Partage De Données En Mode Pair A Pair Sur Réseaux Mobiles Ad Hoc

Hoa Dung Ha Duong

École Doctorale
d'Informatique,
Télécommunications et
Électronique de Paris

Thèse

présentée pour obtenir le grade de docteur

de l'École Nationale Supérieure des Télécommunications

Spécialité : Informatique et Réseaux

Hoa Dung HA DUONG

Partage de données en mode pair à pair sur réseaux mobiles ad hoc

Soutenue le 24 septembre 2010 devant le jury composé de

Président	Michel Raynal	Professeur à l'Université de Rennes 1
Rapporteurs	Eric Gressier-Soudan	Professeur au Conservatoire National des Arts et Métiers
	Pierre Sens	Professeur à l'Université Pierre et Marie Curie
Examinateur	Yves Mahéo	Maître de conférence à l'Université de Bretagne Sud
Directrice de thèse	Isabelle Demeure	Professeur à Télécom Paristech

Remerciements

Quand un de mes aînés, académiques s'entend, m'a prévenue, lors de la rédaction de ce manuscrit, que les remerciements étaient la partie la plus difficile à écrire, je suis restée dubitative. C'est pourtant bien vrai.

Je remercie tout d'abord Mme Isabelle Demeure, qui a dirigé cette thèse, et à qui j'aimerai une fois de plus exprimer non seulement ma gratitude, mais aussi tout le respect et l'admiration que je lui porte.

Je remercie bien entendu les autres membres du jury.
M. Michel Raynal, pour avoir accepté de présider ce jury, et pour son aide concernant l'algorithmique distribuée. Bien que mes travaux aient pris une autre direction, ce fut, pour moi, une discussion enrichissante.
M. Eric Gressier et M. Pierre Sens, qui ont accepté de rapporter cette thèse. Ils ont par ailleurs été mes professeurs lors de mes études de Master, et leurs enseignements m'ont permis de m'intéresser, entre autres choses, aux problématiques des systèmes distribués et à la recherche. Pour cela aussi, je les remercie donc.
M. Yves Mahéo, avec qui j'ai malheureusement peu eu l'occasion de discuter, mais dont les questions lors de la soutenance m'ont montré l'attention qu'il avait porté à ces travaux.

J'ai rencontré lors de ces années de thèse trop de personnes pour pouvoir toutes les nommer. Je remercie donc les personnes et les groupes suivants.

Les permanents et post docs du département InfRes, Céline, Hayette, ainsi que M. Robinet et Mme Besnard pour tous les conseils, les encouragements et les cafés offerts. J'aimerai remercier en particuliers les membres de l'équipe S3 qui, ayant assistés aux pré-soutenances, m'ont permis de grandement améliorer mon discours.

M. Marc Shapiro, pour son aide et ses commentaires concernant la réplication de données, qui constitue une grande part de ces travaux.

Les anciens thésards de Télécom devenus docteurs avant moi et qui m'ont donc montré la faisabilité de la thèse. Je remercie particulièrement Dr Thomas Vergnaud et Dr Eric Varadaradjou, pour m'avoir fait une place dans leur bureau et pour leur soutien moral lors des moments de doute. Par ordre d'ancienneté, Dr Irfan Hamid, parti au pays des séries, Dr Bechir Zalila, just married, Dr Raja Chiky, à qui je dois toujours un verre rue Oberkampf, Dr Meng Song, à qui je retournerai rendre visite bientôt, j'espère, Dr Etienne Borde, revenu récemment au vaisseau mère, Dr Julien Delange, en exil au pays des tulipes, Dr Olivier Gilles, cowboy chasseur, Dr Xavier Renault, l'homme à la blague unique et enfin mon jumeau de thèse, Dr Xavier Grehant. Bien joué les gens !

Les actuels thésards de Télécom : Gilles Lasnier, Tung Thanh Vu, Nora Derouiche, Khalil El Mahrsi, Mike Lafaye, Fabien Cadoret, Simon Perrault, Marilena Oita, Damien Munch, Azin Arya, Cuauthemoc Castellanos, et tous mes voisins d'open space. Bon courage les gens ! J'espère vous voir soutenir.

Les enseignants chercheurs du département Math Info de l'Université Paris V, et les enseignants chercheurs de l'équipe Architecture Systèmes et Réseaux, de l'IUT Paris Descartes, équipes aux seins desquelles j'ai effectué un monitorat puis un ATER. L'apprentissage de l'enseignement passe par la pratique et sans leurs conseils, celle ci aurait été plus ardue.

Les magistériens de la promo 2006, en particuliers Thibault, qui a fait l'aller retour de province pour venir me voir soutenir, Fabio qui a fait l'aller retour de Brunoy, et Jean, qui a fait la plus longue pause déjeuner connue à ce jour, les SAR et les thésards du LIP6 de diverses années, en particulier Lom Messan Hillah dont les conseils m'ont permis de trouver un post doctorat, Mathieu, qui a pu ainsi s'entraîner à préparer et débarrasser un pot de thèse en prévision du sien, tous les amis qui m'ont fait l'honneur de se déplacer pour assister à ma soutenance, et ceux qui n'ont pas pu.

Enfin, j'aimerai remercier mes parents, mes oncles et tantes, mes frères et soeurs, et mes cousins cousines, pour leur soutien et leur exemple. Surtout, j'aimerai remercier ma mère, qui a traqué les fautes de ce manuscrit (j'espère ne pas en avoir réintroduites) et préparé un pot de soutenance dont on m'a dit le plus grand bien. Par ailleurs, je profite de ces dernières ligne pour transmettre un message à mes neveux et nièces, qu'ils ne liront pas : si vous décidez de suivre l'exemple de votre Très Honorable Tante, attention, les remerciements sont la partie la plus difficile à écrire.

Résumé

Le développement d'applications collaboratives sur réseaux mobiles ad hoc présente de nouvelles contraintes, liées à la mobilité et la volatilité des terminaux, à la nature distribuée des MANets, ainsi qu'aux ressources limitées disponibles. Elle nécessite donc la mise en place d'une algorithmique différente de celle utilisée dans les applications collaboratives pour réseaux filaires.
Cette thèse propose donc un ensemble d'algorithmes permettant de mettre en place un système de partage de données distribué sur MANet.
Tout d'abord nous proposons un algorithme de création de grappes de terminaux mobiles stables dans le temps. Cet algorithme présente l'avantage de ne pas utiliser de ressources réseaux car il s'appuie sur des informations inter-couches.
Nous proposons ensuite un algorithme de réplication de données pro-actif, qui vise à créer et maintenir un nombre de répliques proportionnel au nombre de terminaux présents. Il utilise des informations sémantiques afin de placer ces répliques sur les terminaux les plus susceptibles de les utiliser. Ceci permet d'augmenter la disponibilité (les données sont plus rapidement accessibles) et la fiabilité du service de partage de données (en cas de disparition d'un hôte ou de partition du réseau, la probabilité qu'une donnée disparaisse est diminuée).
Nous proposons enfin un algorithme de gestion de cache, qui vise à maintenir le nombre de répliques de chaque donnée au minimum permettant une utilisation efficace du réseau, tout en offrant une bonne disponibilité. D'une part, quand un remplacement de cache est nécessaire, il choisit d'éliminer les données utilisées le moins récemment et pour lesquelles le nombre de répliques est suffisamment élevé pour maintenir la fiabilité et la disponibilité du service. D'autre part, il diminue la charge réseau en éliminant préventivement les répliques de données qui génèrent du trafic réseau inutile car elles ne sont pas utilisées par leur hôte.
Comme les expériences sur MANets sont complexes à déployer, et à rendre reproductibles, nous avons validé notre proposition par simulation, et montré son impact positif sur la disponibilité des données, la fiabilité du service de partage de données, et l'utilisation efficace des ressources.
Enfin, nous présentons un démonstrateur sous la forme de moteur de wiki pair à pair pour MANet, qui utilise TreeDoc pour maintenir la cohérence des données. Il est conçu pour s'intégrer au sein de l'intergiciel Transhumance, et a permis une validation fonctionnelle de notre proposition.

Mots-clés : MANet, Partage de données, Répartition, Pair à Pair, Mobilité

Abstract

Designing collaborative software for mobile ad hoc networks presents news challenges : in a MANEt, devices are mobile, resources are scarce and because of an absence of infrastructure, everything is decentralized. Therefore, algorithms for collaborative application on wired networks cannot be used as is, and new algorithms, adapted to MANet, have to be defined.
This thesis proposes a set of algorithms for developing a distributed data sharing system for MANets.
First of all we propose a clustering algorithm to create groups of terminals stable over time. This algorithm's main advantage is that it does not create any network load since it uses cross-layering information from the routing layer to know which terminals are reachable.
We also propose a proactive replication algorithm, which, for each data, creates and maintains a number of replica proportional to the number of terminals. It uses semantic information about data and users to try and place these replica on the devices where they are most likely to be used. This algorithm enhances data availability (accesses are faster) and data sharing reliability (if a terminal disappears, or if the network splits, the probability of losing a data is lowered).
Finally, we propose a cache management algorithm. This algorithm tries to regulate the best number of replica so as to optimize the use of the network, while maintaining data availability. First of all, when a cache replacement is needed, the algorithm eliminates data less recently used, and for which the number of replica is high enough to maintain data sharing availability. It also reduces the network load by removing useless replica which generate network traffic and are not used by their host.
As it is hard to make reproducible experiments on top of a MANet, our proposition was validated by simulation and we have demonstrated a positive influence on data availability, the data sharing service overall reliability and an efficient use of resources.
We also present, as a demonstrator, a peer to peer wiki engine, design toward mobile ad hoc networks, which we have used to provide a functional validation of our algorithms. It uses Treedoc to maintain data coherency and is designed to be integrated in the Transhumance middleware.
Keyword : MANet, Data sharing, Distributed, Peer to Peer

Table des matières

Chapitre 1

Introduction

Dans cette thèse nous proposons un système de partage de données permettant aux utilisateurs de terminaux mobiles de collaborer en l'absence d'infrastructure.

1.1 Motivation

1.1.1 Contexte

Les terminaux légers équipés de modules de communications sans fil, comme les *smartphones* (iPhone, HTC Magic) ou les *subnotebooks* (EEE PC), sont désormais des objets de consommation courante. Les utilisateurs de ces terminaux sont maintenant habitués à utiliser des applications collaboratives, accessibles via le web (comme par exemple un wiki [60]) ou par d'autres moyens (comme l'éditeur de texte collaboratif *SubEthaEdit* [87]). Cependant, pour utiliser ces applications, il est bien souvent nécessaire que les terminaux soient connectés à travers une infrastructure réseau, comme un LAN ou un réseau cellulaire. Les terminaux pouvant communiquer entre eux sans fil, il est possible à plusieurs utilisateurs de collaborer même en l'absence d'infrastructure. Cela nécessite toutefois de modifier la pile réseau car, par défaut, les terminaux n'ont pas la capacité de relayer l'information ce qui est nécessaire en l'absence d'infrastructure.
Pour ce faire, plusieurs protocoles de routage ont été créés permettant aux terminaux de se constituer spontanément en réseau. Ces réseaux mobiles ad hoc, appelés aussi MANet [1], possèdent donc de nombreuses caractéristiques les différenciant des réseaux filaires classiques [23]. En l'absence d'infrastructure, la gestion du réseau (routage, contrôle de congestion), doit être distribuée. Par ailleurs, la capacité de stockage des terminaux tels

1. Mobile Ad hoc NETwork

que *smartphone* et *netbook* est moindre que celle de serveurs ou de stations de travail, et la durée de vie d'une session de travail est limitée par la batterie. Enfin, le réseau est susceptible de voir sa topologie et sa capacité de transport évoluer du fait de la mobilité.
Les applications collaboratives pour MANet doivent s'adapter à la topologie dynamique de ceux-ci et pouvoir tolérer une partition du réseau. Par ailleurs, comme les terminaux ont des capacités limitées, ces systèmes doivent limiter leur usage des ressources réseaux et de la mémoire. Enfin, comme il n'existe pas de terminal fiable qui puisse faire office de serveur, il est plus sûr de concevoir les applications de façon entièrement distribuée.
Dans cette thèse nous nous intéressons plus particulièrement au problème de partage de données dans les MANets. Une offre de partage de données doit assurer l'*accessibilité* et la *pérennité* des données. Pour ce faire, comme dans notre contexte on ne dispose pas de serveur fiable, les données sont *répliquées* dans le réseau. Cela nécessite donc d'en assurer la *cohérence*, la *sécurité* et la *confidentialité*, ainsi qu'une politique de *remplacement de cache*. Un service de *localisation*, permettant de retrouver une réplique à partir de l'identifiant d'une donnée, et un service de *recherche* sont aussi nécessaires. Enfin, pour pouvoir gérer la mobilité des terminaux, nous avons besoin d'un algorithme capable de classifier les terminaux de mobilité similaire en *grappes* (clustering).
Afin de motiver et d'illustrer notre approche, nous allons tout d'abord présenter un exemple du type d'application présentant un intérêt à être utilisée dans le contexte d'un MANet, et qui nécessite de partager des données. Nous identifions ensuite nos contributions dans ce domaine, puis nous présentons un plan général de la thèse.

1.1.2 Exemple : un wiki distribué nomade

A titre d'exemple d'application collaborative susceptible d'être utilisée sur un MANet, nous proposons le scénario suivant.

Figure 1.1 – En classe... [2]

Figure 1.2 – ou sur le terrain ! [2]

Dans une école, chaque élève est muni d'un terminal mobile léger et solide, équipé de connectivité WiFi. Durant leur scolarité les élèves sont amenés à faire des exposés, des fiches de lectures ou des comptes rendus d'expérience. Ils partagent leurs résultats au sein d'un wiki accessible par tous les élèves et administré par les professeurs.
Chaque année des excursions et des voyages scolaires sont organisés, au musée ou en classe verte. Les élèves peuvent bien sûr continuer à travailler individuellement, mais il est souhaitable qu'ils puissent continuer à collaborer même hors de portée du serveur.

Pour cela, il faut tout d'abord des protocoles de communication permettant aux terminaux d'interagir en absence d'une infrastructure réseau, ce que permettent les protocoles de routage de réseaux mobiles ad hoc. Le wiki doit ensuite être déployé sur les terminaux des élèves. Dans ce contexte de mobilité, il n'existe pas de terminal fiable auquel tous les utilisateurs seraient sûrs de pouvoir être connectés. Une architecture Client/Serveur est donc proscrite, car elle engendrerait trop de problèmes d'accessibilité. Par ailleurs, les terminaux sont susceptibles de disparaître et de réapparaître par intermittence. Enfin, la plupart des terminaux n'ont sans doute pas les ressources nécessaires pour héberger l'ensemble des articles.
Il faut donc, pour fournir ce service de wiki, créer un système de partage de données pair à pair, tenant compte des ressources des terminaux ainsi que de leur mobilité.

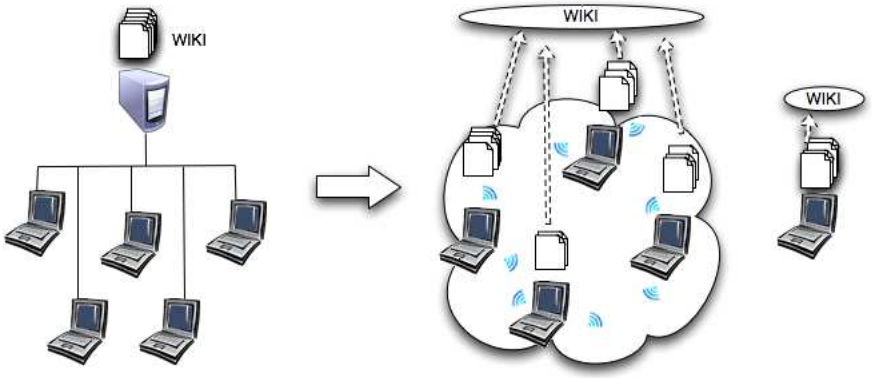


FIGURE 1.3 – Passage d'une solution centralisée sur réseau filaire, à une solution distribuée sur MANet

La majorité des systèmes de partage de données existants ne sont pas conçus pour les systèmes mobiles et ne prennent donc pas en compte les caractéristiques spécifiques des MANets. Il existe cependant des systèmes de partages de données pour MANets. La majorité de ces systèmes sont spécialisés pour des applications spécifiques, comme des réseaux de capteurs avec une modification périodique des données, ou des groupes d'intervention de crise fortement connectés. Ils ne conviennent donc pas non plus à notre situation. Enfin, certains systèmes plus généralistes existent, tels que XMiddle [69] ou adHocFS [14] mais ils n'offrent pas toutes les fonctionnalités que nous souhaitons.

1.2 Contribution

Nous proposons des algorithmes pour le partage de données dans un réseau mobile ad hoc de piétons, constitué d'une centaine de terminaux. Nous postulons qu'il y a des déconnexions et des partitions dans le réseau. Les données partagées peuvent être modifiées et la granularité de l'unité de maintien de la cohérence est de l'ordre du document.
Notre contribution s'axe sur trois propositions : utilisation d'informations sémantiques ; constitution de groupes de mobilité ; réplication pro-active et collaborative de données au sein d'un groupe, se basant sur la sémantique des données.
Ils sont destinés à s'intégrer au sein de l'intergiciel pour MANet Transhumance [81], issu d'un projet de recherche ANR auquel nous avons participé durant cette thèse.

1.2.1 Choix de conception

Nous avons fait différents choix lors de la conception des algorithmes proposés. Tout d'abord, nous avons décidé de ne pas prendre en compte la sécurité. Cependant, un module de sécurité identifiant les utilisateurs et chiffrant les données peut être conçu en parallèle de notre système puis intégré, en préservant notre modèle.
Pour des raisons d'efficacité, nous avons décidé de faire du "cross-layering" (interactions inter-couches), c'est à dire utiliser au sein de notre algorithme de création de grappes des informations issues des couches inférieures du réseau, en l'occurence les tables de routage. Bien que cela rende notre algorithme dépendant de l'algorithme de routage, ce qui n'est pas forcément souhaitable, cela permet un gain significatif en terme de performances, car cela évite de reproduire les informations de routages et leur gestion dans les couches hautes.
Après avoir étudié les différentes possibilités pour gérer la cohérence des données, nous avons décidé d'utiliser le modèle de donnée commutatif répliqué TreeDoc [86]. Nous présentons son fonctionnement et ses avantages, ainsi que les autres alternatives étudiées, dans le chapitre d'état de l'art.

1.2.2 Informations sémantiques

Nous proposons d'utiliser des informations sémantiques qualifiant le contenu sémantique des données et les centres d'intérêts des utilisateurs afin d'offrir une gestion intelligente des données.
Dans de nombreuses applications collaboratives, comme par exemple Flickr [1], les mots-clés sont utilisés pour faire de la recherche de données, de la classification et de la recommandation [70]. Nous voulons donc décrire les intérêts des utilisateurs par des mots-clés et utiliser ces informations pour calculer l'intérêt d'un utilisateur pour une donnée, et prédire ses accès.
Dans le cadre d'une communauté de taille restreinte, partageant des documents avec des annotations sémantiques, un tel calcul peut être appliqué sur l'ensemble, ou au moins une grande partie des données.
Une telle solution ne peut cependant pas être généralisée à tous les systèmes de partage de données. Dans le cas d'une mémoire partagée distribuée, la donnée étant un mot-mémoire, on ne peut pas lui attribuer d'information sémantique ; un tel système serait par ailleurs trop lourd pour ce type de système. A l'autre bout du spectre, dans le cadre d'un système de cache web, la quantité d'informations disponibles est trop importante pour qu'on puisse tout analyser, même s'il existe des systèmes de caches sémantiques.
Enfin, pour déterminer les intérêts des utilisateurs, nous examinons leurs accès aux données afin d'en extraire des motifs s'ils existent.
Nous évaluons notre système sur une base d'accès d'utilisateurs à des documents. Nous avons tout d'abord examiné les premiers accès pour lesquels nous avons déterminé des mots-clés décrivant les utilisateurs avant de vérifier nos prédictions sur le reste de la base.

1.2.3 Gestion de la mobilité : création de groupes de mobilité

Nous proposons un algorithme de création de groupes de mobilité stables dans le temps qui, en utilisant les informations des tables d'un algorithme de routage pro-actif, ne nécessite aucun échange de message.
Dans un MANet, un développeur d'application est amené à prendre en compte la mobilité des terminaux :

– Peut-on prévoir une partition à venir et répliquer un service pour en maintenir la disponibilité ?
– Peut-on prévoir les terminaux se déplaçant de concert, afin de mettre en place des algorithmes collaboratifs ?

Bien que les terminaux soient mobiles, on peut souvent compter sur l'apparition de groupes stables, c'est à dire dont les terminaux restent en communication pendant un certain temps, ceci étant dû à la nature collaborative des applications. Dans notre exemple, la classe se sépare en plusieurs groupes d'activités, chaque groupe restant à portée de communication afin de travailler ensemble.

Comme nous le verrons ci-après divers algorithmes ont été proposés pour gérer la mobilité au sein d'un MANet. Ces algorithmes utilisent différentes métriques pour définir la stabilité d'un lien : prédiction de trajectoire en utilisant des coordonnées GPS, détection de boucle au sein du graphe de routage et distance moyenne.

Notre algorithme prédit la stabilité d'un lien entre deux terminaux en examinant son évolution dans le temps : si un lien était stable dans le passé, alors il le sera dans le futur ; si un lien était volatil dans le passé, alors on ne le considère pas comme stable.

Ceci est fait en utilisant des informations issues des tables de routage. Nous avons fait ce choix d'utiliser des informations inter-couches afin d'être plus efficace : en choisissant un protocole de routage pro-actif, qui maintient les routes, nous savons quels terminaux peuvent être contactés sans avoir à échanger de messages.

Pour créer des grappes, trois types d'approche sont envisagés dans l'état de l'art. Le premier type d'approche consiste à créer des grappes au sein d'un groupe dense de terminaux : le problème est de sélectionner un ensemble de pairs stables (les têtes de grappes) pour placer sur chacun un service, puis d'agréger les pairs par proximité autour de ces têtes de grappes, comme dans [113], où ce choix se base sur les capacités (batterie, stockage, des pairs). Ces techniques ne prennent pas en compte la possibilité de partition. Le second type d'approche consiste à prédire les partitions dans le réseau en se basant sur les positions passées des terminaux, avec des informations GPS ou les informations de routages, comme proposé dans [24], [103] ou [97]. Ces techniques sont coûteuses en calcul car elles supposent de reconstruire les trajectoires des terminaux ; elles nécessitent, par ailleurs, l'échange d'informations de position. Enfin, le troisième type d'approche, dans laquelle nous nous inscrivons, est de créer des groupes stables entre terminaux. Dans [103], ceci est fait en calculant la distance moyenne aux autres pairs dans le graphe de routage, tandis que dans [38], l'algorithme cherche les boucles dans le graphe de routage.

En terme de surcharge réseau, notre algorithme est optimal, puisqu'il ne nécessite *aucun échange de message*. Le calcul effectué est une comparaison de deux listes, ce qui est moins coûteux que la recherche de boucle dans le graphe de routage, et aussi complexe que le calcul de distance moyenne. Notre algorithme passe donc à l'échelle grâce au fait que nous nous appuyons sur l'implémentation du protocole OLSR [22].

Pour évaluer la pertinence de notre algorithme, nous avons créé plusieurs scénarii de mobilité spécifiques (e.g. séparation d'un groupe, rencontre de deux groupes) et vérifié le comportement de notre algorithme sous ces conditions.

1.2.4 Réplication pro-active des données

Nous proposons un algorithme de réplication des données pro-actif et collaboratif au sein d'un groupe de mobilité stable qui vise à créer un nombre assez important de répliques pour éviter la perte de données, et prédit les accès des utilisateurs en se basant sur des

descriptions sémantiques.
Il existe plusieurs modèles de réplication. Parmi les plus simples : pas de réplication, réplication totale ou réplication à la demande. Ces solutions ne conviennent pas à notre système qui a des ressources limitées et peut présenter des partitions du réseau : avoir une seule copie fait courir le risque que la donnée soit indisponible si son créateur disparaît ; une réplication totale est coûteuse en terme d'espace de stockage, ainsi qu'en terme de maintenance de la cohérence, si tant est que les terminaux le permettent ; enfin, la réplication à la demande est, dans la plupart des systèmes, un bon compromis, mais dans notre cas la donnée risque d'être indisponible si aucun accès n'y est fait avant la disparition de son créateur.
Dans les algorithmes proposés pour les MANets, on trouve d'une part des algorithmes de réplication à la demande sous condition (par exemple, au moment d'une demande on ne crée une réplique que si la copie la plus proche est à plus de N sauts), et des algorithmes pro-actifs.
Les algorithmes de réplication pro-actifs existants supposent des informations sur les données, comme, par exemple, leur fréquence d'accès par tous les utilisateurs [34], leur fréquence de rafraîchissement [35] ou la corrélation entre les données [37]. Ces trois algorithmes sont en effet destinés à des réseaux de capteurs où les données sont mises à jour de manière statique et utilisées périodiquement par des machines pour faire des calculs, alors que notre proposition s'oriente plutôt vers des données mises à jour et utilisées de manière sporadique par des humains. De même dans les travaux [115] et [48], si les fréquences d'accès ne sont pas connues par avance, l'algorithme suppose qu'en échantillonnant les accès, on peut déterminer un jeu de fréquences, et par conséquent quelles sont les données les plus intéressantes à répliquer. Enfin, ils ne tirent pas parti d'informations sémantiques sur les données, qui ne sont par ailleurs pas toujours accessibles.
Nous avons donc proposé un modèle de réplication avec deux objectifs. Quand une donnée est créée, nous voulons la répliquer dans le réseau afin d'éviter qu'elle ne disparaisse si son créateur se déconnecte. Afin de diminuer le nombre de messages, ces copies sont par ailleurs placées sur les terminaux les plus susceptibles de les utiliser. Enfin, afin d'optimiser l'usage des ressources, la réplication est aussi collaborative.
En se basant sur les informations sémantiques et les groupes stables nous proposons un algorithme de réplication des données pro-actif et collaboratif :
- On utilise le calcul d'intérêt d'un utilisateur pour une donnée, pour rapatrier les données plus intéressantes.
- Au sein d'un groupe stable, les terminaux échangent leurs intérêts ainsi que des informations sur leurs ressources disponibles (batterie et espace mémoire) puis les agrègent afin de répliquer des données pour les terminaux plus faibles en ressources.
- s'il n'y a pas assez de répliques créées dans le système, on en place de nouvelles aléatoirement.

Bien entendu, comme l'espace-mémoire de chaque terminal est limité, il est aussi nécessaire d'appliquer une politique de remplacement de cache qui maintienne le nombre de répliques suffisamment élevé. Nous proposons donc un algorithme de remplacement de cache collaboratif et préventif, qui agit de manière à limiter le trafic réseau tout en maintenant le nombre de répliques nécessaires :
- Quand on doit libérer de l'espace-mémoire pour placer une nouvelle donnée, on élimine une réplique dont la donnée a déjà un nombre suffisamment élevé de répliques et la moins récemment utilisée.
- Quand on constate qu'une réplique locale génère une charge réseau inutile, on l'élimine.

Nous avons évalué ces algorithmes par simulation, en prenant pour critères le taux de succès (donnée répliquée localement), le nombre de données perdues ou inaccessibles, et la charge du réseau.

1.3 Plan de thèse

Notre thèse est articulée en 2 parties, la première introduisant le contexte du travail, les problématiques du partage de données et l'état de l'art dans le domaine, et la seconde présentant nos contributions.
Après l'introduction, et dans la première partie, nous détaillons les différentes problématiques liées au partage de données et les solutions proposées dans le contexte des MANets. Elle comporte 3 chapitres :

- Le chapitre 1 introduit le **contexte** dans lequel s'inscrit notre travail. Les réseaux mobiles ad hoc peuvent être caractérisés par divers paramètres, comme les capacités des terminaux (capteurs ou stations de travail), et la mobilité (piétons ou véhicules). Les contraintes et les applications utiles diffèrent selon le type de réseaux. Nous décrivons donc nos hypothèses de travail.
- Le partage de données est un problème complexe. Le chapitre 2 présente les différentes **problématiques** sous-jacentes.
- Dans le chapitre 3 nous présentons les **solutions qui ont été proposées** dans l'état de l'art à certaines des problématiques identifiées, à savoir la gestion de la mobilité, la réplication et le remplacement de cache, la cohérence de cache, et ce particulièrement dans le cadre des MANets. Nous voyons d'une part des algorithmes permettant de résoudre ces problèmes, et d'autre part des systèmes complets de partage de données sur MANets.

Nous présentons notre contribution dans la seconde partie, qui s'articule en 4 chapitres :

- Le chapitre 4 présente un algorithme de **création de groupe stable dans le temps** pour réseaux mobiles ad hoc de piétons. Cet algorithme s'appuie sur des informations obtenues de la couche de routage sous-jacente pour déterminer un voisinage stable dans le temps sans échange de messages.
- Le chapitre 5 décrit **un algorithme pro-actif de réplication de données à n%**. Cet algorithme fait usage des informations sémantiques pour prédire les accès utilisateurs et répliquer les données qui semblent intéressantes pour l'utilisateur. Par ailleurs, au sein d'un voisinage stable, la réplication de données se fait de manière collaborative afin d'héberger des données pour les utilisateurs possédant moins de ressources.
- Le chapitre 6 décrit **un algorithme de remplacement de cache**, qui maintient le nombre de données à un niveau assez élevé pour éviter les pertes, tout en limitant la charge du réseau
- Le chapitre 7 présente une proposition d'intégration de nos algorithmes au sein d'un **démonstrateur** d'un wiki pair à pair.

Enfin, nous introduisons dans la conclusion les **perpectives** de recherche.

Première partie

Etat de l'art du domaine

Chapitre 2

Contexte de nos travaux de recherche

Dans ce chapitre nous présentons ce qu'est un MANet [1], et quelle en est l'utilité, avant de préciser le contexte dans lequel nous plaçons notre travail.

2.1 MANet : une définition

Un réseau mobile ad hoc, appelé aussi MANet, est un réseau autoconfiguré qui se constitue spontanément entre des terminaux sans fil mobiles (type PDA, Pocket PC, laptop) se déplaçant librement, sans utilisation d'une infrastructure préalablement établie [23].
Les réseaux ad hoc multisauts possèdent de nombreuses caractéristiques les différenciant des réseaux filaires classiques et des réseaux sans fil à un saut plus anciens :

Routage multisauts. Les MANets présentent la particularité de pouvoir router les messages en l'absence d'infrastructure : deux terminaux pourront donc communiquer sans être à portée l'un de l'autre s'il existe une chaîne de terminaux qui les connectent. La tâche de routage, effectuée dans les réseaux filaires par des terminaux dédiés, incombe ici à tous les terminaux.

1. Mobile Ad hoc NETwork

Gestion du réseau distribuée. En l'absence d'un élément central dans le réseau, sa gestion (sécurité, contrôle de congestion) doit être effectuée de concert et de façon distribuée par les terminaux.

Topologie dynamique. Du fait de la mobilité et de la "volatilité" des terminaux, la topologie du réseau est susceptible d'évoluer en permanence.

Capacité des liens variables. La capacité des liens varie au cours du temps suivant de nombreux critères : puissance d'émission du signal, distance entre les noeuds, bruit, interférences, obstacles. De manière générale, la bande passante est moins "large" que celle d'un réseau filaire.

Terminaux légers. Les terminaux mobiles sont limités en ressources matérielles : ils possèdent une puissance de calcul et un espace mémoire moindre que les stations de travail et les serveurs, et travaillent sur batterie.

2.2 MANet : quel intérêt ?

La couverture 3G étant de plus en plus étendue, elle permet aux terminaux sans fil de se connecter à une infrastructure. Dans de telles circonstances, en quoi un MANet est il utile ?

2.2.1 Absence d'infrastructure

Les travaux les plus anciens sur les MANets se sont fait sur deux scénarii.
Le premier est un scénario militaire. Lors d'un déploiement sur un territoire où il n'y a pas d'infrastructure de communication à disposition, ou dans le cas où ces infrastructures ne sont pas sécurisées, les communications passent par satellite. Cependant, les communications via satellite ont de grands délais de transmission, avec des équipements encombrants et spécialisés (généralement des téléphones). Les MANets sont donc utilisés pour communiquer au sein d'une équipe sur le terrain, avec souvent un seul terminal possédant un accès satellite au sein du groupe.
On trouve de nombreuses références à des travaux de recherche sur les MANets tactiques, notamment ceux menés ou financés par le NRL (Naval Research Laboratory) [66] [19]. Cependant, de par la nature des travaux, il y a peu de rapports d'expériences réelles.
Le second scénario est celui d'une catastrophe naturelle suite à laquelle les infrastructures de communications sont tombées. Les infrastructures de communications cellulaires sont rapidement rétablies, mais comme les chances de retrouver des survivants diminuent très rapidement, les secours doivent donc se faire le plus tôt possible. Dans un tel contexte, les réseaux spontanés sont utilisés pour coordonner des équipes de secours.
Aujourd'hui, les MANets sont étendus à des applications non critiques en l'absence d'infrastructure, comme par exemple le travail nomade.

2.2.2 Eviter d'utiliser l'infrastructure

L'entité propriétaire d'une infrastructure en fait souvent payer l'accès.
Prenons par exemple le cas d'une équipe de chercheurs partie à l'étranger dans le cadre d'un projet. Tous les membres de l'équipe sont logés dans le même hôtel où chaque chambre est équipée d'une connexion Internet, disponible à l'heure moyennant rémunération. Au sein du groupe des fichiers sont édités par plusieurs membres de l'équipe. Dans l'absolu,

un accès à internet n'est pas nécessaire, mais passer par un réseau permet d'envoyer les fichiers à tous sans avoir à échanger une clé USB et en suivant les évolutions du fichier édité en temps réel.
En établissant un MANet, l'édition peut se faire en ligne, par exemple en utilisant un éditeur collaboratif, sans passer par l'infrastructure de l'hôtel.

2.2.3 Extension de l'infrastructure existante

En présence d'un accès limité à Internet, on peut utiliser un MANet en extension de l'infrastructure existante.
Lors d'une réunion, un seul ordinateur peut avoir accès à Internet, via un câble ethernet. Les participants se passent donc le câble régulièrement afin d'accéder à leur boîte mail. Une autre alternative serait que les terminaux se constituent en réseau ad hoc. Le terminal ayant accès à internet devient alors relai et permet à chacun de consulter son mail en même temps.

2.3 Différents MANets, différentes contraintes

Comme nous l'avons vu dans la section précédente, les MANets répondent à un besoin réel, malgré l'étendue de l'infrastructure de communication actuelle. Cependant, un réseau mobile ad hoc peut être constitué dans différentes conditions.
Les caractéristiques variables sont nombreuses. Nous les détaillons ci-dessous, avant de présenter 3 implémentations concrètes de MANets avec différentes contraintes, menées dans le cadre de projets de recherche.

2.3.1 Contraintes

2.3.1.1 Taille du réseau

Le problème de taille du réseau se pose à différents niveaux :
- Le nombre de terminaux impliqués est-il connu par avance ? Dans ce cas, les connait-on tous ? Les terminaux peuvent-ils rentrer dynamiquement dans le réseau ?
- Quel est le nombre maximum souhaité de terminaux possibles ?

Si des terminaux peuvent entrer dynamiquement dans le réseau, on a alors un problème d'attribution d'identifiants uniques, comme une adresse IP.
La taille du réseau influe sur le choix des protocoles de routage et de communications. Pour les réseaux de grande taille, des algorithmes de routage hiérarchique ont été proposés afin de limiter la surcharge créée par la découverte des routes. Par ailleurs, dans un réseau de petite taille, on peut utiliser des techniques d'inondation qui sont trop coûteuses dans un grand réseau.

2.3.1.2 Mobilité

La mobilité des terminaux recoupe plusieurs problèmes :
- Vitesse : quelle est la vitesse de déplacement des terminaux ?
- Schéma de déplacement : comment les terminaux se déplacent-ils les uns par rapport aux autres ?
- Volatilité : quelle est la probabilité qu'un terminal disparaisse du réseau ?

La vitesse des terminaux peut varier de la vitesse d'un piéton à celle d'une voiture. Une vitesse élevée influence les communications au niveau de la couche physique. Par ailleurs, deux terminaux se croisant à grande vitesse ne peuvent échanger que peu d'information.
Le schéma de déplacement peut prendre différentes formes : groupe d'utilisateurs se déplaçant et travaillant ensemble ; utilisateurs se croisant par hasard et échangeant des informations parce qu'ils en ont l'opportunité ; ou encore voitures sur une autoroute où toutes se déplacent à vitesse égale dans la même direction, sans que leurs utilisateurs se soient concertés. Le type d'application dans ces 3 cas ne sera, bien entendu, pas le même.
La volatilité des terminaux reflète la stabilité du réseau. On peut avoir, par exemple, un réseau de capteurs qui s'endorment périodiquement afin de sauvegarder leurs batteries, mais dans lequel on peut cependant compter sur le fait qu'ils se réveilleront à nouveau plus tard, pour effectuer une communication. Sur une autoroute au contraire, une voiture avec laquelle une communication était établie et qui emprunte une bretelle de sortie ne sera probablement jamais recroisée.

2.3.1.3 Autonomie et stockage

Un terminal étant mobile, il travaille a priori sur batterie. L'autonomie d'un terminal définit le temps de travail, que nous appelons ici session, possible pour le terminal.
Un terminal peut avoir une autonomie illimitée, comme par exemple pour une voiture, limitée mais avec possibilité de recharge (session limitée), comme, par exemple, pour un PDA ou ne plus fonctionner une fois que les batteries sont vidées (durée de vie limitée), dans le cas de certains capteurs.
Les sources principales de consommation de la batterie sur un terminal sont l'écran, les mémoires de masse nécessitant une partie mécanique (disque dur) et la carte réseau. De plus en plus des terminaux portables de nos jours utilisent de la mémoire SSD, qui, ne comportant pas de pièces mécaniques, consomment moins. La fréquence des accès disques et l'affichage sont gérés de manière optimisée par le système. Ce ne donc pas des points sur lesquels nous pouvons jouer.
D'après [57], les communications peuvent consommer jusqu'à 50% de la batterie. Ces chiffres datent de 10 ans, et les techniques de gestion de la carte réseau se sont améliorées depuis (passage en mode sleep plutôt que idle). Cependant, les utilisateurs de smartphone peuvent facilement constater le coût de l'utilisation du réseau. Apple annonce, par exemple, une autonomie de 40h de vidéo pour son iphone 3GS, utilisant pourtant le processeur et l'écran, contre 10h si on surfe en wifi.
Par ailleurs, les capacités de stockage d'un terminal mobile sont souvent plus limitées que celles d'un ordinateur de bureau classique. Dans les réseaux de capteurs, des algorithmes d'agrégation des données sont parfois mis en oeuvre afin de sauvegarder de l'espace mémoire.
Enfin, les capacités de calcul d'un terminal mobile sont aussi plus limitées, ce qui peut poser problème par exemple quand on cherche à mettre en place des communications sécurisées, comme expliqué dans la section suivante.
A titre d'exemple, le tableau 2.1 liste les capacités de différents types de terminaux.

2.3.1.4 Sécurité

Le besoin de sécurité a deux aspects :
– la sécurisation des communications : empêcher un tiers de capter et comprendre un échange, voire d'empêcher les communications d'avoir lieu,

TABLE 2.1 – Quelques plateformes

Plate-forme	CPU	RAM	Stockage secondaire	Autonomie	Autres
Capteur Iris Mote	16MHz	8ko	512ko	2 batteries AA	GPS, différents capteurs
OLPC XO-1	433MHz	256 Mio	1Gi	5h, 10 à 12h selon sources	Recharge manuelle
Smartphone N900	600MHz	256Mo	32Go	5h en conversation	GPS
Subnotebook EEE pc 1001 HA	533MHz à 1,6GHz	1Go	160Go	8h30	
Portable macbook 13 pouces	2,4GHz	4G	250Go	10h	
Ordinateur de bureau imac	4*2,8Ghz	4 à 16Go	500Go à 1To		
Serveur Bell Poweredge M910	8 * 2,4GHz	1Go à 512Go	4To à 8To		

– l'authentification et l'identification des pairs : identifier un pair et prouver son identité. Un MANet étant un réseau spontané et sans fil, n'importe quel terminal peut potentiellement s'y joindre et s'en prétendre membre. Il est donc nécessaire d'avoir un moyen d'authentifier les pairs. Dans un réseau classique, cela se fait généralement en utilisant les services d'un tiers de confiance [56], service sur lequel on ne peut pas compter dans un MANet. Cela rend donc le réseau plus sensible aux attaques de type déni de service, un brouilleur créant des interférences électromagnétiques pouvant d'ailleurs empêcher toute communication.
Par ailleurs, même sans essayer de se faire passer pour un membre du MANet, les communications sans fil peuvent être écoutées. Afin de protéger les données, celles ci doivent donc être chiffrées avec un secret partagé. Le chiffrement des données est problématique pour des terminaux avec des processeurs peu puissants car le temps de chiffrement et déchiffrement est proportionnel à la garantie de sécurité offerte. Une meilleure sécurité ralentit donc le travail. Par ailleurs, le calcul est consommateur de batterie.
Ce besoin de sécurité varie selon le type d'application visé. Dans le cas d'un usage militaire, le chiffrement des communications et la détection des intrus sont cruciaux, alors que dans le cas d'un usage civil ces contraintes peuvent être relâchées.

2.3.1.5 Application

Selon le type d'application visé, les terminaux vont générer des profils de trafic différents. Les réseaux de capteurs envoyant périodiquement un relevé de mesure ne posent pas les même contraintes qu'un réseau d'utilisateurs humains partageant des documents en édition. Par ailleurs, certaines applications ne font que des communications locales (1 ou 2 sauts), alors que d'autres ont besoin de joindre un terminal précis appartenant au réseau. Le premier cas ne nécessite pas forcément un algorithme de routage, alors que le second nécessite une découverte du réseau.

Nous allons présenter trois projets de recherche utilisant les MANets pour différents usages, et voir à quel point ces différentes caractéristiques peuvent varier selon le domaine d'application.

2.3.2 Exemple : réseau véhiculaire

Les VANets, ou Vehicular Ad Hoc Networks, sont des réseaux constitués de véhicules.
Le projet StreetSmart [26] propose de constituer les voitures sur une autoroute en VANet, afin de véhiculer des informations sur l'état du trafic. Une forte densité de véhicules avançant à faible vitesse permet de détecter une congestion. Cette information remonte ensuite le flot de véhicules et peut être utilisée par les systèmes de guidage GPS pour optimiser les trajets.
Le nombre de terminaux impliqués dans ces communications est élevé, puisque le réseau couvre l'ensemble du segment d'autoroute. Cependant, l'information ne circule que dans un sens, à l'inverse du flot de voitures, et les communications ne se font pas entre entités spécifiques. Il n'y a donc pas réellement de routage, et on doit simplement connaître les positions relatives des terminaux à un saut afin de décider comment les messages doivent être diffusés. Enfin, aucune information n'a à être stockée après avoir été transmise.
Il n'y a pas de problème d'autonomie ou de stockage. La sécurité n'est pas abordée.

2.3.3 Exemple : réseau de capteurs mobiles, application scientifique

Dans le projet ZebraNet [50], une équipe de chercheurs veut étudier le comportement d'un troupeau de zèbres. Chaque zèbre est équipé d'un collier capteur GPS faisant régulièrement un enregistrement de position. Périodiquement, un des chercheurs passe à proximité du troupeau afin de récupérer les informations. C'est le seul moment dans l'absolu où les terminaux ont à communiquer, ce qui utilise leur batterie, mais des communications sont tout de même maintenues en dehors des périodes où l'information est recueillie.
La création de nouvelles données est périodique. Si on connaît la fréquence de passage du chercheur, on peut donc prédire les capacités de stockage nécessaires. Cependant, la capacité de stockage d'un capteur étant faible, les données recueillies sont agrégées afin de ne pas occuper tout l'espace-mémoire. La durée de vie d'un capteur est par ailleurs limitée par ses batteries. Quand les batteries d'un collier sont vides, il faut capturer à nouveau l'animal portant ce collier pour les remplacer ou changer le capteur. Les données sont donc répliquées sur différents capteurs, afin qu'aucune ne soit perdue.

Figure 2.1 – Un zèbre équipé du collier GPS

Le nombre de terminaux ici est de l'ordre de la centaine. Les terminaux se déplacent en groupe, et les communications sont faites de manière opportuniste pour sauvegarder les informations. L'autonomie des terminaux étant faible, ce système cherche donc à la maximiser.

2.3.4 Exemple : réseau à vitesse humaine, jeu collaboratif

Dans le projet Transhumance [30], un jeu de piste est organisé sur la Butte-aux -Cailles. Les utilisateurs sont organisés en groupes, chaque membre ayant un PDA. Au début d'une session de jeu, un questionnaire est distribué sur les terminaux des joueurs. La première équipe à trouver tous les éléments et revenir à l'arbitre a gagné. A défaut, l'équipe ayant trouvé le plus d'éléments à la fin du temps imparti gagne.
La durée d'une session de jeu pour un groupe est limitée par la batterie de ses terminaux. Différents utilitaires sont proposés aux joueurs, comme un chat, un vote, ou un partage de fichier. Ici, les joueurs sont organisés en groupes collaborant. Même s'ils peuvent se séparer, par exemple pour que chacun aille chercher un indice individuellement, ils se retrouvent par la suite et leurs feuilles de réponses doivent être synchronisées.
Les communications peuvent être chiffrées, pour que les membres des équipes adverses ne puissent pas connaître les réponses, cependant sécuriser les données n'est pas ici critique.

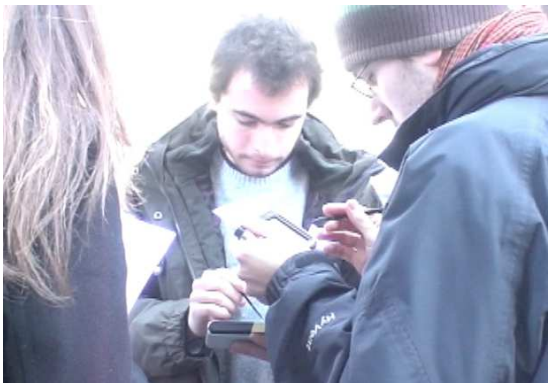


FIGURE 2.2 – Une session de jeu sur la Butte-aux-Cailles

2.3.5 Une solution unique est elle possible ?

Comme on peut le voir dans ces différents exemples, un MANet est un réseau de terminaux mobiles, qui peut prendre de nombreuses formes. Plutôt que d'essayer d'offrir une solution globale non satisfaisante, nous préférons travailler à une solution adaptée à un contexte.
Dans la section suivante nous présentons donc le cadre de nos travaux.

2.4 Nos hypothèses de travail

Dans cette thèse nous voulons proposer une solution facilitant le partage de données pour applications collaboratives. Nous détaillons donc, dans cette section, nos hypothèses de travail.

2.4.1 Mobilité, volatilité

Notre système est destiné à un réseau de piétons, avec des utilisateurs se déplaçant en groupes.
Le nombre de terminaux n'est pas connu par avance, mais ne dépasse pas la centaine. Des terminaux peuvent se joindre spontanément à une session de travail, mais afin de pouvoir

participer à la session de travail collaboratif, ils devront avoir les bonnes applications installées.
Les utilisateurs essayant de travailler en groupes, nous attendons des phases de stabilité au niveau de la présence. Cependant, des utilisateurs peuvent entrer et sortir du groupe, et différents groupes peuvent se croiser, voir fusionner, auquel cas des informations seront échangées.
Par ailleurs, des utilisateurs peuvent disparaître, par exemple à cause des problèmes de batteries susmentionnés.

2.4.2 Autonomie

Nous visons des terminaux de type portable ou PDA, travaillant en mode nomade. Il n'y a donc pas de problème critique de stockage, contrairement par exemple à un réseaux de capteur. Par contre, la batterie limite le temps d'une session de travail. Cependant, les batteries peuvent être rechargées et le terminal peut revenir dans le réseau. La durée d'une session de travail est donc limitée, mais la disparition d'un terminal n'est pas nécessairement définitive (faute transitoire).

2.4.3 Applications visées, trafic

Nous voulons permettre le travail nomade.
Dans leurs bureaux, les utilisateurs utilisent habituellement une application centralisée classique, comme un système de partage de fichiers, pour collaborer. Quand ils passent en mode nomade, une version distribuée de l'application est déployée sur les terminaux utilisateurs. Cette application distribuée reproduit les services offerts par sa version centralisée, avec éventuellement des restrictions (par exemple, les données de taille trop grandes, comme des vidéos, ne sont pas disponibles durant la phase de mobilité). Durant une période de nomadisme, qui peut durer plusieurs sessions de travail, les utilisateurs peuvent donc toujours travailler collaborativement. Quand les utilisateurs rejoignent leurs bureaux à l'issue de leur voyage, les modifications effectuées durant la période nomade sont fusionnées aux données de l'application centralisée.
Comme les données sont éditées par des utilisateurs humains, nous avons les contraintes suivantes :
– Les données sont modifiées.
– Les mises à jour des données sont sporadiques.
– Les mises à jour peuvent être effectuées depuis tous les terminaux.
– L'accès aux données est apériodique.
Il est important de préciser ces contraintes, car des solutions ont déjà été proposées pour des systèmes de capteurs actualisant et utilisant périodiquement des données [34], ou les systèmes avec un serveur de fichier, seul habilité à modifier la donnée [62].
On ne peut donc pas prévoir à l'avance des schémas de trafic précis.
Enfin, nous ne souhaitons pas mettre en œuvre des applications de type flux multimédia qui imposent des contraintes temps réel au niveau des communications.

2.4.4 Sécurité

Ces applications n'étant pas destinées à l'échange de données critiques, nous ne proposons pas ici de système de sécurisation des données et des échanges par authentification et

chiffrement. Cependant, un système de sécurité peut être ajouté à notre proposition par la suite, sans que nous ayons à la modifier.

2.5 Conclusion

Nous avons vu dans cette section dans quelles circonstances un MANet peut s'avérer utile : les premières applications étaient liées aux situations d'urgence, par exemple suite à une catastrophe, ou dans une situation de militaire, mais on peut aussi les utiliser pour un usage civil quand les infrastructures sont inexistantes, ou compliquées d'accès.
Nous avons présenté les différentes contraintes liés aux MANets : taille du réseau, mobilité et capacité des terminaux (autonomie et stockage).
Nous avons ensuite vu trois projets de recherche utilisant les MANets dans des contextes très distincts avant de présenter le contexte de nos travaux : des MANets d'une centaine de terminaux mobiles opérés par des utilisateurs humains, qui utilisent des applications collaboratives.

Chapitre 3

Problématiques liées au partage de données dans les MANets

Nous voulons faciliter le partage de données éditables sur un réseau mobile ad hoc pour des applications collaboratives. Les MANets sont des réseaux à la topologie changeante et où les terminaux peuvent disparaître, et dans lesquels on ne peut pas faire l'hypothèse qu'il existe un terminal fiable susceptible d'être serveur de données.
Le schéma 3.1 résume les différentes contraintes auxquelles nous sommes soumis, les problématiques en découlant et celles auxquelles nous faisons une contribution.
Les contraintes auxquelles notre système est soumis ont 3 origines.

– Des contraintes fonctionnelles, définissant les fonctionnalités attendues de la part de l'utilisateur :
  1. Consultation de données.
  2. Edition de données.
  3. Recherche de données.
– Des contraintes liées à l'environnement cible :
  1. Mobilité des terminaux.
  2. Absence de serveur.
  3. Ressources limitées.

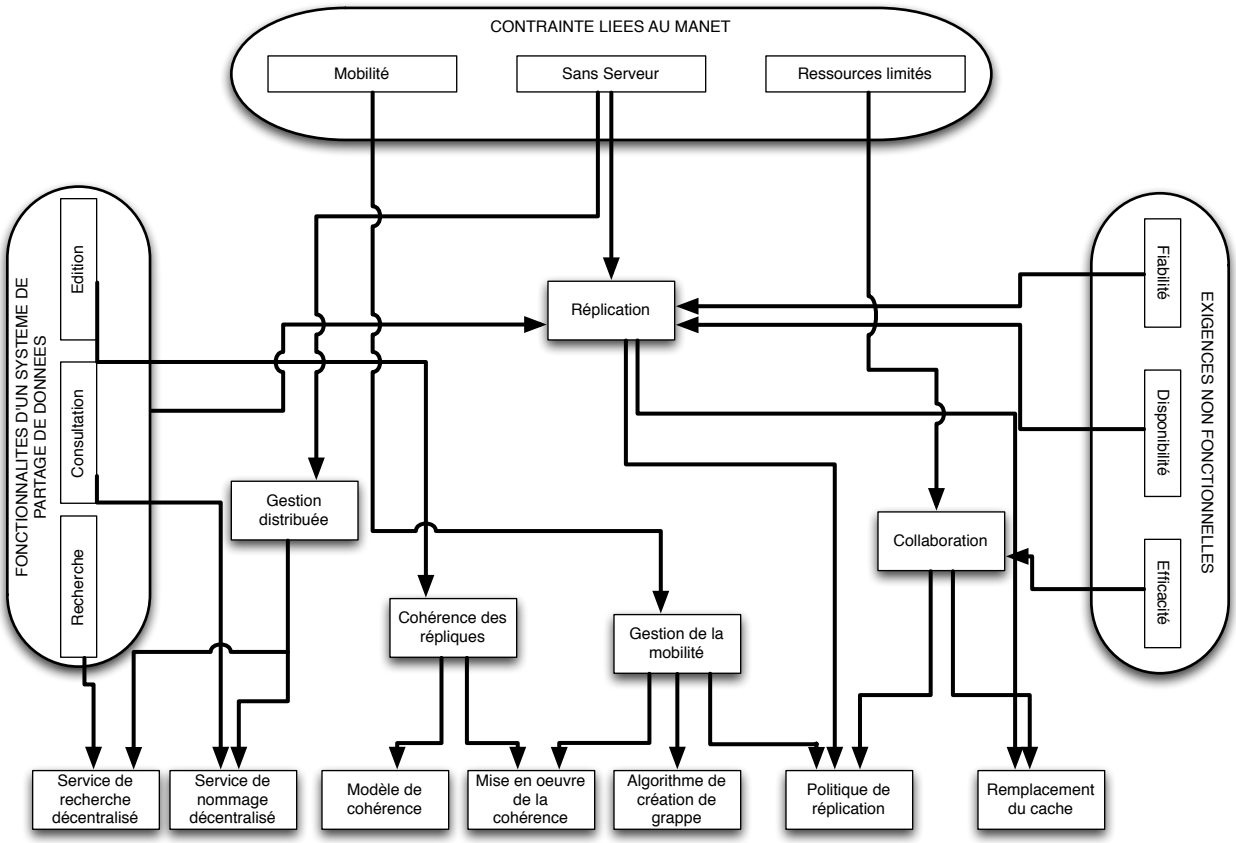


Figure 3.1 – Problématiques du partage de données sur réseaux mobile ad hoc

– Et enfin, des contraintes non fonctionnelles définissant les performances attendues de notre système
  1. La disponibilité : le service de partage de données doit être prêt à être utilisé quand un utilisateur le demande. L'accès à une donnée doit être effectué en temps borné.
  2. La fiabilité : le service de partage de données doit être disponible en continu. Les données ne doivent pas être perdues.
  3. L'efficacité : le service de partage de données doit faire un usage efficace des ressources à sa disposition. Les ressources de notre système sont l'espace de stockage et le réseau : nous devons donc d'une part limiter l'usage du réseau, qui impacte directement la durée de vie d'une session de travail, et, d'autre part, concevoir un système capable de travailler dans l'espace-mémoire restreint à sa disposition.

Afin d'améliorer la *disponibilité* et la *fiabilité*, les données vont être *répliquées* dans le réseau. Le choix de répliquer les données soulève plusieurs problèmes.
Tout d'abord, il faut choisir une *politique de réplication* [79], c'est à dire décider du moment où ces répliques sont créées, de la quantité de répliques nécessaires et des terminaux sur lesquelles elles sont placées.
Pour faire un usage *efficace* des *ressources limitées*, la *collaboration* entre terminaux est préférable : nous voulons donc mettre en place des algorithmes collaboratifs, où, par exemple, un terminal répliquera une donnée pour son voisin. Cependant, comme les terminaux sont *mobiles*, cette collaboration ne doit pas se faire avec n'importe quels terminaux, mais

seulement avec ceux qui restent assez longtemps en présence pour qu'il soit intéressant de collaborer. Nous avons donc besoin de *gérer la mobilité*, sous la forme d'un *algorithme de création de grappe (clustering)*.
Comme il est possible qu'il existe plusieurs copies de la même donnée et que ces copies puissent être modifiées, ces copies doivent être maintenues *cohérentes* [61]. On doit donc définir un *modèle de cohérence de données* qui définit un ordre légal d'accès (lecture, écriture) par les terminaux. Ce modèle doit ensuite être *mis en œuvre*, par exemple en centralisant les modifications sur une copie primaire ou en mettant en place un mécanisme d'exclusion mutuelle distribué.
Dans le cas où l'on voudra éliminer une réplique afin de libérer de l'espace, ou afin de faire diminuer le nombre de répliques car elles génèrent du trafic (pour maintenir la cohérence), il faut définir une politique de *remplacement de cache*. Celle permet d'une part de choisir, au sein d'un cache, quelles répliques éliminer, et d'autre part de choisir, parmi les hôtes d'une donnée, lesquels doivent éliminer leurs répliques, dans le cadre d'un algorithme *collaboratif*.
Enfin, comme un MANet est par nature distribué, il faut mettre en place un *système de nommage* permettant de localiser une copie d'une donnée. Par ailleurs, nous voulons aussi permettre de *rechercher* des données, non pas cette fois pour localiser une réplique par son nom, mais pour trouver une donnée à partir de son contenu.
Nous présentons ces problématiques plus en détail dans la suite de ce chapitre. Dans cette thèse nous ne nous attaquons pas à tous ces problèmes. Pour les problématiques auxquelles nous apportons une réponse dans ces travaux, à savoir la gestion de la mobilité, la réplication des données, la gestion du cache, et le maintien de la cohérence, nous présentons une étude plus approfondie des solutions spécifiques aux MANets dans le chapitre suivant.

3.1 Réplication

Dans le cadre d'un système distribué, un site est susceptible d'accéder à des données stockées sur un site distant. Comme nous l'avons dit plus haut, nous définissons la *disponibilité* d'une donnée comme la capacité pour un terminal d'accéder à cette donnée en un temps borné. La réplication de données vise à augmenter la disponibilité en créant plusieurs copies (répliques), placées sur différents sites.
La création de répliques a un impact sur le trafic réseau :
– créer une réplique permet de ne pas avoir à transmettre la donnée à chaque lecture.
– cependant, si cette donnée est modifiable, la création d'une réplique engendre un trafic nécessaire pour maintenir la cohérence entre les différentes copies de la donnée.

Par ailleurs, une réplique utilise de l'espace-mémoire. Selon les caractéristiques du système, tel le coût de l'envoi d'un message, ou l'espace-mémoire disponible, on cherche donc à atteindre un juste milieu entre trafic engendré d'une part et rapidité d'accès et tolérance aux fautes d'autre part.
Dans un système sujet aux pannes, on utilise aussi la réplication de données pour prévenir la disparition des données.
La réplication est donc utilisée dans la plupart des types de systèmes distribués (mémoire partagée distribuée, cache, système de fichiers distribué, base de données) afin d'en améliorer les performances et d'en augmenter la tolérance aux fautes.
L'espace de stockage des sites peut être limité. Il est donc nécessaire de répliquer en priorité les données utiles sur le site. Par ailleurs, quand on cherche à répliquer une donnée, mais que le cache est plein, un algorithme de remplacement détermine quelle réplique éliminer si cela nécessaire.

Nous allons maintenant décrire différents schémas de réplication et de remplacement.

3.1.1 Schéma de réplication

Un schéma de réplication décrit sur quels critères une donnée doit être répliquée. Les schémas classiques sont les suivants :

- Copie fixe : il n'existe qu'une seule copie de la donnée dans tout le réseau. Elle est placée sur un terminal déterminé. Cette politique permet d'éviter le surcoût nécessaire pour maintenir les différentes répliques cohérentes. Cependant, elle n'améliore en rien l'accessibilité ou la tolérance aux pannes. Cette technique est utilisée dans les systèmes de fichiers réseaux, ou les bases de données.
- Migration : il n'existe qu'une copie de la donnée dans le réseau. Quand un terminal doit accéder à la copie, il en fait la demande. Quand un terminal transmet la donnée, il détruit sa copie.
- Réplication totale *(mirroring)* : les données sont répliquées sur tous les terminaux. Cette technique est coûteuse en espace-mémoire, et si les données sont modifiables, coûteuse en messages. Cependant, une donnée ne peut pas disparaître et est disponible immédiatement pour chaque terminal. Elle est utilisée dans certains systèmes de gestion de base de données comme SQL Server ou Oracle
- Réplication à la demande : une réplique d'une donnée est créée sur un terminal la première fois que celui ci tente d'y accéder. Dans les systèmes filaires, c'est la politique la plus couramment utilisée.

Il existe d'autres techniques de réplication à la demande partielle prenant en compte des paramètres plus complexes. La réplication peut, par exemple, être basée sur les fréquences d'accès : un terminal réplique une donnée s'il l'utilise fréquemment. Le terminal peut aussi prendre en compte le comportement de ses voisins dans sa décision de répliquer.
Il existe aussi des schémas de réplication proactifs utilisant des informations sémantiques sur les données et sur l'utilisateur [8] [28]. Elles visent à répliquer la donnée avant que l'utilisateur n'y accède en prédisant ses accès futurs.
Par ailleurs, dans les MANets, le problème de réplication est souvent abordé de manière collaborative : un groupe de terminaux proches décide collaborativement des données qui seront répliquées, afin de maximiser la diversité des données dans le groupe.

3.1.2 Remplacement de cache

L'espace de stockage d'un terminal étant limité, il arrive que pour répliquer localement une donnée, il soit nécessaire d'éliminer une réplique existante. Les algorithmes de remplacement visent à déterminer quelles sont les données à éliminer.
Les algorithmes utilisés sont, pour certains, identiques à ceux proposés pour la gestion de caches physiques, comme décrit dans [99]. Ils sont donc simples à mettre en œuvre et nécessitent peu de calcul.
Tous ces algorithmes ne sont pas nécessairement adaptés aux caches logiciels distribués. Dans les systèmes distribués, les algorithmes issus de la gestion de cache-mémoire principalement utilisés sont :

- LRU *(least recently used)* : on élimine la donnée qui n'a pas été utilisée depuis le plus longtemps.
- LFU *(least frequently used)* : on élimine la donnée la moins fréquemment utilisée.

Cependant, d'autres critères sont pris en compte dans des caches logiciels distribués :

– la difficulté d'accès à une donnée : on préfère ne pas éliminer une donnée difficile à obtenir, si, par exemple la réplique, la plus proche est éloignée.
– la taille : quand on élimine une donnée de petite taille, il est probable qu'il faille éliminer rapidement une autre donnée, alors que quand on élimine une donnée de taille importante, on libère plus d'espace.

Dans un système de réplication collaboratif où la création d'une réplique prend en compte l'intérêt non pas du seul utilisateur, mais du groupe, le remplacement de cache doit lui aussi être collaboratif.

Nous avons vu sur quels critères une donnée pouvait être mise en cache, ou éliminée du cache. Il est maintenant nécessaire de définir dans quelle mesure les différentes répliques sont maintenues identiques. C'est le problème de la cohérence qui est examiné dans la partie suivante.

3.2 Cohérence

Dans le cadre d'un système de partage de données distribué, on réplique les données afin d'améliorer leur disponibilité. Il est alors nécessaire de maintenir cohérentes les différentes copies d'une donnée. Un modèle de cohérence de mémoire définit quelles sont les garanties sur l'ordre des accès aux données qu'un programmeur obtient du système.
Il est à noter que les anglophones distinguent les termes *consistency* et *coherency* :

Consistency décrit l'état de l'intégralité de la mémoire.

Coherency décrit l'état d'une unité de granularité de la mémoire.

Cette distinction n'est pas faite en français.
On notera aussi que de nombreux travaux ne différencient pas réplication et cohérence. Dans [32], par exemple, Gray parle de *Lazy Replication* et *Eager Replication* pour distinguer deux modes de mises à jour des données distribuées. Dans les systèmes ne faisant pas cette distinction, les données sont généralement totalement répliquées ou répliquées à la demande.
Nous allons voir ici les différents modèles de cohérence existant dans la littérature [75]. Nous verrons ensuite quels problèmes posent ces modèles, et les algorithmes proposés pour les résoudre.

3.2.1 Modèles

Il existe plusieurs types de modèles de cohérence, pour la plupart décrits dans [75].
Nous allons d'abord voir des modèles de cohérence *(consistency)* définis pour des mémoires partagées distribuées. Nous verrons ensuite les modèles de cohérence *(coherency)*, définis pour des systèmes à plus gros grain.
Dans un modèle de cohérence sans synchronisation, la mémoire est maintenue cohérente par le système sans intervention du programmeur. Dans un modèle de cohérence avec synchronisation, les opérations de synchronisation des vues de l'espace partagé sont explicitement appelées par le programmeur.

3.2.1.1 Modèles sans synchronisation

Il existe différents modèles de cohérence sans synchronisation qui définissent un ordre valide sur les événements dans un système.

Dans un modèle de *cohérence atomique*, ou cohérence stricte, tous les processeurs voient les événements dans leur ordre d'apparition. Les événements sont donc strictement ordonnés selon l'ordre temporel.
Dans un modèle de *cohérence séquentielle*, tous les processeurs voient les événements dans le même ordre [72]. Les événements sont strictement ordonnés, mais pas nécessairement selon l'ordre temporel. L'ordre d'exécution sur un processeur donné est respecté.
Dans un modèle de *cohérence causale*, tous les processeurs voient les événements causalement liés dans le même ordre [73]. Les événements indépendants peuvent être vus dans des ordres différents par deux processeurs. Les événements sont donc ordonnés selon un ordre causal partiel.
Dans un modèle de *cohérence processeur* (appelé aussi PRAM ou FIFO), tous les événements générés par un processeur sont vus dans le même ordre par tous les processeurs.
Dans un modèle de *cohérence* Δ, on garantit que si une donnée est modifiée à l'instant t, tous les processeurs verront cette modification à l'instant $t + \Delta$.
Dans un modèle de *cohérence à terme*, on garantit uniquement que les opérations d'écriture seront à terme propagées à tous les processeurs.

Hormis la cohérence à terme et Δ, les autres modèles de cohérence demandent la mise en œuvre par le système d'une synchronisation lourde qui n'est pas visible par l'utilisateur mais peuvent offrir plus que ce qui est requis par l'utilisateur.
Des modèles avec une synchronisation explicitée par le développeur, qui ne devraient donc utiliser que le nombre minimum de points de synchronisation nécessaires, ont donc été proposés.

3.2.1.2 Modèles avec synchronisation

Dans un modèle de cohérence avec synchronisation, c'est au programmeur de définir quand les différentes copies d'une donnée peuvent diverger, et quand elles doivent être réconciliées.
Un modèle de *cohérence faible* [75] offre les garanties suivantes :

1. les accès aux variables de synchronisation sont faits sous un modèle de cohérence séquentielle,
2. l'accès aux variables de synchronisation ne se termine que quand toutes les opérations de lecture et écriture sont terminées,
3. les opérations de lecture et écriture sont autorisées seulement si tous les accès aux variables de synchronisation ont été terminés.

Dans un modèle de *cohérence à la libération*, comme proposé dans le système Treadmarks [51], il existe deux opérations de synchronisation : *acquire* et *release*. Quand un processeur effectue une opération de relâchement, les modifications qu'il a effectuées sont visibles par tous les processeurs effectuant une opération d'acquisiton par la suite.
Dans un modèle de *cohérence à l'entrée*, comme proposé dans le système Midway, à chaque variable est associé un verrou [12]. Une acquisition sur un verrou ne permet donc d'obtenir que les modifications effectuées sur la donnée à laquelle ce verrou est attaché.

3.2.1.3 Cohérence forte/faible

Dans le domaine des systèmes de fichiers distribués, le vocabulaire utilisé pour parler de cohérence est différent. Il s'agit ici, en effet, de modèle de cohérence liée non plus à l'ensemble de la mémoire, mais à une donnée.

Dans un modèle de *cohérence forte*, ou *pessimiste*, tout accès à la donnée retourne la dernière version de cette donnée. Dans un modèle de *cohérence faible*, ou *optimiste*, il est toléré qu'un accès à une donnée ne retourne pas la dernière version. Cependant, on doit garantir que les modifications seront éventuellement intégrées par tous les processeurs. Par la suite, nous emploierons les termes optimistes et pessimistes pour désigner ces deux modèles.
Dans un modèle de *cohérence read-your-write*, tel que proposé par Bayou, on garantit qu'un accès par un pair à une donnée retourne une version de la donnée au moins aussi récente que la dernière version modifiée par ce pair [100].
Dans un modèle de *cohérence close-to-open*, tel que proposé dans des systèmes de fichiers distribués, comme NFS [5] et Pastis [16], la synchronisation se fait à l'ouverture et la fermeture d'un fichier. Quand un fichier est ouvert, la dernière version de ce fichier est mise en cache. Les opérations de lecture et d'écriture se font ensuite sur la copie locale sans vérification de l'existence de modifications ultérieures à l'ouverture. A la fermeture, les modifications sont propagées.

Dans les MANets, le modèle de cohérence souvent utilisé est la *cohérence faible* : quand un donnée est répliquée, un baille, ou TTL *(time to live)*, lui est associé, comme par exemple dans [18]. Quand celui-ci arrive à terme, la donnée est considérée comme trop ancienne et on en recherche une nouvelle version. Les modifications durant le TTL ne sont donc pas prises en compte. C'est un modèle plutôt utilisé pour les données qu'un seul pair peut éditer. Quand plusieurs pairs peuvent modifier la même donnée, des modèles plus complexes sont employés afin de maintenir la cohérence. Ce modèle est en fait la cohérence Δ , avec Δ=TTL [95].

3.2.2 Mise en œuvre

Chaque modèle de cohérence peut être mis en œuvre de différentes manières. Différents algorithmes ont donc été proposés, constituant des briques qui permettent de mettre en œuvre un modèle de cohérence.
Nous allons tout d'abord voir comment, dans certains cas de figure, la cohérence peut être fournie de manière simple si la gestion de la donnée est centralisée. Nous verrons ensuite comment les modifications d'une donnée peuvent être propagées aux pairs l'hébergeant. Nous détaillerons ensuite comment ordonnancer les événements au sein d'un système distribué grâce à l'utilisation d'horloges logiques. Nous verrons finalement les différents types algorithmes d'exclusion mutuelle distribuée puis le problème de réconciliation de versions.

3.2.2.1 Hiérarchie des répliques

Dans les systèmes répliquant les données, toutes les répliques n'ont pas forcément un rôle équivalent.
Dans un système à *serveur*, les répliques principales sont stockées sur le serveur, qui est donc un point de centralisation. Par ailleurs, deux terminaux communicants ne peuvent pas travailler ensemble s'ils ne sont pas connectés au serveur. Cependant, cette technique a l'avantage de fournir un point central où les décisions, comme l'ordonnancement des événements, le verrouillage d'une donnée, ou la réconciliation, peuvent être prises. Les systèmes de bases de données, ou de gestions de versions sont généralement construits avec un système de copie-maître [82].

Dans un système à copie-maître (ou copie primaire), chaque donnée est associée à un terminal particulier, généralement celui ayant créé la donnée [33]. La copie hébergée par ce terminal est la *copie-maître*. Le propriétaire de la copie-maître fait donc office de serveur pour la donnée. Quand un processeur modifie la donnée, il doit faire valider ses modifications par le processeur en charge de la copie-maître, qui se charge de la propagation, et des réconciliations si besoin est. L'avantage de cette technique par rapport à l'utilisation d'un serveur, est que toutes les copies maîtres ne sont pas nécessairement sur le même terminal. Par ailleurs, pour une donnée, le terminal hébergeant la copie-maître peut changer. La charge est donc mieux répartie et le système plus tolérant aux pannes.

Dans un système *pair à pair* où toutes les répliques jouent le même rôle, on doit mettre en place des solutions totalement distribuées. Ces solutions, qui s'appuient sur des algorithmes comme l'exclusion mutuelle distribuée ou l'ordonnancement des événements, sont décrites plus en détail dans les sections suivantes.

3.2.2.2 Propagation des modifications

Quand la donnée est modifiée sur un site, les répliques situées sur les autres sites deviennent invalides. L'invalidation d'une réplique peut être faite de deux manières :

- *Push* : le pair ayant modifié la donnée en informe les autres.
- *Pull* : chaque pair s'informe auprès du pair susceptible de modifier la donnée pour savoir si une modification a eu lieu.

L'approche *Pull* est simple à mettre en œuvre dans le cas d'un serveur ou d'une copie-maître, mais n'est pas efficace si la donnée peut être potentiellement modifiée sur plusieurs terminaux : il faut, dans ce cas, contacter tout le monde. On préfère alors le *Push*.

Dans le cas où le terminal ayant modifié la donnée propage ses modifications (*Push*), il a deux possibilités [51] :

- Mise à jour : le pair ayant modifié la donnée envoie la nouvelle version aux autres pairs.
- Invalidation : le pair ayant modifié la donnée signale aux pairs que leur version en cache est invalide.

Le choix de la mise à jour ou de l'invalidation dépend du type des données et de leur utilisation. Pour des données de grande taille, ou auxquelles ont fait peu d'accès, il est plus intéressant d'utiliser l'invalidation. Ainsi, on sauvegarde les ressources qui auraient été utilisées pour envoyer des données non lues. Au contraire, pour les données de petite taille, ou pour les données fréquemment utilisées, les mises à jour sont plus intéressantes : une donnée de petite taille prend peu de place par rapport à un message d'invalidation. Par ailleurs on économise les messages nécessaires pour obtenir la nouvelle version dans le cas d'une invalidation.

Certains systèmes utilisent une propagation hybride : certains sites, choisis sur des critères de fréquence d'accès, ou de capacités, reçoivent une mise à jour, tandis que les autres reçoivent une invalidation. Quand l'un de ces sites veut accéder à la donnée, il s'adresse au possesseur de la dernière mise à jour le plus proche.

3.2.2.3 Ordonnancement des événements

Pour implanter un modèle de cohérence séquentielle ou causale, les processeurs doivent pouvoir construire un ordre, total ou partiel, sur des événements. Dans un système où les modifications ne sont pas centralisées, ceci peut être fait de plusieurs manières.

La première méthode utilise une horloge physique. Elle consiste à synchroniser les horloges systèmes de tous les processeurs. On calcule le RTT [1] entre deux terminaux puis la différence de leurs dérives d'horloge pour la corriger. En associant une date à chaque événement, ceux-ci peuvent alors être ordonnés totalement.
La seconde méthode consiste à utiliser des horloges logiques pour établir un ordre causal [90]. Il existe plusieurs types d'horloges logiques.
Les horloges les plus simples sont les horloges scalaires, ou horloge de Lamport [58]. Comme illustré par la figure 3.2, chaque processeur utilise un compteur qui est incrémenté à chaque événement (accès, envoi et réception de message). A chaque envoi de message, cette valeur est jointe. Quand un processeur reçoit un message daté à M, il augmente sa propre horloge C à la valeur max(M, C)+1. Ces horloges permettent de connaître la relation partielle *arrivé avant* entre deux événements en comparant leurs dates.

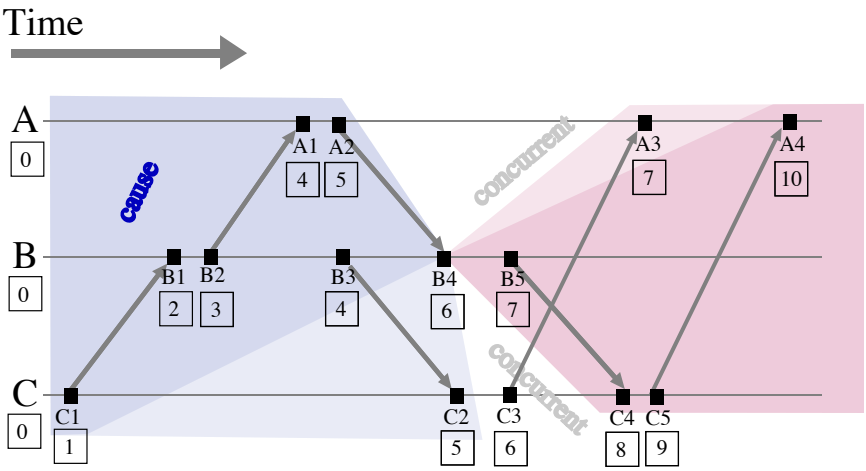


FIGURE 3.2 – Un exemple d'horloges scalaires [2]

Une proposition plus complexe, les horloges vectorielles, ou horloge de Mattern [71], permettent d'obtenir un ordre causal sur les événements, comme illustré par la figure 3.3 :
- Pour une application distribuée sur N sites, une horloge est un vecteur VC de taille N, initialisé à 0.
- Quand un événement (émission ou réception de message, et tout autre événement notable) survient sur le site i, il incrémente VC[i] de 1.
  - Si cet événement est une émission, l'horloge est jointe au message.
  - Si cet événement est une réception, l'horloge est mise à jour en prenant pour chaque VC[j] le max entre la valeur locale et la valeur reçue.

La technique de synchronisation des horloges physiques est peu efficace dans le cadre d'un MANet. Dans un réseau mobile, le RTT entre deux terminaux varie en permanence. Par ailleurs, cette technique génère un grand nombre de messages. On ne peut donc pas utiliser une telle méthode dans un MANet, à moins de matériel spécifique, comme un GPS.
Les horloges de Lamport ne créent pas de charge supplémentaire conséquente (on joint un entier à chaque message), mais elles ne permettent pas de déterminer un ordre causal. Les

1. Round Time Trip : temps entre l'envoi d'un message et la réception de la réponse

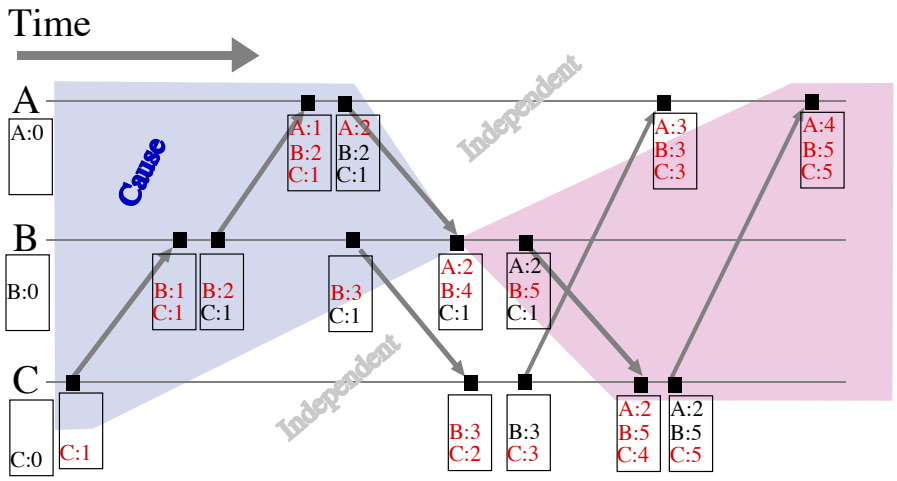


Figure 3.3 – Un exemple d'horloges vectorielles [2]

horloges de Mattern permettent de déterminer un ordre causal sur les événements mais nécessitent d'échanger plus d'information.

3.2.2.4 Exclusion mutuelle distribuée

Dans certains modèles de cohérence, comme par exemple la cohérence à l'entrée, ou la cohérence séquentielle, il est nécessaire d'utiliser des verrous permettant d'assurer que si un terminal possède un verrou, il est le seul à exécuter une action : cette action est exécutée en section critique. Cela peut servir, par exemple, à garantir qu'un terminal est le seul à pouvoir éditer une donnée. On utilise pour cela des algorithmes d'exclusion mutuelle distribuée.

Un algorithme d'exclusion mutuelle distribuée doit garantir deux propriétés :

- la vivacité : un terminal demandant l'entrée en section critique est sûr de l'obtenir en un temps fini,
- la sûreté : un seul terminal peut être en section critique à un instant donné,
- l'équité : un terminal ne doit pas obtenir plus souvent la section critique qu'un autre (en proportion de leur nombre de demandes).

Dans une architecture où la donnée est gérée par un serveur ou par un terminal fixe (qui possède donc la copie-maître), la gestion du verrou peut être effectuée de manière centralisée par l'hôte possesseur de cette copie. Quand un pair veut obtenir le verrou, il envoie une requête au responsable de la donnée qui attribue généralement le verrou de manière premier arrivé, premier servi (*FIFO*).

Dans un modèle pair à pair, en revanche, la décision d'accorder le droit d'entrer en section critique doit être prise par tous les terminaux. On parle alors vraiment d'exclusion mutuelle distribuée.

Il existe deux classes d'algorithme d'exclusion mutuelle distribuée : les algorithmes à jeton, et les algorithmes à permission [89].

Dans les algorithmes à permission, comme Maekawa [67], Suzuki Kasami [98] ou Ricart Agrawala [91], quand un pair veut entrer en section critique, il doit obtenir la permission d'un nombre suffisant de sites. Il est nécessaire de déterminer de quel groupe de pair on doit obtenir la permission. Les algorithmes proposent un groupe constitué : soit de

l'ensemble des pairs ; soit de la majorité des pairs ; soit d'un groupe de pairs, le quorum. Les quorums sont constitués de manière à ce que si un pair obtient l'accord de l'ensemble de son quorum pour rentrer en section critique, on garantisse qu'aucun autre pair ne puisse également obtenir un accord de l'intégralité de son quorum.
Dans un système où les terminaux apparaissent et disparaissent, ces groupes doivent être maintenus, ce qui cause un surcoût en messages. Ces algorithmes posent aussi le problème de pouvoir retirer sa permission : si la moitié du groupe a donné sa permission à A et l'autre moitié à B, le système se trouve bloqué jusqu'à ce que certains pairs retirent leur permission. Ceci peut être géré grâce aux horloges logiques permettant d'ordonner les requêtes. Par ailleurs, ces algorithmes ne montent pas en charge car le nombre de messages nécessaires pour obtenir l'entrée en section critique est trop important.
Dans les algorithmes à jeton, comme Naimi Trehel [77] ou Raymond [88], la permission d'entrée en section critique est symbolisée par la possession d'un jeton qui circule entre les terminaux. Se pose alors la question de localisation du jeton afin de faire une demande d'entrée en section critique. Par ailleurs, si le possesseur du jeton disparaît, il est nécessaire de pouvoir régénérer un jeton unique. Cela crée donc un point de faiblesse dans le système. Ces algorithmes génèrent cependant moins de messages que les algorithmes à permission.

3.2.2.5 Réconciliation

Dans certains modèles de cohérence, notamment les modèles à cohérence faible, il est possible que deux copies de la donnée soient autorisées à diverger, i.e. deux processeurs accèdent à la donnée en écriture en même temps.
Dans des systèmes avec un point de centralisation, comme un serveur, ou un système de copie primaire, la réconciliation peut être effectuée de manière centralisée. C'est ce qui est effectué par exemple dans le système de fichiers distribué Coda [54], qui supporte les opérations quand un pair est déconnecté du serveur, ou le système de version Subversion [84].

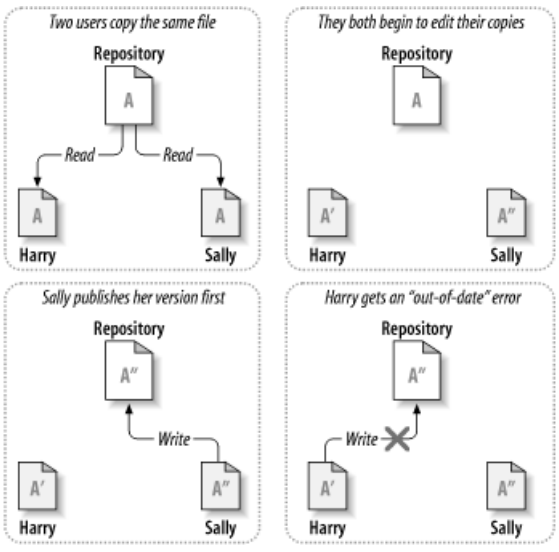


FIGURE 3.4 – Subversion détecte les modifications concurrentes

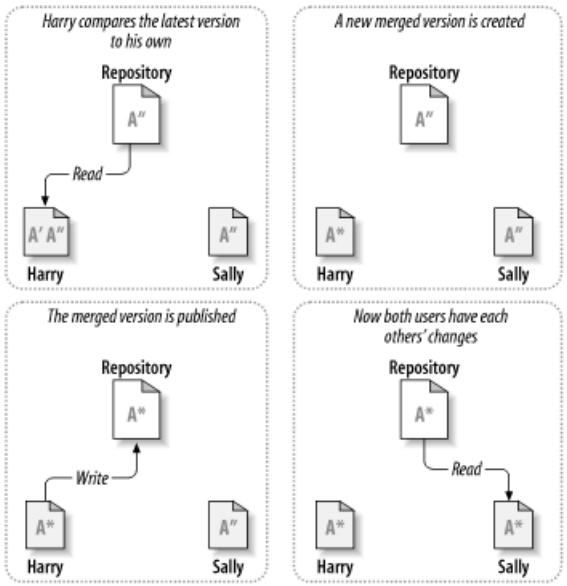


FIGURE 3.5 – et les fusionne automatiquement

Dans les systèmes les plus simples, la première version reçue par le serveur est validée et

propagée, et les suivantes sont ignorées. Dans des systèmes plus évolués, et si la nature des données le permet, les différentes versions sont réconciliées pour créer une version nouvelle intégrant toutes les modifications. Quand cette réconciliation automatique n'est pas possible, l'utilisateur est alors sollicité pour résoudre les conflits.
Dans des systèmes totalement distribués, le problème est plus complexe puisque tous les terminaux doivent effectuer une réconciliation et obtenir le même résultat. Si les événements sont totalement ordonnés, le premier dans l'ordre choisi est conservé. Si l'on veut fusionner les modifications, les différentes versions doivent être conservées afin de pouvoir déterminer un ancêtre commun quand une nouvelle version est proposée par un autre terminal. Les conflits qui ne peuvent pas être résolus par la machine sont laissés à l'humain. C'est ce qui est effectué dans certains cas par XMiddle [69] et adHocFS [14], deux systèmes que nous allons décrire dans le chapitre suivant.

3.2.2.6 Type de Donnée Commutatif Répliqué

Un *CRDT (Commutative Replicated Data Type)*, est un type de donnée conçu de manière à ce que les modifications de la donnée soient commutatives, excepté les modifications effectuées par le même utilisateur qui doivent être appliquées dans l'ordre. On peut donc travailler de manière déconnectée sur ces données, et en fusionner par la suite les modifications.
Les CRDT ont été utilisés pour des systèmes de Wiki pair à pair. Comme nous avons comme objectif de démontrer nos algorithmes avec un wiki pair à pair pour MANet, ce type de données, qui offre une solution élégante applicable dans les MANets, est intéressant. Au chapitre 4, nous présentons deux CRDT [59], WOOT [78] et Treedoc [86], qui en expliqueront mieux, par l'exemple, le fonctionnement.

3.3 Création de grappe *(clustering)*

Le *"clustering"* consiste à regrouper des terminaux en sous-groupes ayant des propriétés particulières, appelés grappes (*cluster*). Ces algorithmes s'intéressent ici à la constitution de groupes de proximité, comme illustré par 3.7. Ces grappes sont ensuite utilisées dans la gestion de la réplication et la cohérence des données.
Dans le cadre d'un réseau filaire hiérarchique, on peut simplement déterminer un groupe de proximité optimal en terme de distance, qui est ensuite invariant, moyennant les arrivées et les départs. Dans un MANet, du fait de la mobilité, les terminaux voisins d'un pair varient dans le temps. Il faut donc périodiquement déterminer la composition du groupe. Par ailleurs, on cherche à construire des groupes stables dans le temps. Il ne suffit donc pas d'examiner son voisinage immédiat pour déterminer un groupe stable, mais il faut aussi considérer les terminaux présents par le passé.
Cette technique est par exemple utilisée dans adHocFS pour déterminer des groupes de proximité dans lesquels appliquer une cohérence forte, et une réplication préventive. Dans adHocFS, ce sont tous les terminaux en vue qui sont dans la même grappe.
Pour créer des grappes, trois types d'approches sont utilisées.
La première approche consiste à créer des grappes au sein d'un groupe dense de terminaux : le problème est de sélectionner un groupe de pairs stables pour y placer un service, puis d'agréger les pairs par proximité autour de ces têtes de grappes, comme dans [113], où ce choix se base sur les capacités (batterie, stockage) des pairs. Ces techniques ne prennent pas en compte la possibilité de partition du réseau.

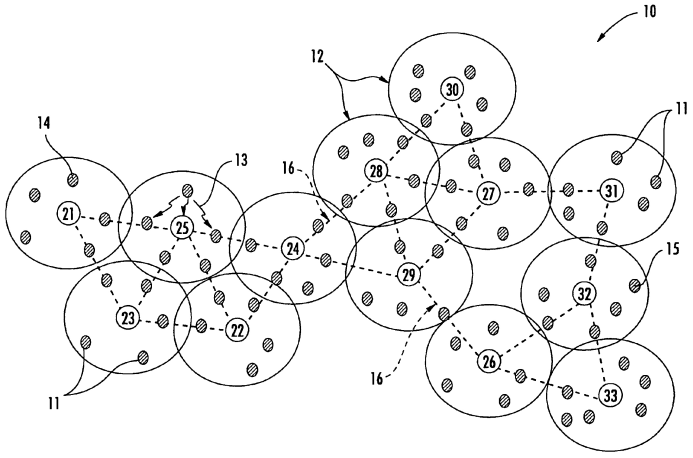


FIGURE 3.6 – Un exemple de clustering, ici pour faire du routage hiérarchique

La seconde approche consiste à prédire les partitions dans le réseau en se basant sur les positions passées des terminaux, avec des informations GPS ou les informations de routage, comme proposé dans [24], [103] ou [97]. Ces techniques sont coûteuses en calcul car elles supposent de reconstruire les trajectoires des terminaux. Elles nécessitent par ailleurs l'échange d'information de position, ou le calcul de vitesses relatives entre terminaux, par exemple en se basant sur la puissance du signal.
Enfin, la troisième approche est de créer des groupes stables entre terminaux. Dans [103], ceci est fait en calculant la distance moyenne aux autres pairs dans le graphe de routage, tandis que dans [38], l'algorithme cherche les boucles dans le graphe de routage.

3.4 Recherche/découverte

La recherche et la découverte de données sont deux problèmes différents, qui peuvent cependant être résolus avec des solutions communes :
– La *découverte (lookup)* consiste à localiser une donnée à partir de son identifiant.
– La *recherche* consiste à obtenir une liste d'identifiants de données à partir d'une requête.
Une recherche et une découverte se font donc dans une structure de type index, ou table de hash, qui associe une valeur (la liste des endroits ou se trouve la donnée, ou la liste des identifiants des données correspondant à la requête) à une clé (l'identifiant ou la requête). Selon le contexte, ce problème peut être résolu de manière centralisée avec un serveur ou avec plusieurs serveurs hiérarchisés, ou de manière distribuée par inondation (réseau de petite taille) ou avec des tables de hash distribuées. Nous les présentons ci-dessous.

3.4.1 Système centralisé

Dans un serveur de fichiers, les données sont localisées à un emplacement connu, et l'index pour la recherche se trouve sur le serveur lui-même. Il n'y a donc ici pas de problème lié à la distribution.
Cependant une centralisation de l'index peut aussi être utilisée dans un système distribué. Les anciens systèmes pair à pair, comme Napster [6], utilisaient aussi un serveur centralisé

pour héberger la liste des données et des pairs.
Ce serveur peut être répliqué entièrement plusieurs fois afin de supporter une montée en charge, avec les différentes répliques maintenues cohérentes, mais cela reste un système centralisé.
Notons que dans un réseau MANet, le serveur peut être élu dynamiquement, voir migrer périodiquement afin de préserver les capacités des terminaux élus.

3.4.2 Serveur, contenu distribué

Dans un système hiérarchique, chaque serveur a connaissance d'une partie des informations et est connecté aux autres serveurs. Quand un serveur reçoit une requête, s'il n'a pas connaissance de la réponse, il fait suivre la requête. Les serveurs sont organisés en graphe, afin de pouvoir contacter l'ensemble des serveurs.
C'est le cas des caches DNS *(Domain Name System)* [85], qui permettent de traduire les adresse IP en URL. Les serveurs sont organisés en arbre, avec des serveurs de noms ayant autorité sur un domaine d'adresse et dont la réponse est correcte. Cependant, pour éviter de les surcharger, les associations IP/URL, sont mises en cache dans des serveurs DNS secondaires, comme par exemple ceux des ISP.

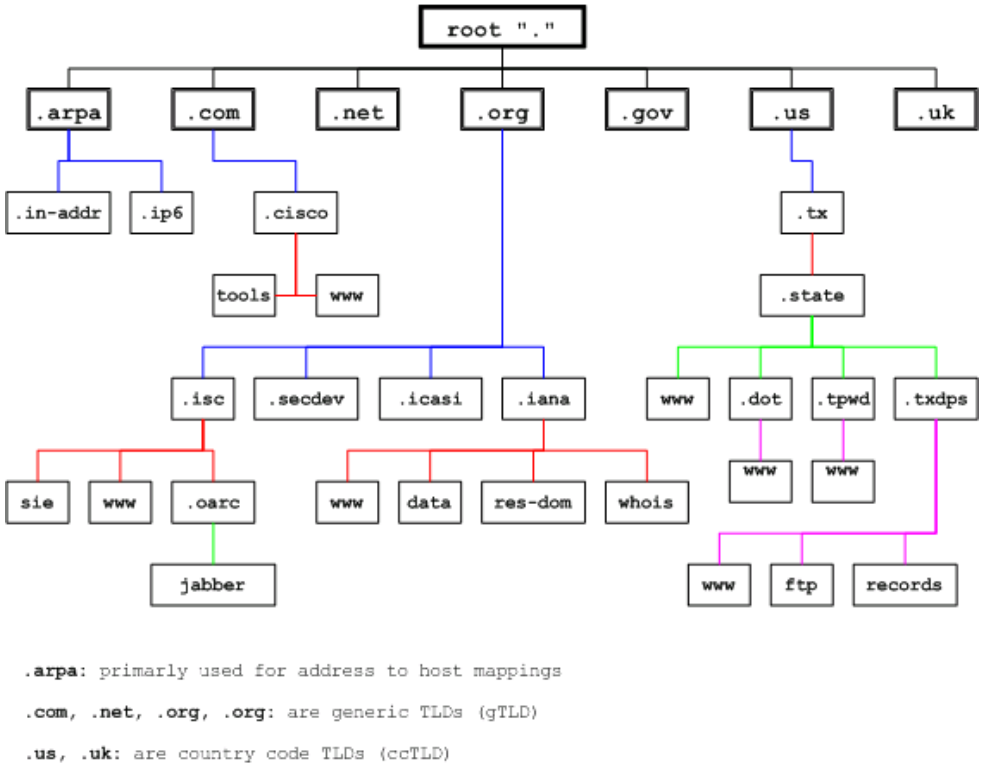


Figure 3.7 – Une vue partielle de la hiérarchie DNS

Dans un tel système l'information est distribuée et répliquée afin de distribuer la charge correspondant à la réponse aux requêtes et à la réunion les informations. Il s'appuie cependant sur une hiérarchie rigide de serveurs connus, qui n'est pas adaptée aux MANets.
Ici aussi les serveurs peuvent être dynamiquement élus. Dans certains systèmes de caches pour MANets par exemple, les terminaux sont organisés en grappes et la tête de chaque

grappe a une connaissance globale des données stockées par les terminaux dans sa grappe. Une requête est donc faite à la tête de la grappe locale, qui la fait suivre aux autres têtes de grappes si l'information n'existe pas localement.

3.4.3 Cache et inondation

Quand aucune infrastructure n'est mise en place pour la découverte des données, chaque terminal doit gérer sa propre localisation des données.
Pour localiser une donnée, la méthode la plus simple, mais aussi la plus coûteuse et terme de messages, est d'inonder le réseau avec une requête. Les techniques d'inondation sont rarement utilisées seules, et un cache de chemins (ou de réponses à une requête en cas de recherche) est généralement maintenu.
Pour améliorer cette technique, un pair peut, par ailleurs, écouter les requêtes et les réponses circulant sur le réseau pour peupler son cache, soit avec la localisation des données, soit avec les données elles-mêmes.

3.4.4 DHT

Dans les systèmes pair à pair grande échelle, l'ajout d'un serveur hébergeant un index et un système de découverte des données pose des problèmes de passage à l'échelle et de confidentialité.
Des solutions, comme Chord [96] ou Pastry [93], ont donc été proposées pour permettre de localiser une valeur en se basant sur une clé. La clé peut être par exemple l'identifiant d'un objet, et la valeur l'objet lui-même.
Un identifiant (grand, généralement 128bits) est associé à chaque pair, et à chaque donnée. Les données sont placées sur le pair dont l'identifiant est le plus proche (pour des raisons d'optimisation, les données sont souvent répliquées et placés sur les N pairs dont les identifiants sont les plus proches). A partir d'un identifiant, on peut donc trouver le pair hébergeant la donnée sans passer par un service de nom.

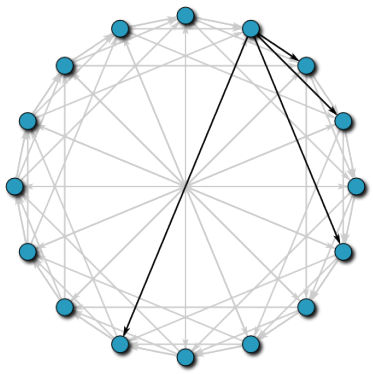


FIGURE 3.8 – Connaissance partielle du réseau

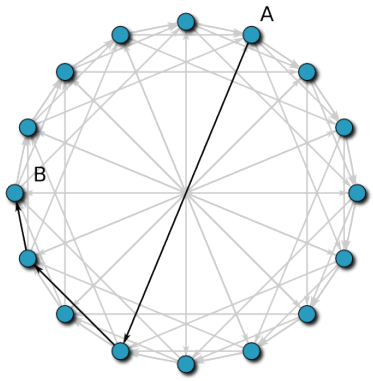


FIGURE 3.9 – Routage d'une requête

Par ailleurs, par construction, afin de pouvoir passer à l'échelle, chaque noeud n'a pas à connaître l'ensemble du réseau, mais seulement un sous-ensemble, construit de manière à

ce qu'une requête puisse être routée à sa destination. Un noeud envoie donc sa requête au noeud de sa connaissance, et dont l'identifiant est le plus proche de celui recherché.
Ces systèmes peuvent aussi être utilisés pour la recherche de donnée, en calculant l'identifiant d'un mot-clé par hachage. La liste des données associées au mot-clé est ensuite placée sur le pair correspondant.

3.4.5 Synthèse

Dans un MANet, les systèmes centralisés, ou les systèmes avec des serveurs hiérarchiques fixes ne conviennent pas. Les DHT sont conçus pour des systèmes décentralisés, et pour être tolérants aux fautes. Cependant, ils sont aussi conçus pour des systèmes à très large échelle, ce qui n'est pas notre cas, et ils ne prennent pas en compte la localisation.
Dans les MANets, les solutions d'inondation sont beaucoup utilisées, mais ne conviennent qu'à des réseaux de petite taille. Plusieurs solutions ont aussi été proposées pour adapter les DHTs aux MANets. Dans un réseau de plus grande taille, nous pensons qu'utiliser des serveurs hiérarchiques élus dynamiquement est un bon compromis.

3.5 Discussion

Développer un intergiciel complet résolvant toutes ces problèmatiques relève d'un projet de recherche complexe. Dans le cadre de cette thèse, nous avons participé à deux projets d'intergiciel pour réseaux mobiles ad hoc regroupant plusieurs équipes de recherches sur 2 ans, Popeye [13] et Transhumance [81].
La solution proposée par cette thèse a d'ailleurs été conçue dans l'idée de s'intégrer au sein de Transhumance.
Notre contribution consiste en un *un algorithme collaboratif de réplication proactive au sein de voisinages stables, tirant partie d'informations sémantiques sur les données.* Nous proposons aussi, dans la seconde partie, la mise en œuvre de ces algorithmes pour le développement d'un moteur de wiki pair à pair pour MANets.
Dans le chapitre suivant nous présentons les solutions qui ont été proposées pour répondre aux problématiques que nous venons de présenter

Chapitre 4

Etat de l'art

Ce chapitre décrit les travaux de recherche effectués dans le domaine des réseaux mobiles ad hoc, qui concernent certaines des problématiques évoquées dans le chapitre précédent : réplication des données et remplacement de cache, maintien de la cohérence, gestion de la mobilité, et recherche/localisation des données.
Dans le chapitre précédent nous avons vu des solutions générales aux systèmes de partage de données. Dans ce chapitre nous présentons des solutions conçues spécifiquement pour les MANets. Nous ne nous attachons pas à présenter les solutions de localisation et de recherche des données, car dans le cadre de notre thèse nous nous appuyons sur un système déjà existant, développé pour l'intergiciel Transhumance.
Nous allons tout d'abord présenter des systèmes de partage de données sur MANet, puis des systèmes de cache pour MANets. Nous verrons ensuite des algorithmes de création de grappes et leurs usages, avant de présenter les schémas de réplication pour MANets.
Dans les deux sections suivantes, nous verrons les solutions proposées pour la cohérence de données. Tout d'abord, différentes techniques pour la cohérence pessimiste (invalidation de cache et algorithmes d'exclusion mutuelle distribuée adaptés aux MANets), puis pour la cohérence à terme (CRDT).

4.1 Systèmes de partage de données

Il existe des systèmes de partage de données non modifiables, comme par exemple Orion[55] ou 7DS [80]. Le problème est alors la localisation des données, et ces systèmes proposent soit des algorithmes permettant d'inonder efficacement le réseau avec une requête, soit de répliquer les solutions de localisation par le contenu (DHT) utilisés dans les réseaux pair à pair grande échelle.
Dans cette section nous nous intéressons aux systèmes de partage de données modifiables pour MANets. Ces solutions se rapprochent donc plutôt des systèmes de fichiers distribués.

4.1.1 adHocFS

Le système adHocFS [14] est un système de fichier distribué destiné aux réseaux mobiles ad hoc. Il a été développé à l'INRIA en 2003. Les fichiers partagés sont de type Unix.
Dans adHocFS, les pairs partagent des fichiers au sein d'un groupe de sécurité(GdS). Un pair peut appartenir à plusieurs GdS, un fichier appartient à un seul GdS. A l'intérieur d'un GdS, adHocFS constitue des groupes de proximité entre les pairs capables de communiquer. Ces groupes sont utilisés dans les schémas de réplication et de cohérence.
Les membres de chaque groupe élisent un leader. Ce rôle est tournant afin de ne pas surcharger un terminal en particulier. Le leader est chargé de maintenir la liste des pairs dans le groupe. A l'intérieur du groupe, les terminaux échangent leurs profils qui, pour un terminal, décrit les répliques hébergées ainsi que les capacités de stockage du terminal.

4.1.1.1 Réplication

Les répliques de données sont de deux natures : les terminaux créent tout d'abord à la demande des *répliques de travail* pour leur propre besoin. S'il y a seulement une seule réplique de travail, des *répliques préventives* sont ensuite créées (une par donnée). Le choix des hôtes pour ces répliques est basé sur le profil des terminaux.

4.1.1.2 Cohérence

La cohérence est hybride. A l'intérieur d'un groupe, les différentes copies sont gérées suivant un modèle de cohérence pessimiste avec un système de verrou pour obtenir un accès. Entre groupes, la cohérence est optimiste : les modifications sont journalisées afin de réconcilier automatiquement les différentes versions quand deux groupess se rejoignent. En cas de divergence, la réconciliation doit être faite manuellement par l'utilisateur.

4.1.2 Haddock

Haddock FS [10] est un système de fichiers distribué pour réseau mobile ad hoc
Dans Haddock FS, un terminal hébergeant une donnée est appelé gestionnaire de réplique (*replica manager*). La politique de réplication n'est pas présentée, on suppose qu'elle est faite à la demande.

4.1.2.1 Cohérence

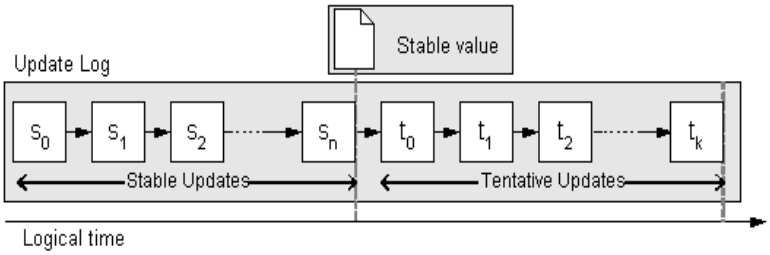


FIGURE 4.1 – Journalisation des mises-à-jour, extrait de [10]

Haddock FS propose un protocole de cohérence hybride.
Au sein d'un groupe fortement connecté, HaddockFS met tout d'abord en place un protocole de cohérence pessimiste basé sur un algorithme d'exclusion mutuelle à jeton. La détection d'un groupe fortement connecté et la mise en place de l'exclusion mutuelle n'est pas décrite.
Entre les groupes, la cohérence optimiste est mise en place en journalisant les modifications. Celles ci sont triées par ordre causal. Dans son journal, chaque gestionnaire de réplique stocke donc des mises à jour, ainsi que la version stable de la donnée, c'est à dire la dernière version sur laquelle il sait que tous les gestionnaires de répliques se sont accordés. Les mises à jour plus anciennes que la version stable sont appelés mise-à-jour stable (*stable update*), celles plus récentes sont appelées mises à jour provisoires *tentative update* 4.1.
La réconciliation est faire entre couple de gestionnaires de réplique, dès que ceux ci sont en contact. Cette réconciliation vise à construire une liste de mises à jour totalement ordonnée suivant l'ordre causal (un ordre partiel). Quand il y a plusieurs modifications en parallèle,

une seule sera donc choisie. Avec une granularité de l'ordre du fichier, c'est un choix très contraignant, puisque cela empêche la modification concurrente des fichiers, même dans des cas où un algorithme tel que diff-3[53], capable de construire la différence minimale entre deux chaînes de caractères ayant un ancêtre commun, ne détecterait pas de conflit.
Pour obtenir une version stable, chaque donnée a une copie primaire, la première version à avoir été créée. C'est cette version qui décide au final en cas de conflit, et qui est autorisée à créer des versions stables. Le statut de copie primaire peut être transféré, mais crée un point de centralisation.

4.1.2.2 Stockage et mise à jour

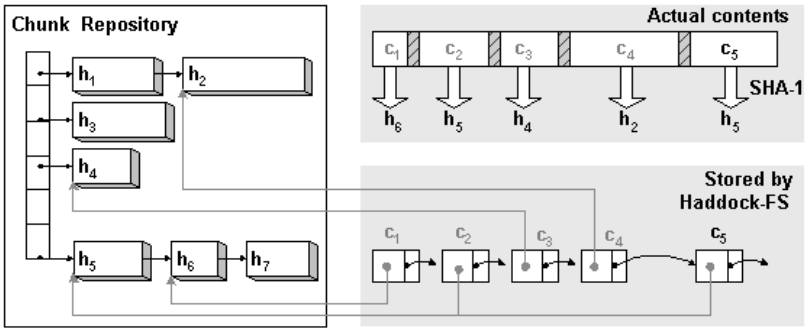


Figure 4.2 – Stockage des *chunks*, extrait de [10]

Haddock FS cherche à limiter l'occupation mémoire. Il faut donc réduire la taille des journaux de mises à jour stockés en mémoire, et la taille de la mise à jour elle-même échangée entre deux pairs.
Pour diminuer la taille des échanges, Haddock FS utilise la fonction de hash SHA-1. Celle ci prend en entrée une chaîne de caractères et retourne une chaîne de caractères statistiquement unique, et de plus petite taille.
Chaque donnée est tout d'abord découpée en morceaux, appelés *chunk*(morceau), comme dans 4.2. Un pair indexe ensuite les données qu'il héberge par leur valeur hachée . Quand la donnée est modifiée, un nouveau *chunk* est créé et inséré dans la liste de *chunk* existants. Au moment de la propagation des modifications, les valeurs hachées sont tout d'abord échangées afin de vérifier s'il est nécessaire de propager la modification elle-même.
Pour limiter la place prise par la journalisation des modifications, Haddock FS élimine d'abord les mises à jour antérieures à la version stable de la donnée (obtenu grâce à la copie primaire), et pour lesquelles il est certain que tous les pairs ont appliqués ces mises à jour. Le choix d'éliminer des mises à jour n'ayant pas encore abouti à une version stable est laissé à l'utilisateur et dans ce cas les mises à jour les plus anciennes sont éliminées, car ce sont celles qui ont le plus de chance d'avoir déjà été propagées.

4.1.3 XMiddle

XMiddle [69] est un système de partage de données éditables destiné aux MANets. Il a été développé en 2002 à l'University College à Londres.
La particularité de XMiddle est que les données partagées sont structurées sous forme d'arbre XML. Cela permet de partager des sous-arbres : un utilisateur n'a donc pas besoin

de répliquer l'intégralité d'une donnée. Par ailleurs, l'utilisation de balises XML permet d'introduire des informations sémantiques sur les données. Elles sont utilisées par XMiddle dans le cadre de la recherche de données.

4.1.3.1 Cohérence

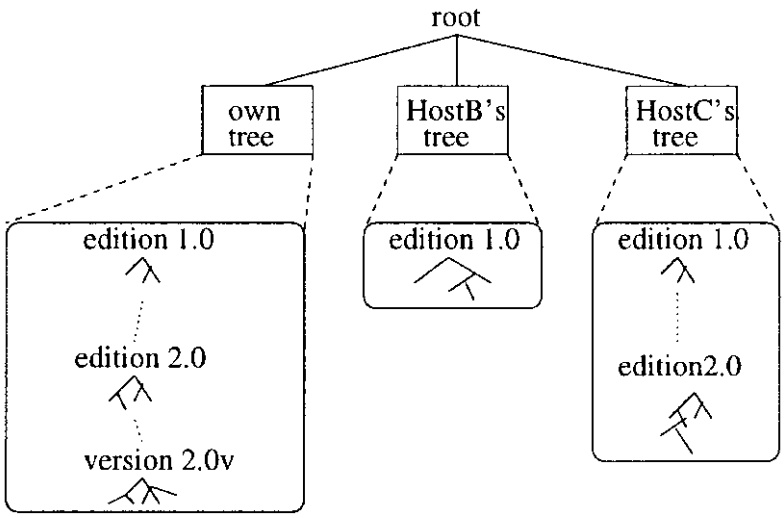


FIGURE 4.3 – Arbre des versions dans XMiddle, extrait de [69]

XMiddle offre un modèle de cohérence hybride : le propriétaire du document héberge la copie primaire et séquentialise les accès à la donnée dans la partie du réseau à sa portée. Hors de portée du propriétaire du document, les différentes répliques d'une donnée peuvent être éditées en parallèle. Les différentes versions de la donnée sont conservées jusqu'à ce qu'une réconciliation avec le propriétaire soit validée, comme illustré par 4.3. Les terminaux non propriétaires effectuent des réconciliations intermédiaires entre eux. Un algorithme de fusion d'arbre XML permet de réconcilier automatiquement les différentes versions.

4.1.3.2 Réplication

Dans XMiddle, la réplication est laissée au soin de l'utilisateur : une donnée est répliquée à la demande. Il n'y a donc aucune réplique superflue de créée. Cependant, ce mode de réplication ne garantit pas la continuité d'accès à la donnée dans le réseau si tous les terminaux l'ayant répliquée deviennent hors de portée.

4.1.4 Synthèse

Nous avons vu ici trois systèmes permettant le partage de données sur réseaux mobiles ad hoc.
XMiddle demande l'utilisation d'un modèle de donnée spécifique (un arbre XML), ce qui limite son utilisation à des applications qui puissent s'adapter. Cependant, l'utilisation d'informations structurée en XML est courante, et elle permet à XMiddle de fournir un algorithme de réconciliation applicable à toutes les données partagées. Même avec la journalisation des modifications, la réconciliation n'est donc pas forcément toujours possible.
AdHocFS permet le partage de fichiers de type Unix.

HaddockFS propose un module d'extension à un système de fichiers existant. Les opérations sont donc totalement transparentes pour l'utilisateur.
XMiddle laisse la réplication à la charge de l'utilisateur. Il n'offre donc aucune garantie de disponibilité sur la donnée en cas de partition. AdHocFS au contraire garantit la survivance d'une copie en cas de disparition. On ne sait cependant pas à quelles données est accordée la préférence s'il n'y a pas assez de terminaux pour tout conserver. Le modèle de réplication dans HaddockFS n'est pas précisé.
Ces trois systèmes proposent un modèle de cohérence hybride où certains terminaux travaillent en cohérence pessimiste alors que d'autres travaillent en cohérence optimiste. Le choix du modèle de cohérence n'est pas laissé au développeur. Il pourrait pourtant être parfois nécessaire pour certains documents d'appliquer un modèle de cohérence forte même entre différentes parties du réseaux, alors que pour d'autres, la cohérence pourrait être optimiste au sein d'une même partie. Par ailleurs, malgré la mise en place d'une cohérence pessimiste, une réconciliation manuelle peut être nécessaire.
Enfin, ces systèmes ne font pas usage de métadonnées dans la gestion des données, bien que XMiddle ait possibilité de le faire. Ces métadonnées sont utilisées pour la recherche de données.

4.2 Systèmes de cache de données

Dans cette section nous présentons les systèmes de cache de données pour MANets.
Ces caches contiennent des données non modifiables, comme des données Web, ou modifiables par une seule source, et gérées avec un modèle de cohérence faible. Des algorithmes d'invalidation de cache plus complexes sont présentés par la suite.
Les données partagées dans ces caches ne sont pas modifiables, et chaque copie en cache reste valide durant un temps limité. Il n'y a donc ici pas de problème de cohérence complexe : une copie de donnée est simplement associée à un Time To Live (TTL).
Ces travaux s'attachent donc à déterminer comment localiser les données intéressantes, quand les répliquer et surtout comment nettoyer le cache, c'est à dire quelles données effacer quand on doit libérer de l'espace-mémoire.

4.2.1 Cache web pour terminaux mobiles sensible à l'énergie

Dans [94], Sailhan propose un système de cache de données Web coopératif prenant en compte le problème du coût en terme d'énergie.
Les terminaux sont constitués en réseau mobile ad hoc, lié en certains points à des stations fixes *(base stations)* connectées à internet. Ils peuvent donc acquérir des données Web à travers un nombre limité de passerelles.

4.2.1.1 Réplication des données, localisation d'une réplique

Les données sont distribuées sur les terminaux en fonction des accès faits par les utilisateurs. Sur un terminal T, un profil est donc maintenu pour chaque terminal A avec lequel il a interagi. Ce profil contient un entier représentant la somme du nombre de fois où A a satisfait une demande de T, et où T a satisfait une demande de A. Pour chaque terminal A, T calcule ensuite une valeur F correspondant au profil divisé par sa distance à T.
Quand T cherche une donnée, il va interroger les terminaux tour à tour, par F décroissant. Si un terminal a la donnée, il envoie un message *hit* à T, contenant le *TTL* de la copie de

la donnée en sa possession, et *Capacity*, une évaluation de ses ressources.
Parmi les terminaux ayant répondu positivement, T choisi la source de la donnée en prenant en compte la distance, la fraicheur de la donnée, et la capacité du terminal source.

4.2.1.2 Nettoyage du cache

Chaque terminal effectue une gestion locale de son cache quand il a besoin de place, en sélectionnant les données à éliminer sur 3 critères :
- popularité : le nombre d'accès, local et distant, à la donnée depuis qu'elle est en cache. Cette valeur sert à prédire le nombre d'accès futurs.
- coût d'accès : le coût d'accès à la copie de la donnée la plus proche.
- cohérence : si le TTL de la donnée est dépassé, celle ci n'est plus valide. Cependant, s'il reste peu d'énergie au terminal, on préfère conserver la donnée même si elle est périmée, plutôt qu'avoir à nouveau à la récupérer à distance.

4.2.2 CachePath, CacheData, HybridCache

[112] décrit un système de cache pour MANET.
Dans ce système, les données sont stockées sur un serveur et ne peuvent être modifiées qu'au niveau du serveur. Un pair peut répliquer une donnée s'il le souhaite. Yin propose 3 algorithmes, exécutés par tous les pairs, qui ne visent pas à répliquer les données pour leur intérêt propre mais pour diminuer le trafic global. Pour ce faire, il doit examiner le trafic concernant les données (requêtes et réponses). Par défaut, les requêtes sont envoyées au serveur.
Yin propose tout d'abord deux algorithmes simples :
- *CachePath* : le pair examine les réponses à des requêtes. Quand il voit une donnée en provenance d'un pair P1 pour un pair P2, il sait que P2 a une copie de la donnée. Si sa distance à P1 est supérieure à sa distance à P2, il mémorise le chemin pour cette donnée comme allant vers P2 : il fera suivre la prochaine requête pour cette donnée à P2.
- *CacheData* : le pair examine le nombre de requêtes faites pour une donnée. Si ce nombre de requêtes est élevé et qu'elles proviennent de différents pairs, il réplique la donnée.

Selon la taille et la fréquence de modification de la donnée, l'une ou l'autre méthode diminue la charge réseau :
- si la donnée est de taille faible, *CacheData* est intéressant puisque peu d'espace mémoire est utilisé.
- si le TTL est bas, *CachePath* n'est pas intéressant puisque la copie qu'on indique sera rapidement invalidée et qu'il faudra alors retransmettre la réponse au serveur.
- pour un pair découvrant l'existence d'une réplique sur un pair i, s'il est proche de ce pair, il est intéressant d'utiliser *CachePath*.

Yin propose donc un algorithme, *HybridCache*, combinant ces deux techniques. Cette technique fixe des bornes limites de taille et de TTL puis selon les valeurs attachées à la donnée, adopte un comportement ou l'autre.
La cohérence du cache est gérée par invalidation avec un TTL. Quand le TTL d'un chemin expire, il est éliminé du cache. Quand une donnée est invalidée ou que son TTL expire, elle est éliminée du cache mais son identifiant est conservé, avec la mention invalide. Quand le pair voit un message contenant une copie d'une donnée qu'il a possédée mais qui a été invalidée, il peut pro-activement en faire une copie.

4.2.3 ZC, LUV

Dans [20], Chand propose une technique de gestion de cache coopératif qui vise à éliminer du cache les données les moins utilisées.
Cette politique LUV *(least value utility)* est appliquée au niveau d'un cache coopératif au sein d'une zone de coopération ZC.

4.2.3.1 Politique de réplication

Quand un noeud accède à une donnée, il décide d'en garder une copie en cache en se basant sur la distance entre lui et le noeud lui ayant servi la donnée.
La cohérence des données est un modèle de cohérence faible gérée avec des TTL.

4.2.3.2 LUV

Quand un noeud doit libérer le cache, il choisit d'éliminer en premier les données les moins utiles. Pour cela, plusieurs critères sont utilisés :
– popularité : le nombre d'accès en moyenne glissante,
– distance : la distance entre le noeud et la source de la donnée,
– l'âge de la donnée : si le TTL a expiré, la donnée n'est plus valable,
– la taille : il est plus intéressant d'éliminer une donnée de grande taille.

4.2.4 COOP

COOP [27] est un système de cache coopératif où les modifications d'une donnée ne peuvent être issues que d'une seule source. Il propose un modèle de localisation des données, ainsi qu'une technique de gestion du cache. La cohérence des données n'est pas examinée.

4.2.4.1 Localisation des données

Trois techniques sont proposées pour la localisation des données :
– *Adaptative Flooding* : l'inondation d'une requête n'est faire que sur n sauts, n étant calculé en fonction de la densité du réseau.
– *Profile based résolution* : chaque pair P examine les requêtes de données qui passent et conserve une liste de paires associant l'identifiant d'un pair R ayant fait la requête à l'identifiant de la donnée, assortie d'un TTL (time to live). Quand un pair veut accéder à une donnée, il examine ce cache. Si une requête correspond et est récente, le pair P interroge le pair R avant d'inonder le réseau.
– Si aucune des techniques ci-dessus n'a donné de résultat, la donnée est demandée directement à sa source.

4.2.4.2 Gestion du cache

Le voisinage d'un pair est constitué de ses voisins à N sauts (N étant un paramètre adaptable de l'intergiciel).
COOP cherche à maximiser le nombre de succès. Il préfère donc éliminer du cache les données en double dans le voisinage. Quand un pair reçoit une donnée, il la classe donc comme primaire ou secondaire :
– si la donnée vient d'un pair hors du voisinage, elle est primaire.
– si la donnée vient d'un pair dans le voisinage, plusieurs cas sont considérés :

– si elle provient d'un pair l'ayant classée comme primaire, la donnée est classée secondaire.
– si elle provient d'un pair l'ayant classée comme secondaire, celui ci doit joindre l'identité du pair la lui ayant fourni ; si celui ci est hors du voisinage, la donnée est classée primaire, sinon elle est classée secondaire.

Pour nettoyer son cache, un pair garde, en priorité, les données primaires. Au sein d'une classe, LRU est appliqué.

4.2.5 Synthèse

Le tableau 4.1 résume les différents systèmes proposés.

TABLE 4.1 – Synthèse des propositions de cache pour MANET

Travaux	Localisation	Mise en cache ?	Nettoyage de cache	Cohérence
[94]	Cherche d'abord sur les terminaux avec lesquels on a le plus interagi	Mise en cache à la demande	Basée sur popularité, coût et cohérence	TTL
[112]	Inondation, examen et cache des chemins	Cache la localisation quand on voit une demande, cache la donnée si elle est très demandée		TTL
[20]		Cache selon distance de la copie servie	Basée sur popularité, taille, âge et coût	TTL
[27]	Inondation limitée, cache des chemins, demande à la source	Réplique à la demande, et classe les données par priorité (primaire/secondaire) selon la distance à l'hôte fournissant la copie, et à sa classification	Garder en priorité les données primaires, LRU	

4.3 Création de grappes de mobilité

Les systèmes que nous avons vus précédemment prennent en compte la mobilité des terminaux. AdHocFS par exemple met en place une politique de cohérence hybride, pessimiste au sein d'un groupe de terminaux en vue, et optimiste entre groupes disjoints. Nous verrons dans la section suivante que des algorithmes de réplications utilisent eux aussi des groupes de mobilité pour répliquer les données.
Afin de pouvoir anticiper la mobilité des terminaux, plusieurs algorithmes de création de grappes dans les MANets ont donc été proposés.
Dans cette section nous présentons ces algorithmes.

4.3.1 Une métrique basée sur la mobilité pour former des grappes dans les MANets

Dans [11], Basu propose une technique de construction de grappes de 2 sauts de large. Cet algorithme est totalement distribué et se base sur la puissance des signaux reçus pour calculer les distances.
Chaque noeud diffuse périodiquement un message à un saut. A chaque itération, il calcule sa distance à ses voisins. Après plusieurs itérations, en se basant sur les vecteurs de distance obtenus, il calcule ensuite la vitesse relative de chaque voisin. Chaque noeud calcule ensuite sa vitesse relative moyenne aux autres noeuds et la diffuse à un saut. Le terminal avec la vitesse relative la plus basse devient une tête de grappe à laquelle se rattachent ses voisins. Si un terminal voit deux têtes de grappes, il devient une passerelle *(gateway)*.
Ces grappes sont utilisées pour un service de routage et de multi-diffusion *(multicast)* plus efficace.
Cette proposition a été validée sur NS-2. Les deux se déplaçant à 20m/s (72km/h), cette proposition est destinée aux réseaux véhiculaires. De plus, la surcharge réseau de cette proposition n'a pas été évaluée. A grande vitesse, le calcul de distance basé sur la puissance du signal manque de précision.

4.3.2 Une approche basée sur la mobilité pour offrir une gestion de la mobilité et du routage multidiffusion dans les MANets

Dans [7], An propose un algorithme de construction de grappes basé sur des informations de position GPS.
Chaque terminal n effectue périodiquement une série de mesures de positions GPS grâce auxquelles il construit un vecteur vitesse V(n,t) qu'il diffuse. A réception d'un V(n, t), un terminal calcule la vitesse relative de n par rapport à lui-même. Une grappe est constituée de terminaux dont la vitesse relative entre chaque paire de terminaux est inférieure à un seuil. Le terminal avec l'adresse IP la plus petite est désigné comme tête de grappe.
Ces grappes sont utilisées pour le routage et la multi-diffusion.
Cette proposition requiert l'utilisation d'un GPS.

4.3.3 Prédiction de la mobilité et routage dans les MANets

Dans [97], Su propose une technique permettant de prédire la durée de vie d'un lien. Cet algorithme est distribué et nécessite des informations de distance (calculées ici à partir d'informations GPS), et la synchronisation des horloges des terminaux (horloge du GPS). Ces conditions pourraient être satisfaites avec un calcul de distance basé sur la puissance du signal, et en utilisant le protocole NTP, mais les mesures seraient moins précises. Il est aussi nécessaire de connaître le diamètre de propagation du signal.
Un terminal calcule son vecteur vitesse à partir d'une série de mesures de position, puis, bien que ce ne soit pas explicité dans le papier, le diffuse à un saut. Deux de terminaux peuvent donc calculer le temps pendant lequel ils resteront connectés à partir de leurs positions et leurs vélocités respectives, et de la portée du signal. Cette information est utilisée au niveau de la couche réseau pour offrir un routage et une diffusion plus efficaces. L'algorithme a été validé sur le simulateur GloMoSim, avec divers scénarii. Le surcoût en terme de messages n'est pas mesuré.
Cette proposition nécessite un GPS et des horloges physiques synchronisées (par exemple, celles des GPS). Par ailleurs, dans la description des scenarii d'évaluation, les terminaux

se déplacent à une vitesse comprise entre 18km/h et 72km/h. Cette proposition est donc destinée aux réseaux véhiculaires.

4.3.4 Couverture de services efficace et fiable dans les MANets

Dans [103], Wang fait l'hypothèse que les utilisateurs se déplacent en groupe et sont munis de terminaux identiques, équipés d'un GPS. Il distingue des services critiques, comme par exemple un serveur de base de données, qui peuvent être répliqués (terminaux identiques). Il incombe au serveur de prédire la possibilité de partition parmi ses clients afin de répliquer son service. Wang propose alors deux algorithmes permettant la reconnaissance de groupes de mobilité :

– le premier, exécuté par les serveurs, se base sur la vitesse des terminaux pour calculer la vitesse moyenne d'un groupe. Cet algorithme permet d'obtenir des groupes précis mais est coûteux (calcul des déplacements de tous les terminaux) et nécessite de centraliser les informations de position.
– le second algorithme, exécuté par les clients, ne nécessite pas de centralisation des informations : un terminal T détermine, tout d'abord, ses voisins dans le graphe de routage. Pour chaque voisin V, il effectue ensuite L mesures de sa distance à V puis calcule la moyenne M, ainsi que l'écart-type S de cet échantillon. V fait partie du même groupe que T si M et S ne dépassent pas certains seuils fixés par le système. Cet algorithme est exécuté périodiquement.

Ces approches ont été validées par simulation, avec un outil non précisé.
Si la première proposition est centralisée, la seconde proposition est totalement distribuée, mais nécessite quand même l'utilisation de mesures GPS pour calculer les distances.

4.3.5 Mobilité de groupe et prédiction de partition dans les MANets

Dans [104], Wang propose un algorithme de création de grappe séquentiel.
Chaque terminal calcule son vecteur vitesse et sa position. Une grappe est caractérisée elle aussi par son vecteur vitesse (la moyenne des vitesses de ses membres) et son centre (le barycentre des positions de ses membres). La construction est ensuite faite de manière séquentielle en utilisant par exemple l'ordre sur les adresses IP. Pour démarrer l'algorithme, un terminal est tout d'abord choisi comme appartenant à une première grappe. Chaque terminal t calcule sa distance au centre de chaque grappe existante. Si la distance minimale est inférieure à une valeur α, il se rattache à la grappe correspondante. Sinon, une nouvelle grappe est créée avec t comme premier membre.
La validation a été faite, semble-t-il, par simulation.
Il n'y a pas d'implantation proposée et les échanges de messages ne sont pas explicités. Comme l'algorithme nécessite de pouvoir déterminer des vecteurs vitesse et des distances entre terminaux, un système de positionnement, par exemple un GPS, est nécessaire.

4.3.6 Détection de partition dans les MANets

Dans [38], Hauspie propose un algorithme de prédiction de partitions qui se base sur le calcul de la robustesse d'un lien entre deux noeuds. Cette robustesse est basée sur la redondance des chemins dans le graphe de routage entre les deux noeuds : plus il y a de chemins possibles, plus la probabilité que les deux noeuds soient déconnectés est faible.
Ces informations sont utilisées pour répliquer les services en cas de partition.

L'algorithme est évalué par simulation, avec des terminaux se déplaçant à une vitesse de piéton cependant aucune implémentation n'est proposée. Il n'y a donc pas d'évaluation du coût de l'algorithme, mais comme la solution nécessite de recréer le graphe de routage, on peut supposer au plus simple une implémentation centralisée, où chaque pair envoie ses tables de routages à un serveur qui prédit ensuite les partitions.

4.3.7 Gérer la mobilité de groupe dans la réplication de données en environnement mobile

Dans [43], Huang propose un algorithme pour créer des groupes de mobilité au sein d'un réseau. Cette technique se base sur des informations GPS.
Chaque terminal crée une liste de positions qui est ensuite diffusée jusqu'à une distance fixée par le système dans le réseau, accompagné d'une estampille temporelle. Le terminal avec la plus petite adresse IP parmi le groupe joignable devient responsable de zone (*Zone-Master*). Il organise les terminaux en grappes selon leurs vecteurs vitesse, calculés à partir des positions passées, et sélectionne les têtes de grappe.
Ces grappes sont ensuite utilisées pour répliquer des données.
L'algorithme de création de grappe lui-même n'est pas évalué en tant que tel, mais le système de réplication l'utilisant est évalué sur un simulateur développé pour l'occasion, avec des terminaux se déplaçant à vitesse piétonne.
Bien qu'il ne soit pas explicitement cité, la création d'une liste de positions nécessite un système de positionnement, GPS ou équivalent.

4.3.8 Un algorithme de réplication de données au sein de grappes dans les MANets pour améliorer la disponibilité

Dans [114], Zheng propose un algorithme de construction de voisinage α-stable. La valeur α représente une probabilité de connexion entre deux noeuds à l'instant t + 1. La probabilité de connexion entre deux voisins est calculée à partir des distances entre les noeuds à l'instant t, et en considérant une vitesse de déplacement maximale. La probabilité de connexions entre deux noeuds non-voisins est inférée à partir de la valeur de chaque saut. Un lien est dit α-stable si cette probabilité est supérieure à α.

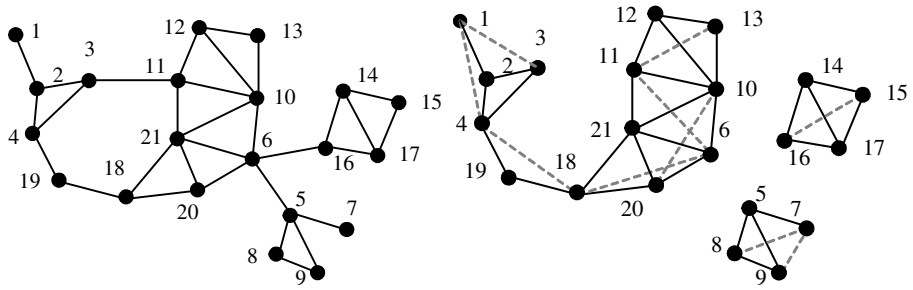


Fig. 1. UDG **Fig. 2.** *α-stable graph* G

Figure 4.4 – Exemple de topologie et graphe α-stable associé, extrait de [114]

L'algorithme proposé est distribué. Chaque terminal calcule sa distance à tous ses voisins, puis la probabilité de connexion pour chacun. Deux ensembles sont créés : SN l'ensemble des voisins stables et SP, l'ensemble des noeuds avec lesquels le lien est α-stable (au départ

SN = SP). A partir de ces valeurs, chaque terminal calcule une nouvelle valeur de SP. Si un noeud n'appartient à aucune grappe et a l'adresse IP la plus petite parmi l'ensemble des noeuds non-membres d'une grappe, il devient tête de grappe et construit une grappe comme l'ensemble l'incluant et tel que le lien entre chaque paire est α-stable et l'algorithme est exécuté à nouveau jusqu'à ce que tous les noeuds soient dans une grappe.

Ces grappes sont utilisées pour répliquer les données afin qu'elles restent accessibles en cas de partition.

Chaque pair utilise sa distance par rapport à tous ses voisins, mais la technique utilisée pour obtenir cette valeur n'est pas indiquée. Elle requiert soit un système de positionnement de type GPS puis une diffusion à un saut, soit un calcul de distance basé sur la puissance du signal.

4.3.9 Réplication au sein de grappes pour MANets de grand échelle

Dans [113], Yu propose un algorithme de construction de grappes basé sur les ressources des terminaux.

Pour initialiser la construction des grappes, l'algorithme élit tout d'abord des têtes de grappes en se basant sur leurs capacités. Le nombre de grappes souhaité M et le nombre de pairs N est connu de tous. Les terminaux diffusent leur profil (batterie, mémoire, identité) dans tout le réseau. Chaque terminal classe les terminaux par capacité et les M premiers deviennent les têtes de grappe *(clusterhead)* initiales. Chacune de ces tête de grappe diffuse alors un message annonçant son statut et l'identité de sa grappe. En se basant sur la distance parcourue par ces messages, un pair peut alors sélectionner la grappe à laquelle il appartient. Si une tête de grappe disparaît, le noeud parmi ses voisins à 1 saut ayant l'identifiant le plus petit devient tête de grappe.

Ces grappes sont ensuite utilisées pour mettre en place des tables de hachage distribuées, utilisées, par exemple, pour la recherche de données.

Cette solution est intéressante, car elle tente de prendre en compte les ressources des terminaux. Cependant, si, au début, les têtes de grappes sont effectivement choisies pour leur ressources, quand une tête de grappe disparaît, le choix du noeud la remplaçant est simplement fait sur son identifiant. De plus, si tous les noeuds avec le plus de ressources sont au même endroit dans le réseau, les grappes ne seront pas régulières.

4.3.10 Un algorithme de prédiction de partition pour gérer la mobilité dans les MANets

Dans [25], Ha propose un algorithme de prédiction de partition se basant sur les vecteurs vitesses des terminaux.

Dans cette proposition, les terminaux sont déjà constitués en groupe, et chaque groupe est modélisé par un vecteur-vitesse et une aire de couverture. Une comparaison de ces vecteurs permet de déterminer quand deux groupes se séparent, comme illustré dans 4.5.

Cet algorithme est validé par simulation mais il n'y a pas de proposition concrète d'implémentation. La mise en œuvre de cet algorithme nécessiterait d'une part que les terminaux soient déjà organisés en groupe, d'autre part que chaque groupe puisse calculer son vecteur-vitesse.

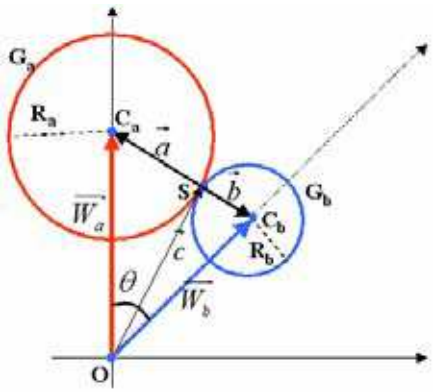


Figure 4.5 – Prédiction des partitions en se basant sur les vecteur-vitesse et les couvertures, extrait de [25]

4.3.11 Prédiction des déconnexions dans les MANets pour aider le travail coopératif

Dans [24], De Rosa propose un algorithme centralisé de prédiction de partitions.
A l'instant t, tous les terminaux calculent leur distance à leurs voisins en fonction de la puissance du signal reçu. Ces informations sont ensuite envoyées à un coordinateur qui calcule le graphe probable à l'instant t+1 en se basant sur les distances passées. Il essaye ensuite de prévenir les déconnexions, en déplaçant des terminaux pour servir de pont.
Ainsi, le graphe du réseau est toujours connexe, et il n'y a jamais de partition.
Cette proposition est destinée à des réseaux de piétons. Elle a été évaluée en terme de précision (l'algorithme prédit-il correctement les distances entre terminaux) sur NS et Glomosim, avec de bons résultats. Cependant, il n'y a pas d'évaluation du coût en nombre de messages échangés.
L'idée de prévenir la partition du réseau en déplaçant les utilisateurs est intéressante. Cependant, cette proposition fait l'hypothèse d'une équipe coordonnée (ici, une équipe cherchant à évaluer les dégâts subis par un site de fouille archéologique après un sinistre), une hypothèse forte, et contraignante pour les utilisateurs.

4.3.12 Synthèse

Nous avons vu ici des algorithmes de création de grappes pour réseaux mobiles ad hoc.
Pour construire une grappe, les algorithmes se basent sur plusieurs critères possibles :
– vecteurs de déplacement calculés à partir de positions absolues
– vitesses relatives
– graphe de routage
Au sein d'une grappe, la tête de grappe, s'il en existe une, est choisie soit sur des critères de capacité, soit sur des critères de distances.
Il existe aussi des grappes construits en choisissant tout d'abord les têtes de grappes, sur des critères de capacités, avant de regrouper les terminaux selon la distance. Il est possible d'obtenir des grappes dont la répartition dans le réseau n'est pas homogène, si les terminaux ayant le plus de capacités ne sont pas uniformément répartis.

Notons que, pour la plupart de ces algorithmes, la surcharge réseau engendrée n'est pas évaluée.
Les tableaux 4.2, 4.3 et 4.4 résument les travaux présentés.

4.4 Méthodes de réplication

Afin d'améliorer l'accessibilité aux données, la plupart des systèmes de partage de données pour MANet créent des répliques des données. Il est alors nécessaire de déterminer sur quels terminaux ces données sont placées. Le problème de placement de données consiste donc à choisir un ensemble d'hôtes pour répliquer la donnée tel qu'il minimise le nombre de messages tout en maximisant la disponibilité.
La décision de répliquer peut être prise individuellement par un pair sur des critères individuels, comme par exemple son intérêt pour les données, ou sur des critères globaux, comme l'intérêt collectif pour une donnée, ou la position du pair dans le réseau. Certains algorithmes prennent aussi en compte les caractéristiques de la donnée elle-même, comme sa taille ou sa mutabilité.
Nous allons voir ici six algorithmes de réplication de données, dont la plupart sont cependant destinés à des systèmes de cache, où les données ne peuvent pas être éditées de manière concurrente. Elles ne sont donc pas directement applicables, mais pourraient être adaptées par exemple dans un système utilisant des copies-maîtres.

4.4.1 Travaux de Takahiro Hara, effectués de 2001 à 2004

Takahiro Hara a travaillé de 2001 à 2004 sur des algorithmes de réplications, destinés à des réseaux mobile ad hoc où les terminaux ont des capacités de stockage limitées (capteur). Les algorithmes proposés sont adaptés à différentes utilisations des données. Dans tous les cas, la donnée est stockée en permanence par le pair dont elle est issue.

Dans son premier article[34], Hara propose un schéma de réplication de données pour des données non modifiées et dont on connaît par avance la fréquence fixe d'accès par chaque pair.
Il propose alors trois méthodes de placement global des données. La distribution n'est pas faite indépendamment pour chaque donnée mais en tenant compte de l'ensemble des données. Cela permet donc d'équilibrer la charge dans le réseau. Les trois méthodes sont :
- *Static access frequency (SAF)* : chaque pair réplique individuellement des données par ordre de fréquence d'accès décroissante. Cet algorithme crée potentiellement de nombreuses répliques voisines. La proposition suivante permet donc de réduire ces répliques tout en gardant un placement pertinent par rapport aux fréquences d'accès.
- *Dynamic Access Frequency and Neighborhood (DAFN)* : dans cette méthode, chaque pair prend en compte les fréquences d'accès ainsi que la topologie avoisinante. La première allocation se fait comme SAF. A chaque période de réallocation suivante, chaque pair diffuse ses fréquences d'accès aux données et détermine les répliques existantes et leur localisation. Si deux voisins à un saut ont une réplique de la même donnée, celui qui a la fréquence d'accès la plus basse élimine cette réplique. Il réplique à la place une donnée n'étant possédée par aucun de ses voisins. Cette technique diminue le nombre de répliques créées. Cependant, elle ne prend en compte que les voisins à un instant donnée. D'une étape à l'autre, les voisins, et donc les répliques peuvent donc beaucoup

Table 4.2 – Synthèse des algorithmes de création de grappe pour MANET 1/2

Travaux	Contexte	Contraintes	Technique	Implémentation, Validation	Complexité
[11]	Améliorer le routage/la diffusion	Calcul des vitesses relatives nécessite mesure de la puissance du signal	Calcul de la vitesse relative moyenne à tous ses voisins à un saut. Échange, dans un voisinage à un saut. Le terminal avec la plus basse vitesse relative devient tête de grappe	Distribuée, validation sur NS	Initialement diffusion d'un vecteur de vitesse moyenne, maintenance des grappes non précisée
[7]	Améliorer la multidiffusion	Positionnement GPS	Construction de vecteurs vitesse, échange, calcul de la vitesse relative à tous les terminaux et regroupement des terminaux de vitesses relatives faibles	?	Diffusion périodique des vecteurs vitesses
[97]	Améliorer le routage des paquets unicast/multicast, en prédisant les partitions du réseau pour choisir les routes	Positionnement, synchronisation des horloges	Echange d'un vecteur de positions à un saut, prédiction à partir des positions et vecteurs vitesses du temps pendant lequel deux pairs vont rester en contact	Distribuée, validation sur GloMoSim	N diffusion à 1 saut périodique des positions, période non indiquée
[103] - 1er algorithme	Réplication de services avant partition	Positionnement	Chaque serveur récupère les vecteurs de déplacement de tous les terminaux. Il prédit alors les partitions futures, et réplique les services.	Centralisé, validé par simulation (outil non précisé)	Tous les terminaux envoient périodiquement leurs positions à tous les serveurs

TABLE 4.3 – Synthèse des algorithmes de création de grappes pour MANET 2/3

Travaux	Contexte	Contraintes	Technique	Implémentation, Validation	Complexité
[103] - 2ème algorithme	Choisir le meilleur serveur parmi ceux hébergeant le service répliqué	Positionnement	Chaque terminal N récupère les informations de positions de chacun des autres noeuds sur un intervalle et calcule une distance moyenne. Si cette distance est inférieure à un seuil, le noeud fait partie du voisinage de N.	Distribué, validé par simulation (outil non précisé)	Tous les terminaux échangent périodiquement un jeu de positions
[104]	Prédire les partitions pour répliquer des services	Positionnement	Les distances entre pairs sont connues. Le terminal à l'identifiant le plus bas devient tête de grappe. Pour chaque terminal (trié par ordre d'identifiant) : si la distance minimale à une des tête de grappes est inférieure à un seuil, il rejoint la grappe associée, sinon, il crée une nouvelle grappe.	Pas d'implémentation proposée, peut être fait en distribué, mais séquentiel.	
[38]	Prédire les partitions pour répliquer des services	Accès aux tables de routages	Quand le nombre de chemins redondants dans le graphe de routage entre deux noeuds chute, prédiction d'une partition	Pas d'implémentation proposée	
[43]	Grouper les terminaux en grappes pour répliquer les données et en améliorer la disponibilité	Positionnement	Constitution d'une liste de positions et diffusion. Le terminal avec l'identifiant le plus bas constitue des grappes en fonction des positions et vitesses et désigne les têtes de grappe	Pas de validation de l'algorithme de création de grappes, validation du système global	Initialement diffusion de vecteurs de positions, battement de coeur périodique

Table 4.4 – Synthèse des algorithmes de création de grappes pour MANET 3/3

Travaux	Contexte	Contraintes	Technique	Implémentation, Validation	Complexité
[114],	Grouper les pairs en grappes pour répliquer les données afin d'en améliorer la disponibilité	Distance entre les pairs (technique non précisée)	Construction d'une liste de pairs α-stable, i.e. dont la probabilité de perte de connexion est inférieure à α.	Distribuée, validation par simulation	Echange d'information pour le calcul de distance, broadcast de position à 1 saut ? non précisé
[113]	Regrouper les terminaux en grappes gérés par des terminaux avec beaucoup de ressources, et donc moins susceptibles de disparaître	Accès aux tables de routages, évaluation des ressources de chaque terminal	Echange des ressources, tri des pairs par capacités descendantes, les M premiers deviennent têtes de grappe, les autres se rattachent à la tête la plus proche	Distribuée, simulation sur NS-2	Diffusion initiale de toutes les ressources, battement de coeur à la tête de grappe
[25]	Les groupes étant déjà constitués, prédire dans combien de temps ils perdront contact	Groupes déjà constitués	Sachant les vecteurs vélocité et la couverture de chaque groupe, on peut prédire la date à laquelle les deux groupes perdront contact	Pas d'implémentation proposée	
[24]	Equipe de piétons, prévenir les partitions en prédisant la mobilité	Possibilité de contrôler le déplacement des utilisateurs	Centraliser les distances relatives des terminaux (obtenues via la puissance du signal), prédire les distances futures	Centralisée, validation sur NS et Glomosim	N-1 messages à chaque période, durée d'une période non précisée.

changer. L'algorithme suivant détermine des groupes stables au sein desquels effectuer la réplication.
– *Dynamic Connectivity based Grouping (DCG)* : à chaque période d'allocation, chaque pair diffuse ses fréquences d'accès et détermine tous les pairs auxquels il peut être connecté par deux chemins de taille bornée. Cet ensemble de pairs constitue un groupe stable au sein duquel est calculé une fréquence moyenne d'accès pour le groupe. Les données sont alors allouées par fréquence moyenne décroissante dans le groupe jusqu'à ce qu'il n'y ait plus d'espace. Une donnée est répliquée sur le pair dans le groupe l'utilisant avec la plus grande fréquence et ayant encore de l'espace disponible.

Dans son second article[35], Hara améliore les algorithmes proposés dans [34]. Ces nouveaux algorithmes sont destinés à des données mises à jour de manière périodique. Cette mise à jour ne peut être effectuée que sur la copie du propriétaire, la copie primaire.
On considère connaître la fréquence de mise à jour d'une donnée et la fréquence d'accès d'un pair à une donnée. Pour chaque donnée et chaque pair, Hara définit la métrique PT, avec $pt_{i,j} = p_{i,j} * t_j$ où $p_{i,j}$ est la probabilité que le pair i accède à la donnée j durant une période entre deux mises à jour et t_j est le temps restant jusqu'à la prochaine mise à jour de la donnée j) ; cette valeur est plus élevée si le pair accède fréquemment à la donnée, ou si celle ci reste valide pour une longue période.
Cette nouvelle métrique est utilisée à la place de la fréquence d'accès dans les 3 algorithmes proposés, extensions de SAF, DAFN, et DCG :
– E-SAF : chaque pair réplique par PT décroissant
– E-DAFN : au début de chaque période, les voisins directs échangent leurs listes de valeurs PT. Si deux voisins ont intérêt à répliquer une donnée, seul celui pour qui PT est le plus élevé la réplique
– E-DCG : des groupes stables sont constitués, un PT moyen est calculé pour le groupe et les données sont ensuite répliquées par PT décroissant. Une donnée est placée sur le pair pour lequel la valeur de PT est la plus élevée.

Dans son troisième article [37] publié en 2004, Hara améliore les algorithmes de réplication proposés dans [34] pour prendre en compte la corrélation des données.
La corrélation de deux données, $p_{i-j,k}$ est définie comme la probabilité que l'utilisateur i ait accès aux données j et k en même temps. Cette mesure est déterminée pour chaque couple de données et ne change pas.
Hara introduit une autre mesure attachée à la données : sa priorité. La priorité d'une donnée est déterminée de la manière suivante :

1. On calcule la fréquence d'accès à la donnée en sommant la corrélation avec chacune des autres données. Pour chaque combinaison de X données parmi N (N est le nombres totale de données, X est un paramètre du système) parmi les données ayant la plus grande fréquence, on choisit les deux données les plus corrélées.
2. Parmi les données non sélectionnées, on classe successivement les données en fonction de la somme des corrélations de chaque donnée avec l'ensemble des données déjà sélectionnées.
3. Les données créées pas le pair i sont répliquées en priorité. Viennent ensuite les données choisies à la première étape. La priorité des autres données est déterminée par l'ordre dans lequel on les choisit à l'étape 2.

Hara définit 3 nouveaux algorithmes, basés sur SAF, DAFN et DCG :

– CSAF (correlated SAF) : dans cet algorithme, chaque noeud réplique par priorité descendante.
– CDAFN : le fonctionnement général est celui de DAFN. Quand deux voisins ont une réplique de la même donnée, celui pour qui la donnée est la moins corrélée aux autres données qu'il héberge la détruit.
– CDCG : comme dans DCG, l'algorithme construit des groupes stables. Une corrélation moyenne puis une priorité moyenne pour le groupe sont calculées pour chaque donnée et les données sont allouées dans le groupe par priorité descendante. Une donnée est placée sur le pair dans le groupe pour laquelle la donnée est la plus corrélée avec les données qu'il héberge déjà.

Dans ce dernier article [36], publié en 2004, Hara propose une extension de son article de 2001 [34] proposant la prise en compte des mises à jour des données. On considère toujours connaître les fréquences d'accès aux données, bien que celles-ci puissent changer dans le temps. Les données sont modifiées de manière apériodique, et seul le propriétaire de la donnée peut la modifier.
Tout d'abord, à chaque donnée et pour chaque pair est associé le ratio de lecture sur écriture RWR, $RWR = \frac{probabilite\ de\ lecture}{probabilite\ d'ecriture}$. Si RWR est élevé, on fait plus de lectures que d'écritures dans un segment de temps et la donnée va être répliquée. Si RWR est bas, on fait plus d'écritures que de lectures et la donnée ne va pas être répliquée.
Hara étend ensuite les trois algorithmes SAF, DAFN et DCG. Ces trois algorithmes sont réexécutés périodiquement et au début de chaque période d'allocation, le propriétaire de la donnée diffuse à tous sa probabilité d'écriture dans la période. A la réception de cette information, chaque pair calcule le RWR de chaque donnée.
– Dans E-SAF+, chaque terminal réplique au départ des données par RWR décroissant.
– Dans E-DAFN+, à chaque période d'allocation, chaque pair diffuse sa liste de RWR à ses voisins. Quand deux terminaux voisins sont susceptibles de répliquer la même donnée, seul celui dont le RWR est le plus élevé réplique la donnée.
– Dans E-DCG+, des groupes fortement connectés sont créés et maintenus à chaque période d'allocation. Pour chaque donnée, une valeur de RWR est calculée pour le groupe. Les données sont alors répliquées dans le groupe par RWR décroissant. Une donnée est placée sur le pair dont le RWR individuel est le plus élevé.

4.4.2 ARAM, EARAM, CDRA

Dans [115] ainsi que dans [48], publiés en 2004, Jing propose deux algorithmes d'allocations de données.
Le coût de création d'une réplique est évalué en nombre de sauts selon les paramètres suivants :
– accès en lecture : diminution du nombre global de sauts pour y accéder.
– accès en écriture : augmentation due aux messages nécessaires pour maintenir cette réplique à jour.
Le premier algorithme, ARAM *(Adaptative replica allocation algorithm in MANet)* est exécuté périodiquement. Chaque noeud récupère les accès que ses voisins ont effectués durant la période. Pour chaque donnée, un noeud détermine le nombre d'accès en lecture et en écriture, ce qui permet de calculer un coût pour la donnée :
– En supposant que le nombre d'accès soit identique durant la période suivante, si créer une réplique diminue le coût d'accès à cette donnée, on la crée.

– Si le coût en écriture dans le voisinage est supérieur au gain en lecture, on détruit la réplique
– Si pour une donnée D répliquée sur le pair S, le coût serait inférieur en la répliquant sur le pair P, la donnée est transférée à P et S arrête de l'héberger.

Le second algorithme, EARAM (Enhanced ARAM) travaille au niveau d'un groupe stable plutôt que sur le voisinage. Il élimine donc la réplication faite au bénéfice des voisins de passage et améliore donc ARAM en se concentrant sur les données susceptibles d'être utilisées régulièrement.

Dans un travail de la même équipe, [114], nous avons vu plus haut comment les terminaux sont regroupés en grappes.

Ces grappes servent ensuite à mettre en oeuvre l'algorithme CDRA (Cluster based replication algorithm).

– Au sein d'une grappe, quand un pair cherche à accéder à une donnée, il envoie une requête à la tête de grappe qui la diffuse au sein de la grappe locale. S'il y a une réplique locale, son hôte sert la requête directement.
– Sinon, la tête de grappe propage la requête aux autres têtes de grappe, qui diffusent la requête au sein de leur grappe. S'il y a une réplique dans la grappe, son hôte l'envoie à la tête de grappe locale, qui la transmet à la tête de la grappe où se trouve la requête.
– Celui ci transmet la donnée au pair l'ayant demandée, et la donnée est répliquée dans la grappe, en choisissant en priorité comme hôte.
– au sein d'une grappe, l'algorithme ARAM est appliqué.

En fait de schéma de réplication, cette proposition met plutôt en place un système de recherche des données avec une inondation partielle au niveau de la grappe. Cette description présente par ailleurs des faiblesses, avec de multiples retransmissions des données elles-mêmes, retransmissions qui créent du trafic inutile.

4.4.3 Gérer la mobilité de groupe dans la réplication de données en environnement mobile

Nous avons vu plus haut comment les grappes sont formées dans [43] .

Chaque pair dispose d'une capacité de stockage C. L'espace disponible dans la grappe est donc la somme de ces capacités. Les données sont triées par taille décroissante.

L'algorithme de réplication est appliqué au niveau de la grappe et se fait de manière séquentielle tant qu'il reste de la place dans la grappe :

– La donnée de taille la plus grande, et dont il n'existe aucune réplique, est sélectionnée
– Une réplique est placée sur le pair ayant le plus grand espace de stockage disponible, ce qui laisse donc le plus grand résidu.
– La capacité de stockage restante est recalculée

Cet article est plutôt centré sur l'algorithme de création de grappes lui-même. Cet algorithme est séquentiel et statique, mais ne s'adapte pas à la mobilité (apparition/disparition de noeud, changement de grappe).

4.4.4 Synthèse

Nous avons vu ici des algorithmes de placement de données. Les travaux de Hara cherchent à créer des répliques sur les pairs susceptibles de les utiliser, tout en limitant leur nombre grâce à une réplication collaborative au sein d'un voisinage. ARAM, EARAM et CDRA vise à contrôler le nombre de réplique de manière à limiter le trafic réseau. Le dernier

algorithme vise à optimiser l'utilisation de l'espace disponible au sein d'un groupe en plaçant en priorité les données de grande taille, ce qui limite le résidu.
.
Le tableau 4.5 résume les travaux présentés, leurs avantages et leur limites. Certains de ces travaux n'ont pas été présentés ici mais dans les section précédentes concernant la création de grappe [103] et la gestion de cache [112].

4.5 Cohérence des données

Dans cette section, nous allons voir plusieurs méthodes utilisées pour maintenir la cohérence des données.
Nous verrons tout d'abord des protocoles d'invalidation de cache, utilisables dans un système à copie primaire. Nous verrons ensuite des algorithmes d'exclusion mutuelle distribuée pour MANet, avant de présenter deux types de donnée commutative répliqué *(Commutative Replicated Data Type, CRDT)*, une solution permettant la mise en œuvre de cohérence optimiste.

4.5.1 Invalidation de cache pour MANET

Nous voyons ici les algorithmes d'invalidation destinés aux systèmes de cache pour MANet. Dans ces systèmes, chaque donnée est attachée à l'hôte l'ayant émise, cet hôte étant appelé maître, ou source. La copie de la donnée possédée par le maître est la seule à pouvoir être éditée. Les terminaux hébergeant les autres copies sont les hôtes caches. Le problème est alors de limiter le nombre de messages de mise à jour au strict minimum.
L'invalidation de cache peut venir de deux sources : l'invalidation peut être envoyée explicitement par la source, ou elle peut être envoyée à la demande quand un hôte-cache le demande à la source. Le premier cas *(push)* permet d'offrir un modèle à cohérence forte mais est coûteux en messages. Dans le second cas *(pull)*, la demande est généralement faite périodiquement, et non pas à chaque accès. Cette technique est donc moins coûteuse mais ne permet d'offrir qu'une cohérence faible.
Les algorithmes présentés ci-dessous visent à déterminer dans quel cas le maître doit effectuer un *push*.

4.5.1.1 Invalidation de cache pour données mises à jour dans les MANets

Cet article [39] s'inscrit dans les travaux effectués par Hara dans [34]. Il propose un modèle d'invalidation de données répliquées quand celles ci sont mises à jour en cas de MANet avec partition.
Dans ce système, chaque donnée ne peut être modifiée que par l'hôte l'ayant émise, c'est à dire par la source. Il est par ailleurs nécessaire que chaque hôte conserve pour chaque donnée, une estampille temporelle (*time stamp*) indiquant la date à laquelle cet hôte a entendu parler de la donnée pour la dernière fois.
Quand la source modifie la donnée, elle envoie un message d'invalidation aux hôtes ayant une réplique. Cependant, en cas de partition, certains hôtes ne sont pas joignables. Par ailleurs, du fait de la mobilité, les messages peuvent être retardés indéfiniment.
Hara propose deux méthodes d'invalidation permettant de propager une invalidation d'un hôte à un hôte en l'absence de la source. Un message d'invalidation comporte l'identifiant de la donnée ainsi qu'une estampille.

TABLE 4.5 – Synthèse des algorithmes de placement de données pour MANet

Travaux	Objectif	Editables	Info utilisées	Limitation	Collaboration
[103]	Minimiser nb copies : une par partie non connexe	?	Positions	Calculs de trajectoire	non
[34]	Créer copie sur pairs y accédant fréquemment	Non mutable	Fréquence d'accès	Nb de pairs et fréquence d'accès connus	SAF : pas de collaboration ; DAFN, DCG : réplication pour ses voisins
[35]	Créer copie sur pairs y accèdant avant la mise à jour	Mise à jour périodique	Fréquence des mises à jour et des lectures	Nb de pairs, fréquence d'accès et de mise à jour connues	
[37]	Créer copies des données utilisées en même temps	?	Fréquence d'accès, corrélation des données	Fréquence d'accès, corrélation des données et nb de pairs connus	
[36]	Créer copies de données plus souvent lues qu'écrites	Mise à jour apériodique	Ratio nombre accès lecture sur nombre accès écriture		
[48]	Créer des répliques au sein d'un voisinage pour diminuer son coût d'accès	Données éditables	Ratio nombre accès lecture sur nombre accès écriture	?	Prise en compte des accès du voisinage et transfert sur un autre pair s'il est plus intéressant qu'il héberge la donnée
[112]	Selon la taille de la donnée et la fréquence d'accès, mettre en cache une copie de la donnée ou un chemin vers la donnée	Données éditables	Examen du trafic (requêtes et réponses)	Modèle avec des lecteurs organisés en étoile autour d'une unique source	Cache pour ses voisins plus éloignés de la source
[43]	Au sein d'une grappe, placer de manière séquentielle les données par ordre de taille décroissant, pour limité le résidu	?	Taille des données	Séquentiel	Oui, une réplique par cluster

– *Update Broadcast Method* : quand un hôte reçoit un message d'invalidation pour une donnée, il compare l'estampille temporelle du message avec celle qui est associée à sa copie en cache. Si la sienne est plus récente, il ignore le message. Sinon, il jette sa copie, modifie l'estampille et transmet le message à ses voisins.
– *Connected Rebroadcast Method* : dans cette méthode, quand deux hôtes se rencontrent, ils échangent leurs tables d'estampille afin de mettre à jour leurs caches.

4.5.1.2 Stratégies d'invalidation de cache pour MANet

Dans [62], Li s'intéresse à l'invalidation de cache. Le contexte est celui d'un réseau ad hoc, où, parmi les pairs présents dans le réseau se trouve un terminal le MSS *(Mobile Support Station)* ayant accès à un réseau filaire et à des services présents sur ce réseau. Les autres terminaux mobiles MT *(Mobile Terminal)* accèdent à des informations à travers le MSS. Ils sont mobiles et capables de router les requêtes des autres terminaux.
Li propose 3 algorithmes d'invalidation de cache :
– POD *polling on demand* : quand un MT recoit une requête, il contacte le MSS pour déterminer si la donnée qu'il a en cache est valide.
– MAT *modified amnesic terminals* : le MSS crée puis diffuse périodiquement une liste des données mises à jour, IR, avec une période de L. Quand un MT reçoit un IR, il met à jour son cache. Si un MT est hors de portée de communication pour un temps supérieur à L, il invalide tout son cache. Quand un MT reçoit une requête, s'il a la donnée en cache, il attend le prochain IR avant de répondre.
– PAT *pull-based amnesic terminals* : le MSS crée périodiquement (période L) une liste des données mises à jour, qu'il date et garde en mémoire. Chaque MT récupère périodiquement (période L sans doute, cela n'est pas explicité dans l'article) et conserve les K derniers IRs. Avec les K derniers IRs, un MT peut garantir qu'une donnée n'est pas invalide depuis plus de L secondes. Si un MT perd la connectivité pour moins de K*L secondes, il récupère les derniers IRs avant de répondre à des requêtes. Si un MT perd la connectivité pour plus de K*L secondes, il invalide l'intégralité de son cache et ses IRs. Tandis que les deux premiers algorithmes maintiennent une cohérence forte, PAT maintient une cohérence Δ, avec $\Delta = L$.

4.5.1.3 Cohérence de cache coopératif dans les systèmes pair-à-pair mobiles sur MANets

Dans [18], Cao propose un système de cache à deux niveaux reposant sur la détection de pairs fiables. Chaque donnée est liée à un utilisateur hôte, la source, qui est le seul susceptible de la modifier. Cette proposition vise à minimiser le trafic réseau en limitant le nombre de messages, tout en tolérant les déconnexions.
Au sein du réseau, les pairs-relais sont sélectionnés sur trois critères : cœfficient d'accès (nombres d'accès au cache effectué sur le pair), cœfficient de stabilité (nombre de changement de statuts déconnecté/reconnecté), et cœfficient d'énergie (état de la batterie). Si un pair satisfait à ces trois critères (valeur supérieure à un seuil), il se propose comme pair-relai à la source. Ces critères sont périodiquement réévalués, et un pair peut donc arrêter d'être relai. Les pairs-relais forment par construction un réseau stable avec lequel la source est facilement en contact. La source envoie périodiquement *(push)* ses modifications aux pairs-relais. Quand un pair non relai souhaite accéder à la donnée, il la demande (pull) au pair-relai le plus proche plutôt qu'à la source. Quand un cache reçoit une requête, trois modes de cohérence sont offerts :

– cohérence faible WC *(weak consistency)* : le cache sert immédiatement la donnée.
– cohérence δ DC *(δ consistency)* : un cache associe à une donnée le compteur TTP (time to pull), réinitialisée à δ quand la donnée est mise à jour et décrémentée périodiquement. Si TTP est positif, la donnée est servie, sinon le cache interroge le pair-relai le plus proche, qui effectue le même test que pour la cohérence forte.
– cohérence forte SC *(strong consistency)* : le cache demande la donnée au pair-relai le plus proche. Sur un pair-relai, le compteur TTR *(time to refresh)* est associé à chaque donnée, réinitialisé à la période de *push* quand la donnée est mise à jour et décrémenté périodiquement. En cas de requête, si TTR est positif, la donnée est à jour et le relai répond donc à la requête du cache. Sinon, le relai attend la mise à jour de la source avant de satisfaire le cache.

Ce système de cache à deux niveaux permet de diminuer le nombre de messages en créant de multiples copies maintenues à jour dans le réseau. Par ailleurs, en plaçant des répliques sur des pairs stables, la donnée est toujours disponible à jour, même si la source est susceptible d'être déconnectée.
Cependant la sélection des pairs-relais ne prend pas en compte la topologie, c'est à dire leur position dans le réseau. La sélection se fait par ailleurs sur des critères absolus. En fin de vie du réseau, le niveau de batterie de tous les pairs ayant diminué, il risque donc de ne plus y avoir assez de pairs-relais disponibles. Il serait donc intéressant de prendre en compte des cœfficients de capacités relatifs aux autres pairs, ainsi que la distribution dans le réseau pour la sélection des pairs-relais.

4.5.1.4 Un algorithme de *push* sélectif pour maintenir la cohérence d'un cache coopératif pour MANets

Dans [45], Huang propose deux algorithmes pour maintenir un cache cohérent.
Dans ce système, chaque donnée est associée à un terminal, la source, qui est le seul pouvant la mettre à jour. La donnée peut être répliquée. Pour chaque réplique, la source connaît la fréquence des requêtes servies par cette source et associe une période de rafraîchissement δ.
Dans le premier algorithme, *Pull with TTR*, quand un noeud du cache récupère une copie, il arme le TTR à δ. Quand le TTR d'un noeud de cache expire, il récupère la dernière version de la donnée et réarme le TTR à δ. Cet algorithme garantit donc la cohérence δ mais n'est pas efficace en terme du nombre de messages échangés à cause du mécanisme de pull.
Dans le second algorithme, *Selective Push*, deux mécanismes sont mis en jeu par lesquels la donnée est mise à jour sur les noeuds de caches afin de diminuer le nombre de pull. D'une part, quand le TTR d'un noeud de cache expire, il récupère la dernière version de la donnée et réarme le TTR à δ. D'autre part, quand la donnée est mise à jour, la source construit un ensemble de pairs à qui il faut pro-activement envoyer la mise à jour. Les pairs appartenant à cet ensemble sont ceux pour qui il est probable qu'ils recevront une requête avant la prochaine mise à jour à la source, ou probable que la nouvelle mise à jour aura lieu avant la fin de leur TTR.

4.5.1.5 Une approche prédictive pour mettre en œuvre la cohérence dans un cache coopératif pour MANets

Dans [44], Huang propose un algorithme pour obtenir la cohérence faible dans un système de cache pour MANet.

Pour chaque donnée il existe une copie-maître que seule la source peut éditer. Huang propose un algorithme, PPC, permettant au maître, en cas de mise à jour, de déterminer s'il doit envoyer la nouvelle version aux caches (push) et aux caches de déterminer, en cas d'invalidation, s'ils doivent demander la nouvelle version au maître (pull).
Le maître et les caches conservent un historique des mises à jour envoyées par le maître et des mises à jour demandées par les caches. En se basant sur le passé, le maître calcule le nombre probable de pull qui arriveront entre l'instant présent et la prochaine mise à jour. Si ce nombre est assez élevé, le maître diffuse la donnée. De même, quand un hôte reçoit un message d'invalidation, il calcule la probabilité qu'il n'y ait pas de mise à jour du serveur entre deux requêtes. Si cette probabilité est assez élevée, il demande la nouvelle version au maître.

4.5.1.6 Synthèse

Nous visons des données modifiables depuis plusieurs terminaux. Les solutions proposées ici ne sont donc pas directement utilisables. Les deux dernières solutions utilisent la fréquence d'utilisation de la donnée par chaque pair. Dans un système où chaque hôte est susceptible d'envoyer un message d'invalidation, les fréquences devraient être distribuées à tous et le nombre de messages seraient alors beaucoup plus élevé (pour n hôtes, n^2 messages au lieu de n en cas de centralisation), et encore il n'est pas certain qu'on puisse déduire une fréquence représentative à partir d'accès sporadiques par plusieurs pairs. On pourrait cependant utiliser une solution similaire avec des fréquences déterminées statiquement.
L'utilisation d'estampille temporelle, comme dans la première solution proposée, est par contre envisageable avec des horloges logiques de Lamport. Elles permettent de déterminer qui, entre deux pairs, a les informations les plus récentes sur une donnée. Par contre, elles ne diminuent pas le nombre de messages ou le temps d'accès à une donnée.
Le tableau 4.6 résume les différents travaux présentés. La colonne message représente le nombre de messages échangés lorsqu'une donnée est invalidée à la source. Le cas moyen est dépendant des fréquences des accès sur les pairs et de la période de rafraîchissement de la donnée ; il n'est donc pas indiqué ici.

4.5.2 Exclusion mutuelle distribuée

Il existe des algorithmes d'exclusion mutuelle distribués tolérants aux fautes mais ceux-ci ont été conçus pour des réseaux filaires. Ils ne sont donc pas adaptés à une topologie dynamique comme celles des MANets.
Les algorithmes que nous allons présenter sont destinés spécifiquement aux réseaux mobiles ad hoc, et pour la plupart cherche à diminuer le trafic réseau en diminuant le chemin parcouru par le jeton.

4.5.2.1 Un algorithme d'exclusion mutuelle pour MANets

Dans [102], Walter propose un algorithme d'exclusion mutuelle distribuée à jeton circulant le long d'un arbre, inspiré de Raymond.
Cet algorithme construit tout d'abord un graphe acyclique orienté *(Directed Acyclic Graph, DAG)* qui suit la topologie du réseau. Ce graphe a un seul puits, qui est le possesseur du jeton. Ce DAG est construit en attribuant à chaque noeud un poids, qui correspond à la distance au possesseur du jeton. Le possesseur du jeton a donc toujours le poids le plus

TABLE 4.6 – Synthèse des systèmes d'invalidation de cache pour MANet

Travaux	Mécanisme	Hypothèses	Cohérence	Messages
[39]	les messages d'invalidations sont accompagnés d'estampille temporelle que deux pairs peuvent échanger afin de mettre leurs caches à jour sans que la source soit présente	estampille temporelle	Forte pour les pairs en contact avec la source	Pire cas : N invalidation, N pull (2*N) ; Meilleur cas : N invalidations
[44]	La source calcule le nombre probable de demandes de mises à jour jusqu'à la prochaine mise à jour, et si ce nombre est assez élevé elle diffuse la donnée	Historique des accès pour déterminer la probabilité de demande de mise à jour	Forte	Pire cas : N Invalidation, N pull (2*N messages) ; Meilleur cas : N invalidation
[62], POD	à chaque accès, le pair demande à la source si son cache est valide		Forte	Pire cas : infini ; Meilleur cas : 0
[62], MAT	La source diffuse périodiquement une liste des données mises à jour ; en cas de requête, un pair attend la prochaine liste avant de répondre		Forte	Pire cas : N liste mise à jour (L'envoi d'une liste plutôt que de message d'invalidation rend ce coût dépendant du nombre de données) + N pull ; Meilleur cas : 0 (si la période de rafraîchissement de la donnée est plus courte que celle de l'envoi de la liste de mise à jour)
[62], PAT	La source diffuse périodiquement la liste de données invalidées et les K derniers envois sont conservés par les pairs. A partir de ces informations un pair peut savoir depuis quand une donnée est invalide	conservation des listes d'invalidation	Δ	Pire cas : N listes mises à jour (L'envoi d'une liste plutôt que de message d'invalidation rend ce coût dépendant du nombre de données) + N pulls ; Meilleur cas : 0.
[45]	La source détermine un sous -groupe de pairs auxquels envoyer l'invalidation afin de diminuer le trafic	Période de rafraîchissement connue pour chaque donnée	Forte	Pire cas : 0 push, N pulls (2*N messages) ; Meilleur cas : 0 push, 0 pull ;

petit. Par ailleurs, un arbre de couverture de ce DAG est construit où chaque pair choisit comme voisin menant au jeton l'arbre ayant le poids le plus faible.
Quand un pair veut entrer en section critique, il envoie un message le long de l'arbre jusqu'au possesseur du jeton. Quand un pair reçoit une demande de jeton, il stocke l'identité des demandeurs afin de pouvoir leur transmettre le jeton. En sortant de section critique, le pair P possédant le jeton le transmet au premier pair dans la liste de requête, N, puis adapte son poids de manière à ce qu'il soit supérieur à celui de N.
Si un lien de l'arbre disparaît, un pair envoie un message à tous ses voisins dont le poids est inférieur pour retrouver un chemin vers le possesseur du jeton, ou la personne à qui ce jeton doit être transmis. Ses voisins propagent à leur tour un message jusqu'à ce que le pair soit trouvé. La disparition du jeton n'est pas gérée.
Quand un pair perd tout contact avec ses voisins ou quand un nouveau lien est créé, le DAG est recréé partiellement dans la région concernée.

4.5.2.2 Un algorithme d'exclusion mutuelle distribuée pour MANets

Dans [9], Baldoni propose un algorithme d'exclusion à jeton avec un anneau logique pour réseaux mobiles ad hoc qui s'adapte à la mobilité des pairs.
L'algorithme est organisé en tours, au cours duquel un pair est désigné comme coordinateur. Le droit d'entrer en section critique est symbolisé par un jeton qui circule entre les pairs. Au jeton est associé la liste des pairs ayant fait une demande d'entrée en section critique, *pendingRequests*.

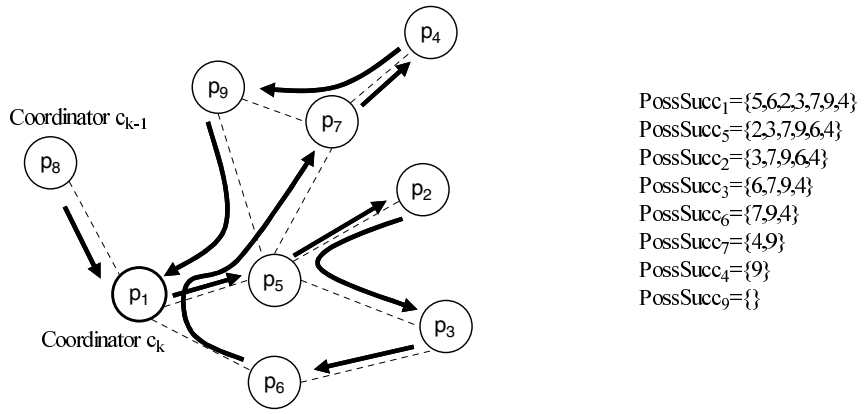


Figure 4.6 – Organisation en anneau logique, extrait de [9]

Un pair entre donc en section critique quand il reçoit le jeton. Quand le pair finit sa section critique, il s'ôte de *pendingRequests* et envoie le jeton au pair de *pendingRequests* qui est le plus proche. Si la liste est vide, il retourne le jeton au coordinateur et celui-ci se place dans un état d'attente. Quand un pair veut entrer en section critique, il envoie une demande au coordinateur puis attend le jeton. Une requête émise au tour t est donc satisfaite au tour t+1.
Quand le coordinateur reçoit une requête, il place le pair dans une nouvelle liste *pendingRequests*. Quand le jeton revient au coordinateur, Il attache cette nouvelle liste au jeton puis transmet le jeton au pair ayant émis une requête le plus proche, qui devient

alors coordinateur (si cette liste est vide, le coordinateur attend une requête). L'ancien coordinateur indique aux autres pairs l'identité du nouveau coordinateur. Commence alors un nouveau tour, et le nouveau coordinateur transmet le jeton au pair le plus proche dans *pendingRequests*, à l'exclusion de l'ancien coordinateur puisque celui-ci vient d'avoir la possibilité de section critique.
La sûreté est garantie par l'existence d'un jeton tandis que la vivacité est garantie par l'organisation en tours.

4.5.2.3 Exclusion mutuelle auto-stabilisante à base de jeton pour MANets

Dans [68], Malpani propose 4 adaptations pour optimiser en terme de messages la circulation d'un jeton sur un anneau logique destiné aux MANets. Cette problématique plus générale est utile dans les algorithmes d'exclusion mutuelle à jeton.
Pour ce faire, deux paramètres sont considérés :
- La fréquence d'accès d'un pair au jeton. Pour cela, le jeton porte un vecteur de compteurs indiquant, pour chaque pair, combien de fois le pair a été visité.
- La fraîcheur d'accès à un jeton. Pour cela, une horloge est attachée au jeton qui est incrémentée à chaque fois qu'il visite un pair. Le jeton porte aussi un vecteur d'horloge indiquant, pour chaque pair, la date à laquelle ce pair a été visité pour la dernière fois.

Les pairs diffusent leur présence à K sauts dans le réseaux, afin que chaque pair puisse connaître ses voisins. Les adaptations proposées sont alors des variations sur la manière de choisir le pair à qui le jeton est passé.
- *Local Frequency* : Un pair choisit comme successeur le noeud le moins visité parmi ses voisins.
- *Local Recency* : Un pair choisit comme successeur le noeud qui a été visité le moins récemment parmi ses voisins.
- *Global Frequency* : Un pair choisit comme successeur le noeud le moins visité.
- *Global Recency* : Un pair choisit comme successeur le noeud qui a été visité le moins récemment.

Notons que les algorithmes *Local Frequency* et *Local Recency* ne garantissent pas la vivacité.
Une amélioration des algorithmes globaux est ensuite proposée : quand le pair a déterminé le pair N le moins visité, ou visité le moins récemment, il choisit comme successeur son voisin direct sur le chemin menant à N.

Dans [21], Chen propose un algorithme d'exclusion mutuelle à jeton basé sur les travaux décrits dans [68].
Leur algorithme se base sur l'algorithme *Local Recency*, où un pair choisit comme successeur le pair dans son entourage ayant été en possession du jeton le moins récemment. Un tel algorithme garantit la propriété de sûreté mais pas de vivacité : selon la mobilité du réseau, il est possible qu'un pair ayant requis une section critique ne reçoive jamais le jeton.

4.5.2.4 Un algorithme d'exclusion mutuelle passant à l'échelle pour MANets

Dans cet article [111], Wu propose un algorithme d'exclusion mutuelle basé sur les permissions et destiné aux MANets.
Chaque pair P maintient les structures de données suivantes :
- *Info_ Set* : l'ensemble des pairs à qui P doit demander la permission s'il veut entrer en section critique.

– *Status_ Set* : l'ensemble des pairs qui, s'ils veulent entrer en section critique, demandent la permission à P.
– *ts* : la date de la demande de section critique, NULL si pas de demande. Elle est utilisée pour calculer la priorité d'un pair quand deux pairs font une requête.
– Q_{req} : la liste des pairs ayant demandé l'entrée en section critique à P.

Ces ensembles sont initialisés de la manière suivante : un pair est choisi comme initiateur. Il crée une matrice M triangulaire haute de taille NxN, N étant le nombre de pairs, et l'initialise en mettant des valeurs 0 ou 1 au hasard dans la partie haute. Cette matrice est ensuite envoyée à tous les pairs qui remplissent la partie basse avec les valeur $m_{i,j} = 1 - m_{j,i}$. Chaque pair i remplit alors *Info_ Set* avec les pairs j tels que $m_{i,j} = 0$ et *Status_ Set* avec les pairs j tels que $m_{i,j} = 1$.

Quand un pair R veut entrer en section critique, il initialise *ts* à la date courante puis envoie un message*(Request)* à l'ensemble *Info_ Set* et attend les réponses. Quand R a reçu l'autorisation *(Reply)* de tous les pairs d'*Info_ Set*, il effectue sa CS. Quand R sort de section critique, il envoie un message *Reply* à tous les pairs dans *Info_ Set*.

Quand un pair P reçoit une requête d'un pair R, il place R dans Q_{req}. Si P n'a pas demandé la CS, ou est moins prioritaire (*ts* plus grand que celui de R), il envoie une autorisation et ôte R de Q_{req}. Si P a demandé la section critique mais est moins prioritaire, il place R dans son *Info_ Set* et envoie un *Request* à P.

Quand un pair veut temporairement arrêter de participer *(doze)* ou se déconnecter, il place l'ensemble des pairs dans *Info_ Set* puis envoie un message aux autres pairs qui le placent alors dans leur *Status_ Set*. Un pair qui se réveille peut donc recommencer l'algorithme sans nécessité d'étape particulière.

4.5.2.5 Un algorithme d'exclusion mutuelle distribuée à deux jetons tolérant aux fautes pour MANets

Dans [110], Wu propose un algorithme d'exclusion mutuelle à jeton avec un anneau logique qui supporte la disparition d'un pair.

Cet algorithme fait usage de deux jetons, un jeton primaire P et un jeton secondaire S et fonctionne par tour.

Chaque jeton est accompagné d'une liste des pairs qu'il n'a pas visités D, d'un numéro de séquence c, du coordinateur actuel et d'un booléen ft indiquant si l'autre jeton est en vie. Chaque pair possède un numéro de séquence ct, qui est le numéro de séquence du dernier jeton qu'il a reçu.

Pour chaque jeton, et pour chaque tour un coordinateur est désigné, choisi comme le pair le plus proche du coordinateur du tour précédent. Les deux jetons circulent donc indépendamment.

Au cours d'un tour chaque jeton visite tous les terminaux. Quand un pair entre en possession du jeton T (P ou C), il vérifie s'il est en possession de l'autre jeton. Si c'est le cas, il incrémente P.c et décrémente S.c. Il compare ensuite T.c et ct. Si ct est différent de T.c, c'est que l'autre jeton a visité le pair depuis le précédent tour. T.ft est donc mis à vrai. Si le jeton est P et qu'il a une demande de section critique, le pair effectue sa CS. Il s'ôte ensuite de la liste D et envoie le jeton au pair le plus proche n'ayant pas été visité. Si D est vide, le jeton est renvoyé au coordinateur.

A la réception d'un jeton, le coordinateur essaye alors de détecter la disparition d'un jeton :
– si T.ft est vrai, c'est que l'autre jeton a visité un pair depuis deux tours.

– si T.ft est faux, l'algorithme considère que si l'autre jeton n'a visité aucun pair depuis deux tours, c'est qu'il a été perdu. Le compteur de T est incrémenté si T=P, décrémenté si T = S et le jeton perdu est recréé.

Il ajoute tous les pairs à D puis désigne son successeur pour un nouveau tour et lui transmet le jeton.

4.5.2.6 Synthèse

Nous avons vu que la majorité des algorithmes proposés sont à jeton. Ceci est dû à la complexité de la constitution de quorum dans un environnement mobile où les terminaux sont amenés à apparaître et disparaître. Ces algorithmes prennent en compte la mobilité des terminaux et visent à minimiser le trafic réseau mais seul [110] gère la possible disparition du jeton.

Le tableau 4.7 résume les différents algorithmes proposés.

TABLE 4.7 – Synthèse des algorithmes d'exclusion mutuelle distribuée pour MANet

Travaux	Type	Sûreté	Vivacité	Adaptation aux MANets
[102]	jeton, arbre	oui, jeton	oui, FIFO	Gestion disparitions de liens
[9]	jeton, anneau logique	oui, jeton	oui, tour	Prise en compte de la proximité
[21]	jeton, anneau logique	oui, jeton	dépend de la mobilité	Choix d'un pair proche comme successeur
[111]	permission	oui	oui, FIFO	Permet les déconnections et la non participation à l'algorithme
[110]	jeton	oui, jeton	oui, tour	Détection perte d'un jeton et régénération

4.5.3 Synthèse des algorithmes pour la mise en place de la cohérence pessimiste

L'implémentation d'un système de cohérence pessimiste pour MANet pose de nombreux problèmes à cause de la mobilité et la volatilité des terminaux.

Dans le cas d'un système d'invalidation de cache, les données et les modifications sont centralisées sur un serveur avant d'être redistribuées. C'est un cas de figure qu'on peut trouver dans certaines applications, par exemple un réseau ad hoc tactique où le serveur est situé sur un véhicule, et les soldats ont chacun un PDA, mais n'est pas dans notre cadre de travail.

Bien qu'un système mettant en place un mécanisme d'exclusion mutuelle distribuée ne soit pas, par nature, centralisé, l'incertitude du nombre de terminaux présents ne permet pas

la mise en place d'un système de quorum, tandis que les algorithmes à jeton nécessitent, comme dans le cas d'un système à copie primaire, de faire la différence entre une partition du réseau et la disparition définitive du porteur du jeton avant d'en générer un nouveau. Dans le contexte de nos travaux, nous pensons qu'il vaut mieux s'orienter vers un modèle de cohérence optimiste.

4.5.4 Cohérence optimiste - Type de donnée répliqué commutatif

Les CRDT *Commutative Replicated Data Type* sont des types de données conçus pour mettre en place un modèle de cohérence
Nous allons dans cette section présenter WOOT et TreeDoc, deux CRDT.

4.5.4.1 WOOT, Wooto

WOOT [78] est une structure de données, avec des algorithmes associés, conçue de manière à ce que les opérations sur la structure soient commutatives.

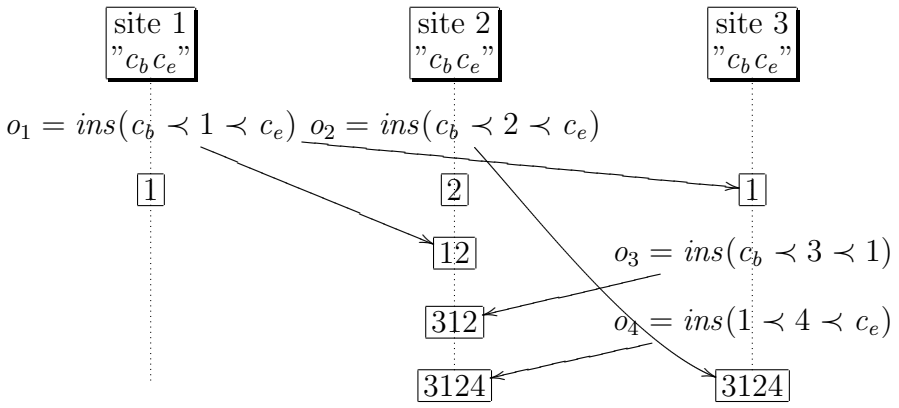


Figure 4.7 – Un exemple d'insertion concurrente dans WOOT, extrait de [78]

Un document est structuré comme une chaîne d'atomes (par exemple la ligne, ou le caractère), chaque atome possédant un identifiant unique. Pour ce faire, l'identifiant d'un atome est constitué du nom du site l'ayant généré, et d'une valeur issue d'un compteur (horloge logique) qui est incrémenté à chaque création d'un atome. Pour un même site, les modifications peuvent donc être ordonnées. Chaque atome est constitué d'un tuple :
– son identifiant unique,
– la valeur de l'atome,
– un booléen indiquant si le caractère est visible,
– l'identifiant de l'atome précédent,
– l'identifiant de l'atome suivant.
Deux opérations sont possibles :
– insertion entre deux positions : un nouvel atome est inséré entre deux identifiants A et B. Si ces deux identifiants ne sont pas côte à côte, c'est qu'il y a déjà eu en parallèle, une insertion entre eux. Les atomes entre A et B sont alors placés de manière à être triés selon l'ordre de leurs identifiants. La figure 4.7 représente un exemple d'insertion en parallèle.

– suppression d'une position : l'atome n'est pas effacé, mais le caractère devient invisible. L'opération est donc triviale. Un atome ne peut pas être réellement supprimé de la chaîne, car des opérations d'insertion l'impliquant peuvent être effectuées en parallèle.

Les opérations ne sont pas totalement commutatives et ne peuvent pas être effectuées dans un ordre quelconque : pour supprimer un atome, il faut que l'opération d'insertion de cet atome ait été effectuée. De même, pour effectuer une insertion, il faut que les atomes suivant et précédent existent. Il existe donc un ordre partiel sur les opérations (ordre causal).
Quand un site reçoit une opération qu'il ne peut pas effectuer, elle est mise en attente et appliquée dès que ses pré-conditions (opérations précédentes selon l'ordre partiel) ont été satisfaites.
L'inconvénient de cette approche est que comme les atomes supprimés, appelés *tombstones*, ne sont pas effacés, la donnée grandit en taille sans limitation. Cependant, il serait possible de mettre en place une étape de synchronisation qui permettrait l'exécution d'un ramasse-miette distribué et éliminerait ainsi les caractères invisibles. Pour effectuer cette opération, il faut s'assurer que tous les hôtes aient effectué le même ensemble d'insertions et de suppressions, et soient tous présents, puis interdire les écritures sur la donnée jusqu'à ce que tous les sites aient signalés que l'opération de nettoyage s'est bien terminé, avec un protocole de validation en 2 phases.

4.5.4.2 TreeDoc

TreeDoc [86] est un type de donnée répliquée, où une donnée est structurée comme un arbre binaire.

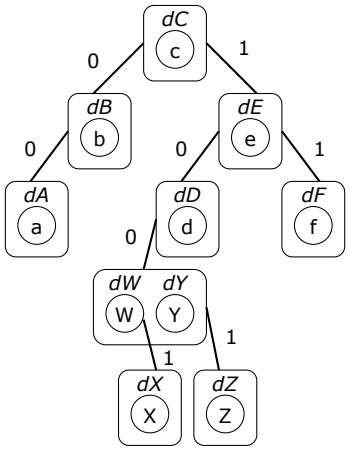


FIGURE 4.8 – Structure d'un arbre TreeDoc, avec à 100 un mininoeud créé par une insertion concurrente, extrait de [86]

Chaque noeud de l'arbre représente un atome, comme illustré par 4.8. Un noeud est composé des éléments suivants :
– la valeur de l'atome,
– une référence au sous-arbre gauche qui représente l'ensemble des atomes précédant le noeud dans l'ordre de la donnée,
– une référence au sous-arbre droite qui représente l'ensemble des atomes succédant au noeud,

– un booléen indiquant si le caractère est visible,
– l'identifiant du site ayant créé l'atome, appelé désambiguateur, et utilisé en cas d'insertions concurrentes.

L'identifiant unique d'un atome est son chemin dans l'arbre, soit une succession de bit 0 (gauche) ou 1 (droite).

Afin de pouvoir passer d'une chaîne d'atomes à l'arbre et inversement, deux opérations sont nécessaires :

– *explode* : transforme une chaîne de caractère en arbre binaire équilibré tel que, pour tout noeud, le sous arbre gauche représente l'ensemble des caractères précédant la valeur de l'atome dans la chaîne de caractères, et le sous arbre droit l'ensemble des caractères succédant la valeur de l'atome dans la chaîne de caractères.
– *flatten* : parcourt l'arbre en ordre infixe pour recréer le contenu de la donnée représentée par l'arbre.

Ces opérations sont aussi utilisées, sous certaines conditions, pour nettoyer l'arbre, comme nous le verrons plus bas.

Deux opérations d'éditions de base sont proposées :

– *insert(pos, atome)* : insertion à une position d'un atome. Un nouveau noeud est créé à la position indiquée et l'atome est inséré. Dans le cas de deux éditions au même endroit de l'arbre, un mini noeud est créé contenant tous les atomes insérés au même endroit et triés grâce au désambiguateur.
– *del(pos)* : suppression d'une position. Tout comme dans WOOT, l'atome n'est pas supprimé mais marqué comme invisible.

Les modifications étaient effectuées sur une chaîne de caractères, au moment d'une insertion une opération *newPosID* permettant de créer l'identifiant du nouvel atome, et indiquer une position à *insert*, est donc nécessaire.

Comme dans WOOT, il existe un ordre partiel sur les opérations. Les opérations non applicables sont donc conservées jusqu'à ce que leurs pré-conditions soient remplies.

Tout comme pour WOOT, les atomes invisibles ne sont jamais effacés, à moins de mettre en place un ramasse-miette distribué. La taille du document grandit donc potentiellement sans limitation. Si l'opération de barrière, permettant à tous les pairs de se synchroniser pour effectuer un ramasse miette distribué, est mise en place, un *flatten* suivit d'un *explode* permettent non seulement d'enlever les atomes invisibles, mais aussi de rééquilibrer l'arbre binaire, ce qui diminue la taille des identifiants.

4.5.4.3 Synthèse

L'intérêt de TreeDoc par rapport à WOOT est un gain de place mémoire. En effet, dans TreeDoc , les identifiants ne sont pas explicites mais induits par la structure de l'arbre.

Par ailleurs, dans TreeDoc la taille moyenne des identifiants, qui sont échangés lors des envois de mise à jour, dépend de la structure de l'arbre : si l'arbre est une chaîne, les identifiants sont plus longs que si l'arbre est équilibré. Une phase de rééquilibrage de l'arbre est proposée, afin de limiter la taille des identifiants, alors que l'opération de nettoyage de WOOT ne permet pas de gain à ce niveau. Cette opération nécessite cependant une validation en 2 phases.

4.6 Synthèse

Dans ce chapitre nous avons présenté tout d'abord des systèmes de partage de données pour MANet existants, puis des systèmes de cache de données. Nous avons ensuite présenté les solutions apportées à plusieurs problèmes, à savoir : la création de grappes, les méthodes de réplications, et la mise en oeuvre de la cohérence des données, par invalidation du cache, exclusion mutuelle distribuée, et CRDT.
Parmi les algorithmes de créations de grappes, nombre d'entre eux s'appuient sur un module GPS, pour gérer la mobilité des terminaux, ceci parce que ces solutions sont à destination de VANets. De plus, certaines solutions sont proposées comme un problème mathématique de graphes sans proposer d'implantation concrète.
Les algorithmes de réplications proposés reposent souvent sur des hypothèses fortes sur les données : données non éditables, données mises à jour périodiquement, corrélation entre les données connues, ou fréquence d'accès de chaque donnée par chaque pair connue.
Pour la mise en place de la cohérence, nous avons tout d'abord vu les techniques d'invalidation de cache : celles-ci ne sont pas destinées à des données modifiables par de multiples éditeurs, et ont une seule source de modification, ce qui n'est pas ce que nous cherchons à mettre en place. Cependant, ces algorithmes pourraient être adaptés à un système à copie-maître.
Nous avons ensuite vu les algorithmes d'exclusion mutuelle distribuées pour MANets. La majorité sont des algorithmes à jeton, car la construction de quorum dans un réseau dynamique est problématique. Ces algorithmes tentent de s'adapter aux MANets en diminuant le parcours effectué par le jeton, mais un seul propose un modèle de recréation du jeton. Les partitions du réseau ne sont pas gérées.
Enfin, nous avons vu les types de données commutatifs répliqués WOOT et TreeDoc, qui permettent de mettre en place une réplication optimiste des données. Nous pensons que les CRDT sont donc la solution la plus adaptée dans le contexte de ces travaux.

Deuxième partie

Contribution

Positionnement

Les algorithmes de gestion de grappes présentés dans la section précédente dépendent pour la plupart d'un système de positionnement relatif ou absolu, nécessitant donc soit un GPS, soit un calcul de vitesse relative basée sur la puissance du signal reçu. Par ailleurs, le coût de ces propositions en surcharge réseau n'est pas évalué. **Notre proposition n'utilise pas de système dédié au positionnement, et, en faisant usage d'information inter-couche, ne crée pas de charge réseau supplémentaire**.
Nous utilisons les grappes pour mettre en place des **politiques de gestion des données collaboratives**.
Le modèle de réplication des données le plus utilisé est la réplication à la demande sous condition (par exemple, on ne réplique pas si la donnée nous est servie par un terminal proche). Cette technique permet de limiter le coût de maintenance, mais peut rendre les données indisponibles en cas de partition. Nous nous intéressons donc plutôt aux autres modèles proposés, visant à répliquer pro-activement les données, c'est à dire avant que l'utilisateur ne demande à y accéder.
Nous proposons **un algorithme de réplication pro-active collaboratif au sein d'un groupe de pairs stable faisant usage d'informations sémantiques** pour placer les répliques.
Nous proposons aussi de le coupler à **un algorithme de remplacement de cache prenant en compte le nombre de répliques existantes, et agissant de manière préventive pour diminuer la charge réseau.**
De nombreux systèmes se proposent de faire de la cohérence optimiste, ou de la cohérence hybride. En pratique, ils reposent sur des algorithmes de réconciliation qui peuvent éventuellement nécessiter l'intervention d'un humain. A l'issue d'une réconciliation, par exemple, dans le cas de deux groupes se croisant assez longtemps pour propager les modifications, puis où la réconciliation a lieu en parallèle dans chaque groupe, on peut même arriver après réconciliation à deux versions différentes.
Pour maintenir la cohérence des données, nous faisons le choix de **structurer les données avec TreeDoc** afin de mettre en place un modèle de cohérence optimiste et d'éviter ainsi les problèmes liés à la réconciliation manuelle des données.
Nous voyons tout d'abord un algorithme de construction de groupes stables qui tire partie des tables de routage. Nous présentons ensuite l'algorithme de réplication lui-même, avant de voir sa contrepartie, la gestion du remplacement de cache.
Nous verrons ensuite une proposition de démonstrateur sous la forme d'un moteur de wiki pair à pair pour réseaux mobiles ad hoc.

Chapitre 5

Algorithme de création de groupes stables

Dans ce chapitre nous présentons un algorithme de création de groupes stables dans le temps au sein d'un réseau mobile ad hoc.

Dans les MANets, des événements tels que la disparition ou l'apparition de pairs et la partition du réseau, surviennent de manière fréquente.
C'est un problème quand on veut développer des applications distribuées pour réseaux mobiles :
– on ne peut pas considérer que l'ensemble des terminaux participants vont rester présents,
– on ne peut donc pas considérer la disparition d'un terminal comme un événement exceptionnel et permanent.

Dans ce chapitre nous présentons donc notre approche pour gérer la mobilité.
Nous voyons tout d'abord nos hypothèses de travail concernant la nature du réseau et le comportement des pairs le constituant. Nous motivons ensuite le choix de conception d'utiliser des informations inter-couches et présentons les éléments du protocole de la couche réseau avec laquelle notre algorithme interagit, OLSR [22].
Dans les sections suivantes, nous présentons les structures de données utilisées puis l'algorithme lui même, illustré par un exemple, avant de discuter du paramètrage de cet algorithme.
Enfin, nous détaillons notre méthode de validation avant de présenter nos résultats puis de conclure.

5.1 Hypothèses de travail

Dans cette proposition, nous faisons différentes hypothèses sur le comportement des utilisateurs, les protocoles réseaux utilisés par les terminaux, ainsi que sur les communications. Nous les présentons dans cette section.

5.1.1 Utilisateurs

Notre proposition est destinée à des utilisateurs piétons, qui collaborent via des terminaux mobiles équipés de cartes wifi et de protocoles réseaux ad hoc. Ils se déplacent en groupe et leur organisation peut être décrite par le modèle de mobilité *Reference Point Mobility Group* (RPGM) [41]. Leur vitesse varie entre 0 km/h et 4km/h.

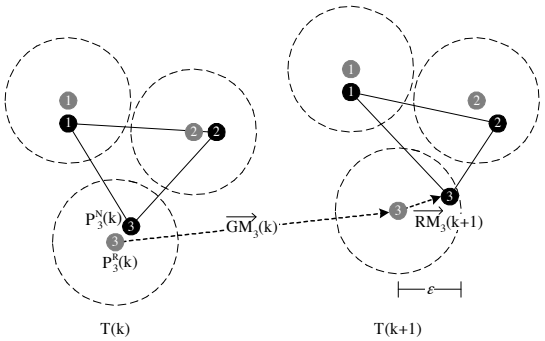


Figure 5.1 – Simulation de la mobilité des terminaux avec le modèle Reference Point Mobility Group, le RP est représenté ici en gris, extrait de [41]

Dans le modèle RPGM, les pairs se déplacent en un groupe, dont les mouvements sont décrits par un point de référence RP. Les terminaux se placent de manière à être à portée

de communication du RP, comme illustré par 5.1. Ce modèle est utilisé dans d'autres travaux proposant des algorithmes de construction de groupes mobiles, comme [43].
Par construction, la topologie du réseau est donc confinée dans un disque de 2 portées de communications de large, décrit par deux terminaux à portée maximale de communication du RP et situés à des positions diamétralement opposées . Cependant, comme il peut être plus économique d'effectuer deux communications de courte portée plutôt qu'une longue, les routes construites ne sont pas nécessairement de longueur inférieure à 2.

5.1.2 Protocoles réseaux et communications

Les terminaux des utilisateurs sont équipés de carte wifi utilisant 802.11b, la norme wifi la plus couramment utilisée. La portée de communication maximale théorique en extérieur est de 100m.
Les communications sont symétriques : à un instant, si A est à portée de communication de B, alors B est à portée de communication de A.
Nous faisons l'hypothèse d'un protocole de routage pro-actif, qui maintient les routes en l'absence de messages échangés et nous avons basé nos expérimentations sur le protocole OLSR [22]. Celui ci est présenté plus en détail dans la section suivante. Nous verrons notamment que OLSR construit des groupes de terminaux dont les communications sont symétriques.
Cette hypothèse est nécessaire car ici l'algorithme proposé utilise des informations inter-couche, issues de la couche de routage. Dans la section suivante, nous expliquons ce choix.

5.1.3 Définition d'un groupe stable

L'algorithme proposé cherche à construire des groupes de terminaux, les voisins, stables dans le temps.
Nous définissons un groupe stable comme un groupe de terminaux capables de communiquer en continu sur une période.
En d'autres termes, pour être des voisins stables à t, A et B doivent avoir été en contact depuis au moins δ_p secondes. Si par la suite ils ne peuvent pas établir le contact pendant δ_f secondes, ils arrêtent d'être voisins. On considère donc que A est capable de communiquer continuellement avec B au temps t si :
– B a reçu tous les messages diffusés par A depuis t- δ_p
– B recevra tous les messages diffusés par A depuis t et t+ δ_f
Si ces conditions sont vérifiées, B est un voisin stable de A et comme les communications sont par hypothèse symétriques, A est un voisin stable de B.
Le groupe stable de P est donc l'ensemble des pairs-voisins stables de P.

5.2 Choix de conception : information inter-couche

5.2.1 Motivation : optimisation des ressources

On préfère habituellement (e.g. pile OSI) maintenir chaque tâche différente dans un couche séparée, afin d'être modulable. Cependant l'utilisation d'information inter-couche *(cross-layering)*, permet d'optimiser l'utilisation des ressources en évitant de reconstruire plusieurs fois la même information.
Cette technique est très utilisée dans les protocoles pour réseaux mobiles ad hoc.

C'est le cas par exemple de Chapar [52] le système d'événements développé pour Transhumance [81]. Il utilise les nœuds MPRs d'OLSR, que nous présentons plus bas, comme distributeurs d'événements en se basant sur leur propriété d'accessibilité par les autres membres du réseau.
Plusieurs algorithmes de création de grappes, que nous avons présentés dans l'état de l'art, utilisent la puissance du signal reçu, une information issue de la couche physique, pour déterminer la distance à l'émetteur.
Dans nos travaux, nous proposons un protocole de niveau applicatif qui interagit avec la couche de routage OLSR pour y récupérer des informations de présence.

5.2.2 Un algorithme de routage pro-actif : OLSR

Dans un réseau, un protocole de routage permet de choisir un chemin, c'est à dire une suite de nœuds du réseau, pour acheminer un paquet entre la source d'un message et son destinataire.
Notre travail s'inscrit dans les travaux de recherche menés lors du projet ANR Transhumance, destinés à produire un intergiciel pour MANets.
La proposition d'un nouvel algorithme de routage ne relevant pas du projet Transhumance, les membres du projet ont été amenés à choisir un algorithme de routage, le protocole et son implantation se devant d'être complètement validés.
Bien que de nombreux algorithmes aient été proposés dans la littérature, la plupart n'offrent pas d'implantation et sont validés par simulation uniquement. Ceux qui ont été implantés l'ont souvent été à des fins de prototypages, et ne sont pas utilisables. Seuls les protocoles DSR [49], AODV [83] et OLSR [22] bénéficient de véritables implantations, et le choix de l'équipe Transhumance s'est porté sur OLSR du fait de sa solide implantation OLSR-Unik [65], et de sa communauté active, notamment sur la liste IETF-manet.
OLSR est un protocole de routage pro-actif à état de lien proposé par le projet HIPERCOM-INRIA en 2001. Dans un protocole à état de lien, comme OSPF [76] - qu'on oppose aux protocoles à vecteurs de distances, comme RIP [40] et aux protocoles à vecteurs de chemin comme BGP [64] - tous les terminaux inondent le réseau avec la liste de leurs voisins, afin que chacun puisse reconstituer le graphe de routage.
Le terme pro-actif signifie que les routes sont maintenues, par opposition aux protocoles réactifs, tel que AODV, où les routes sont construites à la demande. Il existe d'autres types de protocoles, mais ces deux classes sont les plus génériques. Les deux approches ont leurs avantages et leurs inconvénients, et [46] montre que pour un trafic sporadique, les algorithmes pro-actifs sont plus adaptés.
Les travaux dans lesquels s'inscrit notre algorithme visent au support d'applications collaboratives sur MANet, et le trafic, étant généré par des utilisateurs humains, est donc sporadique.

5.2.2.1 Structures de données d'OLSR

OLSR maintient sur chaque terminal une table de routage construite, comme nous le verrons dans la section suivante, grâce aux messages TC. Ces tables contiennent une liste de tuples, chacun contenant les informations suivantes :
– adresse destination R_dest_addr.
– prochaine adresse R_next_addr : adresse à laquelle il faut envoyer le message pour qu'il soit routé jusqu'à R_dest_addr.

– nombre de sauts *R_ dist* : la distance en terme d'arcs dans le graphe de routage entre le terminal et *R_ dest_ addr*.
– adresse de l'interface locale *R_ iface_ addr*.

Par ailleurs, grâce aux messages HELLO, OLSR construit et maintient, comme nous le verrons dans la section suivante :

– l'ensemble N de ses voisins à 1 saut,
– l'ensemble *N2* de ses voisins à 2 sauts,
– l'ensemble *MPR Selector* qui contient la liste des terminaux ayant sélectionné le terminal local comme MPR

Par la suite, dans nos travaux nous utilisons des informations issues des tables de routages.

5.2.2.2 Fonctionnement d'OLSR

OLSR établit tout d'abord la topologie du réseau. Tout d'abord, l'ensemble des communications possibles entre terminaux, est calculé. A partir de cet ensemble, OLSR établit ensuite le plus court chemin entre chaque paire de nœuds. Ceci est fait sur chaque terminal et les terminaux doivent divulguer la liste de leurs voisins à portée de communication à l'ensemble du réseau.

L'idée derrière OLSR est de limiter le nombre de terminaux envoyant des messages d'inondation, en élisant des pairs relais, nommés MPR *(MultiPoint Relays)* : quand un pair veut communiquer, en diffusion ou en point à point, il doit envoyer son message à travers un des MPR, seuls pairs autorisés à faire suivre des messages dans le réseau. Un MPR est par construction un voisin à 1 saut, comme nous allons le voir par la suite.

Quatre types de messages sont utilisés : HELLO, TC (Topology Control), MID (Multiple Interface Declaration) et HNA (Host and Network Association). Nous allons présenter brièvement le fonctionnement d'OLSR et les messages MID et HNA ne seront donc pas détaillés.

Les messages HELLO servent tout d'abord à détecter le voisinage d'un pair :

– périodiquement, chaque pair P diffuse à un saut un message de HELLO, avec la liste L de ses voisins P_i et en indiquant si le lien est symétrique (P_i reçoit les messages de P) ou asymétrique (à la connaissance de P, P_i ne reçoit pas ses messages). Cette liste est au départ vide.
– quand un pair P_1 reçoit un message de HELLO de P_2, il enregistre P_2, comme un voisin, asymétrique.
– quand un pair P_1 reçoit un message de HELLO de P_2 avec sa liste de voisin L_2, et que P_1 se trouve dans L_2, il enregistre P_2, comme un voisin symétrique, puisque P_2 voit les messages de P_1.

Un message HELLO émis par P contient donc entre autres informations :

– sa liste de voisins, indiquant pour chacun :
  – si la connexion est symétrique, asymétrique, ou si elle a disparu.
  – si le voisin a été sélectionné comme MPR pour le terminal.
– sa disponibilité à agir lui-même en tant que MPR, avec 5 degrés de disponibilité : *Never, Low, Default, High, Always*.

Ces messages HELLO sont ensuite utilisés pour élire les MPR. Suite à l'échange de HELLO, chaque terminal connaît l'ensemble N de ses voisins symétriques à 1 saut et l'ensemble *N2* de ses voisins symétriques à 2 sauts. Il applique ensuite l'heuristique suivante afin de sélectionner la liste des MPR :

1. Initialiser l'ensemble MPR avec les membres de N (voisins à 1 saut) acceptant de jouer le rôle de MPR avec une disponibilité *Always*.
2. Pour chaque terminal y dans N, calculer le degré $D(y)$, c'est à dire le nombre de ses voisins à un saut symétriques :
   – qui ne sont pas voisins à 1 saut du terminal local (donc pas dans N : en calculant 2 sauts, on n'emprunte pas deux fois le même lien).
   – qui sont donc par construction à deux sauts du terminal local.
3. Pour tout terminal dans *N2*, si un seul terminal t dans N permet la communication, t est ajouté à l'ensemble MPR, quelle que soit sa disponibilité.
4. Tant qu'il existe un terminal dans *N2* auquel on ne peut pas accéder via un terminal dans l'ensemble MPR :
   (a) Pour chaque terminal t de N, pas encore MPR, on calcule son accessibilité, c'est à dire le nombre de terminaux dans *N2*, non accessibles via un MPR existant, auxquels on pourrait accéder si t devenait MPR.
   (b) On sélectionne comme MPR le terminal avec la plus haute disponibilité, et qui a une accessibilité non nulle. En cas de possibilités multiples, on sélectionne le terminal qui présente la plus forte accessibilité. S'il reste encore des possibilités multiples, on choisit le terminal qui possède le degré D le plus élevé.
5. Une fois construit l'ensemble MPR, une optimisation possible est de classer les MPR par disponibilité ascendante et de vérifier tour à tour si on peut retirer un MPR tout en continuant à joindre tout N2. Le cas échéant, ce MPR peut être retiré de l'ensemble MPR.

Cet algorithme est une heuristique permettant de construire l'ensemble MPR, et d'autres heuristiques ont été proposées pour améliorer OLSR [31].

Les messages TC sont utilisés pour construire la topologie du réseau. Un message TC contient la liste des liens du terminal, et un numéro de séquence afin de pouvoir discriminer entre deux messages lequel est le plus récent (l'autre n'étant donc pas pris en compte). Ces messages sont envoyés périodiquement, avec par défaut une période de 5s. Ils sont diffusés via les MPR.
A partir des messages TC, chaque terminal peut reconstruire la topologie du réseau. Il applique ensuite un algorithme de type Dijkstra ou Bellman-Ford pour construire le plus court chemin entre lui et chacun des terminaux, et peuple ainsi les tables de routages.

5.3 Structures de données de l'algorithme proposé

Pour exécuter cet algorithme chaque terminal T maintient deux structures de données résumant la connaissance qu'on a des autres pairs :
– *stableNeighbourhood* qui contient une liste d'adresses IP représentant les pairs appartenant au groupe stable autour de T. C'est cette structure qui représente l'information utilisée par les applications des couches supérieures.
– *heardOf* qui associe à une adresse IP un compteur de présence. Elle contient la liste des pairs dont on a récemment entendu parler.
La section suivante présente la manière dont ces structures sont gérées.

Nous faisons usage des tables de routage OLSR pour déterminer la liste des destinations possibles, *Liste_destinations*.
Les tables de routages sont lues depuis la couche de routage périodiquement. L'information est récupérée sous la forme d'un ensemble de paires (IP, distance), retourné par la fonction *getRoutingTableFromRoutingLayer*.

5.4 Algorithme

La figure 5.2 présente le code de notre algorithme en python.
Rappelons les opérateurs ensemblistes en python :
– A - B : éléments appartenant à l'ensemble A et n'appartenant pas à l'ensemble B
– A & B : intersection des ensembles A et B
Cet algorithme est exécuté périodiquement et se comporte de la manière suivante, en deux phases :

1. Phase d'observation : quand un pair P arrive en vue, on lui associe un compteur, nommé *Presence Counter*, ou PC, initialisé à 1 et stocké dans *heardOf*. Chaque période on vérifie si P est présent. Si c'est le cas, PC est incrémenté (ligne 28), et quand PC dépasse le seuil de stabilité *stableThreshold*, il devient membre du groupe stable (ligne 34). Sinon, *PC* est décrémenté (ligne 25), et quand il atteint 0, on arrête d'observer P (ligne 23).
   Pour devenir un voisin stable, P doit donc être présent pendant au moins *stableThreshold* périodes + le nombre d'absences depuis le début de l'observation.
2. Quand le compteur de présence d'un pair a atteint *stableThreshold*, il devient un voisin stable, et on entre donc dans la phase de maintenance
3. Phase de maintenance : lors de cette phase, on continue d'observer P à chaque période et de modifier PC en fonction de sa présence ou de son absence. *PC* peut être incrémenté jusqu'à un second seuil, *stableThreshold + maxAbs*, suite à quoi il est plafonné à cette valeur. Si *PC* passe en dessous du premier seuil de stabilité *stableThreshold*, il est sorti du groupe stable.
   On tolère donc un certain nombre d'absences sporadiques consécutives de la part du pair avant qu'il soit retiré du groupe stable.

5.5 Exemple

Nous présentons ici un exemple de fonctionnement de l'algorithme proposé, illustré par la figure 5.3. Ici, le seuil de stabilité est fixé à 5, et le nombre maximum d'absences tolérées à 3. L'algorithme est exécuté sur le terminal A, qui examine le comportement du terminal P. Le compteur de présence associé par A à P est appelé PC.
La première ligne représente le temps en nombre de périodes de l'algorithme, la seconde ligne indique si à cette période le pair est en contact, et la troisième ligne est l'évolution de PC.

1. Avant t=0, P n'est pas connu de A.
2. De t=0 à t=2, PC est incrémenté.
3. À t=3, P est absent, donc son compteur de présence PC est decrémenté.
4. De t=4 à t=6, PC est incrémenté ; à t=6, PC atteint la valeur *stableThreshold* et P devient un voisin stable de A.

```
PERIOD_SEC=2
stableThreshold = 120
maxAbs = 60

class Peer :
    def __init__(self, group, filename):
        self.group = group
        self.heardOf =dict()
        self.stableNeighbourhood=[]

def updateHeardOf(self, routes) :
    heardOfSet= set(self.heardOf.keys())
    routesSet= set(routes.keys())
    absent = heardOfSet-routesSet
    stillpresent= routesSet & heardOfSet
    newcomers = routesSet-heardOfSet

    for p in newcomers :
        self.heardOf[p]= 1

    for p in absent :
        if self.heardOf[p] == 1 :
            del self.heardOf[p]
        else:
           self.heardOf[p]= self.heardOf[p] - 1

    for p in stillpresent :
        if self.heardOf[p]<stableThreshold+maxAbs :
            self.heardOf[p]=self.heardOf[p]+1

def buildStableGroup(self):
    self.stableNeighbourhood=[]
    for p in self.heardOf.keys():
        if self.heardOf[p]>stableThreshold :
            self.stableNeighbourhood.append(p)

def buildStableGroup(self) :
    while true:
        routes = getRoutesFromRoutingLayer()
        self.updateHeardOf(routes)
        self.buildStableGroup()
        sleep(PERIOD)

```

Figure 5.2 – Algorithme de construction de groupe stable

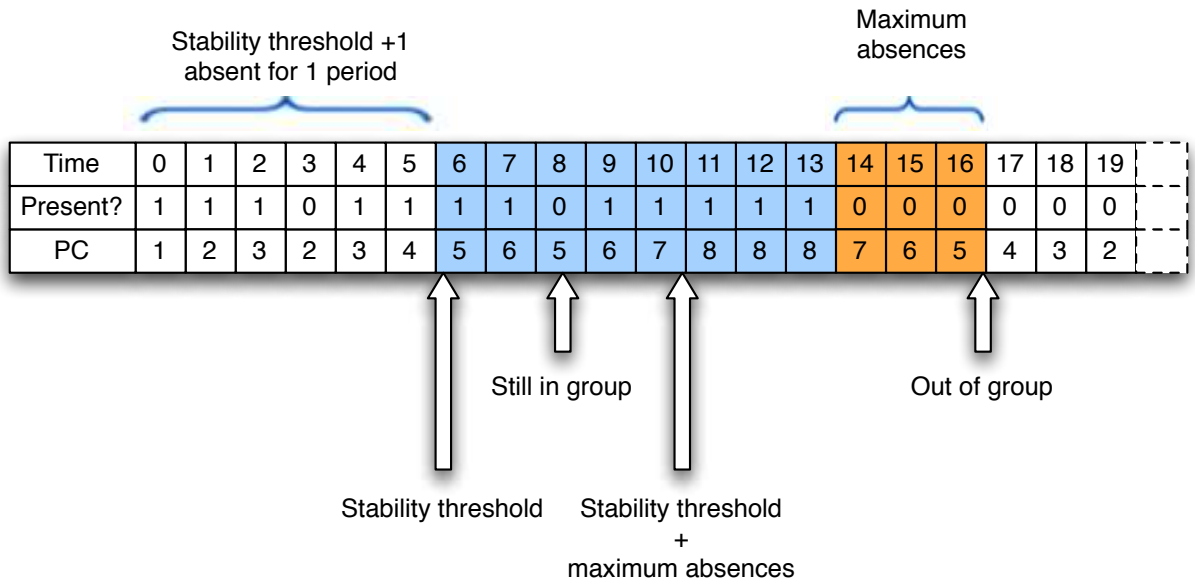


FIGURE 5.3 – 1 group

5. De t=6 à t=7, PC est incrémenté.
6. À t=8, P n'est pas accessible, PC est donc decrémenté ; P reste dans le groupe : une absence sporadique est tolérée.
7. De t=9 à t= 13, PC est incrémenté jusqu'à atteindre la valeur *stableThreshold+maxAbs* ; il se stabilise.
8. À t=15, P disparaît ; PC est décrémenté, mais P reste dans le groupe.
9. de t=14 à t=16, PC est décrémenté.
10. À t=18, PC passe en dessous de *stableThreshold* : le nombre maximum d'absences *maxAbs* tolérées est dépassé ; P est donc sorti du groupe.

5.6 Paramètres de l'algorithme

On peut voir que différents paramètres influencent le comportement de l'algorithme proposé :
– le taux de rafraîchissement, représenté par la période *PERIOD* (ligne 42),
– le seuil de stabilité que le pair doit atteindre pour être considéré comme un voisin stable, représenté par *stableThreshold* (ligne 28 et ligne 34),
– le nombre maximum d'absences tolérées avant qu'un pair ne soit sorti du groupe stable, représenté par *maxAbs* (ligne 28).
Dans cette section, nous présentons la signification physique de chaque paramètre et nous voyons comment choisir leurs valeurs.

5.6.1 Période de rafraîchissement du voisinage

L' algorithme proposé est périodique et cherche à prédire quels pairs seront présents à la période suivante, en se basant sur les périodes passées. On veut donc connaître la probabilité

que le statut d'un pair change entre deux échantillonnages.
La période d'envoi de messages HELLO permettant de rafraîchir les tables de routage est, par défaut, de 2 secondes. Notre algorithme se basant sur le contenu des tables de routage pour déterminer qui se trouve à portée, une période inférieure à 2s est donc inutile.
Nous avons donc fixé *la période de rafraîchissement* du voisinage à **PERIOD=2s**.
Soit un pair A du groupe stable au temps t, moment de l'échantillonnage. Nous voulons évaluer la probabilité que ce pair reste joignable durant l'intervalle [t ; t+PERIOD].
Voyons tout d'abord ce qui se passe pour un terminal à portée de communication, comme illustré par la figure 5.4. On considère une portée de communication R comprise entre $r_{min} = 30m$ et $r_{max} = 100m$. La vitesse du pair est v=1m/s. Dans le pire des cas, si deux pairs se déplacent dans des directions opposées, la distance entre eux va donc grandir de 4m.
Si un pair est à portée de communication, deux cas sont possibles :
– le cas A : il est à une distance inférieure à R-4, auquel cas il ne perdra pas contact
– le cas B : il est à une distance supérieure à R-4, et se trouve donc dans la zone bleue de la figure 5.4. Dans ce cas, selon son déplacement, il risque de se retrouver dans la zone orange où la communication sera perdue.

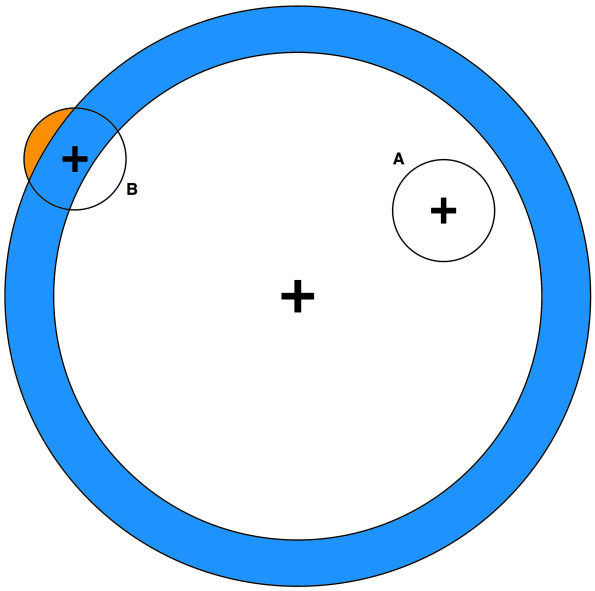


Figure 5.4 – Période de rafraîchissement du voisinage

Nous avons évalué la probabilité de garder le contact par la méthode de Monte Carlo [92]. Pour chaque portée R de communication entre 10m et 100m (pas de 10m), on effectue 10^9 fois cette expérience :

1. Soit un repère orthogonal, avec le terminal C positionné à l'origine.
2. On place aléatoirement un autre pair D à une position P, telle que la distribution des positions sur le disque de rayon R et de centre (0,0) soit uniforme. D est donc à portée de communication de C. Pour cela, on génère aléatoirement des coordonnées polaires, avec les lois de probabilité suivantes :
   – l'azimut ρ suit une distribution uniforme continue sur l'intervalle [0 ;2π]

- la coordonnée radiale θ suit une distribution de loi $R * \sqrt{(X)}$, avec X une v.a. suivant une loi uniforme continue sur l'intervalle [0 ;1].

3. Si θ est inférieur à R-4, quels que soient les déplacements effectués par C et D, ils resteront à portée de communication.
4. Sinon, on génère le déplacement de D pendant le temps PERIOD. Rappelons qu'on travaille dans un référentiel de centre C, et que D peut donc se déplacer d'au plus 4m :
   (a) comme à l'étape 2, on génère une coordonnée polaire distribuée uniformément dans le disque de centre P et de rayon 4.
   (b) on calcule la distance [CD] suite au déplacement : si celle-ci est supérieure à R, le contact est perdu.

Pour chaque valeur de R, nous avons comptabilisé le nombre de fois où le terminal reste à portée, puis avons calculé la probabilité recherchée. Les résultats sont présentés dans le tableau 5.5.

Portée, en m	10	20	30	40	50	60	70	80	90	100
Probabilité, en%	92.88	96.64	97.80	98.36	98.69	98.91	99.07	99.19	99.28	99.35

FIGURE 5.5 – Probabilité de contact au temps T+1, selon la portée de communication, pair à portée de communication

Cette probabilité concerne les terminaux à portée de communication. Nous faisons l'hypothèse que les terminaux se déplacent suivant le modèle RPGM, et nous avons vu que, dans ce cas, le réseau fait 2 portées de communication de large. Les routes peuvent cependant être plus ou moins longues selon la densité du réseau.
Dans le pire cas, deux terminaux A et C ne peuvent communiquer qu'à travers B. La probabilité que A et C gardent contact est donc la probabilité que A et B restent en contact et que B et C restent en contact. On a donc résumé dans le tableau 5.6 les probabilités de perdre le contact avec un membre du groupe.

5.6.2 Limite de stabilité

Le seuil de stabilité *stability threshold* indique pendant combien de périodes de l'algorithme un pair doit être vu avant de pouvoir être considéré comme faisant parti du voisinage stable. C'est donc une approximation de δ_p, le nombre de secondes pendant lesquelles deux pairs doivent être capables de communiquer avant de devenir voisins stables.
Dans notre algorithme, deux pairs deviennent voisins après *stableThreshold + le nombre de périodes durant laquelle la communication était perdue**PERIOD secondes. La durée entre le premier contact et le moment où deux pairs deviennent voisins stables est donc supérieur ou égal à *stableThreshold**PERIOD.
L'algorithme proposé doit différencier deux groupes qui se croisent de deux groupes qui fusionnent. Le seuil *stabilityThreshold* doit donc être suffisamment élevé pour ce faire, mais

Portée, en m	10	20	30	40	50	60	70	80	90	100
Probabilité, en %	86.26	93.39	95.64	96.74	97.39	97.83	98.14	98.38	98.56	98.70

FIGURE 5.6 – Probabilité de contact au temps T+1, selon la portée de communication, cas général

assez bas pour que la collaboration puisse commencer le plus rapidement possible en cas de fusion.
Le temps de contact entre deux groupes mobiles varie en fonction de l'angle α de leurs trajectoires. Dans le meilleur des cas, les deux groupes vont dans des directions opposées. Dans le pire des cas, ils empruntent des trajectoires quasi parallèles et reste donc en contact longtemps.
On veut pouvoir distinguer deux groupes se croisant avec des trajectoires orthogonales ($\frac{\pi}{2} rad$) ou un angle plus grand.
Comme nous supposons un modèle de mobilité de type RPGM, et une portée de communication maximale R= 100m, le diamètre de l'aire maximale de couverture d'un groupe est donc de 4R = 400m.

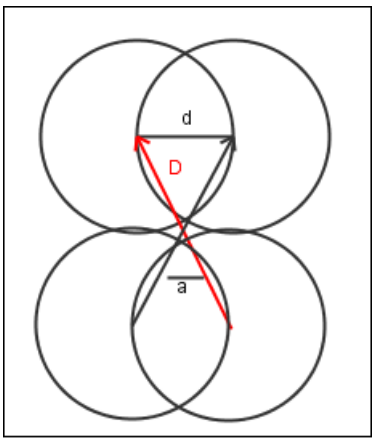


Figure 5.7 – Deux groupes se croisent à angle a

Nous devons donc calculer le temps T durant lesquels deux groupes, dont les trajectoires, qu'on assimile à des droites, se croisent avec un angle α restent en contact, connaissant d la couverture de communication maximale d'un groupe. Dans le schéma 5.7, on peut voir que T est le temps nécessaire pour parcourir la distance D. Connaissant la vitesse v du groupe, calculer D nous permet donc de trouver T.
On a donc :

$$\frac{0.5 * d}{0.5 * D} = sin(0,5 * \alpha) \implies D = \frac{d}{sin(0,5 * \alpha)}$$

Comme v=1m/s, on a :

$$T = \frac{D}{v} = \frac{d}{sin(0,5 * \alpha)}$$

On veut que les groupes restent distincts pour un angle $\alpha = \frac{\pi}{2}$. Dans ce cas :

$$T = \frac{200}{sin(\frac{\pi}{4})} \approx 235$$

Comme PERIOD=2s, nous avons fixé *stableThreshold* à 140 : un pair devient un voisin stable s'il est vu pendant au moins 4 minutes 40s.
Notre algorithme peut alors distinguer s'il y a fusion ou s'il s'agit de deux groupes se croisant avec un angle $\alpha \geq \frac{\pi}{2}$. Pour un angle α des trajectoires plus faible, il existe donc

Angle de croisement (en degré)	80	70	60	50
Temps (en minutes)	1mn2s	2mn17s	4mn40s	5mn53s
Angle de croisement (en degré)	40	30	20	10
Temps (en minutes)	7mn44s	10mn52s	17mn11 s	36mn14s

FIGURE 5.8 – Temps nécessaire à l'algorithme pour corriger une détection de groupe fallacieuse

une période *error* durant laquelle l'algorithme détectera de manière erronée un seul groupe au lieu de deux. Nous avons donc calculé *error* pour différentes valeurs de α.
A t_{deb}, supposons deux groupes qui entrent en contact, et le compteur de présence croît. On suppose qu'il n'y a pas d'absence sporadique. A $t_{grp} = 280$, les deux groupes ont fusionnés. A t_{end} =T, PC décroît. Dans ce cas :
- Soit T > 2*(stableThreshold+maxAbs)
  - PC = stableThreshold+maxAbs
  - l'erreur est corrigée au bout de 2*maxAbs secondes
  - l'algorithme était donc incorrect pour :
    $error = t_{end} + 2maxAbs - t_{grps} = T + 2 * maxAbs - 280$
- Sinon
  - $PC = \frac{T}{2}$
  - l'erreur est corrigée au bout de $2 * (\frac{T}{2} - stableThreshold)$
  - l'algorithme était donc incorrect pour :
    $error = t_{end} + 2 * (\frac{T}{2} - stableThreshold) - t_{grp} = 2 * T - 560$

La table 5.8 présente quelques résultats en fonction de l'angle selon lequel les deux groupes se croisent.
Comme on peut le voir, plus l'angle de croisement est grand, plus la durée de l'erreur est élevée. Un observateur humain ferait cependant sans doute la même erreur.
Notons que le temps de correction est en fait lié à deux paramètres : le temps d'attente avant de construire un groupe *stability threshold*, fixé ici à 280s, et le temps d'attente avant de sortir quelqu'un du groupe, fixé ici par *maxAbs* à 2 minutes. La raison pour fixer ce paramètre est présentée dans la section suivante.

5.6.3 Tolérance aux absences transitoires

Le paramètre *maxAbs* représente le nombre maximal de périodes durant lesquelles on tolère qu'un voisin stable soit absent avant de le sortir du voisinage stable.
Dans la définition du groupe stable donnée en 5.1.3, *maxAbs* est donc une approximation de δ_f, la durée durant laquelle deux pairs stables sont supposés être capables de communiquer. Quand deux voisins stables perdent contact, leur compteur de présence PC est compris entre *stableThreshold* et *stableThreshold+maxAbs*.
Ce paramètre a donc deux usages :
- Quand un voisin stable disparaît, on veut le sortir du groupe stable le plus rapidement possible. Il en va de même quand un groupe se sépare en deux.
- Quand un voisin stable est absent pour peu de temps, on ne peut pas être certain qu'il a vraiment disparu. On veut donc que notre algorithme tolère ces absences sporadiques et garde le voisin dans le groupe stable.

La fréquence de départ des pairs n'a pas d'influence sur *maxAbs*, à ceci près que *maxAbs* doit être assez bas pour permettre de les sortir rapidement du groupe.

Par contre, si on connaissait la fréquence et la durée des absences temporaires dans le pire cas, on pourrait fixer *maxAbs* de manière à les supporter. Ces absences peuvent avoir des causes diverses : un obstacle temporaire entre deux terminaux , comme par exemple un mur, ou une surcharge temporaire du réseau qui crée une perte de paquets
Comme nous n'avons pas de contrôle sur ces situations, ni de possibilité de les prédire, nous avons choisi *maxAbs* en nous basant sur la définition du groupe : si un pair faisant parti de notre groupe stable est absent pendant plus de 2 minutes, il doit être sorti du groupe.
On a donc : $maxAbs = \frac{60*2}{PERIOD} = 60$.

5.7 Critères d'évaluation classiques : Messages, Calcul, Passage à l'échelle

Cet algorithme a été tout d'abord évalué en terme de propriétés calculables a priori, à savoir le nombre de messages échangés et la complexité du calcul, et ce que l'on peut en déduire sur sa capacité à passer à l'échelle. L'algorithme proposé est avantageux sur ce point car du fait de l'utilisation d'informations inter-couches, aucun message n'est échangé pour la mise à jour du voisinage, et on ne fait donc pas de calcul complexe.
Tout d'abord, notre méthode ne nécessite *aucun échange* de message entre les pairs car elle profite des informations contenues dans les tables de routage. Bien que nous nous soyons ici basés sur OLSR, cet algorithme peut s'appuyer sur n'importe algorithme de routage proactif. Il peut aussi être exécuté au dessus d'un algorithme réactif, mais ne construira qu'une vue partielle du voisinage.
Par ailleurs, le rafraîchissement du voisinage est calculé toutes les 2 secondes, qui est la période d'envoi des messages HELLO d'OLSR. Ceci est fait en calculant les différences entre les listes *hearfOf*, et *table de routage*. Si les deux listes, de taille M et N sont triées, la complexité de cet algorithme est max(M,N). On travaille, par ailleurs, dans un réseau d'une centaine de terminaux, ce qui borne le temps de calcul.
Cette technique passe donc à l'échelle dans la même mesure que le protocole de routage sur lequel elle s'appuie.
Sur ces critères, notre algorithme est satisfaisant. Cependant, cette évaluation n'est pas suffisante. On propose un algorithme prédictif. Afin d'en montrer la validité, on doit vérifier que ses prédictions sont correctes.

5.8 Simulation

La validation de protocoles et d'applications pour MANets par des expériences réelles est complexe à mettre en œuvre. En effet, la mise en oeuvre de MANets pour la validation est problématique pour plusieurs raisons :
- les terminaux étant mobiles, on a besoin d'un opérateur, humain ou robot, par terminal, capable de reproduire des schémas de mobilité et des actions.
- les expérimentations sont difficilement reproductibles. Les communications étant radio, elles peuvent être perturbées par des signaux extérieurs sur lesquels l'expérimentateur n'a pas de contrôle.

Il existe des environnements de validation pour applications mobiles, comme par exemple Emulab [107], mais ils sont rares et peu accessibles, ce qui ne permet pas de reproduire plusieurs fois nos expériences.

Valider notre algorithme sur un MANet déployé nécessiterait donc un groupe d'opérateurs mobiles et de terminaux, ainsi qu'une salle isolée. N'ayant pas de telles ressources, nous avons donc validé notre algorithme par simulation sur le simulateur NS-3 [105].

5.8.1 NS-3

NS-3 (Network Simulator 3) est un simulateur de réseaux à événements discrets. Il se veut successeur de NS-2, et a été développé initialement par la National Science Foundation, et l'INRIA. C'est un projet open source sous licence GNU GPLv2, et tout le monde peut y contribuer.
NS-3 est implémenté en C++ pour le moteur de simulation et les modules (applications, protocoles et modèles physique), avec une sur-couche python pour créer des simulations.
L'implémentation de ce type de simulateur se décompose en un moteur de simulation et des modules. Le moteur de simulation comprend une horloge logique et une liste d'événements triés par date logique. Quand on crée une simulation, on la configure avec des modules, comme par exemple des protocoles réseaux, ou des générateurs de trafic, qui vont réagir à certains événements et en générer de nouveaux. Quand tous les événements à une date sont exécutés, l'horloge logique est avancée, ce qui active de nouveaux événements.
NS-3 est open source et gratuit. Le projet étant récent, NS-3 est plus simple à prendre en main que son ancêtre NS-2. Par ailleurs, le protocole OLSR est implanté dans NS-3, ainsi que la norme WIFI 802.11b avec différents modèles physiques, et les protocoles internet.
Nous avons donc pu nous concentrer sur le développement des modules spécifiques à l'algorithme proposé.

5.8.2 Méthodologie

Nous avons effectué une série d'expérimentations avec NS-3 pour valider l'algorithme de création de groupes proposé.
Chaque expérimentation est paramètrée par les valeurs suivantes :
- la taille de l'aire simulée.
- le nombre de groupes mobiles.
- le nombre de terminaux par groupe.
- le type d'événement de mobilité qu'on souhaite générer (fusion, séparation, disparition), sachant que le modèle de mobilité est RPGM.
- la volatilité des terminaux.

Le processus de simulation se déroule ensuite, comme illustré par la figure 5.9, ainsi :

1. Un script python génère :
   - des traces de mobilité (liste de coordonnées cartésiennes dans l'aire) pour chaque terminal.
   - des traces de volatilité pour chaque terminal. La période entre deux absences et le temps d'absence sont générés aléatoirement et suivent des lois normales gaussiennes.
2. NS-3 est ensuite configuré avec les modules suivants :
   - couche physique wifi et liaison de donnée 802.11b.
   - couche réseau IPV4 et OLSR.
   - *ReadFromTracesApplication* : modèle de mobilité que nous avons développé, lisant les traces générées à l'étape 1.
   - deux applications, que nous avons développées pour cette simulation :

- *OffOnDeviceApplication* : éteint et allume la carte wifi pour simuler des *erreurs transitoires*, en fonction des valeurs générées auparavant.
- *DumpRoutingTableApplication* : récupère le contenu des tables de routage depuis le module OLSR toutes les deux secondes, les transforme sous la forme d'une liste de paire IP/Distance, et à la fin de la simulation, les sauvegarde sur disque.

3. Un script python utilise ces routes pour exécuter ensuite plusieurs fois l'algorithme avec différentes valeurs pour les paramètres *maxAbs* et *stableThreshold*.

Découpler ces différentes étapes du simulateur nous permet d'optimiser le temps de validation :
- Générer la mobilité en dehors de NS-3 par un script python et n'utiliser qu'un lecteur de traces dans NS3 est beaucoup plus flexible et rapide que développer plusieurs modules NS-3.
- Découpler l'exécution de l'algorithme de la simulation physique est beaucoup plus rapide : en effet, on réutilise les même schémas de mobilité pour examiner le comportement de l'algorithme pour différentes valeurs des paramètres, sans avoir à simuler plusieurs fois les couches basses.
- On peut ainsi utiliser directement le code de l'algorithme, sans avoir à le réimplémenter dans NS-3.

5.8.3 Critère d'évaluation : une métrique de précision

Nous avons vu dans la section précédente comment l'algorithme proposé s'évaluait selon les critères de complexité classique.
Cependant, l'algorithme proposé étant de nature prédictive, nous voulons pouvoir l'évaluer en mesurant l'acuité de ses prédictions en supposant l'existence d'un oracle qui puisse déterminer des groupes idéaux en se basant sur la connaissance de l'ensemble des positions et des mouvements des terminaux.
Pour cela, nous allons simuler des terminaux mobiles se déplaçant en groupes, suivant le modèle RPGM. L'oracle connaît ces groupes simulés, et idéalement, ceux-ci devraient être détectés par notre algorithme.
Pour déterminer la précision de l'algorithme proposé, nous comparons donc la composition du groupe idéal *ideal*, ici le groupe simulé, au groupe déterminé par l'algorithme *computed* (indice de Jaccard) :

$$accuracy(t) = \frac{|ideal \cap computed(t)|}{|ideal \cup computed(t)|}$$

On notera que $|ideal \cap computed(t)|$ n'est jamais nul, puisque le pair lui même est toujours contenu dans les deux ensembles eux-mêmes.
Le résultat de cette métrique a valeur dans $]0; 1]$. Plus un algorithme est précis, plus cette valeur tend vers 1.

5.8.4 Sensibilité à la densité : calibrer la simulation

Si les terminaux évoluent dans une aire trop petite, ils vont interagir du fait de la densité de nœuds même s'ils appartiennent à des groupes différents. Dans les simulations où l'on essaye de distinguer plusieurs groupes, il faut donc utiliser une aire simulée assez grande pour éviter ces interactions.

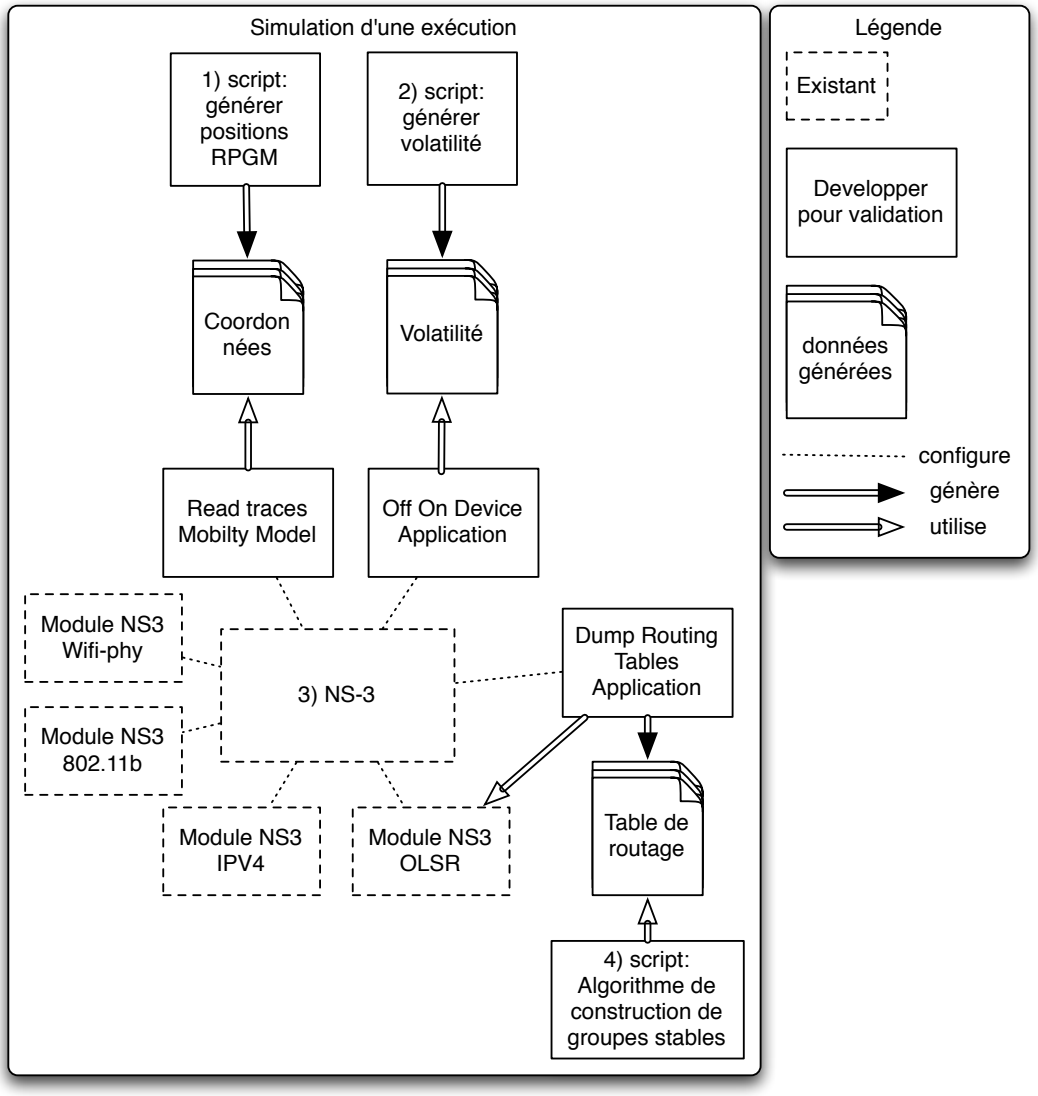


FIGURE 5.9 – Processus de validation

Ce problème est aussi lié à l'aire de couverture d'un groupe, qui dépend elle-même de comment la mobilité est simulée. La figure 5.10 représente différentes organisations possibles du groupe simulé.

Dans notre simulation, nous utilisons le modèle RPGM, correspondant à la figure 3 dans la figure 5.10, que nous avons présenté dans nos hypothèses. La portée de communication maximum de 802.11b en extérieur étant de 100m, l'aire de couverture maximale d'un groupe est donc un disque de diamètre 200m.

Par la suite, nous avons choisi la taille de l'aire de simulation de manière à pouvoir placer les groupes hors de portée de communication.

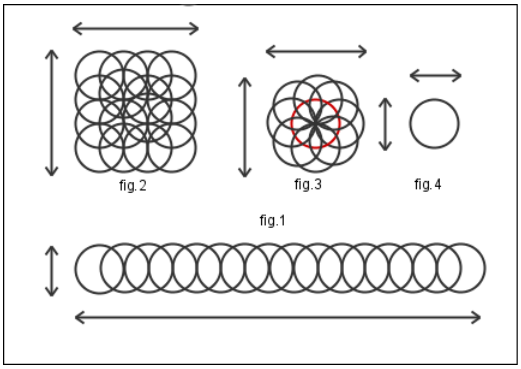


Figure 5.10 – Plusieurs configurations possibles pour le groupe

5.8.5 Scenarii

Afin de valider notre algorithme, nous avons déterminé différentes situations particulières à tester.
Tout d'abord, nous simulons les interactions entre groupes, sans absence transitoire, afin de tester le paramètre du seuil de stabilité *stableThreshold* ainsi que de *maxAbs*, permettant de détecter la séparation d'un groupe.
– le cas par défaut : déplacement de groupes sans interactions.
– deux groupes fusionnent.
– un groupe se sépare en deux.
Nous introduisons ensuite des fautes transitoires, afin de vérifier l'impact de *maxAbs*.
Enfin, nous exécutons un scénario général, avec une mobilité de type RPGM, sans trajectoire déterminée, et des absences transitoires des pairs.

5.8.5.1 Un groupe mobile

Dans ce scénario, illustré par la figure 5.11 nous voulons tester la validité de notre algorithme dans le cas le plus simple : un groupe de pairs se déplace sans qu'il y ait de fautes ou d'interaction avec d'autres groupes.

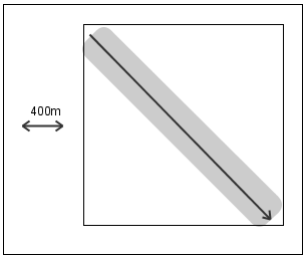


Figure 5.11 – Un groupe

Scénario : un groupe de pairs se déplace en diagonale depuis le coin haut gauche de l'aire simulée. On veut savoir quelle valeur de *stableThreshold* maximise l'exactitude de notre algorithme. Celle-ci varie entre 100 et 180 dans notre test.
Résultats attendus : on s'attend à ce que le groupe simulé ne soit pas correct entre 0 et 2**stableThreshold* secondes ; l'exactitude est ensuite de 1 pour le reste de la simulation. La plus petite valeur de *stableThreshold* devrait donc maximiser l'exactitude de l'algorithme.

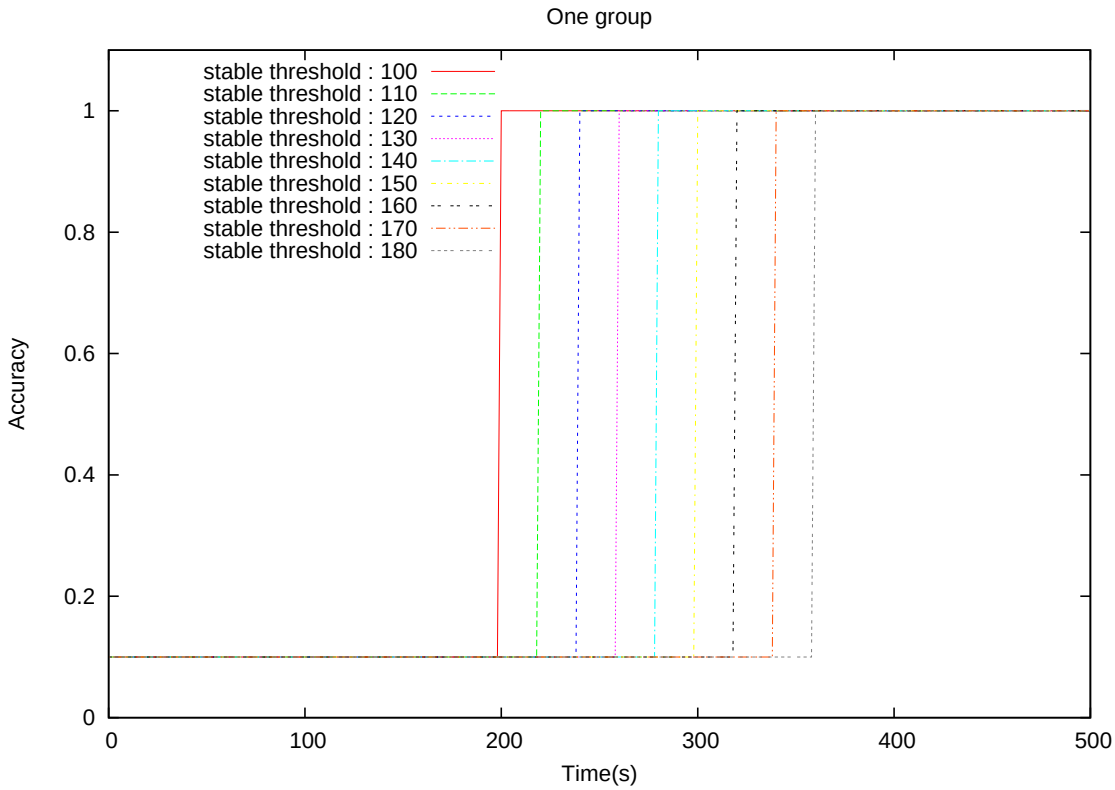


FIGURE 5.12 – Un groupe, $\neq$ seuils de stabilité

Résultats observés : le graphe 5.12 montre les résultats de la simulation. Chaque courbe représente l'exactitude de l'algorithme en fonction du temps pour une valeur donnée de *stableThreshold*. On peut voir que la simulation valide les résultats attendus : pour une valeur de *stableThreshold* plus grande, la formation du groupe prend plus longtemps.
Notons que l'algorithme est exécuté toutes les 2 secondes, et en l'absence d'erreurs, détecte tous les terminaux comme faisant parti de son groupe stable, au bout de 2**stableThreshold*.
Le passage de la précision de 0 à 1 apparait comme une impulsion, mais cela est du à l'échelle du graphe.

5.8.5.2 Deux groupes sans interaction

Dans le scénario illustré par 5.13, nous testons si les simulations sont correctement calibrées : étant données la taille de l'aire simulée et la portée de communication, deux groupes se déplaçant dans des zones non recouvrantes ne devraient pas interagir.
Scénario : deux groupes de terminaux suivent les bords opposés de l'aire. Ils ne sont jamais en contact. Le paramètre *stableThreshold* varie entre 100 et 180
Résultats attendus : les deux groupes n'interagissant pas, le graphe obtenu devrait être semblable à 5.12

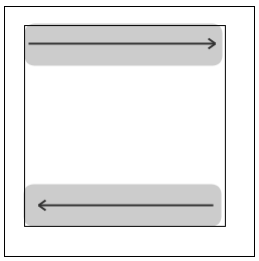


Figure 5.13 – Deux groupes sans interactions

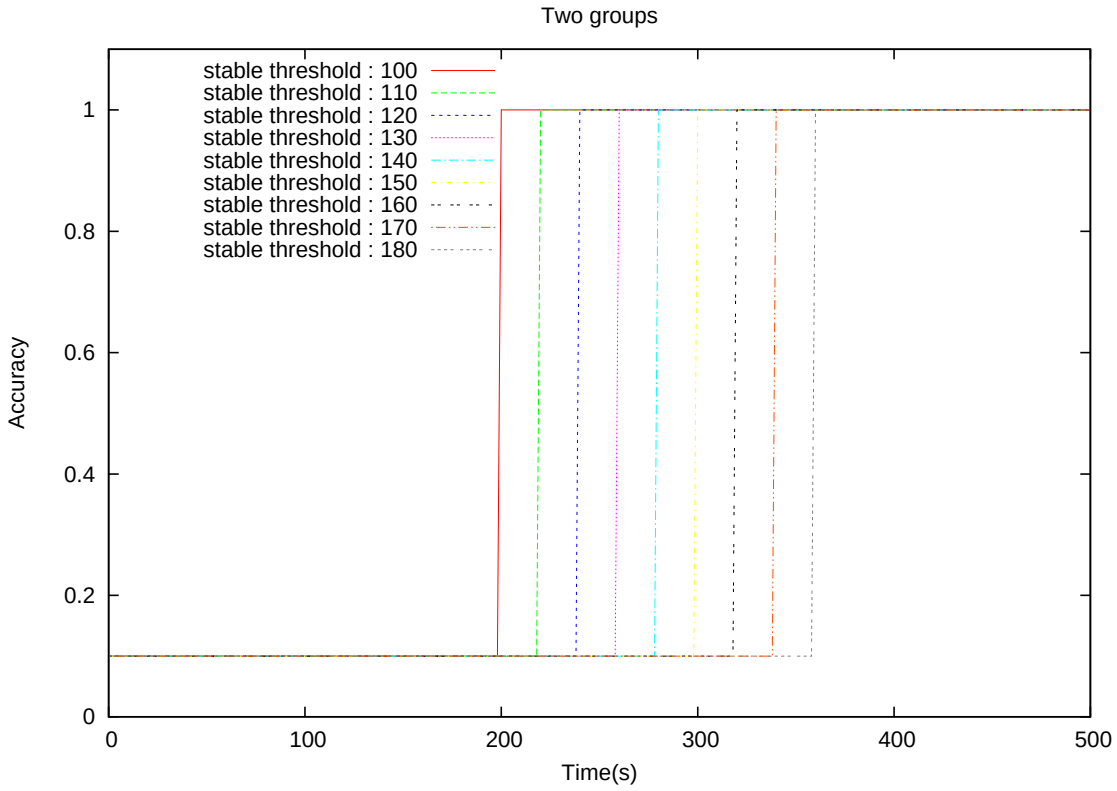


Figure 5.14 – Deux groupes, $\neq$ seuils de stabilités

Résultats observés : le graphe 5.14 montre que la précision de l'algorithme varie comme dans l'expérience précédente. La simulation est donc bien calibrée.

5.8.5.3 Deux groupes dont les chemins se croisent

On cherche ici à valider le comportement de l'algorithme dans le cas où deux groupes se croisent.
Tout d'abord, nous vérifions pour un seuil de stabilité fixe, la précision de l'algorithme de

grappes en fonction de l'angle des trajectoires des deux groupes. Nous vérifions ensuite pour différentes valeurs du seuil de stabilité.

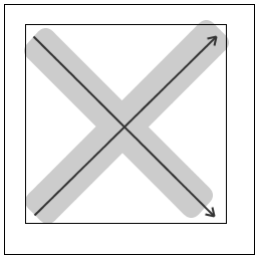


FIGURE 5.15 – Deux groupes dont les chemins se croisent

Scénario : dans ce scénario, on veut vérifier le comportement de l'algorithme en fonction de la valeur du seuil de stabilité *stableThreshold*, qui varie entre 100 et 180. Les deux groupes partent chacun en diagonale des coins droits de l'aire simulée et leurs trajectoires se croisent à angle $\frac{\pi}{2}$rad au centre de l'aire à t=1415. On observe la précision de l'algorithme autour de cette date.
Résultats attendus : pour les valeurs de *stableThrehold* supérieures à 140, la précision de l'algorithme devrait rester à 1. En dessous de 140, la précision devrait se détériorer comme la valeur de *stableThreshold* décroît.
Résultats observés : la figure 5.16 présente le résultat, chaque courbe représentant une simulation pour une valeur différente de *stable threshold*. On peut voir que les résultats sont même meilleurs que ceux attendus, et les deux groupes restent distincts pour un seuil de stabilité de 120 : seuls des seuils de stabilité de 110 et 100 créent une confusion entre les deux groupes.
Scénario : dans ce jeu de scénarios, deux groupes sont placés hors de portée de communication, et leurs trajectoires se croisent au centre de l'aire. L'angle α formé par les trajectoires varie entre 0 et π rad. Leurs positions de départ sont choisies de manière à ce que les groupes se croisent à t=1000, comme illustré dans la figure 5.17. Nous voulons vérifier que pour le paramètre *stableThreshold* fixé à 140, l'algorithme sépare bien les groupes dont les trajectoires ont des angles supérieurs ou égaux à $\frac{\pi}{2}$ rad.
Résultats attendus : si les deux groupes ont des trajectoires d'angle 0, celles ci sont parallèles. Comme on sait qu'elles se croisent par construction à t=1000, elles sont donc confondues. Les deux groupes évoluent en fait comme un et la courbe de précision devrait valoir 0,5. Pour les valeurs de α dans $[0\,;\,\frac{\pi}{2}]$, l'erreur diminue comme l'angle grandit. Pour les valeurs supérieures à $\frac{\pi}{2}$, la précision de l'algorithme reste à 1.
Résultats observés : dans la figure 5.18, chaque courbe représente l'évolution de la précision de l'algorithme pour un angle α donné. On peut voir que les résultats attendus pour α=0 et $\alpha > \frac{\pi}{2}$ sont ceux obtenus. La précision de l'algorithme est bien meilleure quand α grandit et l'on voit même que le temps *error* durant lequel l'algorithme est confus est inférieur à celui calculé théoriquement. Pour $\alpha = 45°$ par exemple, *error* vaut 4mn40, une valeur inférieure même à celle attendue pour $\alpha = 50°$ de 5mn53.

5.8.5.4 Deux groupes fusionnent

Scénario : dans ce scénario, nous voulons observer le comportement de l'algorithme quand deux groupes fusionnent. Comme illustré par la figure 5.19 deux groupes partant des coins bas opposés de l'aire simulée se rejoignent au centre de l'air pour fusionner, à t=1415, en

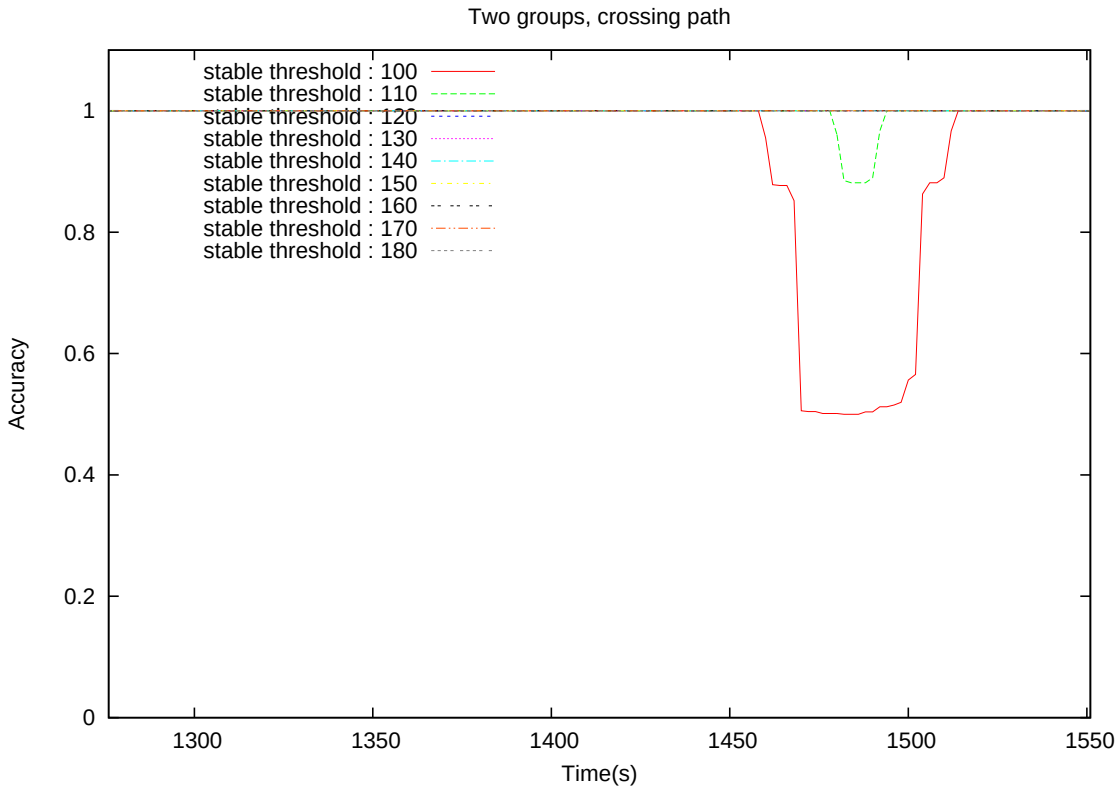


Figure 5.16 – Deux groupes dont les chemins se croisent, $\neq$ seuils de stabilité

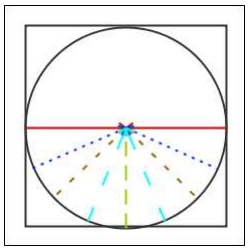


Figure 5.17 – Même distance avant rencontre, angles différents

un seul groupe qui se dirige vers le haut. Nous voulons vérifier que l'algorithme détecte la fusion, et le temps nécessaire à cela, en fonction de différentes valeurs de *stableThreshold* qui varie donc entre 100 et 180.

Résultats attendus : plus le seuil de stabilité *stableThreshold* est bas, plus la fusion des groupes sera rapidement détectée.

Résultats observés : la figure 5.20 présente le résultat de l'expérimentation sur la fusion de groupes. On peut voir que, comme on l'attendait, un seuil de stabilité de 100 permet de fusionner les deux groupes plus rapidement qu'un seuil de 180.

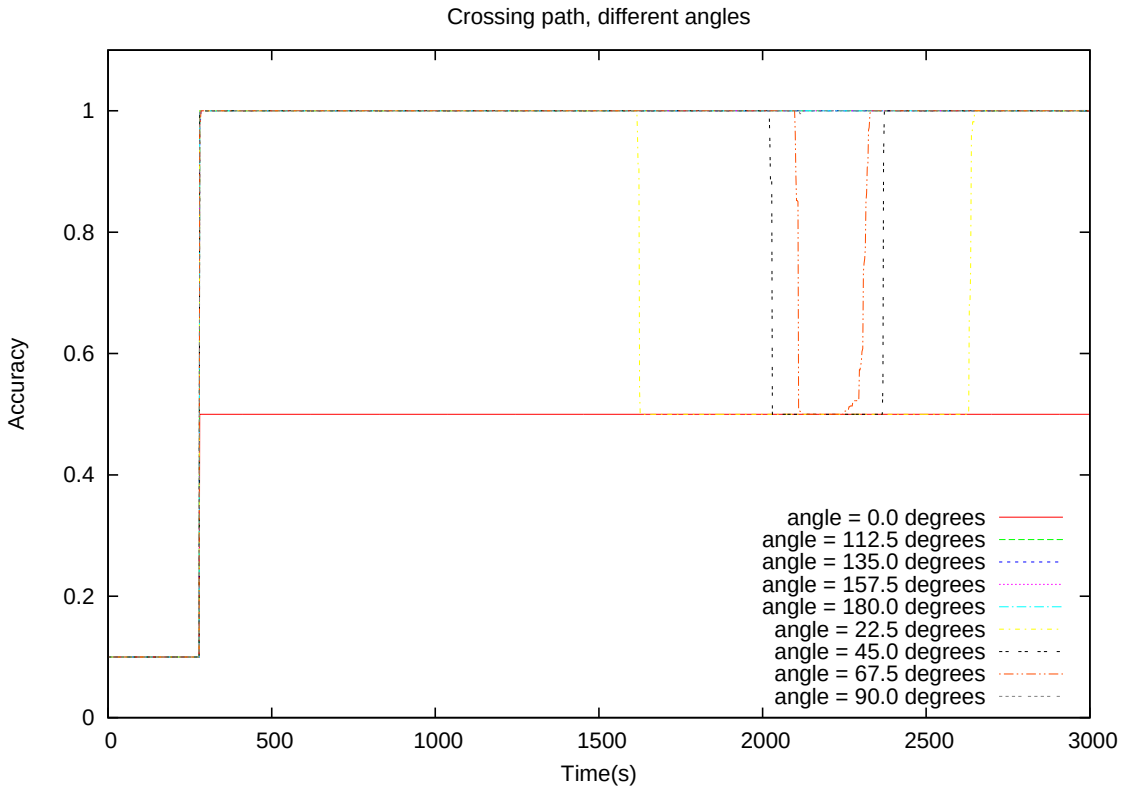


FIGURE 5.18 – 2 groups crossing, $\neq$ angles

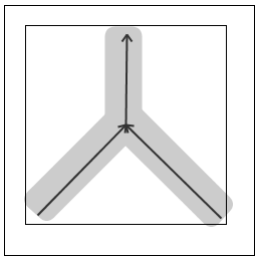


FIGURE 5.19 – Fusion de deux groupes

Notons que cette expérience est une généralisation du cas d'un pair rejoignant le groupe, ce qui valide donc notre cas de test.

5.8.5.5 Un groupe se sépare en deux

Dans ce scénario, illustré par la figure 5.21 un groupe de pairs se déplace assez longtemps pour que le groupe soit détecté par l'algorithme proposé, avant de se séparer en deux. On examine le comportement de notre algorithme au moment de la séparation.

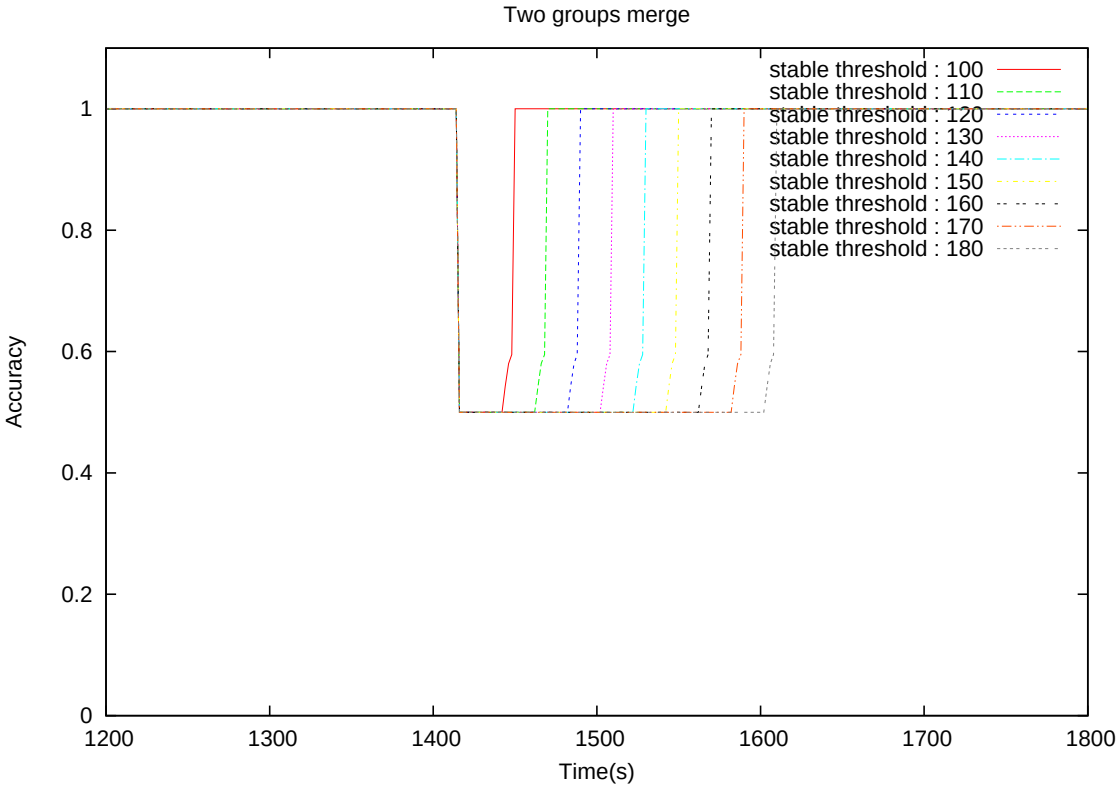


Figure 5.20 – Deux groupes fusionnent, $\neq$ seuils de stabilité

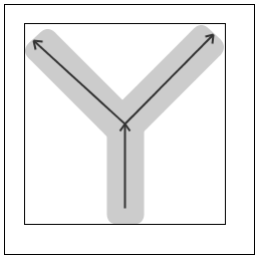


Figure 5.21 – Un groupe se sépare en deux groupes

Scénario : dans ce scénario, un groupe se déplace du centre bas de l'aire simulée au centre. Il se sépare alors en deux groupes, chaque groupe partant en diagonale vers les coins opposés de l'aire. On veut vérifier que notre algorithme détecte bien la séparation, et mesurer le temps nécessaire en fonction du nombre maximum d'absences tolérées. Le paramètre *stableThreshold* est fixé à 140, et le paramètre maxAbs varie entre 20 et 90.
Résultats attendus : on examine le comportement de notre algorithme autour de t=1000 sec, moment de la séparation en deux groupes qui partent dans des direction distinctes. L'algorithme ne peut pas être absolument exact car après la séparation les 2 groupes restent

encore en communication pour, au plus, 224 sec. Cependant, nous attendons de ce test que plus le paramètre *maxAbs* est bas, plus notre algorithme se corrige rapidement.

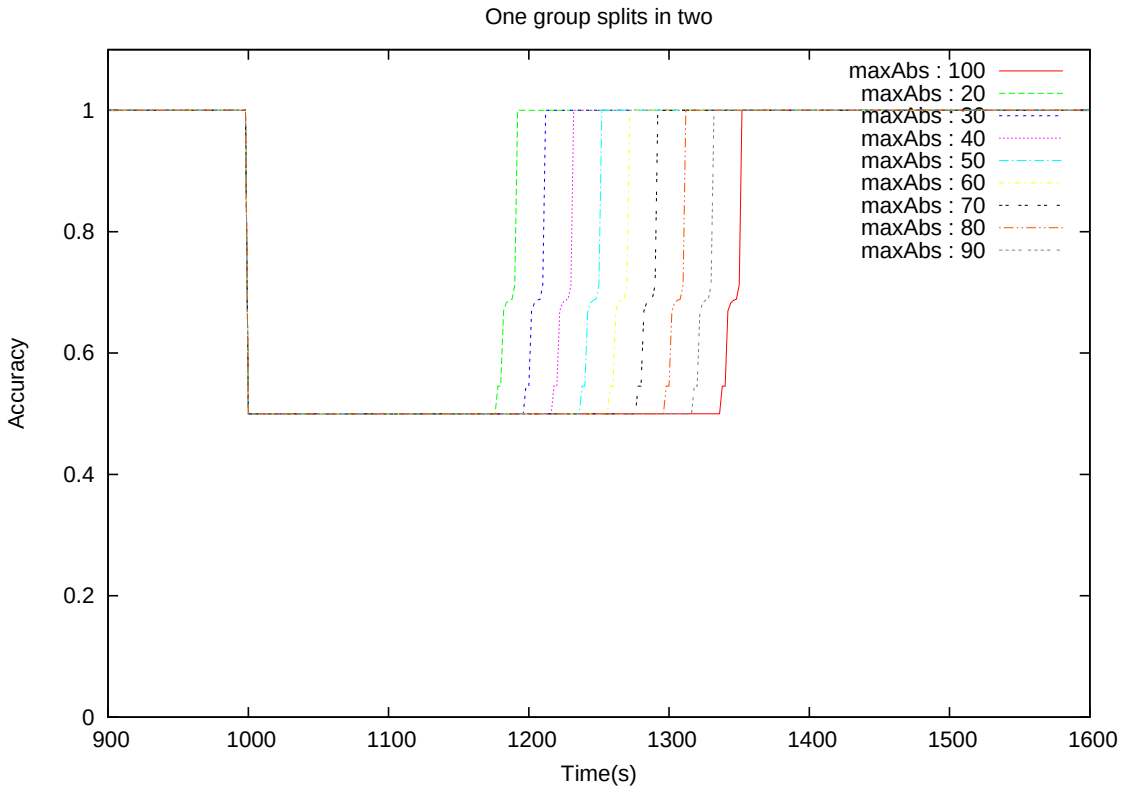


FIGURE 5.22 – Un groupe se sépare en 2, $\neq$ valeurs pour le nombre maximum d'absences tolérées

Résultats observés : dans la figure 5.22, chaque courbe représente l'évolution de la précision de l'algorithme au cours du temps en fonction d'une valeur donnée de *maxAbs*. On voit que la séparation des deux groupes est plus rapidement effective quand *maxAbs* est bas, ce qui valide nos attentes.
Notons que cette expérience est une généralisation du cas d'un pair quittant un groupe, et valide donc ce cas de test.

5.8.5.6 Un groupe mobile, absence transitoire d'un pair

Dans ce cas de test nous voulons évaluer l'impact de la durée d'une absence sporadique d'un terminal sur la précision de l'algorithme. Ces absences sont absorbées dans une certaine mesure puisque *maxAbs* absences sont normalement tolérées.
Scénario : un groupe de 10 terminaux se déplace selon un modèle RPGM et un pair subit des absences transitoires. Ces absences sont de plus en plus longues, de 1 à 7 minutes.

L'intervalle entre deux absences est assez long pour que le groupe retrouve sa stabilité. Le paramètre *stableThreshold* est fixé à 140 et *maxAbs* varie entre 20 et 100.
Résultats attendus : le pair fautif disparaît ainsi : à t=560 pour 1min, à t=1020 pour 2min, à t=1540 pour 3min, à t=2120, pour 4mn, à t=2760 pour 5min, à t= 3460 pou 6 min et à t=4220 pour 7min. Pour une valeur de *maxAbs* donnée, seules les absences de durée inférieure à 2**maxAbs* n'impactent pas la précision de l'algorithme. Par exemple, pour une absence de 2 minutes, seules les courbes 20, 30,40 et 50 devraient subir un changement.

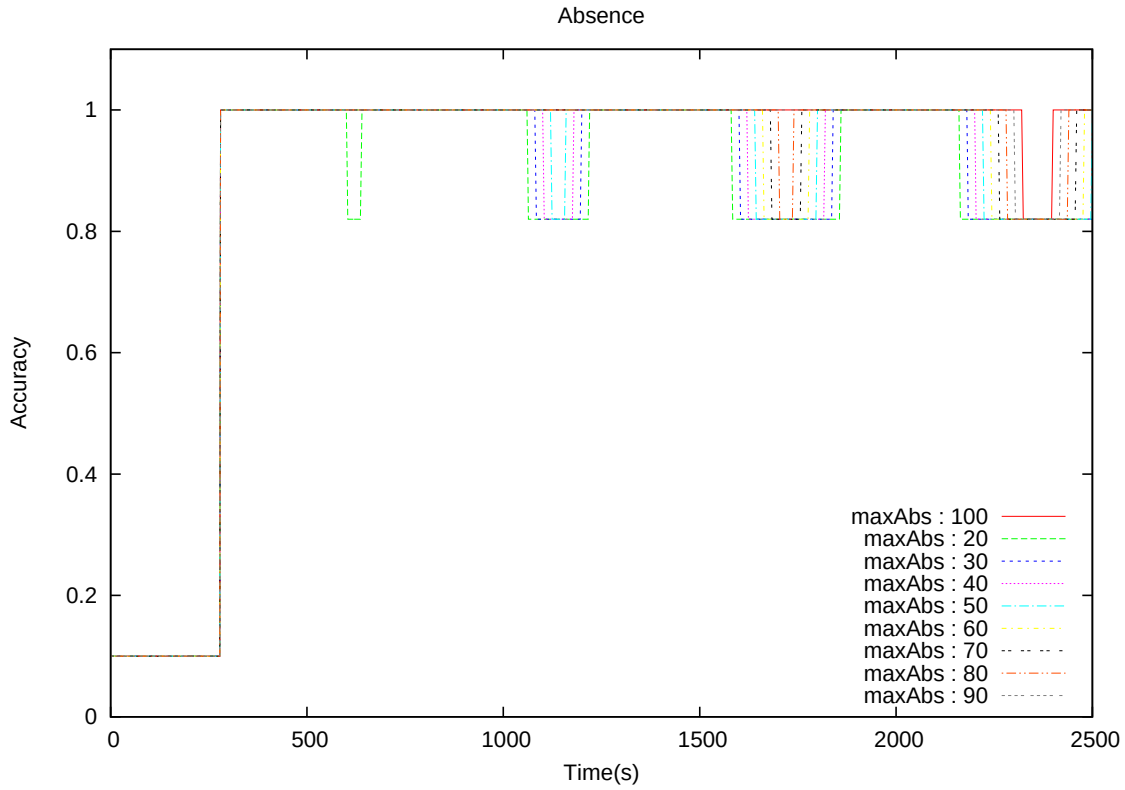


Figure 5.23 – Un groupe se sépare en deux groupes, absence de 1,2,3,4 minutes

Résultats observés : les figures 5.23 et 5.24 présente l'évolution de la précision de l'algorithme de groupe en fonction du temps. On a bien le résultat attendu : une absence d'une minute est considérée comme une sortie de groupe seulement pour *maxAbs* inférieur à 30, une absence de deux minutes pour *maxAbs*<60, etc.

5.8.5.7 Plusieurs groupes aux déplacements aléatoires absence transitoire

Pour finir, nous avons évalué notre algorithme dans un cadre plus général.
Dans ce scénario, 4 groupes de 30 terminaux évoluent dans une aire carrée de 5000mx5000m pendant 2 heures. Chaque groupe suit un modèle de mobilité *Reference Point Group Mobility*, où le point de référence suit un modèle de mobilité *Random Waypoint*.

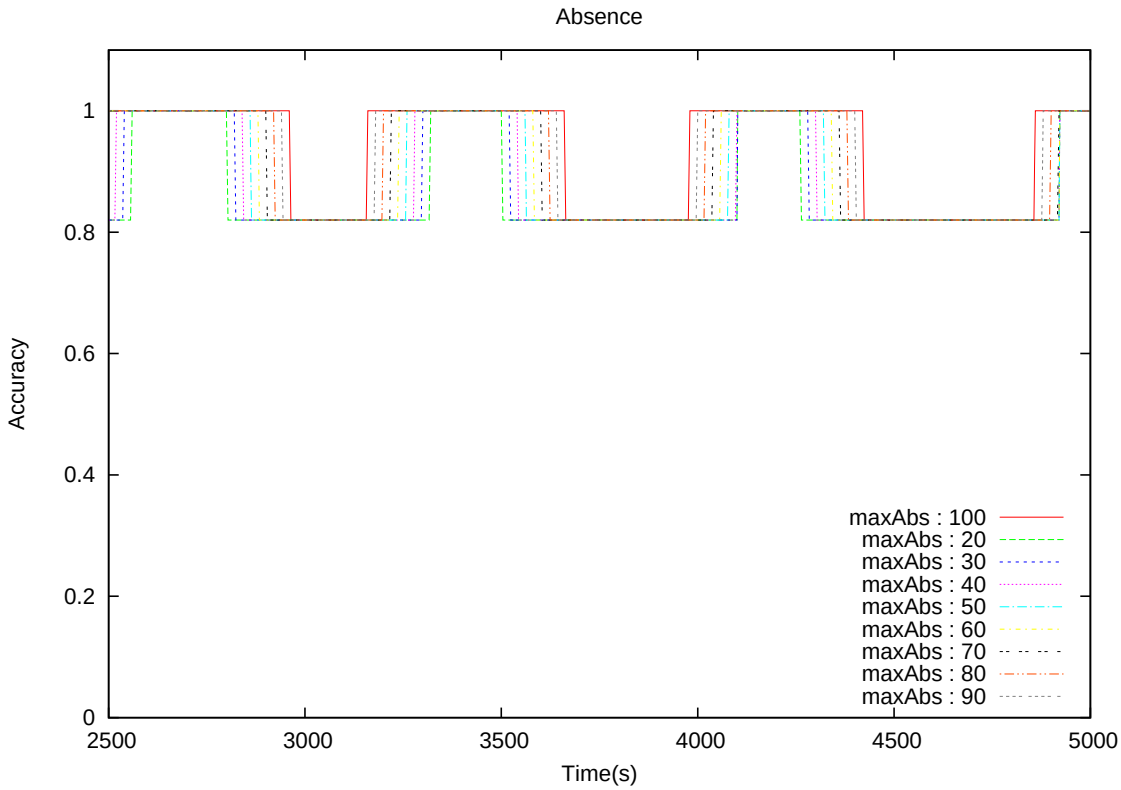


FIGURE 5.24 – Un groupe se sépare en deux groupes, absence de 5, 6, 7 minutes

Chaque terminal est susceptible d'absences transitoires, modélisées de la façon suivante :
- le temps d'absence est modélisé par une loi Normale de paramètre μ=60 et σ=30 : en moyenne une absence dure 1 minute, avec une dispersion de 30s.
- le temps d'intervalle entre deux absences est modélisé par une loi Normale de paramètre μ=5*60 et σ=60 : en moyenne, le temps entre deux absences est de 5 minutes, avec une dispersion d'1 minute.

Le temps moyen d'absence étant inférieur à 2 minutes, la précision de l'algorithme devrait donc rester à 1.
Comme les figures 5.25 et 5.26 le montrent, l'algorithme proposé produit un bon résultat dans le cas général : bien que subissant de faibles variations, la précision de l'algorithme reste proche de 1.

5.8.6 Synthèse

Dans cette section nous avons validé notre algorithme par simulation dans des cas précis afin de tester les paramètres de notre algorithme, *maxAbs* et *stableThreshold*. Nous avons ensuite vu un cas général avec des déplacements de groupes aléatoires.
Ceci nous a permis de valider que :

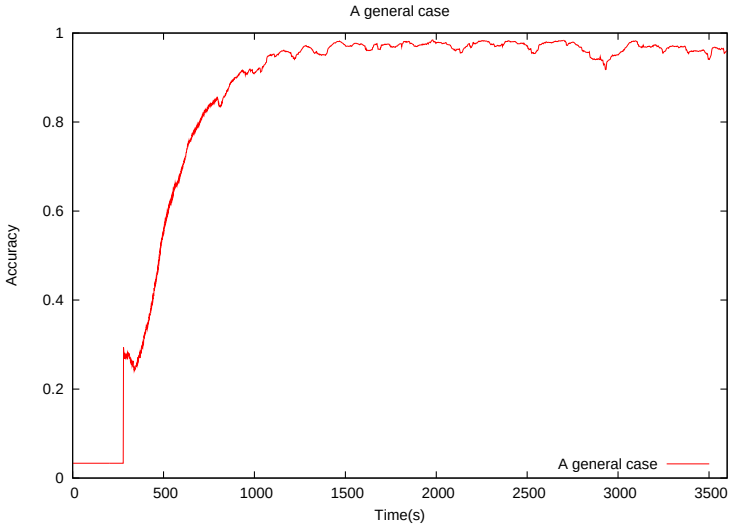


Figure 5.25 – 1ère heure

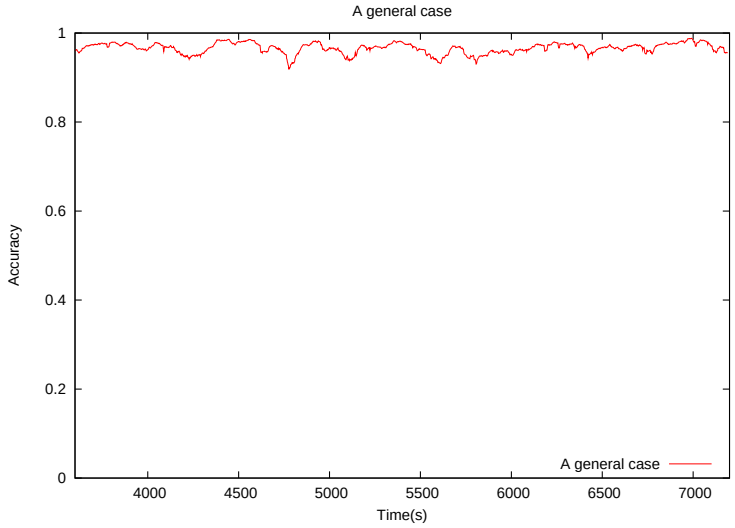


Figure 5.26 – 2ème heure

– Quand l'angle d'approche est supérieur ou égal à 90°, notre algorithme peut différencier deux groupes se croisant de deux groupes fusionnant ; dans le cas contraire, il répare son erreur dans un temps inférieur à la valeur théorique calculée.
– Notre algorithme peut supporter les absences transitoires des terminaux, de durée inférieure à 2 minutes.

5.9 Comparaison à l'existant

Le point fort de cet algorithme est *l'absence de surcoût réseau*. On notera d'ailleurs que la plupart des propositions existantes n'évaluent pas leur algorithme en terme de messages. En effet, tous les autres algorithmes nécessitent un échange périodique d'informations (du moins coûteux au plus coûteux) :
– à la tête de grappe une fois que celle ci est élue : [43] et [113]
– à un saut : [11], [97] et [114]
– à l'intégralité du réseau : [7] et [103]
Notre algorithme ne crée pas de surcoût réseau car il s'appuie sur des informations inter-couche issues des couches de routage. Il nécessite donc un protocole de routage pro-actif, et passe donc à l'échelle dans les limites de celui-ci.
Par ailleurs, notre proposition est totalement distribuée, et ne s'appuie pas sur un serveur ([103] et [24]), ni sur la nécessité d'élire une de tête de grappe pour lui déléguer des décisions.
Nous n'utilisons pas de système de position, comme GPS, ni de synchronisation d'horloge, comme [7] [97] [103], [43] et probablement [104] et [114].
Nous l'avons validé sur NS-3, un simulateur reconnu par la communauté des chercheurs en réseaux.
Enfin, nous en proposons un prototype concret et ne faisons pas uniquement une évaluation mathématique théorique, contrairement à [104] , [38] et [25].

5.10 Synthèse et conclusion

Dans un environnement où les utilisateurs sont mobiles, pour améliorer l'efficacité de certains services, comme la diffusion de messages (choisir des relais stables pour la multi-diffusion), ou le stockage collaboratif de données (stocker pour des terminaux susceptibles de rester présents assez longtemps pour utiliser la donnée), nous voulons pouvoir déterminer parmi un essaim de terminaux, quels sont les sous-groupes ayant une mobilité similaire. On a donc besoin d'un algorithme de création de grappes.
Le temps d'une session de travail étant limité par la durée de vie de la batterie, et la carte wifi étant une des sources majeures de consommation, il est important, quand on crée un algorithme, de limiter les communications.
Dans cette section nous avons donc proposé un algorithme de construction de groupe stable dans le temps qui se base sur des informations inter-couches (les tables de routages d'un protocole pro-actif, OLSR) à usage pour un groupe de piétons.
Etant donné les problèmes pour effectuer une validation dans un environnement réel, nous l'avons validé sur le simulateur NS-3. Notons que c'est le cas de la plupart des algorithmes de gestion de la mobilité, car la simulation est beaucoup moins coûteuse en ressources et permet la reproduction des expérimentations à volonté.
Notre algorithme a l'avantage de ne pas créer de surcharge réseau. On notera que les algorithmes existants sont validés de manière théorique, et par simulation, mais que peu d'entres eux examinent le surcoût en communications réseau. Il est totalement distribué, et ne dépend pas d'un matériel spécifique pour obtenir la position des terminaux. Il est cependant dépendant de l'utilisation d'un protocole de routage pro-actif.
Dans les chapitres suivants, nous allons voir comment ces groupes stables sont utilisés pour implémenter un algorithme collaboratif de réplication de données.

Chapitre 6

Réplication de données

Dans ce chapitre nous présentons notre algorithme de réplication pro-actif qui fait usage des groupes stables, présentés au chapitre précédent, pour limiter la perte des données, et répliquer collaborativement.

Comme nous l'avons vu, la réplication des données permet de les rendre plus disponibles, et en cas de disparition d'hôte, de fiabiliser le système de partage.
Nous présentons ici un modèle de réplication des données dans un MANet permettant de préserver l'intégrité de l'espace de partage.
Nous motivons tout d'abord le choix d'utiliser des informations sémantiques pour aider à la réplication des données. Nous verrons comment celles-ci sont extraites et utilisées sur un exemple.
Nous présentons ensuite le schéma de réplication de que nous utilisons avant d'en évaluer la pertinence et le gain qu'il apporte en terme de fiabilité puis le surcoût qu'il engendre. Nous verrons enfin comment il se compare à l'existant.

6.1 Choix de conception : utilisation d'informations sémantiques

Dans le contexte des MANets, nous proposons un système de partage d'information de données dont la granularité est de l'ordre du fichier et où les données sont compréhensibles par l'humain. On veut utiliser ce sens pour prédire les accès des utilisateurs.
Trois problèmes se présentent alors à nous :

1. extraire des informations pertinentes du contenu,
2. caractériser l'utilisateur, en extrayant des informations de ses accès passés,
3. à partir de ces informations, prédire ce qui intéressera l'utilisateur par la suite.

6.1.1 Indexation et recherche sur le contenu

L'utilisation la plus standard des informations sémantiques dans le partage de données est la recherche.
Pour ce faire, les données sont tout d'abord indexées. Cela peut être fait de différentes manières :
– Indexation automatique par le contenu des données (extraction de mots-clés) ; ex. : les moteurs de recherche web modernes.
– Tag par des utilisateurs (mots-clés) et la machine (geotag et timetag) ; ex. : youtube, flickr.
Dès lors une recherche associe un jeu de données triées par ordre de pertinence de la requête à une liste de mots-clés.
Dans ces systèmes, il n'y a pas de caractérisation de l'utilisateur, et on ne peut donc pas répondre pro-activement à ses besoins.

6.1.2 Recommandation, filtre collaboratif

L'extraction de contenu est une technique maîtrisée pour les documents de type texte, comme l'atteste le succès de Google, mais pour le contenu multimédia, l'indexation par le contenu est plus ardue. L'indexation de contenu multimédia est un sujet de recherche actuel, mais les résultats obtenus ne sont pas aussi satisfaisants que pour les documents textes.
Une solution est de laisser le soin aux utilisateurs humains d'annoter les données : c'est le but, par exemple, du jeu *Google Image Labeler*, qui a permis d'améliorer la pertinence des résultats de Google Image.

Une autre classe de solution est le filtrage collaboratif [15].
Cette technique consiste à regrouper non pas les données similaires, mais les utilisateurs susceptibles de s'intéresser aux mêmes sujets.
Dans des réseaux sociaux, comme Facebook, ceci est fait simplement en recommandant ce que les noeuds proches dans le réseau ont aimé. Pour des systèmes sans notion de réseau, on applique les techniques de regroupement de données *(data clustering)* sur les utilisateurs [101].
Les filtres collaboratifs sont utilisés avec succès pour faire de la recommandation dans de nombreux systèmes commerciaux, comme par exemple amazon ou iTunes. On peut aussi les utiliser pour répliquer des pages web par anticipation, comme dans [47]
On n'a alors pas besoin d'information sur le contenu des données, mais il faut, par contre, maintenir un historique de tous les accès de tous les utilisateurs.
Une telle solution est, par nature, centralisée, et peut poser des problèmes de vie privée, car elle nécessite de conserver des informations sur les utilisateurs.

6.1.2.1 Assistants personnels

Les deux premières techniques proposées ci-dessus, malgré leur efficacité, ont le défaut, pour un MANet, d'être centralisées.
Des solutions ont été proposées pour la navigation internet assistée sur un poste, comme par exemple dans [63]. Ces solutions reposent sur les techniques suivantes :
– Parcours des liens depuis une page,
– Construction à l'aide d'une ontologie des concepts intéressant l'utilisateur, et des concepts de chaque page.

Le parcours des liens crée une charge réseau que nous voulons éviter. De même, nous ne voulons pas embarquer un moteur d'inférence et une ontologie sur chaque terminal léger.

Les techniques actuelles permettant de prédire quelles données vont intéresser un utilisateur se basent soit sur la connaissance de l'ensemble des données pour les trier par intérêt croissant, soit sur la connaissance de l'ensemble des accès de tous les utilisateurs, afin de ranger ceux-ci par similarité d'accès.
Ces techniques sont très efficaces, mais elles ne sont cependant pas aisées à mettre en œuvre dans un MANet, car elles nécessitent un serveur. Nous allons donc utiliser un algorithme de prédiction d'intérêt plus simple, mais qui ne sera sûrement pas aussi pertinent.

6.2 Acquistion de mots-clés et prédiction d'intérêt

6.2.1 Structures de données

Afin de mettre en place la réplication des données, nous utilisons les structures de données suivantes sur chaque terminal :
– *accesses* : Un dictionnaire associant un mot-clé à un compteur d'accès, dont nous verrons plus bas comment il est peuplé.
– *interests* : L'ensemble des mots-clés ayant été détectés comme intéressant l'utilisateur.

Par ailleurs, à chaque donnée est associé l'ensemble de mots-clé *keywords*, qui décrivent son contenu.

6.2.2 Extraction des informations sémantiques

6.2.2.1 Mots-clés des données

À chaque donnée doit être associé un jeu de mots-clés dont la construction dépend de la nature des données.
Dans la section suivante, nous présentons le résultat d'expériences faites sur les données issues de wikipedia.

6.2.2.2 Intérêts des utilisateurs

Afin de déterminer les intérêts des utilisateurs, nous examinons quelles données ils utilisent et cherchons des mots-clés communs à l'ensemble des accès.
A chaque fois qu'un utilisateur fait un accès, on utilise les métadonnées associées à la donnée afin de mettre à jour *accesses* :
– si le mot-clé n'est pas présent dans *accesses*, on l'y ajoute en lui associant un compteur de 1,
– si le mot-clé est présent dans *accesses*, la valeur du compteur associé est incrémentée.
La construction des intérêts se fait dès lors de la manière suivante :

1. on calcule μ, le nombre moyen d'accès par mot-clé, et σ l'écart-type,
2. tout mot-clé dont le compteur est supérieur à $\mu + \sigma$ devient un intérêt.

6.2.2.3 Intérêts collaboratifs

Pour mettre en place de la réplication collaborative, on calcule des intérêts agrégés prenant en compte les intérêts de nos voisins. Cependant, tous les pairs n'ayant pas les mêmes capacités de stockage, on veut répliquer des données pour nos voisins uniquement si ceux-ci ont moins de capacité que nous. Chaque mot-clé agrégé est donc, par ailleurs, pondéré par les capacités des terminaux dont ils sont issus.
Soit l'ensemble de nos voisins à n sauts. Pour chaque voisin V on dispose de deux informations :
– sa liste d'intérêts $interests_V$,
– sa capacité (proportionnelle à l'espace mémoire dont il dispose) C_V.
Un terminal connaît par ailleurs sa propre capacité C.
On construit donc l'ensemble des mots-clés agrégés *CollaborativeInterests* :
– on place tout d'abord tous nos propres mots-clés dans *CollaborativeInterests*, en leur attribuant le poids 1,
– si $C \leq C_V$, on ne réplique pas pour le terminal.
– sinon, pour chaque mot-clé M dans $interests_V$:
  – si M est dans *CollaborativeInterests* on ajoute à son poids $p = \frac{C-C_V}{C}$,
  – sinon on l'ajoute dans *CollaborativeInterests* avec un poids initial de $p = \frac{C-C_V}{C}$.
  – comme, $C_V \leq C$, p est compris entre 0 et 1, ce qui assure que l'intérêt du pair local prime ; par ailleurs, plus la différence entre C et C_V est grande, et moins le voisin à de ressource, plus le pair local sera susceptible de répliquer pour lui.

6.2.3 Calcul d'intérêt potentiel

Soit UI l'ensemble des intérêts de l'utilisateur, et KW l'ensemble des mots-clés d'une donnée.

Pour prédire l'intérêt PI de l'utilisateur U de la donnée D, nous utilisons la métrique suivante :

$$PI = 0.5 * (\frac{|UI_U \cap KW_D|}{|KW_D|} + \frac{|UI_U \cap KW_D|}{|UI_U|})$$

Afin de décider de si une donnée doit être répliquée, il est nécessaire de définir un seuil au-dessus duquel la donnée est jugée intéressante. Nous utilisons comme valeur de seuil l'*intérêt moyen des données utilisées*.
Pour le calcul d'intérêt agrégé, la formule est sensiblement la même, mais prend en compte les poids des mots-clés. Soit AG les intérêts agrégés de la donnée, c'est à dire une liste de couple *(mot-clé, poids)*, et soit AG_KW l'ensemble de ces mots-clés.

$$PI = 0.5 * (\frac{|AG\_KW_U \cap KW_D|}{|KW_D|} + \frac{\sum_{poids} |AG\_KW_U \cap KW_D|}{\sum_{poids} AG})$$

6.3 Exemple : extrait d'informations sémantiques simples

Dans cette section, nous avons testé une méthode d'extraction d'informations sémantiques simples dans un wiki : les catégories servent de mots-clés pour un article. Nous avons fait nos tests à partir d'un dump de Wikipedia France.
Il est intéressant d'utiliser ici les catégories parce qu'elles représentent déjà un classement effectué par l'utilisateur humain. On peut donc se passer d'une ontologie.

6.3.1 Extraction d'un jeu de données

Tout d'abord, en partant sur un dump de Wikipedia, nous devons extraire les informations nous intéressant :
- Analyse des accès sur une période pour obtenir les intérêts des utilisateurs.
- Prédiction des accès en calculant les intérêts.
- Vérification des prédictions.

6.3.1.1 Données de départ

Pour faire nos calculs, nous nous basons sur un jeu de données généré à partir de Wikipedia France [4] :
- *stub-history.xml* contient pour chaque article un historique de l'ensemble des accès en écriture effectués par les utilisateurs depuis le début de frwiki. Nous nous en sommes servi pour extraire une liste d'accès par utilisateur. Au 22 juin 2010, ce fichier faisait 15.91 Go.
- *pages-article.xml* contient la version courante de l'ensemble des articles. Nous nous en sommes servi pour extraire une liste de mots-clés pour chaque donnée. Au 22 juin 2010, ce fichier faisait 6,39 Go.

Il est théoriquement possible de charger chacun de ces fichiers en mémoire grâce à une API DOM, puis de les interroger en XPath. En pratique, du fait de la taille des données, il était impossible de charger même un des deux arbres xml en mémoire. Nous avons donc tout d'abord utilisé des parsers SAX pour extraire et segmenter les informations requises. Notre procédé d'extraction est donc assez complexe.

6.3.1.2 Extraction des informations pertinentes

A partir de *stub_ meta_ history.xml*, nous avons extrait :
– la liste des articles ayant existés id/titre dans *usersList.xml* (220,6Mo),
– la liste des utilisateurs ayant existés id/nom dans *dataList.xml* (10,5Mo),
– pour chaque utilisateur *id*, un fichier *id.xml* listant par article les éditions effectuées (3 Go).
A partir de *pages-articles.xml*, nous avons extrait :
– la liste des articles avec pour chacun les catégories auxquelles il appartient *dataCatList.xml* (303,4Mo),
– la liste des catégories avec pour chacune ses surcatégories *categoryList.xml* (27,3 Mo).
Dans *dataCatList.xml*, les catégories extraites sont sous forme d'une chaîne de caractère. En nous basant sur ces deux fichiers nous avons pu construire les données suivantes :
– pour chaque article, un fichier contenant les identifiants des catégories auxquelles il appartient (1er degré), et les identifiants des surcatégories auxquelles ces catégories elles-même appartiennent (2eme degré).
– pour chaque catégorie, un fichier contenant les identifiants des articles appartenant à cette catégorie.

6.3.1.3 Quelques statistiques

Nous avons calculé quelques statistiques, présentées dans le tableau 6.1 :

Table 6.1 – Statistiques des éditions de wikipedia france depuis 2004

Nombre total de données	3 917 494
Nombre total d'éditeurs	276 980
Nombre de bots	580
Nombres d'administrateurs	189
Nombre total d'accès	28 338 685

La figure 6.1 (resp 6.2) représente en abscisse un nombre N de données modifiées (resp. un nombre N de modifications effectuées), et en ordonnée le nombre d'utilisateurs ayant modifié N données (resp. effectué N modifications). Deux lignes sont tracées, représentants la médiane et le 95 ème centile. On peut donc voir que :
– 50% des utilisateurs font moins de 3 accès, et 95% font moins de 99 accès.
– 50 % des utilisateurs éditent une seule donnée, et 95% éditent moins de 36 données.
La grande majorité des utilisateurs enregistrés ne font qu'une seule édition. Il n'est cependant pas dit que ces utilisateurs ne sont pas contributeurs, puisqu'ils peuvent avoir par ailleurs participé aux discussions.
Notre approche étant basée sur la détection de mots-clés basée sur un historique des accès, la majorité des utilisateurs ne sont donc pas utiles à sa validation. Nous avons donc sélectionné pour nos tests un groupe d'utilisateurs, selon les critères suivants :
– utilisateurs non bot,
– utilisateurs non administrateurs,
– nombre d'accès le plus élevé, parmi ceux inférieurs à 20.000.
Notons que les articles étant créées par des utilisateurs humains, certaines erreurs sont présentes. Certaines, comme l'absence d'une majuscule au début d'un lien (nécessaire dans

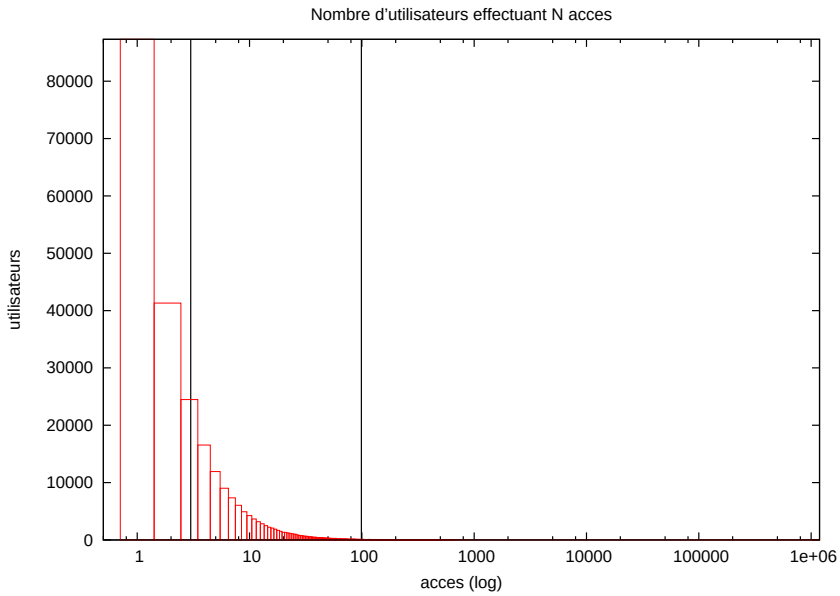


FIGURE 6.1 – Nombre d'utilisateurs effectuant N accès

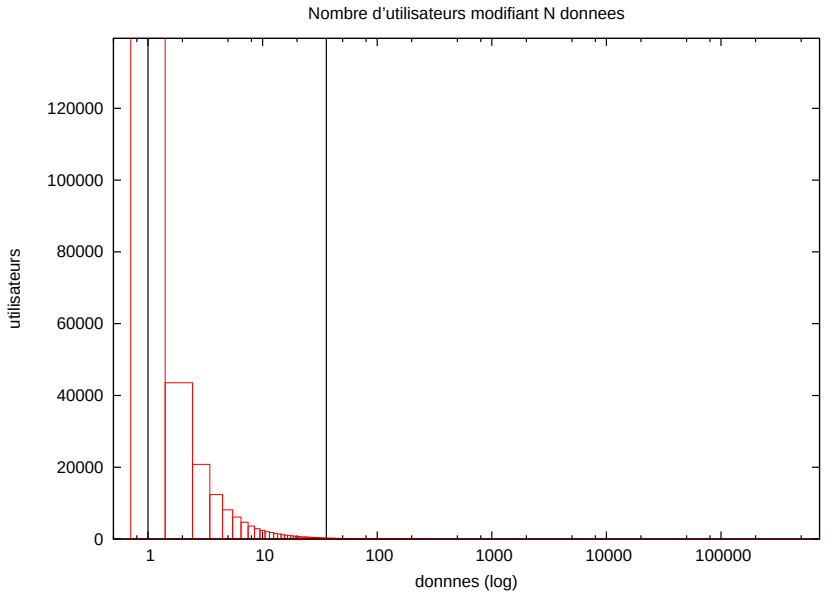


FIGURE 6.2 – Nombre d'utilisateurs modifiants N données différentes

la syntaxe wiki), peuvent être corrigées automatiquement mais d'autres, comme les fautes

d'orthographes sont trop complexes à corriger automatiquement. Une partie des informations sémantiques liant les données ont donc été perdues, mais le taux d'erreur est faible.

6.3.2 Evaluation

6.3.2.1 Protocole de validation

Pour valider notre proposition, nous sommes partis des éléments suivants, construits comme décrit dans la section précédente :
- une liste des 300 utilisateurs ayant effectué le plus d'accès, tels que leur nombre d'accès ne dépasse pas 20.000, et qu'ils ne soient ni administrateurs ni bot : *selected.users*,
- pour chacun de ces utilisateurs u, une liste de leurs accès triés par date (ordre croissant) *u.sorted.accesses*,
- un index : pour chaque donnée d, une liste de mots-clés, de 1er degré et de 2ème degré *d.keywords*,
- un index inverse : pour chaque catégorie c , une liste des données pour qui c est un mot-clé *c.data*.

6.3.2.2 Evolution de l'intérêt

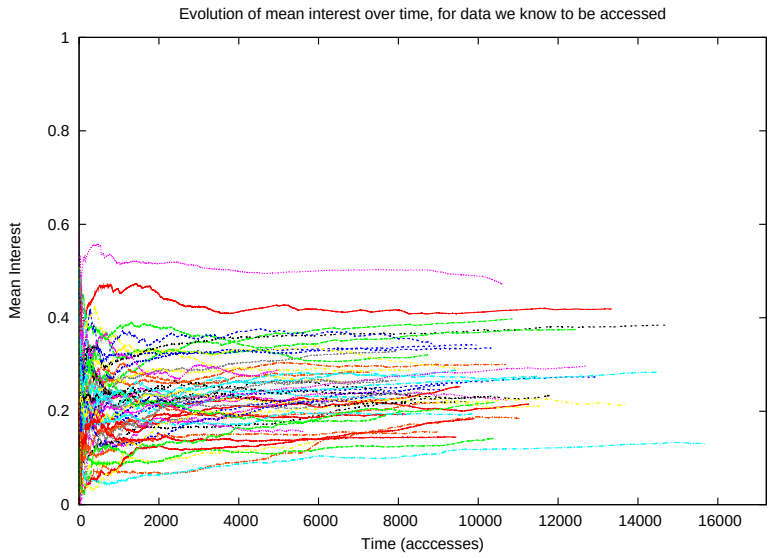
Nous avons tout d'abord, sur une liste de données dont on sait qu'elles intéressent l'utilisateur, calculé l'évolution du calcul de l'intérêt potentiel en fonction du nombre d'accès utilisés pour construire l'ensemble Intérêts.
Pour chaque utilisateur U, on applique donc le traitement suivant :

1. On charge la liste d'accès *U.sorted.accesses*.
2. On construit l'ensemble des données qu'il va éditer.
3. Pour chacune de ces données d, la liste des mots-clés associés, stockée dans *d.keywords* est chargée en mémoire.
4. Les 2000 premiers accès sont utilisés pour construire un ensemble d'intérêts de départ.
5. On parcourt ensuite la liste d'accès dans l'ordre chronologique, pour chacun :
   (a) On calcule l'intérêt potentiel associé à la donnée .
   (b) On utilise les mots-clés de la donnée pour mettre à jour les intérêts.
6. L'ensemble final d'intérêts est sauvegardé.

Chaque courbe dans les graphes 6.3 à 6.6 représente l'évolution de l'intérêt d'un utilisateur pour les données qu'il accède réellement. L'abscisse représente les accès, ordonnés par date, et l'ordonné présente l'intérêt moyen de l'utilisateur, mis à jour suite à cet accès.
On peut voir que l'intérêt moyen des utilisateurs pour les données qu'ils utilisent se stabilise. Nous pouvons donc utiliser cette moyenne comme discriminant pour les nouvelles données.

6.3.2.3 Erreurs

La section précédente nous a permis de voir qu'après un certains nombre d'accès, l'intérêt moyen de l'utilisateur pour l'ensemble des données qu'il a modifiées ne varie plus. On peut donc s'en servir comme seuil pour discriminer les données intéressantes.
En utilisant les mots-clés construits dans l'expérience décrite précédemment, nous avons donc calculé quatre valeurs :



GURE 6.3 – Evolution des intérêts au cours du temps

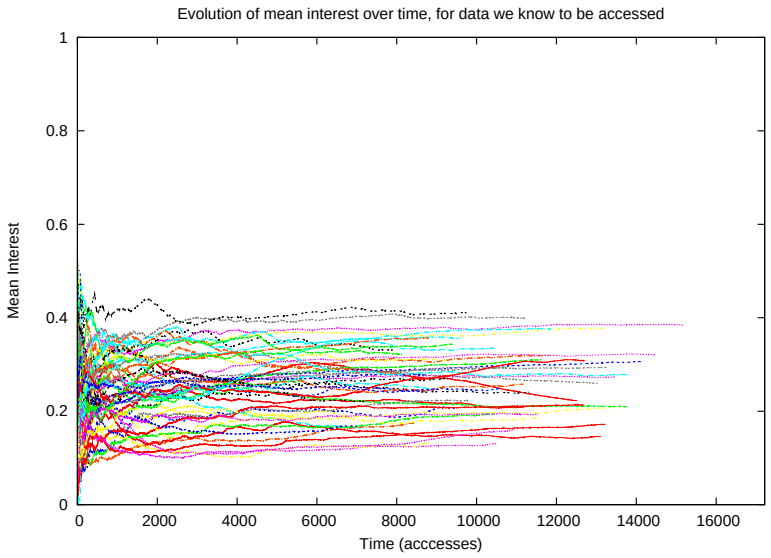
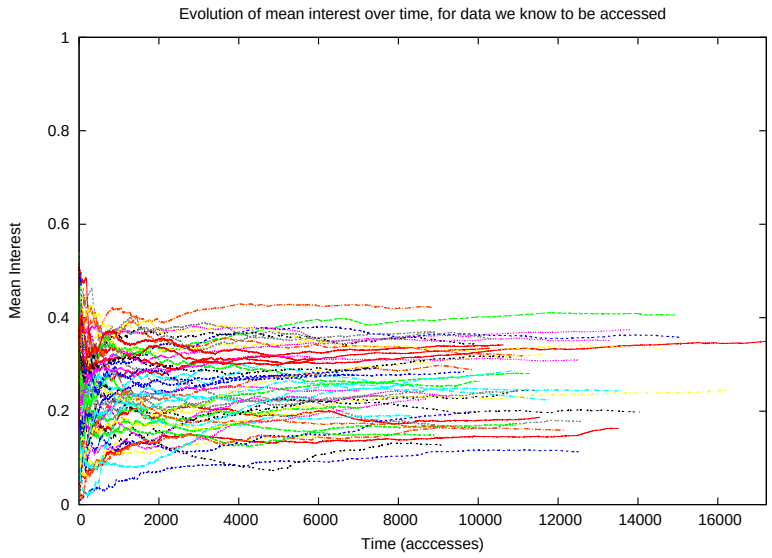


FIGURE 6.4 – Evolution des intérêts au cours du temps



GURE 6.5 – Evolution des intérêts au cours du temps

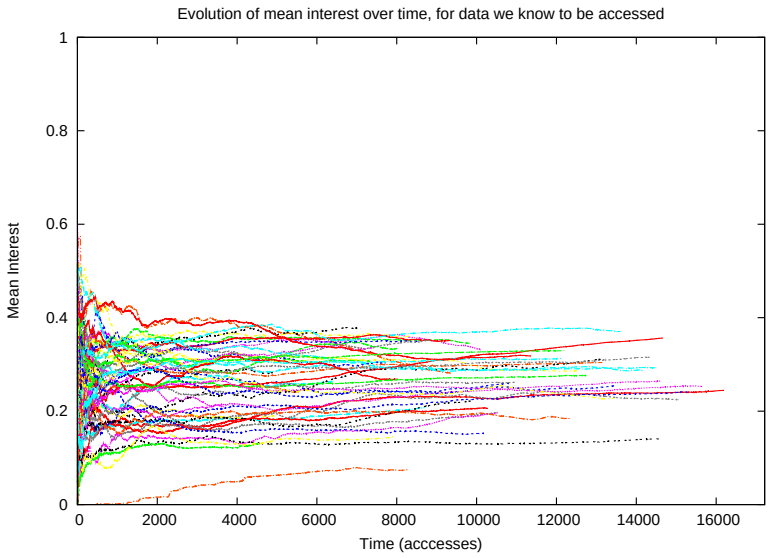


FIGURE 6.6 – Evolution des intérêts au cours du temps

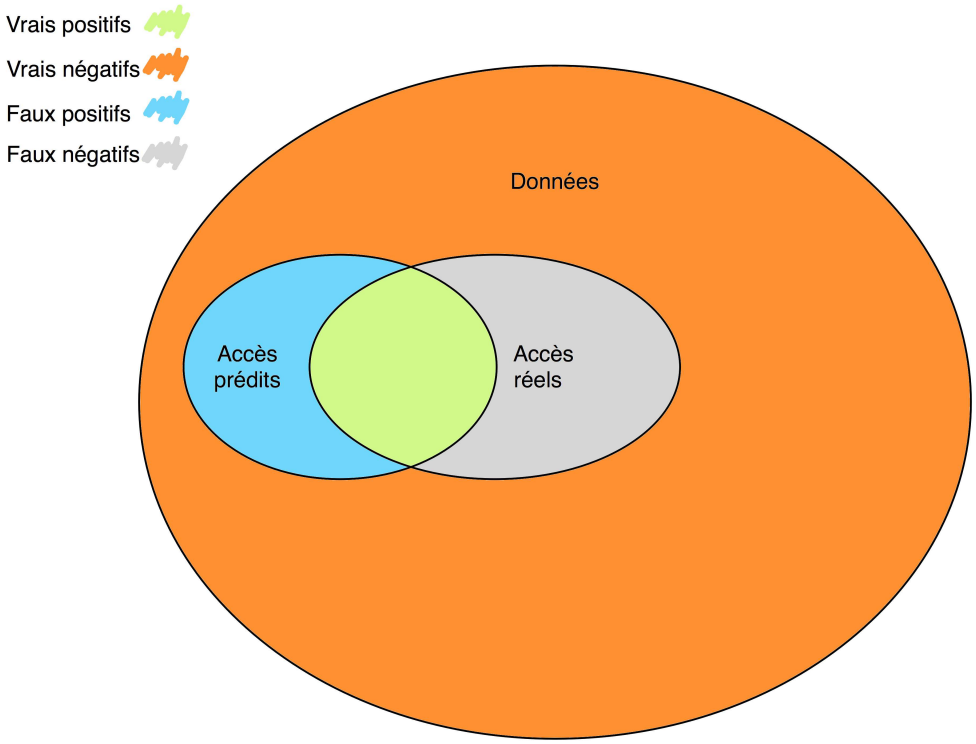


Figure 6.7 – Distribution des erreurs

– nombre de Vrais Positifs : les données dont on a prédit l'accès, et qui sont effectivement éditées.
– nombre de Vrais Négatif : les données dont l'algorithme ne considère pas qu'elles sont intéressantes, et qui ne sont effectivement pas éditées.
– nombre de Faux Positifs : les données détectées comme intéressantes pour l'utilisateur, mais qu'il n'a pas modifiées.
– nombre de Faux Négatifs : les données considérées comme non intéressantes, mais qui sont en fait éditées

On ne peut pas charger l'ensemble des associations donnée/mots-clés en mémoire. Ce calcul a donc été fait de la manière suivante :

1. On connaît N, le nombre total de données.
2. On charge en mémoire l'index inverse.
3. pour chaque utilisateur dans *selected.users* :
   (a) On charge la liste d'intérêt *interests*.
   (b) On charge la liste des données éditées *actual*.
   (c) A partir de l'index inverse, on construit *potential*, l'ensemble des données ayant au moins un mot-clé en commun avec *interests*.
   (d) Pour chaque donnée dans *potential*, on détermine si elle est intéressante, et on construit donc ainsi l'ensemble *theorical*.
   (e) On détermine ensuite les valeurs suivantes :
   – $True\ Positive = |theorical \cap actual|$,
   – $False\ Positive = |theorical \setminus actual|$,

- $False\ Negative = |actual \setminus theorical|$,
- $True\ Negative = N - |theorical \cup actual|$.

Dans les figures 6.8 à 6.11, l'abscisse représente le nombre moyen d'accès par donnée pour le pair. La figure 6.8 représente le pourcentage de Vrais Positifs parmi les Positifs et la figure 6.9 représente le pourcentage de Vrais Négatifs parmi les Négatifs . La figure 6.10 représente le pourcentage de données utilisées qui sont détectées comme intéressante et la figure 6.11 représente le pourcentage de données non utilisées qui sont détectées comme telles.
On voit que notre algorithme est très efficace pour détecter les données n'intéressant pas le pair : quand une donnée est rejetée par l'algorithme, on voir sur 6.9 pour qu'en moyenne 99,95% des cas, il est correct. Parmi les données non utilisées, en moyenne 96,11% sont rejetées.
Cependant, le nombre de données non utilisées est bien plus important que le nombre de données utilisée, et les 3.9% restant représentent donc une part non négligeable.
Le résultat est donc moins probant pour les données intéressants les pairs. Le graphe 6.10 montre que les données intéressantes sont détectée comme intéressant pour 62,49% en moyenne. Le graphe 6.8 montre que parmi les données détectées comme intéressante, la portion de Faux Positif est très importante : en moyenne, seulement 2% des données détectées comme intéressantes sont effectivement accédées.

6.3.3 Discussion des résultats

A partir de l'historique de wikipedia, nous avons tenté de construire pour un ensemble d'utilisateurs un ensemble de mots-clés capable de décrire les données qu'ils ont utilisées. Nous avons utilisé une technique simple, en utilisant les catégories comme mots-clés.
Cette technique a donné des résultats positifs sur certains points :
- l'extraction des informations sémantique est simple, il suffit de rechercher dans le texte les balises correspondant aux catégories.
- les structures de données associées et les calculs sont simple et peu coûteux.
- les données non intéressantes sont bien détectées (99,95%)
- les données intéressantes sont bien détectées (62,49%)

Cependant, cette technique créée beaucoup de Faux Positifs. Ces erreurs peuvent avoir plusieurs origines :
- Notre critère de sélection de mots-clés pour les données ou les utilisateurs n'est pas assez restrictif.
- Nos mots clés sont trop restrictifs, on pourrait utiliser les liens, faire de l'analyse de texte.
- La base de test ne prend en compte que les accès en écriture.
- Les accès récents devraient avoir plus de poids que les accès anciens dans le calcul des intérêts.
- Les utilisateurs faisant beaucoup d'accès sont plus susceptibles d'être des contributeurs wikiElf, wikiFairy ou wikiGnome (terminologie wikipedia), qui veillent au bon fonctionnement du wiki en corrigeant la syntaxe ou l'orthographe, trouvant les auteurs de citations, organisant les articles, créant de nouveaux templates, etc. On voit d'ailleurs sur le graphe 6.8 que le nombre moyen d'accès par donnée et par utilisateur tend vers 10, ce qui est faible au vu du nombre d'accès qu'ils effectuent, entre 10809 et 19997. Dans un tel cas, leurs accès ne sont pas fait sur un critère d'intérêt, mais sur un critère de fraîcheur (ils examinent les données qui viennent d'être modifiées).

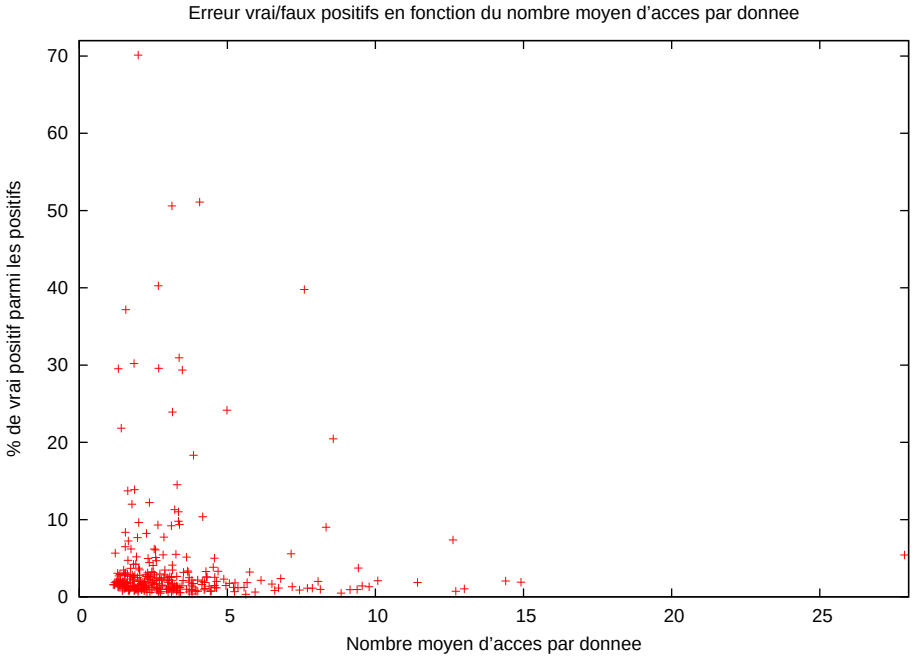


Figure 6.8 – Portion de vrais positifs parmi les positifs

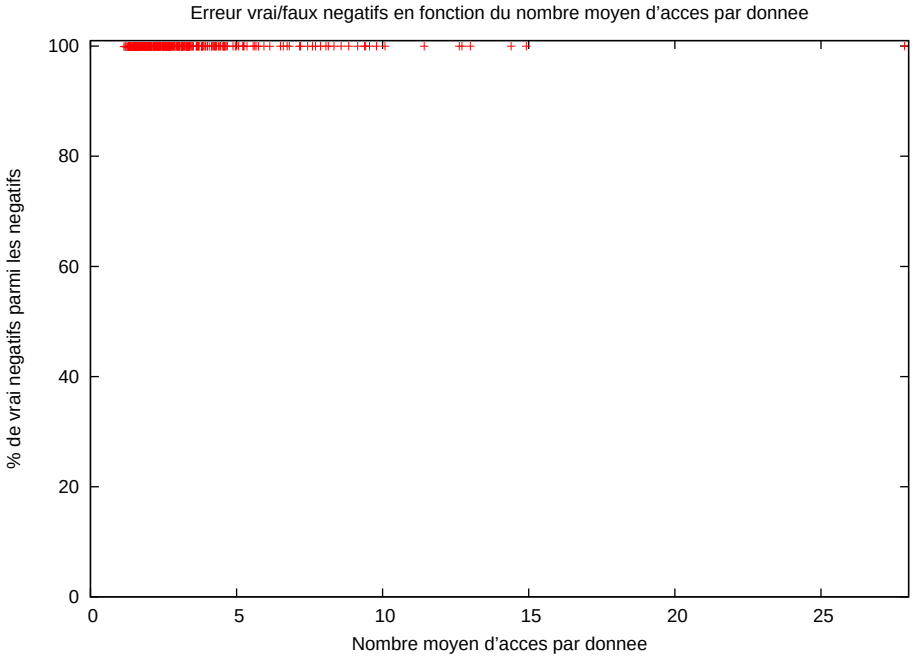


Figure 6.9 – Portion de vrais négatifs, parmi les négatifs

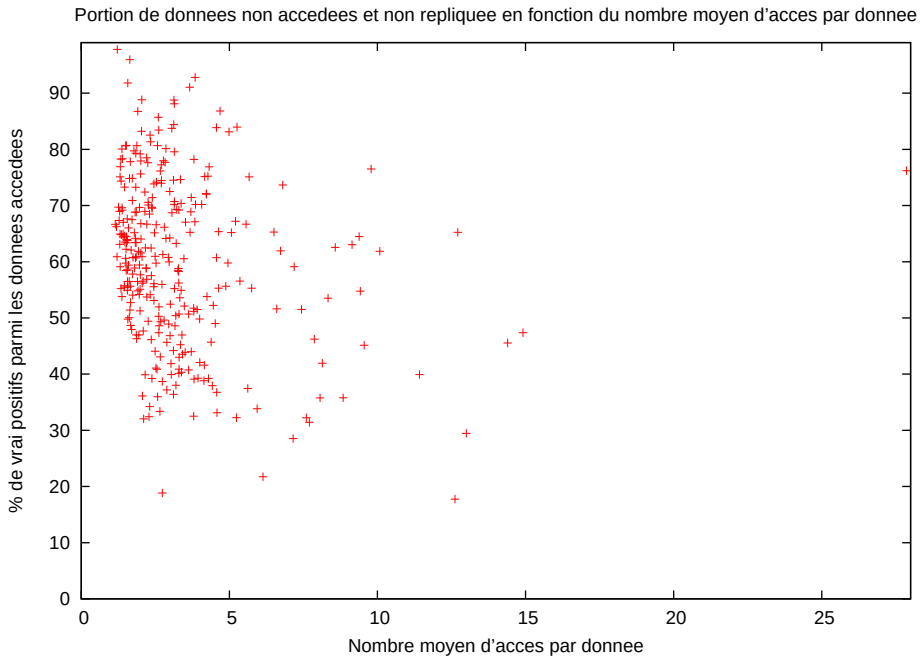


FIGURE 6.10 – Portion de vrais positifs parmi les données utilisées

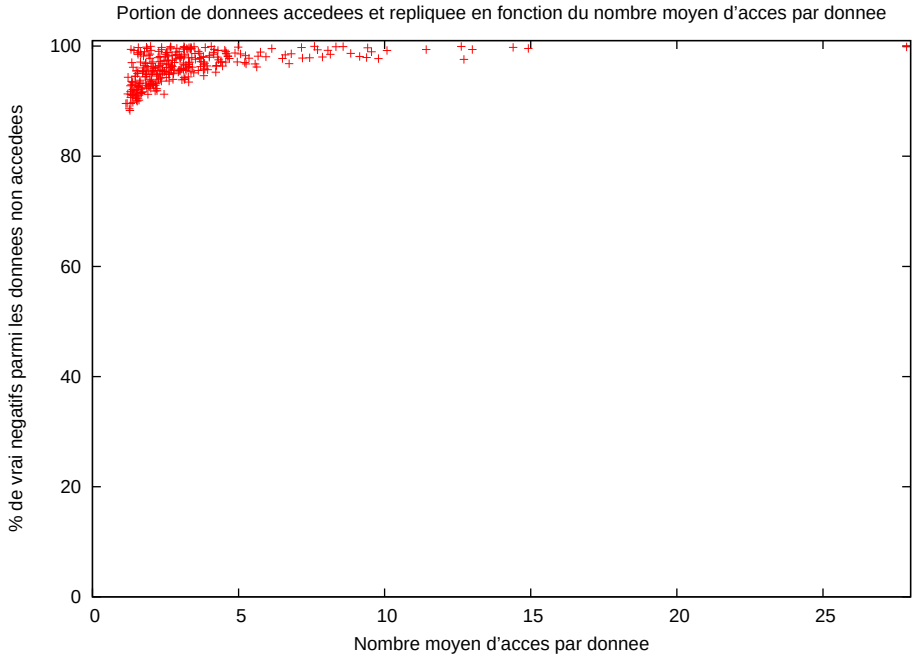


FIGURE 6.11 – Portion de vrais négatifs, parmi les données non utilisés

Enfin, ces résultats sont liés à la base de données utilisée ici, et le comportement de ses utilisateurs.

6.4 Paramètres, et modules extérieurs

Dans cette section nous présentons tout d'abord comment nous avons modélisé l'espace mémoire disponible pour répliquer les données.
Nous voyons ensuite les modules extérieurs utilisés par notre algorithme, et les services qu'ils offrent : la gestion des intérêts des utilisateurs, la gestion du réseau par construction de groupes stables, et enfin un module simple de localisation.

6.4.1 Modélisation de la mémoire

Afin de valider nos algorithmes de réplication de données, et de remplacement de cache dans le chapitre suivant, nous discutons ici de comment modéliser la mémoire et son occupation.

6.4.1.1 Données

Afin de modéliser le nombre maximum de données qu'un utilisateur peut stocker sur son terminal, il convient de décider de comment modéliser l'espace mémoire d'une donnée.
Ceci dépend beaucoup de si la donnée est modifiable. Par exemple, l'ensemble des pdfs constituant l'état de l'art de cette thèse fait environ 250 Mo. La taille de chaque fichier varie entre 150ko et 2,5Mo selon le nombre d'images, la plupart pesant autour de 400ko.
Si on prend pour exemple un article de taille moyenne (12 pages) écrit en latex, ses sources occupent 1,5Mo, dont 100ko de texte, pour un document final de 654ko. On a donc un rapport de 3 entre les sources et le pdf final. L'intégralité des sources de notre état de l'art occuperait donc 750Mo.
Une donnée chargée en mémoire dans un éditeur de texte prend par ailleurs encore plus d'espace à cause des données de gestion, comme par exemple les identifiants de chaque atome dans WOOT.
Dans les modèles utilisés pour la simulation, nous considérons donc qu'une donnée, avec toutes les informations nécessaires à la gestion de la cohérence, de la réplication et du remplacement de cache, occupe 1Mo.

6.4.1.2 Terminaux

Après avoir caractérisé la taille nécessaire pour stocker une donnée, il nous faut caractériser les capacités de stockage des terminaux, afin de déterminer la taille du cache disponible.
Le OLPC [42], un terminal construit pour fonctionner en MANet, possède un Go de mémoire secondaire, sous forme de SSD. D'autre part, on trouve facilement de nos jours des ordinateurs portables avec quelques centaines de Go de mémoire secondaire. D'autres applications utilisant la mémoire, nous nous limitons dans ce cas à 1Go, ce qui est déjà considérable si on considère en majorité des fichiers textes.
On considère donc que la taille du cache que s'octroie notre application est limitée, et qu'elle varie entre 50 Mo et 1Go.

Par la suite, nous modélisons la taille du cache de données par un entier, correspondant au nombre d'emplacements disponibles en mémoire, avec un emplacement correspondant

à 1Mo. Nous considérons que les données manipulées sont toutes de la même taille (1Mo), et occupent chacune un emplacement.
Un cache maximum de 50Mo est donc modélisé par un scalaire 50, un cache maximum de 1Go par un scalaire 1000. Dans les expériences présentées par la suite, nous considérons donc une taille de cache variant entre 50 et 1000.
Cette taille du cache est par ailleurs utilisée pour modéliser les ressources des terminaux au moment de la réplication collaborative.

6.4.2 Modules extérieurs

Dans cette section, nous présentons les fonctionnalités utilisées par l'algorithme proposé. Il s'appuie sur le module gérant les intérêts des utilisateurs que nous avons présenté plus haut, le module gérant la mobilité des terminaux, que nous avons présenté dans le chapitre précédent, et le service de nommage/localisation.
Du point de vue de l'algorithme de réplication, ces modules sont des boîtes noires dont on ne connaît pas le fonctionnement. Ainsi, dans le calcul du coût en nombre de messages par exemple, on ne prend pas en compte les messages nécessaires à la localisation des données, car leur nombre dépend de l'implantation de ce module.

6.4.2.1 Utilisateurs

Nous avons vu dans la première partie de ce chapitre comment les accès des utilisateurs étaient analysés pour en extraire des mots-clés caractérisant leurs intérêts.
Ces mots-clés, regroupés dans la structure de données *UserInterestsSet* sont utilisés dans l'algorithme de réplication. On utilise aussi des mots-clés décrivant les intérêts de nos voisins, regroupés dans *CollaborativeInterestsSet*.
On dispose donc d'un module *Intérêts* offrant les méthodes suivantes :
- *computeSelfInterest(metadata)* : calcule l'intérêt d'un utilisateur pour une donnée.
- *computerCollaborativeInterest(metadata)* : calcule l'intérêt d'un utilisateur à répliquer la donnée pour ses voisins.
- *shouldReplicate(interest)* : sachant un niveau d'intérêt pour une donnée, indique s'il est intéressant de la répliquer.

Par ailleurs, le module *Intérêts* offre une interface *updateOnAcc(metadata)*, permettant d'indiquer quand une donnée est utilisée, quels sont ses mots-clés associés, afin de mettre à jour le dictionnaire *accesses*.

6.4.2.2 Groupes de mobilité

Pour faire de la réplication collaborative, et mettre en place un algorithme de mise à jour hybride, nous nous basons sur les groupes de mobilité construits par l'algorithme présenté dans le chapitre précédent.
Nous disposons donc d'un module de création de groupe stable, *StableGroup* qui expose les méthodes suivantes :
- *stableGroupList()* : retourne la liste des membres du groupe.
- *howMany()* : retourne le nombre de membres du groupe.
- *isNeighbour(peerID)* : indique si *peerID* fait partie du groupe.
- *nHopNeighbours(n)* : retourne les membres du groupe de mobilité, situés à moins de n sauts.

6.4.2.3 Localisation d'une réplique

On dispose d'un module de localisation, qui nous permet de trouver une réplique d'une donnée à partir de son identifiant. Dans ces travaux, nous ne proposons pas d'approche innovante pour la localisation des répliques. Cependant, comme il est nécessaire d'avoir un module de localisation pour tester les autres algorithmes, nous utilisons dans les simulations un module élémentaire avec le fonctionnement suivant :
– Quand on reçoit une information sur l'existence d'une réplique, on la place dans un cache de localisation.
– Quand on veut accéder à une donnée, on consulte tout d'abord le cache local.
– Si on n'a pas d'information locale, on inonde le réseau.
Dans l'implantation des algorithmes au sein de Transhumance, on utilisera le système d'événements, au comportement optimisé.
Le module Localisation expose donc la méthode *locate(dataID)*, qui retourne l'identifiant d'un hôte hébergeant la donnée.

6.5 Réplication

Nous travaillons dans un contexte où les terminaux sont volatils, et où le temps de travail est borné par la durée de vie de la batterie des terminaux.
Notre algorithme de réplication a donc deux objectifs :

1. S'assurer de la disponibilité des données, c'est à dire diminuer le temps d'accès à la donnée, et surtout s'assurer que celle-ci ne disparaît pas.
2. Diminuer la charge réseau, afin d'allonger la vie de la batterie

Ces deux objectifs ne sont pas orthogonaux : pour rendre la donnée plus disponible, on va créer des répliques. Dans le cas où la donnée n'est pas modifiée, cela diminue le trafic réseau puisqu'une fois répliquée, la donnée n'aura plus à transiter sur le réseau. Une donnée modifiable crée, en revanche, du trafic réseau lié aux mises à jour des différentes répliques.
L'algorithme de réplication vise essentiellement la première problématique : on crée des répliques afin de prévenir la disparition des données et de les rapprocher de leurs utilisateurs. La limitation du trafic réseau est gérée par l'algorithme de remplacement de cache, que nous présentons dans la section suivante.
Nous allons voir ici quels événements sont susceptibles de faire appel à la politique de réplication, et comment celle-ci est alors définie.

6.5.1 Création d'une nouvelle donnée

À la création d'une donnée, on veut créer d'emblée assez de répliques pour éviter que la donnée ne disparaisse, par exemple si son créateur disparaît avant qu'une réplique à la demande n'ai été créée.
Pour cela, nous voulons faire de la réplication à r% : pour M pairs dans le réseau, il doit y avoir au minimum $\frac{r*M}{100}$ répliques de chaque donnée. Nous voyons dans la section de validation comment n est fixé, et que cela nous permet bien de limiter la disparition des données.
Quand une donnée est créée, la source diffuse un message dans le réseau avec les informations nécessaires pour prédire l'intérêt. Chaque terminal passe alors par les étapes suivantes :

Réplication égoïste : on réplique tout d'abord pour soi :
- Chaque pair calcule son intérêt personnel pour la donnée, si celui-ci est supérieur à un seuil (que nous avons fixé à la moyenne des intérêts des données utilisées jusque là).
- Si un pair crée une réplique, il diffuse un message pour l'annoncer.
  1. Il demande la donnée au pair l'ayant créée.
  2. Il diffuse l'information de création dans le groupe stable.
- Chaque pair arme un temporisateur correspondant au temps de traversée du réseau, soit la taille de la plus longue route.

Réplication collaborative : quand le temporisateur se déclenche, chaque pair comptabilise le nombre de répliques :
- Si celui-ci est suffisant, l'algorithme de réplication s'arrête.
- Sinon, on vérifie s'il est intéressant de répliquer pour nos voisins.

Chaque pair arme un nouveau temporisateur.

Réplication statistique : si à l'issue des deux premières étapes, il n'y a pas assez de répliques, on place les répliques manquantes au hasard dans le réseau :
- k copies ont été créées pour chaque donnée ; pour M terminaux, avec un taux de réplication de r, il manque donc M*r-k copies.
- Chaque pair n'étant pas déjà hôte, réplique donc la donnée avec une probabilité de $\frac{M*r-k}{M-k}$.
- Si un pair réplique une donnée, il diffuse cette information dans le réseau.
- Chaque terminal arme une alarme, et quand il se réveille effectue à nouveau le test de réplication statistique jusqu'à ce que le nombre de répliques voulu soit atteint.

6.5.2 Création d'une réplique

En dehors de la phase de création initiale, d'autres répliques peuvent être créées, comme nous allons le voir dans les sections suivantes.
Quand un hôte fournit une réplique d'une copie à un pair, il y joint *hosts*, la liste des hôtes déjà existants. Le nouvel hôte envoie ensuite un messages aux pairs de *hosts* existants afin d'être inclus dans les échanges de mises à jour.

6.5.3 Accès à une donnée

Une donnée peut être utilisée en lecture, ou en écriture. Quand la donnée n'est pas présente localement sur le terminal, celui-ci fait une requête à un hôte distant, en lui indiquant le type d'accès (lecture ou écriture).
Comme nous l'avons indiqué plus tôt, afin de mettre en place un modèle de cohérence à terme, nos données sont structurées sous la forme d'un arbre TreeDoc.
Quand la donnée est utilisée en écriture, on doit nécessairement rapatrier une copie locale de l'arbre TreeDoc et créer une réplique.
Quand une donnée est utilisée en lecture, se pose en revanche la question de savoir si l'hôte demandeur doit créer une nouvelle réplique. L'hôte recevant la requête vérifie le nombre d'hôtes pour la donnée :
- Si celui est supérieur au nombre de répliques requis, il envoie une version linéarisée de la donnée, c'est à dire la chaîne de caractères plutôt que la structure de données TreeDoc.
- Sinon, il envoie l'arbre TreeDoc en demandant au demandeur de créer une nouvelle réplique. Celui ci peut refuser.

6.5.4 Suppression d'une réplique

Quand on reçoit un message indiquant la suppression d'une réplique, celle-ci peut avoir été causée par deux situations : élimination pour libérer de l'espace dans le cache, ou élimination pour diminuer la charge réseau. Nous présentons dans le chapitre suivant comment et quand ces éliminations sont effectuées par l'algorithme de gestion du cache.
On ne crée pas de nouvelles répliques quand on reçoit un message nous informant de la suppression d'une copie. En effet, comme nous l'exposons dans le chapitre suivant, l'élection de la donnée à éliminer d'un cache se fait sur le nombre de copies existantes. On suppose donc que le pair ayant choisi d'éliminer cette donnée a donc fait au mieux. Cependant, comme indiqué dans la section précédente, de nouvelles répliques peuvent être créées en cas d'accès.

6.5.5 Disparition d'un hôte

Quand un hôte disparaît, ou lors d'une partition d'un groupe stable, pour chaque donnée la tâche incombe à l'hôte ayant le plus petit identifiant de vérifier s'il y reste suffisamment de répliques.
Si le nombre de répliques restantes est insuffisant, il envoie une demande de création statistique des k répliques manquantes.

Ces propositions sont évaluées dans la section suivante.

6.6 Evaluation

Dans cette section, nous évaluons l'algorithme de réplication collaboratif pro-actif.
Nous voyons tout d'abord la pertinence de chercher à créer un nombre de répliques en fonction du nombre de terminaux, plutôt que d'avoir un nombre de copies fixes, puis comment le taux de réplication est choisi. Nous calculons alors le temps nécessaire à la partie probabiliste de l'algorithme pour converger. Nous présentons ensuite le gain de la réplication sémantique.
Enfin, nous présentons le surcoût en terme de charge réseau lié à la réplication pro-active.

6.6.1 Réplication à n : pertinence

Dans notre proposition, nous voulons créer préventivement des données afin qu'en cas de perte de contact dûe à la mobilité, les données restent disponibles.
Quand on crée ces répliques se pose la question de savoir le nombre de répliques à créer. Nous proposons de créer au sein d'un groupe un nombre de répliques proportionnel au nombre de pairs dans le groupe.
Afin de vérifier la pertinence de cette proposition, et de fixer un taux de réplication, nous avons examiné plusieurs modèles de réplications :
– Nombre de répliques au prorata de la taille du groupe, différents taux de réplication.
– Nombre de répliques fixes, différents nombres de répliques.
Pour ce faire, nous avons simulé plusieurs scenarii dans lesquels les données sont répliquées soit avec un nombre fixe de répliques, soit avec un nombre de répliques au prorata du nombre de terminaux, puis dans lequel un événement de mobilité survient susceptible de faire disparaître des données. Nous avons ensuite calculé, pour chaque scénario, combien de données sont perdues.

Chaque scénario de simulation est décrit par 4 paramètres :
– un nombre de pairs,
– un nombre de données,
– un événement de mobilité (disparition de n pairs, ou séparation en k groupes),
– un modèle de réplication (nombre fixe de répliques, ou pourcentage de réplication).
Une simulation se déroule de la manière suivante :

1. On calcule le nombre de répliques par pair.
2. Les données sont ensuite placées au hasard sur les terminaux, de manière à ce que le nombre de répliques par terminal soit uniforme.
3. L'événement de mobilité est joué, suivant deux scenarii possibles :
   – N terminaux sont choisis au hasard et éliminés du groupe.
   – Le groupe est séparé en k nouveaux groupes, dans lesquels les terminaux sont placés au hasard, mais de manière à ce que les groupes soient de tailles égales.
4. Le taux de couverture (i.e., le nombre de données encore accessibles suite à l'événement de mobilité par le nombre de données accessibles avant) est calculé.

Chaque scénario est répété 10000 fois, et on calcule le taux de couverture moyen suite à l'événement de mobilité.
Ici nous évaluons deux types d'événements :
– N pairs disparaissent simultanément.
– Un groupe se sépare, en k nouveaux groupes.

6.6.1.1 Disparition simultanée de N pairs

Dans ce jeu de tests, nous voulons voir quel est le taux de données perdues suite à la disparition d'un pair.
Nous avons fait varier les paramètres suivants :
– Nombre de données : dans l'ensemble décrit par NbData= {100} ∪ {k*1000 pour k dans [1-9]}.
– Nombre de pairs dans le groupe dans l'ensemble décrit par NbPeers ={k*20+10 pour k dans [0-6]}.
– Nombre de pairs quittant le groupe, dans l'ensemble NbLeaving=[1-10].
Tout d'abord, nous avons pu constater que le nombre de données, si ce nombre est compris entre 1000 et 9000, n'influence pas le taux de couverture final pour un taux donné de réplication, ou un nombre fixe donné de répliques. C'est logique, puisque dans ces scenarii, nous n'avons pas borné la taille des caches (ceci est évalué dans la section sur le remplacement de cache), et que les données sont répliquées statiquement au début de la simulation.
Les graphes 6.12 et 6.13 le montrent : ces graphes représentent le taux de couverture au terme d'une expérimentation en fonction du nombre de répliques ou du taux de réplication. Les courbes pour 1000 à 9000 données se confondent : la proportion de données perdues est donc la même. Bien entendu, la quantité de données perdues est plus élevée s'il y a plus de données au départ.
Bien que nous ayons fait les tests pour l'ensemble NbData, nous ne présentons donc par la suite que les résultats obtenus pour 5000 données.
Les figures 6.14 à 6.19 représentent les résultats de ces expériences. Chaque ligne correspond au nombre de terminaux quittant simultanément le groupe. La figure du haut correspond aux expérimentations avec un nombre fixe de répliques, tandis que la figure du

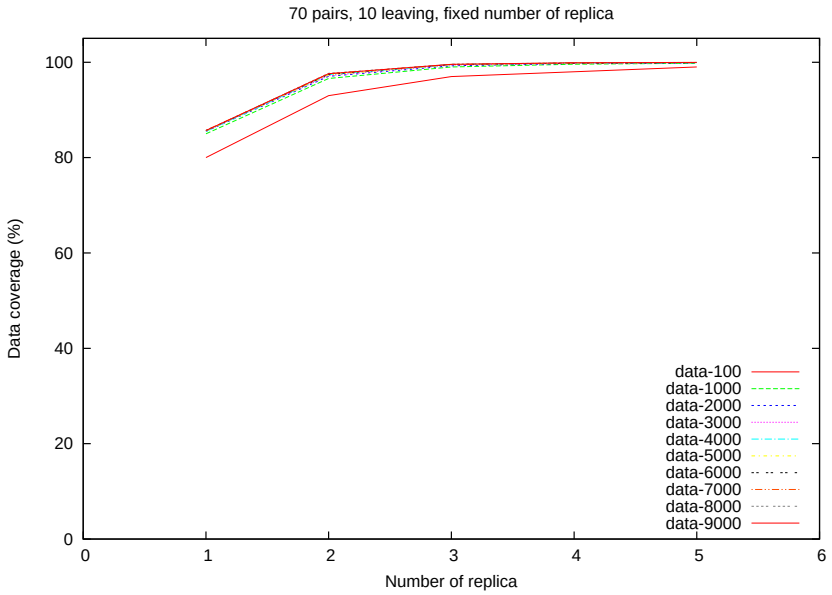


Figure 6.12 – Influence du nombre de données sur le taux de couverture suite à un départ de pairs, nombre fixe de répliques

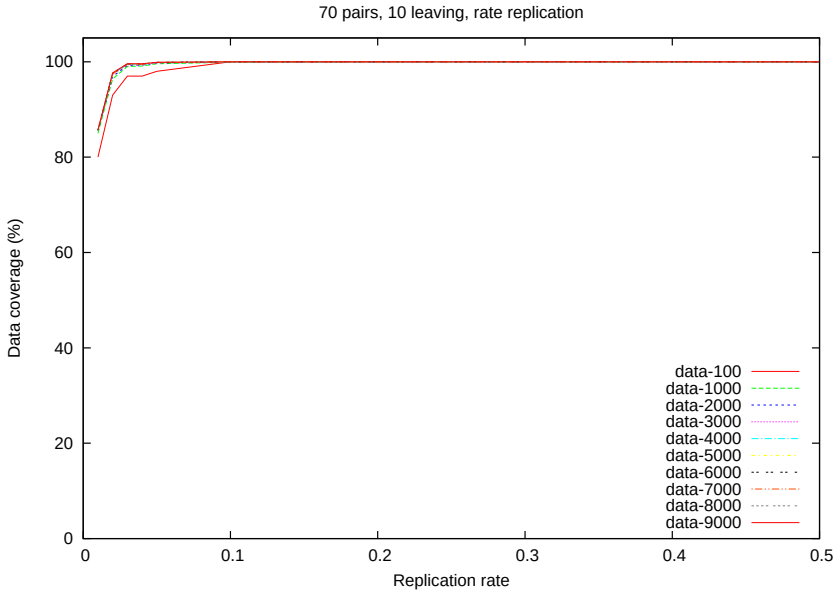


Figure 6.13 – Influence du nombre de données sur le taux de couverture suite à un départ de pairs, répliques au prorata

bas correspond aux expérimentations avec un nombre de répliques qui est un pourcentage du nombre de terminaux.
Bien qu'ayant effectué l'ensemble des calculs indiqués ci-dessus, pour ne pas alourdir ce chapitre, nous ne montrons les résultats que pour 1, 5 et 10 pairs disparaissant.
On peut voir ici qu'à partir de 3 répliques fixes, le système résiste aussi bien au départ de pairs qu'avec un taux de réplication, qui génère un nombre de réplique variable.
Nous voyons dans la section suivante pourquoi un nombre fixe de répliques ne convient pas.

6.6.1.2 Séparation en plusieurs groupes

Nous avons vu dans la section précédente que quand il s'agit de supporter la disparition simultanée d'un certain nombre de pairs, avoir un nombre fixe de répliques pour le système fonctionne bien. Dans ce cas, nous supposons que les pairs qui disparaissent arrêtent de travailler. On ne prend donc pas en compte les données auxquelles ils peuvent encore accéder.
Cependant, dans les MANets, il est possible que la disparition de pairs ne soit pas liée à la fin d'une session de travail, mais à une perte de communication, par exemple à cause de la mobilité. Chaque partie du réseau entend donc continuer à pouvoir travailler.
Nous avons donc répété les expériences de la section précédente, mais cette fois les événements de mobilité sont des partitions du groupe. A l'issue de la séparation, le taux de couverture est calculé pour chacune des parties du réseau.
Tout comme pour l'expérimentation précédente, on voit sur 6.20 et 6.21 que le nombre de données n'influence pas les résultats. Nous présentons donc, par la suite, les résultats pour un nombre de données fixé à 5000.
Les figures 6.23 à 6.28 montrent la couverture des données suite à un événement de mobilité. La figure du haut représente les simulations ayant utilisé un taux de réplication, et la figure du bas représente les simulations ayant utilisé un nombre fixe de répliques.
On voit ici que la réplication au prorata du nombre de pairs offre une meilleure approximation du nombre nécessaire de répliques pour éviter la perte de données. Le taux de perte de données se stabilisant autour de 0.15, nous avons donc fixé le taux de réplication r à 15%.

6.6.1.3 Taux de réplication, nombre de données et taille du cache

Soit N donnéeset p nœuds, ayant chacun un cache de taille c. Quel est le taux de réplication maximum r possible ?
Le nombre de répliques par donnée est r*p. Le nombre total de réplique est donc r*p*N. Comme l'espace mémoire total disponible est de p*c, on peut en déduire l'inégalité suivante :

$$r * p * N \leq p * c$$

Le taux de réplication maximum est donc borné par :

$$r \leq \frac{c}{N}$$

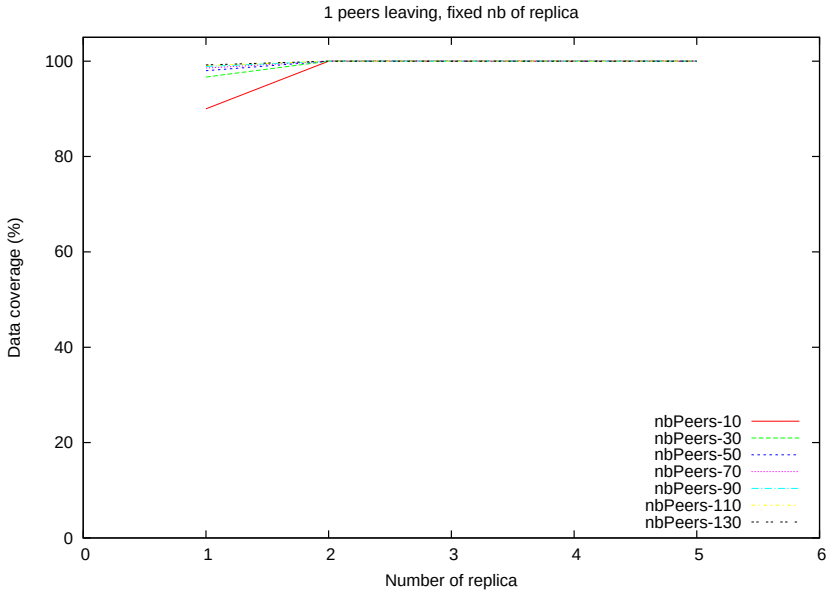


Figure 6.14 – 1 pair disparaît, nombre fixe de répliques

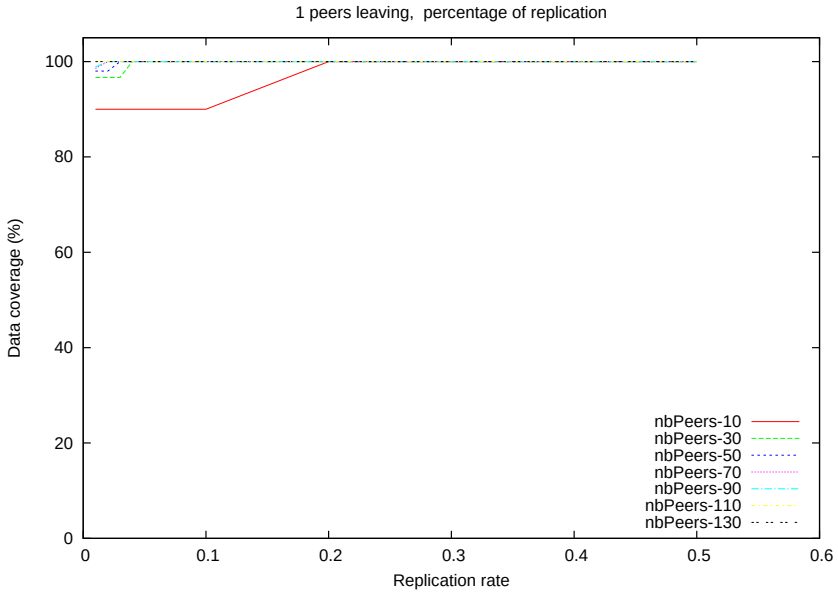


Figure 6.15 – 1 pair disparaît, répliques au prorata du nombre de terminaux

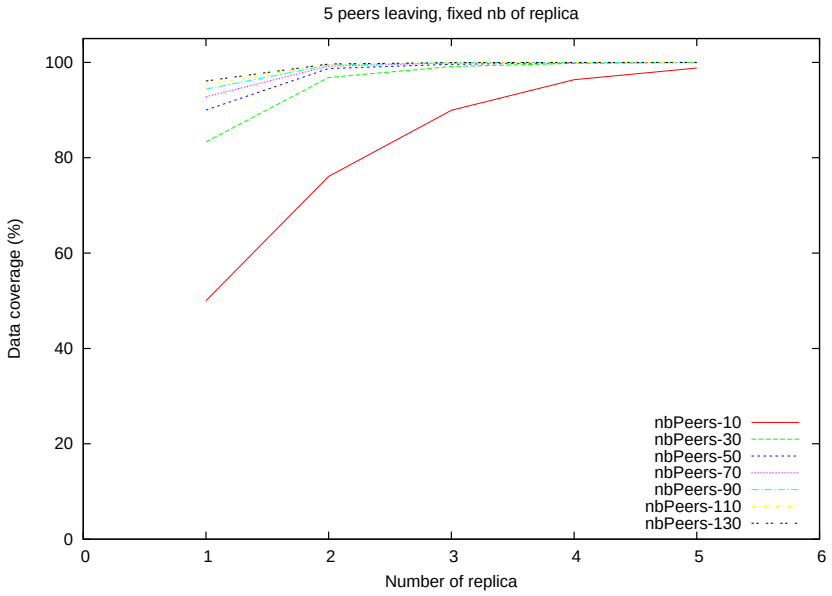


FIGURE 6.16 – 5 pairs disparaissent, nombre fixe de répliques

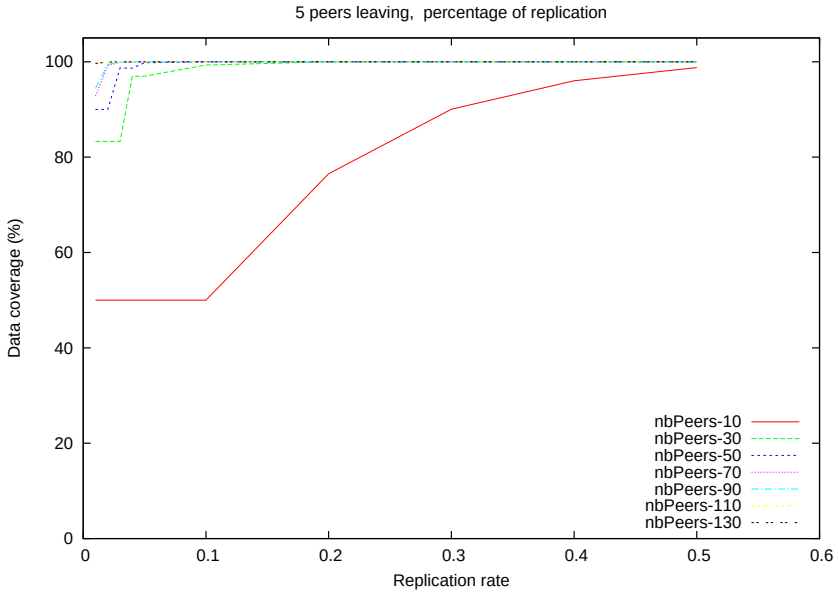


FIGURE 6.17 – 5 pairs disparaissent, répliques au prorata du nombre de terminaux

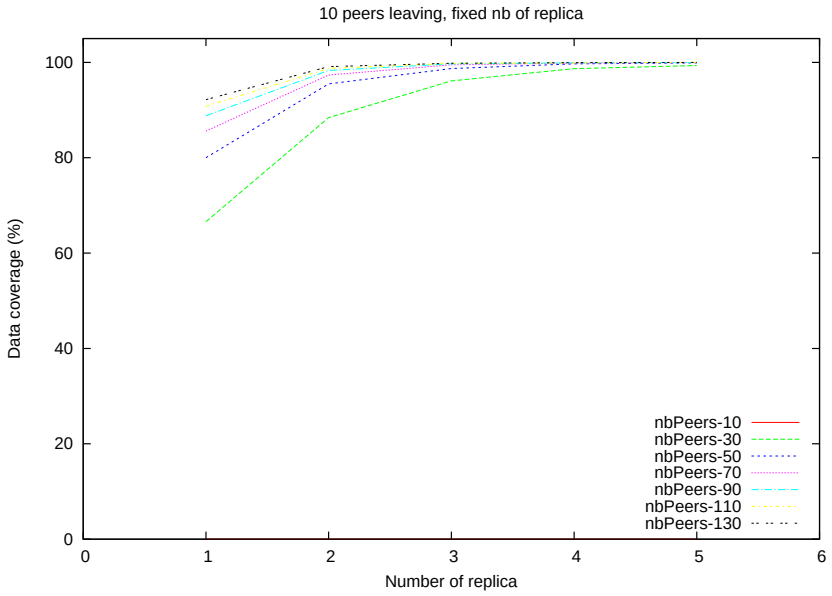


Figure 6.18 – 10 pairs disparaissent, nombre fixe de répliques

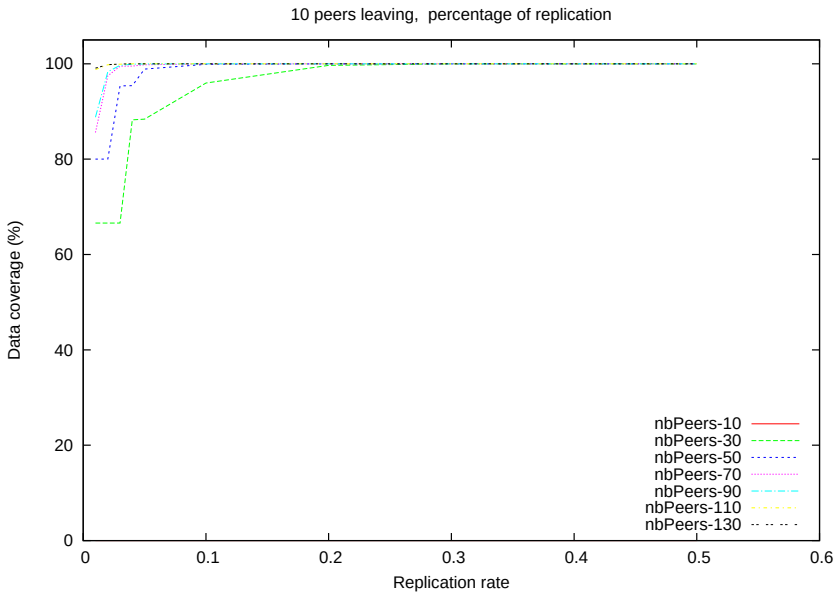


Figure 6.19 – 10 pairs disparaissent, répliques au prorata du nombre de terminaux

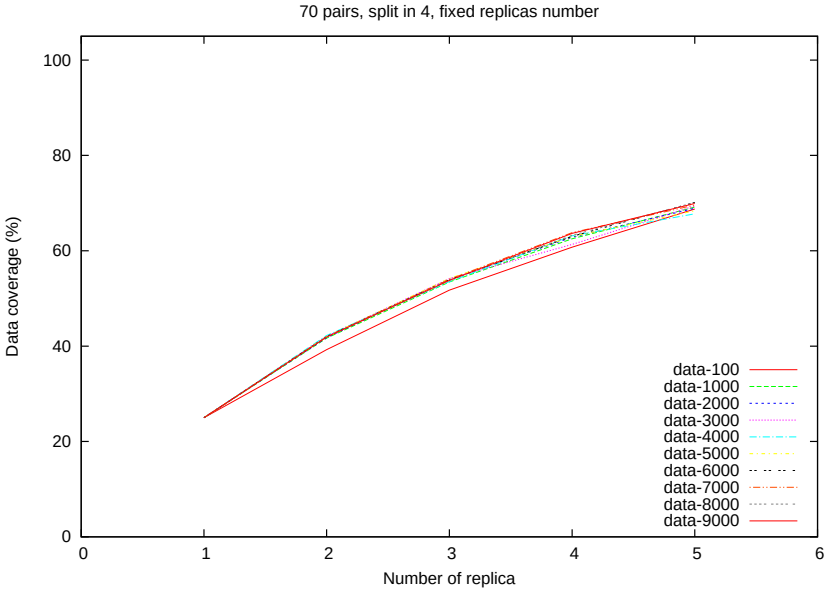


FIGURE 6.20 – Influence du nombre de données sur la couverture suite à une partition, nombre fixe de répliques

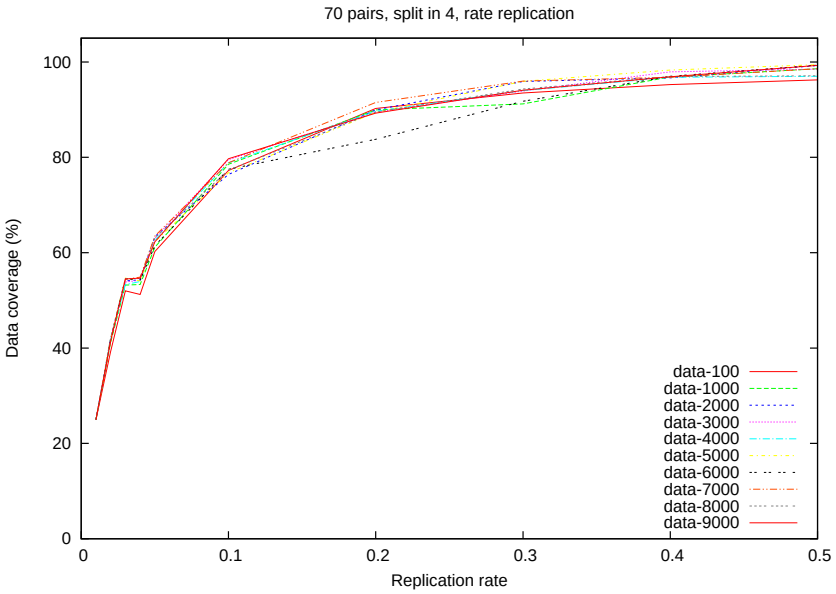


FIGURE 6.21 – Influence du nombre de données sur la couverture suite à une partition, répliques au prorata du nombre de terminaux

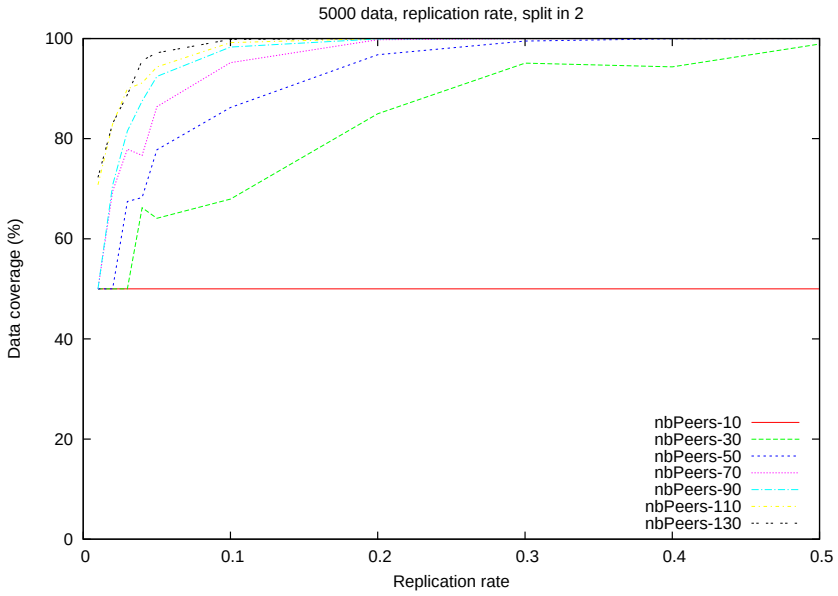


Figure 6.22 – Le groupe se sépare en 2, réplication au prorata du nombre de terminaux

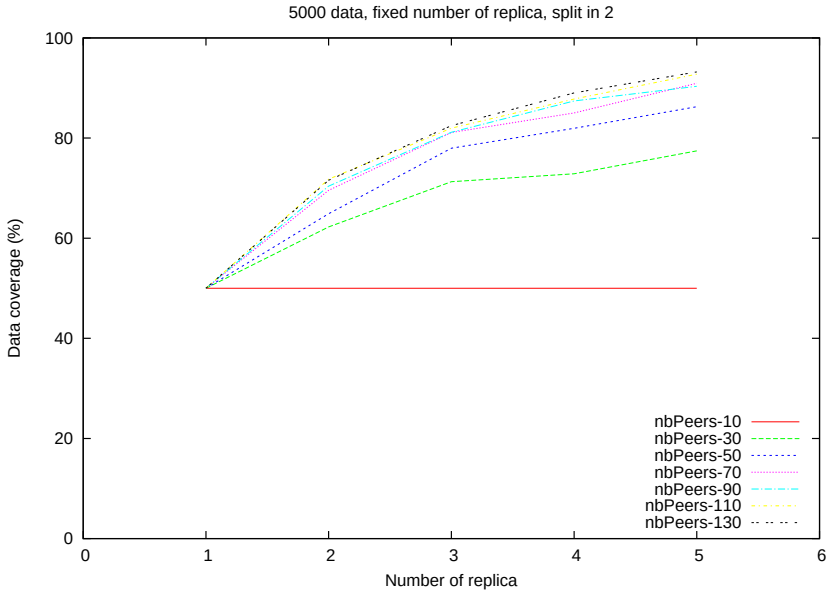


Figure 6.23 – Le groupe se sépare en 2, nombre fixe de répliques

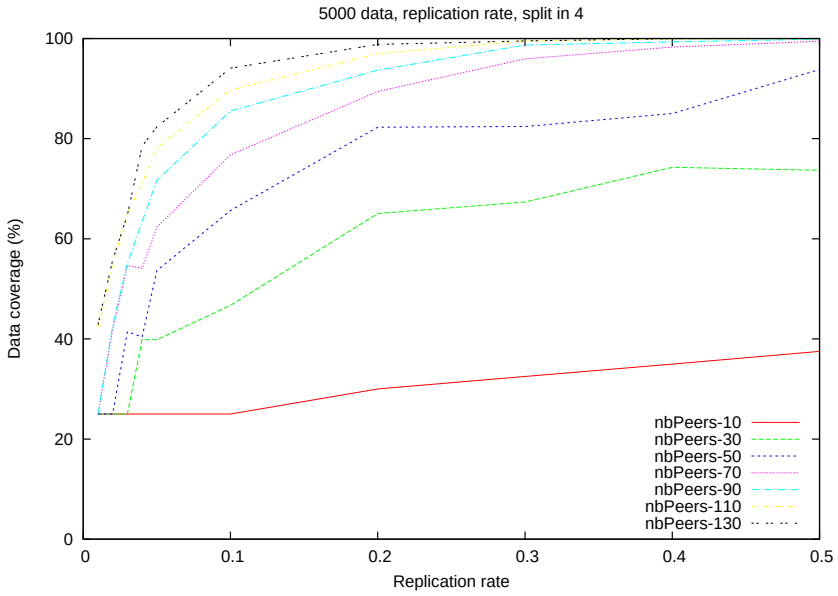


FIGURE 6.24 – 4 nouveaux groupes, réplication au prorata du nombre de terminaux

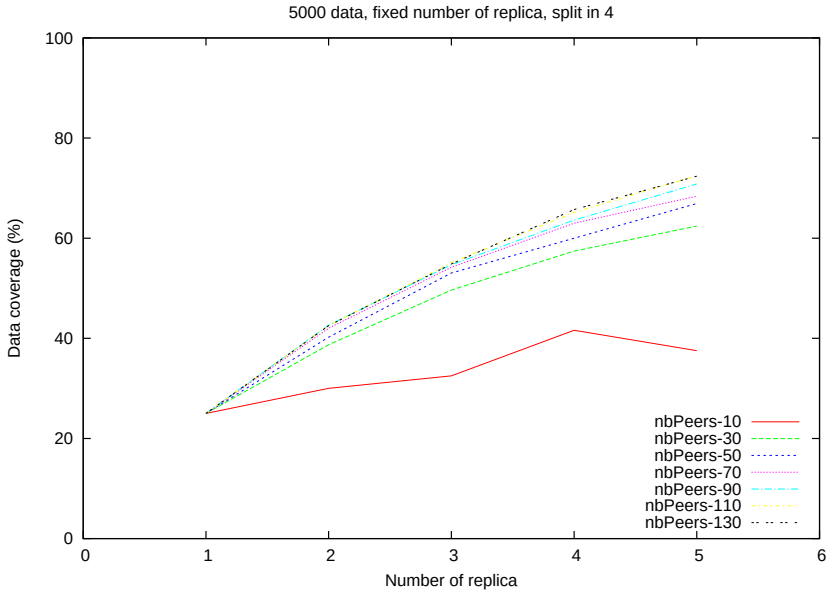


FIGURE 6.25 – 4 nouveaux groupes, nombre fixe de répliques

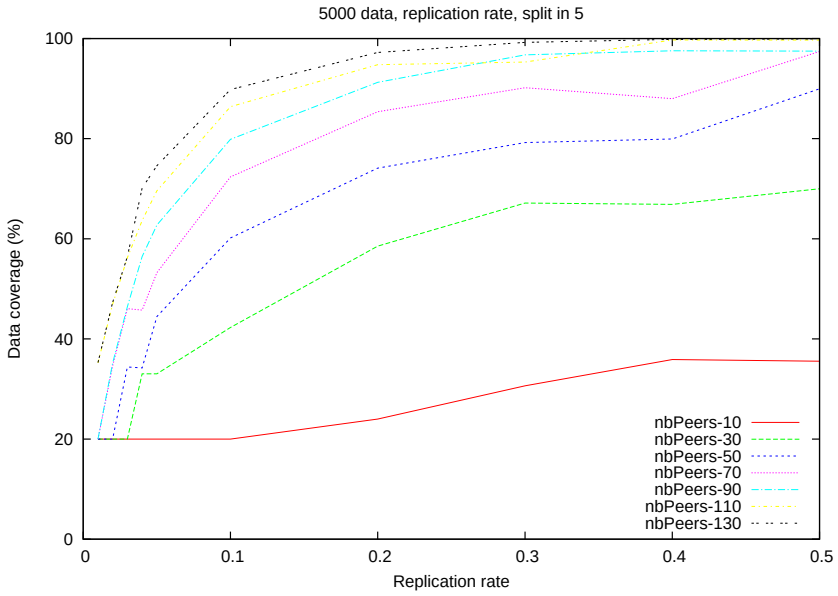


Figure 6.26 – Le groupe se sépare en 5, réplication au prorata du nombre de terminaix

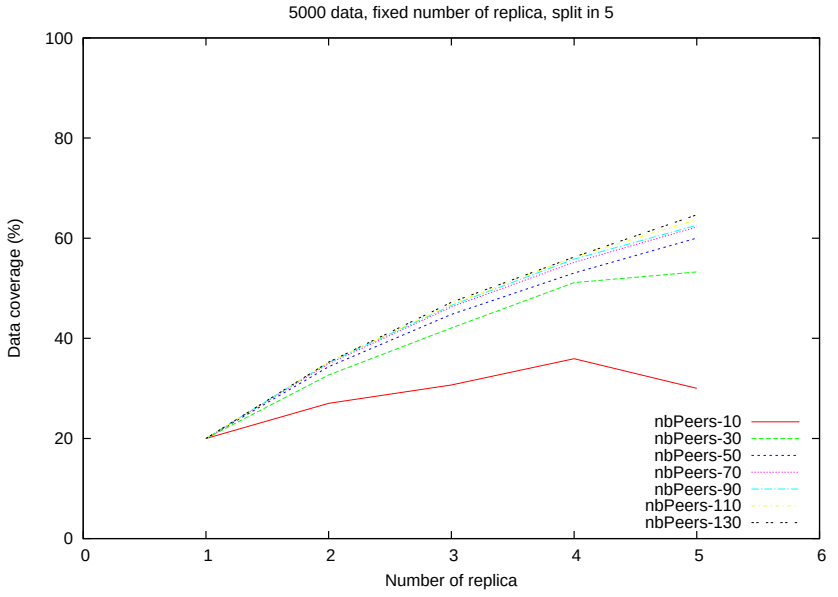


Figure 6.27 – Le groupe se sépare en 5, nombre fixe de répliques

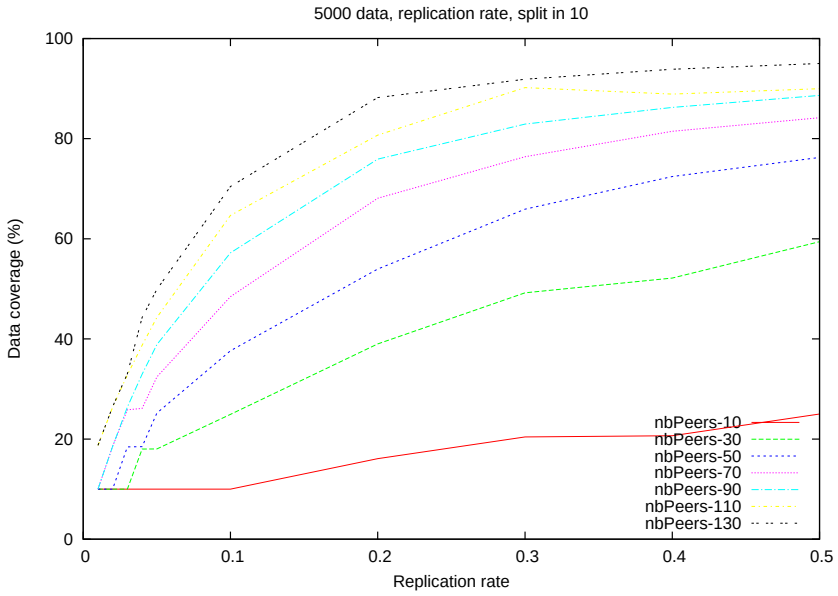


FIGURE 6.28 – Le groupe se sépare en 10, réplication au prorata du nombre de terminaux

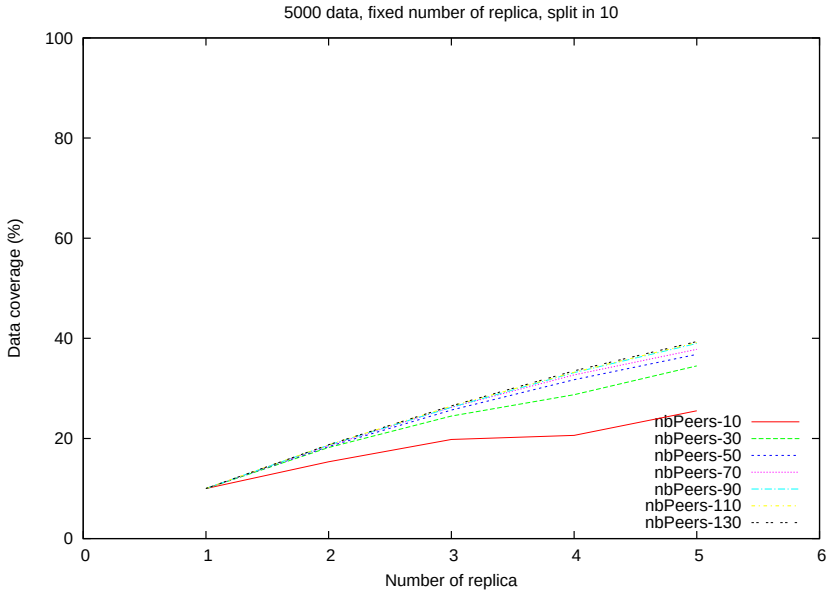


FIGURE 6.29 – Le groupe se sépare en 10,nombre de répliques fixes

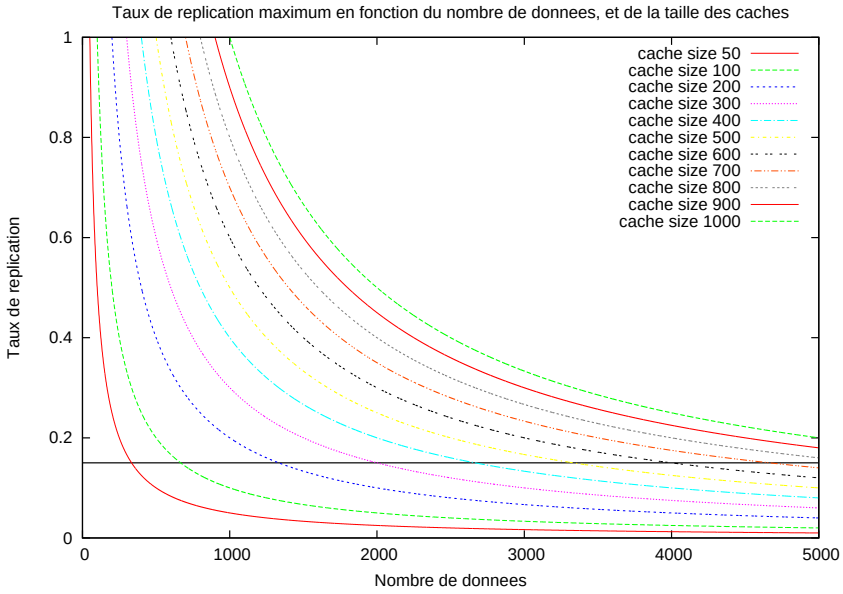


Figure 6.30 – Taux de réplication maximum en fonction du nombre de données, et de la taille des caches

6.6.2 Réplication sémantique

L'idée derrière la réplication sémantique est de diminuer le temps d'accès moyen à une donnée en plaçant des copies par avance sur les terminaux susceptibles de l'utiliser.
Afin de valider cet algorithme, nous avons donc examiné le nombre moyen de sauts nécessaires pour accéder à une donnée en fonction de son intérêt.
Tout d'abord, nous initialisons la simulation :
– Création d'un jeu de mots-clés.
– Création d'un jeu de données, avec des mots-clés associés.
– Création d'un jeu de terminaux, avec les intérêts de leurs utilisateurs associés.
– Construction d'un graphe à une seule partition sur lequel on place les terminaux au hasard.

Les données sont répliquées au prorata du nombre de terminaux suivant 3 méthodes différentes :

1. placement purement statistique *(statistic)*,
2. placement égoïste puis statistique *(selfish-statistic)*,
3. placement égoïste, collaboratif, puis statistique *(semantic)*.

A l'issue du placement, on construit un dictionnaire associant à une valeur d'intérêt, la distance moyenne à laquelle se trouve la donnée.
Cette opération est effectuée en faisant varier plusieurs paramètres :
– Le nombre de terminaux, *nbPeers*.
– La taille de base du cache *baseCache*.

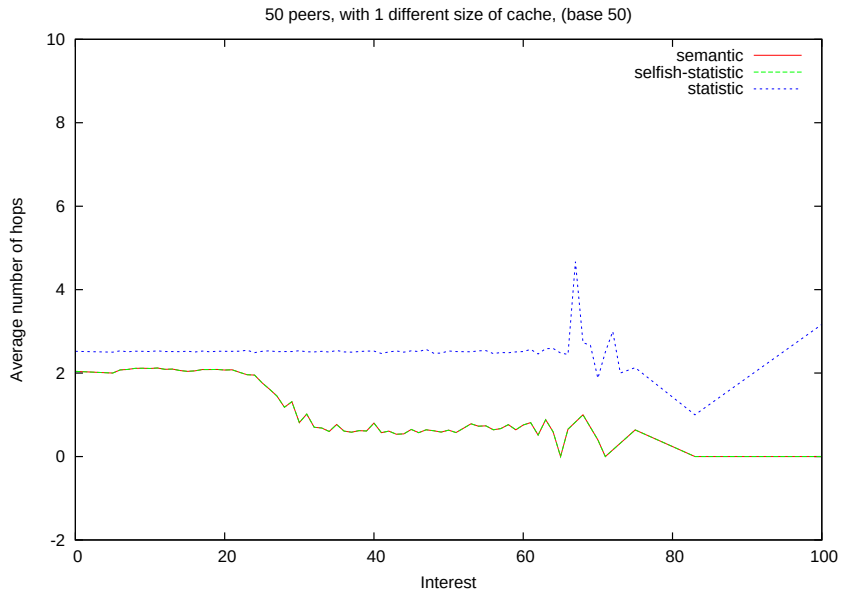


FIGURE 6.31 – Cache de 50

– Le nombre de types de terminaux différents, c'est à dire de capacités différentes. Pour 4 types de terminaux, on aura donc $\frac{nbPeers}{4}$ terminaux de capacité *baseCache*, $\frac{nbPeers}{4}$ terminaux de capacité *2*baseCache*, $\frac{nbPeers}{4}$ terminaux de capacité *3*baseCache* et $\frac{nbPeers}{4}$ terminaux de capacité *4*baseCache*.

Pour chaque triplet de paramètres, on effectue 20 simulations afin de lisser les résultats.

6.6.2.1 Résultats

Les figures 6.31 à 6.34 présentent le résultat des expérimentations, pour 50 terminaux, et une taille de cache de 50, 350, 650 ou 950 emplacements.

Ces figures montrent que l'algorithme de placement statistique donne une valeur constante de distance moyenne à la donnée, quel que soit son intérêt, alors que l'algorithme de placement collaboratif diminue bien la distance aux données plus intéressantes.

Notez que la courbe n'est pas lissée pour les valeurs d'intérêts importantes, car il y a moins de points (moins de données très intéressantes, que de données faiblement intéressantes).

Le véritable gain dépend de la confiance qu'on place dans le critère d'intérêt : si celui-ci détecte dans 80% des cas qu'une donnée va être utilisée localement, le nombre de sauts moyens pour accéder aux données sera diminué d'autant.

On voit, par ailleurs, que la réplication collaborative et la réplication égoïste sont confondues. C'est parce que les terminaux ont tous les même capacités, et qu'il n'y a donc pas de réplication collaborative mise en œuvre.

Pour différentes capacités des terminaux, on ne perçoit une différence que pour des tailles de cache élevées : pour une taille restreinte, un terminal n'a pas assez d'espace pour répliquer pour ses voisins. Même dans ce cas, la différence entre la réplication sémantique et la réplication égoïste est faible, car le rapport entre les tailles de cache est faible.

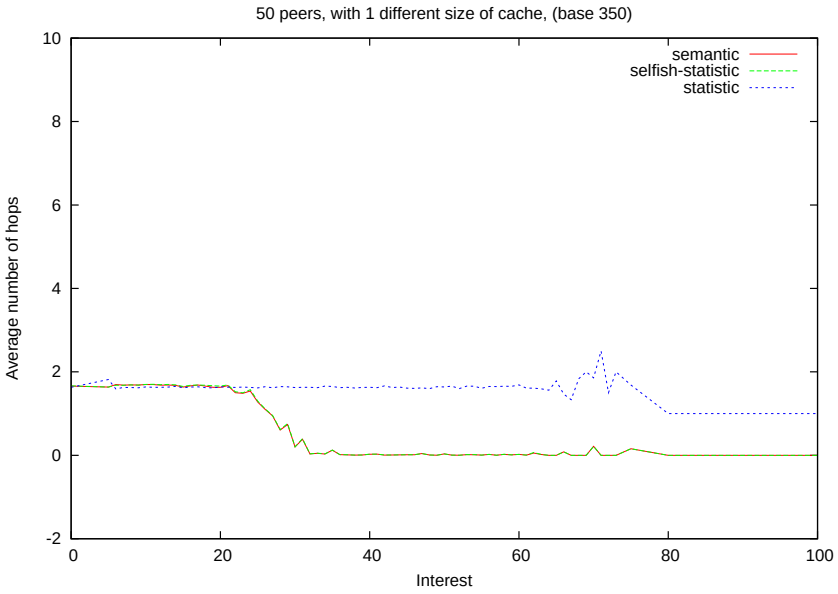


Figure 6.32 – Cache de 350

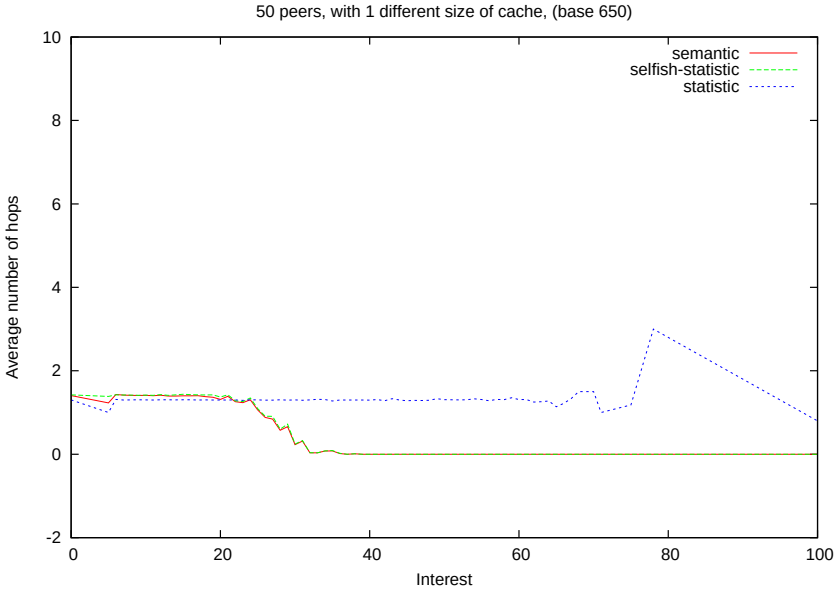


Figure 6.33 – Cache de 650

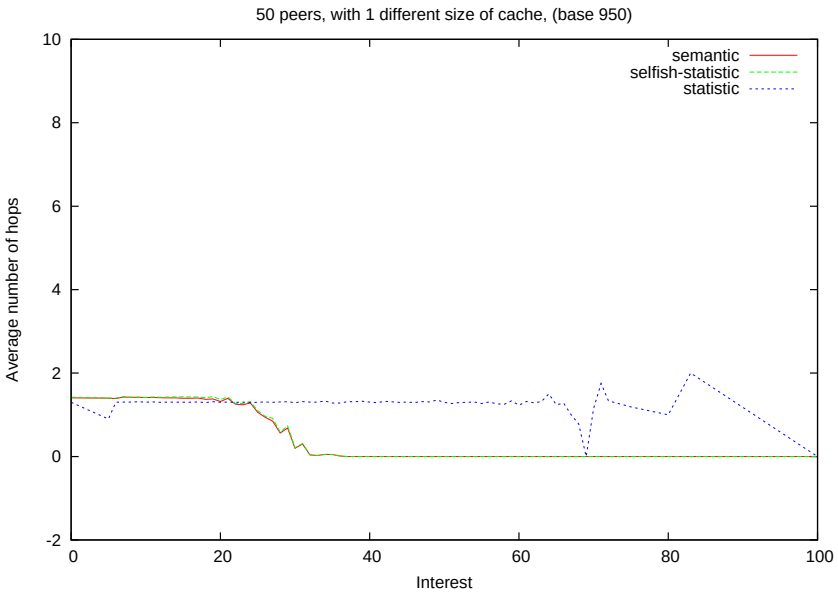


FIGURE 6.34 – Cache de 950

Nous avons donc fait une expérience avec des tailles de cache plus contrastées.
Dans cette simulation, on crée 100 terminaux :
– 10 terminaux ont un cache de taille 1000 : ils représentent des ordinateurs portables,
– 90 terminaux ont un cache de taille 50 : ils représentent des terminaux légers, comme des PDA.

Le reste de la simulation se déroule comme expliqué précédemment.
Le résultat de cette simulation est présenté pas la figure 6.35. On peut voir ici que la réplication collaborative permet bien de diminuer la distance d'accès aux données.
Comme la réplication collaborative ne se fait que pour les terminaux ayant moins de capacités que soit, elle est efficace quand il existe une hétérogénéité des ressources entre les terminaux.

6.6.3 Réplication statistique

Si à l'issue de la réplication collaborative assez de répliques ont été créées, la réplication statistique n'a pas lieu d'être. Sinon, en combien d'itérations la réplication statistique converge-t-elle ?
Considérons M pairs, avec un taux de réplication de r. En tout, K=M*r répliques doivent être créées.
Lors d'une itération de l'algorithme, on suppose L répliques déjà existantes. Il reste donc Q = K-L répliques à créer et placer sur N=M-L pairs. L'expérience probabiliste "un des N pairs n'hébergeant pas la donnée réplique avec une probabilité de $p = \frac{Q}{N} = \frac{K-L}{M-L}$." peut être vue comme une variable de Bernouilli de paramètre p.
Les variables associées à chaque pair sont indépendantes.

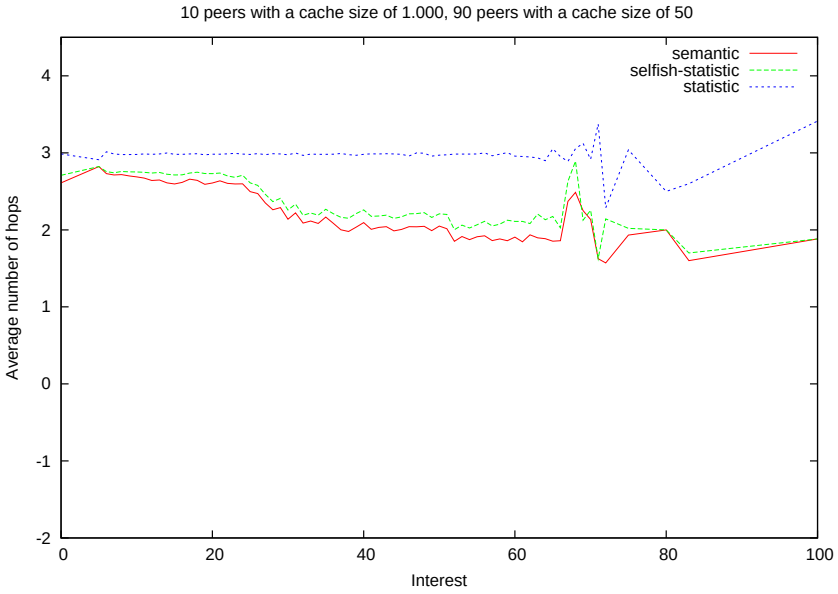


Figure 6.35 – Réplication collaborative : 10 pairs avec une capacité de cache de 1000 répliquent collaborativement pour les 90 autres pairs, qui disposent d'une capacité de 50

La variable aléatoire discrète X décrivant le nombre de répliques créées à une itération, sachant le nombre de terminaux total M, le nombre de répliques existantes L, et le nombre de répliques à atteindre K, définie de *[0,M-L]* dans *[0,1]* suit donc une loi binomiale de paramètres (N,p).
On a donc :

$$proba(X = v) = \binom{M-L}{v} p^v * (1-p)^{N-v}$$

$$proba(X >= Q) = \sum_{i=0}^{Q-1} proba(X = i)$$

Soit Y_t la variable aléatoire décrivant la probabilité que k répliques existent suite à l'itérations t, sachant M le nombre total de terminaux, L le nombre de répliques existante et K le nombre de répliques à atteindre. Elle est définie de *[0,N]* dans *[0,1]*, et s'écrit de manière récurrente :
Si t=0 :

$$proba(Y_0 = y) = \begin{cases} 0 \ si \ y < L, \\ \binom{M-L}{L-k} \left(\frac{K-L}{M-L}\right)^{k-L} * \left(1 - \frac{K-L}{M-L}\right)^{N-k+L} . \end{cases}$$

Sinon, si $t \geq 0$:

$$proba(Y_t = y) = \sum_{z=0}^{y} \Big(proba\big(Y_{t-1} = z\big) * \Big(proba\big((y-z) \ repliques \ crees \ a \ t\big)\Big)\Big)$$

Soit :

$$proba(Y_t = y) = \sum_{z=0}^{y} \left(proba\big(Y_{t-1} = z\big) * \binom{M-z}{y-z} * \Big(\frac{K-z}{M-z}\Big)^{y} * \Big(1 - \frac{K-z}{M-z}\Big)^{M-z-y} \right)$$

L'algorithme s'arrête si le nombre de répliques y est supérieur ou égal à K.
Soit Z la variable aléatoire décrivant la probabilité que l'algorithme s'arrête à l'itération t. A t-1, il existe au plus K-1 répliques. Elle est définie de $[0, \infty]$ dans [0,1] et s'écrit de manière récurrente.
Si $t = 0$:

$$proba(Z = 0) = \begin{cases} 1 \ si \ L \geq K, \\ \sum_{v=K-L}^{M-K} \left(\binom{M-L}{v} * \big(\frac{K-L}{M-L}\big)^{v} * \big(1 - \frac{K-L}{M-L}\big)^{N-v} \right). \end{cases}$$

Si $t \geq 0$:

$$proba(Z = t) = \sum_{0}^{K-1} \left(P\big(Y_{t-1} = y\big) * \left(\sum_{q=1}^{N-y} (creer \ q \ replique \ a \ t) \right) \right)$$

$$proba(Z = t) = \sum_{y=0}^{K-1} \left(P\big(Y_{t-1} = y\big) * \left(\sum_{q=1}^{N-y} \left(\binom{M-y}{q} * \Big(\frac{K-y}{M-y}\Big)^{q} * \Big(1 \frac{K-y}{M-y}\Big)^{K-y-q} \right) \right) \right)$$

Nous avons évalué la valeur moyenne du nombre d'itérations nécessaires pour atteindre le nombre de répliques souhaité par simulation.
La figure 6.36 présente les résultats.
On peut voir qu'en moyenne, l'algorithme se termine donc en moins de 2 itérations.

6.6.4 Surcoût lié à l'algorithme de réplication

Dans cette section, nous évaluons le coût en terme de messages de notre algorithme, et surtout le surcoût induit par la création de répliques préventives, par rapport à la réplication à la demande.

6.6.4.1 Création d'une nouvelle donnée

La création d'une donnée engendre de nombreux messages car on veut créer immédiatement des répliques plutôt qu'attendre qu'elles soient créées à la demande.
Sachant qu'on a M, le nombre de terminaux, et r le taux de réplication demandé, on a donc :

1. 1 message de diffusion *(bcast)* indiquant la création de la donnée,
2. r*M-1 messages de diffusion indiquant la création d'une réplique,
3. r*M-1 messages point à point *(ucast)* de demande de réplication, plus si la réplique est très populaire,

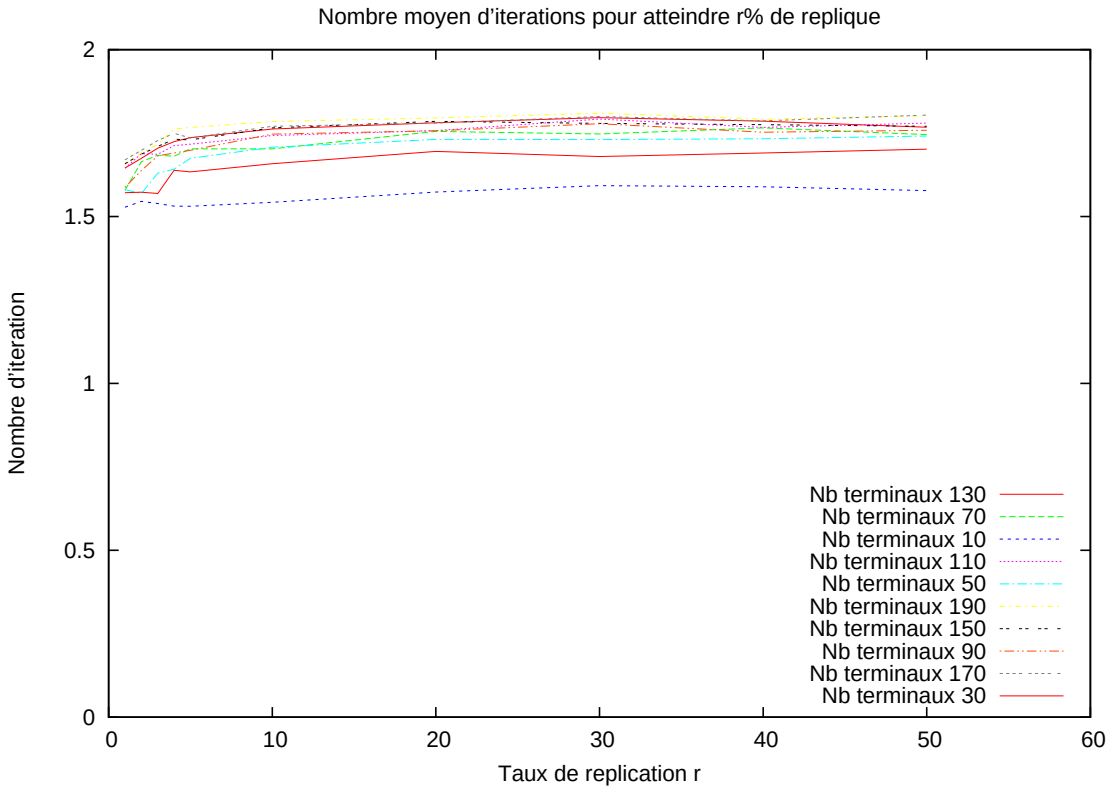


Figure 6.36 – Nombre moyen d'itérations de l'algorithme

4. r*M-1 messages point à point convoyant la donnée.

Le nombre total de messages lors d'une création est donc de :
(r*M) *bcast* signalisations + (r*M-1) *ucast* requêtes + (r*M-1) *ucast* de réponses
Supposons maintenant qu'il existe une seule réplique de départ, et qu'on utilise uniquement la réplication à la demande. Le coût de la création de r*M-1 répliques supplémentaires se décompose alors ainsi :

- la copie originale, et première réplique, diffuse son existence,
- la deuxième réplique nécessite 1 requête, et 1 réponse, point à point,
- la troisième réplique nécessite 1 requête, 1 réponse, et 1 message pour indiquer à l'hôte ne lui ayant pas servi la donnée qu'il faut désormais l'inclure dans la mise à jour des données,
- la troisième réplique nécessite 1 requête, 1 réponse, et 2 messages pour indiquer aux hôtes ne lui ayant pas servi la donnée qu'il faut désormais l'inclure dans la mise à jour des données,
- ...
- la r*M-ème réplique nécessite 1 requête, 1 réponse, et r*M-2 messages pour indiquer aux hôtes ne lui ayant pas servi la donnée qu'il faut désormais l'inclure dans la mise à jour des données.

Pour avoir r*M réplique d'une donnée, le coût est donc :
1 *bcast* + (r*M-1) *ucast* requêtes + (r*M-1) *ucast* réponses + $\sum_{i=0}^{r*M-2} i$ signalisation, soit :
1 *bcast* + (r*M-1) *ucast* requêtes + (r*M-1) *ucast* réponses + $\frac{(r*M-2)(r*M-3}{2}$ signalisation

Afin de pouvoir faire la différence entre le coût de la création de k copies avec notre algorithme, et celui de de la création de k copies avec l'algorithme de réplication à la demande, nous prenons pour équivalent du coût d'un message de diffusion, le coût de M-1 messages point à point.
Ceci est bien entendu une approximation : un message de diffusion a un coût moins élevé que M-1 messages point à point. Cependant, la différence de coût dépend de la topologie du réseau. Nous prenons donc cette approximation afin de pouvoir calculer un résultat qui majore notre coût.
On a donc :

- pro-actif : r*M*(M-1) sign + (r*M-1) req + (r*M-1) rep
- on demand :((M-1) + $\frac{(r*M-2)(r*M-3}{2}$) sign +(r*M-1) req + (r*M-1) rep

La différence est donc :

$$rM(M-1) - ((M-1) + \frac{(rM-2)(rM-3)}{2})$$

$$\frac{2rM^2 - 2rM - 2M + 2 - r2M^2 + 3rM + 2rM - 6}{2}$$

$$\frac{M^2(2r - r^2) + M(3r - 2) - 4}{2}$$

Pour r=0.15, le coût de la réplication pro-active est donc supérieur à celui de la réplication à la demande à partir de 8 terminaux, si on considère que le coût d'une diffusion est équivalent au coût d'un message point à point à chaque membre du réseau.
Cependant, cet envoi permet de remplir les caches de localisation, et donc d'éviter d'interroger les services de localisation. Ce coût est par ailleurs ponctuel (uniquement à la création d'une nouvelle donnée), et permet d'éviter la disparition de données.

La figure 6.37 représente de manière graphique le surcoût lié à la création immédiate de C répliques. Les axes du plan représentent le nombre de terminaux présents, et le taux de réplication effectif, tandis que l'axe vertical représente le nombre de messages en surcoût. On peut voir que la majorité du plan reste en deça de 200.

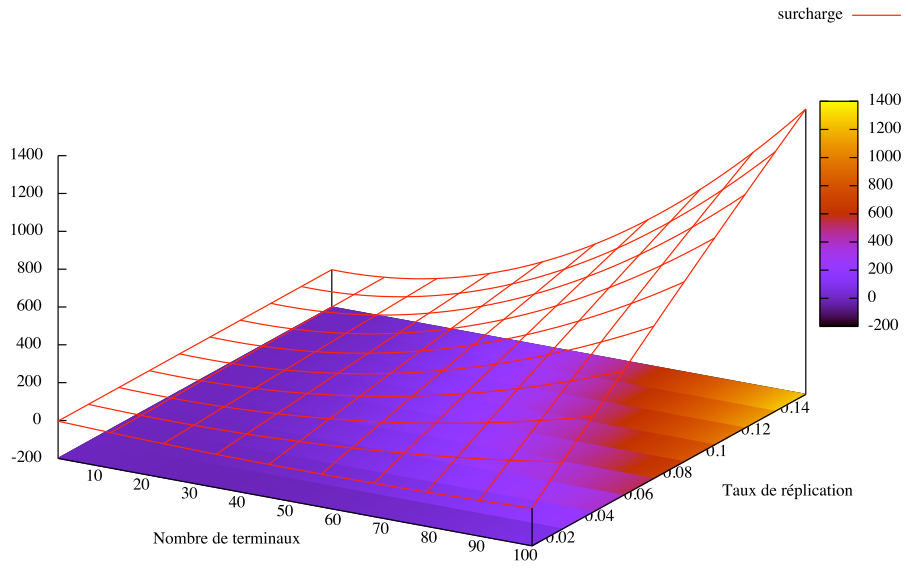


Figure 6.37 – Surcoût lié à la création immédiate de C répliques

6.6.4.2 Surcoût lié au taux de réplication

Nous avons vu que la création de répliques à r% permettait d'éviter la disparition des données. Cependant, ces répliques induisent un coût par rapport à l'algorithme de réplication à la demande, que nous mesurons dans cette section.

Soit N le nombre de terminaux, r le taux de réplication et C le nombre de copies existantes. Deux situations sont possibles :

- C>N*r : on a plus de copies que nécessaire, et on fait donc l'hypothèse qu'elles sont toutes utiles. Cette hypothèse est permise par notre algorithme d'éviction de répliques présenté dans le chapitre suivante.
- C=N*r : on a exactement le nombre de copies nécessaire. Dans ce cas, il est possible que certaines ne soient pas utilisées par leur hôte.

Nous allons voir dans la section suivante comment, si de nombreuses copies ont été créées à un instant mais ne sont plus utilisées par la suite, et génèrent donc du trafic, elles sont éliminées, ce qui permet donc de maintenir le nombre de répliques au minimum nécessaire. Suite à l'application de cet algorithme, si le nombre de copies existantes est supérieur au nombre minimum requis, ces copies sont toutes utilisées. Il n'y a donc pas de surcoût.

Quel est alors le surcoût de la réplication à n% par rapport à la réplication à la demande si on a seulement le nombre de copies maintenu par notre algorithme de réplication, c'est à dire que C=N*r ?

Soit c_1 le nombre de copies utilisées par leur hôte, et c_2 le nombre de copies non utilisées. On a, bien entendu, $c_2 = C - c_1$. On suppose que si on utilisait un algorithme de réplication à la demande, seuls c_1 copies existeraient.

Sur une période d'évaluation, chaque hôte utilisant la donnée génère en moyenne k mises à jour. Le nombre total de messages sur la période est donc :

$$nbTotMsg = c_1 * k * (C - 1) = c_1 * k * (N * r - 1)$$

Les messages à destination des hôtes utilisant la donnée sont utiles, et génèrent donc le même nombre de messages que pour la réplication à la demande :

$$nbMsgUtile = c_1 * k * (c_1 - 1)$$

Le nombre de messages inutiles est donc :

$$nbMsgSurcout = c_1 * k * (c_2) = c_1 * k * (N * r - c_1)$$

On peut alors calculer le pourcentage de trafic généré en plus, par rapport à la réplication à la demande :

$$portionSurcout = \frac{nbMsgSurcout}{nbTotMsg} = \frac{c_1 * k * (N * r - c_1)}{c_1 * k * (N * r - 1)}$$

$$portionSurcout = \frac{N * r - c_1}{N * r - 1}$$

On voit donc que plus le nombre d'hôtes utilisant la donnée est élevé, plus la portion de messages inutiles diminue.
Cependant, cela ne signifie pas nécessairement qu'en nombre de messages, avoir moins d'hôtes utilisant la donnée crée plus de trafic. En effet, cette mesure est une portion du nombre total de messages : si c_1=0, c'est à dire qu'aucun terminal ne crée de mise à jour, le trafic total est nul.
Afin de déterminer le coût maximum en terme de messages, on dérive donc la fonction f permettant de calculer le nombre de messages en surcoût, en considérant r, N et k fixes.

$$f(c) = c * k * (N * r - c) = c * k * N * r - c^2$$

$$f'(c) = k * N * r - 2 * c$$

La dérivée f' s'annule pour $c_0 = \frac{k*N*r}{2}$
On peut donc en conclure que pour un nombre $c_0 = \frac{k*N*r}{2}$ de terminaux utilisant la donnée, on génère le pire surcoût en terme de messages inutiles.

$$f(c_0) = c_0 * k * N * r - c_0^2$$

$$f(c_0) = \frac{k * N * r}{2} * k * N * r - \frac{k * N * r}{2}^2$$

$$f(c_0) = \frac{(k * N * r)^2}{4}$$

La figure 6.38 représente le pire surcoût en terme de message de manière graphique. Les axes du plan représentent le nombre de terminaux présents, et le nombre de répliques utilisées, tandis que l'axe vertical représente le nombre de messages en surcoût. On peut voir que si toutes les répliques, ou aucune, sont utilisées, aucun message supplémentaire n'est créé.

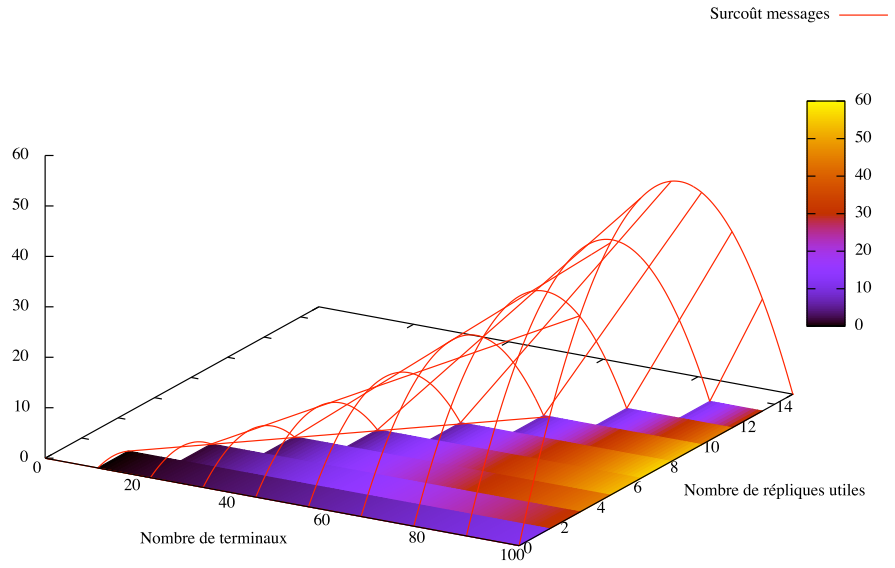


Figure 6.38 – Surcoût lié au maintien de C répliques

6.7 Comparaison à l'existant

De nombreux systèmes ne proposent pas d'algorithme de réplication spécifique. Nous avons alors considéré que la réplication se fait à la demande. Un autre critère algorithme de réplication simple proposé dans l'état de l'art est une restriction de la réplication à la demande, où on crée une copie de la donnée en fonction de la distance de l'hôte la servant. Enfin, une troisième proposition est de créer des répliques des données très demandées, afin de les rapprocher de leurs utilisateurs.
Ces systèmes supposent l'existence d'un serveur, et ne prennent pas en compte la mobilité des terminaux. Dans une approche totalement distribuée, l'utilisation d'un algorithme de réplication à la demande crée le risque de perte des données les moins utilisées.
Le système adHocFS maintient par contre au moins une copie préventive de chaque donnée, en sus d'une réplique de travail, au sein d'un groupe. L'hôte de cette copie est choisi en fonction de ses capacités, notamment pour éviter qu'il disparaisse à cause d'un faible niveau de batterie. Ceci permet de tolérer la disparition d'un terminal mais n'est pas robuste en cas de partition.
Les algorithmes de réplication proposés par Hara placent pro-activement les données sur les terminaux les plus susceptibles de les utiliser, et répliquent collaborativement. Cependant, ces algorithmes sont utilisés dans des réseaux de capteurs, et selon les algorithmes, supposent une connaissance totale des fréquences d'accès, une connaissance totale des fréquences de mise à jour ou une connaissance de la corrélation pour chaque paire de données. Notre proposition ne nécessite pas une connaissance de la totalité des données, ni des informations citées ci-dessus.

6.8 Conclusion

Dans ce chapitre nous avons présenté un algorithme de réplication de données pro-actif et se basant sur une prédiction des accès utilisateurs pour placer les données.

Nous avons vu que créer pro-activement des données permet d'en limiter les disparitions, qui, comme nous travaillons dans un système sans serveur, risquent d'être définitives. Par ailleurs, la réplication au prorata du nombre de terminaux permet de mieux supporter les partitions du réseau.
Nous avons ensuite vu que le placement prédictif permet de faire diminuer la distance moyenne d'accès à une donnée. Ce résultat nécessite cependant d'avoir une bonne métrique pour prédire les accès. La réplication collaborative permet elle-aussi de diminuer cette distance pour les terminaux ayant un espace de stockage plus réduit que celui de leurs voisins.
Enfin, nous avons déterminé que la partie probabiliste de notre algorithme se termine en moyenne en moins de deux itérations.
Nous avons ensuite comparé cet algorithme à la réplication à la demande et borné le surcoût qu'il engendre en terme de charge réseau. Nous avons tout d'abord calculé le surcoût engendré par la création des K répliques attendues par notre algorithme, plutôt que d'attendre leur création à la demande, puis le surcoût engendré par les répliques non utilisées.

Un terminal ne peut pas héberger des répliques à l'infini : il doit parfois éliminer une copie locale afin de faire place à une nouvelle donnée. Dans le chapitre suivant nous présentons l'algorithme de remplacement de cache utilisé dans ces circonstances.

Chapitre 7

Remplacement de cache et éviction de répliques

Dans ce chapitre, nous présentons un algorithme de remplacement de cache, qui, d'une part, choisi les répliques à éliminer de manière à limiter la perte de donnée, et qui, d'autre part, diminue la charge réseau en éliminant les répliques non utilisées et qui génèrent du trafic réseau.

Dans le chapitre précédent nous avons vu un modèle de réplication de donnée. Parfois, quand on tente de répliquer une donnée, et que le cache est plein, il est nécessaire d'élire une réplique à éliminer du cache afin d'y placer la nouvelle.
Dans les MANets, le réseau est une ressource limitée : non seulement, comme dans les réseaux filaires, on veut éviter les problèmes de congestion liés à un trafic trop important, mais on veut aussi éviter d'utiliser inutilement la batterie.
L'élimination de répliques est donc à faire dans deux circonstances :
– une nouvelle donnée doit être répliquée, et le cache est plein.
– une donnée génère trop de trafic localement, alors que l'utilisateur n'en a pas l'usage.
Le premier problème a trait à la ressource de stockage. Le remplacement de cache est nécessaire quand un terminal tente de répliquer localement une donnée, et que son cache est plein. Il doit donc libérer de l'espace-mémoire en éliminant certaines répliques. Pour cela on définit un critère de tri sur les répliques permettant de sélectionner celles à éliminer.
Le deuxième problème a trait à la ressource-réseau. Il peut arriver qu'une réplique génère du trafic dû aux mises à jour, mais n'est pas utilisée localement. L'éliminer permet donc de diminuer la charge réseau.
Dans le premier cas, il s'agit donc pour un ensemble de données répliquées localement, de décider quelle donnée doit être éliminée. Dans le second cas, il s'agit pour un ensemble de répliques de la même donnée, d'éliminer les répliques générant du trafic superflu.
Dans ce chapitre, nous présentons un protocole de gestion de cache en deux parties.
Nous voyons tout d'abord le mécanisme utilisé pour libérer la mémoire, puis celui utilisé pour diminuer la charge réseau. Nous présentons ensuite notre méthode de validation avant d'exposer nos résultats et de conclure.

7.1 Libérer de l'espace mémoire

L'algorithme présenté ici permet de sélectionner quelle donnée ôter du cache quand celui-ci est plein et qu'une nouvelle donnée doit être répliquée localement.
Nous allons tout d'abord voir le principe, puis l'algorithme lui-même, avant de l'illustrer par un exemple.

7.1.1 Principe

Comme dans LRU, on conserve une liste d'identifiants des données triés par ordre d'accès. Cette liste est mise à jour à chaque accès local, en plaçant l'identifiant de la donnée utilisée en fin de liste.
Quand un remplacement de cache est demandé, on choisit la donnée à éliminer selon le critère suivant :

1. On parcourt la liste et on cherche la première donnée dont le nombre de répliques est supérieur au nombre de répliques de garde, nécessaire à la disponibilité des données. Si aucune donnée ne satisfait cette condition, on passe à l'étape suivante.
2. On parcourt la liste à partir de la fin, et on cherche la donnée avec le plus grand nombre de répliques. Si deux données ont le même nombre de répliques, on sélectionne celle qui a été utilisée le moins récemment.

Il est possible que toutes les copies disparaissent suite au remplacement de cache si :
– de manière concurrente, tous les hôtes décident de la supprimer.
– tous les hôtes disparaissent avant que de nouvelles copies n'aient été créées

```
def cooperativeLRU(self):
    rmCandidate= self.sorted_LCL_AccessList[0]
    i=1
    while datas[rmCandidate].howMany()<=rate*len(peers) and i<len(self.sortedAccessList):
        rmCandidate=self.sortedAccessList[i]
        i+=1
    if i==len(self.sortedAccessList) :
        posMax =len(self.sortedAccessList)-1
        j=len(self.sortedAccessList)-1
        while j>0 :
            postMaxPeer=self.sortedAccessList[posMax]
            currentPeer= self.sortedAccessList[j]
            if datas[postMaxPeer].howMany()<=datas[currentPeer].howMany() :
                posMax=j
            j-=1
        rmCandidate=self.sortedAccessList[posMax]

    self.rmReplica(rmCandidate)
    self.sortedAccessList.remove(rmCandidate)
```

FIGURE 7.1 – Algorithme de remplacement de cache coopératif

7.1.2 Algorithme

La figure 7.1 présente l'algorithme Cooperative-LRU.
On peut voir les variables suivantes :
- *sorted_LCL_AccessList* : la liste des données présente dans le cache, triée par date d'accès.
- *peers* : la liste des pairs dans le groupe stable.
- *rate* : le taux de réplication demandé.
- *datas* : une structure de données qui stocke les copies locales, et la liste des autres hôtes.

7.1.3 Exemple

La figure 7.2 présente un exemple d'application de notre algorithme, dans un cas où l'application de LRU conduit à la perte d'une donnée.
Il se déroule en 2 étapes.
Etape 1 : A accède à la donnée 10, B accède à la donnée 3

1. La donnée 10 n'est pas présente sur A, le cache de A n'est pas plein, 10 est donc répliqué sur A.
2. La donnée 3 n'est pas présente sur B, le cache de B est plein, B exécute donc l'algorithme de remplacement de cache.
3. 2 algorithmes :

   LRU : la donnée sortie est 1.

   Cooperative-LRU : dans le réseau il existe :
   - 2 copies de 1, 2 étant le nombre de copies de réserve, on ne l'élimine pas,
   - 3 copies de 5, la donnée sortie est donc 5.

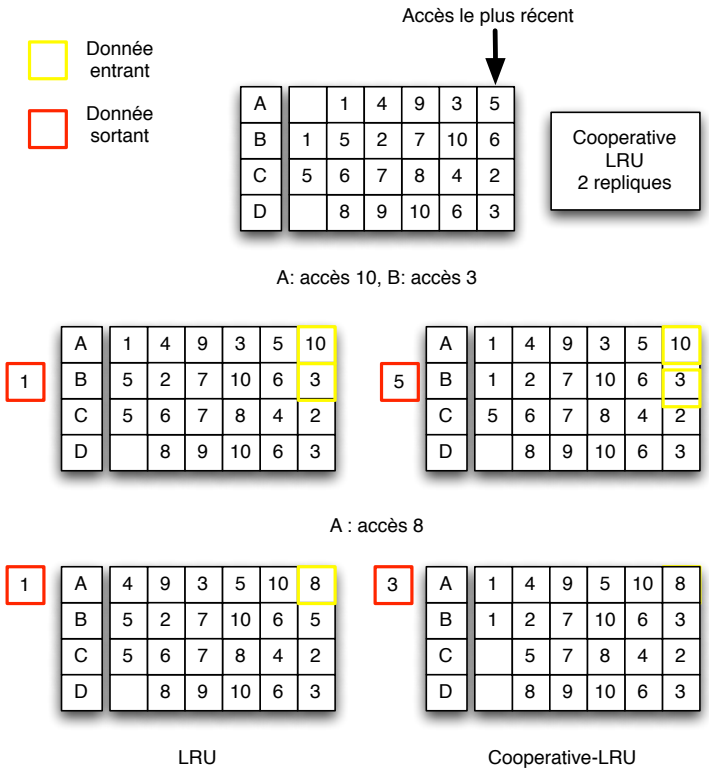


Figure 7.2 – Remplacement de cache, Cooperative LRU vs LRU

Les modifications des états des caches sont propagées avant l'étape 2.
Etape 2 : A accède à la donnée 8

1. la donnée 8 n'est pas présente sur A, et le cache de A est plein, A exécute donc l'algorithme de remplacement de cache.
2. 2 algorithmes :

   LRU : la donnée sortie est 1.

   Cooperative-LRU : dans le réseau il existe :
   - 2 copies de 1,
   - 2 copies de 4,
   - 2 copies de 9,
   - 3 copies de 3, la donnée sortie est donc 3.

On voit donc qu'à l'issue de ces deux étapes, la donnée 1 est perdue si on utilise LRU. En revanche, Cooperative-LRU préserve la donnée 1.

7.2 Diminuer la charge réseau

Chaque nouvelle réplique d'une donnée génère une charge liée à la maintenance de la cohérence. Toutes ces répliques ne sont pas nécessairement utiles cependant : elles peuvent avoir été créées suite à un accès passé, mais ne plus intéresser l'utilisateur. Le trafic généré par ces répliques est donc superflu. Afin de diminuer la charge du réseau, nous voulons donc éliminer ces répliques.

On veut donc pour ce faire déterminer l'utilité de la copie locale d'une donnée. Nous allons d'abord voir le principe appliqué puis l'algorithme, avant de présenter un exemple.

7.2.1 Principe

Une réplique peut avoir été créée à deux fins : soit elle est utilisée localement, soit elle est nécessaire pour maintenir la disponibilité des données en cas d'événement de mobilité.
Si une réplique n'est dans aucun de ces deux cas, on peut l'éliminer. Cependant, il n'est pas nécessaire d'éliminer toutes les répliques de ce type : on préfère garder des répliques de données tant qu'elles n'ont pas d'impact sur l'utilisation du réseau, afin d'être plus susceptible de tolérer une partition.
La décision d'éliminer une copie superflue se fait donc localement, si la fréquence d'accès depuis l'extérieur dépasse la fréquence d'accès local.
Pour ce faire, à chaque donnée on associe un compteur d'accès local cpt_L, et un compteur d'accès extérieur cpt_E, afin de calculer :
– f_E la fréquence de mise à jour depuis l'extérieur
– f_L la fréquence d'accès à la donnée localement
Les compteurs sont incrémentés à chaque accès à la donnée, depuis le terminal ou depuis le réseau.
L'algorithme d'éviction est ensuite appliqué périodiquement, avec une période T.
Tout d'abord, les fréquences d'accès sont mises à jour :
– $f_E = 0.5 * f_E + 0.5 cpt_E$
– $f_L = 0.5 * f_L + 0.5 cpt_L$
Pour chaque donnée, on effectue ensuite les tests suivants :

1. le rapport $\frac{f_L}{f_E}$ est-il inférieur à *EvictThreshold*, que nous avons fixé à 0,1 ?
2. le nombre total de répliques dans le réseau est-il supérieur au nombre de répliques désiré pour maintenir la disponibilité des données ?

Alors on peut éliminer la donnée du cache, et on diffuse un message aux autres hôtes pour le signaler.

7.2.2 Algorithme

La figure 7.3 présente les deux algorithmes utilisés pour déterminer quelles répliques éliminer. La méthode du cache *preemptClean* parcourt les données répliquées localement et pour chaque donnée appelle la méthode *shouldClean*. Celle-ci calcule les fréquences d'accès puis retourne un booléen indiquant si le rapport des fréquences d'accès est tel qu'éliminer la donnée réduirait le trafic. Si ce résultat est positif, le cache vérifie combien de copies de la donnée existent dans le réseau avant de l'éliminer.

7.2.3 Exemple

La figure 7.4 est un exemple d'application de l'éviction préventive de copies, où on constate une baisse de la charge réseau.
Le contexte est celui d'un MANet à 10 terminaux, où il faut donc maintenir 2 copies de chaque donnée. Nous nous intéressons uniquement aux terminaux A, B et C représentés sur le schéma.
Au moment de la phase de mise à jour des fréquences d'accès, A constate que :
– il a fait 0 accès local à 1 et reçu 2 mises à jour,

```
# méthode du conteneur de donnees

    def shouldClean(self) :
        self.freq_E = 0.5*self.freq_E + 0.5*(self.cpt_E/float(PERIOD))
        self.freq_L = 0.5*self.freq_L + 0.5*(self.cpt_L/float(PERIOD))
        return self.fred_L/self.freq_E<0.1

# méthode du module de cache

    def preemptClean(self, dataID) :
        if datas[dataID].howMany()<rate*len(peers) :
            if self.replicaDict[dataID].shouldClean() :
                self.rmReplica(dataID)
                self.sorted_LCL_AccessList.remove(dataID)
```

Figure 7.3 – Recherche des données générant trop de trafic

– il a fait 0 accès local à 2 et reçu 2 mises à jour.
Dans les deux cas, le rapport $\frac{f_L}{f_E}$ est donc nul : les données 1 et 2 génèrent un trafic local qui n'est pas utile à A et pourraient être éliminées. Comme il n'y a que deux copies de la donnée 2 dans le réseau, A ne peut pas éliminer cette réplique. Par contre, C ayant répliqué la donnée 1, il y a maintenant 3 répliques dans le réseau. A élimine donc la réplique de la donnée 1 de son cache et en informe les autres hôtes de 1.
Par la suite, A reçoit toujours les mises à jour concernant la donnée 2, mais celles concernant la donnée 1 lui sont épargnées. La charge réseau globale est donc réduite.

7.3 Surcoût mémoire et réseau

Dans cette section, nous étudions le surcoût engendrés par les algorithmes de remplacements de cache et d'éviction en utilisation de mémoire et du réseau.
Comme nous l'avons introduit dans les problématiques, l'utilisation du réseau a un impact sur le temps de travail de l'utilisateur. Nous voulons donc examiner le nombres de messages engendrer par nos algorithmes. Par ailleurs, l'espace-mémoire disponible étant limité, on veut éviter des structures de gestion trop lourdes. Nous examinons donc les données nécessaires à la mise en œuvre de ces algorithmes.

7.3.1 Charge réseau

Quand une réplique est éliminée, un message est envoyé aux autres hôtes de la même donnée pour signaler de ne plus envoyer de mise à jour.
Par ailleurs, les deux algorithmes concernés prennent des décisions en se basant sur des informations locales. Ils ne créent donc pas de surcoût réseau direct supplémentaire.
Il est possible qu'on crée un surcoût, en éliminant la réplique d'une donnée qui est ensuite immédiatement utilisée, mais ceci n'est pas induit dans le surcoût direct, et existe quelle que soit la technique de remplacement de cache. Nous verrons dans la section suivante, validation par simulation ce qu'il en est, en comparant à LRU.

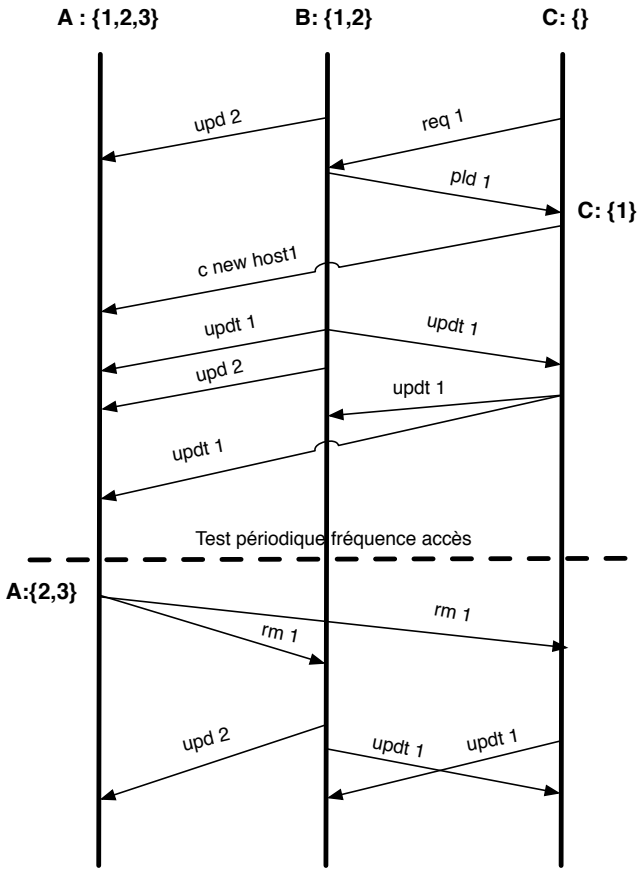


FIGURE 7.4 – Eviction de données créant une surchage réseau

7.3.2 Coût mémoire

L'algorithme Cooperative-LRU nécessite :

- une liste des données répliquées localement, triées par ordre d'accès,
- le nombre d'hôtes pour une donnée stockée localement, qui est obtenu à partir du module de Cohérence et ne fait donc pas partie des données de gestion du remplacement de cache.

Le coût mémoire de Cooperative-LRU est donc identique à celui de LRU.

L'éviction préventive de réplique nécessite par ailleurs :

- les structures de données de Cooperative-LRU,
- deux entiers et deux flottants pour chaque donnée :
  - un compteur d'accès local AccLcl, et un compteur d'accès distant AccDist,
  - une fréquence d'accès local FrqLcl, et une fréquence d'accès distant FrqDist.
- un thread périodique qui met à jour les fréquences à partir des compteurs puis décide de l'éviction.

Sachant la taille maximale du cache de N places, le coût total est donc d'une liste d'entiers de tailles N, plus N*2(sizeof(int)+sizeof(float)), et nécessite par ailleurs un thread de gestion, qui peut être fusionné avec d'autres tâches périodiques si nécessaires.

7.4 Validation par simulation

Comme nous l'avons expliqué dans les chapitres précédents, la validation par simulation permet la répétition des expériences à volonté. Nous n'avons pas validé notre algorithme sur NS pour des raisons de performance des simulations : en effet, la simulation de l'intégralité des couches de protocole, et de nos applications, allonge considérablement le temps d'une simulation.
Nous avons donc écrit un simulateur ad hoc pour notre validation, qui simplifie le système existant en abstrayant les couches réseaux.
Tout d'abord, nous allons voir comment les expériences sont paramètrées, et quelles sont les hypothèses simplificatrices qui ont été faites, avant de présenter comment se déroule une expérience. Les résultats sont présentés dans la section suivante.

7.4.1 Paramétrage d'une simulation

Afin de valider notre protocole, nous avons comparé 4 types de remplacement de cache :
- *LRU* : on ne prend en compte que les accès locaux,
- *Global-LRU* : on prend en compte les accès locaux, et les accès extérieurs,
- *Cooperative-LRU* : on applique l'algorithme décrit dans la section 7.1,
- *Cooperative-LRU_Preventive* : on applique les algorithmes décrits dans les sections 7.1 et 7.2.

Et nous avons examiné 4 variables :
- Nombre de données disparues (i.e. dont il n'existe plus aucune copie) à la fin de la simulation.
- Taux de succès et d'échecs.
- Nombre total de messages, rapporté au nombre d'accès.

Afin de pouvoir obtenir ces résultats, nous traçons donc les valeurs suivantes durant la simulation :
- le nombre total de messages,
- le nombre total d'accès,
- le nombre total de succès.

Différentes valeurs on été testées pour les paramètres suivants :
- le nombre de terminaux varie dans $[10 + k * 20 \ avec \ k \in [0; 7]]$,
- le nombre de données varie dans $[500 * k \ avec \ k \in [1; 4]]$,
- la taille du cache, en unité de stockage varie dans $[50 + 100 * k \ avec \ k \in [0; 10]]$.

Pour chaque combinaison, une expérience est répétée 20 fois afin d'obtenir des résultats génériques.
Nous avons par ailleurs fait des hypothèses simplificatrices, pour des raisons de performances, ou de clarté de résultats :
- Dans ces tests, toutes les données sont créées au début de l'expérience, à 15%. Nous avons fait ce choix afin d'éviter que la métrique du nombre de données disparues soit altérée par le nombre de données créées.
- Pour les mêmes raisons, il n'y a pas d'événements de mobilité (entrée/sortie d'un pair, partition). Nous avons déjà vu, dans le chapitre précédent, l'influence des événements de mobilité, et ici nous voulons étudier le comportement de notre algorithme au sein d'un groupe stable.
- Comme il n'y a pas d'événements réseaux, il n'y a pas de perte de messages.

– Nous avons aussi utilisé des tailles de cache uniforme, afin de pouvoir obtenir des résultats facilement analysables.
– Le modèle de propagation est *"eager update"* [108].

7.4.2 Protocole de validation

Une expérience est paramétrée, comme nous l'avons exprimé ci-dessous, par le nombre de terminaux *nbPeers*, un nombre de données *nbData* et une taille de cache *nbCache*. Elle se déroule de la manière suivante.
Tout d'abord une phase d'initialisation, durant laquelle on crée :
– Un ensemble de 500 mots-clés.
– Un ensemble de *nbData* données.
– Un ensemble de *nbPeers* terminaux, ayant chacun un cache de taille *nbCache*.
A chaque terminal et à chaque donnée est attribué un ensemble de mots-clés. Le nombre de mots-clés associés à une donnée est choisi au hasard, selon une gaussienne de paramètres $\mu = 5$ et $sigma = 2$, et on fait de même pour les terminaux, avec une gaussienne de paramètres $\mu = 15$ et $sigma = 5$.
Selon la taille du cache et le nombre de données, on calcule le taux de réplication des données.
On place ensuite les données en suivant l'algorithme proposé dans le chapitre précédent : réplication égoïste, la réplication collaborative n'a techniquement pas lieu puisque tous les terminaux ont des capacités équivalentes, puis finalement réplication statistique.
Une fois l'initialisation terminée, on effectue une simulation durant 10000 tours. Chaque tour se déroule de la manière suivante :

1. Chaque pair traite, tout d'abord, ses messages reçus. Les messages peuvent être de différentes natures :
   – mises à jour,
   – demande de donnée,
   – signalisation de la destruction d'une réplique,
   – donnée, en réponse à une demande,
   – message indiquant que la demande ne peut être satisfaite (le pair à qui on l'a demandée ne possèdant plus de réplique).
2. Chaque pair tente ensuite un accès :
   – s'il était en attente d'une donnée, 3 cas :
     – il n'a pas reçu de réponse, il reste en attente,
     – il a reçu une réponse, contenant la donnée, dans ce cas il effectue l'accès (cf plus bas),
     – il a reçu une réponse négative, dans ce cas il trouve un nouvel hôte de la donnée via le système de nommage (ici on interroge simplement un dictionnaire), et renvoie une requête.
   – sinon, il effectue un nouvel accès, avec une probabilité de 1 sur 10 que ce soit une écriture, en choisissant la donnée utilisée comme décrit ci-après. Ce choix est arbitraire : dans des traces d'accès, on peut voir qu'il existe plus d'accès en lecture qu'en écriture, mais le rapport dépend des données, des utilisateurs et de la nature de l'utilisation.
     – si la donnée n'est pas en cache, il trouve l'hôte le plus proche et lui envoie une requête,
     – sinon, si l'accès est en lecture, aucun échange de message n'est nécessaire,

  - dans le cas d'un accès en écriture, un message de mise à jour est envoyé à chacun des autres hôtes.
- quand un message est envoyé, il est reçu au tour suivant.

Afin de simuler des accès utilisateurs tenant compte des préférences liées aux mots clés, nous mettons en place la structure suivante lors de la phase d'initialisation : pour chacune des données, dans l'ordre de leurs identifiants, on calcule l'intérêt de l'utilisateur ; on construit ensuite un tableau *interestsArray* où pour la case i, on stocke la valeur de 1000 fois l'intérêt arrondi, ou 1 si l'intérêt était de 0. On somme ensuite les intérêts dans *totInt*. Un tirage uniforme est effectué entre 0 et *sumInterests*, et on cherche dans le tableau la donnée correspondante. Ainsi, les données ayant un intérêt perçu plus grand ont une probabilité plus grande d'être utilisées.
Par exemple, si pour les données A,B,C,D on a des intérêts respecitfs 0,45 ; 0 ; 0,1 ; 0,2, on construit la liste [450, 1, 100, 200]. Si lors du choix de la donnée, on tire 512, c'est la donnée C qui sera accédée puisqu'on a 450+1<512<450+1+100.

7.5 Résultats

Nous avons effectué les tests ci-dessus pour un nombre de terminaux dans $\{10 + k * 20 \text{ avec } k \in [0; 7]\}$. Par souci de concision, nous ne présentons donc ici que les résultats pour 10, 70 et 150 terminaux.
Nous allons tout d'abord voir les résultats en termes de disparition des données, puis en taux de succès, et enfin en nombres de messages.

7.5.1 Disparition des données

Les figures 7.5, 7.6 et 7.7 représentent, pour un nombre fixe de pairs et une simulation donnée, le nombre de données perdues en fonction de la taille des caches. Chacun des 4 graphes dans chaque figure correspond à un nombre de données fixe.
On peut constater plusieurs choses.
Le nombre de répliques est limité par trois facteurs :
- si l'espace total disponible est assez élevé pour avoir un taux de réplication de 15%, il est fonction du nombre de terminaux.
- sinon, connaissant c la taille du cache et N le nombre de données, il est limité par le rapport $\frac{c}{N}$.

Pour que la donnée ne soit pas perdue, il suffit qu'il existe au moins une réplique. La perte de donnée diminue donc quand le nombre de terminaux augmente, puisqu'il y a plus de répliques. On peut le constater en regardant dans chaque cadran le graphe correspondant au même nombre de données.
De même, quand le nombre de données augmente, le nombre de répliques autorisé diminue, et le taux de perte augmente. On peut le constater sur un cadran donné, en comparant les 4 graphes, chaque graphe représentant un nombre différent de données.
Enfin, quand la taille des caches augmente, le nombre de répliques autorisé augmente, et le taux de perte diminue donc. On peut le constater sur chaque graphe individuel : au-dessus de 600, on n'a d'ailleurs aucune perte.
Au sujet de algorithmes proposés, nous pouvons constater que même s'ils n'empêchent pas totalement la disparition des données dans les cas où le rapport entre la taille du cache et le nombre de données est faible, ils donnent de meilleurs résultats que LRU et Global-LRU, particulièrement quand il y a peu de terminaux, donc peu de répliques.

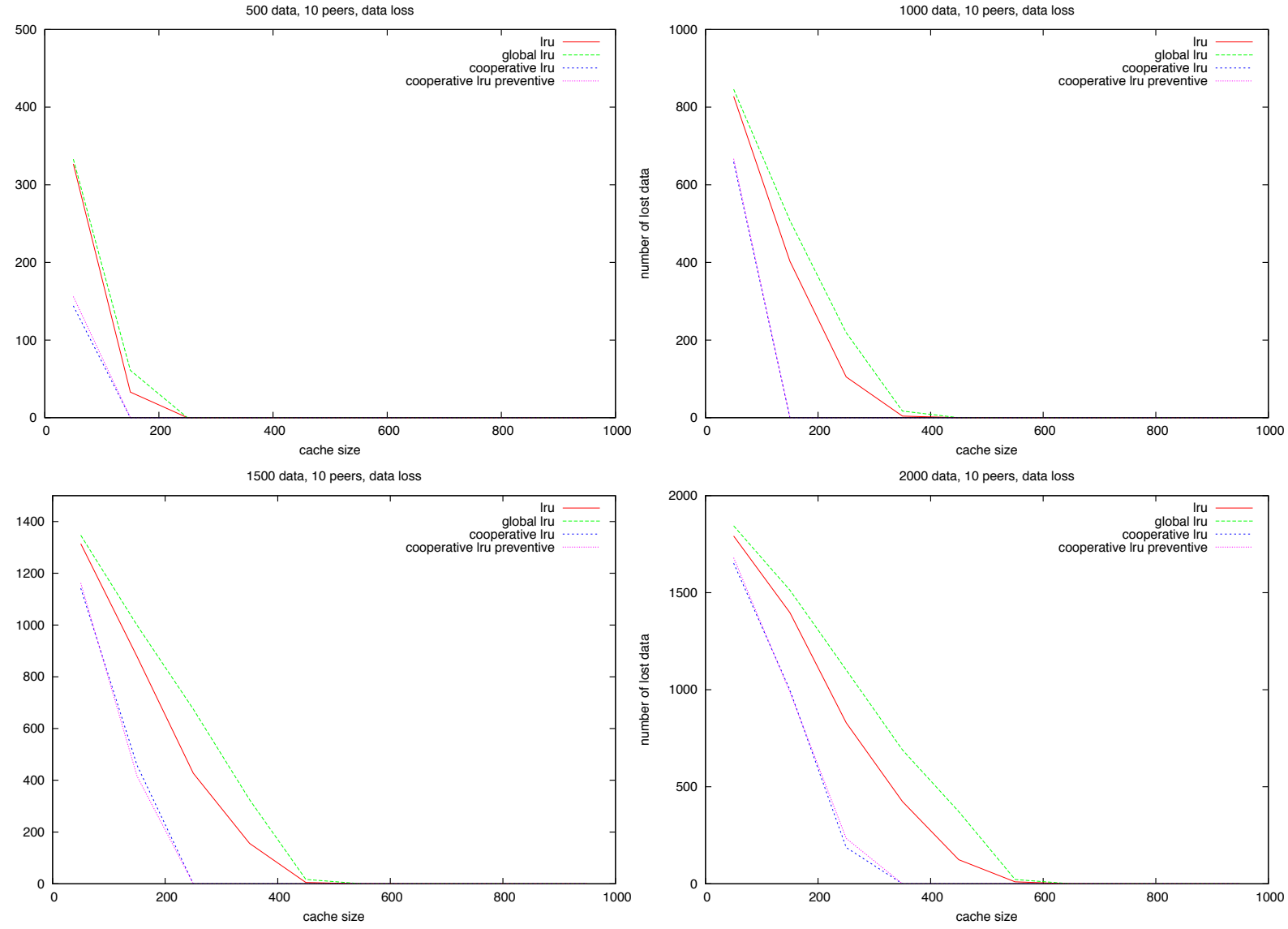


FIGURE 7.5 – Nombre de données perdues, 10 utilisateurs

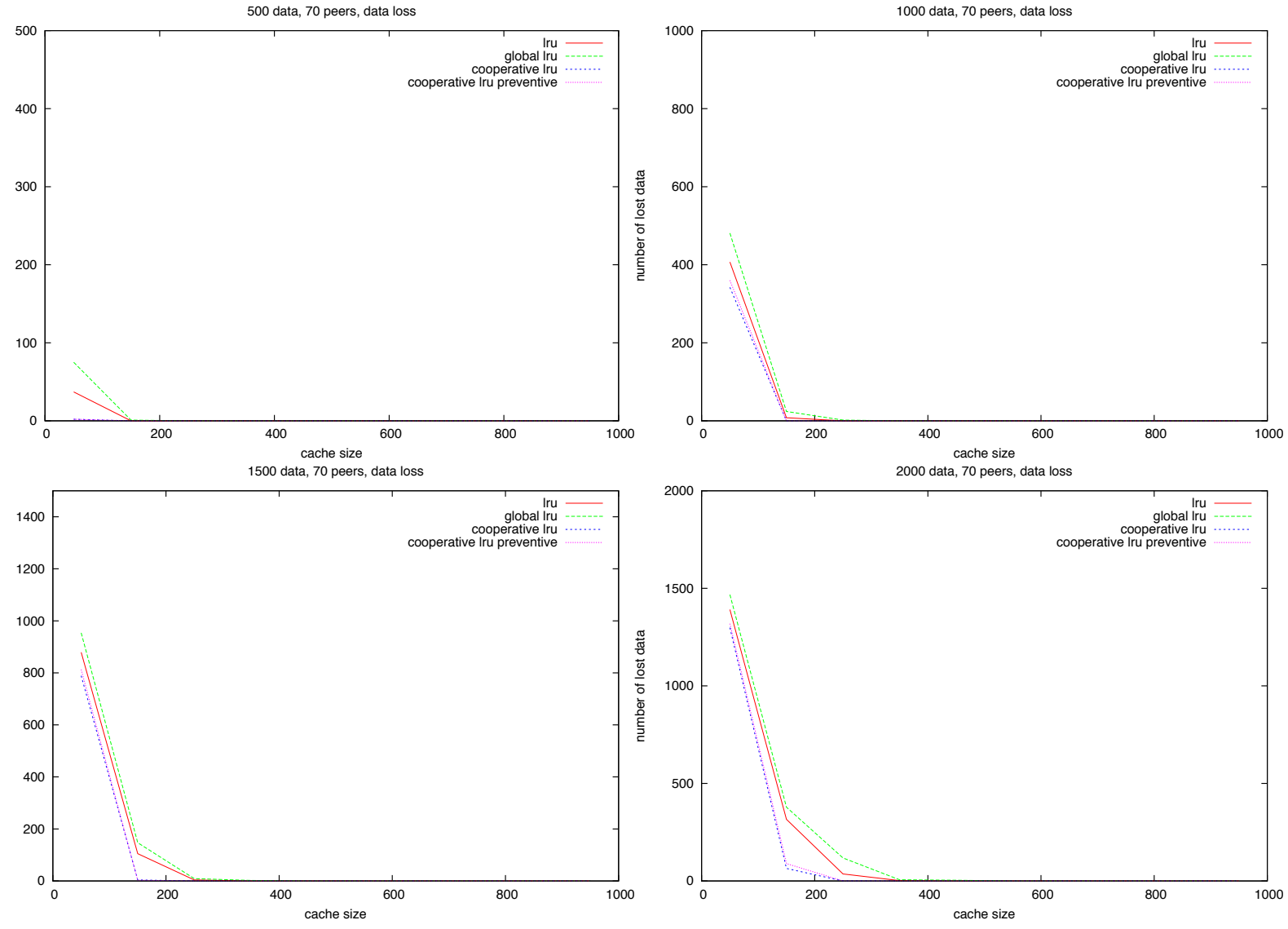


Figure 7.6 – Nombre de données perdues, 70 utilisateurs

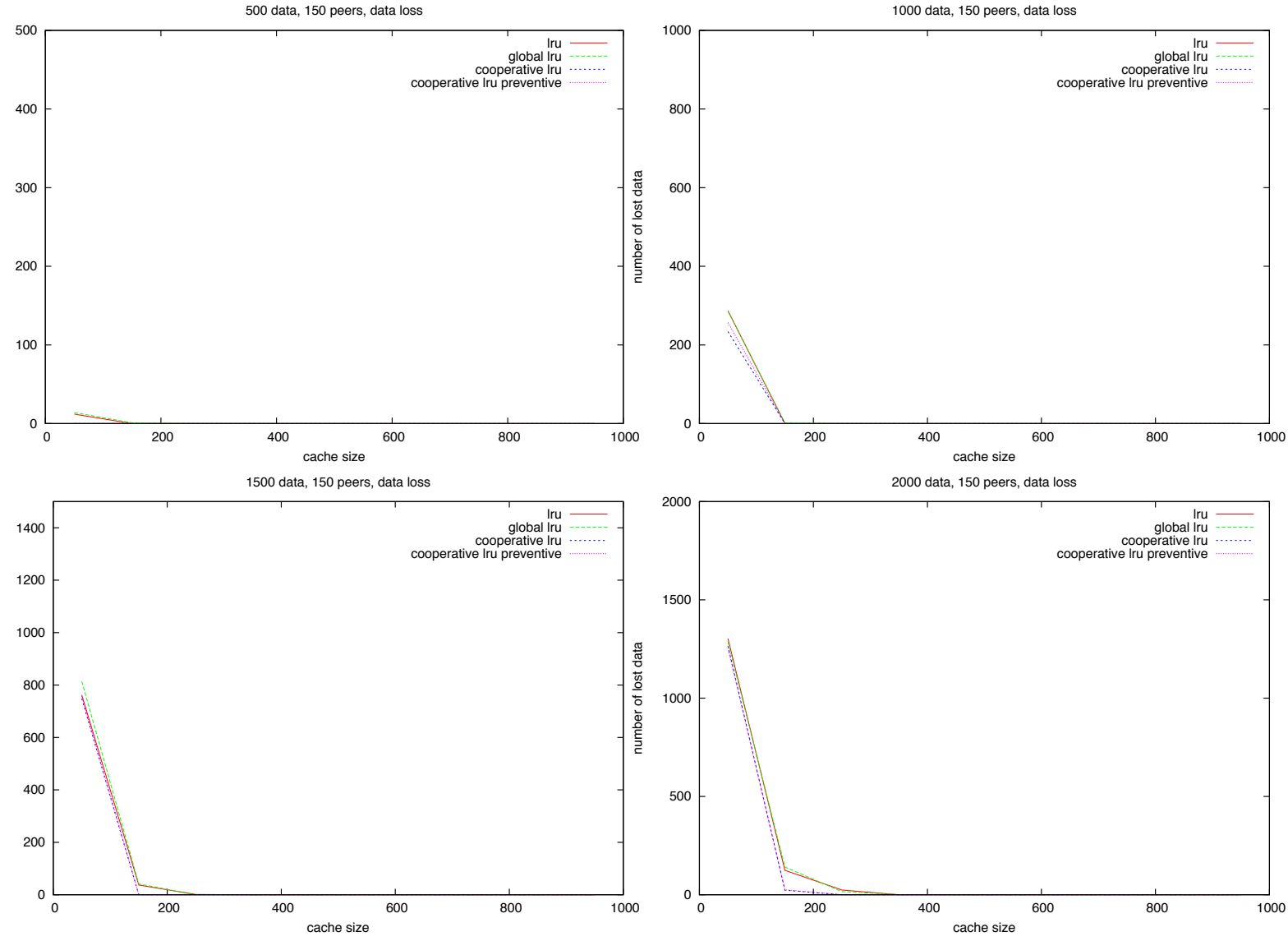


FIGURE 7.7 – Nombre de données perdues, 150 utilisateurs

7.5.2 Taux de succès, taux d'échec

Les figures 7.8, 7.9 et 7.10 représentent, pour un nombre de terminaux fixe, le taux de succès en fonction de la taille des caches. Dans une figure, chaque graphe représente les simulations pour un nombre fixe de données.

Lors de chaque simulation, pour chaque terminal, nous avons comptabilisé son nombre total d'accès, et son nombre total de succès. Nous avons ensuite calculé un taux de succès puis fait une moyenne pour l'ensemble des pairs.

On peut constater tout d'abord sur chaque graphe que le taux de succès s'écroule quand la taille des caches diminue. Ceci est dû aux données disparaissant, comme on a pu le voir dans l'expérience précédente. On peut voir, par exemple, sur la figure 7.10 que le taux de succès pour 2000 données se stabilise pour un cache de taille 250. On constate sur la figure 7.10 que c'est la taille du cache nécessaire pour n'avoir aucune perte de données.

Pour une taille de cache telle qu'il n'y a aucune perte de données, le taux de succès se stabilise.

Le taux de succès augmente quand le nombre de données diminue puisque la probabilité que parmi toutes les données existantes, on ait une copie de celle demandée en cache est plus importante s'il y a moins de données parmi lesquelles choisir. On peut le constater sur chaque cadran : pour 500 données le taux de succès est plus important que pour 2000.

On peut voir par ailleurs que l'algorithme *Cooperative-LRU* a sensiblement le même taux de succès que *LRU*. *Cooperative-LRU-Preventive* par contre, a un taux de succès légèrement plus bas. Ceci est dû au fait qu'on élimine préventivement des données.

Nous avons vu dans l'expérience précédente que *Cooperative-LRU-Preventive*, même s'il élimine des répliques, ne fait pas disparaître plus de données. Le problème est alors de savoir si ces défauts de cache génèrent un trafic supérieur à celui qu'ils évitent. C'est ce que nous vérifions dans l'expérience suivante.

7.5.3 Nombre total de messages

Les figure 7.11, 7.12 et 7.13 représentent pour un nombre fixe de terminaux le nombre moyen de messages par donnée. Pour une figure, chaque graphe représente les résultats pour un nombre différent de données.

Pour chaque terminal, nous avons comptabilisé le nombre d'accès à une donnée qu'il a effectués, ainsi que le nombre de messages qu'il a émis. Nous avons ensuite rapporté le nombre de messages au nombre d'accès, et fait une moyenne pour l'ensemble des pairs.

On voit tout d'abord que pour des tailles de cache petites, le nombre de messages nécessaires par accès est faible. Cela peut paraître contre-intuif, mais est dû au fait que quand une donnée disparaît, aucun message de gestion n'est plus échangé. Quand un pair tente un accès, il est comptabilisé, mais comme le service de nommage, dont nous ne comptabilisons pas les messages puisque nous le considérons comme une boîte noire, lui indique que la donnée n'existe pas, aucun message n'est échangé.

On voit aussi que le nombre de messages augmente en fonction du nombre de terminaux. Ceci est lié au fait que le nombre de répliques est proportionnel au nombre de terminaux, et que le nombre de messages augmente en fonction du nombre de répliques à qui envoyer des messages de cohérence.

On constate enfin que la suppression préventive de copies permet un gain net en terme de trafic réseau.

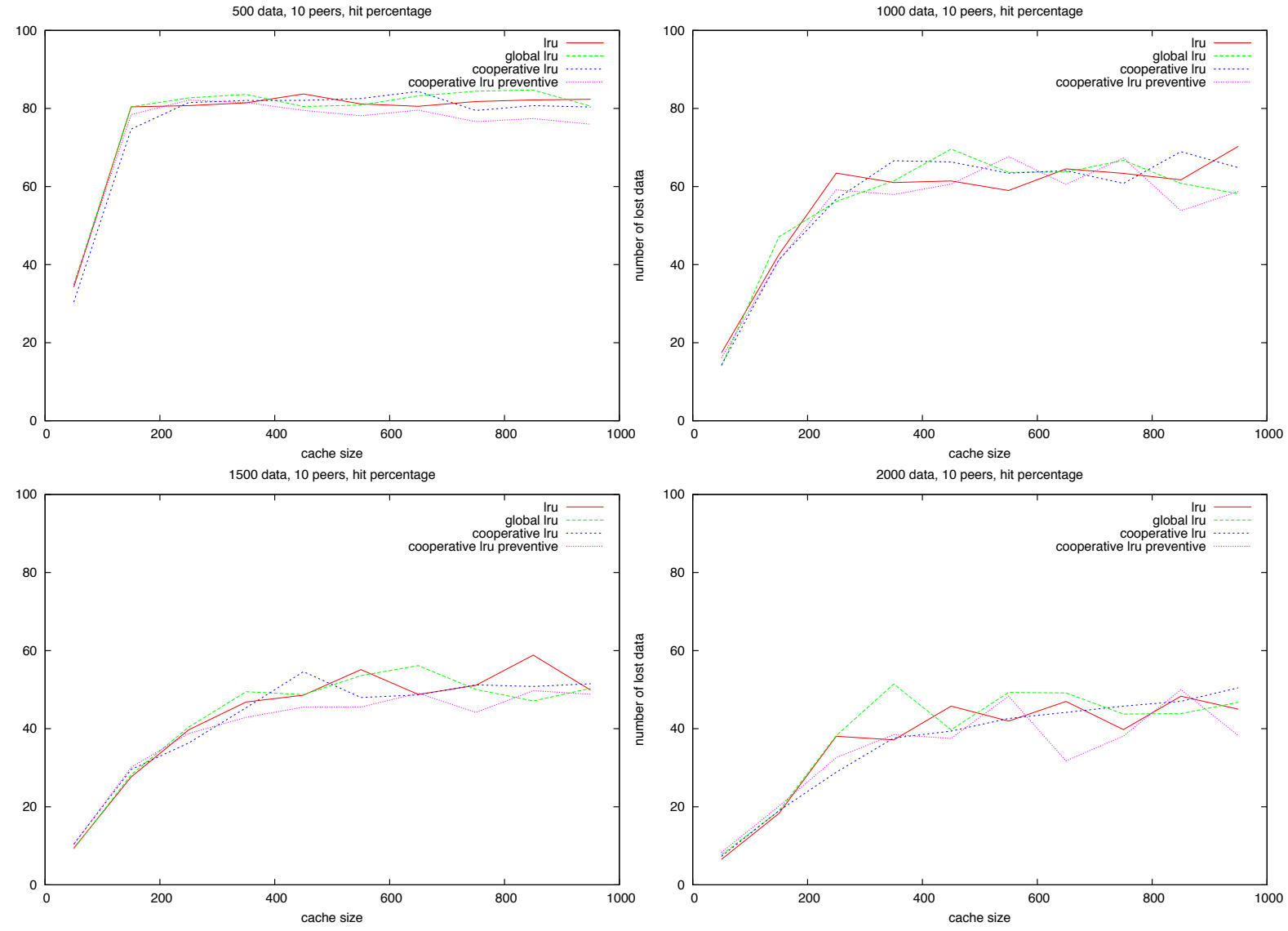


FIGURE 7.8 – Taux de succès, 10 utilisateurs

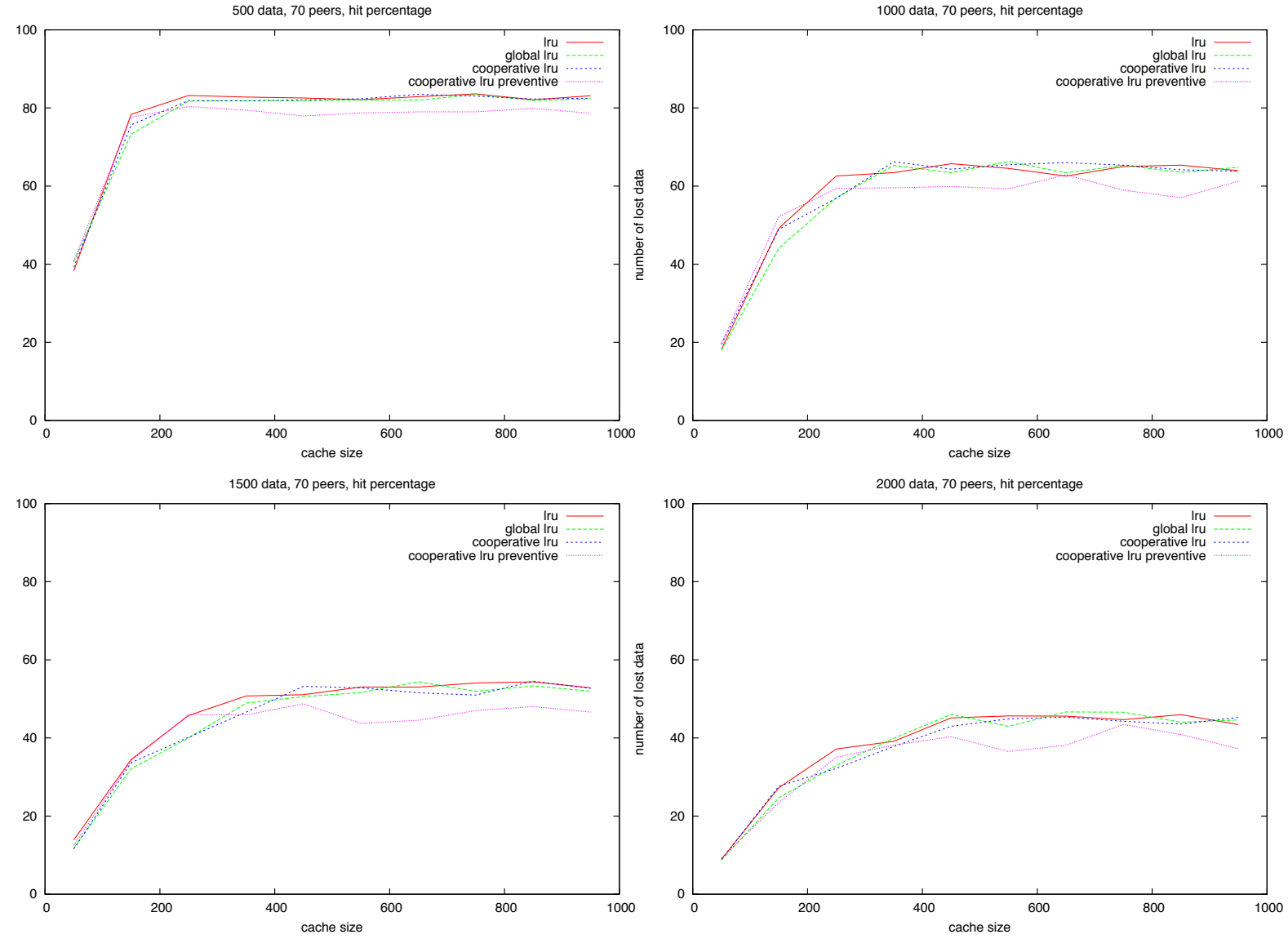


Figure 7.9 – Taux de succès, 70 utilisateurs

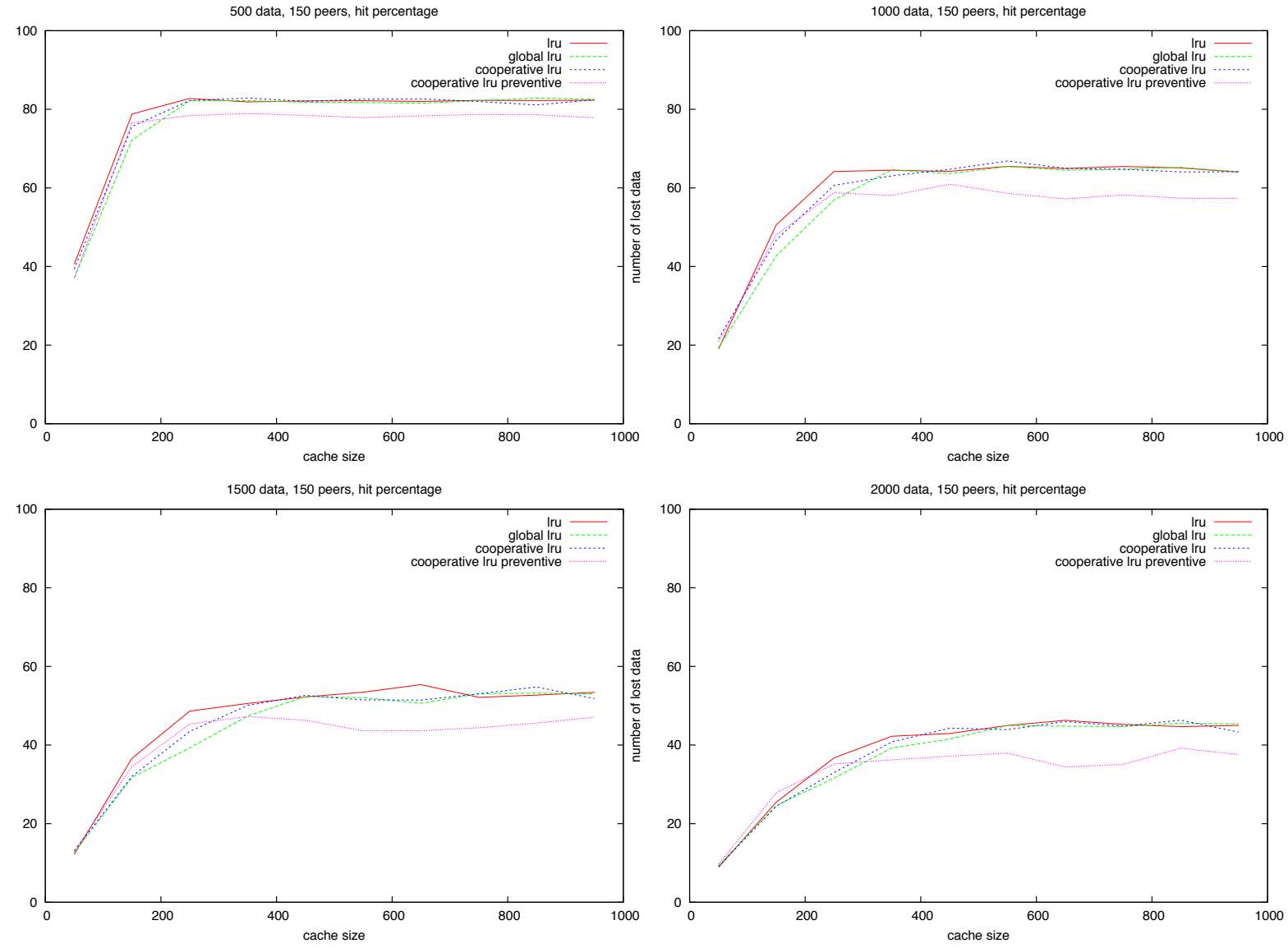


FIGURE 7.10 – Taux de succès, 150 utilisateurs

7.6 Comparaison à l'existant

Tout d'abord, de nombreux systèmes ne précisent pas quel algorithme de remplacement de cache ils utilisent. Nous avons alors considéré qu'ils utilisaient un algorithme simple, comme LRU et LFU, et nous avons vu dans ce chapitre qu'ils ne permettaient pas de préserver l'ensemble des données.
Parmi les algorithmes de remplacement de cache que nous avons vu dans l'état de l'art, la plupart suppose l'existence d'un, ou plusieurs serveurs fiables, seuls habilités à modifier les données. Ces algorithmes sont appelé quand un remplacement de cache est nécessaire. Ils éliminent alors les répliques de manière à limiter le coût d'accès global à la donnée, en fonction de critères comme sa popularité, sa fraîcheur, où la distance de la réplique la plus proche. Cependant l'assurance d'un serveur fiable autorise l'élimination de toutes les copies d'une donnée.
Seul ARAM et EARAM cherchent à minimiser le trafic réseau en éliminant des répliques ayant un coût en écriture. Ceci est mis en œuvre en prenant en compte les voisins stables (probabilité de déconnexion inférieure à un seuil). Tous les voisins stables échangent périodiquement l'ensemble du nombre d'accès en lecture et en écriture pour chaque donnée. Chaque terminal calcule ensuite un coût d'accès pour chaque réplique basé sur le ratio lecture sur écriture, et élimine les données trop coûteuses.
Cet algorithme induit donc une charge réseau périodique, que notre algorithme ne nécessite pas. Notons par ailleurs que si la propriété de voisin stable est symétrique et transitive (possible dans une période de faible mobilité), alors tous les voisins obtiennent des coûts d'accès identiques pour chaque donnée, et il y a donc un risque que toutes les répliques soient éliminées.
La probabilité que notre algorithme élimine toutes les copies d'une donnée n'est pas nulle. Cependant, comme les terminaux ne synchronisent pas leurs décisions, et utilisent chacun une mesure différente, elle n'est pas, contrairement à ARAM/EARAM dans la situation décrite ci-dessus.

7.7 Conclusion

Dans ce chapitre nous avons présenté un protocole de gestion du cache reposant sur deux principes :

1. Préserver les données,
2. Diminuer la charge réseau.

Cet algorithme agit donc à deux moments. D'une part, en cas de demande de remplacement de cache, on cherche à éliminer des répliques de manière à ce qu'il reste une autre copie de la même donnée dans le réseau. D'autre part, on cherche périodiquement à éliminer les données créant un trafic inutile dans le réseau.
Nous avons validé cet algorithme par simulation en le comparant à LRU et avons pu constater qu'il limite la perte de données et permet de diminuer le nombre moyen de messages par accès significativement, jusqu'à $\frac{1}{3}$.
La plupart des algorithmes de gestion de cache que nous avons vus dans l'état de l'art considèrent soit qu'il existe un serveur central sur lequel les données sont stockées de manière pérenne, soit, au pire, que chaque terminal source d'une donnée conserve une version qui sera toujours accessible. Elles visent donc à diminuer le coût d'accès aux données, mais n'envisagent pas les problèmes de mobilité, comme la disparition du terminal source.

La solution proposée ici prend en compte l'absence de serveur fiable et la mobilité des terminaux.

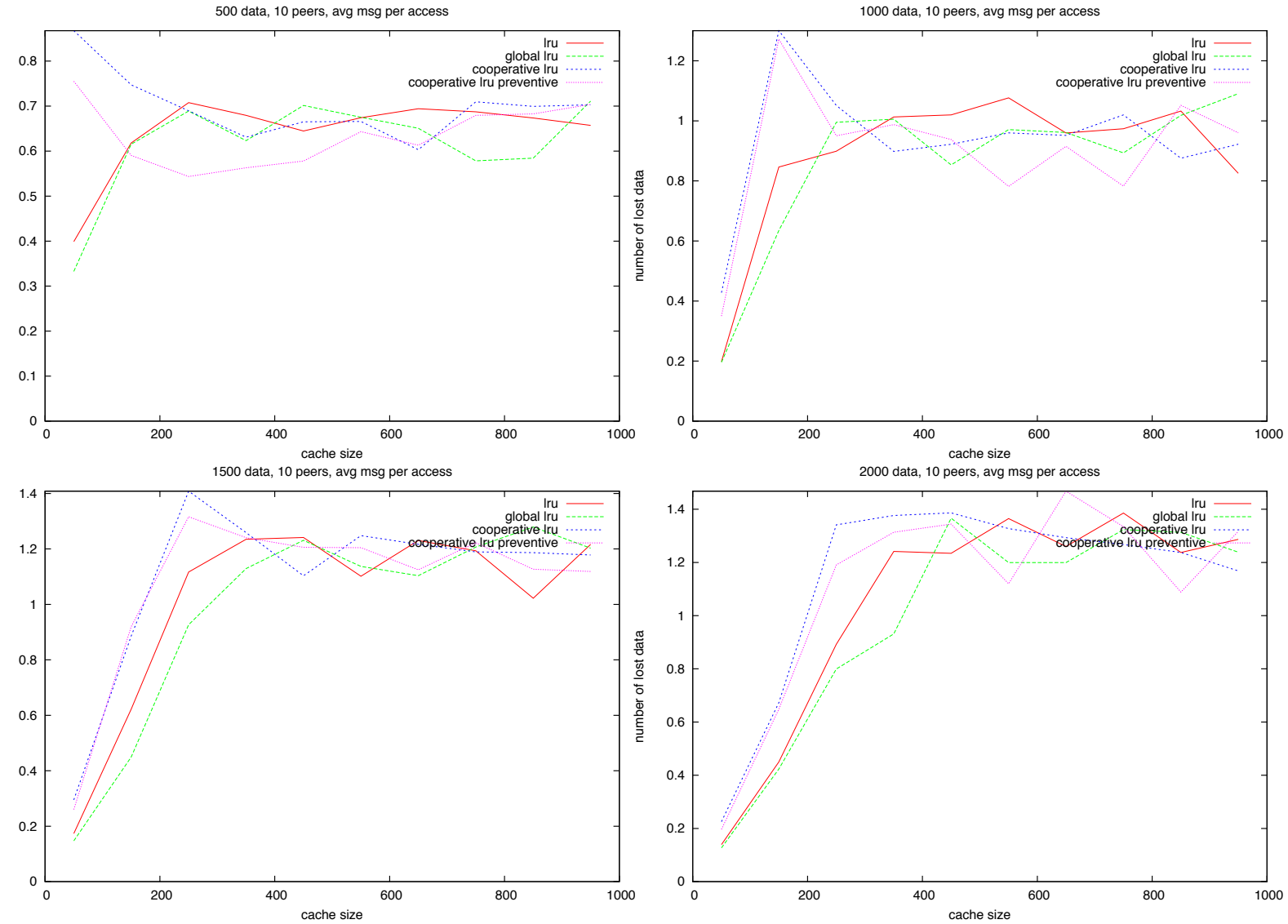


Figure 7.11 – Nombre de messages par accès, 10 utilisateurs

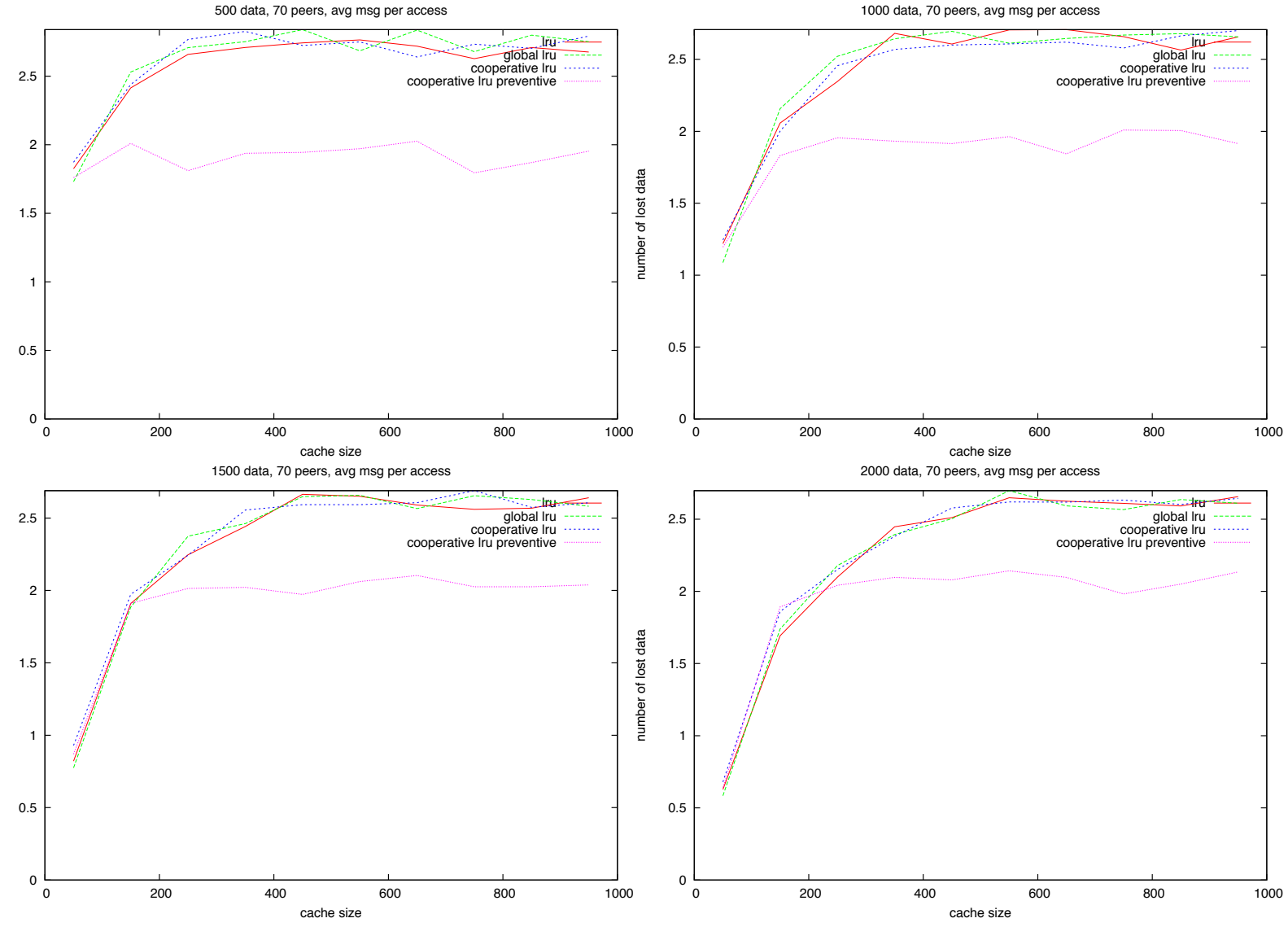


FIGURE 7.12 – Nombre de messages par accès, 70 utilisateurs

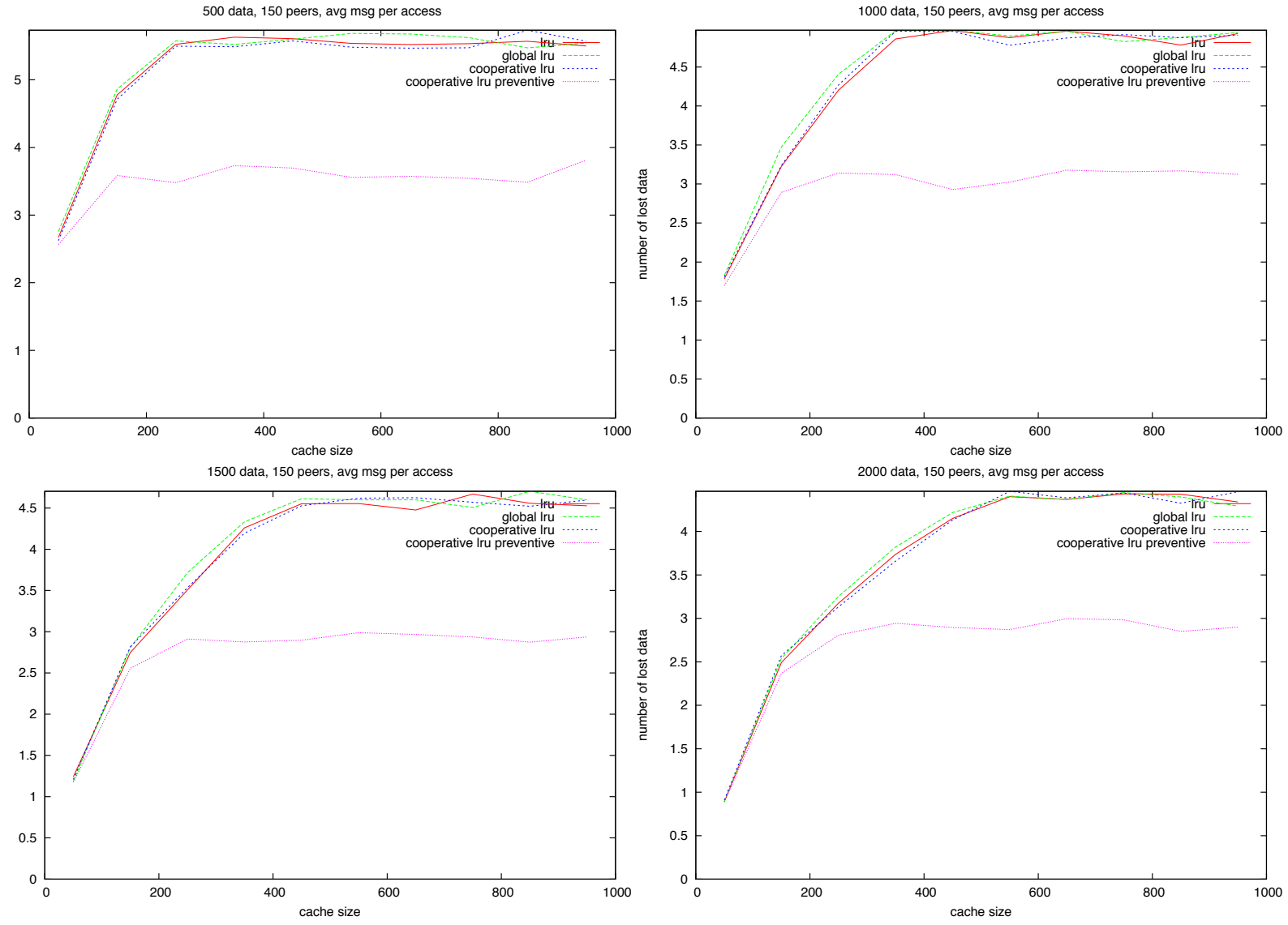


Figure 7.13 – Nombre de messages par accès, 150 utilisateurs

Chapitre 8

Prototype : un moteur de wiki pair à pair pour MANet

Dans ce chapitre nous présentons un prototype permettant de valider la faisabilité de nos algorithmes, et de les intégrer sous la forme d'un moteur de wiki pair à pair pour MANet

Il est très complexe de valider de manière systématique des algorithmes pour réseaux mobiles en les déployant en situation réelle. En effet, cela nécessite des opérateurs capables de reproduire *ad nauseam* des schémas de mobilité, et dans notre cas des accès aux données. En outre, la reproductibilité des expériences est mise à mal par les interférences que peuvent subir les communications sans fil.
C'est pourquoi nos propositions, comme tous les travaux présentés dans notre état de l'art, ont été validées par simulation.
Cependant, afin de valider de manière fonctionnelle des algorithmes proposés, nous avons aussi développé un prototype sous la forme d'un moteur de wiki pair à pair pour MANet, que nous présentons dans ce chapitre.
Dans la première section, nous présentons tout d'abord les fonctionnalités attendues du prototype.
Nous présentons ensuite les moteurs de wiki pair à pair existants. Certains permettent le travail en mode déconnecté, mais ils sont plutôt destinés aux réseaux filaires.
Nous voyons ensuite l'architecture du démonstrateur, et comment il s'intègre au sein de l'intergiciel pour MANet Transhumance, dont nous présentons brièvement l'architecture.
Dans ce démonstrateur, la cohérence des données est mise en œuvre grâce à la structure de données TreeDoc. Nous présentons donc TreeDoc plus en détail dans la section suivante, ainsi que les modifications que nous lui avons apportées afin de l'adapter à notre contexte de travail.
Enfin, nous discutons la validation fonctionnelle que nous a permis ce démonstrateur.

8.1 Démonstrateur : un wiki P2P pour MANet

Afin de démontrer la faisabilité des algorithmes proposés, nous les avons développés et intégrés au sein d'une application collaborative s'exécutant sur un intergiciel pour MANet, Transhumance
Le prototype proposé est un wiki pair-à-pair permettant le travail collaboratif en nomadisme au-dessus d'un MANet.
Dans le scénario de test, les utilisateurs travaillent habituellement au-dessus d'un réseau filaire avec un moteur de wiki centralisé. Cependant, il leur arrive de partir de leurs bureaux en groupe, par exemple pour une réunion, et de ne plus avoir accès au serveur de wiki, ou même à une infrastructure.
Le wiki pair à pair leur permet de continuer à travailler collaborativement durant cette phase nomade. Les fonctionnalités proposées sont identiques à celles offertes par un moteur de wiki monolithique classique : ils peuvent donc partager de nouveaux articles, consulter et éditer des articles existants déjà sur le wiki monolithique, et faire des recherches. Bien entendu, lorsque la phase de nomadisme se termine et que les utilisateurs retrouvent leurs bureaux, les modifications qu'ils ont apportées au wiki nomade doivent être répercutées dans le wiki monolithique.
Ce service doit être disponible (une donnée est accessible en temps borné), fiable (les données ne disparaissent pas en cas de disparition ou de partition), et doit faire un usage efficace des ressources afin de minimiser l'impact du service sur la durée de vie de la batterie.
Dans la suite de ce chapitre, nous allons tout d'abord voir pourquoi nous n'avons pas réutilisé un moteur existant, avant de présenter l'architecture du démonstrateur. Nous verrons ensuite comment la cohérence est gérée, avant de présenter l'intergiciel Transhumance.

8.2 Moteur de wiki pair à pair sur réseaux mobiles ad hoc

Avant de développer notre propre moteur de wiki, nous avons cherché s'il était plus intéressant d'adapter un moteur existant.

Nous avons étudié les moteurs de wiki pair à pair existant, que nous présentons donc dans cette section. Nous voyons aussi leur adaptabilité aux MANets.

Les motivations principales pour passer d'une architecture client/serveur à une architecture pair à pair pour un moteur de wiki sur réseau filaire sont les suivantes. Dans un moteur de wiki classique, le serveur stockant les articles constitue un goulot d'étranglement, tant au niveau de l'espace de stockage que de la bande passante. En cas de panne du serveur, le wiki devient inaccessible. Par ailleurs, le fait que les articles soient tous stockés au même endroit rend l'information plus facilement contrôlable par un petit nombre de personnes. Enfin, on voudrait distribuer le contenu du wiki sur les terminaux des utilisateurs afin de ne pas avoir à supporter le coût de l'infrastructure (stockage et bande passante) pour un serveur.

8.2.1 Distriwiki

Dans [74], Morris cherche à contourner ces problèmes en proposant Distriwiki, un moteur de wiki pair à pair pour réseaux filaires.

Distriwiki est basé sur l'intergiciel JXTA [109], une librairie de protocoles pair à pair spécifiée par SUN. JXTA permet la constitution d'un réseau de recouvrement *(overlay)*, et la découverte/publication de service. Les articles sont donc localisés grâce à ce service et répliqués à la demande.

Le passage à JXTA permet de lever les limitations sur la bande passante, et d'empêcher que les données puissent être contrôlées en étant centralisées et en un seul exemplaire. Cependant, la cohérence des données n'est pas assurée, et c'est à l'utilisateur de régler les conflits. Par ailleurs, les articles étant répliqués à la demande, il est possible de voir du contenu disparaître avant d'avoir été répliqué.

8.2.2 Wooki

Wooki [106] est un moteur de wiki pair à pair pour réseaux filaires développé au Loria. Tout comme Distriwiki, Wooki cherche à régler les problèmes liés à la centralisation du wiki. Il veut aussi permettre l'édition d'articles en mode déconnecté. Dans Wooki, chaque site réplique l'intégralité du wiki et utilise Wooto, une version optimisée de WOOT, pour gérer la cohérence des articles, avec comme granularité la ligne.

Comme nous l'avons expliqué dans la section précédente, les structures WOOT grandissent en taille sans limitation, à moins de pouvoir mettre en place une phase de compression nécessitant les conditions suivantes :

– tous les terminaux ont effectué le même ensemble de modifications,
– aucun terminal ne fait d'édition avant la fin de la phase de compression.

Sur un wiki à grande échelle, il est peu probable, bien que possible, que ces conditions soient satisfaites, Wooki étant conçu pour fonctionner en mode déconnecté.

Bien que conçu pour le travail en mode ad hoc, Wooki n'a pas été testé dans ces conditions.

8.2.3 XWiki Concerto

Dans Xwiki concerto [17], chaque terminal est un serveur de wiki embarquant l'intégralité des données.
XWiki offre un modèle de cohérence optimiste utilisant WOOT pour gérer la fusion de différentes versions des articles. Il permet le travail nomade : un utilisateur peut modifier une donnée quand il est déconnecté du reste du réseau.

8.2.4 Synthèse

Distriwiki ne gère pas la cohérence des données, et ne nous intéresse donc pas dans le cadre de ces travaux.
Wooki et Xwiki ont, en revanche, été conçus pour travailler en mode déconnecté. Cependant, l'ensemble du wiki est embarqué sur chaque terminal, ce qui n'est pas toujours possible sur un terminal mobile, et crée une charge réseau liée à la propagation des mises à jour.
Nous voulons donc mettre en place un moteur de wiki avec une gestion plus fine de la réplication des données, en utilisant TreeDoc, qui prend moins de place en mémoire, et dont la taille des identifiants plus réduite diminue la taille des messages échangés. Par ailleurs, nous voulons que le démonstrateur s'intègre à Transhumance.
Il n'est donc pas intéressant de modifier un moteur existant.

8.3 Wiki P2P : Architecture

Suite à l'étude des moteurs existants, nous avons donc décidé de développer notre propre moteur, utilisant les algorithmes décrits dans cette thèse.
Comme nous l'avons vu dans le chapitre 3, le partage de données se décompose en multiples problématiques, et nous n'avons pas fait de propositions nouvelles concernant la localisation des ressources, la cohérence des données et la recherche.
Nous nous basons d'une part sur l'intergiciel Transhumance pour les fonctions de communications, de recherche et de localisation, et d'autre part nous utilisons le type de donnée commutatif répliqué TreeDoc pour gérer la cohérence.
La figure 8.1 présente l'architecture de notre démonstrateur, et ses interactions avec l'intergiciel Transhumance.
Comme celui-ci est un wiki, l'interface présentée à l'utilisateur est celle d'un navigateur web, et notre démonstrateur est encapsulé dans un serveur http léger.
L'orchestrateur reçoit les requêtes HTTP créées par l'utilisateur et les transmet ensuite au module correspondant. De même, le gestionnaire de messages traite les messages en provenance du réseau.
Le module de réplication est chargé de mettre en place la politique de réplication que nous avons décrite dans 6. De même, le module de contrôle de cache met en place la solution décrite dans 7.
De Transhumance, que nous présentons plus en détail ci dessous, nous utilisons 3 modules :

1. Le transport avancé, qui permet les communications point à point,
2. Le service d'événements, qui permet la création d'événements persistants,
3. Le plug-in de routage, qui permet d'obtenir le contenu des tables de routage.

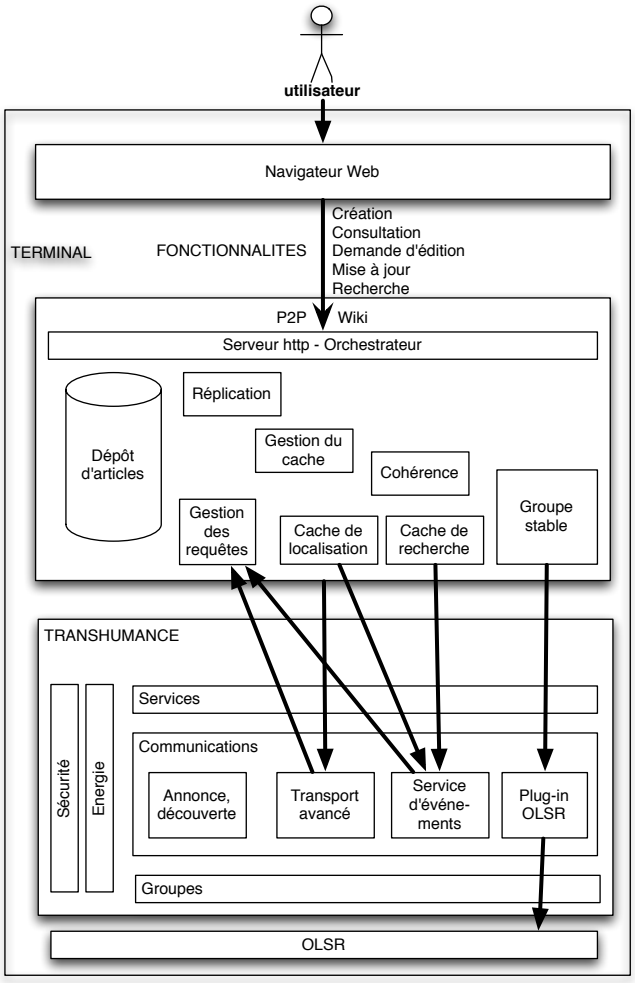


FIGURE 8.1 – Architecture du prototype, et articulation avec Transhumance

Le contenu des tables de routage est utilisé pour la création de groupes stables, comme décrit dans 5.

Les modules de localisations et de recherche utilisent le service d'événements.

Pour ce faire, le service d'événements doit être peuplé avec des événements permanents :

– quand une donnée est créée, un événement permanent contenant ses mots-clés est créé.
– quand une réplique est créée, un événement permanent indiquant quel est son hôte, est créé.
– quand une réplique est détruite, l'événement associé est supprimé.

Le module de localisation (resp. de recherche) maintient un cache de localisation (resp. de recherche), qui est peuplé quand une annonce de création de donnée, ou de création de réplique est reçue, ou grâce aux résultats d'une requête distante. Quand il est nécessaire de localiser une réplique, ou de faire une recherche, on consulte tout d'abord le cache local. Si celui-ci ne retourne pas de résultats, on interroge le service d'événements. Les résultats sont placés dans le cache local.

Enfin, le module de cohérence met en œuvre la cohérence avec TreeDoc, telle que nous la décrivons en détail dans la section suivante.

Notons que pour mettre en forme les pages de wiki, nous avons utilisé la syntaxe wiki Creole, ainsi que le parser associé [3].

8.4 Gestion de la cohérence avec TreeDoc

8.4.1 Fonctions de TreeDoc

8.4.1.1 Fonctionnalités de base

Afin d'implémenter TreeDoc, un certain nombre d'opérations sont nécessaires.
Tout d'abord, les opérations *explode* et *flatten* permettent de passer respectivement de la version tampon à la version arbre binaire d'une donnée, et inversement.
La figure 8.2 représente l'arbre obtenu suite au *explode* de la phrase "IL FAIT BEAU"

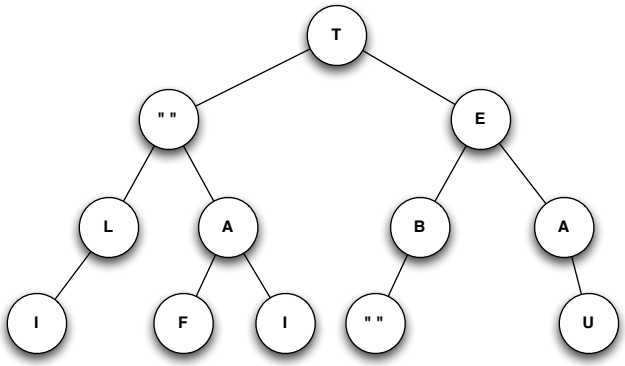


Figure 8.2 – Arbre obtenu suite au *explode*

Les deux opérations de modifications proposées sont *insert* et *delete*.
L'opération *delete* n'efface pas un caractère de l'arbre, mais le rend simplement invisible : en effet, il peut servir de référence pour le placement de nouveaux atomes pour une modification concurrente. L'opération *insert* ajoute de nouveaux nœuds dans l'arbre, mais ne modifie pas les nœuds existants.
La figure 8.3 présente l'arbre obtenu suite à une édition transformant la phrase "IL FAIT BEAU" en "HIER, IL A FAIT MAUVAIS".
Par ailleurs, une opération *newUID* est utilisée, pour construire, à partir des identifiants d'atomes précédent et successeur de l'atome à insérer, le placement le plus court dans l'arbre.
Notons que les opérations proposées ici concernent uniquement l'arbre, alors que l'édition se fait sur une version linéaire de la donnée. Nous proposons ci-dessous notre méthode pour transformer le tampon contenant les modifications de l'utilisateur, en une série d'opérateurs *insert* et *delete*.

8.4.1.2 Optimisations de TreeDoc

Nous avons vu que quand un atome était supprimé, il restait dans l'arbre. Celui-ci peut donc grandir indéfiniment.
Deux opérations sont proposées pour diminuer la taille de l'arbre :

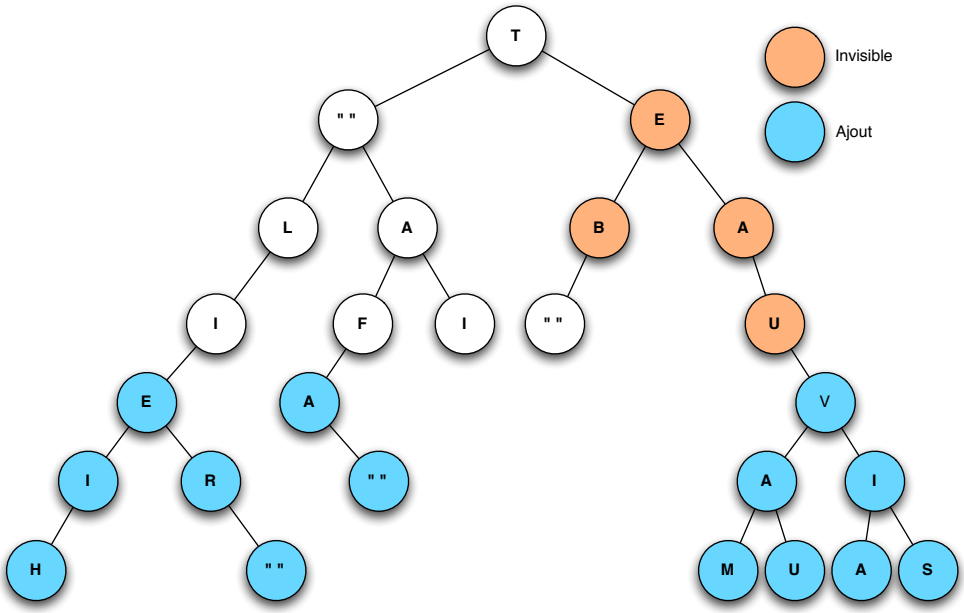


FIGURE 8.3 – Etat de l'arbre suite à une modification

– *stabledel* et *gc* : effectue une suppression d'un sous-arbre dont tous les nœuds sont invisibles.
– *flatten/explode* : rééquilibre la structure en passant sous forme linéaire, ce qui élimine les nœuds invisibles, puis en recréant un arbre binaire équilibré.

Ces deux opérations nécessitent une barrière de synchronisation, afin d'être effectuées de manière transactionnelle par tous les terminaux sur la même version de l'arbre.
En phase de mobilité, nous n'essayons pas de mettre en place un consensus en 2 phases : en effet, du fait de la mobilité, il peut exister des répliques d'une donnée hors de portée de communication dont on ne connaît pas l'existence. Il existe bien entendu des situations où le consensus serait possible, mais nous préférons ne pas mettre en place ces optimisations, car les conditions nécessaires (connaissance de l'ensemble des membres du réseau et garantie qu'ils sont tous présents) sont trop restrictives dans notre contexte de travail.
La suppression des atomes invisibles et le rééquilibrage de l'arbre ont donc uniquement lieu sur le serveur, entre deux sessions de travail nomade.

8.4.2 Ajouts de procédure

TreeDoc est un modèle de données développé à l'origine pour un éditeur de texte collaboratif, avec des utilisateurs modifiant une donnée de manière concurrente. Les modifications sont donc envoyées au fur à mesure de leur occurence.
Notre contexte de travail est différent, car les modifications peuvent avoir lieu dans des parties déconnectées du réseau. Par ailleurs, pour des raisons de charge réseau, les modifications ne sont pas envoyées au fur à mesure de l'édition, mais quand l'utilisateur le demande explicitement. La phase d'édition est donc délimitée par l'instant où l'utilisateur clique sur le bouton *Edit*, et celui où il clique sur *Commit*.
Les éditions sont donc vues au sein d'un groupe connecté selon un modèle *close-to-open*
Cependant, du fait des partitions du réseau, le modèle de cohérence global est bien la

```
def toBuffer(self, withOrdinal=False) :
    if withOrdinal:
        if not self.isInvisible :
            res =[(self.c, self.cpt)]
        else :
            res=[]
    else:
        if not self.isInvisible :
            res =self.c
        else :
            res=''
    if self.left !=None :
        res = self.left.flatten(withOrdinal)+res
    if self.right != None :
        res=res+ self.right.flatten(withOrdinal)
    self.currentFlattenUpTodate =True
    self.flattenValue=res
    return res
```

Figure 8.4 – Linéariser un arbre TreeDoc et indiquer la position de chaque nœud dans le tampon obtenu

cohérence à terme.
Nous avons donc fait quelques modifications sur la structure TreeDoc pour l'adapter à nos besoins.

8.4.2.1 Au début de l'édition

Quand un utilisateur veut lire, ou éditer un document, l'édition ne peut pas se faire directement sur l'arbre. On crée donc une version linéaire de l'arbre dans un tampon.
Afin de pouvoir trouver rapidement où placer les modifications par la suite, quand on parcourt l'arbre pour le linéariser, on indique sur chaque nœud de l'arbre sa place dans le tampon. La procédure récursive utilisée pour ce faire est présentée dans 8.4.
Afin de ne pas exécuter systématiquement cette étape, on conserve la version linéaire ainsi qu'un booléen *dirtyBuffer*, indiquant si le tampon est à jour. Ce booléen devient faux à chaque modification.
La figure 8.5 représente l'état de l'arbre suite à une opération *toBuffer*. Le résultat obtenu est le tampon "HIER IL A FAIT MAUVAIS".

8.4.2.2 A la fin de l'édition

A la fin de la phase d'édition, le résultat retourné par l'utilisateur est un nouveau tampon. Il faut donc comparer le tampon d'origine et le nouveau tampon, puis reconstruire à partir de cette comparaison les modifications sur l'arbre TreeDoc.
On effectue tout d'abord un *diff* entre les deux tampons, dont le résultat s'interprète de la manière suivante :
– '+c' : le caractère c est présent dans le nouveau texte, pas dans l'ancien
– '-c' : le caractère c est présent dans l'ancien texte et pas dans le nouveau

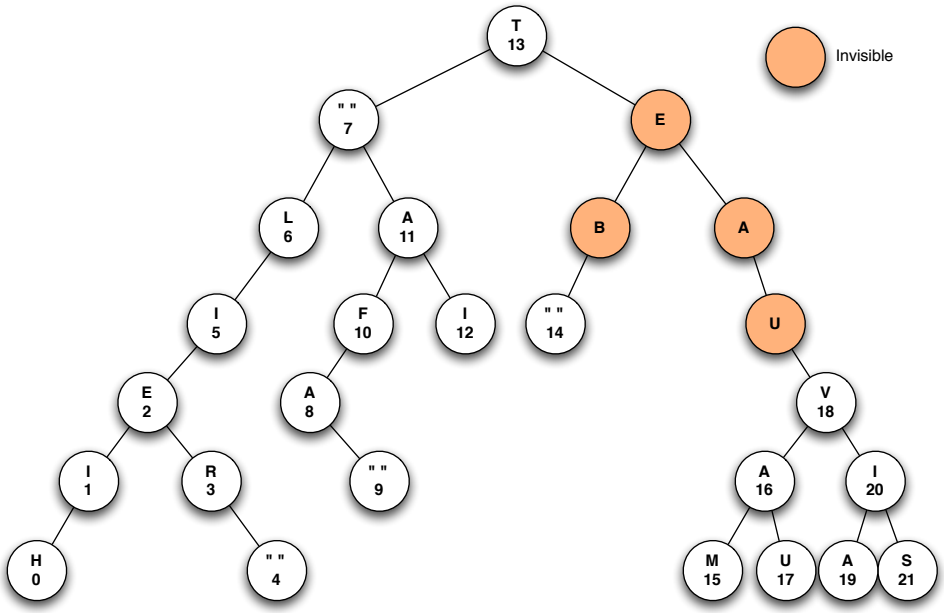


FIGURE 8.5 – Etat de l'arbre suite au *flatten*

– ' c' : le caractère c est présent dans les deux textes
Prenons pour exemple une phrase de départ :

```
il fait beau et chaud
```

Et pour phrase modifiée :

```
Hier, il a fait beau mais pas chaud
```

La différence minimale, générée par un algorithme de *diff*, est donc :

```
+h+i+e+r+,+  i l+ +a   f a i t   b e a u  -e-t+m+a+i+s+ +p+a+s   c h a u d
```

En comparant la chaîne originelle et la chaîne de différence minimale, on obtient donc une liste de modifications structurées ainsi :
– si c'est une suppression : *('delete', identifiant ordinal)*
– si c'est une insertion : *('insert', chaîne à insérer, identifiant ordinal de l'atome précédent, identifiant ordinal de l'atome suivant)*
Une structure TreeDoc étant un arbre binaire, pour N nœuds, trouver la correspondance entre les identifiants ordinaux et les nœuds de l'arbre se fait alors en $log_2(N)$.
On reconstruit alors les modifications sur l'arbre et on les diffuse dans le réseau.
Par ailleurs, on n'envoie pas une opération d'insertion pour chaque caractère, mais pour chaque suite de caractères contigüs. Comme tous les utilisateurs utilisent le même algorithme *explode*, l'arbre obtenu suite à l'insertion est identique sur tous les sites, et la taille du message envoyé est plus faible.

8.4.2.3 Différences d'arbre

Afin de pouvoir réconcilier deux arbres TreeDoc ayant beaucoup divergés, par exemple quand deux groupes d'utilisateurs se rencontrent, il est nécessaire de pouvoir reconstruire les modifications.

On utilise donc une opération permettant de parcourir deux arbres TreeDoc et de créer, à partir des différences, une liste d'insertions et de suppressions.

8.4.3 Structures de données pour la mise en œuvre de la cohérence

Chaque nœud de l'arbre est désormais constitué des informations suivantes :
- La valeur de l'atome,
- Le booléen isVisible,
- L'identifiant de l'utilisateur ayant créé cet atome,
- L'horloge,
- Le compteur ordinal indiquant la position,
- Un pointeur vers le fils droit,
- Un pointeur vers le fils gauche.

Par ailleurs, pour chaque arbre on conserve :
- une liste des modifications qu'on a reçues mais pas encore appliquées,
- une liste des hôtes p hébergeant la donnée, avec un booléen $dirtied_p$ indiquant pour chaque s'il a une copie plus récente.

8.4.4 Mise en œuvre de la cohérence

Nous présentons tout d'abord les étapes de départ en mode nomade, qui signale le début de l'utilisation de TreeDoc pour mettre en œuvre la cohérence, et de retour au serveur. Nous voyons ensuite comment, en mode nomade, les modifications sont propagées dans une situation stable. Enfin, nous présentons comment les structures de données nécessaires à la cohérence sont maintenues dans le cas d'une fusion de groupes.

8.4.4.1 Début et fin de la phase de nomadisme

Hors des phases de nomadisme, la cohérence des données est gérée de manière classique centralisée. TreeDoc est utilisé uniquement pendant les phases de nomadisme.
Quand les terminaux initient une session de nomadisme, tous les terminaux doivent partir avec la même version de la donnée afin d'avoir des arbres TreeDoc de départ identiques.
Le serveur calcule le taux de réplication en fonction du nombre de données, et du type des terminaux. Les données sont ensuite répliquées de manière égoïste puis de manière statistique.
Elles ne sont transformées en arbre TreeDoc que quand une édition a lieu, mais tous les terminaux ayant la même version de la donnée, et le même algorithme *explode*, ils obtiennent tous indépendamment le même arbre TreeDoc.
Lors de la phase de nomadisme, la cohérence des répliques est gérée avec TreeDoc.
Quand les terminaux reviennent au serveur, ils lui transmettent toutes les modifications effectuées puis éliminent toutes les copies de leurs caches. Tant que tous les terminaux ne sont pas revenus, les données sont conservées sous forme TreeDoc afin de pouvoir intégrer les modifications.
Une fois que tous les terminaux se sont signalés, l'arbre TreeDoc est aplati puis détruit et la gestion de la cohérence reprend de manière classique.

8.4.4.2 Propagation des modifications lors de la phase de nomadisme

La propagation des données est hybride :

– au sein d'un groupe stable : invalidation.
– hors du groupe stable : mise à jour.
Quand un pair met à jour une donnée, il envoie ses modifications aux hôtes de la donnée hors de son groupe stable . Au sein du groupe stable, il envoie seulement une invalidation. A la réception d'une invalidation du pair p, le booléen $dirtied_p$ est mis à Vrai.
Par la suite, quand un pair utilise une donnée, il demande les mises à jour aux terminaux dont il sait qu'ils ont une version plus récente. Quand la modification du pair p est récupérée, $dirtied_p$ passe à Faux.

8.4.4.3 Fusion de groupes

Lors de la fusion de deux groupes, il est nécessaire que les hôtes hébergeant la même donnée se découvrent et s'associent afin que chaque groupe intègre les modifications de l'autre groupe.
Dans chaque groupe, le terminal ayant l'identifiant le plus faible devient responsable de groupe. Pour chaque donnée, et dans chaque groupe, le dernier terminal à avoir effectué une modification devient responsable de la donnée.
Les responsables de donnée se signalent auprès de leur responsable de groupe respectif qui les associent par paire (si deux terminaux se déclarent responsables pour cause d'édition concurrente, celui avec le moins de données en charge est choisi), et leur signalent quel terminal est leur interlocuteur.
Les terminaux ainsi appariés échangent les arbres TreeDoc ainsi que la liste des hôtes dans chaque groupe, puis construisent la liste des modifications. Chacun diffuse ensuite ces modifications, ainsi que la liste des hôtes, au sein de son ancien groupe.

8.5 Transhumance

Transhumance est un intergiciel pour MANet issu du projet ANR du même nom. Sa particularité est que Transhumance s'adapte à l'énergie : chaque service est proposé en différentes variantes offrant chacune un compromis différent entre qualité de service et utilisation de la batterie.
L'application de démonstration de Transhumance est un jeu de piste [30] qui a été testé par des étudiants de Télécom Paris.

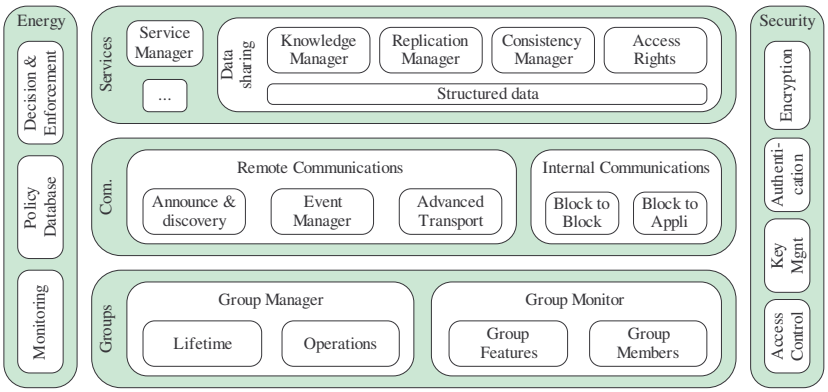


FIGURE 8.6 – Architecture de Transhumance,extrait de [81]

Dans cette section, nous présentons l'architecture de Transhumance 8.6, et les particularités de chaque module.

8.5.1 Groupes

L'organisation du travail dans Transhumance se fait par groupes d'intérêt. Par défaut, tous les terminaux appartiennent au groupe Transhumance. Ils peuvent ensuite décider de rejoindre d'autres groupes. Chaque groupe partage un secret, utilisé pour le chiffrement.
Les groupes sont la base du travail dans Transhumance : les applications collaboratives sont instanciées par groupe (par exemple les données sont partagées au sein du groupe), et la sécurité des communications est mise en place en sein d'un groupe, via le chiffrement.
Ces groupes ne sont pas liés aux groupes de stabilité que nous avons décrit dans 5. Les membres d'un groupe d'intérêt peuvent donc ne pas être en contact.

8.5.2 Communication

8.5.2.1 Routage

Le développement d'un protocole de routage ne faisant pas partie des objectifs du projet, le protocole OLSR, présenté en détail dans le chapitre 5, a été choisi.
L'implantation OLSR-Unik [65], choisie pour sa robustesse, a par été ailleurs adaptée pour permettre de remonter des informations sur la topologie aux modules supérieurs.

8.5.2.2 Transport

Un protocole de transport dédié à Transhumance a été développé. Il est basé sur UDP et propose 3 variantes :
- mode simple : fonctionne comme UDP.
- mode avec acquittement : un temporisateur est armé pour chaque message envoyé. Quand un terminal reçoit un message, il envoie un acquittement. Si le temporisateur se déclenche avant réception de l'acquittement, le message est renvoyé.
- mode sécurisé : fonctionne comme le mode avec acquittement, mais les communications sont par ailleurs chiffrées.

8.5.2.3 Gestionnaire d'événement

Transhumance permet la communication basée sur les événements sur un modèle *"publish/subscribe"*. Un utilisateur peut donc émettre un événement avec un type (donnée, requête, etc) et un sujet auquel un contenu est associé, ou souscrire à un ensemble d'événements d'un certain type avec un sujet précis.
Ce système se base sur le protocole OLSR en utilisant les MPR comme *"broker"* : les événements sont publiés et diffusés à travers un MPR, et les souscriptions à un événement sont stockées sur les MPR.
Ce sont donc eux qui associent un événement (type et sujet) à une souscription (type et sujet) et transmet l'événement au pair ayant effectué la souscription.

8.5.2.4 Annonces et découvertes de services

Le module d'annonce et découverte de service permet à un utilisateur de diffuser au sein d'un groupe un nouveau service : création d'un nouveau groupe, partage d'une nouvelle

donnée, etc.
Il se base pour cela sur le gestionnaire d'événements.

8.5.3 Gestion de l'énergie

La gestion de l'énergie dans Transhumance se fait en adaptant les services proposés par l'intergiciel au niveau de la batterie.
Un compromis est donc fait entre la qualité du service offert et la durée de vie de la session de travail.
Par exemple, un terminal ayant un niveau de batterie bas peut décider de passer d'un mode de communication chiffrée à une communication en clair, ou décider de ne plus servir de données.

8.5.4 Sécurité

La gestion de la sécurité est répartie en 4 modules :
– l'authentification des utilisateurs,
– la gestion des secrets,
– le chiffrement des données, qui utilise les clés du module précédent,
– le contrôle d'accès, qui assure qu'un utilisateur ne peut pas, par exemple, accéder aux données d'un groupe auquel il n'appartient pas.

8.5.5 Partage de données

Dans le cadre de ce démonstrateur, nous n'utilisons pas le partage de données proposé par Transhumance, car nous implantons notre propre gestion des données.

8.6 Implantation et validation

8.6.1 Implantation

8.6.1.1 Modules implémentés

Nous n'avons implanté que la partie nomade du wiki, car c'est durant cette phase que les algorithmes proposés s'appliquent. Cependant, la phase de départ et la phase de retour utilisant des algorithmes développés pour la phase nomade, ajouter cette gestion ne nécessite pas un développement important.
La version actuelle du démonstrateur, développée avec l'aide d'un groupe d'étudiants de Télécom Paristech, permet donc aux utilisateurs de rechercher et d'accéder à des articles, de créer de nouveaux articles, et de les modifier. Notons que, pour l'instant, la cohérence des données n'est pas gérée avec TreeDoc, et la dernière modification l'emporte donc sur les autres.
Nous avons développé la variation de TreeDoc que nous avons présentée plus haut, adaptée au besoin de gestion de cohérence de notre application mais elle n'est pas encore intégrée au démonstrateur.
L'extraction de mots-clés depuis des accès n'a par contre pas été validée avec ce prototype, car elle nécessiterait une utilisation régulière du wiki avec de véritables accès.

8.6.1.2 Emulation sur réseau filaire.

Le portage sur Transhumance nécessite une adaptation de Transhumance lui-même (pour l'instant, les applications doivent être compilées directement avec l'intergiciel), adaptation en cours au sein de l'équipe.

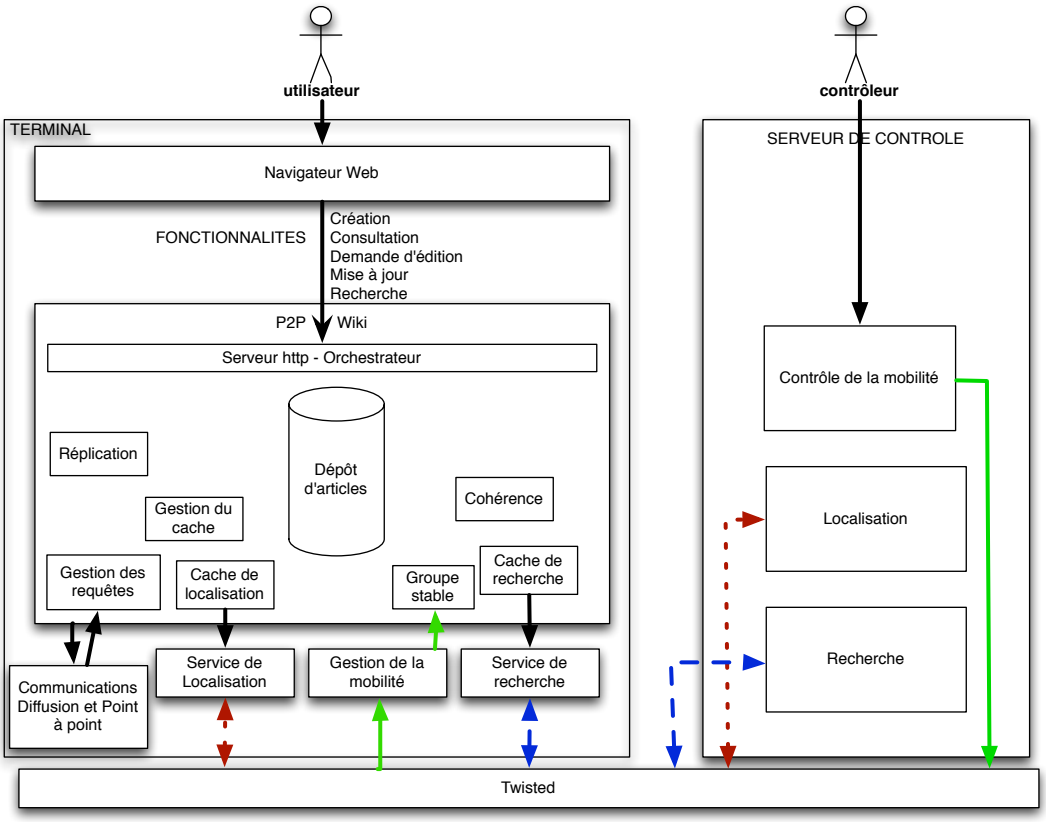


Figure 8.7 – Architecture du prototype, adapté pour s'exécuter sur un réseau filaire

Nous avons donc développé quatre services pour émuler Transhumance, comme représenté dans le schéma 8.7 :

- Un module de localisation : il remplace le module de recherche basé sur le système d'événements, et interroge un serveur contenant l'ensemble des informations de localisation pour chaque réplique.
- Un module de recherche : il remplace le module de recherche basé sur le système d'événements, et interroge un serveur contenant l'ensemble des informations sémantiques pour chaque donnée.
- Un module de mobilité : il nous permet de contrôler à un instant quels terminaux sont à portée de communication.
- Un module de communication : il permet de faire communiquer les différents terminaux exécutant le moteur de wiki pair à pair en diffusion et en point à point au-dessus d'un réseau filaire. Ce module a été développé à l'aide du framework python Twisted [29].

8.6.2 Validation

Grâce à notre prototype, nous avons pu valider les fonctionnalités présentées plus tôt grâce au scénario suivant.
Quatre utilisateurs, A, B, C et D, utilisent le wiki. Pour pouvoir mettre en exergue le remplacement de cache, les tailles de caches sont artificiellement limitées à 4. On doit donc avoir deux copies de chaque article.
La distribution au départ est la suivante :
– A héberge les articles *Niamey, Tokyo, Hanoi, Yaounde* (triés ici par date d'accès, Niamey ayant été utilisé le plus récemment).
– B héberge les articles *Yaounde, Rome, Tokyo* .
– C héberge les articles *Managua, Niamey, Tokyo, Rome.*
– D héberge les articles *Rome, Caracas, Managua.*
Durant une période, B et C modifient chacun deux fois l'article *Tokyo*, alors que A ne l'utilise pas du tout. Quand vient le moment de décider de l'éviction préventive des répliques données, A élimine donc *Tokyo.* C, D et B accèdent à *Rome* en lecture.
L'état des répliques est le suivant :
– A : *Niamey, Hanoi, Yaounde.*
– B : *Tokyo, Rome, Yaounde* .
– C : *Tokyo , Rome, Managua, Niamey.*
– D : *Rome, Caracas, Managua.*
C crée une nouvelle donnée, *Ottawa.* Tout d'abord, il doit sortir une donnée de son cache. L'article utilisé le plus anciennement est *Niamey.* Cependant, il n'en existe que deux copies, il n'est donc pas sorti. Il en est de même pour *Managua.* L'article *Rome* est donc éliminé. Ceci fait, il place donc *Ottawa* dans son cache puis diffuse l'information de création. Comme nous n'avons pas extrait de mots-clés pour chaque utilisateur, on utilise uniquement la réplication statistique, et A, B et D répliquent avec une probabilité de 1/3. Seul D crée une réplique de *Ottawa.*
L'état des répliques est le suivant :
– A : *Niamey, Hanoi, Yaounde.*
– B : *Tokyo, Rome, Yaounde.*
– C : *Ottawa, Tokyo , Managua, Niamey.*
– D : *Rome, Caracas, Managua, Ottawa.*
B modifie l'article Rome, et quand A l'accède en lecture, il voit les modifications de B.
Enfin, quand le terminal C disparaît, et que B essaie d'accéder à la donnée *Ottawa*, celle-ci est disponible et présente sur D.

Ce prototype nous a donc permis de valider la faisabilité d'une grande partie des fonctionnalités supportées par les algorithmes proposés.

8.7 Conclusion

Dans ce chapitre nous avons présenté un prototype de moteur de wiki pair à pair pour réseaux mobile ad hoc, permettant de valider la faisabilité de nos propositions.
Ce prototype intègre les algorithmes proposés dans cette thèse pour gérer la mobilité des terminaux, la réplication et le remplacement de cache. Il s'appuie sur l'intergiciel pour réseaux mobiles ad hoc Transhumance pour les fonctions de communications, de localisation et de recherche et sur TreeDoc pour la gestion de la cohérence. Nous avons donc présenté

Transhumance, ainsi que TreeDoc, que nous avons adapté à notre contexte. Comme l'intégration sur Transhumance n'a pas été possible, nous avons développé des modules simples permettant d'émuler les services de Transhumance.
Grâce à ce prototype, nous avons pu valider les fonctionnalités proposées dans cette thèse sur un scénario simple. Les propriétés non fonctionnelles de disponibilité, fiabilité et disponibilité ont, elles, été validées de manière systématique par simulation.

Chapitre 9

Conclusion

9.1 Bilan

Les terminaux mobiles équipés de modules de communications sans fil étant maintenant courants, de nombreuses applications collaboratives usuellement utilisées sur des terminaux fixes peuvent désormais être portées sur ces réseaux sans fil.
Bien que les infrastructures de communications haut débit soient de plus en plus accessibles, notamment avec le déploiement du réseau 3G+, les MANets peuvent s'avérer utiles dans des circonstances particulières, mais pas nécessairement improbables.
MANet est un terme générique, qui regroupe un ensemble de configurations de réseaux allant du réseau de capteurs au réseau de véhicules. Il est donc nécessaire de préciser que nous travaillons dans le cadre de réseaux d'humains piétons, enclins à former des groupes pour collaborer.
La conception et le déploiement d'applications collaboratives distribuées partageant des données soulève de nombreuses problématiques dans les réseaux filaires, comme la cohérence des données ou leur localisation. Le contexte du MANet entraîne encore de nouvelles contraintes, et nécessite donc une conception différente du partage de données.
Dans cette thèse nous avons proposé des algorithmes pour le partage de données sur MANets.

9.2 Contributions

Nous avons travaillé avec 2 objectifs :
– s'assurer que le service de partage de données soit disponible et fiable : les données ne doivent pas disparaître et doivent être accessibles en temps borné.
– limiter l'utilisation du réseau : l'utilisation du réseau, même en écoute, impacte la durée de vie de la batterie.

Nous avons proposé un ensemble d'algorithmes pour mettre en œuvre le partage de données éditables, conçu pour être intégré au sein de l'intergiciel Transhumance : un algorithme de

création de grappes de mobilité, un algorithme de réplication pro-actif et un algorithme de remplacement de cache.
L'algorithme de création de groupes stables dans le temps afin de gérer la mobilité a été proposé pour mettre en place des algorithmes collaboratifs, où les terminaux mettent leurs ressources en communs pour une plus grande efficacité de leur utilisation. Il n'est intéressant de collaborer qu'avec des terminaux présents en continu sur une période de temps.
Pour cela, nous avons choisi de profiter du choix fait que le système Transhumance utilise un algorithme de routage pro-actif, OLSR, et de baser les informations de présence nécessaires sur le contenu des tables de routage.
L'atout de cette proposition, par rapport aux algorithmes existants, est de ne créer aucune surcharge réseau, et de ne pas nécessiter de matériel particulier, comme par exemple un capteur GPS. En sus de cette propriété simplement quantifiable, nous avons aussi mesuré par simulation la précision de notre algorithme au sein du simulateur NS-3, et obtenu des résultats satisfaisants.
Nous avons ensuite utilisé ces groupes pour mettre en place un algorithme de réplication des données pro-actif et collaboratif. Il tente de créer un nombre de répliques préventives proportionnel au nombre de terminaux dans le groupe (au maximum 15%). Il place celles-ci en priorité sur les terminaux les plus susceptibles de les utiliser (selon une métrique d'intérêt basée sur le contenu de la donnée). Si le nombre de répliques est insuffisant, les terminaux ayant le plus de ressources vont répliquer pour leurs voisins directs, si ceux-ci sont intéressés. Enfin, s'il manque toujours des répliques, on place le reste au hasard.
Cet algorithme a pour but d'une part de diminuer le temps d'accès aux données, et donc d'en augmenter la disponibilité, et d'autre part de prévenir la disparition des données en cas de partition, ou même simplement de remplacement de cache. Les critères d'évaluation ont donc été, d'une part le nombre de données disparaissant en cas de partition du réseau, et d'autre part, bien-sûr, la surcharge réseau par rapport à un algorithme de réplication à la demande.
Nous avons validé cet algorithme par simulation et constaté qu'il prévenait la disparition des données, et diminuait le nombre de sauts moyens nécessaires pour accéder à une donnée. La surcharge réseau induite par notre algorithme est liée au coût de mise à jour de répliques non utilisées. Elle est donc dépendante de la distribution des accès aux données et de leur mode (lecture/écriture). Pour des données utilisées en lecture seulement, il n'y a pas de surcoût par rapport à l'algorithme de réplication à la demande. Pour des données utilisées en écriture, si plus d'utilisateurs sont intéressés par la donnée que le nombre de copies préventives, il n'y a pas de surcoût par rapport à l'algorithme de réplication à la demande.
Parmi les systèmes existants, beaucoup d'entre eux n'indique pas quel algorithme de réplication ils utilisent. Nous avons donc considéré qu'ils utilisaient la réplication à la demande. D'autres propositions améliorent la réplication à la demande en ne répliquant que si l'hôte servant la données est à une certaine distance. Dans une troisième proposition, qui suppose un graphe de réseau assez stable, chaque pair examine les requêtes qui transitent par lui ; si une donnée est très demandée, il en crée une copie locale.
Ces solutions ne garantissent pas une couverture des données en cas de partition, puisque les données les moins utilisées sont peu répliquées. De fait, elles considèrent souvent que ces données sont hébergées par un serveur fiable et passent outre les problèmes de partition.
Le système adHocFS utilise par contre des copies préventives afin de prévenir la perte d'une donnée si les copies de travail disparaissent. Dans le cas d'une donnée peu utilisée à un instant donné, on crée seulement une copie préventive, ce qui mène le nombre de répliques à deux. Cela permet effectivement de tolérer les disparitions de pairs, mais comme nous

l'avons vu dans le chapitre concerné, en cas de partition du réseau, ne garantit pas un taux de couverture raisonnable.

Enfin, les travaux de Hara cherchent à placer globalement les données de manière à avoir le minimum de copies nécessaires et maximiser l'utilisation de l'espace mémoire. Pour ce faire, ils placent collaborativement les données au sein d'un groupe de proximité. Ces algorithmes sont destinés à des réseaux de capteurs et supposent la connaissance par tous les pairs d'information comme la fréquence d'accès de chaque pair à chaque donnée, la fréquence de mise à jour de toutes les données, ou la corrélation de chaque paire de données.

Les terminaux ayant une capacité de stockage limitée, vient un moment où il est nécessaire d'éliminer des répliques. Utiliser un simple algorithme LRU, qui se base des informations locales uniquement, fait courir le risque de faire disparaître des données.

Nous avons donc proposé un algorithme qui vise d'une part à limiter la disparition des données, en éliminant les répliques de manière à ce qu'il en reste toujours suffisamment, et d'autre part un algorithme qui élimine préventivement les répliques de données générant du trafic inutile.

On demande à un algorithme de remplacement de cache d'éliminer les données les moins susceptibles d'être à nouveau utilisées, afin de diminuer le nombre de défauts de cache. Nous l'avons donc évalué en terme de taux de succès. Une particularité de notre système étant que si la dernière réplique d'une donnée est détruite, celle-ci n'existe plus, nous avons donc évalué aussi ces algorithmes en terme de données perdues à l'issue d'une session de travail. Enfin, l'éviction préventive des données générant trop de trafic cherche à satisfaire la contrainte non fonctionnelle d'efficacité : nous avons donc évalué l'algorithme en terme de charge réseau.

Nous avons comparé nos algorithmes à LRU et constaté qu'ils donnent lieu à une perte de données bien moindre. Le nombre de taux de succès est légèrement diminué par l'algorithme préventif, mais le nombre de messages échangés en moyenne par accès peut être diminué de moitié dans certains cas.

Parmi les algorithmes de gestion de cache proposés dans l'état de l'art, seul l'un d'entre eux tente d'éliminer préventivement les répliques générant trop de trafic réseau. Pour ce faire il nécessite un échange périodique au sein d'un voisinage stable du nombre d'accès en lecture et en écriture de chaque pair, à chaque donnée. Notre algorithme ne crée pas cette surcharge réseau. Par ailleurs, comme nous l'avons vu dans le chapitre concerné, cet algorithme peut dans certaines situations entraîner l'élimination de toutes les répliques d'une donnée.

Les autres algorithmes éliminent les données uniquement quand un remplacement de cache est nécessaire. Quand une réplique à éliminer doit être choisie, les critères utilisés sont la distance de la copie la plus proche, l'utilité de la donnée (fréquence d'accès) et la fraîcheur de la donnée. Ces systèmes font l'hypothèse d'un ou plusieurs serveurs de données fiables, ou que la réplique est toujours, dans le pire des cas, disponible à la source : la possibilité de partition ou de disparition n'est pas prise en considération.

Dans cette thèse, nous n'avons pas proposé de modèle de cohérence innovant car après étude des travaux existants, nous avons choisi le paradigme des Types de Données Commutatives Répliquées pour mettre en œuvre un modèle de cohérence à terme. Nous avons ainsi choisi d'utiliser TreeDoc.

Enfin, nous avons conçu un démonstrateur de nos algorithmes sous la forme d'un moteur de wiki distribué, qui est partiellement implanté.

9.3 Recherches futures et problèmes en suspens

Comme nous l'avons vu, les problèmes liées au partage de données sur MANet sont multiples et nous ne les avons pas tous traités dans cette thèse.
Nous présentons ici quelques perspectives de recherche, visant l'amélioration et la validation de nos propositions, ou l'intégration à nos travaux de nouvelles contraintes non fonctionnelles.
Lors de ces travaux nous faisons l'hypothèse que tous les utilisateurs dans le réseau ont droit d'utiliser les données placées dans l'espace de partage. Dans Transhumance, les applications collaboratives sont partagées au sein de groupes de sécurité. Se pose alors la question d'intégrer notre proposition dans ce modèle de sécurité.
Nous avons deux alternatives :

- Répliquer les données uniquement sur les terminaux du groupe de sécurité, (auquel cas la cohérence voudrait qu'on exclut les non membres du routage des messages) au prix d'un espace de partage plus restreint. Cependant, comme nous l'avions indiqué plus haut, cela ne nécessite pas de modification de notre proposition.
- Chiffrer les données avant de les héberger hors du groupe, et devoir payer le coût du chiffrement et du déchiffrement (coût d'usage du processeur, et donc de la batterie, et coût de la qualité de service perçue par l'utilisateur qui subit la latence induite par le déchiffrement). Chiffrer les données a une autre limite : les mises à jours ne peuvent pas être appliquées sur une donnée chiffrée, à moins d'un modèle de cohérence où les accès sont totalement ordonnés, et suivis d'un renvoi de la donnée dans son intégralité.

Pour des données non modifiables (images, son, vidéo), les cycles de chiffrement/déchiffrement successifs liés aux modifications ne constituent pas un problème. Une méthode mixte pourrait donc être envisagée.
Un autre problème est celui de la protection de la vie privée. Tout d'abord, afin de pouvoir répliquer des données pro-activement, nous proposons de créer un profil des intérêts de l'utilisateur en examinant ses accès. Bien que la plupart des utilisateurs acceptent de voir leurs activités examinées en contrepartie de meilleures offres ou recommandations, il faut pouvoir désactiver ce système pour ceux ne le souhaitant pas. Cependant, dans les systèmes tels qu'Amazon ou Google, ces données sont centralisées sur les serveurs des firmes en questions, qui mettent en rapport les profils des utilisateurs pour faire du filtrage collaboratif sans que ceux-ci aient accès à des informations ne les concernant pas.
Nous travaillons dans un environnement par nature distribué, ce qui rend plus problématique l'échange de mots-clé au sein d'un groupe afin de mettre en place de la réplication collaborative.
Pour préserver un droit à la vie privée, et si on ne souhaite pas désactiver cette option, il serait possible de travailler non pas avec les mots-clés eux-même mais avec des empreintes, réalisées par exemple avec SHA. Ce garde fou pourrait cependant être contourné, comme pour le chiffrement des données, par un utilisateur ayant l'ensemble des mots-clés et un peu de temps à sa disposition pour explorer l'ensemble de l'espace.
Notons que même en désactivant ce système, examiner les requêtes et les mises à jour émises par un utilisateur permet d'identifier à quels documents il s'intéresse.
On peut aussi mettre en regard les risques relatifs au fait de stocker ses mails et ses documents à l'étranger sur les serveurs d'une société privée et le risque présenté par notre système, mais ceci est l'objet d'un tout autre débat.
Concernant les mots-clés, il serait intéressant de se pencher plus longtemps sur les problèmes d'extraction d'information depuis le contenu, et de recommandation. En effet, la

réplication sémantique est d'autant plus efficace que la prédiction des accès est précise.
Un autre aspect à considérer est la diversité des types de données. Un article de wiki, par exemple, est une donnée multimédia composée de texte, d'image, et parfois de son, mais dans ces travaux nous le considérons comme une seule donnée. Lors de la phase de remplacement de cache, plutôt que d'éliminer un article dans son ensemble, il serait intéressant d'explorer la possibilité de donner une priorité à la disponibilité de certains media.
Nous avons vu que pour mettre en œuvre la cohérence des données, nous utilisons TreeDoc, avec une propagation des mises à jour paresseuse *(lazy)* au sein d'un groupe, et sans délai *(eager)* entre groupes. Le problème au sein d'un groupe est alors que les copies créées pour maintenir une fiabilité et une disponibilité élevée mais qui ne sont pas utilisées risquent de ne pas être à jour, ce qui engendre une perte d'informations lors d'une partition. Il faudrait étudier un modèle qui, sans générer un trafic aussi important que la mise à jour sans délai, maintienne ces copies cohérentes.
Enfin, nous aimerions effectuer plus de tests avec des jeux d'accès et de mobilité réels.
Quand nous avons cherché un jeu de données pour valider notre algorithme de réplication, nous avons trouvé deux types de traces. D'une part, des traces issues de systèmes d'exploitation, indiquant les accès en lecture et en écriture fait par les utilisateurs, mais sans information sur les données accédées. D'autre part, des traces de wiki et de subversion, avec possibilité d'obtenir les données elles-même, mais indiquant uniquement les accès en écriture. En effet, beaucoup de systèmes, comme wikipedia ou les dépôts svn hébergés sur sourceforge, demandent d'être authentifié pour effectuer une modification, et journalisent celles-ci, mais peuvent être lus de manière anonyme, et ces lectures ne sont donc pas tracées.
De même, nous aimerions tester notre algorithme de création de grappes sur des traces de mobilité réelles. Celles-ci n'étant pas disponibles, les acquérir nécessiterait une campagne de tests d'envergure.

Publications

Hoa Hà Duong and Isabelle Demeure. Handling the M in MANet : an Algorithm to Identify Stable Groups of Peers Using Cross-layering Information accepté à Mobicase 2010

Hoa Hà Duong and Isabelle Demeure. Proactive data replication using semantic information within mobility groups in manet. In *Mobile Wireless Middleware, Operating Systems, and Applications, Second International Conference, Mobilware 2009, Berlin, Germany, April 28-29, 2009, Proceedings*, volume 7 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*. Springer, 2009.

Hoa Hà Duong and Isabelle Demeure. A nomadic wiki for mobile ad hoc networks. In *CTS '09 : Proceedings of the 2009 International Symposium on Collaborative Technologies and Systems*, pages 528–535, Washington, DC, USA, 2009. IEEE Computer Society.

Juan A. Botia, Antonio F. Gomez-Skarmeta, Hoa Hà Duong, and Isabelle Demeure. A context-aware data sharing service over manet to enable spontaneous collaboration. *WETICE '08 : Proceedings of the 16th IEEE International Workshops on Enabling Technologies, IEEE International Workshops on*, 0 :159–164, 2008.

Hoa Hà Duong and Isabelle Demeure. Data sharing over mobile ad hoc networks. In *CDUR '08*, pages 1–6, New York, NY, USA, 2008. ACM.

Marcel Arrufat, Hoa Dung Hà Duong, Christian Melchiorre, Eike Michael Meyer, Ignacio Nieto, Patrizio Pelliccione, and Frederique Tastet-Cherel. Popeye : A simple and reliable collaborative working environment over mobile ad-hoc networks. *International Conference on Collaborative Computing : Networking, Applications and Worksharing*, 0 :399–407, 2007.

Hoa Dung Hà Duong, Christian Melchiorre, Eike Michael Meyer, Ignacio Nieto, Gerard Paris, Patrizio Pelliccione, and Frederique Tastet-Cherel. A software architecture for reliable collaborative working environments. In *WETICE '07 : Proceedings of the 16th IEEE International Workshops on Enabling Technologies : Infrastructure for Collaborative Enterprises*, pages 176–177, Washington, DC, USA, 2007. IEEE Computer Society.

Bibliographie

[1] Flickr. http ://www.flickr.com/.

[2] Pictures shared under creative commons attribution 3.0 unported, found at. http ://en.wikipedia.org/wiki/One_Laptop_per_Child.

[3] Wikicreole. http ://www.wikicreole.org/.

[4] Wikimedia. http ://download.wikimedia.org/backup-index.html.

[5] NFS : Network file system version 3 protocol specification, June 25 1994.

[6] Napster. http ://www.napster.com/, 2006.

[7] Beongku An and Symeon Papavassiliou. A mobility-based clustering approach to support mobility management and multicast routing in mobile ad-hoc wireless networks. *Int. J. Netw. Manag.*, 11(6) :387–395, 2001.

[8] Emil Sit Andreas, Andreas Haeberlen, Frank Dabek, Byung gon Chun, Hakim Weatherspoon, Robert Morris, M. Frans Kaashoek, and John Kubiatowicz. Proactive replication for data durability. In *In Proceedings of the 5th Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[9] R. Baldoni, A. Virgillito, and R. Petrassi. A distributed mutual exclusion algorithm for mobile ad-hoc networks. *Computers and Communications, 2002. Proceedings. ISCC 2002. Seventh International Symposium on*, pages 539–544, 2002.

[10] 1979-João Pedro Faria Mendonça Barreto. Haddock-FS : A distributed file system for mobile ad-hoc networks. Master's thesis, Instituições portuguesas – UTL-Universidade Técnica de Lisboa – IST-Instituto Superior Técnico – -Departamento de Engenharia Informática, 2004.

[11] Prithwish Basu, Naved Khan, and Thomas D.C. Little. A mobility based metric for clustering in mobile ad hoc networks. In *In International Workshop on Wireless Networks and Mobile Computing (WNMC2001*, pages 413–418, 2001.

[12] B.N. Bershad, M.J. Zekauskas, and W.A. Sawdon. The midway distributed shared memory system. *Compcon Spring '93, Digest of Papers.*, pages 528–537, 22-26 Feb 1993.

[13] Juan A. Botia, Antonio F. Gomez-Skarmeta, Hoa Ha Duong, and Isabelle Demeure. A context-sware data sharing service over manet to enable spontaneous collaboration. *Enabling Technologies, IEEE International Workshops on*, 0 :159–164, 2008.

[14] M. Boulkenafed and V. Issarny. Adhocfs : sharing files in wlans. *Network Computing and Applications, 2003. NCA 2003. Second IEEE International Symposium on*, pages 156–163, 16-18 April 2003.

[15] John S. Breese, David Heckerman, and Carl Kadie. Empirical analysis of predictive algorithms for collaborative filtering. In Gregory F. Cooper and Serafín Moral, editors, *Proceedings of the 14th Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 43–52, San Francisco, July 24–26 1998. Morgan Kaufmann.

[16] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis : A highly-scalable multi-user peer-to-peer file system. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 1173–1182. Springer, 2005.

[17] Gérôme Canals, Pascal Molli, Julien Maire, Stéphane Laurière, Esther Pacitti, and Mounir Tlili. Xwiki concerto : A p2p wiki system supporting disconnected work. In *CDVE '08 : Proceedings of the 5th international conference on Cooperative Design, Visualization, and Engineering*, pages 98–106, Berlin, Heidelberg, 2008. Springer-Verlag.

[18] Jiannong Cao, Yang Zhang, Li Xie, and Guohong Cao. Consistency of cooperative caching in mobile peer-to-peer systems over manet. In *ICDCSW '05 : Proceedings of the Third International Workshop on Mobile Distributed Computing (MDC) (ICDCSW'05)*, pages 573–579, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Ian D. Chakeres and Joseph P. Macker. Mobile ad hoc networking and the ietf. *SIGMOBILE Mob. Comput. Commun. Rev.*, 11(4) :80–82, 2007.

[20] Narottam Chand, R. C. Joshi, and Manoj Misra. Cooperative caching in mobile ad hoc networks based on data utility. *Mob. Inf. Syst.*, 3(1) :19–37, 2007.

[21] Yu Chen and Jennifer L. Welch. Self-stabilizing mutual exclusion using tokens in mobile ad hoc networks. In *DIALM '02 : Proceedings of the 6th international workshop on Discrete algorithms and methods for mobile computing and communications*, pages 34–42, New York, NY, USA, 2002. ACM.

[22] T. Clausen and P. Jacquet. Optimized Link State Routing Protocol (OLSR). RFC 3626 (Experimental), October 2003.

[23] S. Corson and J. Macker. Mobile Ad hoc Networking (MANET) : Routing Protocol Performance Issues and Evaluation Considerations. RFC 2501 (Informational), January 1999.

[24] Fabio De Rosa, Alessio Malizia, and Massimo Mecella. Disconnection prediction in mobile ad hoc networks for supporting cooperative work. *IEEE Pervasive Computing*, 4(3) :62–70, 2005.

[25] Abdelouahid Derhab, Nadjib Badache, and Abdelmadjid Bouabdallah. A partition prediction algorithm for service replication in mobile ad hoc networks. *Wireless on Demand Network Systems and Service, International Conference on*, 0 :236–245, 2005.

[26] Sandor Dornbush and Anupam Joshi. Streetsmart traffic : Discovering and disseminating automobile congestion using VANET's. In *VTC Spring*, pages 11–15. IEEE, 2007.

[27] Yu Du and Sandeep K. S. Gupta. Coop - a cooperative caching service in manets. In *ICAS-ICNS '05 : Proceedings of the Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services*, page 58, Washington, DC, USA, 2005. IEEE Computer Society.

[28] Alessandro Duminuco, Ernst Biersack, and Taoufik En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *CoNEXT '07 : Proceedings of the 2007 ACM CoNEXT conference*, pages 1–12, New York, NY, USA, 2007. ACM.

[29] Abe Fettig and Glyph Lefkowitz. *Twisted network programming essentials.* O'Reilly & Associates, Inc., pub-ORA :adr, 2006.

[30] Annie Gentes, Aude Guyot-Mbodji, and Isabelle Demeure. Gaming on the move : urban experience as a new paradigm for mobile pervasive game design. In *MindTrek '08 : Proceedings of the 12th international conference on Entertainment and media in the ubiquitous era*, pages 23–28, New York, NY, USA, 2008. ACM.

[31] Nabil Ghanem, Selma Boumerdassi, and Éric Renault. New energy saving mechanisms for mobile ad-hoc networks using olsr. In *PE-WASUN '05 : Proceedings of the 2nd ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, pages 273–274, New York, NY, USA, 2005. ACM.

[32] Jim Gray and Pat Helland. The dangers of replication and a solution. In *In Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 173–182, 1996.

[33] Leanne Guy, Peter Kunszt, Erwin Laure, Heinz Stockinger, and Kurt Stockinger. Replica management in data grids. Technical report, Global Grid Forum Informational Document, GGF5, 2002.

[34] Takahiro Hara. Effective replica allocation in ad hoc networks for improving data accessibility. In *INFOCOM*, pages 1568–1576, 2001.

[35] Takahiro Hara. Replica allocation methods in ad hoc networks with data update. *Mob. Netw. Appl.*, 8(4) :343–354, 2003.

[36] Takahiro Hara and Sanjay Kumar Madria. Dynamic data replication using aperiodic updates in mobile adhoc networks. In *DASFAA*, pages 869–881, 2004.

[37] Takahiro Hara, Norishige Murakami, and Shojiro Nishio. Replica allocation for correlated data items in ad hoc sensor networks. *SIGMOD Rec.*, 33(1) :38–43, 2004.

[38] Michal Hauspie, David Simplot, and Jean Carle. Partition detection in mobile ad hoc networks, May 20 2003.

[39] Hideki Hayashi, Takahiro Hara, and Shojiro Nishio. Cache invalidation for updated data in ad hoc networks. In *CoopIS/DOA/ODBASE*, pages 516–535, 2003.

[40] C.L. Hedrick. Routing Information Protocol. RFC 1058 (Historic), June 1988. Updated by RFCs 1388, 1723.

[41] Xiaoyan Hong, Mario Gerla, Guangyu Pei, and Ching-Chuan Chiang. A group mobility model for ad hoc wireless networks. In *MSWiM '99 : Proceedings of the 2nd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 53–60, New York, NY, USA, 1999. ACM.

[42] Juan Pablo Hourcade, Daiana Beitler, Fernando Cormenzana, and Pablo Flores. Early olpc experiences in a rural uruguayan school. In Mary Czerwinski, Arnold M. Lund, and Desney S. Tan, editors, *Extended Abstracts Proceedings of the 2008 Conference on Human Factors in Computing Systems, CHI 2008, Florence, Italy, April 5-10, 2008*, pages 2503–2512. ACM, 2008.

[43] Jiun-Long Huang, Ming-Syan Chen, and Wen-Chih Peng. Exploring group mobility for replica data allocation in a mobile environment. In *CIKM '03 : Proceedings of the twelfth international conference on Information and knowledge management*, pages 161–168, New York, NY, USA, 2003. ACM.

[44] Yu Huang, Jiannong Cao, and Beihong Jin. A predictive approach to achieving consistency in cooperative caching in manet. In *InfoScale '06 : Proceedings of the 1st international conference on Scalable information systems*, page 50, New York, NY, USA, 2006. ACM.

[45] Yu Huang, Beihong Jin, Jiannong Cao, Guangzhong Sun, and Yulin Feng. A selective push algorithm for cooperative cache consistency maintenance over manets. In Tei-Wei Kuo, Edwin Hsing-Mean Sha, Minyi Guo, Laurence Tianruo Yang, and Zili Shao, editors, *EUC*, volume 4808 of *Lecture Notes in Computer Science*, pages 650–660. Springer, 2007.

[46] Aleksandr Huhtonen. Comparing AODV and OLSR routing protocols, May 27 2004.

[47] Michel Jaczynski and Brigitte Trousse. Www assisted browsing by reusing past navigations of a group of users. In *In Proceedings of EWCBR-98, European Workshop on Case-Based Reasoning*, pages 160–171. Springer Verlag, 1998.

[48] Zheng Jing, Wang Yijie, Lu Xicheng, and Yang Kan. A dynamic adaptive replica allocation algorithm in mobile ad hoc networks. In *PERCOMW '04 : Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, page 65, Washington, DC, USA, 2004. IEEE Computer Society.

[49] D. Johnson, Y. Hu, and D. Maltz. The Dynamic Source Routing Protocol (DSR) for Mobile Ad Hoc Networks for IPv4. RFC 4728 (Experimental), February 2007.

[50] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking : Design tradeoffs and early experiences with zebranet. In *ASPLOS-X : Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 96–107, New York, NY, USA, 2002. ACM Press.

[51] P. Keleher, A.L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. *Computer Architecture, 1992. Proceedings., The 19th Annual International Symposium on*, pages 13–21, 1992.

[52] Amir R. Khakpour and Isabelle M. Demeure. Chapar : A cross-layer overlay event system for manets. In Jean-Marie Bonnin, Carlo Giannelli, and Thomas Magedanz, editors, *MOBILWARE*, volume 7 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 325–339. Springer, 2009.

[53] Sanjeev Khanna, Keshav Kunal, and Benjamin C. Pierce. A formal investigation of diff3. In Arvind and Prasad, editors, *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, December 2007.

[54] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, US, 1991. http ://www.cs.cmu.edu/afs/cs/project/coda/Web/docs-coda.html.

[55] Er Klemm, Christoph Lindemann, and Oliver P. Waldhorst. A special-purpose peer-to-peer file sharing system for mobile ad hoc networks, February 28 2003.

[56] John T. Kohl and B. Clifford Neuman. The Kerberos network authentication service (V5). Internet Request for Comment RFC 1510, Internet Engineering Task Force, 1993.

[57] Robin Kravets and P. Krishnan. Power management techniques for mobile communication. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98)*, pages 157–168, New York, October 25–30 1998. ACM Press.

[58] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7) :558–565, 1978.

[59] Mihai Letia, Nuno M. Preguiça, and Marc Shapiro. Crdts : Consistency without concurrency control. *CoRR*, abs/0907.0929, 2009.

[60] Bo Leuf and Ward Cunningham. *The Wiki way : quick collaboration on the Web.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[61] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 229–239, New York, NY, 1986. ACM Press.

[62] Wenzhong Li, Edward Chan, Yilin Wang, and Daoxu Chen. Cache invalidation strategies for mobile ad hoc networks. In *ICPP '07 : Proceedings of the 2007 International Conference on Parallel Processing*, page 57, Washington, DC, USA, 2007. IEEE Computer Society.

[63] Henry Lieberman. Letizia : An agent that assists web browsing. In *INTERNATIONAL JOINT CONFERENCE ON ARTIFICIAL INTELLIGENCE*, pages 924–929, 1995.

[64] K. Lougheed and Y. Rekhter. Border Gateway Protocol (BGP). RFC 1163 (Historic), June 1990. Obsoleted by RFC 1267.

[65] Eivind Are Lundqvist. Implementing the optimized link state routing protocol for J-sim, 2006.

[66] Joseph Macker, Ian Downard, Justin Dean, and Brian Adamson. Evaluation of distributed cover set algorithms in mobile ad hoc network for simplified multicast forwarding. *SIGMOBILE Mob. Comput. Commun. Rev.*, 11(3) :1–11, 2007.

[67] Mamoru Maekawa. An algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2) :145–159, 1985.

[68] Navneet Malpani, Yu Chen, Nitin H. Vaidya, and Jennifer L. Welch. Distributed token circulation in mobile ad hoc networks. *IEEE Transactions on Mobile Computing*, 4(2) :154–165, 2005.

[69] Cecilia Mascolo, Licia Capra, Stefanos Zachariadis, and Wolfgang Emmerich. Xmiddle : A data-sharing middleware for mobile computing. *Wirel. Pers. Commun.*, 21(1) :77–103, 2002.

[70] Adam Mathes. Folksonomies - cooperative classification and communication through shared metadata. December 2004.

[71] Friedmann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editors, *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Amsterdam, 1989. Elsevier Science Publishers.

[72] Masaaki Mizuno, Michel Raynal, and James Z. Zhou. Sequential consistency in distributed systems. In Kenneth P. Birman, Friedemann Mattern, and Andre Schiper, editors, *Theory and Practice in Distributed Systems*, volume 938 of *Lecture Notes in Computer Science (LNCS)*, pages 224–241. Springer-Verlag (Heidelberg), Dagstuhl Castle, Germany, September 1994, Selected Paper 1995.

[73] M. Moallemi, M.H.Y. Moghaddam, and M. Naghibzadeh. A fault-tolerant mutual exclusion resource reservation protocol for clustered mobile ad hoc networks. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, 2 :528–533, July 30 2007-Aug. 1 2007.

[74] Joseph C. Morris. Distriwiki : a distributed peer-to-peer wiki network. In *WikiSym '07 : Proceedings of the 2007 international symposium on Wikis*, pages 69–74, New York, NY, USA, 2007. ACM.

[75] David Mosberger. Memory consistency models. *SIGOPS Oper. Syst. Rev.*, 27(1) :18–26, 1993.

[76] J. Moy. OSPF Protocol Analysis. RFC 1245 (Informational), July 1991.

[77] Mohamed Naimi, Michel Trehel, and André Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *J. Parallel Distrib. Comput.*, 34(1) :1–13, 1996.

[78] Gérald Oster. Réplication optimiste et cohérence des données dans les environnements collaboratifs répartis, November 03 2005.

[79] Prasanna Padmanabhan, Le Gruenwald, Anita Vallur, and Mohammed Atiquzzaman. A survey of data replication techniques for mobile ad hoc network databases. *The VLDB Journal*, 17(5) :1143–1164, 2008.

[80] Maria Papadopouli and Henning Schulzrinne. Design and implementation of a peer-to-peer data dissemination and prefetching tool for mobile users, May 11 2003.

[81] Guilhem Paroux, lsabelle Demeure, and Laurent Reynaud. A power-aware middleware for mobile ad-hoc networks. In *NOTERE '08 : Proceedings of the 8th international conference on New technologies in distributed systems*, pages 1–7, New York, NY, USA, 2008. ACM.

[82] F. Pedone, M. Wiesmann, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *20th International Conference on Distributed Computing Systems (ICDCS '00)*, pages 464–474, Washington - Brussels - Tokyo, April 2000. IEEE.

[83] C. Perkins, E. Belding-Royer, and S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. RFC 3561 (Experimental), July 2003.

[84] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, 2 edition, September 2008.

[85] J. Postel. RFC 1591 : Domain name system structure and delegation, March 1994. Status : INFORMATIONAL.

[86] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Le ?ia. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 395–403, Montreal, Québec Canada, 2009. IEEE Computer Society.

[87] Jiten Rama and Judith Bishop. A survey and comparison of cscw groupware applications. In *SAICSIT '06 : Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 198–205, , Republic of South Africa, 2006. South African Institute for Computer Scientists and Information Technologists.

[88] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1) :61–77, 1989.

[89] Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Oper. Syst. Rev.*, 25(2) :47–50, 1991.

[90] Michel Raynal and Mukesh Singhal. Logical time : Capturing causality in distributed systems. *Computer*, 29(2) :49–56, 1996.

[91] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Commun. ACM*, 24(1) :9–17, 1981.

[92] Christian P. Robert and George Casella. Monte carlo statistical methods, 1998.

[93] Antony I. T. Rowstron and Peter Druschel. Pastry : Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01 : Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350, London, UK, 2001. Springer-Verlag.

[94] Françoise Sailhan and Valérie Issarny. Energy-aware web caching for mobile terminals. In *ICDCSW '02 : Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 820–825, Washington, DC, USA, 2002. IEEE Computer Society.

[95] Aman Singla, Umakishore Ramachandran, and Jessica Hodgins. Temporal notions of synchronization and consistency in beehive. In *In Proc. of the 9th Annual ACM Symp. on Parallel Algorithms and Architectures*, pages 211–220, 1997.

[96] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4) :149–160, 2001.

[97] William Su, Sung-Ju Lee, and Mario Gerla. Mobility prediction and routing in ad hoc wireless networks. *Int. J. Netw. Manag.*, 11(1) :3–30, 2001.

[98] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 3(4) :344–349, 1985.

[99] Andrew S. Tanenbaum. *Modern operating systems*. Prentice–Hall, Englewood Cliffs, NJ, second edition, 2001.

[100] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15), Copper Mountain Resort, Colorado*, 1995.

[101] L. H. Ungar and D. P. Foster. Clustering methods for collaborative filtering. In *Proceedings of the Workshop on Recommendation Systems at the Fifteenth National Conference on Artificial Intelligence*, Menlo Park, CA, 1998. AAAI Press.

[102] Jennifer E. Walter, Jennifer L. Welch, and Nitin H. Vaidya. A mutual exclusion algorithm for ad hoc mobile networks. *Wireless Networks*, 7(6) :585–600, 2001.

[103] Karen Wang and Baochun Li. Efficient and guaranteed service coverage in partitionable mobile ad-hoc networks. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Society (INFOCOM-02)*, volume 2 of *Proceedings IEEE INFOCOM 2002*, pages 1089–1098, Piscataway, NJ, USA, June 23–27 2002. IEEE Computer Society.

[104] Karen H. Wang and Baochun Li. Group mobility and partition prediction in wireless ad-hoc networks, May 29 2002.

[105] Elias Weingartner, Hendrik vom Lehn, and Klaus Wehrle. A performance comparison of recent network simulators. In *Proceedings of the IEEE International Conference on Communications 2009 (ICC 2009)*, Dresden, Germany, 2009. IEEE.

[106] Stéphane Weiss, Pascal Urso, and Pascal Molli. Wooki : A P2P wiki-based collaborative writing tool. In Boualem Benatallah, Fabio Casati, Dimitrios Georgakopoulos, Claudio Bartolini, Wasim Sadiq, and Claude Godart, editors, *WISE*, volume 4831 of *Lecture Notes in Computer Science*, pages 503–512. Springer, 2007.

[107] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[108] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. *Distributed Computing Systems, International Conference on*, 0 :464, 2000.

[109] Brendon J. Wilson. *JXTA*. New Riders, June 2002.

[110] Weigang Wu, Jiannong Cao, and Michel Raynal. A dual-token-based fault tolerant mutual exclusion algorithm for manets. In *MSN*, pages 572–583, 2007.

[111] Weigang Wu, Jiannong Cao, and Jin Yang. A scalable mutual exclusion algorithm for mobile ad hoc networks. *Computer Communications and Networks, 2005. ICCCN 2005. Proceedings. 14th International Conference on*, pages 165–170, Oct. 2005.

[112] Liangzhong Yin and Guohong Cao. Supporting cooperative caching in ad hoc networks. *IEEE Transactions on Mobile Computing*, 5(1) :77–89, 2006.

[113] Hao Yu, Hossam Hassanein, and Patrick Martin. Cluster-based replication for large-scale mobile ad-hoc networks. In *International Conference on Wireless Networks, Communications in Computing*, pages 552–557, June 2005.

[114] Jing Zheng, Jinshu Su, and Xicheng Lu. A clustering-based data replication algorithm in mobile ad hoc networks for improving data availability. In *Proceedings of 2nd International Symposium on Parallel and Distributed Processing and Applications (ISPA 2004)*, pages 399–409, 2004.

[115] Jing Zheng, Jinshu Su, Kan Yang, and Yijie Wang. Stable neighbor based adaptive replica allocation in mobile ad hoc networks. In *International Conference on Computational Science*, pages 373–380, 2004.