] Rizzo, L. (1997). Effective erasure codes for reliable computer communication protocols. *ACM SIGCOMM Computers Communication Review*, 27(2):24–36. 49

[Rodrigues et Liskov, 2005] Rodrigues, R. et Liskov, B. (2005). High Availability in DHTs : Erasure Coding vs. Replication. *In IPTPS*, pages 226–239. 50, 54

[Romito et Bourdon, 2012a] Romito, B. et Bourdon, F. (2012a). Reducing Correlated Failures Impact in Peer-to-peer Storage Systems Using Mobile Agents Flocks. *In Proceedings of the 2012 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology*, WI-IAT '12. (à paraitre). 159

[Romito et Bourdon, 2012b] Romito, B. et Bourdon, F. (2012b). Réduction de l'impact des fautes corrélées dans les réseaux pair-à-pair en utilisant des nuées d'agents mobiles. *In JFSMA*. (à paraitre). 159

[Romito *et al.*, 2011] Romito, B., Pommier, H. et Bourdon, F. (2011). Repairing flocks in peer-to-peer networks. *In Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 02*, WI-IAT '11, pages 308–312. 135

[Roth, 2004] Roth, V. (2004). Obstacles to the Adoption of Mobile Agents. *In IEEE International Conference on Mobile Data Management*, pages 296–297. 93

[Rowstron et Druschel, 2001a] Rowstron, A. I. T. et Druschel, P. (2001a). Pastry : Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *In Middleware*, pages 329–350. 64

[Rowstron et Druschel, 2001b] Rowstron, A. I. T. et Druschel, P. (2001b). Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *In SOSP*, pages 188–201. 64

[Rowstron *et al.*, 2001] Rowstron, A. I. T., Kermarrec, A.-M., Castro, M. et Druschel, P. (2001). Scribe : The design of a large-scale event notification infrastructure. *In Networked Group Communication*, pages 30–43. 31

[Russell et Norvig, 2010] Russell, S. et Norvig, P. (2010). *Artificial Intelligence : A Modern Approach*. Prentice Hall Series in Artificial Intelligence. Prentice Hall. 78, 80, 81

[Sahai et Morin, 1998] Sahai, A. et Morin, C. (1998). Mobile agents for enabling mobile user aware applications. *In Proceedings of the second international conference on Autonomous agents*, AGENTS '98, pages 205–211. 92

[Sandberg, 2006] Sandberg, O. (2006). Distributed routing in small-world networks. *ALENEX*. 26

[Saroiu *et al.*, 2003] Saroiu, S., Gummadi, K. P. et Gribble, S. D. (2003). Measuring and analyzing the characteristics of Napster and Gnutella hosts. *Multimedia Systems*, 9(2):170–184. 11

[Satoh, 2002] Satoh, I. (2002). Physical mobility and logical mobility in ubiquitous computing environments. *In Proceedings of the 6th International Conference on Mobile Agents*, MA '02, pages 186–202. 99

[Searle, 1969] Searle, J. (1969). *Speech Acts : An Essay in the Philosophy of Language*. Cambridge University Press. 85

[Sigaud et Buffet, 2010] Sigaud, O. et Buffet, O. (2010). *Markov Decision Processes in Artificial Intelligence*. Wiley-IEEE Press. 82

[Smith, 1980] Smith, R. G. (1980). The Contract Net Protocol : High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transaction on Computers*, 29(12):1104–1113. 85

[Stoica *et al.*, 2001] Stoica, I., Morris, R., Karger, D. R., Kaashoek, M. F. et Balakrishnan, H. (2001). Chord : A scalable peer-to-peer lookup service for internet applications. *In SIGCOMM*, pages 149–160. 19, 20

[Suh et Ramchandran, 2010] Suh, C. et Ramchandran, K. (2010). Exact regeneration codes for distributed storage repair using interference alignment. *CoRR*, abs/1001.0107. 54

[Suri *et al.*, 2000] Suri, N., Bradshaw, J., Breedy, M. R., Groth, P. T., Hill, G. A. et Jeffers, R. (2000). Strong Mobility and Fine-Grained Resource Control in NOMADS. *In Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents*, ASA/MA 2000, pages 2–15. 96

[Szymonacedaski *et al.*, 2005] Szymonacedaski, Deb, S., Medard, M. et Koetter, R. (2005). How Good is Random Linear Coding Based Distributed Networked Storage ? *In 1st Workshop on Network Coding, Theory and Applications (NetCod)*. 53

[Teranishi, 2009] Teranishi, Y. (2009). PIAX : toward a framework for sensor overlay network. *In Proceedings of the 6th IEEE Conference on Consumer Communications and Networking Conference*, CCNC'09, pages 1212–1216. 96

[Tisseau, 2001] Tisseau, J. (2001). Réalité Virtuelle — autonomie in virtuo —. Habilitation à diriger les recherches, Université de Rennes 1. 80, 83, 86

[Topaloglu et Bayrak, 2008] Topaloglu, U. et Bayrak, C. (2008). Secure mobile agent execution in virtual environment. *Autonomous Agents and Multi-Agent Systems*, 16(1):1–12. 93

[Urra *et al.*, 2009] Urra, O., Ilarri, S., Mena, E. et Delot, T. (2009). Using hitchhiker mobile agents for environment monitoring. *In Proceedings of the 7th International Conference on Practical Applications of Agents and Multi-Agent System*, pages 557–566. 99

[Vaquero *et al.*, 2009] Vaquero, L. M., Rodero-merino, L., Caceres, J., Lindner, M. et Desarrollo, T. I. Y. (2009). A break in the clouds : Towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, pages 50–55. 47

[Voyager, 2012] Voyager (2012). `http://www.recursionsw.com/products/voyager/`. 96

[Wagner et Bruckstein, 1999] Wagner, I. et Bruckstein, A. (1999). Hamiltonian(t) - An Ant-Inspired Heuristic for Recognizing Hamiltonian Graphs. *In Proceedings of Congress on Evolutionary Computation (CEC99)*. 83

[Wagner *et al.*, 2008] Wagner, I. A., Altshuler, Y., Yanovski, V. et Bruckstein, A. M. (2008). Cooperative cleaners : A study in ant robotics. *International Journal of Robotics Research*, 27(1):127–151. 83

[Walsh *et al.*, 1998] Walsh, T., Paciorek, N. et Wong, D. (1998). Security and Reliability in Concordia. *In Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7 - Volume 7*, HICSS '98, pages 44–. 96

[Watanabe *et al.*, 2004] Watanabe, Y., Sato, S. et Ishida, Y. (2004). An Approach for Self-repair in Distributed System Using Immunity-Based Diagnostic Mobile Agents. *In Knowledge-Based Intelligent Information and Engineering Systems*, volume 3214 de *Lecture Notes in Computer Science*, pages 504–510. 99, 100

[Watts et Strogatz, 1998] Watts, D. J. et Strogatz, S. H. (1998). Collective dynamics of small-world networks. *Nature*, 393(6684):440–442. 26

[Weatherspoon et Kubiatowicz, 2002] Weatherspoon, H. et Kubiatowicz, J. (2002). Erasure coding vs. replication : A quantitative comparison. *In Proceedings of the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 328–338, London, UK, UK. Springer-Verlag. 50

[Weatherspoon *et al.*, 2002] Weatherspoon, H., Moscovitz, T. et Kubiatowicz, J. (2002). Introspective failure analysis : Avoiding correlated failures in peer-to-peer systems. *In Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, pages 362–, Washington, DC, USA. IEEE Computer Society. 71, 74, 156

[White, 1994] White, J. E. (1994). Telescript technology : The foundation for the electronic marketplace. Rapport technique, General Magic, Inc. 96

[Wireless Broadband Alliance Ltd., 2011] Wireless Broadband Alliance Ltd. (2011). Global Developments in Public Wi-Fi. `http://www.wballiance.com/resource-centre/global-developments-wifi-report.html`. xi, 46

[Wong *et al.*, 1997] Wong, D., Paciorek, N., Walsh, T., DiCelie, J., Young, M. et Peet, B. (1997). Concordia : An Infrastructure for Collaborating Mobile Agents. *In Proceedings of the First International Workshop on Mobile Agents*, MA '97, pages 86–97. 96

[Wooldridge et Jennings, 1995] Wooldridge, M. et Jennings, N. (1995). Intelligent agents : Theory and practice. *Knowledge engineering review*, 10(2):115–152. 80, 82

[Wu *et al.*, 2007] Wu, Y., Dimakis, R. et Ramchandran, K. (2007). Deterministic regenerating codes for distributed storage. *In Allerton Conference on Control, Computing, and Communication (Urbana-Champaign, IL)*. 54, 55, 56

[Wuala, 2012] Wuala (2012). `http://www.wuala.com/`. 69

[Zakon, 1997] Zakon, R. (1997). Hobbes' Internet Timeline. RFC 2235 (Informational). xi, 8

[Zakon, 2011] Zakon, R. (2011). Hobbes' Internet Timeline 10.2. `http://www.zakon.org/robert/internet/timeline/`. xi, 8

[Zhang *et al.*, 2007] Zhang, Z., Lian, Q., Lin, S., Chen, W., Chen, Y. et Jin, C. (2007). BitVault : a highly reliable distributed data retention platform. *Operating Systems Review*, 41(2):27–36. 60, 62, 69

[Zhao *et al.*, 2004] Zhao, B. Y., Huang, L., Stribling, J., Rhea, S. C., Joseph, A. D. et Kubiatowicz, J. D. (2004). Tapestry : A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22:41–53. 19, 22

[Zhuang *et al.*, 2001] Zhuang, S., Zhao, B. Y., Joseph, A. D., Katz, R. H. et Kubiatowicz, J. (2001). Bayeux : an architecture for scalable and fault-tolerant wide-area data dissemination. *In NOSSDAV*, pages 11–20. 31

Stockage décentralisé adaptatif : autonomie et mobilité des données dans les réseaux pair-à-pair

Nous étudions une nouvelle approche pour le stockage décentralisé dans les réseaux pair-à-pair, qui consiste à transférer la responsabilité du stockage des données, des pairs aux documents eux-mêmes. Les documents ne sont plus des ensembles de données inertes sur lesquels sont appliqués des traitements. Ils deviennent autonomes et responsables de leur propre durabilité. À l'aide d'une modélisation sous la forme d'un système multi-agents mobiles et d'algorithmes bio-inspirés, nous transformons chaque document en une nuée d'agents mobiles, capable de se déplacer dans le réseau. Nous nous intéressons, dans un premier temps, à la faisabilité de cette approche, que nous évaluons au moyen de plusieurs expériences, réalisées sur un prototype déployé dans un réseau pair-à-pair réel. Nous constatons, sous certaines hypothèses, que nos algorithmes de déplacement en nuées sont valides et que les relations topologiques entre agents sont suffisantes pour faire émerger le comportement global de flocking. Dans un second temps, nous nous intéressons aux mécanismes permettant d'assurer la durabilité de ces nuées en présence de fautes. Ces nuées présentent des capacités d'auto-adaptation, qui leur permettent notamment de trouver le schéma de fragmentation adéquat, pour l'instance de réseau considérée et pour le niveau de disponibilité requis. Nous étudions plus en détail cette capacité d'adaptation dans le contexte des fautes corrélées, en proposant et en évaluant un algorithme de placement décentralisé qui permet de réduire l'impact des fautes corrélées sur les systèmes de stockage décentralisés.

Adaptive and decentralized storage : data autonomy and mobility in peer-to-peer networks

We study a new approach for decentralized data storage in peer-to-peer networks. In this approach, the responsibility of data management is transferred from the peers to the documents. It means that documents are not passive data sets anymore but become autonomous and responsible for their own durability. Thanks to a multi-agent system modeling and bio-inspired algorithms, we transform each document into a mobile agents flock able to move into the network. Firstly, we assess the feasibility of this approach with several experiments done on a prototype deployed in a real peer-to-peer network. We note that, given some hypothesis, our motion algorithms are sound. We also note that, topological relationships between the agents are enough for the emergence of a global flocking behavior. Secondly, we focus on mechanisms required to ensure flocks durability. We note that those flocks are self -adaptive and that, this property can be used to find the accurate fragmentation parameters, given a network instance and a required level of availability. Finaly, we study this self-adaptation property in the context of correlated failures. We propose and we analyze a decentralized flock placement algorithm aimed at reducing the correlated failures impact on data storage systems.

Mots-clés :

– **indexation Rameau : Systèmes multi-agents ; Réseaux pair-à-pair ; Agents mobiles ; Stockage décentralisé ; Auto-adaptation ; Mobilité ; Intelligence artificielle**

Discipline : Informatique et applications

Groupe de Recherche en Informatique, Image, Automatique et Instrumentation de Caen,
CNRS UMR 6072,Université de Caen Basse-Normandie BP 5186,14032 Caen Cedex, FRANCE