Modélisation et analyse de la sécurité dans un système de stockage pair-à-pair

Samira Chaou

Modélisation et analyse de la sécurité dans un système de stockage pair-à-pair

Samira Chaou

Laboratoire IBISC , Université d'Evry
UbiStorage SA., Amiens

Thèse à défendre en janvier 2013

Directeurs :	Pr. Franck Pommereau	IBISC, Université d'Evry
	Dr. Gil Utard	UbiStorage SA., Amiens
Rapporteurs :	Pr. Fabrice Kordon	LIP6, Université Pierre et Marie Curie
	Pr. Maryline Laurent	SAMOVAR, Télécom Sud Paris
Examinateurs :	Pr. Gaétan Hains	LACL, Université Paris Est Créteil
	Pr. Hanna Klaudel	IBISC, Université d'Evry

Remerciements

Je tiens tout d'abord à remercier les directeurs de cette thèse, M. Franck Pommerau et Mr Gil Utard, pour m'avoir fait confiance et de m'avoir guidée, encouragée, conseillée, et sans qui je ne serai pas là où j'en suis aujourd'hui.

Je remercie les rapporteurs de cette thèse M. Fabrice Kordon et Mme. Maryline Laurent pour l'intêret qu'ils ont porté à ma thèse, et je remercie les membres de jury qui ont accépter de juger ce modeste travail M. Gaétan Hains et Mme. Hanna Klaudel.

Je tiens à remercier tous ceux qui m'ont aidée durant ces trois années dans mon travail, les membres de la société UbiStorage (M. Randriamaro , M. Le ...), et tous les membres du laboratoire IBISC qui m'ont adoptée et intégrée dans leur groupe et avec qui j'ai partagé mes moments de doute et de joie.

Mes remerciements vont aussi à ma soeur qui a été à mes côté durant ces trois années de thèse, et qui a su gérer et supporter mes sauts d'humeur et qui a partagé mes moments de joie. Je n'oublie pas mes frères qui ont toujours eu confiance en moi et m'ont toujours encouragée à toujours aller plus loin. Je remercie tous mes amis qui ont toujours été là pour moi (la liste est langue ils se reconnaiteront).

Mes pensées vont à mes prents sans qui je ne serai pas ici aujourd'hui, ils ont été mon moteur et ma raison de tenir. Mon seul regret est qu'ils ne soient pas là pour partager la joie de l'aboutissement de tous leurs efforts et leurs sacrifices.

Enfin je tiens à remercier tous les gens qui ont contribué de près ou de loin à l'aboutissement de ce travail et qui ont permis que ce jour arrive.

Sommaire

Chapitre 1

Introduction

Un système pair-à-pair est constitué de plusieurs pairs pouvant communiquer entre eux par l'intermédiaire d'un réseau qui les relie. Dans un réseau pair-à-pair, chaque pair joue le rôle de client et le rôle de serveur en même temps. L'architecture pair-à-pair présente un avantage par rapport à l'architecture client-serveur car elle confère une meilleure résistance aux pannes et une plus grande disponibilité (aucun pair n'est indispensable au fonctionnement du réseau). Les systèmes pair-à-pair ont été conçus pour partager des ressources sur internet. Ils ont connu alors un grand succès grâce à la capacité de passer à l'échelle, et leurs utilisations multiples (partage de CPU, diffusion de news ou encore la publication de fichiers, ...). L'essort des systèmes pair-à-pair (*peer-to-peer*, ou simplement *P2P*) a mené au développement de systèmes dédiés au partage d'espace disque [10]. Le but de ces systèmes est de fournir une solution de stockage distribuée dont les principaux enjeux sont la pérennité et la disponibilité des données. Afin d'assurer la pérennité des données, les systèmes de stockage P2P utilisent la redondance des données et un mécanisme de reconstruction de données en fonction des états des pairs du réseau. Les mécanismes de redondance peuvent être soit la réplication simple, soit des codes auto-correcteurs. Ces derniers sont plus efficaces en terme d'utilisation d'espace disque, en effet les codes correcteurs permettent de fragmenter les données et de les disperser dans le réseau, contrairement à la duplication qui reproduit la même donnée sur plusieurs pairs. Un des système de stockage pair-à-pair est le système de stockage de données d'Ubiquitous Storage (UbiStorage).

UbiStorage est une société qui développe et commercialise une solution de stockage de données basée sur la technologie pair-à-pair qui permet aux utilisateurs de stocker leurs données dans le système et de les récupérer en toute sécurité. Ce système est complétement distribué et utilise un protocole de routage basé sur une table de hachage distribuée (DHT) qui permet aux pairs du réseau d'échanger de manière décentralisée. UbiStorage met à la disposition de ses clients un boîtier prêt à l'emploi, sur lequel est installé le logiciel de stockage pair-à-pair qui fonctionne dans un environnement Linux. Une des principales préoccupations d'UbiStorage est d'assurer la disponibilité et la pérennité des données de leur clientèle. La sécurité du protocole de routage est prouvée de manière qualitative en utilisant la vérification formelle [46].

L'utilisation de la modélisation formelle associée au model-checking automatique a permis de découvrir des attaques possibles sur le système en utilisant la technique de rejeu, des solutions ont été proposées (dont l'utilisation du chiffrement) et leurs efficacité prouvée en utilisant la même méthode. Il a été prouvé qu'un attaquant extérieur ne pourra jamais apprendre quoi que se soit sur les données stockées dans le système UbiStorage et ce même s'il arrive à mettre la main sur des fragments de données.

Dans ce travail, nous allons compléter les premiers travaux et vérifier la disponibilité des données qui est directement liée aux pairs du système qui les stockent. Notre objectif est l'analyse de la résistance et la robustesse du système et de proposer des solutions le cas échéant (chapitre : [**?**]).

L'objectif de notre thèse est d'analyser la robustesse du système en la présence d'attaquants internes et de proposer une solution pour assurer la disponibilité des données. La méthodologie suivie est la suivante. Dans un premier temps nous avons listé tous les comportements malveillants qu'un pair du système peut adopter. Cette liste a été affinée et nous n'avons retenu que les attaques qui avaient des conséquences sur les données (qui causaient des pertes de données). Pour l'analyse de la robustesse du système aux attaques retenues nous avons opté pour deux méthodes : la vérification formelle et la simulation. La simulation nous a permis d'avoir des résultats quantitatifs et de manière plus rapide ce qui nous a donné une vision globale sur les effets des attaques sur les données. Ces comportements malveillants ont été implémentés dans un premier temps dans le système de stockage UbiStorage et en utilisant la simulation on a obtenu des résultats qui ont confirmé nos craintes concernant la perte des données. Les résultats obtenus par la simulation nous ont conduits à proposer une solution pour pallier la perte de données, et on a opté pour un système de détection des comportements malveillants basé sur la réputation des pairs. Le système de réputation proposé est basé sur deux niveaux de notations un niveau pour noter les transactions entre les pairs et un deuxième niveau pour évaluer la confiance en les notes accordées aux pairs. Et enfin l'efficacité de la solution proposée est vérifiée en utilisant la vérification formelle ainsi que la simulation. Le document est organisé de la façon suivante :

La suite du **chapitre 1** est une introduction pour présenter le contexte du travail et un rappel des définitions de bases des réseaux pair-à-pair ainsi que les propriétés de sécurité.

Le **chapitre 2** propose une vue d'ensemble des systèmes pair-à-pair et des systèmes pair-à-pair dédiés au stockage de données. On y propose aussi un état de l'art sur les différents modèles de confiance basés sur la réputation des réseaux pair-à-pair.

Le **chapitre 3** est une description du système de stockage UbiStorage et

son principe de fonctionnement.

Le **chapitre 4** présente notre contribution, on y fait une description des différents comportements malveillants qui ont été considérés ainsi que la solution proposée pour éviter la perte de données.

Le **chapitre 5** décrit le formalisme utilisé pour la modélisation du système UbiStorage et la technique utilisée pour la vérification formelle.

Le **chapitre 6** aborde l'architecture de la simulation ainsi que les résultats quantitatifs obtenus en utilisant la simulation.

Enfin nous concluons par une synthèse des principales contributions et la présentation des perspectives de recherche dans les domaines abordés tout au long de ce document.

1.1 Les systèmes pair-à-pair

Les systèmes pair-à-pairs au début de leur création ont été considérés comme des systèmes distribués proches du modèle client-serveur, mais chaque nœud peut être client et serveur en même temps. Depuis,la vision des systèmes pair-à-pair a évoluée ; et un système pair-à-pair ou *peer-to-peer* est vu comme un système d'échange de ressources entre utilisateurs. Les ressources partagées peuvent être :

- Le contenu : les fichiers présents sur la machine
- La bande-passante : messagerie-téléphonie, streaming audio-vidéo
- La puissance de calcul ou la mémoire : calculs scientifiques
- L'espace disque : sauvegarde croisée

Les réseaux pair-à-pair ont été popularisés par les systèmes de partage de fichiers comme : Gnutella , Napster et Freenet. Depuis les systèmes pair-à-pair sont devenus plus connus et plus utilisés dans plusieurs domaines :

- partage de fichiers
- communication
- calcul distribué
- travail collaboratif

Pour mieux comprendre l'architecture et le fonctionnement des systèmes pair-à-pair, nous allons d'abord commencer par présenter l'ancêtre des systèmes répartis qui est le modèle client-serveur et ensuite on verra dans le détails les systèmes pair-à-pair et leurs applications.

1.1.1 Modèle client-serveur

Le modèle client-serveur est un réseau d'ordinateurs où chaque client (processeur distant) envoie des requêtes et reçoit des services d'un serveur centralisé (un ordinateur hôte). Les services tels que : l'échange de mail, le web et l'accès aux bases de données ...etc, sont basés sur un modèle client-serveur.

Le modèle client-serveur présente certains avantages qui sont :

- des ressources centralisées : étant donné que le serveur est au centre du réseau, il peut gérer des ressources communes à tous les utilisateurs, comme par exemple une base de données centralisée, afin d'éviter les problèmes de redondance et de contradiction
- une administration au niveau serveur : les clients ayant peu d'importance dans ce modèle, ils ont moins besoin d'être administrés
- un réseau évolutif : grâce à cette architecture il est possible de supprimer ou rajouter des clients sans perturber le fonctionnement du réseau et sans modification majeure

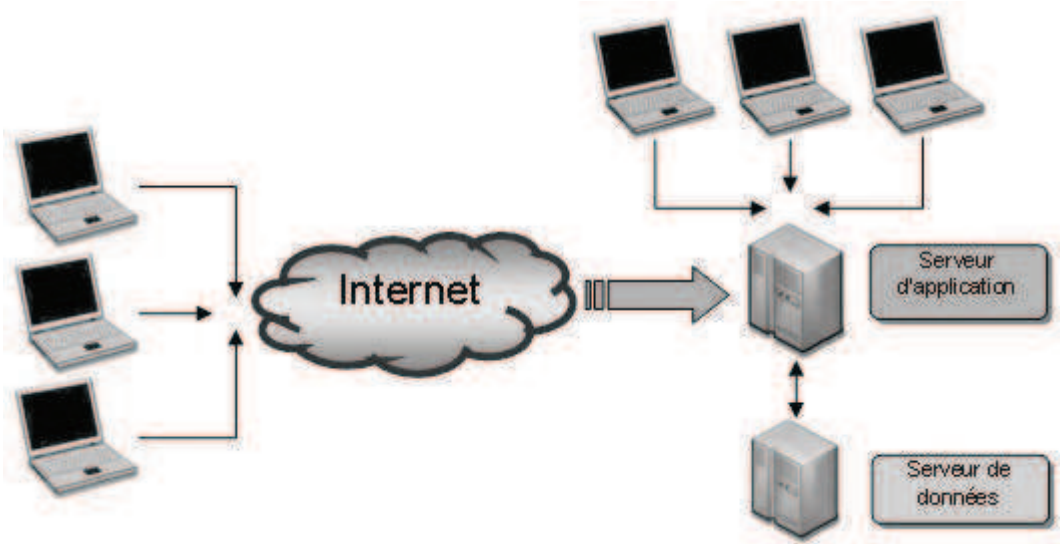

FIGURE 1.1 – Modèle Client/serveur

L'architecture client-serveur a tout de même quelques lacunes parmi lesquelles :
- un coût élevé dû à la technicité du serveur
- un SPOF (single point of faillure) : le serveur est le seul maillon faible du réseau client-serveur, étant donné que tout le réseau est architecturé autour de lui, donc afin d'assurer la continuité du service en cas de panne il faut dupliquer le serveur pour éviter le perte des données (raid, architecture à haute disponibilité...Etc) ce qui induit de grand coûts.

1.1.2 Modèle pair-à-pair

Depuis la création desréseaux pair-à-pair, ils ont connu des évolutions sur tout les aspects, notamment leur architecture. Nous allons vous présenter les trois principales architectures des réseaux P2P.

Architectures centralisées

Le réseau le plus connu de ce modèle de P2P est Napster. Cette architecture se compose d'un unique serveur, dont le but est de recenser les fichiers proposés par les différents pairs du système (Client) . Contrairement au modèle client-serveur, ce serveur centralisé ne dispose pas de fichiers, mais uniquement de méta-informations décrivant les données et leurs propriétaires ; celles sur les données (nom, taille, ...), et celles sur le client (nom utilisé, IP, nombre de fichiers, type de connexion, ...). Du côté du client, rien

de plus simple : une fois connecté grâce au logiciel spécifique, il peut faire des recherches comme avec un moteur classique. On obtient alors une liste d'utilisateurs disposant de la ressource désirée. Il suffit alors de cliquer sur le bon lien pour commencer le téléchargement. Cette étape est complètement indépendante du serveur, et représente la seule partie P2P du logiciel. Les avantages de cette architecture sont :

- le confort de l'utilisation dû à l'unicité du serveur
- la facilité de la recherche

et les inconvénients sont :

- le serveur constitue le maillon faible du réseau
- sensible aux attaques comme l'attaque de déni de service DoS
- ce système n'assure pas l'anonymat puisque chaque utilisateur est identifié par le serveur.

Architectures centralisées à serveurs multiples

Afin de pallier les déficiences introduites par un unique serveur, une amélioration consiste à remplacer le serveur central par un réseau de serveurs. L'exemple le plus connu est sans doute le réseau eDonkey. Dans ce type de réseau, on diffère encore une fois très bien les serveurs des clients. Ce sont d'ailleurs deux programmes totalement distincts (serveur et client). Le serveur est, encore une fois, une machine puissante, (généralement) dédiée à faire tourner le programme du serveur (non pas que le programme soit lourd, mais les requêtes des utilisateurs sont nombreuses). Tout le problème d'un serveur est de se faire connaître du public (des clients et des autres serveurs).

Les serveurs sont ensuite en mesure de se connecter entre eux, en fonction de leur connaissance les uns des autres. On peut donc avoir plusieurs réseaux indépendants Lorsqu'un client lance le logiciel, il est alors obligé de choisir un serveur auquel se connecter. Pour cela, il doit connaître au moins un serveur. Des listes de serveurs sont donc disponibles sur le net. Tout le problème résidant alors dans le fait de choisir le bon... Pour ce faire, le logiciel client permet de disposer d'une description de chaque serveur avec son nom, son IP, une description (faite par le propriétaire du serveur), le ping (temps moyen pour l'envoi d'une requête et la réception de la réponse entre le client et le serveur), le nombre d'utilisateurs connectés, le nombre maximal d'utilisateurs et le nombre de fichiers partagés.

Architectures décentralisées

Dans ce type d'architecture il n y plus de serveur au sens commun : tout le monde est à la fois client et serveur. Cela pose principalement un

problème qui n'apparaît pas avec une architecture centralisée : comment connaître la topologie du réseau. Lorsqu'un client souhaite se connecter, il va donc être nécessaire d'envoyer un message broadcast afin de savoir quelles autres personnes du réseau sont actives. Seules ces personnes répondront au message broadcast. On est alors connecté au réseau. Afin de garder des informations cohérentes, un utilisateur n'est pas connecté directement à plus de 3 ou 4 nœuds, et la hauteur d'arbre est généralement de 7 (la distance entre le nœud source (racine) et le nœud final (feuille)). Lors des recherches, chaque nœud propage la requête à ses voisins (généralement 4), qui font de même. Le nombre de propagation est toutefois limité (généralement à 7), et il est possible de détecter les cycles grâce à l'identificateur des paquets.

1.1.3 Applications pair-à-pair

Le calcul distribué

Historiquement une des premières applications des technologies pair à pair est le calcul distribué, un exemple parlant aux mathématiciens est le calcul de matrices. En marge des exemples emblématiques tels Seti@home (ref-Seti) en astrophysique ou, plus récemment, le projet Decryptron(decrypton) en génétique, le calcul distribué se développe notamment sous la forme du Grid-computing. Ce dernier propose d'appliquer le même principe à d'autres échelles, comme celle d'une entreprise, demandeuse de gros calculs, souhaitant optimiser l'exploitation de son parc informatique, en distribuant des calculs sur l'ensemble des machines (en profitant des ressources non exploitées de chaque machine). Ce même principe peut être utilisé également dans le cadre du stockage massif de données ou encore dans le cadre du partage de temps processeur (chez IBM ou HP par exemple) [40].

Le partage de fichiers

Le partage de fichiers consiste à échanger des fichiers (musique, vidéos, logiciels, photos, etc.) entre différents utilisateurs connectés simultanément à Internet. Plus le fichier téléchargé est populaire, plus il y a d'internautes qui le télécharge et donc autant qui le redistribue ainsi il sera facile à télécharger. Il existe plusieurs réseaux P2P comme : BitTorrent Gnutella, Gnutella2, eDonkey2000, DirectConnect, DC++, Mute ... Chaque réseau a ses différences et possèdent ses propres fonctionnalités. Exemple, si un fichier est échangé via Gnutella il ne sera pas forcément disponible sur eDonkey. Pour partager votre fichier vous devez télécharger un logiciel comme BitTorrent ou Emule disponible sur de nombreux sites de téléchargements. Le logiciel vous per-

met de partager un dossier sur votre disque dur, faites donc attention à ce que vous mettez dans ce dossier pour éviter de diffuser des fichiers, photos ou vidéos qui n'étaient pas destinés à être partagés. Si au contraire vous recherchez un fichier en particulier, des moteurs de recherches intégrés au logiciel de P2P sont disponibles. Le P2P offre aussi la possibilité de faire du partage privé, en effet, si un utilisateur voulait partager des fichiers volumineux (par exemple des vidéos ou photos de vacances) avec ses proches, Il devait jusqu'à présent les télécharger sur un serveur dont l'hébergeur pouvait limiter l'espace et l'accés, mais il est assez difficile de restreindre l'accès aux photos à quelques utilisateurs uniquement. Ces fonctionnalités peuvent être assurées par des applications P2P telles que Qnext, GigaTrib ou TribalWeb.

La messagerie

D'autres applications telles que la messagerie instantanée (ICQ, AIM) ou encore le Netmeeting (visioconférence), proposé par Microsoft, profitent des avantages techniques du P2P. Cette application pourrait s'étendre aux logiciels de courrier électronique. La transmission des messages ne nécessitant plus le passage par un serveur, elle s'épargnerait des problèmes de stockage, de délais, et diverses défaillances techniques. Cette solution permet en outre de contrer plus aisément la propagation de virus et autres spams [5].

Une autre application connaissant depuis peu, un essor prometteur : la voix sur IP. En effet, certaines sociétés de télécommunications commencent à s'intéresser au P2P pour désengorger leurs serveurs téléphoniques [5], et l'adoption d'une technologie P2P par Skype, un service de voix sur IP (VoiP), qui s'est rapidement diffusé, marque peut-être le début d'une importante évolution du marché traditionnel de la téléphonie.

Au vu des exemples « classiques » mentionnés, on pourrait croire que les applications « légales » du P2P demeurent l'apanage des seuls chercheurs en informatique. Il n'en est rien. Des applications destinées au plus grand nombre se tournent aujourd'hui vers le P2P et ses solutions. L'une des plus aptes à se répandre est sans doute celle des jeux en ligne, et particulièrement les jeux massivement multi-joueurs [1, 51]. Une autre application, finalement techniquement assez proche, est celle de la diffusion de flux continus. Il peut s'agir par exemple du logiciel PeerCast, lancé en avril 2002, qui permet la diffusion de WebRadio. À ce jour, le logiciel permet à tout utilisateur d'écouter en flux continu de la musique, des émissions ou sa propre radio. Cet outil permet également aux internautes, même sur bas débit, d'écouter en streaming (flux continu) ces contenus. Ce type de solutions technologiques n'intéresse pas seulement les amateurs, mais également le secteur professionnel. En effet, des groupes comme Contiki (société de service) ou Akamai (four-

nisseur d'infrastructure de e-business) utilisent la technologie P2P pour aider les entreprises à diffuser leurs contenus multimédias et leurs présentations de ventes. Bien connus pour d'autres raisons, le transfert et le partage de fichiers en P2P ont été mis en œuvre par divers acteurs. Certaines banques, par exemple, utilisent déjà des technologies de partage de fichiers en P2P pour le transfert de données avec leurs succursales. À terme, des plateformes dédiées au partage et à la distribution d'informations propriétaires verront le jour dans les banques et les sociétés d'assurance. Autres cas, moins surprenant, Lindows (distribution Linux), poursuit une expérimentation consistant à offrir son logiciel via des réseaux P2P, à un tarif réduit de moitié par rapport au prix annoncé sur leur site Web ; l'idée étant de bénéficier de la réduction des coûts du travail en ligne, ainsi que de la capacité de cette technologie à permettre un nombre élevé des téléchargements simultanés [1, 51]. Le partage de fichier est également un élément important pour des institutions telles que les gouvernements ou les universités. Penn State et le MIT aux États-Unis, ainsi que la British Columbia's Fraser University au Canada ont commencé à développer des réseaux P2P afin d'assurer un partage de données, plus rapide et plus pratique. Le gouvernement des États-Unis utilise également un système décentralisé d'échange de données, et les agences fédérales américaines ont mentionné avoir commencé à utiliser une technologie P2P pour obtenir des statistiques et des informations en provenance de machines appartenant à plus d'une centaine d'agences gouvernementales différentes [25].

Routage et DHT

Les premiers systèmes de stockage avaient une architecture monolithique, *i.e*les différentes fonctionnalités du systèmes étaient interdépendantes. La tendance actuelle pour la conception des systèmes de stockage pair-à-pair est de se baser sur architecture à 3 niveaux. Cette architecture à trois niveaux a été introduite par CFS (Chrod File System [38]), elle est composée de :

- Niveau 2 : couche application, qui a pour objectif l'exploitation des données ;
- Niveau 1 : DHT, qui pour objectif de pérenniser les données ;
- Niveau 0 : Overlay, qui a pour objectif de localiser les données.

Le but de cette architecture est de fournir une abstraction de la table de hashage distribuée DHT (Distributed Hash Table). Le premier niveau propose un mécanisme de routage entre les pairs par un sur-réseau tolérant aux pannes, nommé *overlay*. Quelques exemples d'overlays : Chord [26, 21], Pastry [32, 44], Tapestry [14] (overlay utilisé parOceanStore) et CAN [29]. En général, le principe des overlays, est que chaque pair est identifié au sein d'un grand espace de noms. La génération des identificateurs et des clés est

faite par le moyen d'une fonction de hachage (la plus fréquemment utilisée est SHA-1)sur, respectivement, les identificateurs de fichiers et les fichiers à stocker identifiés par les clés. Cette fonction assure que les identifiants et les clés sont uniques et uniformément répartis dans l'espace des noms. Pour router les messages, chaque pair maintient une table de routage avec les identificateurs des autres pairs et leurs adresses IP. Le deuxième niveau implémente un dictionnaire distribué est redondant sur l'overlay. Ce dictionnaire est nommé DHT ; chaque entrée de ce dictionnaire est composé d'une clé et d'un objet associé (*e.g.*le fichier). L'objet est inséré dans la DHT qui le réplique afin d'assurer un certain niveau de tolérance aux pannes. Au sein de la DHT, il existe une fonction qui projette l'espace des clés du dictionnaire dans l'espace de noms des pairs, par exemple l'identité si les deux espaces de noms (celui de l'overlay et la DHT) sont identiques. Ainsi, chaque pair est responsable d'un ensemble de clés. Lors de l'insertion d'un nouvel objet dans la DHT, le pair qui le stockera est celui qui sera atteint par l'algorithme de routage en fonction de la clé projetée de l'objet. En général l'algorithme de routage désigne le pair qui a l'identificateur le plus proche de la projection de la clé. L'objet inséré est répliqué par la DHT pour faire face à la disparition de pairs. Typiquement, chaque pair maintient une liste des pairs qui ont un identifiant voisin (au sens de l'algorithme de routage) dans l'espace de noms, l'objet est alors dupliqué sur k pairs parmi ses pairs voisins (k paramètre du système). Pour interagir avec la DHT, l'utilisateur dispose d'un ensemble simple de primitives : search, put, get, parfois free. Le rôle de la DHT est donc de gérer les données stockées dans le réseau. Pour cela, elle s'appuie sur un overlay pour connaître et localiser les pairs du réseau. Finalement, le dernier niveau s'appuie sur les deux précédents (overlay et DHT) et est responsable de l'organisation logique des données.

1.2 Propriétés de sécurité

Dans tout système informatique certaines propriétés de sécurité doivent être assurées. Plusieurs niveaux de sécurité peuvent être mis en place selon la criticité des données et des processus.

1.2.1 Confidentialité

Une donnée est dite confidentielle lorsque elle est accessible que par les entités autorisées à y accéder. Plusieurs techniques sont utilisées pour assurer la confidentialité dans un système :

– Chiffrement

– Authentification
– Contrôle d'accès

Les techniques d'authentification et de chiffrement ainsi que les techniques utilisées pour la signature numérique sont basées sur des algorithmes de chiffrement. On distingue, d'une part, les algorithmes symétriques et, d'autre part, les algorithmes asymétriques. Dans le premier cas, la même clé (clé secrète) est utilisée pour chiffrer et déchiffrer les données. Dans le second cas, deux clés sont utilisées (clé secrète, clé publique), les données sont alors chiffrées avec une des clé et déchiffrées avec l'autre clé.

Chiffrement symétrique

Le chiffrement symétrique est la plus ancienne technique connue pour le chiffrement. Une clé secrète est utilisée pour chiffrer le message et pour le déchiffrer. L'avantage du chiffrement symétrique est qu'il est moins coûteux en temps, mais l'inconvénient de cette technique est le partage de la clé secrète. En effet, une clé est nécessaire pour chaque correspondant avec lequel on souhaite communiquer.

Chiffrement asymétrique

Le chiffrement asymétrique est une alternative au chiffrement symétrique. Le chiffrement asymétrique nécessite une paire de clés. Une clé publique qui est disponible pour quiconque voudrait envoyer un message à quelqu'un et une clé secrète qui est connue que par le propriétaire. Tous les messages chiffrés à l'aide de la clé publique peuvent uniquement être déchiffrés à l'aide de la clé privée correspondante, et vice versa les messages chiffrés par la clé privée ne peuvent être déchiffres que par la clé publique correspondante. L'avantage du chiffrement asymétrique est qu'on a pas à se soucier de la circulation des clés publiques sur internet, toutefois l'inconvénient réside dans le fait que le temps de calcul est beaucoup plus long pour le chiffrement/déchiffrement asymétrique. Un autre problème qui se pose avec l'utilisation du chiffrement asymétrique est l'authentification des clés et la non-répudiation des messages. La clé publique étant accessible à tout le monde, il n'y a aucun moyen de prouver l'identité de la partie qui envoie le message. Pour résoudre ce problème une solution a été mise en place qui consiste à associer à chaque paire de clés un certificat numérique.

Certificats numériques

Pour utiliser le chiffrement asymétrique, les utilisateurs doivent pouvoir trouver les clés publiques. L'outil utilisé pour mettre ces clés publiques à

disposition des autres est le certificat numérique. Un certificat est un ensemble d'informations qui identifie un utilisateur ou un serveur, les informations contenues dans un certificat sont notamment, le nom du propriétaire du certificat(*i.e*, l'identité certifiée), l'entreprise qui a délivré le certificat (appelée autorité de certification), le pays du propriétaire, sa clé publique ainsi que les dates de début et de fin de validité du certificat.

Une combinaison des deux techniques de chiffrement permet de réduire le coût du chiffrement, en effet quand deux entités souhaitent établir une communication sécurisée , elles s'authentifient mutuellement à l'aide de leurs certificats et du chiffrement asymétrique, puis ils s'échangent une clé secrète symétrique temporaire chiffrée avec les clés asymétriques qui ne sera valable que durant cette communication (on l'appelle aussi clé de session).

1.2.2 Intégrité

L'intégrité est la propriété assurant qu'une donnée n'a pas été modifiée de façon non autorisée. L'intégrité des données comprend quatre éléments :

- l'intégralité : la donnée n'a pas été tronquée ou augmentée (*e.g.*la taille d'un fichier reste la même que la taille initiale).
- la précision : la donnée doit être précise et lisible en entier (*e.g.*pas d'ambiguïté).
- l'authenticité : la donnée reste la même tout au long de sa vie.
- la validité : la donnée doit être à jour et valide lors de son exploitation.

Il existe plusieurs techniques pour s'assurer de l'intégrité on peut citer les plus populaires :

- fonction de hachage : associer un condensé à une donnée est un moyen de s'assurer que cette dernière n'a pas été altérée.
- signature numérique : la signature numérique permet d'authentifier l'auteur d'un message et son altération entre la signature et la réception.
- contrôle d'accès : permet d'assurer que l'accès et la modification d'une donnée n'est fait que par un ayant droit.

1.2.3 Disponibilité

La disponibilité est définie comme l'aptitude d'un dispositif, sous les aspects combinés de sa fiabilité, de sa maintenabilité et de la logistique de maintenance, à remplir ou à être en état de remplir une fonction à un instant donné ou dans un intervalle de temps donné.

1.3 Conclusion

Dans ce chapitre, nous avons défini les concepts de bases nécessaires à la compréhension du fonctionnement des systèmes pair-à-pair, ainsi que les principales propriétés de sécurité qui doivent être assurées dans tout système informatique. Dans le prochain chapitre, nous allons présenter un état de l'art non exhaustif des principaux systèmes de stockage distribués existant dans la littérature et quelques systèmes de réputation existants.

Chapitre 2

Etat de l'art

2.1 Systèmes de stockage pair-à-pair

Dans cette partie nous présentons quelques systèmes de stockage distribués pair-à-pair, nous n'allons pas faire une présentation exhaustive mais nous allons distinguer un sous ensemble des systèmes les plus connus. Un système de stockage distribué offre deux principales fonctionnalités : la sauvegarde et le versioning. Plusieurs systèmes offrant un des services individuellement ont été implémentés mais il en existe certains qui offrent les deux en même temps. Des systèmes de stockages pair-à-pair existants, on peut citer Napster et Gnutella, qui ont été largement utilisés et qui offrent des mécanismes de recherche de fichiers et de téléchargement à partir d'un large groupe d'utilisateurs. Les deux se focalisent plus sur la récupération d'information que la sauvegarde.

Freenet, en plus de la sauvegarde/récupération des données, assure l'anonymat en utilisant plusieurs mécanismes : le chiffrement des mots clés lors de la recherche, le *spoofing* des noeuds sources... Freenet ne maintient que les données populaires et supprime fréquemment les données non accédées pour faire de la place aux nouveaux ajouts.

Eternity [15] propose la redondance et la dispersion de l'information pour la réplication des données et comme Freenet offre l'anonymat. Les requêtes sont envoyées en diffusion à tout le réseau, et la réponse est envoyée de manière anonyme.

SFSRO [28] est un système de partage de fichiers qui offre l'accès de manière sécurisée en utilisant l'authentification à une base de données de réplicats en lecture seule. Comme SFSRO, CFS [37] a pour objectif d'assurer la disponibilité des données en utilisant la redondance sans affecter les performances. Il assure l'intégrité en offrant un système de fichier en lecture seule, sans avoir recours à des bases de données pour les réplicats. CFS sauvegarde les données sous forme de blocs dans un réseau distribué et utilise Chrod comme mécanisme de recherche.

PAST [10] utilise la même approche que CFS mais utilise Pastry [32] pour la recherche des données.

Plusieurs systèmes proposent un schémas de stockage qui permet la contribution des pairs à l'espace de stockage, Mojo Nation [6], passent par une troisième entité de confiance pour l'augmentation du quota d'un utilisateur lorsque ce dernier offre de l'espace de stockage, de la bande passante ou de la ressource CPU au système. PAST quant à lui utilise des quotas fixes définis sur la même carte à puce qui sert à l'authentification.

Plusieurs systèmes de stockage distribués, tentent de proposer des solutions qui supportent le passage à l'échelle. Inspirés de Napster et Gnutella CFS et PAST utilisent des algorithmes de routage et de stockage qui permettent le passage à l'échelle. OceanStore est conçu à la base pour le stockage des données du monde entier. Mais ces systèmes n'offrent que la sauvegarde et l'archivage et ne favorise pas la mise à jour des données.

2.1.1 Napster

Napster, a été créé en 1999 par Shawn Fanning alors qu'il était encore à la Northeastern University de Boston et Sean Parker. A l'origine Napster était un service pair-à-pair destiné uniquement à l'échange de fichiers musicaux. Le service original a fonctionné entre juin 1999 et juillet 2001. Sa technologie a permis aux gens d'échanger facilement des chansons au format MP3, ce qui a conduit l'industrie musicale à porter des accusations de violation massive du droit d'auteur. Bien que le logiciel ait été arrêté par décision judiciaire, il a ouvert la voie à de nombreux programmes P2P décentralisés, qui se sont révélés plus difficiles à contrôler [7]. Les noeuds du système Napster sont construits autour d'un serveur central qui joue le rôle d'un annuaire (architecture non entièrement distribuée), chaque nœud agit en tant que client et serveur en même temps pour l'échange de fichiers musicaux. Lorsqu'un pair rejoint le réseau pour la première fois, il transmet des informations sur tous les fichiers musicaux qu'il met à disposition des autres utilisateurs à un serveur central de méta-information appelé courtier ou agent de change. Lorsqu'un utilisateur de Napster souhaite avoir un fichier quelconque, il interroge la base de données du serveur de méta-données ; le serveur lui renvoie alors une liste des fichiers et leurs propriétaires (les pairs à partir desquels on peut les télécharger). Une fois les fichiers souhaités localisés, le pair les récupère directement à partir des propriétaires sans passer par le serveur de méta-informations. Outre la recherche et le partage de musique, Napster fournit aussi un service de messagerie pair-à-pair, des salles de discussion et des listes d'utilisateurs populaires (hot list : pairs avec lesquels les pairs ont déjà interagis).

2.1.2 Gnutella

Gnutella 0.4 [4], proposé en 2000 par Justin Frankel et Tom Pepper, est le premier réseau pair-à-pair non structuré. Il se base dans sa version originale sur une inondation totale (transmission à tous les voisins). Chaque requête possède un identifiant unique et est retransmise à tous les voisins pendant TTL sauts, TTL valant au maximum sept , ce qui doit assurer une inondation totale avec une grande probabilité. Les réponses remontent ensuite en sens inverse, chaque pair sachant de quel pair il avait reçu le message portant cet identifiant. Ce réseau est entièrement décentralisé, aucun pair n'ayant officiellement un rôle plus important que les autres. Il n'existe aucun point central, aucun point de faiblesse et, par conséquent, aucune autorité ponctuelle. Les requêtes peuvent être arbitrairement évoluées (portant sur le nom du fichier mais également ses méta-données) puisque interprétées par chaque pair du réseau. Cependant, cette approche pose le problème du passage à l'échelle, la charge de chaque pair augmentant linéairement avec le nombre de pairs du réseau. La version 0.6 intègre un système de super-pairs tel que présent dans FastTrack.

2.1.3 FastTrack

FastTrack, introduit en mars 2001 par Niklas Zennström, Janus Friis et Jaan Tallinn et utilisé notamment par Kazaa [30] et Skype [12], est un système pair-à-pair non structuré qui divise les pairs en deux catégories : les pairs ordinaires et les super-pairs. Chaque pair ordinaire est connecté à un super-pair sur lequel il publie ses fichiers partagés. Lorsqu'un pair ordinaire souhaite rechercher un fichier, il transmet la requête à son super-pair qui la retransmet à son tour par inondation à l'ensemble des super-pairs ; chaque pair ordinaire ayant publié ses partages sur un super-pair, la recherche sur les super-pairs permet de retourner toutes les réponses avec une charge réseau bien inférieure à celle de Gnutella. Le transfert du fichier est ensuite réalisé directement entre les pairs concernés. Le choix des super-pairs reste cependant problématique puisque, pour un réseau fonctionnant de manière optimale, ces super-pairs doivent être puissants et bien connectés. De plus, Kazaa se limite à deux catégories de pairs quand les différences de capacités entre les pairs peuvent, en fait, atteindre plusieurs ordres de magnitude. Gia [11], non présenté ici, propose une évolution continue entre pairs ordinaires et super-pairs : chaque pair accepte d'autant plus de connexions d'autres pairs (en devenant ainsi un point de passage important du réseau) qu'il a une grande capacité. Dans le cas de Skype, un serveur d'authentification est ajouté au centre du réseau. Ce serveur, opéré de manière privée, est responsable de la

création des comptes ainsi que de l'authentification.

2.1.4 eMule/eDonkey

La première version d'eDonkey [34] a été distribuée en 2000 par Jed McCaleb. Contrairement à Napster, le réseau eDonkey est basé sur l'agrégation d'un ensemble de serveurs. Le logiciel permettant de mettre en place un serveur est disponible publiquement et un grand nombre de serveurs sont donc proposés. Chaque utilisateur se connecte sur un seul serveur qui stocke la liste de ses fichiers partagés : les serveurs ne jouent que le rôle d'annuaire et ne partagent aucun fichier. Lors d'une requête, un utilisateur interroge son serveur qui, à son tour, interroge les autres serveurs. À la différence des systèmes à super-pairs (tels que Kazaa, les serveurs exécutent un code différent des clients, demandent d'importantes ressources et nécessitent une administration avancée : la relation entre les utilisateurs et les serveurs est donc bien une relation client-serveur et non pair-à-pair. La multiplication des serveurs ne résout toutefois pas tous les problèmes que pose Napster. Pour accueillir un grand nombre d'utilisateurs, plusieurs centaines de serveurs sont nécessaires, dont les plus populaires doivent être puissants. En 2004, l'association Razorback ouvrait un nouveau serveur accueillant 730.000 pairs, basé sur un bi-processeur AMD Opteron 248 avec 16GO de mémoire vive et connecté à internet à 50Mbps ; à l'époque, il s'agit clairement d'une machine extrêmement puissante et reliée à internet par un lien rapide. Le problème de disponibilité s'illustre ici, comme dans Napster, par la fermeture successive des différents serveurs pour des raisons légales et financières. Progressivement, les utilisateurs migrent vers les versions non centralisées d'eDonkey/eMule utilisant le réseau structuré Kademlia [35].

2.1.5 FreeNet

Freenet est un réseau informatique anonyme et distribué construit sur l'Internet. Il vise à permettre une liberté d'expression et d'information totale fondée sur la sécurité et l'anonymat, et permet donc à chacun de lire comme de publier du contenu. Il offre la plupart des services actuels d'Internet (courriel, Web, etc.). En raison de son anonymat, Freenet publie du contenu souvent illégal, des textes politiques comme de la pornographie infantile directement accessible depuis l'index des sites. Freenet a été créé suite à une inquiétude croissante à propos des libertés sur internet. Cette citation de Mike Godwin datant de 1996 résume cette inquiétude : « Je suis tout le temps soucieux au sujet de mon enfant et d'Internet, bien qu'elle soit encore trop jeune pour se connecter. Voilà ce qui m'inquiète. Je redoute que dans

10 ou 15 ans elle vienne me voir et me demande : "Papa, où étais-tu quand ils ont supprimé la liberté de la presse sur Internet ?"» [1]. Freenet est un data-store distribué, similaire par exemple aux tables de hachage distribuées des clients BitTorrent, le grand avantage étant que les données sont stockées sous forme chiffrée. Contrairement à BitTorrent où on choisit spécifiquement de partager certains fichiers, Freenet va commencer par relayer des données, et en stocker une partie. De cette manière, les données les plus populaires sont aussi les plus redondantes, un excellent moyen de résister à la censure. Le fait de stocker et de faire transiter les données sous forme de blocs chiffrés permet un possible déni (répudiation) puisque, lorsqu'on ne possède pas la clé pour déchiffrer le contenu, on ne peut identifier ce qui est stocké ou relayé par le nœud. La conception distribuée du réseau interdit à quiconque — même à ses concepteurs — d'interrompre son fonctionnement.

2.1.6 OceanStore

OceanStore [17], est un système de stockage de données conçu pour s'adapter à des milliards d'utilisateurs. N'importe quel utilisateur peut se joindre à l'infrastructure, et accéder au stockage en échange d'une compensation financière. Le service de stockage est fourni par une confédération de plusieurs compagnies. Les utilisateurs s'abonnent à un fournisseur de services OceanStore unique, même si elles peuvent consommer du stockage et de la bande passante fournies par plusieurs fournisseurs différents. Les prestataires peuvent acheter et vendre de la capacité et de la couverture entre eux, de manière transparente à l'utilisateur. Le modèle combine ainsi les ressources des systèmes fédérés pour fournir une meilleure qualité de service à l'utilisateur. OceanStore met en cache les données pèle-mêle, n'importe quel serveur peut créer une réplique locale de n'importe quel objet de données. Ces répliques locales permettent d'accéder plus rapidement aux données et d'assurer la robustesse du réseau et réduisent ainsi la congestion. Des techniques de redondance et de chiffrement sont utilisées pour protéger les données sur les serveurs qui les stockent, pour éviter que les données ne soient compromises en cas d'attaques. OceanStore emploie aussi un protocole de tolérance aux fautes et permet d'assurer une forte cohérence entre les différentes répliques des données. Le système OceanStore a deux objectifs principaux : la faculté d'être construit sur une infrastructure non fiable et la prise en charge des données nomades. OceanStore suppose que l'infrastructure est initialement

1. I worry about my child and the Internet all the time, even though she's too young to have logged on yet. Here's what I worry about. I worry that 10 or 15 years from now, she will come to me and say "Daddy, where were you when they took freedom of the press away from the Internet ?

non fiable, en effet un serveur peut tomber en panne sans préavis et peut être accessible à un tiers. Pour cela le OceanStore utilise des techniques de chiffrement telles que dans [27]. La large utilisation d'OceanStore, implique une disponibilité locale des données n'importe où et n'importe quand, pour cela OceanStore utilise une politique de promiscuité en mettant en cache des données autorisées. Noter que la duplication des données complique la cohérence des données. La principale unité d'information dans OceanStore est un *objet persistant*, les utilisateurs interagissent avec des objets pour stocker les données et pour communiquer avec d'autres utilisateurs. Que ce soit le nommage, la cohérence ou le contrôle d'accès, tout est centré autour de la notion d'objet. OceanStore offre plusieurs services comme : le versionning, le contrôle d'accès, le chiffrement des données, les répliques flottantes et l'optimisation de la localisation des données. OceanStore existe au format de prototype actuellement, l'API est développé en Java en utilisant Jaguar [55]. Initialement, OceanStore va communiquer avec les autres applications via une interface de système de fichiers UNIX et avec le web via un proxy en lecture seule.

2.1.7 Farsite

Farsite [54], est un système de stockage distribué. Il est conçu pour fonctionner comme un serveur de fichier mais qui est implémenté de manière distribuée sur un réseau constitué de près de 100000 noeuds non fiables avec un très haut débit. Farsite est destiné à un usage dans le milieu académique comme les universités et supporte le stockage de fichiers, l'exécution des applications scientifiques nécessitant des ressources importantes et les bases de données partagées. Farsite nécessite une administration minimale et non centralisée. Farsite est conçu en se fixant quelques contraintes sur l'environnement :

- la majorité des noeuds doivent être accessibles tout le temps ;
- les données ne peuvent être accessibles en lecture et en écriture en même temps ;
- optimisme par rapport au nombre de noeuds malveillants potentiels ;
- le système tourne sous Windows, et n'utilise pas la technique de *cryptopaging* [45, 52], ce qui augmente l'exposition des données des utilisateurs.

Chaque client Farsite joue 3 rôles : il est client, un membre d'un répertoire de groupe et propriétaire de fichiers (rôle ignoré). Un client est une machine qui interagit avec l'utilisateur et un répertoire de groupe est un ensemble de machines qui gèrent collectivement des fichiers. Chaque membre du groupe stocke une copie des fichiers et prend en charge les requêtes des client en lec-

ture/écriture des fichiers. Farsite implémente deux composants : un service au niveau utilisateur (*daemon*) et un pilote au niveau noyau (*driver*). Le driver exécute des opérations nécessitant les performances du noyau comme : le chargement de l'interface du système de fichiers, l'interaction avec le gestionnaire de cache et les opérations de chiffrement. Toutes les autres opérations sont implémentées au niveau de l'application : réplication, gestion des méta-inforamtions, validation du contenu des fichier....

2.1.8 Pstore

Pstore [16], est un système de stockage pair-à-pair. Pstore donne à l'utilisateur la possibilité de sauvegarder ses données dans un réseau distribué composé de pairs non fiables et de les restaurer de manière sûre. Pstore offre quatre opérations à l'utilisateur : sauvegarder, mettre à jour, restaurer et supprimer des données via plusieurs interfaces (ligne de commande, système de fichiers ou GUI). Pstore offre le versionning en maintenant des images différentes des données. Pstore a trois objectifs : la disponibilité, la sécurité et l'exploitation efficace des ressources. La disponibilité est assurée en utilisant la réplication, des copies sont mises à disposition sur plusieurs serveurs. La sécurité est assurée en utilisant les techniques de chiffrement et les données ne sont accessibles en lecture que par leur propriétaire. Quant à la gestion des ressources, Pstore réduit les coûts en partageant les données déjà stockées et en effectuant de l'échange de données que lorsque c'est nécessaire. Pstore est implémenté en C++ et utilise une librairie à trois parties pour le chiffrement : RSA pour le chiffrement à clé publique, Rijndael (AES) pour le chiffrement symétrique et SHA-1 pour le hachage. Pstore est au stade de prototype, et la version actuelle fourni une interface en ligne de commande pour les différentes opérations (stockage,mise à jour, restauration, suppression des données et génération des clés).

2.1.9 Pastis

Pastis [50, 49], est un système de fichiers distribué multi-écrivain. Il vise à rendre l'utilisation de la capacité de stockage totale de centaines de milliers d'ordinateurs connectés à l'Internet au moyen d'un réseau pair-à-pair (P2P). La réplication permet un stockage persistant malgré un mouvement perpétuel et transitoire des nœud du réseau [36]. Des techniques cryptographiques garantissent l'authenticité et l'intégrité des données stockées dans le système de fichiers. Le routage et le stockage de données dans pastis sont pris en charge par le protocole de routage Pastry [32] et la table de hachage distribuée PAST [10]. Les bonnes propriétés de localisation de Pastry/PAST

permettent à Pastis de minimiser les temps d'accès réseau, réalisant ainsi un bon niveau de performance lors de l'utilisation d'un modèle de cohérence détendue. En outre, Pastis n'emploie pas de protocoles lourds tels que BFT (tolérance aux pannes byzantines), comme les autres systèmes de fichiers P2P multi-écrivains font. La validation d'une mise à jour d'un fichier dans Pastis nécessite la présentation d'un certificat signé par le propriétaire du fichier, afin de prouver l'authenticité de la requête d'écriture et autoriser l'accès au fichier en écriture.

Un prototype de Pastis a été développé. Il est codé en Java 1.4 et utilise FreePastry 1.4.1 et l'implémentation open source Pastry / PAST. L'évaluation préliminaire réalisée par l'émulation et en utilisant le simulateur LS3 suggère que Pastis passe bien à l'échelle que ça soit en taille du réseau et en nombre de fichiers utilisateurs. Selon cet évaluation Pastis est en moyenne 1.6 plus lent que NFS et à priori plus performant que Ivy et Oceanstore.

2.2 La confiance et les systèmes de réputation

Dans cette partie nous allons présenter et décrire de manière non exhaustive un ensemble de systèmes de réputations existants dans les réseaux pair-à-pair. L'objectif principal des modèles de confiance basés sur la réputation est d'identifier parmi les noeuds qui sont disponibles les plus fiables pour leur assigner des tâches ou leur envoyer des requêtes. Globalement, ces modèles suivent le même processus pour la sélection des noeuds de confiance. La première étape consiste à se baser sur sa propre expérience avec un nœud donné pour évaluer sa fiabilité ; c'est ce qu'on appelle la confiance directe. Cette confiance directe peut être évaluée en prenant en compte plusieurs facteurs comme : la qualité des précédentes transactions, leur nombre, l'importance de chacune d'entres elles et la satisfaction obtenue après chaque transaction...

En plus de l'expérience personnelle, un pair peut utiliser les expériences des autres pairs (expérience indirecte) pour évaluer le degré de fiabilité d'un autre pair. Le crédit accordé aux expériences indirectes est différent de l'-expérience personnelle d'un pair, des modèles comme Trust Evolution [57], Semantic Web et Global Trust [58], font la différence entre la confiance faite à un pair en tant que fournisseur de service et en tant que recommandateur (expérience indirecte) ; et permettent ainsi de filtrer les pairs qui ont un comportement malveillant en notation (les pairs qui recommandent des pairs malveillants) des pairs qui ont un comportement malveillant en tant que fournisseurs de services. La deuxième étape, est le calcul de la réputation en se basant sur les différentes expériences récoltées (directes et indirecte). A

la fin de cette étape un pair décide s'il effectue une avec un autre pair ou pas. A la fin de chaque transaction, unetransaction réévaluation est faite en fonction du résultat de la transaction, et le fournisseur de service est évalué en fonction de la satisfaction du client (récompensé ou puni en conséquence).

La réputation et la confiance sont évaluées de manière différente pour chaque modèle, certains utilisent les réseaux bayésiens (comme BNTBM [23]), d'autres utilisent la logique floue (*fuzzy logic*) (comme PATROL-F [18]), d'autres s'inspirent de la biologie (comme AntRep, TDTM [60] et TACS), et d'autres donnent une expression analytique pour le calcul de la confiance (comme Global Trust et Semantic Web). Chacune de ses mécanismes a ses avantages et inconvénients, la logique floue permet de modéliser la réputation et la confiance de manière plus compréhensible à l'homme. Cependant elle est très difficile à implémenter sur de très grands réseaux (problème de passage à l'échelle).

Les réseaux bayésiens, procurent une flexibilité qui permet de modéliser les différentes facettes de la confiance et les différents contextes de l'environnement. Mais cette technique coûte énormément en temps de calcul. Les expressions analytiques sont plus faciles à lire et à comprendre, mais ne fournissent pas beaucoup de possibilité pour la modélisation des différents facteurs impliqués dans la modélisation de la confiance. Enfin, les techniques inspirées de la biologie ont démontré leur efficacité lors du passage à l'échelle et leur adéquation dans les réseaux pair-à-pair. Cependant, dans certains cas leurs indéterminismes et leurs approximations mènent à choisir des pairs malveillants comme étant les plus fiables. Dans ce qui suit, certains des systèmes cités ci-dessus seront présentés en détails.

2.2.1 CuboidTrust

CuboidTrust [24], est un modèle de confiance global pour les réseaux pair-à-pair basé sur la réputation, qui est construit autour de quatre relations basées sur trois facteurs de confiance : la contribution d'un pair au système (ressources), la fiabilité d'un pair (calculée à partir des retours des autres pairs : feedback) et la qualité des ressources mises à disposition par un pair. Chacun de ses facteurs est représenté par un vecteur formant un cuboide de coordonnées x, y, z et dénoté par $P_{x,y,z}$, où z représente la ressource stockée par y et évaluée par x et $P_{x,y,z}$ l'évaluation de la ressource z par le pair x. Les valeurs attribuées à la qualité de la ressources sont binaires : $+1$ (positif) ou -1 (négatif). La ressource est évaluée positivement si à la fin d'un téléchargement sans incident la ressource est jugée intègre et authentique et évaluée négativement si elle n'est pas intègre ou si le téléchargement a été interrompu. Les cuboides sont associés à deux matrices de coefficient E et

D telles que chaque matrice représente un plan du cuboide : le plan E selon l'axe des Y et le plan D selon l'axe des Z. Soit un réseau dont le nombre de pairs est M et N le nombre de ressources. Les éléments des matrices D et E son respectivement :

$$D_{ij} = avg(P_{i,j,*}) \in [-1,1] \text{ tq } 1 \leq i \leq M, 1 \leq j \leq M$$
$$E_{ij} = avg(P_{i,*,j}) \in [-1,1] \text{ tq } 1 \leq i \leq M, 1 \leq j \leq N$$

$P_{i,j,*}$ représente le vecteur avec $x = i$ et $y = j$ dans le cuboide, et $Pi, *, j$ représente le vecteur avec $x = i$ et $z = j$ dans le cuboide. La matrice D contient les notes moyennes attribuées par un pair i à un pair j, et la matrice E contient les notes moyennes attribuées par un pair i à une ressource j. Les quatre relations sont basées sur les trois facteurs précédemment cités : la contribution, la fiabilité et la qualité de la ressource respectivement représentés par les trois vecteurs : C, T et Q. La première relation combine deux d'entres eux : la contribution et la fiabilité. Cette relation est représentée par la formule suivante :

$$C_i = \sum_{j=0}^{M}(D_{ij} \times T_j), 1 \leq i \leq M$$

où T_j est la fiabilité du pair j, et C_i représente la contribution du pair i dans le système du point de vue des autres pairs qui ont déja eu affaire au pair i. La deuxième relation combine deux facteurs : la fiabilité et la qualité de la ressource. et elle est représentée par la formule suivante :

$$T_i = \sum_{j=0}^{N}(E_{ij} \times Q_j), 1 \leq i \leq N$$

où Q_j est la qualité de la ressource j, et T_i la fiabilité du pair i. La troisième relation combine la fiabilité du pair dans le système ainsi que la qualité de la ressource échangée et elle est décrite par la formule suivante :

$$Q_i = \sum_{j=0}^{N}(E_{ij}^T \times T_j), 1 \leq i \leq N$$

où E_{ij}^T est l'élément de coordonnées (i, j) de la transposée de la matrice E, et il représente la note moyenne accordée à la ressource i par le pair j. T_j est la fiabilité du pair j, et Q_i la qualité de la ressource i dans le système.

et la dernière relation prend en compte la contribution et la fiabilité de la façon suivante :

$$T_i = \sum_{j=0}^{M}(D_{ij}^T \times C_j), 1 \leq i \leq M$$

où D_{ij}^T est l'élément de coordonnée (i,j) de la transposée de la matrice D, soit la note moyenne attribuée par le pair j au pair i, C_j représente la contribution du pair j , et T_i est la fiabilité du pair i. En combinant ces quatre relation on obtient :

$$C = D \times T = D \times E \times Q = D \times E \times E_T \times T = \\ D \times E \times E_T \times D_T \times C = (D \times E) \times (D \times E)T \times C \ (1)$$

en faisant k itérations de l'équation (1), on obtient :

$$C_{(k)} = R \times C_{(k-1)} = R_1 \times C_{(k-2)} = ... = R_k \times C_{(0)}$$

où $R = (D \times E) \times (D \times E)_T$, et $C_{(k)}$ représente la note attribuée à la contribution globale pour chaque pair après k itérations. Une autre combinaison des quatre relations nous donne aussi l'équation suivante :

$$Q = E_T \times T = E_T \times D_T \times C = E_T \times D_T \times D \times T = \\ E_T \times D_T \times D \times E \times Q = (D \times E)T \times (D \times E) \times Q \ (2)$$

en faisant k itérations de l'équation (2), on obtient :

$$Q_{(k)} = S \times Q_{(k-1)} = S_1 \times Q_{(k-2)} = ... = S_k \times Q_{(0)}$$

où $S = (D \times E)_T \times (D \times E)$, et $Q_{(k)}$ représente la note globale attribuée à la qualité de chaque ressource après k itérations. Dans CuboiTrut, la note globale attribuée à pair (le taux de confiance en ce pair) est représentée par la note globale de sa contribution au système.

2.2.2 EigenTrust

EigenTrust [47], est l'un des systèmes de réputation les plus cités et le plus utilisé comme référence dans les modèles de réputation des réseaux pair-à-pair. L'idée de EigenTrust est d'attribuer une note de confiance globale à chaque pair qui est calculée à partir des notes locales attribuées par chaque pair et pondérées par le taux de confiance en ces pairs. L'attribution des notes se base sur l'historique d'échanges entre les différents pairs. Le modèle EigenTrust est complétement distribué et permet d'assurer l'anonymat aux pairs qui attribuent les notes. Chaque transaction est évaluée localement par le pair i qui reçoit la ressource du pair j, notée $tr_{i,j}$ elle prend la valeur (+1) si la transaction est satisfaisante et (-1) sinon. La note de confiance locale s_{ij} est définie par la formule suivante :

$$s_{ij} = sat(i,j) - unsat(i,j)$$

où $sat(i,j)$ est le nombre de transactions satisfaisantes qu'a effectué i avec j et $unsat(i,j)$ le nombre de transactions non satisfaisantes.

La note de confiance locale c_{ij} est normalisée de manière à ce qu'elles soient comprises dans l'intervalle $[0,1]$:

$$c_{ij} = \frac{\max(s_{ij},0)}{\sum_j(s_{ij},0)}$$

Une distribution de probabilité $\mathbf{P}$ (avec $p_i \in [0,1]$) est définie sur les pairs pré-fiables. Si un ensemble de pairs p sont déjà connus pour être fiables, alors $p_i = 1/ \mid P \mid$ Si $i \in P$ et $p_i = 0$ sinon. Grâce à cette définition on peut normaliser une fois de plus la valeur de la confiance locale $c_{ij} \in [0,1]$ comme suit :

$$c_{ij} = \begin{cases} \frac{\max(s_{ij},0)}{\sum_j \max(s_{ij},0)} \text{ Si } \sum_j \max(s_{ij},0) \neq 0 \\ P_j \text{ sinon} \end{cases}$$

Afin d'avoir une note de confiance globale, la note locale est agrégée avec les autres notes locales des pairs que le pair i connait (ses amis). Pour cela le pair i demande toutes les notes détenues par les autres pairs à propos du pair k dont il veut calculer la note de confiance, et il pondère les notes obtenues avec la confiance qu'il a lui même des pairs sollicités, et il utilise la formule de calcul suivante :

$$t_{ik} = \sum_j c_{ij} c_{jk}$$

L'ensemble des notes est écrit sous forme d'une matrice C composée des différents $[c_{ij}]$, et l'ensemble des notes de confiance globales est stocké dans un vecteur $\overrightarrow{t_i}$ qui contient l'ensemble des t_{ik} (noter que $\sum_k t_{ik} = 1$), alors on a : $t_i = C_T c_i$. On appliquant une itération (correspondant au nombre de fois que le pair i sollicite les autres pairs pour avoir leurs confiance en un pair k), le pair i aura une vue complète du réseau et on obtient :

$$t_i = (c_T)^n c_i$$

2.2.3 GroupRep

GroupRep [53], est un modèle de confiance qui met en évidence trois niveaux de classifications pour la relation de confiance, la confiance entre groupes, la confiance entre les groupes et les pairs et enfin la confiance entre les pairs. La première relation est celle qui décrit la confiance que possède un groupe i en un groupe j, elle est calculée comme suit :

$$Tr_{G_iG_j} = \begin{cases} \frac{u_{G_iG_j} - c_{G_iG_j}}{u_{G_iG_j} + c_{G_iG_j}} \text{ Si } u_{G_iG_j} + c_{G_iG_j} \neq 0 \\ Tr_{G_iG_j}^{reference} \text{ Si } u_{G_iG_j} + c_{G_iG_j} = 0 \text{ et } \exists Trust_{G_iG_j}^{path} \\ Tr_{G_iG_{strange}} \text{ Sinon} \end{cases}$$

où $u_{G_iG_j} \geq 0$ et $c_{G_iG_j} \geq 0$ sont respectivement : l'utilité et le coût que les pairs du groupe j ont accordés aux pairs du groupe i. Soit un ensemble de chemins référence entre le groupe G_i et le groupe G_j, alors le chemin le plus fiable et le plus sûr est celui qui contient le maximum de groupes fiables. Cependant, $Tr_{G_iG_j}^{reference}$ est définie comme étant la valeur minimale de confiance rencontrée tout au long du chemin référence le plus fiable.

La confiance que peut avoir un groupe i en un groupe étranger (avec qui il n'a jamais eu d'échanges) se calcule comme suit :

$$Tr_{G_iG_{strange}} = \begin{cases} \frac{u_{G_iG_{strange}} - c_{G_iG_{strange}}}{u_{G_iG_{strange}} + c_{G_iG_{strange}}} \text{ Si } u_{G_iG_{strange}} + c_{G_iG_{strange}} \neq 0 \\ 0 \text{ Sinon} \end{cases}$$

La deuxième relation de confiance entre un groupe G_i et un pair j notée $Tr_j^{G_i}$ est donnée par l'équation suivante :

$$Tr_j^{G_i} = \begin{cases} \frac{u_j^{G(j)} - c_j^{G(j)}}{u_j^{G(j)} + c_j^{G(j)}} \text{ Si } u_j^{G(j)} + c_j^{(j)} \neq 0 \text{ et } j \in G_i = G(j) \\ Tr_{stange}^{G(j)} \text{ Si } u_j^{G(j)} + c_j^{G(j)} = 0 \text{ et } j \in G_i = G(j) \\ \min Tr_{G_iG(j)}, Tr_j^{(j)} \text{ Si } j \ni G_i \end{cases}$$

Lorsque le pair est étranger au groupe, la confiance est calculée de la manière suivante :

$$Tr_j^{G_i} = \begin{cases} \frac{u_{strange}^{G(j)} - c_{strange}^{G(j)}}{u_{strange}^{G(j)} + c_{strange}^{G(j)}} \text{ Si } u_{strange}^{G(j)} + c_{strange}^{G(j)} \neq 0 \\ 0 \text{ Si } u_{strange}^{G(j)} + c_{strange}^{G(j)} = 0 \end{cases}$$

où $G(j)$ est le groupe auquel le pair j appartient, et $c_j^{G(j)} \geq 0$ et $u_j^{G(j)} \geq 0$ sont respectivement l'utilité et le coût que le pair j accorde aux pairs dans le groupe $G(j)$.

la dernière relation est celle de confiance entre deux pairs et elle est donnée par la formule suivante :

$$Tr_{ij} = \begin{cases} \frac{u_{ij} - c_{ij}}{u_{ij} + c_{ij}} \text{ Si } u_{ij} + c_{ij} \neq 0 \\ Tr_j^{G(i)} \text{ Si } u_{ij} + c_{ij} = 0 \end{cases}$$

La mise à jour des notes de confiances locales se font à la base des notes attribuées aux autres pairs. Un groupe mettra à jour les informations concernant ses membres, les groupes amis, les pairs étrangers et groupes étrangers en les pondérant par les notes de confiances qu'il a reçu par les autres. Quand un pair i a fini son interaction avec le pair j, il met à jour sa note locale en premier et ensuite il transmet la mise à jour à $G(i)$, si $j \in G(i)$ alors la note de j est mise à jour , sinon met à jour la note de $G(j)$ (groupe auquel appartient j). Une fois la note de $G(j)$ est mise à jour au niveau du groupe de i, elle est transmise au groupe de j afin que ce dernière mette à jour la confiance de j. Un groupe mettrait à jour les informations de confiance des pairs ou des groupes étrangers si un de ses membres leur a déjà fourni un service ou bien ces étrangers ont déjà fourni un service à l'un des membres du groupe.

2.2.4 AntRep

AntRep [59], est un modèle de réputation distribué pour les réseaux pair-à-pair. Il est basé sur le paradigme de l'intelligence en essaim [essaim]. Il s'inspire sur un système de fourmis éclaireuses [3] pour la construction de relations de confiance dans les réseaux P2P efficacement. Dans le modèle AntRep, chaque pair dispose d'une table de réputation (RT), dont le principe se rapproche de celui de la table de routage à vecteur de distance [8]Les deux tables diffèrent en deux points :

- chaque pair dans la table de réputation correspond à une seule réputation.
- la métrique est la probabilité de choisir chaque voisin comme saut suivant.

Il ya deux mode d'envoi de fourmis pour un pair donné (avec une réputation) :

- fourmis *unicast*, envoyées au voisin avec la probabilité la plus élevée dans la table de réputation,
- fourmis *broadcast*, envoyées quand il n'y a pas de préférence entre les voisins. Ce mode est aussi choisi lorsqu'aucun chemin au pair n'a été exploré ou les informations sur ce pair ne sont pas à jour

Une fois que les fourmis envoyées trouvent la preuve nécessaire (informations sur la réputation), elles font un retour en arrière (un retour de fourmis est généré). A chaque retour arrière de fourmis à un nœud i la table de réputation de ce nœud est mise à jour en même temps. Cette mise à jour est effectuée grâce à la règle suivante :

$$P_i(t) = \frac{[\tau_j(t)]_\alpha [\eta_j(t)]_\beta}{\sum_{j \in N} [\tau_j(t)]_\alpha [\eta_j(t)]_\beta}$$

où α et β sont des constantes qui varient en fonction de l'environnement du réseau, η_i est la qualité du lien entre le pair courant et son pair voisin i. τ_i est le "dépôt de phéromones", qui est défini comme suit : si au temps $t + \Delta t$, le pair courent reçoit un retour de fourmis du pair i, alors :

$$\tau_i(t + \Delta t) = f(\tau_i(t), \Delta t) + \Delta p$$
$$\tau_j(t + \Delta t) = f(\tau_j(t), \Delta t) \ , \ j \in N, j \neq i$$

où $\Delta p = \frac{k}{f(c)}$, tel que $k > 0$ est une constante ; $f(c)$ une fonction croissante du coût c et c peut être n'importe quel paramètre qui donne des informations sur les éléments de preuves ou bien le scénario du réseau actuel. $f(\tau_i(t), \Delta t)$ est la fonction d'évaporation de phéromones définie comme suit :

$$f(\tau_i(t), \Delta t) = \frac{\tau_i(t)}{e^{\frac{\Delta t}{k}}}$$

Une des autres fonctions des phéromones, est de décider quand envoyer des fourmis de broadcast. Lorsqu'un pair k reçoit une requête à un instant t, il cherche d'abord s'il a une entrée dans sa table de réputation qui correspond aux données demandées, s'il n'a pas d'entrée correspondant il envoie simplement des fourmis de broadcast. Dans le cas contraire, il trouve le pair avec la plus grande probabilité.

2.2.5 Semantic Web

Semantic web [19], est un modèle où la confiance entre deux pairs : P et Q est calculée en additionnant la confiance de tous les pairs qui se trouvent sur les chemins qui peuvent les relier : au départ une recherche est faite sur tous les chemins qui connectent P et Q ; puis, pour chaque chemin les notes associées à chaque pair aux deux extrémités du chemin sont multipliées et enfin toutes les notes obtenues sont additionnées. Soient : N le nombre de chemins qui peuvent relier les pairs P et Q, D_i le nombre de sauts entre P et Q sur le i^{me} chemin. On appelle M l'ensemble des amis (pairs en qui Q a confiance) de Q, et m_i le voisin direct de Q sur le chemin i. Enfin chaque chemin i est pondéré par un poids w_i ; le poids de chaque chemin est calculé comme suit (donnant un plus grand poids au plus court chemin) :

$$w_i = \frac{\frac{1}{D_i}}{\sum_{i=1}^{N} \frac{1}{D_i}}$$

Si P et Q sont amis (noté $P \rightarrowtail Q$), ou voisins (noté $P \leftrightarrow Q$, alors la confiance de P en Q, $T_{P \rightarrowtail Q}$ peut être calculée directement sinon on la calcule comme suit :

$$T_{P \rightarrowtail Q} = \sum_{i=0}^{N} \frac{T_{m_i \rightarrowtail Q} \times \prod_{i \rightarrowtail j \bigcup i \leftrightarrow j} R_{i \rightarrowtail j} \times \frac{1}{D_i}}{\sum_{i=0}^{N} \frac{1}{D_i}}$$
$$= \sum_{i=0}^{N} T_{m_i \rightarrowtail Q} \times \prod_{i \rightarrowtail j \bigcup i \leftrightarrow j} R_{i \rightarrowtail j} \times w_i$$

où $R_{T_{i \rightarrowtail j}}$ est le facteur qui décrit la confiance que le pair i peut avoir en l'opinion de j.

2.2.6 PeerTrust

PeerTrust [56], est un modèle de confiance basé sur la réputation, qui compare et quantifie la fiabilité des pairs en se basant sur les retours (*feedback*) après les transactions. Il a deux principales caractéristiques :

- le modèle introduit trois facteurs de fiabilité et deux facteurs adaptatifs lors du calcul de la fiabilité des pairs.
- Il définit une métrique qui combine tous ces facteurs pour le calcul de la confiance.

Les facteurs de fiabilité sont : le retour reçu par les autres pairs, le nombre de transactions qu'un pair a effectué et la crédibilité des pairs sources. Les facteurs adaptatifs sont : le contexte de la transaction et le contexte de la communauté (environnement). Pour calculer la confiance en un pair u on pose : $I(u, v)$ le nombre total de transactions entre u et v, $I(u)$ le nombre de transactions faites entre u et les autres pairs, $P(u, i)$ le pair participant à la $i^{\text{ème}}$ transaction de u, $S(u, i)$ est la taux de satisfaction reçue par u de la part de $P(u, i)$ lors de la $i^{\text{ème}}$ transaction et $Cr(v)$ est la crédibilité du retour fait par le pair v. Enfin on définit $TF(u, i)$ le facteur adaptatif du contexte de la $i^{\text{ème}}$ transaction du pair u et $CF(u, i)$ le facteur adaptatif lié au contexte de la communauté de u ; alors la confiance du pair u notée $T(u)$ est donnée par l'équation suivante :

$$T(u) = \alpha \; times \sum_{i=1}^{I(u)} S(u, i) Cr(P(u, i)) TF(u, i) + \beta CF(u)$$

où α et β sont les poids des deux facteurs adaptatifs de l'évaluation collective et le contexte de communauté respectivement. Le modèle PeerTrust propose deux méthodes pour le calcul de la crédibilité. La première consiste à considérer la confiance qu'on a en un pair comme sa crédibilité, donc un retour d'un pair dont la confiance est grande est plus crédible et son poids est plus grand lors du calcul. La crédibilité dans ce cas est calculée comme suit :

$$Cr(P(u, i)) = \frac{T(P(u,i)}{\sum_{j=1}^{I(u)} T(P(u,j))}$$

La seconde façon de calculer la crédibilité d'un pair consiste à se baser sur les similarités entre un pair w et v (transactions avec les mêmes pairs) pour évaluer le poids qu'on doit donner aux retours faits par un pair v. Soit $IS(v)$ l'ensemble des pairs qui ont eu une interaction avec v et $IJS(v,w)$ est l'ensemble de pairs qui ont eu une interaction avec v et w, la crédibilisé se calcule alors comme suit :

$$Cr(P(u,i)) = \frac{Sim(P(u,i),w)}{\sum_{j=1}^{I(u)}(Sim(P(u,i),w))}$$

où

$$Sim(v,w) = 1 - \sqrt{\frac{\sum_{x \in IJS(v,w)} \left(\frac{\sum_{i=1}^{I(x,v)} S(x,i)}{I(x,v)} - \frac{\sum_{i=1}^{I(x,w)} S(x,i)}{I(x,w)} \right)^2}{|IJS(v,w)|}}$$

Le facteur de contexte de la transaction, se mesure en prenant en compte plusieurs paramètres comme la taille, la catégorie, l'horodatage de la transaction.... Par exemple les transactions récentes auront plus de poids que les anciennes.

Enfin le facteur de contexte de communauté qui représente le nombre de retours qu'un pair a pu fournir aux autres pairs à propos de ses transactions avec les autres se calcule comme suit :

$$CF(u) = \frac{F(u)}{I(u)}$$

où $F(u)$ est le nombre total de retours que le pair u a fait aux autres pairs.

2.2.7 TACS

TACS (Trust Ant Colony System) [2, 33, 22], est un modèle de réputation pour les réseaux pair-à-pair basé sur le modèle biologique des colonies de fourmis ACS (Ant Colony System). Dans ce modèle on assimile la trace de phéromone au taux de confiance qu'un pair peut avoir en son voisin. L'objectif du modèle est non seulement de trouver le pair le plus fiable mais aussi de déterminer le chemin le plus fiable pour y arriver. La procédure de localisation du pair le plus fiable et le chemin le plus fiable qui mène à lui est la suivante :

- Le client TACS C exécute l'algorithme pour localiser le serveur optimal S qui fourni le service s ;
- TACS lance l'algorithme ACS pour mettre à jour la trace de phéromone du réseau ;

- Une fois l'algorithme fini un chemin optimal menant à S' qui fourni le service s est identifié et C en est informé ;
- Le client envoie une requête au serveur S' ;
- A la fin de la transaction (service s fourni), C évalue la qualité de la transaction avec S' ;
- Si le C est pas satisfait il puni S' en supprimant la trace du chemin qui mène à lui.

Chaque pair possède sa propre trace pour chaque chemin qu'il connait du réseau. Avant chaque transaction un pair doit lancer l'algorithme TACS pour chercher le meilleur chemin en fonction des mises à jour faites par les autres pairs. Au parcours du réseau à la recherche du meilleur chemin, les fourmis prennent une décision à chaque nœud d'arrêter ou de continuer l'exploration des autres chemins. elles s'arrêtent au niveau d'un nœud offrant le service demandé si le niveau de confiance du chemin menant à ce nœud est supérieur au égal à un seuil de confiance minimum fixé sinon elles choisissent un autre nœud jamais visité en utilisant la probabilité suivante :

$$p_k(c,s) = \frac{\tau_{cs}}{\sum_{u \in J_k(c)} \tau_{cu}}$$

où τ_{cs} est la trace de phéromone du chemin connectant c et s, $J_k(c)$ est l'ensemble des voisins de c qui n'ont pas encore été visités par les fourmis k. Si aucun nœud connu n'est rencontré et aucun des noeuds voisins n'ait atteint le seuil de confiance minimum $q_o \in [0,1]$ alors celui qui a la plus grande confiance $q \in [0,1]$ est choisi. A chaque parcours d'un chemin la trace de phéromone est mise à jour par les fourmis de la manière suivante :

$$\tau_{cs} = \tau_{cs} + (1-\varphi) \times \varphi \times (1-\tau_{cs}) \times \tau_{cs}$$

où $\varphi \in [0,1]$ est une constante utilisée pour contrôler la mise à jour locale de phéromone. A chaque exploration des chemins par les fourmis, elles remontent l'information au client qui doit faire le choix de garder le chemin sélectionne par les fourmis ou bien choisir un autre qu'il jugera meilleur. Pour pouvoir évaluer quel chemin est meilleur, le client utilise l'équation suivante :

$$Q(S_k) = \frac{\%A_k}{\sqrt{Length(S_k)}} \times \overline{\tau^k}$$

où S_k est la solution trouvée par la fourmi k, $\%A_k$ est le pourcentage de fourmis qui ont sélectionné le même chemin que la fourmi k, $Length(S_k)$ est la longueur du chemin S_k, et $\overline{\tau^k}$ est la moyenne de phéromones de la solution.

La mise à jour de phéromones se fait en choisissant le meilleur chemin comme suit :

$$\tau_{cs} = \tau_{cs} + \rho\tau_{cs}^2 Q(S_{Best})$$

ou $\rho \in [0, 1]$ est une constante utilisée pour le contrôle de la mise à jour globale de la trace de phéromones. Ce processus de lancement de l'algorithme ACS et la mise à jour des traces de phéromones se répète de nombreuses fois en fonction de la taille du réseau. Une fois l'algorithme terminé, le client envoie une requête de service pour le nœud sélectionné par TACS. La non satisfaction du client se traduit par l'effacement du chemin entre le client et le serveur en question (évaporation de phéromones). Cependant, si la satisfaction du client est supérieure ou égale à 0,5 (dans $[0, 1]$ alors le calcul de la satisfaction se fait comme suit :

$$\tau_{cs} \leftarrow \tau_{cs} - \varphi(1 - Sat)2df_{cs}$$

sinon, si la satisfaction du client est inférieure à 0,5 alors la formule utilisée est la suivante :

$$\tau_{cs} \leftarrow (\tfrac{\tau_{cs}}{df_{cs}} - \varphi)Sat$$

où df_{cs} est la fonction qui assigne des valeurs dans $[0, 1]$ à tous les chemins et est définie comme suit :

$$df_{cs} = \sqrt{\tfrac{d_{cs}}{L(L-d_{cs}+1)'}} \text{ tq, } d_{cs} \in \{1, 2, ...L\}$$

où L est la longueur actuelle de tout le chemin et d_{cs} la distance entre le client et le serveur (nombre de sauts).

2.3 Détection des pairs malveillants

Comme on le verra dans les chapitres suivants, la présence de pairs malveillants (qui ont un ou plusieurs des comportements malveillants décrit ci-dessous), induit la perte de données. Afin d'éviter cette perte de données, nous avons proposé de mettre en place un système de confiance basé sur la réputation pour détecter les pairs malveillants avant qu'ils ne puissent causer la perte de données. Le système de confiance proposé est basé sur la réputation des pairs, chaque transaction entre deux pairs est évaluée et les appréciations des pairs sont échangés entre eux pour avoir une vue globale de l'état du système. Dans cette section nous allons décrire en détail un système de réputation qui a été développé pour les réseaux Ad-Hoc. Ce système à été validé par simulations et a montré son efficacité en la présence de pairs malveillants dans les réseaux Ad-Hoc [39, 48].

2.3.1 Le protocole de confiance

Le principe général du protocole de confiance vise à faire évoluer au cours du temps la relation de confiance entre les noeuds d'un réseau, basée à la fois sur leurs propres connaissances et sur les connaissances des autres. Le protocole de confiance repose sur trois principes : la décentralisation des informations, leur diffusion et la maintenance de l'historique des transactions.

2.3.2 L'architecture du modèle de confiance

Totale décentralisation

L'architecture du système, et l'absence de serveur centralisé a fait que le modèle de confiance est complétement décentralisé. Un nœud du réseau traite localement les informations concernant la qualité de ses transactions avec le reste des noeuds, en utilisant les informations locales et celles reçues par le réseau. Ainsi, les informations qu'un nœud i conserve peuvent être rassemblées sous la forme d'un vecteur V_i, qui est appelé vecteur de confiance. Chaque nœud du réseau possède son propre vecteur de confiance, qui contient les informations de confiance sur tous les noeuds du réseau avec lesquels il a interagi : le $j^{\text{ème}}$ élément $V_i[j]$ du vecteur de confiance, avec $i \neq j$, contient les informations de confiance que le nœud i possède du nœud j (noter que l'élément $V_i[i]$ n'existe pas, car un nœud ne possède pas de notion de confiance sur lui-même).
Les informations de confiance, sont conservées de manière permanente par les noeuds. Lorsqu'un nœud se déconnecte du réseau, il maintient les renseignements qu'il a collecté. Ainsi, lors de sa reconnexion, s'il revient au même endroit (même voisinage), alors il a déjà des données qui lui permettent de juger la confiance qu'il peut mettre en ses voisin. En revanche, s'il se reconnecte à un endroit différent (nouveaux voisins), alors les informations sauvegardées peuvent éventuellement lui donner des informations sur l'état de confiance des ses nouveaux voisins (informations qu'il aura potentiellement reçu par les autres noeuds du réseau). Les noeuds du réseau ne possèdent pas de mémoire partagée, le réseau est le seul moyen permettant le partage d'informations. L'envoi régulier de messages est un point essentiel pour faire évoluer le système.

Les critères de confiance

Comme on l'a vu dans le chapitre 2, chaque système de réputation se base sur plusieurs critères et plusieurs informations pour le calcul de la confiance. Nous allons voir dans ce qui suit, les critères sur lesquels se base le calcul de

confiance ainsi que les informations qui sont maintenues par chaque nœud du réseau (dans le vecteur V_i). Avant qu'un nœud ne décide d'interagir avec un autre noeud il mesure la confiance qu'il peut avoir en lui, en se basant sur deux informations essentielles : sa propre expérience et l'expérience des autres pairs. Grâce à ces informations il peut quantifier la confiance qu'il peut avoir en un nœud, et cette mesure qui est la base de tout le protocole est appelée : *note de confiance*. Les critères sur lesquels est basé le calcul de la confiance sont les suivants :

•La liste des notes de confiance antérieures

Lorsque le nœud i souhaite interagir à l'instant t avec le nœud j, il possède un certain nombre d'informations à son sujet, qu'il a lui même rassemblées durant l'intervalle de temps $[0..t-1]$. Ces informations proviennent des interactions passées du nœud i avec le nœud j. Le nœud i peut se baser sur son expérience personnelle acquise durant les interactions antérieures, pour décider s'il peut interagir de nouveau avec le nœud j. En effet si le nœud i a reçu plus de mauvaises informations que de bonnes informations de la part du nœud j par le passé, il y a de fortes chances que la prochaine transaction soit non bénéfique pour ce premier. Mais la question qui se pose, est combien d'interactions antérieures doivent être prises en compte, pour que l'expérience personnelle soit fiable. L'utilisation de l'ensemble de toutes les transactions antérieures n'est pas très convaincante dans le sens où le nœud j pourrait se comporter de manière correcte pendant très longtemps, jusqu'à ce que la confiance que lui porte i augmente et atteigne la valeur maximale, puis agir de manière malveillante à partir de l'instant $t+1$. A l'inverse, on ne peut pas se contenter de la dernière interaction, car si le nœud j est honnête mais qu'il commet une seule action malveillante alors il sera considéré comme malhonnête. La meilleure solution, serait de borner le nombre d'interactions antérieures, prises en compte, par une valeur Θ_{max}. Donc le nombre d'interactions antérieures Θ est compris dans l'intervalle $[1..\Theta_{max}]$

•La liste de confiance

Un autre critère très important pour le calcul de la confiance, est l'expérience extérieure. Il s'agit des connaissances des autres noeuds avec lesquels le nœud i a déjà interagi dans le passé. Les informations détenues par chaque nœud sur les autres noeuds, sont transférées lors de chaque interaction avec un nœud donné, qui récupère par la même occasion les connaissances du nœud avec lequel il communique (échange des notes de confiance). Une fois récupérée, la liste des notes de confiance est alors appelée liste de confiance.

•Les critères adjacents

Jusque là on a recensé trois critères : les notes de confiance, la liste des notes de confiances antérieures et la liste de confiance. A ceux là s'ajoutent trois autres attributs qui permettent d'affiner les relations de confiance entre les noeuds. On fait correspondre à chaque note de confiance un indice de confiance, qui permet de distinguer entre deux notes de confiances identiques. Un nœud A pourra donc distinguer la qualité d'un nœud B de celle d'un nœud C même si ils ont une note de confiance identique. Ceci en utilisant les notes d'interactions antérieures, en effet si on fixe, par exemple, le nombre d'interactions précédentes à cinq, et qu'on note chacune sur une échelle de 0 à 1 (0 pour les mauvais échanges, 1 pour les bons et 0.5 est le seuil à partir duquel on peut faire confiance à un nœud), on peut estimer qu'un nœud qui a toujours obtenu la même note 0.7 et donc évalué à 0.7 est plus fiable qu'un nœud qui est évalué à 0.7 et qui obtenu les notes suivantes : 1 ; 0.2 ; 0.8 ; 1 ; 0.5, en effet, cet historique montre qu'il a déjà été malhonnête. De même, il existe un indice de confiance pour la liste de confiance qui permet de déterminer si les listes de confiance reçues sont crédibles ou non. A chaque diffusion de la liste de confiance la liste des indices de confiance est transférée aussi.

2.3.3 Gestion de la dynamique du protocole de confiance

Comme nous l'avons vu précédemment un nœud i détient six attributs sur le nœud j, que nous résumons dans ce qui suit :

- la note de confiance : $V_i[j].NC \in [0, 1]$
- l'indice de confiance en la note de confiance : $V_i[j].NCNC \in [0, 1]$
- la liste des notes de confiance antérieures : $V_i[j].LNCA$ avec $V_i[j].LNCA[k] \in [0, 1]$ et $k \in [0, \Theta_{max}]$
- la liste des notes de confiance de j : $V_i[j].LC$ avec $V_i[j].LC[k] \in [0, 1]$ et $k \in [0, \Theta_{max}]$
- la liste des indices de confiance que j possède en ses notes de confiance : $V_i[j].ILC$ avec $V_i[j].ILC[k] \in [0, 1]$ et $k \in [0, \Theta_{max}]$
- l'indice de confiance en la liste de notes de confiance de j : $V_i[j].NCLC \in [0, 1]$

Avant de décrire le fonctionnement du modèle et son initialisation, il est important de noter que les attributs de confiance donnés ci-dessus ont tous une valeur réelle comprise entre 0 et 1. Seules les interactions sont évaluées de manière binaire (0 si mauvaise et 1 si bonne). Les notes données sont donc différentes des notes de confiance, mais le calcul de la note de confiance se base sur les notes données aux interactions. Une note de confiance est néga-

tive si elle appartient à l'intervalle $[0; 0.5[$ et elle est positive si elle appartient à l'intervalle $[0.5; 1]$

Lorsque le réseau est construit, le modèle de confiance est initialisé en fonction des connexions entre les noeuds. Ainsi, lorsque deux noeuds i et j sont voisins, alors les notes de confiance, les indices de confiance en les notes de confiance et en les listes de confiance sont toutes initialisées à la première note positive, soit : 0.5. Les autres attributs ne sont initialisés qu'après la première interaction de i et j.

Le maintien des données et leur évolution

Tous les attributs qui composent le vecteur de confiance, sont mis à jour au fil du temps et en fonction des interactions. Nous allons voir dans cette partie l'évolution de chacun des attributs du modèle à la suite d'une interaction, *i.e*, l'évolution des attributs après le passage du système de son état à l'instant t à un nouvel état à l'instant $t + 1$.

•**Les notes de confiance** Les notes de confiance représentent le point critique du modèle, puisque c'est uniquement en se basant sur cette note qu'un nœud décide si oui ou non il va effectuer une transaction avec un autre nœud. On définit la fonction *Inter* qui renvoie vrai si un nœud i décide d'interagir avec un nœud j et faux sinon :

$$Inter_t(i, j) = \text{vrai} \Leftrightarrow V_{i,t}[j].NC \geq 0.5$$

Comme nous l'avons vu plus haut, la note de confiance est basée sur les connaissances de chaque nœud du réseau. Lorsqu'un nœud i souhaite interagir avec un nœud j, il va se fier en plus de ses connaissances personnelles aux connaissances qu'il a reçues des noeuds avec lesquels il a interagit (noeuds différents de j). Il va donc évaluer ses propres connaissances en se basant sur la note donnée au dernier échange, qu'on notera α, mais aussi sur les les Θ notes des dernières interactions, avec $\Theta \leq \Theta_{max}$. La moyenne notée PK pour *personal knowledges* de ces $\Theta + 1$ notes est calculée comme suit :

$$PK_{t+1} = \frac{\alpha + \sum_{k=1}^{\Theta} V_{i,t}[j].LNCAk}{\Theta + 1}$$

Cette évaluation personnelle du nœud j n'est pas suffisante, c'est pourquoi nous allons ajouter les connaissances des autres noeuds en lesquels i a une forte confiance (80% de confiance en les listes de confiances récupérées chez ses noeuds), qui ont une note négative au sujet de j et qui ont suffisamment confiance en cette note négative (60%). On procède alors au calcul des connaissances extérieures notées EK pour *internal knowledges* comme suit :

$$EPK_{t+1} = \frac{1}{p} \sum_{k=1} S*^{n} V_{i,t}[k].LC[j]$$

où $S*$ est l'ensemble des conditions paramétrables que $V_{i,t}[k]$ doit respecter, défini comme suit :

$$S* = \begin{cases} k \neq i, j \\ V_{i,t}[k].NCLC > 0.8 \\ V_{i,t}[k].LC[j] < 0.5 \\ V_{i,t}[k].ILC[j] \geq 0.6 \end{cases}$$

$$V_{i,t+1}[j] = \frac{\epsilon.[\Theta \times IK_{t+1}] + \zeta.[\Theta_{max} \times EK_{t+1}]}{\epsilon.\Theta + \zeta.\Theta_{max}}$$

où ϵ et ζ sont deux facteurs permettant de paramétrer le poids accordé aux connaissances des autres par rapport aux connaissances personnelles. Plus le facteur ζ sera grand par rapport au facteur ϵ plus vite la perception de la malhonnêteté se fera (y'a néanmoins un certain équilibre à trouver, car au bout d'une certains valeur de ζ la perception sera faussée). L'objectif est de trouver un bon équilibre pour éviter de négliger des connaissances par rapport aux autres.

•**Les indices de confiance en les notes de confiance** L'évolution des indices de confiance en les notes de confiance est déterminée par deux attributs : la note α attribuée à l'interaction qui a eu lieu à l'instant t et la liste des notes des interactions antérieures. Ces indices de confiance sont l'avis personnel sur la qualité de la note de confiance. Considérons les deux fonctions E et σ qui calculent l'espérance et l'écart-type usuels (utilisées dans l'analyse de données statistiques), on calcule $E(V_{i,t}[j].LNCA)$ et $\sigma(V_{i,t}[j].LNCA)$. On fixe un seuil ω_{NCNC} et on vérifie si la note α appartient à l'intervalle :

$$]E(V_{i,t}[j].LNCA) - \omega_{NCNC} \ldots E(V_{i,t}[j].LNCA) + \omega_{NCNC}[$$

En cas d'appartenance, l'indice de confiance est alors incrémenté avec une valeur X proportionnelle à $\sigma(V_{i,t}[j].LNCA)$. Dans le cas contraire, si α n'appartient pas à l'intervalle, alors l'indice de confiance est décrémenté d'un valeur Y aussi proportionnelle à $\sigma(V_{i,t}[j].LNCA)$.

X et Y sont calculés comme suit :

$$X = (1 - \sigma(V_{i,t}[j].LNCA)) \times (\omega_{NCNC} - |\alpha - E(V_{i,t}[j].LNCA)|)$$

$$Y = (1 - \sigma(V_{i,t}[j].LNCA)) \times (\eta|\alpha - E(V_{i,t}[j].LNCA)|)$$

•**Les indices de confiance en les listes de confiance** Les indices de confiance en les listes de confiance sont mis à jour de manière différente, après une interaction ce n'est plus l'attribut $V_{i,t+1}[j].NCLC$ relatif à j qui est mis à jour, mais tous les autres $V_{i,t+1}[k].NCLC$ tel que $k \in [1..n]$ et $k \neq i, j$. A cette seule différence, la méthode de mise à jour est proche de celle expliquée plus haut. On fixe un seuil d'évaluation noté ω_{NCLC}, et on vérifie l'appartenance de chaque $V_{i,t+1}[k].LC[j]$ à l'intervalle suivant :

$$]\alpha - \omega_{NCLC} \ldots \alpha + \omega_{NCLC}[$$

Si $V_{i,t+1}[k].LC[j]$ appartient à cet intervalle, on augmente l'indice de confiance que porte i en la liste de confiance du nœud k avec une valeur X calculée comme suit :

$$X = \omega_{NCLC} - (|V_{i,t+1}[k].LC[j] - \alpha|)$$

Sinon l'indice est décrémenté d'une valeur comprise entre $0,1$ et 0.2 en fonction du résultat de X (*e.g.*il est abaissé de 0.1 si $X = 0$, de 0.2 si $X = 1$).

2.4 Conclusion

Ce chapitre est un état de l'art des différents systèmes de stockage distribués et des différents systèmes de réputation existants. Nous avons présenté les ancêtres des systèmes de stockage pair-à-pair, et cité quelques systèmes de stockage opérationnels existants. Un ensemble de systèmes de réputations permettant la détection des pairs malveillants ont été abordés ainsi que leurs mécanismes de fonctionnement. Dans le prochain chapitre nous allons présenter le système pair-à-pair qui a fait l'objet de notre étude qui est le système de stockage pair-à-pair développé par la société UbiStorage.

Chapitre 3

Le système UbiStorage

3.1 Architecture logicielle

Le système de stockage pair-à-pair est développé et distribué par la société UbiStorage. Le logiciel utilise la technologie pair-à-pair ; chaque pair du système est un nœud du réseau qui correspond à un boîtier physique assignée au client d'UbiStorage (appelé NoeBox). Différents services qui composent le système sont exécutés dans un environnement Linux. Le système est structuré en trois couches principales :

- la couche application : c'est l'intermédiaire entre interface utilisateur et le système, la couche application offre trois primitives à l'utilisateur qui lui permettent de stocker, récupérer et supprimer ses données dans le système : la primitive *Put* qui permet de stocker des fichiers dans le système, la primitive *Get* qui permet de récupérer des fichiers stockés dans le système et la primitive *Delete* qui permet de supprimer les fichiers stockés dans le système.
- la couche de communication : consiste en une table de hachage distribuée (DHT), qui stocke les informations nécessaires pour localiser les noeuds du réseau.
- la couche de routage : utilise un protocole de routage basé sur les clés (identifiants des pairs) qui permet de diffuser les messages dans le réseau.

Chaque pair est identifié par un identifiant unique sur 128 bits appelé *PeerId*, et chaque pair est responsable d'un certain nombre d'identifiants de fichiers appelés *FileIDs* sur (128 bits). Un pair a aussi à sa disposition un nombre d'identifiants de fichiers qu'il doit choisir dans la liste des identifiants non alloués qu'il peut attribuer à ses propres fichiers. Quand un pair souhaite stocker un fichier identifié par FileID, il envoie une requête de stockage au pair qui est responsable de ce FileID (service de méta-information).

3.1.1 La couche application

La couche application offre trois primitives à l'utilisateur pour lui permettre de gérer ses données stockées dans le système. La primitive *Put* est

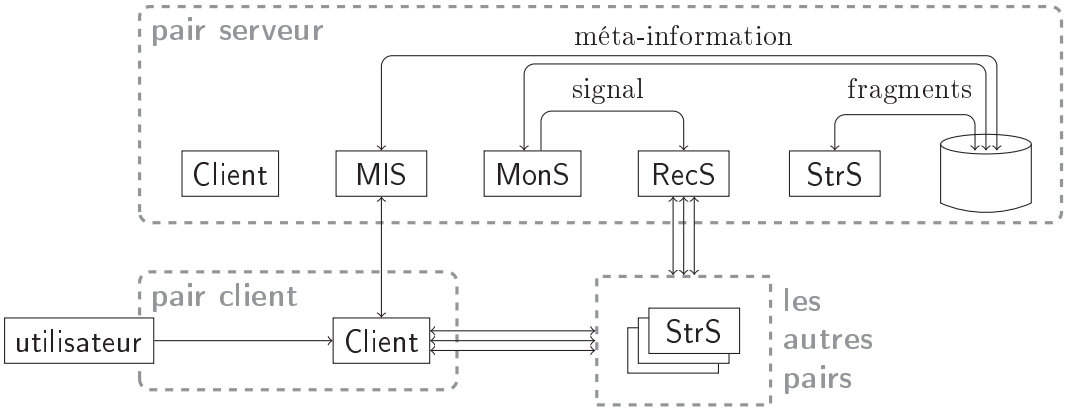

FIGURE 3.1 – Organisation de chaque pair et le protocole de communication avec les autres pairs.

appelée quand un utilisateur veut stocker un fichier dans le système. La procédure de stockage se déroule comme suit : d'abord le fichier est fragmenté en plusieurs blocs de données, chaque bloc est à son tour fragmenté en petits fragments en utilisant le code correcteur Reed-Solomon [43]. L'utilisation des codes correcteurs, permet contrairement à la duplication simple, de gagner de l'espace de stockage et le temps de communication réseau, tout en garantissant la disponibilité des données. Les fragments résultant de cette phase constitue l'unité de base de données qui sont échangées entre les pairs. Le code Reed-Solomon est paramétré avec deux nombres entiers positifs s et r, l'algorithme génère $s + r$ fragments de chaque bloc de fichier, et l'algorithme assure la reconstruction du bloc de fichier à partir de seulement s fragments. Chaque fichier enregistré se voit ainsi attribuer un identifiant unique qui est aussi l'identifiant de chaque fragment de ce fichier. Les informations sur l'emplacement des fragments de chaque fichier, ce qu'on appelle *méta-information de fichier* ou *FileMI*, sont maintenues dans le réseau afin d'être en mesure de récupérer des fichiers, soit pour reconstruire un fichier demandé par un utilisateur lorsque la primitive *get* est invoquée, ou pour assurer que le nombre de fragments nécessaires à la reconstruction d'un bloc de fichier est toujours disponible dans le système. Ce dernier aspect nécessite une surveillance constante afin de détecter la disparition possible de pairs et de déclencher la reconstruction et la redistribution des fragments de fichier avant qu'ils ne deviennent indisponibles.

Chaque pair est organisé comme illustré dans la figure 3.1 pour fournir deux principaux services : gestion des fragments et gestion des méta-informations des données. Les différents services fournis par un pair sont les suivants :

- le *service client* : il est responsable de l'exécution des requêtes Put, Get et Delete en réponses aux requêtes du l'utilisateurs final ;
- le *service de stockage* (StrS : il est responsable du stockage des fragments de données envoyés par d'autres pairs, où la restitution des fragments ultérieurement stockés à la demande du propriétaire des données ;
- le *service de méta-information* (MIS) :il est responsable du maintien des méta-informations relatives aux fragments de données stockés localement. Chaque MIS est responsable d'un intervalle d'identifiants(FileID) et il est le noyau central d'un ensemble pairs qui forment son voisinage dans le système et qu'on appelle *leaf-set*. Chaque leaf-set se charge de stocker les fragments gérés par le MIS. Noter que chaque pair peut faire partie de plusieurs leaf-set et être lui même responsable que d'un seul leaf-set ;
- le *service de monitoring* (MonS) : il est responsable de la surveillance du service de méta-information et envoie un signal au service de reconstruction à chaque fois qu'il trouve un fichier critique.
- le *service de reconstruction* (RecS) : il est responsable de la reconstruction des fichiers lorsqu'il reçoie le signal du service de monitoring et la redistribution des fragments dans le leaf-set après reconstruction.

La récupération d'un bloc de fichier se fait en deux phases : premièrement récupérer les méta-informations relatives au bloc fichier qu'on souhaite récupérer, qu'on appelle *FileMI*, cette méta-information est composée de l'identifiant du fichier FileID ainsi que la liste de tous les *storers* (services de stockages) qui stockent un fragment de ce fichier. Deuxièmement le propriétaire du fichier envoie une requête à chacun des storers de la liste afin de récupérer les fragments de données stockés par ces derniers et qui permettront de reconstruire le fichier par son propriétaire. De la même façon le stockage d'un fichier se fait aussi en deux phases : premièrement le client récupère la liste des storer disponibles pour stocker son fichier. Cette liste est obtenue du service de méta-information qui est responsable du FileID, le MIS renvoie alors la liste des pairs (PeerID) en ligne qui peuvent stocker un fragment de ce fichier. une requête de stockage à chaque storer de la liste avec le fragment à sauvegarder. Après chaque sauvegarde réussie le storer envoie au service de méta-information la méta-information relative au fragment de fichier qu'il a stocké ainsi qu'un acquittement de succès à l'initiateur de l'action de stockage, et de même il envoie un acquittement en cas d'échec.

3.1.2 Les couches de communication et de routage

Les communications dans le système sont basées sur une table de hachage distribuée (DHT) Pastry [32]. Les nœuds du réseau sont organisés en anneau

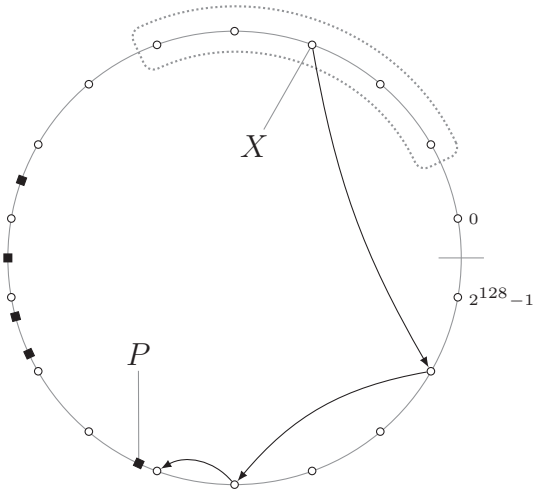

FIGURE 3.2 – Protocole de routage à base de clés.

en suivant un adressage logique sur 128 bits. Chaque pair est identifié dans le réseau logique (DHT) par son identifiant unique PeerID. Pour faciliter la communication chaque pair maintiens plusieurs tables d'indexations qui lui permettent de localiser les autres pairs ; ces tables sont les suivantes :

- *la table de routage* : après chaque communication avec un nouveau pair, le pair met à jour sa table de routage en y ajoutant son adresse IP ainsi que le PeerID correspondant ;
- *table du leaf-set* : une table de voisinage est maintenue par chaque pair, elle est mise à jour à chaque départ ou arrivée d'un pair dans le voisinage ;
- *table des sauts* : les pairs dont les PeerID se trouvent à une distance 2^k (k un entier naturel) sont stockés dans une table de routage secondaire appelés table de sauts. Cette table permet de trouver les meilleures routes à travers la DHT.

Le fonctionnement de la DHT est illustré dans la figure 3.2.

Les identifiants de fichiers FileIDs sont eux aussi adressés sur 128 bits, ils sont gérés par le pair qui a un PeerID qui se rapproche le plus du FileID dans l'anneau (représentés par des carrés sur la figure). Considérons l'exemple du nœud étiqueté par X , son *leaf-set* (voisinage) est représenté par la zone en pointillés et contient tous les pairs voisins (selon l'adressage dans la DHT) et dont les adresses IP sont maintenus dans la table de "leaf-set" du pair. Supposons que le pair à la position X a besoin d'envoyer un message à un autre pair qui se trouve à la position P, pour lui demander une liste de storers

pour un fichier dont l'identifiant est FileID. Il y a deux cas de figures pour l'acheminement du message du pair à la position X au pair à la position X : Le premier cas (non illustré), le nœud à la position P est dans l'une des tables du nœud à la position P , l'expéditeur lui envoie alors directement le message en se servant de l'une de ses tables de routage. Le deuxième cas, quand le pair destination n'est pas référencé chez le pair X, ce dernier se servira des tables de routage pour acheminer le message à travers la DHT : X envoie le message au pair le plus proche de la position P dont il a la visibilité dans sa table de sauts, puis ces pairs intermédiaires transmettront le message en se basant sur le même principe jusqu'à l'arrivée du message à destination. Noter que si le pair censé être à la position P est hors ligne le message sera acheminé au pair en ligne dont la position est la plus proche de P. La table de routage agit en tant que mémoire cache de la DHT, tandis que la table de sauts permet d'accélérer le routage en fournissant des raccourcis à travers la DHT

3.2 Le code auto-correcteur Reed-Solomon

3.2.1 Principe de l'encodage

Le code Reed-Solomon est un code correcteur par bloc, *i.e*, l'encodage d'un bloc de données de taille fixe, transforme en un autre bloc de taille également fixée. Le principe de base de ces codes correcteurs repose sur la théorie des corps de Galois. Il s'agit de construire un polynôme formel à partir des données à transmettre et de le sur-échantillonner. La redondance de ce sur-échantillonnage permet au récepteur du message encodé de reconstruire le polynôme même s'il y a eu des erreurs pendant la transmission. Les codes Reed-Solomon permettent ainsi de corriger deux types d'erreurs :

- les erreurs induisant la modification des données (certains bits passent de 0 à 1 ou vice versa) ;
- les erreurs d'effacements (*i.e*, les pertes d'informations).

Les codes de Reed-Solomon sont formés de n symboles, chacun appartenant au corps de Galois $GF(q)$ à $q = 2^m$ éléments, avec m le nombre de bits par symbole, dont la valeur est généralement 8 (un octet) ou 16. Le nombre de bits par symbole a tendance à suivre l'augmentation de la longueur des mots traités par les processeurs. On trouve des implémentations avec m = 32 ou m = 64. Soient $2 \times t$, le nombre de symboles de contrôle et k, le nombre de symboles d'information (la charge utile). Alors, le nombre de symboles transmis n (charge utile et correction d'erreur) est égal à $k + (2 \times t)$, tel que t représente le nombre de symboles d'erreurs que ce code sera capable de

corriger. Le codage Reed-Solomon ainsi décrit permet de corriger $(n-k)/2$ erreurs [41].

3.2.2 Schéma de redondance et encodage

Dans le code Reed-Solomon, les blocs de données de taille T sont découpés en s fragments de taille T/s. A partir de ces fragments sont calculés r fragments de redondance, également de taille T/s. Les r fragments sont tels qu'à partir de s fragments quelconques parmi les $s+r$ il est possible de reconstituer le bloc initial (de taille T). Soit le bloc de données B vu comme un vecteur de dimension s $B = (b_j), j \in [1..s]$ où les b_j sont les fragments construits à partir de B. A partir de B on compose un vecteur $E = e_i, i \in [1..(s+r)]$ de dimension $s+r$ dont les éléments seront les fragments à distribuer dans le réseau (incluant les r fragments nécessaires à la redondance). L'encodage se fait comme suit :
Le vecteur E est créé à partir d'une matrice A de dimension $(s+r) \times s$, telle que s lignes de la matrice A sont linéairement indépendantes.
Soit $A = a_{i,j}$ tq $i \in [1..(s+r)]$ et $j \in [1..s]$, une matrice de dimension $(s+r) \times s$ dont s lignes sont linéairement indépendantes. Alors le produit $E = AB$ donne un vecteur de dimension $s+r$, et dont les éléments e_i sont les fragments à disperser dans le réseau.

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1s} \\ a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ a_{s1} & a_{s2} & \cdots & a_{ss} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(s+r)1} & a_{(s+r)1} & \cdots & a_{(s+r)s} \end{pmatrix} \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix} = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_s \\ \vdots \\ e_{s+r} \end{pmatrix}$$

3.2.3 Reconstruction du bloc

Le vecteur B peut être reconstruit à partir de s fragments quelconques de E. Soit E' le vecteur des s fragments utilisés pour la reconstruction. On construit la matrice A' à partir des lignes de A qui correspondent aux fragments de E' (A' est une matrice carrée de dimension $s \times s$). Par construction de E, on déduit que :

$$A'B = E'$$

Ce qui nous intéresse, est de retrouver B à partir de A' et de E'. Par hypothèse, on a s lignes de A sont linéairement indépendantes, il en est donc de même pour A'. A' est inversible donc nous avons :

$$A'^{-1}E' = B$$

$$\begin{pmatrix} a'_{11} & a'_{12} & \cdots & a'_{1s} \\ a'_{21} & a'_{22} & \cdots & a'_{2s} \\ \vdots & \vdots & \ddots & \vdots \\ a'_{s1} & a'_{s2} & \cdots & a'_{ss} \end{pmatrix}^{-1} \begin{pmatrix} e'_1 \\ e'_2 \\ \vdots \\ e'_s \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{pmatrix}$$

3.3 La réplication dans la DHT

Dans une DHT, un identifiant (clé) est associé à chaque pair et à chaque bloc de données. La clé d'un bloc de données est obtenue par l'application d'une fonction de hachage sur le contenu du bloc. Le pair dont l'identifiant est le plus proche de la clé du bloc est appelé racine du bloc (c'est le pair qui est en charge du stockage du bloc et de sa surveillance). Chaque pair a une connaissance locale du réseau, *i.e*l'ensemble des pairs qui constituent son voisinage dans l'anneau et qu'on appelle leafset. Un leafset a une taille fixe L, chaque pair racine a donc $L/2$ voisins dans le sens positif de l'anneau et $L/2$ voisins dans le sens négatif. Un pair maintien à jour son leafset en supprimant les pairs déconnectés et en ajoutant les nouveaux pairs. Chaque pair possède un leafset, et fait partie de plusieurs leafsets. Afin daugmenter la disponibilité des données dans les réseaux pair-à-pair dont le départ et l'arrivée des pairs est aléatoire, les blocs de données sont fragmentés et répliqués en utilisant les code-correcteur reed-solomon et ensuite distribué sur le réseau. Chaque fragment de donnée est identifié par un identifiant unique propre à lui ainsi que l'identifiant du bloc auquel il appartient. Les k $(s+r)$ fragments générés seront stockés sur k pairs du leafset du pair racine qui est en charge de l'identifiant du bloc (dont l'identifiant est le plus proche de l'identifiant du bloc). Le pair racine est responsable du placement initial des fragments dans son leafset et de leur maintenance. Le système UbiStorage a deux stratégies dans le choix des pairs du leafset où stocker les fragments : choix de pairs de manière aléatoire ou bien des pairs contigüs. Chaque pair du leafset qui stocke un fragment de la donnée, envoie les méta-informations relatives au fragment qu'il a stocké au pair racine (qui joue le rôle d'un service de méta-information) en y indiquant son identifiant ainsi qu'identifiant du fragment stocké et son rang dans le bloc. Ces méta-informations sont mises à jour en fonction des départs et arrivées des pairs dans le leafset. Quand le service de monitoring constate que le nombre de fragments restants dans le système a atteint un nombre critique, il sollicite le service de reconstruction pour reconstruire le bloc de données et le redistribuer dans le système. La reconstruction permet de récupérer le bloc original à partir de s fragments

seulement, et avec le code correcteur reed-solomon $s + r$ fragments sont de nouveau générés et puis redistribués sur le leafset.

3.4 Conclusion

Dans ce chapitre, on a présenté le système de stockage pair-à-pair UbiStorage et son fonctionnement. On aussi présenté un système de notation sur lequel on s'est reposé pour la proposition d'un système de détection de pairs malveillants. Dans le prochain chapitre nous allons décrire des attaques implémentées pour l'analyse de la robustesse du système de stockage en la présence de pairs malveillants, ainsi que la solution proposée pour la détection de ces derniers.

Chapitre 4

Contributions à la sécurité

4.1 Introduction des pairs malveillants

Dans cette section, nous allons présenter plusieurs comportements malveillants que peut avoir un pair dans le système. Ces attaques sont possibles par exemple, quand un client arrive à prendre le contrôle sur sa NoeBox, ainsi dans ce qui suit nous allons voir comment une ou plusieurs NoeBoxes corrompues peuvent influencer le comportement du système global. Il y a plusieurs scénarios d'attaques possibles. Nous nous sommes focalisé sur celles qui peuvent avoir un impact sur la disponibilité des données, propriété très critique dans les systèmes de stockage de données. En effet les travaux de [Sandjabi & al] ont démontré que l'utilisation du chiffrement permet de minimiser (voir rendre nulle) les effets des fs de type *man in the middle* ou une intrusion dans la NoeBox. Une des attaques qu'on a écartée de notre analyse est : la reconstruction illégitime des fichiers : un pair malveillant essaie de reconstruire de manière illégitime les données des clients quand il arrive à rassembler assez de fragments du même fichier. Ce comportement n'a pas été pris en compte, car après notre analyse, on a conclu que cette attaque ne pouvait se produire que dans les très petits réseaux, ce qui est très loin du cas réel, en effet dans un grand réseau les fragments sont distribués de manière à ce qu'il est statistiquement difficile d'en obtenir assez pour tenter une reconstruction. En outre, les fichiers fragmentés sont chiffrés en utilisant un algorithme de chiffrement sécurisé ce qui rend inutile une reconstitution. On a pu distinguer quatre comportements malveillants :

- service de stockage malveillant : il ment à propos du résultat de la sauvegarde d'un fragment de données ;
- service de méta-informations malveillant : il renvoie de fausses méta-informations ;
- service de reconstruction malveillant : il ignore les alertes du service de monitoring et ne reconstruit pas les fichiers critiques ;
- service de monitoring malveillant : il n'alerte pas le service de reconstruction lorsqu'il trouve des fichiers critiques.

Pour notre analyse nous n'avons retenu que les trois premiers cas, et le dernier cas n'a pas été traité car les résultats sont prévisibles.En effet, l'ordre d'in-

térvention des deux service lors du processus de recontruction est séquentiel,(c'est le service de monitoring qui signale une reconstruction au service de reconstruction),donc un service de monitoring malveillant est équivalent à un service de reconstruction malveillant : la reconstruction ne se fait pas dans deux cas. Comme discuté ci-dessus, la principale menace est la perte de fichiers dans les trois cas considérés.

4.1.1 Service de stockage malveillant

Deux comportement peuvent être envisagés pour le service de stockage malveillant : prétendre qu'il a stocké un fragment alors qu'il ne l'a pas fait, ou bien prétendre qu'il n'a pas stocké un fragment alors qu'il l'a fait. Dans le premier cas, le storer peut aussi stocker le fragment et lorsque son propriétaire demande à le récupérer, le storer remplace le vrai fragment du fichier par un faux fragment qu'il génère lui même et le renvoie au client. Dans les deux cas, le storer peut essayer de rassembler des fragments pour la reconstruction des blocs fichier, néanmoins le deuxième comportement malveillant est la meilleure stratégie pour pouvoir récolter des fragments du même fichier. En effet, lorsqu'un storer prétend ne pas avoir sauvegardé un fragment du fichier, il peut être candidat à la sauvegarde lors de la reconstruction de ce fichier (maintenance des données par le service de reconstruction). Des reconstructions malveillantes (illégitimes) ont été observées lors de nos simulations dans des petits réseaux, mais comme on l'a déjà expliqué dans notre le cas réel la taille du réseau rend la probabilité que cela se produise très faible, et même si cela arrivait à se produire l'utilisation du chiffrement rendrait la reconstruction inutile au pair malveillant car il ne pourrait rien apprendre des données récoltées. Notre travail s'est centré sur les comportements malveillants qui affectent la perte de données, qui est une propriété très critique d'un système de stockage de données et qui est la principale inquiétude des clients (retour des clients via le support client dont je fais partie).

Faux succès lors du stockage (attaque A1)

De toute évidence, ce premier comportement malveillant peut entraîner des pertes de fichiers. Lorsqu'un storer est sollicité pour stocker un fragment, le service de stockage prétend qu'il a réussi, mais en fait le fragment est stocké dans une mémoire secondaire (hors de l'espace de stockage dédié aux fichiers clients), mais ne sera jamais renvoyé à la demande, ou, de façon équivalente, un fragment non valide est renvoyé à la place (ce qui est plus difficile à détecter comme un comportement malveillant, donc nous l'avons choisi comme cas à mettre en œuvre). Si au moins $r + 1$ storers trichent de

cette façon pour un fichier donné, il ne sera jamais possible de reconstituer le fichier, car on aura à disposition moins que les s fragments nécessaires à maintenir le fichier disponible dans le système. Il est assez difficile de protéger le système contre une telle attaque, en particulier si elle se produit au cours d'une reconstruction déclenchée par le service de monitoring. En effet, on peut imaginer qu'après une action de "Put", le client peut garder le fichier un certain temps et, avant de déclarer un succès, il peut envoyer des *challenges* (défis) aux storers pour s'assurer que les fragments sont effectivement stockés ; par exemple le client envoie aux storers des requêtes de calcul d'un nouveau hachage du fragment stocké associé à un nonce aléatoirement choisi par le client ; les fonctions de hachage n'étant pas bijectives le storer ne peut pas réutiliser un hachage précédent du fragment de données pour générer le nouveau demandé. Mais après que l'action "Put" a réussi, le fichier est supposé être stocké en toute sécurité sur le système et est susceptible d'être supprimé de l'ordinateur de l'utilisateur. Ainsi le client attache aux fragments un ensemble de challenges pré-calculés qui pourront être utilisés par le service de méta-information pour vérifier que les bons fragments sont toujours stockés par les storers.

Faux échec lors du stockage (attaque A2)

Lorsqu'un storer est sollicité pour le stockage d'un fragment d'un fichier donné, le service de stockage peut, de manière malveillante, prétendre que le stockage a échoué. Par conséquent on demandera jamais de fragment à ce storer, car il n'est pas supposé le détenir. Si un grand nombre de services de stockage se comportent de la sorte, une action de stockage "Put" échouera nécessairement car il n'y aura pas assez de storers nécessaires pour stocker un fichier. Cela ne constitue pas en soi un problème de sécurité, mais si ce comportement malveillant se produit lors de la reconstruction d'un fichier, elle échouera et cela mènera à la perte du fichier, chose qui est désormais notre principale préoccupation. Noter que ce comportement malveillant est assez différent des échecs de stockage qui peuvent se produire à cause d'un disque plein ou d'une panne de disque. L'échec lié à des problèmes de disque peut être résolu, en augmentant l'espace de stockage total, qui lui peut être contrôlé à partir d'un point de vue global. On peut aussi ajouter des NoeBoxes dans le système qui pourront servir au stockage ou bien ajouter des disques supplémentaires sur les NoeBoxes déjà existantes. Comme le comportement malveillant précédent, celui-ci est difficile à combattre car il ne peut être détecté que lorsqu'il est trop tard (perte de fichiers produite). Ainsi, dans les deux cas il est important de mesurer la résistance du système contre les attaques du service de stockage.

4.1.2 Service de méta-information malveillant (attaque B)

Un service de méta-information est responsable du stockage et du maintient des informations relatives à la localisation des fragments d'un fichier donné dans le système (quel storer stocke quel fragment). Un service de méta-information malveillant peut intentionnellement détruire ou corrompre ces informations (lors du stockage ou lorsque le client sollicite les méta-informations). Nous avons mis en place un service de méta-information malveillant qui renvoie des méta-informations (FileMI) corrompues à chaque requête du client. Ce comportement est probablement le plus difficile à détecter car on a toujours la vision d'une service de méta-information qui se comportement correctement. Bien sûr, ce comportement conduit à la perte d'accès immédiate aux fichiers : les fragments du fichier sont sur le système mais impossible à localiser.

Ce comportement malveillant est en quelque sorte moins critique que les précédents, car les méta-informations ne représentent pas une grande quantité de données et peuvent être facilement dupliquées. En fait, c'est déjà le cas en quelque sorte, car une exploration du système peut permettre de retrouver les fragments perdus et de reconstruire la méta-information. En effet, chaque service de stockage détient un fragment de donnée et un fragment de la méta-information qui le concerne (FragMI), il est donc possible de récupérer les fragments de méta-information et reconstruire le FileMI complet qui permet la récupération du fichier ; mais cette contre-mesure a un coût très important qui doit être considéré. On ne peut évaluer ce coût qu'une fois que la résistance du système à ce comportement malveillant est connue.

4.1.3 Service de reconstruction malveillant (attaque C)

La reconstruction de fichier survient lorsqu'un nombre de pairs détenteur d'un fragment d'un fichier donné sont déconnectés (un nombre de fragments insuffisant dans le système), ce nombre de fragments est déterminé au départ et fixé, il est appelé seuil de criticité. Le nombre de fragments restant pour le fichier est surveillé par le service de monitoring, ce dernier fait des tests réguliers et dès qu'il constate qu'un fichier a atteint un état critique il le signale au service de reconstruction qui se charge de reconstruire le fichier (à partir des fragments restants) et de le redistribuer à d'autres services de stockage. Ces deux services sont complémentaires, et sont d'une importance capitale dans le système. Si l'un de ces services est malveillant et ne fait pas son travail correctement, les fichiers seront inévitablement perdus. Pour lutter contre de telles attaques, la surveillance et la reconstruction peuvent être

dupliqués, *i.e*, plusieurs pairs dans un leaf-set peuvent être simultanément responsables de l'exécution de ces tâches. Mais, comme dans le cas précédent, cela induit un coût et une coordination supplémentaires entre les services redondants, ce qui augmente la complexité du système.

4.2 Détection des pairs malveillants

Dans cette section nous allons présenter le système de réputation qui sera implémenté dans le système de stockage UbiStorage pour la détection des pairs malveillants. La solution que nous avons proposée est l'implémentation d'un système de détection de pairs malveillants basé sur la réputation des pairs dans le système.Avant de se lancer dans une implémentation coûteuse d'un tel système, nous avons d'abord analysé par simulation ce que pourrai apporter un système de notation simple à notre système ; et les résultats ont été assez concluants 5. Dans un premier temps nous allons implémenter et évaluer par simulation et model-checking.un système de notation (évaluation des transactions entre pairs) à un seul niveau. Nous allons étendre le système de notation pour qu'il devienne un système de réputation, en ajoutant un deuxième niveau de notation comme dans [39, 48] : le premier niveau permet de construire une réputation en attribuant des notes aux transactions faites entre les pairs, et le second niveau attribut des notes de confiance aux notes des transactions. Pour la construction de la réputation un pair se basera sur sa connaissance personnelle ainsi que la connaissance des autres pairs. Comme montré dans [39, 48], l'utilisation de deux niveaux de réputation permet une détection rapide des comportements malveillants ainsi que les pairs qui protègent des pairs malveillants. Comme nous allons le voir dans les prochains chapitres, la présence de pairs malveillants dans le système provoque non seulement la perte de données, mais certains comportements (attaque B) provoquent une exclusion de pairs honnêtes injustement en les faisant passer pour des pairs malveillants au près des clients. L'utilisation de deux niveaux de confiance nous permettra de palier à ce problème de faux positifs à un certain niveau (intuition à confirmer par model-checking). Nous proposons deux solutions additionnelles, pour résoudre le problème des faux positifs. La première consiste à partager les mauvaises notes entre tous les pairs participants à une opération de "Get", soient : tous les services de stockage et le service de méta-information. L'attribution des notes sera en fonction de la réputation de chacun des acteurs incriminés, pour éviter qu'une coalition de services de stockages malveillants mènent à l'exclusion d'un service de méta-information honnête et inversement. La deuxième solution, serait de modifier le protocole de l'opération "Get" de manière à ce que

le service de méta-information puisse fournir une preuve avec le fichier de méta-information transmis au client. Le service de méta-information devra prouver que chacun des storers contenu dans la liste fournie au client stocke un fragment de la donnée à restaurer. Cette preuve pour être obtenue en utilisant la signature de chaque fragment stocké par le storer qui le stocke. A chaque publication des méta-informations le storer devra signer l'information transmise. Ce mécanisme évitera au service de méta-information de transmettre une fausse liste de storers sous peine d'être détecté, car il impossible de produire de fausse signatures.

4.3 Formalisme ABCD

Dans cette section nous allons décrire le formalisme utilisé pour la modélisation du système UbiStorage. Le formalisme ABCD [11 sec 5.2] est une algèbre de réseaux de Petri colorés *i.e*, une algèbre de -processus avec une sémantique basée sur les réseaux de Petri colorés. Dans notre cas le domaine de couleurs est le langage Python [12] *i.e*, les données et les calculs sont exprimés en utilisant le langage Python.

Un modèle ABCD peut être composé de plusieurs éléments qui sont :

1. une série (éventuellement vide) de déclarations, chaque déclaration peut être :
   - définition d'une fonction Python
   - l'instruction Python "import" d'un module ou d'une fonction
   - la définition d'un buffer ABCD (conteneur de données)
   - la définition d'un ou plusieurs "sub-net" ABCD (similaire à des sous programmes)
2. un processus ABCD (similaire à la fonction "main" d'un programme en C).

Comme dans le langage Python, un modèle ABCD est une imbrication de blocs et de sous blocs (net et sub-net), dont la hiérarchie est mise en évidence en utilisant l'indentation.

4.3.1 Syntaxe du formalisme ABCD

La syntaxe d'ABCD est un mélange entre le langage Python et une algèbre de -processus.

1. Définitions en Python Les déclarations de fonction et les instructions d'importations sont similaires à celles de Python. La définition des classes n'étant pas possible, il est nécessaire de créer le module Python

dans un autre fichier séparé et importer son contenu dans le modèle.
Exemples de définition en Python :

```
1
2 from foo import *
3 from foo import too
4 import math
5 def sqrt(x):
6     return int (math.sqrt(x))
```

2. Définition de buffer Un buffer en ABCD est la seule structure de donnée utilisée, il joue le rôle de la mémoire. Un buffer en ABCD est représenté dans la sémantique des réseaux de Petri par une place Un buffer est :
   - typé : les éléments contenus dans un buffer peuvent être de plusieurs types (entier, réel, booléen, chaîne de caractères...etc), le type objet (le type universel en Python) permet mettre des contenus de n'importe quel type dans un buffer.
   - non borné : à priori il n y a aucune limite quant à la taille d'un buffer, au nombre d'éléments qui peuvent être mis dedans et aux nombre de copies d'un même élément contenu dans un buffer.
   - non ordonné : l'ordre dans lequel les valeurs d'un buffer sont récupérées n'est pas déterministe et n'est en aucun lié à l'ordre dans lequel ces éléments ont été mis dans le buffer.

   Exemples de définition de buffer : La syntaxe de la déclaration d'un buffer est la suivante :

```
1 buffer Name : Type = Initialisation
```

   où *Name* est le nom du buffer , *Type* est le type de son contenu et *initialisation* est la valeur initiale du buffer.

```
1     buffer count : int = 0
2     buffer count : init = () # initialis\'e \'a l'ensemble vide
```

3. Définition des sous-blocs "sub-net" un "sub-net" peut être assimilé à un sous programme dans les langages de programmations classiques (la différence réside dans le fait que l'exécution d'un sous-programme font appel à la même instance de celui-ci et exécutent les mêmes instructions, quand à un sub-net à chaque appel une nouvel instance est créée), la syntaxe de la déclaration d'un "sub-net" est la suivante :

```
1 net Name (Params)
2     Bloc
```

   Où *Name* est le nom du "sub-net", *Params* une liste de paramètres et *Bloc* est une spécification ABCD qui peut contenir un ou plusieurs *Blocs* et des définitions optionnelles et des -processus .

4. Définition de -processus Un -processus ABCD est défini comme un terme d'une algèbre de -processus dont les opérateurs sont des opérateurs de contrôle de flot. Les -processus sont composés d'actions atomiques et/ou des actions composées par les différents opérateurs de contrôle de flot : (';' pour la séquence , '*' pour l'itération , '+' pour le choix et '|' pour le parallélisme). Une action atomique est composée d'une suite d'accès aux buffers et de gardes.

Exemples d'actions atomiques :

```
1
2 [command-('up'), state-('closed'), state+('moving')]
3
4 [count-(x), count+(x+1), shift?(j), buf+(j+x) if x<10]
```

La figure 4.1 montre un fragment de notre modèle. Le symbole "• • •" montre des parties que nous avons omis de montrer dans l'exemple pour une meilleure clarté. Commençons par les déclaration de `net`, un "net" modélise un -processus (programme, fonction, sous programme...etc). Comme en Python la hiérarchie dans le code est représenté par l'utilisation de l'imbrication, on peut ainsi faire la déclaration du programme principal `UbiSystem` (équivalent du main lignes 3–69) , composé par deux sous programmes déclarés par : `Peer` (lignes 9–66) and `User` (lignes 68–69). Les "nets" `Peer` et `User` sont utilisé pour modéliser un nœud du réseau P2P ainsi que son utilisateur. A la fin du "net" `UbiSystem`, une seule instance de `Peer` et une seule instance de `User` sont composée en parallèle (ligne 71) pour définir le comportement du -processus `UbiSystem`. La déclaration du `Peer` est composée d'un ensemble de déclarations des différents services qui tournent au sein d'un pair (services présentés dans la figure 3.1), excepté les services de reconstruction et de monitoring qui n'ont pas été pris en compte car on a mis comme hypothèse que le réseau est stable et qu'il n'y a pas de départs/arrivées de nouveau pairs ce qui implique que la reconstruction et le monitoring n'est pas nécessaire. Mais comme on peut voir dans le modèle le -processus de routage à été modélisé par le "net" `Router`, c'est le modèle qui représente le sous programme responsable de l'acheminement des messages dans le réseau P2P. Un autre service qu'on n'a pas pris en considération est le service de suppression des fichiers, sa similitude avec le -processus de récupération de fichiers fait qu'on ne l'a pas modélisé. En effet le -processus de suppression consiste à faire une requête recherche de recherche de fragments de données, la seule différence entre les deux -processus et que contrairement à la récupération les fragments de données sont supprimés au lieu d'être renvoyés à l'utilisateur.

Un `Peer` est composé de deux sous-processus : le -processus de routage `Router` et d'un -processus client `Client`. Le -processus de routage a été modélisé comme

```
buffer nw : object = ()

net UbiSystem(n,this, initfiles, initmeta
          , initdata, initstorers, tasks,sstype,mistype):
    buffer params : str*int = ()
    buffer notes : (int*int*float) =
                  [(this,y,0.5)  for y in range(6)]

    net Peer () :
        buffer sessionid : int = 0

        net Router () :
            [nw-(s,this,d,m), nw+(s,d,m)]

        net Client () :

            buffer files : int*tuple(str) = initfiles
            buffer Fileid : int = ()

            net GetClient ():
                buffer storers : tuple(int) = ()
                buffer wait : int = ()
                buffer cptr : int = ()
                buffer frags : int*
                      (tuple((str)|enum(None))) = ()
                buffer fid : int = ()
                buffer getsession : int = ()


                [params-(("Get",Fid)),
                 fid+(Fid),
                 sessionid-(ids),getsession+(ids),
                 sessionid+(ids+1),
                 frags+(Fid,(None,)),
                 nw+(this,(Fid-1)% n ,Fid%n
                    ,(Fid,ids,"GetFileMImessage"))]

                ; [getsession?(ids),fid?(Fid),
                   nw-(src,this,(Fid,ids,strs)),
                   storers+(strs)]

                ; [getsession?(ids),fid?(Fid),
                   storers-(strs),
                   wait<<(strs),
                   cptr+(len(strs)),
                   nw<<([(this,s,(Fid,ids
                      ,"GetFileMessage")) for s in strs])]

                ;[fid-(Fid),Fileid+(Fid)]
                ; ●●●
            net PutClient () :
                ●●●
            (GetClient() + PutClient()) * [False]

        net Storer () :
            ●●●
        net MIService () :
            ●●●
        (Client() * [False])
        | (Storer() * [False])
        | (MIService() * [False])
        | (Router() * [False])

    net User () :
        [params<<([t for t in tasks if t[0] != "NOP"])
         if [t for t in tasks if t[0] != "NOP"]]

    Peer() | User()

Ubi0::UbiSystem(6,0,
                (),
                (),
                ((1,"1.1"),),
                ((1,5),),
                (("NOP",0),),
                (0),
                (0)
                )
|Ubi1::UbiSystem(●●●)
```

FIGURE 4.1 – Extrait du modèle ABCD du système UbiStorage.

une abstraction du routage réel : comme montré dans le modèle (ligne 13), un message destiné à un pair k est envoyé via le pair $k-1$ qui lui le transmettra au pair k, de ce fait on a juste pris en compte une seule étape de routage alors quand dans le routage réel le nombre d'intermédiaires peut être quelconque. L'abstraction permet en effet de réduire le nombre de combinaisons possibles lors de la génération de l'éspace d'état (différents nœuds possibles pour le routage) ; ainsi la combinatoire est réduite et on évite une explosion. Le -processus client `Client` est lui même composé de plusieurs sous-processus : `GetClient` et `PutClient` qui modélisent respectivement les primitives "Get" et "Put", et la primitive "Delete" a été ignorée pour les raisons expliquées précédemment. Le "main" du -processus client est une choix entre les deux sous-processus "GetClient" et "PutClient" qui sont exécutés infiniment.

A la fin du -processus `Peer` (lignes 63–66) sont instanciés ses sous-processus, chacun est exécuté dans une boucle infinie et ces boucles sont composées en parallèle. Par exemple, $Client() * [False]$ est une boucle qui permet d'exécuter une instance du -processus Client infiniment, la sortie de boucle n'étant possible que lorsque l'action [False] est exécutée , ce qui n'est pas possible en ABCD car l'action [False] est bloquante.

Maintenant intéressons nous aux paramètres des -processus ainsi qu'à la déclaration des buffers. Le -processus `UbiSystem` a une liste de paramètres qui sont : `n` le nombre de pairs dans le système, `this` l'identité du pair qui est instancié dans la déclaration, `initfiles` les fichiers présents dans le système avant l'exécution , `initmeta` les méta-données initialement possédées par le service de méta-information pair, `initdata` les données sauvegardées initialement par le service de stockage, `initstorers` la liste de storers dans le leaf-set du pair instancié, `tasks` la liste des actions de "Get" et "Put" que va initier l'utilisateur lors de l'exécution, `sstype` le type du service de stockage correct/malveillant et `mistype` le type du service de méta-information. A la ligne 5, un buffer `params` est déclaré, il est de type `str*init` qui indique le produit cartésien entre l'ensemble des chaînes de caractères `str` et l'ensemble des entiers `int`, $i.e$, le buffer `params` contiendra des pairs dont le premier élément est une chaîne de caractères et dont le second élément est un entier naturel. Le buffer `params` est initialement vide (comme montré par l'affectation de l'ensemble vide "`=()`"). Le buffer de paramètres est rempli par le -processus `User` ligne 69 : une action atomique $params \ll tasks$ soumise à une condition $ift[0]! = "NOP"$ qui rempli le buffer par toutes les pairs dont le premier élément est différent de l'action muette "Nop" (noter qie "Nop" n'est pas une action du formalisme ABCD, mais une convention du modèle pour modéliser l'action vide).

Si on examine bien le -processus de routage `Router` et le buffer `nw`, on remarque que ce dernier sert à modéliser le réseau, les messages relayés sont sous forme d'un tuple (s, r, d, m) où s est la source du message, r est le pair

relai (routeur), d la destination finale du message et m le message. Lorsque la destination finale est connue par l'expéditeur le message envoyé est sous la forme d'un tuple (s, d, m), le message est ainsi produit par l'expéditeur dans le buffer `nw` et consommé directement par le destinataire sans qu'il passe par un pair relai (routeur). De manière générale, un pair reçoit toujours un message provenant d'un routeur et la réponse est envoyée directement à l'-expéditeur du moment que son adresse se trouve dans le message relayé. Un routeur `Router` dont l'identité `this` est r dans le message consomme de manière répétitive des 4-uplet (s, d, r, m) du buffer `nw` et remet un 3-uplet (s, d, m) à sa place, cette action modélise le -processus de routage et elle correspond dans le modèle à l'action atomique de la ligne 13 , où `nw-(···)` représenté une consommation à partir du buffer `nw` et `nw+(···)` est une production d'une valeur dans ce buffer.

4.3.2 Modèle du système UbiStorage

Le formalisme ABCD ayant été introduit plus haut, nous allons maintenant présenter le modèle entier du système UbiStorage en commentant chaque service et sa modélisation.

Client

Comme on l'a vu dans la section précédente, le service client est le service qui prend en charge les primitives "Get", "Put" et "Delete" initiées par l'utilisateur (User). Dans notre modèle nous n'avons pris en compte que les deux primitives "Get" et "Put" que nous allons détailler dans ce qui suit.

La figure 4.2 représente le modèle de la primitive "Get" exécutée par le service client pour la récupération des fichiers sauvegardés dans le système. Le sous-processus `GetClient` est une série de déclarations de buffers et une séquence de plusieurs actions atomiques. Les buffers déclarés dans ce sous-processus sont locaux et ne sont accessibles que dans cette partie du modèle, chaque buffer a une utilisation bien spécifique :

- `storers` est typé comme un n-uplet d'entiers qui contient la liste des storers susceptibles de stocker un fragment de fichier (storers envoyés par le service de méta-information),
- `wait` de type entier contient la liste des pairs dont on attend une réponse,
- `cptr` de type entier est utilisé comme compteur,
- `frags` dont chaque élément est un tuple composé d'un premier élément de type entier et le second élément est un tuple de chaînes de caractères ou la chaîne de caractère "None",

```
        net Client () :
            buffer files : int*tuple(str) = initfiles
            buffer Fileid : int = ()

            net GetClient ():
                buffer storers : tuple(int) = ()
                buffer wait : int = ()
                buffer cptr : int = ()
                buffer frags :
                      int*(tuple((str)|enum(None))) = ()
                buffer fid : int = ()
                buffer getsession : int = ()

                [params-(("Get",Fid)),
                 fid+(Fid),
                 sessionid-(ids),getsession+(ids),
                 sessionid+(ids+1),
                 frags+(Fid,(None,)),
                 nw+(this,(Fid-1)% n ,Fid%n
                    ,(Fid,ids,"GetFileMImessage"))]

                ; [getsession?(ids),fid?(Fid),
                   nw-(src,this,(Fid,ids,strs)),
                   storers+(strs)]

                ; [getsession?(ids),fid?(Fid),
                   storers-(strs),
                   wait<<(strs),
                   cptr+(len(strs)),
                   nw<<([(this,s,(Fid,ids,"GetFileMessage"))
                            for s in strs])]

                ;[fid-(Fid),Fileid+(Fid)]

                ;(([nw-(src,this,(Fid,ids,frag)),
                    wait-(src),getsession?(ids),
                    frags-((Fid,(None))),
                    frags+((Fid,(frag,))) ,
                    cptr-(c), cptr+(c-1) if c > 0]
                   +[nw-(src,this,(Fid,ids,"ko")),
                     wait-(src),getsession?(ids),
                     cptr-(c), cptr+(c-1) if c > 0])
                   ;(([nw-(src,this,(Fid,ids,frag)),
                      wait-(src),getsession?(ids),
                      frags-((Fid,s)),
                      frags+((Fid,s+(frag,))) ,
                      cptr-(c), cptr+(c-1) if c > 0]
                      +[nw-(src,this,(Fid,ids,"ko")),
                        wait-(src),getsession?(ids),
                        cptr-(c), cptr+(c-1) if c > 0])
                    * [cptr-(0)]))
```

FIGURE 4.2 – Extrait du modèle ABCD du sous-processus `getclient` qui modélise l'opération "Get".

- `frags` sert à sauvegarder les fragments de données reçus lors de la requête "Get" pour pouvoir les reconstruire durant le -processus de récupération,
- `fid` de type entier qui contient les identifiants des fichiers et enfin `getsession` de type entier qui contient les identifiants de session pour chaque action "Get".

Le déroulement d'une action "Get" est le suivant :

- ligne 13 , le client récupère la liste des actions à faire à partir du "params" qui est au préalable rempli par l'utilisateur, donc une paire (dont le premier élément est la chaîne de caractère `"Get"` et le second élément est lié à la variable `Fid`) est consommée du buffer `params`, cette action de consommation modélise la réception de la requête "Get" de la par de l'utilisateur "User".
- (ligne 14–19) cette partie décrit la réaction du client à la réception de la requête "Get", il commence par sauvegarder l'identifiant du fichier que le "User" souhaite récupérer dans le buffer `Fid`, récupère l'identifiant de la session précédente à partir du buffer `getsession` et incrémente ce dernier pour le remettre dans `getsession` pour courante action de "Get", initialise le buffer `frags` avec l'identifiant du fichier comme premier élément et met la chaine "None" dans le second élément et enfin envoie un message de requête des méta-informations au service de méta-information qui gère ce fichier (dont l'identifiant se rapproche de $Fid\%n$), en passant par l'intermédiaire d'un pair dont l'identifiant est calculé avec la formule : $(Fid-1)\%n$. Le message envoyé contient : la source $this$, le routeur $(Fid-1)\%n$, la destination finale $Fid\%n$ et le message en lui même composé de : l'identifiant du fichier Fid, l'identifiant de session ids et le message de demande des méta-informations $GetFileMIMessage$.

Le message envoyé par le client sera routé et relayé au service de méta-information qui gère le fichier dont l'identifiant est Fid, ce service, dès réception de la requête du client, renvoie une réponse direct au client avec la liste des "sotrers" qui stockent un fragment du fichier demandé. La réponse est consommée à partir du buffer `nw` par le client (ligne 21–23) :

- ligne 21, l'identifiant de fichier est récupéré sans être consommé à partir du buffer `Fid`, et de même pour l'identifiant de session afin qu'ils puissent être comparés aux informations contenues dans la réponse (afin de s'assurer que la réponse reçue est celle qui est attendue) ; ligne 22 le message est reçu et consommé à partir du buffer `nw` (le réseau), le message contient l'identifiant du fichier, l'identifiant de session et la liste des storers attendus ; cette liste est sauvegardée dans le buffer `storers`(ligne 23).

A la réception de la liste des storers, le client envoie une requête de récupéra-

tion de fragments de données à chacun des storers de la liste, enfin le -processus client procède à la reconstruction du fichier à partir des fragments obtenus :

- ligne 25–30, le client récupère la liste des sotrers précédemment reçue de la part du service de méta-information et envoie à chacun d'eux une requête de récupération de fragment de donnée "$GetFileMessage$". Les messages sont envoyés tous en même temps en utilisant l'opération $\ll$ (opération *fill*) en utilisant une boucle sur le nombre de storers de la liste (ligne 29–30) ; à chaque envoi de message à un storer donné celui-ci est mis dans le buffer `wait` afin de garder en mémoire les réponses attendues (ligne 27) ; le compteur `cptr` est aussi initialisé avec le nombre de storers de la liste afin de pouvoir suivre le nombre de réponses reçues et celles encore à venir (ligne 28).
- ligne 34–50, le client récupère les réponses envoyées par les différents storers contactés, le message est reçu et consommé du réseau `nw` (ligne 34). La provenance du message est vérifiée grâce à l'identifiant de session et le pair source est ainsi retiré du buffer `wait`(ligne 35) ; dans la même action le client met à jour le buffer `frags` en y consommant le fragment "None" et y ajoutant le fragment reçu (ligne 36–37) et, à la fin de l'action, le compteur est décrémenté (ligne 38).
- ce -processus est répété en boucle jusqu'à ce que la valeur du compteur soit nulle (ligne 42–50). A la fin du -processus de récupération des fragments le client reconstruit le fichier initial en utilisant l'algorithme *reed-solomon* (procédé dont nous avons fait abstraction dans notre modélisation vu sa complexité et le peu d'information qu'il apporte au modèle).

Maintenant que le sous-processus de récupération a été présenté, nous allons passer à la description du modèle du sous-processus de stockage de fichiers.

la figure 4.3 représente le modèle du sous-processus du stockage de fichier dans le système `PutClient`, comme pour chaque sous-processus on trouve au début la déclaration des buffers locaux :

- `storers` un n-uplet d'entiers qui contient la liste des storers disponibles pour stocker un fragment de fichier (storers envoyés par le service de méta-information),
- `wait` de type entier contient la liste des pairs dont on attend une réponse,
- `cptr` de type entier utilisé comme compteur,
- `fid` de type entier qui contient les identifiants des fichiers,
- `putsession` de type entier qui contient les identifiants de session pour chaque action "Put".

Le déroulement d'une action "Put" est le suivant :

```
1             net PutClient():
2
3                  buffer storers : tuple(int) = ()
4                  buffer fid : int = ()
5                  buffer wait : int = ()
6                  buffer cptr : int = ()
7                  buffer putsession : int = ()
8
9                  [params-(("Put",Fid)),
10                  sessionid-(ids),putsession+(ids),
11                  sessionid+(ids+1),
12                  fid+(Fid),
13                  files?(Fid, frags),
14                  nw+(this,(Fid-1)%n
15                     ,Fid%n,(Fid,ids,"GetStorersList"))]
16
17                 ; [putsession?(ids), fid?(Fid),
18                    nw-(src,this,(Fid,ids,strs)),
19                    storers+(strs)]
20
21                 ; [putsession?(ids),fid?(Fid),
22                    storers?(strs),
23                    files?(Fid, frags),
24                    nw<<([(this,s,
25                     (Fid,ids,"PutFileMessage",f))
26                        for f, s in zip(frags, strs)]),
27                    cptr+(len(frags)),
28                    wait<<(strs)]
29
30                 ; (([nw-(src,this,(Fid,ids,"ok")),
31                      fid?(Fid),
32                      wait-(src),putsession?(ids),
33                      cptr-(c), cptr+(c-1) if c > 0]
34                    + [nw-(src,this,(Fid,ids,"ko")),
35                      fid?(Fid),
36                      wait-(src),putsession?(ids),
37                      cptr-(c), cptr+(c-1) if c > 0])
38                    * [cptr-(0)])
39                 ;[fid-(Fid),Fileid+(Fid)]
```

FIGURE 4.3 – Extrait du modèle ABCD du sous-processus `putclient` qui modélise l'opération "Put".

- lignes 9-15, le client commence par récupérer la liste des actions à partir du buffer `params`, le -processus `PutClient` consomme une pair dont le premier élément est la chaîne de caractère "Put" et dont le second élément est lié à la variable `Fid` ; cette action modélise la requête de "Put" initiée par l'utilisateur "User". A la réception de la requête "Put" la -processus `PutClient` consomme l'identifiant de session précédent et l'incrémente pour obtenir un nouvel identifiant de session qu'il remet dans le buffer `putsession` et sauvegarde l'identifiant du fichier dans le buffer `Fid` pour l'utiliser plus tard. Dans la même action atomique le client envoie un message au service de méta-information pour solliciter une liste de storers qui peuvent stocker un fragment du fichier que l'utilisateur souhaite stocker dans le système.
- lorsque le service de méta-informations reçoit la requête du client, il vérifie les pairs de son voisinage qui sont disponibles et qui peuvent

```
1        net Storer():
2             buffer data : int*str = initdata
3             net CorrectStorer():
4
5                  ([nw-(src,this,(Fid,ids,"GetFileMessage")),
6                    data?(Fid,frag),
7                    nw+(this,src,(Fid,ids,frag))]
8
9                   + [nw-(src,this,
10                     (ids,Fid,"PutFileMessage",frag)),
11                      data+(Fid,frag),
12                      nw+(this,Fid%n,
13                     (Fid,this,"PublishFragMI")),
14                      nw+(this,src,(ids,Fid,"ok"))])
15                  *[False]
```

FIGURE 4.4 – Extrait du modèle ABCD du sous-processus `Storer` qui modélise le service de stockage.

sauvegarder les fragments de données. Lignes 17–19, le client reçoit la liste de storers qui peuvent stocker les fragments du fichier en consommant le message mis dans le buffer `nw` en s'assurant que le message reçu correspond bien à la requête initiée par le client (vérification des paramètres identifiant de session et identifiant du fichier).

- Lignes 21–27 Une fois la liste de storers reçue par le client, ce dernier envoie une requête de stockage d'un fragment de donnée à chacun des pairs de la liste en broadcast comme on peut le voir ligne 24–25 avec l'opération *fill* dans le buffer `nw` ; dans la même action atomique le client initialise le compteur `cptr` et met la liste des storers desquels il attend une réponse dans le buffer `wait`.
- le client se met en attente des réponses des storers ; les réponses peuvent être soit un acquittement de succès ou d'échec du stockage. Lignes 29–31, le client consomme le message d'acquittement de succès de stockage, décrémente le compteur et enlève le sotrer dont il a reçu la réponse du buffer `wait`. La possibilité des deux choix de réponse est modélisé par l'opération de choix "+" (ligne 32). Le message d'acquittement d'échec de stockage est consommé par le client de la même manière que précédemment.

Service de stockage

La figure 4.4 montre un extrait du modèle représentant le service de stockage `Storer`, au début du sous-processus on déclare les buffers propres au service de stockage : `data` dont le type des contenus est le produit cartésien de l'ensemble des entiers et de l'ensemble des chaînes de caractères. Les deux opérations principales du service de stockage sont : le stockage de fragments de données et le renvoi d'un fragment de donnée précédemment stocké . Le -processus

de stockage se fait comme suit :

- lignes 5–7, le service de stockage gère la requête de récupération de fragment de donnée envoyée par le client (sous-processus `GetClient`), il consomme le message $GetFileMessage$ à partir du réseau `nw`, et dans la même action consulte son buffer `date` pour récupérer le fragment sollicité sans le consommer car il ne s'agit pas d'une opération de suppression. Le service de stockage renvoie alors la réponse au client en joignant une copie du fragment de donnée au client.
- lignes 9–14, le service de stockage gère la requête de stockage de fragment de donnée envoyé par le client (sous-processus `PutClient`) en consommant le message $PutFileMessage$ et le fragment de donnée envoyé par le client à partir du réseau `nw`. Un service de stockage qui se comporte correctement et qui n'a aucun problème disque va stocker la donnée dans le buffer `data` (ligne 11), ensuite il enverra la réponse d'acquittement de succès au client(ligne 14), puis il va informer le service de méta-information qui gère le fichier en publiant la méta-information qui permettra au client de retrouver sa donnée ultérieurement. Pour publier la méta-information, le service de stockage produit un message de $PublishFragMI$ dans le buffer `nw`, et ce dernier est relayé au servie de méta-information concerné grâce à la DHT.

Service de méta-information

La figure 4.5 présente le sous-processus `MIService` qui modélise le service de méta-information. Au début du sous-processus on déclare les buffers propres au sous-processus : `meta` dont le type du contenu est le produit cartésien de l'ensemble des entiers et l'ensemble des n-uplets de type chaînes de caractères, le buffer `voisins` de type entier et le buffer `count` de type entier qui joue le rôle d'un compteur. Les principales activités du service de méta-informations sont : fournir la liste des storers qui stockent un fragment de donnée, fournir la liste des storers qui peuvent stocker un fragment de données et maintenir les méta-informations relatives aux fragments de données stockés dans le système.

- lignes 12–15, le service de méta-information traite la requête du client (sous-processus `GetClient`) $GetFileMImessage$ et renvoie la liste des storers qui stockent un fragment du fichier d'identifiant Fid. Le service de méta-information consomme le message envoyé par le client dans buffer `nw`, vérifie qu'il contient les méta-informations relatives au Fid ciblé (ligne 14) et produit un message réponse dans le buffer `nw`.
- lignes17–21, le service de méta-information traite la requête du client (sous-processus `putclient`) $GetStorersList$ et renvoie la liste des storers

```
1         net MIService():
2
3             buffer meta :
4                  int*tuple(int|enum(None)) = initmeta
5             buffer voisins : tuple(int) = initstorers
6             buffer maliciousmeta : int*int =
7                       (((this+n+2)%n, (this+n-2)%n),)
8
9             net CorrectMIService():
10                buffer count : int = ()
11
12                ([nw-(src,this,
13                  (Fid,ids,"GetFileMImessage")),
14                  meta?(Fid,str),
15                  nw+(this,src,(Fid,ids,str))]
16
17                 + [nw-(src,this,
18                   (Fid,ids,"GetStorersList")),
19                    voisins?(strs),meta+(Fid,(None,)),
20                    nw+(this,src,(Fid,ids,strs)),
21                    count+(len(str))]
22
23                 +(([nw-(src,this,
24                       (Fid,src,"PublishFragMI")),
25                        meta-(Fid,(None)),
26                        meta+(Fid,(src,)),
27                         count<>(n=n-1) if count > 0]
28                    +[nw-(src,this,(Fid,src,"timeout")),
29                       meta-(Fid,(None)),
30                       meta+(Fid,(src,)),
31                        count<>(n=n-1) if count > 0])
32                   ;((([nw-(src,this,
33                     (Fid,src,"PublishFragMI")),
34                      meta-((Fid,s)),
35                      meta+(Fid,s+(src,)),
36                      count<>(n=n-1) if count > 0])
37                       + [nw-(src,this,(Fid,src,"timeout")),
38                          count<>(n=n-1) if count > 0])
39                      * [count-(0)])))
40                *[False]
```

FIGURE 4.5 – Extrait du modèle ABCD du sous-processus `MIService` qui modélise le service de méta-information.

qui peuvent stocker un fragment du fichier d'identifiant *Fid*. Le service de méta-information consomme le message envoyé par le client du buffer `nw`, récupère à partir du buffer `voisins` une liste de storer qu'il renvoie au client en produisant un message réponse dans le buffer `nw` et il initialise le buffer `meta` avec l'identifiant *Fid* du futur fichier qui sera stocké par son voisinage et dont il aura la charge de maintenir les méta-informations, enfin il se met en attente des publications des méta-informations par les storers de son leaf-set en initialisant le compteur `count` au nombre de storers qu'il a envoyé au client.

– lignes 23–39, le service de méta-information se met en attente des méta-informations relatives à chaque fragment stocké ; il consomme partir du buffer `nw` les messages à qui ont un message *PublishFragMI* qui sont envoyés par les storers de son voisinage, et en mettant les identifiants des storers donc il reçoit une méta-information dans le buffer `meta` et

décrémenter le compteur à chaque message reçu.

4.4 Conclusion

Dans ce chapitre nous avons présenté la contribution apporté en matière de sécurité. Nous avons analysé la robustesse du système en la présence de pairs malveillants. On a pu mettre en évidence un problème très critique dans les système de stockage, qui est la perte des données, Et une solution a été proposée pour la détection des comportements malveillant pour éviter ces pertes de données.

Chapitre 5

Implémentation et simulation

Pour évaluer la résistance du système UbiStorage à la présences de pairs malveillants tels que définis précédemment, nous allons recourir à la simulation. Cela permet notamment de mesurer combien de fichiers peuvent être perdus en fonction du nombre de pairs malveillants dans le système.

5.1 Architecture de la simulation

Nous avons conçu et mis en œuvre un simulateur entièrement configurable qui intègre le code effectivement exécuté sur les Noébox. Le simulateur prend en entrée un fichier de configuration qui décrit l'environnement de simulation,*i.e*des informations concernant soit : le nombre de pairs dans le réseau simulé, le nombre de pairs malveillants dans le système, le nombre de fichiers stockés dans le système, la durée de la simulation, le seuil de la reconstruction (*i.e*, le nombre minimal de fragments à partir duquel une reconstruction doit être lancée), soit la taille du "leaf-set"...etc

En prenant ces informations en compte, une trace d'événements est générée, ce fichier est appelé "churn trace" et décrit le départ et l'arrivée des pairs dans le système. Trois événements sont considérés : la création d'un nouveau pair (*i.e*, l'installation d'une nouvelle boîte), déconnexion définitive d'un pair (*i.e*, crash d'une Noébox) et la déconnexion temporaire d'un pair (*i.e*, le redémarrage d'une boîte arrêtée pendant la nuit, ou une erreur temporaire du réseau). Le moteur de simulation suit la "churn trace", qui déclenche des réactions des pairs comme dans le système réel. Notez que nous n'avons pas considéré les clients dans la simulation, mais nous avons considéré un système initialisé avec un certain nombre de fichier. En effet, comme expliqué dans l'introduction de ce chapitre, notre objectif principal est d'évaluer la résistance du système de stockage en terme de pertes de fichiers.

Durant la simulation un système de monitoring surveille lu système et son exécution, et nous donne des informations sur son état. En particulier, il indique le nombre de fichiers encore stockées dans le système (*i.e*, qui peuvent être reconstruite), et le nombre de fichiers perdus (*i.e*, pour lesquels les méta-informations ont disparu ou il n'existe pas assez de fragments pour permettre la reconstruction). Nous avons effectué un certain nombre de simulations

avec comme configuration initiale des paramètres différents, en particulier le pourcentage de pairs malveillants de chaque type.

5.2 Résultats expérimentaux

Dans cette section, nous présenterons les observations tirées après une séries de simulations d'un réseau composé de 200 pairs, avec un nombre de 10 000 fichiers stockés et dont les fragments sont uniformément répartis sur les pairs du système,un seuil de reconstruction fixé à 50 % (*i.e*dès que $r/2$ fragments ne sont plus disponibles, une reconstruction est tentée). Ce paramètre peut être réglé facilement pour permettre d'améliorer la résistance du système aux attaques A1 et A2, mais en même temps cela conduit à une quantité accrue d'opérations de reconstruction très coûteuses pour le système. La taille du leaf-set est fixée a 16 nœuds et la durée de simulation a été fixée à 1 an et 23 jours (la durée de dérouler près de 16 000 événements). En moyenne, chaque simulation a coûté environ 75 minute de calculs sur un PC avec un processeur Intel® Core™ i5-520M CPU at 2.40GHz équipée avec 4Gb of RAM. Ces choix des paramètres permet un temps de calcul raisonnable, tout en permettant d'exhiber des comportements intéressants. Nous avons également exécuté des simulations avec un nombre plus élevé ou de pairs et de fichiers, ce qui a entraîné des résultats similaires. Cela a également permis de vérifier que, pour un nombre donné de fichiers, le nombre de fichiers perdus diminue avec l'augmentation du nombre de pairs : le système est plus résistant lorsque chaque poste est en charge de moins de fichiers.

La figure 5.1 et la figure 5.2 résument les résultats de ces simulations. (attaques A1, A2, B et C sont nommés de façon cohérente à l'égard des paragraphes correspondants dans le chapitre 4.

Comme on peut le voir dans la figure 5.2, les attaques B et C provoquent des pertes de données plus tôt que les attaques A1 et A2. Cela n'est pas surprenant car les attaques A1 et A2 concernent les fragments de données, alors que les attaquent B et C concernent directement les fichiers, que ce soit au niveau des méta-informations (attaque B) ou en laissant disparaître (attaque C). Comme il est expliqué ci-dessus, les dommages de l'attaque B peuvent être atténués par le maintien d'une ou plusieurs copies des la méta-informations (de manière centralisée ou distribuée). De même, les dommages de l'attaque C peuvent être atténués en dupliquant les processus de suivi et de la reconstruction, ce qui nécessite également de dupliquer des méta-informations (chaque service de surveillance a besoin des méta-informations FileMI) pour tous les fichiers qu'il est en charge de surveiller). Ainsi, les deux

attaque A1 : faux succés lors du stockage

% malveillants	≤ 5	6	7	10	15	20
% fichiers perdus	0	0	0	0.67	1.29	3.64

attaque A2 : faux échec lors du stockage

% malveillants	≤ 5	6	7	10	15	20
% fichiers perdus	0	0	0	0	1.71	5.35

attaque B : service de méta-information malveillant

% malveillants	0	1	2	3	5	7
% fichiers perdus	0	1.57	3.92	4.08	4.51	5.71

attaque C : service de reconstruction malveillant

% malveillants	0	1	2	3	4	5
% fichiers perdus	0	1.66	2.01	2.60	4.26	5.94

FIGURE 5.1 – Résultats de la simulations des attaques implémentées.

questions sont clairement liées et reposent sur une duplication cohérente des méta-informations. Une partie de ce problème a été déjà résolue car c'est une exigence cruciale dans un réseau où les pairs peuvent être déconnectés à tout moment (et reconnectés plus tard) : en effet, quand un pair se déconnecte un ou plusieurs autres pairs le remplacent dans sa fonction de service de méta-information, car il on ne peut se permettre d'attendre qu'il soit de nouveau en ligne pour pouvoir récupérer les fichiers qu'ils gèrent, et dans le cas ou le pair est définitivement déconnecté ou en panne il faut qu'un autre pair puisse le remplacer facilement et récupérer les informations qu'ils stockaient auparavant. Pour faciliter cette opération nous exploitons le fait que les FileMI peuvent être reconstitués à partir des FragMIs stockés par les service de stockage dans le leaf-set. Quand un pair se déconnecte, toutes les méta-informations dont il était chargés doivent être gérées par d'autres pairs dans son leaf-set, à l'inverse, quand un pair revient en ligne il devient responsable d'un certain nombre de méta informations qu'il ne gérait pas forcement avant de se déconnecter (fichiers ajoutés lors de sa déconnexion). Dans cette situation, les FileMIs sont reconstruits en utilisant les FragMIs attachés à chaque fragment de donnée sauvegardé par un storer. Pour ce faire , le service de méta-information envoie des requêtes à chaque pair de son leaf-set avec la liste des identifiants dont il est responsable , et tous les storers lui renvoient les FragMIs correspondants, c'est ainsi que les méta-informations peuvent être accessibles aux clients. Donc, les dégâts causés par les attaques B et C peuvent être atténués par l'exploitation de ce mécanisme en vue de rechercher un FileMI invalide ou manquant sur les pairs voisins du

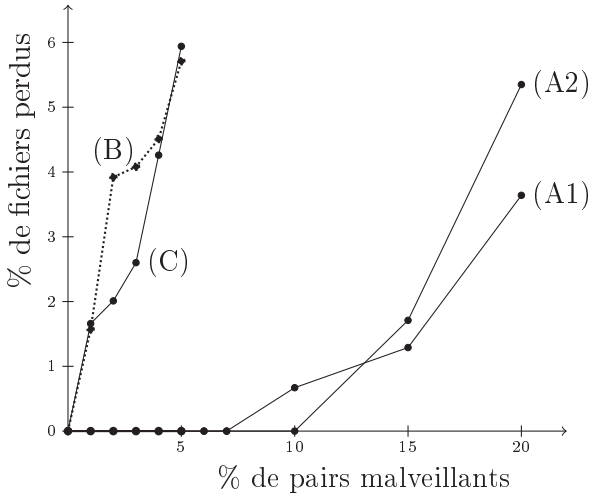

FIGURE 5.2 – Taux de perte de fichiers par rapport au taux de pairs malveillants dans le système.

pair malveillant. Cependant, cela peut conduire à un problème de consensus qui est connu pour être difficile dans les systèmes distribués [6]. Ajouté à cela, le routage sous-jacent ne permet pas de choisir quel pair sera contacté par un autre pair au cours des communications : de par sa structure la DHT permet uniquement au nœud en ligne dont l'identifiant est le plus proche de l'ID demandé d'être directement contacté. Apporter des changements à ce niveau est une extension très complexe qu'il serait préférable d'éviter.

Il est également intéressant de noter que le système UbiStorage existe en deux versions (centralisée et distribuée). La version entièrement distribuée et celle qui a été étudiée dans le présent document, une version antérieure est basée sur un service central de méta-informations a été pris en compte dans [3]. La vulnérabilité aux attaques B et C est à priori un argument en faveur de la version centralisée. Mais cela peut être controversé en considérant que toutes les méta-informations pour l'ensemble du système peuvent être perdues en même temps dans le cas d'une attaque du serveur central (déni de service...) qui devient un point de défaillance unique. Nous croyons donc que la version distribuée prise en compte dans ce document est une meilleure base pour construire un système sécurisé. Un compromis possible consiste à introduire dans le système distribué un service de méta-informations centralisé pour servir de solution de repli dans le cas de défaillance de l'un des services de méta-information situés sur les pairs. Idéalement, un tel service central pourrait être dans le Cloud pour permettre l'amélioration de sa réactivité et sa résistance aux attaques. Notons que cette approche permet de

attaque A1 : faux succès lors du stockage

% malveillants	≤ 20	25	30	40	50
% fichiers perdus	0	0	0	0	0

attaque A2 : faux échec lors du stockage

% malveillants	≤ 20	25	30	40	50
% fichiers perdus	0	0	0	0	0

FIGURE 5.3 – Résultats de la simulations du système de notation

trouver une solution simple pour le problème aussi longtemps que le service de méta-information central est considéré comme fiable.

En ce qui concerne les attaques A1 et A2, elles peuvent être considérées comme des tests réels de la robustesse du système. Et nous pouvons observer une résistance satisfaisante du système, qui peut à coup sûr être encore améliorée en augmentant la redondance de fragments (*i.e*envisager une plus grande valeur du paramètre r). La résistance du système peut également être améliorée en augmentant le seuil critique de la reconstruction, fixé à 50 % dans nos simulations. D'autres simulations réalisées avec un seuil plus élevé ont montré que le nombre de fichiers perdus diminue lorsque le seuil augmente. D'une certaine façon, ce genre d'attaques n'est pas très différent de simples défaillances des pairs, par exemple des : pannes disque (*i.e*les fragments sont perdus), et le système est justement conçu pour résister à ce problème. Donc, on peut supposer qu'une bonne maintenance et une surveillance du réseau de pairs peut efficacement prévenir la multiplication des pairs malveillants et qu'ils peuvent être détectés en utilisant des défis appropriés avant que les pertes de fichiers ne se produisent réellement.

5.3 Résultats de la simulation du système de réputation

Dans cette section, nous allons voir les premiers résultats obtenus lors de la vérification de l'efficacité du système de notation à un seul niveau système vérifié dans le chapitre précédent avec le model-checking).

Nous avons fait la simulation du système de notation à seul niveau (notation des transactions uniquement), avec la même configuration utilisée pour l'analyse de la robustesse du système en la présence des pairs malveillants. Le protocole de diffusion des notes de confiance a été émulé en utilisant une mémoire partagée entre les différents pairs.

Comme on peut le voir dans la figure 5.3, les attaques A1 et A2 sont inefficace en la présence de notre système de notation. On a pu faire des simulations avec 50% de pairs malveillants dans le système pour chaque type d'attaque, et dans les deux cas il n y a eu aucune perte de données. Nous n'avons pas pu aller au-delà de 50%, car la machine utilisée pour la simulation n'était pas assez puissante et n'offrait pas assez de mémoire. Ces résultats de la simulation, nous ont permis d'avoir une première idée sur l'efficacité de notre système de notation (à un seul niveau). Malheureusement comme on l'a vu dans le chapitre précédent.

5.4 Conclusion

Nous avons présenté dans ce chapitre, les résultats de l'analyse de la robustesse du système de stockage pair-à-pair à la présence de pairs malveillants en utilisant la simulation. Par la même méthode, nous avons obtenu des résultats d'évaluation de l'efficacité du système de notation simplifié.

Chapitre 6

Modélisation formelle et vérification

6.1 Les pairs malveillants

Nous avons présenté dans le chapitre précédent plusieurs comportements malveillants des pairs compromis dans le système, et nous avons vu la technique de modélisation du système de stockage pair-à-pair d'UbiStoarge. Dans cette section nous allons voir la manière dont on a modélisé les différents comportements malveillants du système. Pour cela, nous avons du enrichir le modèle précédent du système qui fonctionne correctement en ajoutant des sous-processus et plusieurs buffers de contrôle.

6.1.1 Modèle du service de stockage malveillant

Nous avons distingué deux comportements malveillants que peut avoir le service de stockage, le Faux succès lors du stockage (attaque A1) et le faux échec lors du stockage (attaque A2). Le modèle du service de stockage a été enrichi par deux autres sous-processus correspondant à chaque comportement malveillant ainsi que de plusieurs buffers : `maliciousdata` du même type que le buffer `data` et qui sert à stocker les fragments de données collectés de manière frauduleuse pour tenter une reconstruction quand il y en assez et un buffer `maliciousfrag` propre à l'attaque A1 et qui contient des fragments de données frauduleux que le storer renvoie au client. Les modèles respectifs de chacun de ses attaques sont les suivants :

Faux succès lors du stockage (attaque A1)

La figure 6.1 représente le modèle du sous-processus `WriteOKStorer` qui modélise l'attaque A1, ce sous-processus est composé d'un seul buffer `maliciousfrag` qui lui est local, mais accède aussi aux buffers déclarés dans le sous-processus storer (le sous-processus père). Le comportement du service de stockage malveillant est le suivant :

```
1        net Storer():
2             buffer data : int*str = initdata
3             buffer maliciousdata: int*str = ()
4
5             net WriteOKStorer():
6
7                  buffer maliciousfrag : str  = ("x.x")
8
9                  ([nw-(src,this,
10                      (Fid,ids,"GetFileMessage")),
11                   maliciousfrag?(frag),
12                   nw+(this,src,
13                      (Fid,ids,frag))]
14                  + [nw-(src,this,
15                         (ids,Fid,"PutFileMessage",frag)),
16                     maliciousdata+(Fid,frag),
17                     nw+(this,Fid%n,
18                        (Fid,this,"Publish␣FragMI")),
19                     nw+(this,src,(ids,Fid,"ok"))])
20                     *[False]
```

FIGURE 6.1 – Extrait du modèle ABCD du sous-processus `storer` qui modélise le comportement malveillant du service de stockage : faux succès lors du stockage (attaque A1).

- lignes 9–13, le service de stockage malveillant traite la requête du client de "Get" et consomme le message de *GetFileMessage* à partir du buffer `nw`, étant malveillant il ne va pas renvoyer le bon fragment mais va choisir un mauvais fragment dans son buffer `maliciousfrag` et le renvoie au client, dans notre modèle nous avons modélisé le fragment malveillant par la chaîne de caractère "$X.X$". Ainsi le client ne pourra jamais récupérer son fragment.
- lignes 14–18, le service de stockage malveillant traite la requête "Put" du client et consomme le message de *PutFileMessage* à partir du buffer `nw`, étant malveillant il va sauvegarder le fragment envoyé par le client dans le buffer `maliciousdata` et va envoyer un message d'acquittement de réussite au client.

Faux échec lors du stockage (attaque A2)

La figure 6.2 représente le modèle du sous-processus `WriteKOStorer` qui modélise l'attaque A2, ce sous-processus n'a pas de buffers locaux, mais accède aussi aux buffers déclarés dans le sous-processus storer (le sous-processus père). Le comportement du service de stockage malveillant est le suivant :

- lignes 7–12, le service de stockage malveillant traite la requête du client de "Get" et consomme le message de *GetFileMessage* à partir du buffer `nw`, étant malveillant il ne va pas stocker de fragment et donc ne sera pas sollicité pour la restauration. mais dans le cas contraire (lorsqu'un service de méta-information malveillant le désinge comme

```
1        net Storer():
2             buffer data : int*str = initdata
3             buffer maliciousdata: int*str = ()
4
5             net WriteKOStorer():
6
7                  ([nw-(src,this,
8                        (Fid,ids,"GetFileMessage")),
9                     data?(Fid,frag),
10                    nw+(this,Fid%n,
11                        Fid,this,"timeout")),
12                    nw+(this,src,(Fid,ids,frag))]
13
14                 + [nw-(src,this,
15                        (ids,Fid,"PutFileMessage",frag)),
16                     maliciousdata+(Fid,frag),
17                     nw+(this,src,(ids,Fid,"ko"))])
18                     *[False]
```

FIGURE 6.2 – Extrait du modèle ABCD du sous-processus `storer` qui modélise le comportement malveillant du service de stockage : Faux échec lors du stockage (attaque A2).

storer pour un fragment de donnée) il va renvoyer un message au client avec un message de *TimeOut*, qui dans notre modèle représente une non réponse du service de stockage.

– lignes 14–18, le service de stockage malveillant traite la requête "Put" du client et consomme le message de *PutFileMessage* à part du buffer `nw`, étant malveillant il va sauvegarder le fragment envoyé par le client dans le buffer `maliciousdata` et va envoyer un message d'acquittement d'échec au client. Ce storer en trichant sur le non stockage d'un fragment de donnée sera sollicité par le service de reconstruction lorsqu'un fichier est nécessite une reconstruction, et adoptera toujours le même comportement, cela lui permettra de collecter plusieurs fragments d'un même fichier, qu'il essaiera de reconstruire d'une manière frauduleuse.

Comme on peut le voir dans la figure 6.3, le modèle étendu du service de stockage comporte les différents comportements possibles du service de stockage (correct, attaque A1 et attaque A2), le choix de l'initialisation du service de stockage avec l'un de ces trois états est déterminé par le choix du paramètre `sstype` qui est passé au -processus racine `UbiSystem`. Le buffer `sstype` est de type entier, et on associe un chiffre à chaque type du service de stockage : (0 :correct, 1 :attaqueA1, 2 :attaqueA2). Ligne 15–20 avant l'initialisation du type du service de stockage on vérifie la valeur du paramètre `sstype` dans une action de condition *if* et on exécute de manière séquentielle l'initialisation du service de stockage. La syntaxe du langage ABCD ne permet pas d'exécuter une condition sans déclarer une action à exécuter, pour cela nous avons définie une action anodine qui n'a pas de répercutions sur l'exécution du modèle

```
1
2       net Storer():
3            buffer data : int*str = initdata
4            buffer maliciousdata: int*str = ()
5
6            net CorrectStorer():
7                ...
8
9            net WriteOKStorer():
10               ...
11
12           net WriteKOStorer():
13               ...
14
15           ([notes?((this,this,0.5)) if sstype ==0]
16            ; CorrectStorer())
17           +([notes?((this,this,0.5)) if sstype == 2]
18               ; WriteKOStorer())
19           +([notes?(this,this,0.5) if sstype == 1]
20               ; WriteOKStorer())
```

FIGURE 6.3 – Extrait du modèle ABCD du sous-processus `Storer` qui modélise l'initialisation du programme et la manière dont est défini le type du service de stockage

qui consiste à une simple vérification de la valeur du buffer `notes` dont on expliquera l'utilité dans la section suivante.

Service de méta-information malveillant (attaque B)

La figure 6.4 représente le modèle du sous-processus `MaliciousMIService` qui modélise l'attaque B, comme pour le cas du service de stockage malveillant on a du étendre notre modèle initial avec des buffers complémentaires et des nouveaux sous-processus. Le modèle de la figure. 6.4 contient les déclarations de buffer du -processus `MIService` qui sont accessibles au sous-processus fils `MaliciousMIService`, ainsi qu'un nouveau buffer `maliciousmeta` qui a le même type que le buffer `meta` et qui contient des méta-informations erronées dont se sert le service de méta-information malveillant. Le comportement de ce dernier se fait comme suit :

- lignes 13–16, le service de méta-information malveillant gère la requête "Get" du client en lui renvoyant des méta-information erronées qu'il récupére dans le buffer `maliciousmeta`. Pour des problèmes de complexité nous avons fixé les méta-informations erronées du buffer `maliciousmeta` à une liste de storers que le service de méta-information prend en dehors de son voisinage direct (son leaf-set). Avec les fausses informations envoyées au client on est sûr que la donnée du client ne pourra pas être récupérée.
- lignes 18–35, la procédure de publication des méta-informations est la même que quand le service est correct.

```
1         net MIService():
2
3             buffer meta :
4                  int*tuple(int|enum(None)) = initmeta
5             buffer voisins : tuple(int) = initstorers
6             buffer maliciousmeta : int*int =
7                       (((this+n+2)%n, (this+n-2)%n),)
8
9             net MaliciousMIService():
10
11                buffer count : int = ()
12
13                ([nw-(src,this,
14                     (Fid,ids,"GetFileMImessage")),
15                  maliciousmeta?(Fid,str),
16                  nw+(this,src,(Fid,ids,str))]
17
18                 + (([nw-(src,this,
19                         (Fid,src,"Publish␣FragMI")),
20                       meta-(Fid,(None)),
21                       meta+(Fid,(src,)),
22                        count<>(n=n-1) if count > 0]
23                     +[nw-(src,this,
24                          (Fid,src,"timeout")),
25                       meta-(Fid,(None)),
26                       meta+(Fid,(src,)),
27                        count<>(n=n-1) if count > 0])
28                    ;((([nw-(src,this,
29                             (Fid,src,"Publish␣FragMI")),
30                      meta-((Fid,s))
31                      ,meta+(Fid,s+(src,))
32                              ,count<>(n=n-1) if count > 0])
33                      + [nw-(src,this,(Fid,src,"timeout")),
34                         count<>(n=n-1) if count > 0])
35                       * [count-(0)]))
36
37                 + [nw-(src,this,
38                        (Fid,ids,"GetStorersList")),
39                    voisins?(strs),
40                    meta+(Fid,(None,)),
41                    nw+(this,src,(Fid,ids,strs))])
42
43                   *[False]
```

FIGURE 6.4 – Extrait du modèle ABCD du sous-processus `MaliciousMIService` qui modélise le service de méta-information malveillant.

– lignes 37–41, le service de méta-information malveillant se comporte de la même manière que le service correct, et gère la requête "Put" du client en lui renvoyant une liste de storers dans son leaf-set et la suite de la requête se fait correctement quand les storers fournis sont corrects aussi.

L'initialisation du service de méta-information se fait à la fin du processus `MIService` comme pour chaque processus modélisé en ABCD, le type qui sera affecté au service de méta-information dépend du paramètre `mistype` passé au -processus principal `UbiSystem`. Les éléments du buffer `mistype` sont de type entier et on associe deux chiffres aux différents types du service de méta-information (0 :correct, 1 :malveillant). Ainsi se fait l'initialisation du service de méta-information, de manière analogue à l'initialisation du service de stockage.

6.1.2 Vérification formelle et analyse

Pour analyser la robustesse du système et sa résistance à la présence de pairs malveillants on a eu recours à la vérification formelle. On a pris en compte les différents cas du comportement que peut avoir un pair (honnête/-malveillant), pour chaque cas de comportement malveillant présenté plus haut nous avons pris une instance du modèle avec le paramétrage adéquat (noter que plusieurs combinaisons de comportements malveillants sont possibles). Pour chaque modèle on a défini des scénarios d'exécution qui consistent en une suite d'opérations "Get" et "Put" initiées par le client. Pour plus d'-efficacité on a considéré un modèle pour le comportement correct d'un pair et on l'a pris comme cas référence, on a aussi défini un modèle pour chacun des comportements malveillants des services de stockage et méta-information (attaques A1, A2 et B). Les réseaux de Pétri respectifs de chaque modèle ont été obtenu en compilant les modèles avec un compilateur ABCD fourni par l'outil [42]. Les réseaux de Pétri obtenus ont été ensuite passés par le compilateur [13], lequel à partir d'un réseau de Pétri peut construire une bibliothèque optimisée qui implante de façon efficace les fonctions et structures de données nécessaires à l'exploration des états accessibles du réseau de Petri. Ce processus d'optimisation permet de réduire le coût de la construction de l'espace d'états pour chaque modèle (Snakes fourni aussi la possibilité de construire l'espace d'états mais il est plus coûteux que Neco). Un espace d'états typique dans notre vérification est composé d'environ 400,000 états et peut être calculé en un temps approximatif de 10 minutes (sur une machine Intel® Core™ i5-520M , fréquence CPU à 2.40GHz équipé de 4Gb de RAM), à ce temps de calcul faut ajouter environ 2 à 5 minutes d'analyse de l'espace d'états.

Les résultats de la vérification ont démontré qu'après une séries d'opérations "Get" et "Put" il y a une perte de données. En effet l'analyse des différents buffers du modèle à montré un manque de certains fragments de fichiers dans le buffer `frags` du client et l'absence des méta-informations dans le buffer `meta` du service de stockage. Une analyse du buffer `data` du service de stockage à aussi montré que des fragments de données qui étaient sensés s'y trouver n'y étaient pas ; mais ils ont bien été retrouvés dans le buffer `maliciousdata`.

Cette vérification a confirmé les résultats obtenus par simulation, on constate bien une perte de données mais malheureusement elle ne nous permet pas d'identifier les pairs malveillants impliqués dans la perte de données. Afin de pouvoir détecter ces pairs malveillants nous allons mettre en place un système de notation basique que nous vous présenterons dans la section suivante.

6.2 Le système de notation

Comme l'a montré l'analyse précédente, les comportements malveillants conduisent à la perte de données, une solution proposée pour palier a cette perte de données est de mettre en place un système de réputation basé sur un système de notation à plusieurs niveaux ; il évalue le comportement des différents pairs ainsi que les échanges entre les pairs. En se basant sur les notes le système de réputation peux détecter lorsqu'un pair est malveillant et pouvoir l'exclure avant qu'il ne nuise à la disponibilité des données. Dans un premier temps nous nous sommes contenté d'un seul niveau de notation, car le premier objectif est de vérifier l'efficacité d'un tel système avant de se lancer dans l'implémentation d'un système coûteux qui risque de ne pas être très efficace.

6.2.1 Principe de fonctionnement

Les pairs sont étendus avec un système de notation qui évalue tous les échanges avec les autres pairs du système permettant ainsi de construire un système de réputation. Chaque pair évalue les autres pairs en leur accordant des notes comprises dans l'intervalle $[0, 1]$ et met à jour ces notes à chaque communication. La note $0,5$ étant la note moyenne à partir de laquelle un pair peut être considéré correct, et si la note du pair est en dessous de $0,5$ le pair est ainsi considéré comme potentiellement malveillant, et inversement les pairs dont les notes sont supérieurs à $0,5$ sont considérés comme probablement honnêtes.

Deux échanges sont évalués dans le protocole de stockage, les communications notées sont celles qui ont lieux lors des deux opérations "Get" et "Put" qu'elles soit initiées par le client ou par le service de reconstruction. Les issues de ces deux opération peuvent être évaluées soit elles sont satisfaisantes ou non : le résultat attendu après un "Get" est le fragment de donnée initialement stocké et le résultat attendu après un "Put" est un acquittement de succès de stockage du fragment de donnée. Si les informations attendues sont correctes alors la note du pair source est incrémenté de $0,1$, si un tout autre résultat est reçu, le pair source se voit sa note décrémentée de $0,1$ par le pair destinataire .

L'évaluation des informations envoyées par le service de méta-information sont plus difficiles à évaluer. Le service de méta-information envoie une liste de storers au client/service de reconstruction pour les deux opérations de "Get" et de "Put", dans les deux cas si les storers ne réagissent pas de la façon attendue deux cas sont possibles : soit les services de stockage sont honnêtes et le service de méta-informations a menti ou inversement les services

de stockages sont malveillants et le service de méta-information est honnête. Vu la complexité de l'évaluation du système de méta-information et de reconstruction on s'est contenté dans un premier temps de à l'évaluation des opérations "Get" et "Put".

6.2.2 Modèle du système de notation

La modélisation du système de notation nécessite l'ajout d'un buffer `notes` pour chaque pair et l'initialiser avec des notes de $0,5$ pour chaque pair, ces notes sont ensuite stockées sous formes de triplets (i, j, k) dont les deux premiers éléments sont de type entier et le dernier élément est de type réel. Chaque note stockée aura donc la forme (i, j, k) tel que : i représente l'identité du pair qui détient la note, j représente l'identité du pair noté par i et k est la note donnée par i au pair j.

La figure. 6.5 représente le modèle du système de notation implémenté dans le service client, comme on le voit l'évaluation concerne la réception de fragments de données lors de l'opération "Get" (ligne 19–29) et la réception d'acquittement de succès ou d'échec lors de l'opération "Put"(lignes 33–39).

- lignes 19–21, le message reçu est le fragment attendu alors la note du pair source est incrémenté de $0,1$. Ce cas là laisse supposer que le pair est honnête.
- lignes 23–25, le message reçu est un fragment altéré qui ne correspond pas à celui stocké initialement par le client (ici modélisé par "$X.X$"), la note du pair source est alors décrémentée de $0,1$. Ce cas là laisse supposer que le pair source est potentiellement honnête ou il a des problèmes sur le disque. Dans ce cas là cela nécessitera de toute manière une intervention d'UbiStorage pour le changement de matériel.
- lignes 27–29, le client reçoit un message du service de stockage disant qu'il ne détient pas le fragment attendu par le client, ce dernier décrémente alors la note du pair.
- lignes 33–35, le message reçu après le stockage d'un fragment de donnée est un acquittement de succès, la note du pair est alors incrémentée.
- lignes 37–39, le message reçu après le stockage d'un fragment de donnée est un acquittement d'échec, la note du pair est alors décrémentée. Ce cas là laisse à supposer deux situations : que la pair est malveillant donc sa note est méritée ou que le pair à des problème disque.

L'évaluation de l'opération "Get" est un peu plus complexe, car le client essaie de reconstituer le fichier dès réception des S fragments nécessaires sans vérifier les fragments reçus ; ce n'est qu'après l'échec de la reconstruction du fichier que le client se rend compte que l'un des service de stockage est malveillant. Des pairs malveillants peuvent ainsi ne pas être détectés car le client

```
buffer nw : object = ()

net UbiSystem(n,this, initfiles, initmeta
          , initdata, initstorers, tasks,sstype,mistype):
    buffer params : str*int = ()
    buffer notes : (int*int*float) =
                  [(this,y,0.5)  for y in range(n)]

    net Peer () :
        buffer sessionid : int = 0

            net Client () :

            buffer files : int*tuple(str) = initfiles
            buffer Fileid : int = ()

            net GetClient ():
                ●●●
                ;(([nw-(src,this,(Fid,ids,frag)),
                    notes-(this,src,x),
                    notes+(this,src,x+0.1),●●●])

                  +([nw-(src,this,(Fid,ids,("X.X"))),
                     notes-(this,src,x),
                     notes+(this,src,x-0.1),●●●])

                     +([nw-(src,this,(Fid,ids,"KO")),
                     notes-(this,src,x),
                     notes+(this,src,x-0.1),●●●]))

            net PutClient () :
                ●●●
                ;(([nw-(src,this,(Fid,ids,"OK")),
                     notes-(this,src,x),
                     notes+(this,src,x+0.1),●●●]

                  +[nw-(src,this,(Fid,ids,"KO")),
                     notes-(this,src,x),
                     notes+(this,src,x-0.1),●●●]


            (GetClient() + PutClient()) * [False]


Ubi0::UbiSystem(●●●)
```

FIGURE 6.5 – Modèle du système de notation pour les opérations "Get" et "Put".

n'aura pas utilisé le fragment qu'ils ont envoyés pour reconstruire le fichier. Il est aussi à noter qu'une panne disque ou tout autre problème matériel ou logiciel d'un pair entraîne l'abaissement de sa note ; mais heureusement, ce n'est pas un problème dans le cas d'une défaillance temporaire, car la note du pair sera nécessairement augmenté lors des échanges suivants. Dans le cas d'une défaillance permanente, le pair va être exclu car il sera considéré comme malveillant alors qu'il est réellement en panne ou endommagé. Mais les deux situations nécessitent une intervention manuelle d'UbiStorage pour la vérification de l'état de la NoeBox et éventuellement son remplacement.

6.2.3 Vérification formelle et analyse

De la même manière que pour la vérification de la robustesse du système de stockage et sa résistance à la présence de pairs malveillants, nous avons vérifié par model checking l'efficacité du système de notation mis en place. Pour chaque cas, on a pu vérifier qu'un service de stockage malveillant ne peut éviter une évaluation négative et sa note est toujours décrémentée après chaque échange avec le client ; *i.e*a chaque échange, que le service de stockage renvoie un mauvais fragment ou ne renvoie pas de fragment sa note est automatiquement baissée. Ce résultat nous permet d'affirmer qu'un service de stockage malveillant est toujours détecté ; néanmoins un service de stockage qui alterne entre un état correct et un état malveillant arrivera toujours à maintenir sa note égale à la moyenne mais cela réduit la chance de perte de données.

Les résultats obtenus par la vérification formelle du modèle abstrait ont été comparés aux résultats obtenus par la simulation du système réel étendu avec le système de notation. Le logiciel d'UbiStorage a été étendu de manière similaire au modèle avec un module de notation qui évalue les différents échanges entre les pairs ; et un module de surveillance a été implémenté au-dessus du système de notation et permet d'exclure temporairement de la simulation les pairs jugés malveillants (note $< 0,5$). Les résultats de la simulation ont confirmé les résultats obtenus par la vérification formelle, ainsi le nombre de fichiers perdus a été réduit de manière considérable.

Cependant, les résultats globaux sont moins bons que ce qu'ils ont l'air. En effet, grâce au model-checking qui est une approche systèmatique qui vérifie tous les cas d'exécution possibles, on a pu mettre en évidence des situations où un service de stockage honnête obtient une mauvaise note. Ce cas se produit lorsque le service de méta-information est malveillant, ce dernier envoie un fichier de méta-information erroné au client lorsqu'il veut restaurer sa donnée. Lorsque le client envoie une requête aux services de stockage indiqués par le service de méta-information malveillants, ces derniers ne possédant pas les fragments de cette donnée ne peuvent pas répondre positivement à la requête du client, alors le client leurs attribut des mauvaises notes injustement.

Une des raisons évidente de cette situation, est que nous modélisons pas un système de réputation comme dans [20, 31], dans lesquels les notes de confiance sont échangées entres les différents pairs. Chaque pair attribue des notes aux interactions qu'il effectue avec les autres pairs, et puis calcule la note de confiance en se basant sur ses connaissances personnelles et les connaissances extérieurs, puis diffuse ces informations aux autres pairs. Cela permet d'avoir une vue globale de la réputation que possède chaque pair dans

le système.

6.3 Conclusion

Nous avons décrit dans ce chapitre le formalisme ABCD, et nous avons montré comment on a modélisé notre système de stockage en utilisant ce formalisme. ABCD n'étant pas prévu à la base pour la modélisation de ce genre de systèmes, il a fallu concevoir des schémas de modélisations adaptés. Nous avons aussi présenté les résultats de l'analyse par model checking de l'efficacité du système de notation modélisé en formalisme ABCD.

Chapitre 7

Conclusion et perspectives

7.1 Contributions et résultats

Nous avons étudié dans ce mémoire la robustesse d'un système de stockage pair-à-pair en présence de pairs malveillants. Des comportements malveillants possibles ont été identifiés, et on n'a retenu que ceux qui causaient des pertes de données. Nous avons aussi proposé un système de réputation qui permettrait de détecter les comportements malveillants conduisant à la perte de données. Nous dressons ici un bilan de nos propositions, puis introduisons diverses perspectives pour de futurs travaux.

7.2 Bilan

Dans un premier temps, nous avons commencé par dresser les différentes attaques possibles ainsi que les comportements malveillants que peut avoir un pair du système. Des travaux faits dans le cadre du projet ANR SPREADS [9] nous ont permis d'écarter certaines attaques liées à la sécurité et l'intégrité des données, en effet, ces travaux [46] ont montré que les données stockées dans le système ne sont jamais violées et qu'aucun attaquant externe ne peut apprendre quoi que soit sur ces données. Nous nous sommes alors focalisé sur une propriété plus critique dans les systèmes de stockages pair-à-pair qui est la perte de données. Nous avons répertorié les différents comportements malveillants qui causaient la perte de données. Et on a analysé la robustesse du système en la présence de ces différents comportements. Pour l'évaluation de la perte de données nous avons opté pour la simulation. La simulation nous a permis d'avoir des résultats sur le système réel et aussi de quantifier le taux de ces pertes de données. Cette approche est globalement plus rapide que la modélisation/vérification dans notre cas puisqu'on disposait déjà du système et de l'infrastructure de simulation.

Pour pallier la perte de données, nous avons proposé un système de détection basé sur la réputation. Ce système de réputation a deux niveaux de confiance [39, 48], un niveau de confiance en les pairs (évaluation des transactions entre les pairs) et un autre niveau de confiance en les notes attribuées aux pairs. Le premier de niveau de confiance a été implémenté et

modélisé. Son efficacité à été montrée en utilisant les deux techniques : simulation et model-checking. La première technique nous a permis d'avoir un rapide aperçu de la probable efficacité d'un tel système. Le model-checking a été utilisé pour prouver l'éfficacité du système de notation et a donc confirmé les résultats obtenus par simulation ; et nous a permis de mettre en évidence une autre conséquence de l'un des comportements malveillants qui est les faux positifs. En effet, certains pairs en se comportant mal conduisaient systèmatiquement à l'exclusion des pairs honnêtes en les implicant dans des transactions auxquelles ils ne peuvent pas répondre positivement. Des solutions ont été proposées pour éviter cette descrimination, notamment en partageant la responsabilité des mauvaises notes entre tous les acteurs participants à une transaction pour encourager les uns et les autres à ne pas tricher. Par ailleurs, le second niveau de confiance devrait permettre de ne pas tenir compte des mauvaises notes injustifiées en permettant de détecter et d'exclure les pairs qui les donnent.

7.3 Autres travaux

En parallèle aux travaux présentés dans ce mémoire, nous avons effectué une modélisation avec le formalisme ABCD de l'application d'un point de vue architecture logicielle (contrairement à la modélisation présentée plus tôt qui adopte le point de vue fonctionnel). Pour cela, il a fallu modéliser en ABCD les différents concepts et éléments mis en oeuvre dans le programme (continuations, concepts objets, appel aux méthodes, ...). Il s'agit là aussi d'une utilisation nouvelle d'ABCD qui n'était pas prévue lors de la conception de ce langage. Nous n'avons pas présenté ce travail dans ce mémoire car il n'est pas directement lié à la sécurité mais concerne plus la sûreté de fonctionnement du système. Cependant, un article sur ce sujet est en cours de rédaction et sera soumis à publication.

Dans ce travail, on s'intéresse à des propriétés internes au système UbiStorage, telles que l'absence de blocages, la terminaison des protocoles, ou encore l'intégrité des états du système. Ces propriétés peuvent être vérifiées par model-checking sur le modèle ABCD du programme.

7.4 Travaux en cours et futurs

A la suite des travaux présentés dans ce mémoire, nous envisageons les perspectives suivantes.

À court terme, nous souhaitons étudier le second niveau de confiance.

Pour cela, il faut d'une part l'implémenter (travail presque achevé) et le simuler, et d'autre part, le modéliser en ABCD et en faire la vérification par model-checking.

À moyen terme, sur cette base, il faudra introduire dans les simulations, la notation des services de méta-information et de reconstruction, afin d'en évaluer la qualité. Au niveau du modèle, il faudra trouver une façon de représenter le service de reconstruction qui n'introduise pas trop de combinatoire et permettent d'en analyser le fonctionnement, y compris avec le second niveau de notation. Les difficultés principales sont la modélisation du système de Reed-Solomon ainsi que l'introduction d'une dynamique dans l'arrivée et le départ des pairs.

À plus long terme, nous souhaitons introduire dans le système une infrastructure de gestion de clefs publiques (PKI) incluant un système de révocation qui faciliterait l'exclusion des pairs détectés comme malveillants. En effet, en l'état actuel des choses, pour exclure un pair, il faut récupérer la Noébox chez le client, ce qui est une intervention qui ne peut pas se faire rapidement. Pour s'intégrer de façon satisfaisante dans le système d'UbiStorage, un tel PKI devrait être entièrement distribué, ce qui posera des questions sur la cohérence des informations et la prise décision d'une révocation. La simulation devrait permettre d'évaluer la rapidité de diffusion d'une révocation au sein du système global, alors que le model-checking devrait permettre d'en vérifier la correction.

Chapitre 8

Annexes

8.1 Modèle ABCD du protocole du système UbiStorage

Le code suivant représente le modèle du protocole du système UbiStorage écrit dans le formalisme ABCD.

```
buffer nw : object = ()


##============================================================================
##
## UbiSystem
##
##
net UbiSystem(n,this, initfiles, initmeta, initdata, initstorers, tasks):

    buffer params : object = ()
##============================================================================
##
## PEER
##
##
    net Peer ():

        buffer sessionid : int = 0
##============================================================================
##
## Routeur
##
##
        net Router () :
        # routage d'un message destin      un autre
            [nw-(src,this,adr,data), nw+(src,adr,data)]*[False]

##============================================================================
##
## CLient
##
##
        net Client () :
         # chaque fichier = (Fid, (frag1, ..., fragN))
            buffer files : object = initfiles
            buffer Fileid : object = ()

            net GetClient ():
                buffer storers : object = ()
                buffer wait : int = ()
                buffer cptr : int = ()
                buffer frags : object = ()
                buffer fid : int = ()
                buffer getsession : int = ()
```

```
1               ##envoie le message    son voisin
2                   [params-(("Get",Fid)),
3                    fid+(Fid),
4                    sessionid-(ids),getsession+(ids),
5                    sessionid+(ids+1),
6                    nw+(this,(Fid-1)% n ,Fid%n,(Fid,ids,"GetFileMImessage"))]
7
8               ##reception de la liste des storers
9                   ; [getsession?(ids),fid?(Fid),
10                     nw-(src,this,(Fid,ids,strs)),
11                     storers+(strs)]
12
13
14              ##envoie de requete de lecture de fragments aux storers
15              ##envoie en parrallele
16                  ; [getsession?(ids),fid?(Fid),
17                     storers-(strs),
18                     wait<<(strs),
19                     cptr+(len(strs)),
20                     nw<<([(this,s,(Fid,ids,"GetFileMessage"))
21                           for s in strs])]
22                  ;[fid-(Fid),Fileid+(Fid)]
23
24
25              ##r cup rer les fragments
26                  ;([nw-(src,this,(Fid,ids,frag)),
27                       wait-(src),getsession?(ids),
28                       frags+((Fid,frag)),
29                       cptr-(c), cptr+(c-1) if c > 0]
30                     * [cptr-(0)])
31
32              net PutClient():
33
34                  buffer storers : object = ()
35                  buffer fid : object = ()
36                  buffer wait : int = ()
37                  buffer cptr : int = ()
38                  buffer putsession : int = ()
39
40              ##envoie le message    son voisin
41                  [params-(("Put",Fid)),
42                   sessionid-(ids),putsession+(ids),
43                   sessionid+(ids+1),
44                   fid+(Fid),
45                   files?(Fid, frags),
46                   nw+(this,(Fid-1)%n,Fid%n,(Fid,ids,"GetStorersList"))]
47
48              ##reception de la liste des storers
49                  ; [putsession?(ids), fid?(Fid),
50                     nw-(src,this,(Fid,ids,strs)),
51                     storers+(strs)]
52
53
54              ## envoie des fragements au storers ???
55                  ; [putsession?(ids),fid?(Fid),
56                     storers?(strs),
57                     files?(Fid, frags),
58                     nw<<([(this,s,(Fid,ids,"PutFileMessage",f))
59          #message non envoye
60                           for f, s in zip(frags, strs)]),
61                     cptr+(len(frags)),
62                     wait<<(strs)]
```

```
1
2
3               ##reception des ok
4                   ; ([nw-(src,this,(Fid,ids,"ok")), fid?(Fid),
5                       wait-(src),putsession?(ids),
6                       cptr-(c), cptr+(c-1) if c > 0]
7                      * [cptr-(0)])
8                   ;[fid-(Fid),Fileid+(Fid)]
9
10
11              (GetClient()*[False])
12              |(PutClient()*[False])
13
14  ##=========================================================================
15  ##
16  ## storer
17  ##
18  ##
19          net Storer():
20              buffer data : object = initdata
21              buffer storers : int = initstorers
22
23          ##lecture des fragments
24              ([nw-(src,this,(Fid,ids,"GetFileMessage")),
25                data?(Fid,frag),
26                nw+(this,src,(Fid,ids,frag))]
27
28
29          ##  criture   des fragments
30              + [nw-(src,this,(ids,Fid,"PutFileMessage",frag)),
31                  data+(Fid,frag),
32                  nw+(this,src,(ids,Fid,"ok"))])*[False]
33
34  ##=============================================================================
35  ##
36  ## MIService
37  ##
38  ##
39          net MIService():
40
41              buffer meta : object = initmeta
42              buffer voisins : object = initstorers
43
44          ##envoie de la liste de storers qui stockent le file
45              ([nw-(src,this,(Fid,ids,"GetFileMImessage")),
46                meta?(Fid,str),
47                nw+(this,src,(Fid,ids,str))]
48
49
50          ##envoie de la liste des sotrers dispo pr stocker le file
51              + [nw-(src,this,(Fid,ids,"GetStorersList")),
52                 voisins?(strs), # voisins+(strs),
53                 nw+(this,src,(Fid,ids,strs))])*[False]
54
55
56          (Client()*[False])
57          |(Storer()*[False])
58          |(MIService()*[False])
59          |(Router()*[False])
60
61      net User():
62          #buffer parametre : object = ()
63          ##[params+(("Get",Fid1)),params+(("Put",Fid2))]
64          #[parametre+(task)]
65          [params<<([t for t in tasks if t[0] != "NOP"])
66           if [t for t in tasks if t[0] != "NOP"]]
67
68      Peer() | User()
69
70      ##User1::User((("Get",2),("Put",2)))
71      ##| Peer1::Peer( )
72      ##| Peer2::Peer( )
```

```
##leafset1
Ubi0::UbiSystem(6,0, #n, this
                (), # initfiles
                (), # initmeta
                ((1,("1.1")),), # initdata
                ((1,5),), # initstorers
                (("NOP",0),) #task
                )
|Ubi1::UbiSystem(6,1, #n, this
                  ((4,("4.1","4.2")),), # initfiles
                  ((1,(0,2)),), # initmeta
                  (), # initdata
                  ((0,2),), # initstorers
                  (("Put",4),)#("Nop",4)) #task
                  )
|Ubi2:: UbiSystem(6,2, #n, this
                  (), # initfiles
                  (), # initmeta
                  ((1,("1.2")),), # initdata
                  ((1,3),), # initstorers
                  (("NOP",0),) #task
                  )

## leafset 2
|Ubi3:: UbiSystem(6,3, # this
                  (), # initfiles
                  (), # initmeta
                  (), # initdata
                  ((2,4)), # initstorers
                  (("NOP",0),) #task
                  )
|Ubi4:: UbiSystem(6,4, # n,this
                  (), # initfiles
                  (), # initmeta
                  (), # initdata
                  ((3,5),), # initstorers
                  (("Get",1),) #task
                  )
|Ubi5:: UbiSystem(6,5, #n, this
                  (), # initfiles
                  (), # initmeta
                  (), # initdata
                  ((4,0),), # initstorers
                  (("Nop",0),) #task
                  )
```

8.2 Modèle ABCD d'architecture logicielle du système UbiStorage

Cet annexe contient le modèle au formalisme ABCD de l'architecture logicielle du système de stockage pair-à-pair UbiStorage.

8.2.1 Modèle de la fonction GetFileMI Envoie de la requête GET FIle MI et réception du FileMI

Le modèle qui suit représente le modèle de la fonction GetFileMI qui effectue l'opération de lecture ou de récupération de données déja stockées dans le système.

```
1
2
3
4  # importation du script test.py qui consrtuit
5  #les instances d'objets
6
7  from ubienv import *
8
9  # buffers mod lisant le r seaux
10 buffer MessageResponse : object = ()
11 buffer SentMessage : object = ()
12
13 buffer NextUID : int = 0
14
15 # une instance de la JVM
16 net Peer (StorerId,Fid) :
17
18
19
20     #buffer ou sont sauvegard es les continuations
21     buffer continuations : object = ()
22
23     #buffer ou sont sauvegard s les appels de m thodes
24     buffer calls : object = ()
25
26     # Une classe du net peer , le nom
27     # Client__GetFile__FileMIContinuation permet d' viter  le conflitr
28     # de noms avec d'autres classes c'est une classe qui s'appelle
29     # FileMIContinuation, imbriqu e dans la m thode GetFile de la
30     # classe Client
31
32
33
34     # cette continuation est cr ee dans
35     # la m thode GetFile qu'on abstrait pour garder que
36     # la m thode  GetFileMI
37     net Client__GetFileMI__FileMIContinuation ():
38
39          # le net onResult mod lise la m thode onR sult de la classe
40          # Client (Client__GetFile__FileMIContinuation) elle   xcute
41          # lors de la r ception du r sulat attendu
42
43          net onResult():
44
45               #buffer de sauvegarde de l'identit  du processus appellant
46               #de la m thode
47
48               buffer objectid : object = ()
49
50               #consomme l'appel par l'objet d'identit  this avec le
51               #param tres FILEMI si l'appel lui est bien destin , ce qui
52               #est v rifi  par la garde: l'objet r cup r  par this est
53               #bien un instance de la classe dont on est en train
54               #d' crire  la m thode
55
56               [calls-("call onResult",this,FILEMI),objectid+(this)
57                if this.cls=="Peer.Client__GetFileMI__FileMIContinuation"]
58
59               #appel      la m thode (du m   me objet) qui prend le FILEMI
60               #comme param tre et envoie des messages aux storers dont les
61               #ID sont dans le FIlEMI pour r up rer les frag Datas
62
63               ;[False] # appel de GetFile pour effectuer la r cup ration
64
65               # r cup re le retour de la m thode GetFile appell e
66               # deux choix de r ponse : soit un return qui redonne la main
67               #      la m thode appellante pour continuer son ex cution
68               # ou bien une exception qui interompt l' xecution  du programme
69
70               ;([calls+("return onResult",this,void)]
71                 +[calls+("except onResult",this,("erreur onResult",))])
72
73
74         # le net onException mod lise la m thode onException de la classe
75         # Client Client__GetFile__FileMIContinuation elle   xcute   lors
76         # de la r ception d'une exception
```

```
1
2
3           net onException():
4               #buffer objectid : object = ()
5
6               [calls-("call onException",this,"Echec de reception de FileMI")
7                ,calls+("except onException",this,("echec de reception de FileMI",))
8                if this.cls == "Peer.Client__GetFileMI__FileMIContinuation"]
9
10
11          # le main  de la classe FileMIContinuation
12          # les m thodes OnResult et onException restes actives en attente
13          # d'un appel de la part des objets.
14
15          (onResult() * [False])
16          | (onException() * [False])
17
18
19      # classe MI SERVICE
20
21      net MIService ():
22
23          # buffer pour stocker les fileMIs
24          # chaque fileMI est associ     un fileid
25
26          buffer FileMI : {(int,object)} = ()
27
28          net Initialisaiton(fmi):
29
30              [FileMI+(fmi)]
31
32          net GetFileMI():
33
34              buffer gfm : {(object,object)} = ()
35
36              [calls-("call GetFileMI",fid),gfm-(fid,FILEMI),
37               calls+("return GetFileMI",FILEMI)]
38              +[calls+("except GetFileMI",erreur)]
39
40          GetFileMI() * [False]
41          |Initialisation("filemitest")*[False]
42
43
44
45
46
47      # dans l'impl  => MessageReceiverimpl
48      net MessageReceiver():
49          #     la r ception d'un message du r seau
50          #on consulte les continuations enregistr es
51          #tester si le FILEMI est vide appeller onException sinon onResult
52          buffer MH : object = ()
53
54          #instanciation de l'objet Message Handler
55          [MH+(Instance("Peer.MessageHandler"))]
56
57          # si le message est une r ponse on fait appel
58          #   la m thode treatResponse
59          ; ([MessageResponse-("FileMI Response",sid,UID,FILEMI)
60            ,continuations-(UID,continstance), MH?(mh)
61            ,calls+("call treatresponse ", mh, UID,continstance, FILEMI)]
62          ; ([MH-(mh), calls-("return treatresponse",mh,"void")]
63             + [MH-(mh), calls-("except treatresponse",mh,erreur)]))
64
65
66          #si le message est une requ te on fait appel
67          #    la m thode request
68          +([MessageResponse-(UID,src,fid,"GetFileMI"),MH?(mh)
69            ,calls+("call treatrequest",fid,src, UID,"Get FileMI")]
70          ; ([MH-(mh), calls-("return treatrequest",mh,"void")]
71             + [MH-(mh), calls-("except treatrequest",mh,erreur)]))
72
73      #net message handler , ce lui qui prend en  main le message
74      #r ponse et le traite?
```

```
1
2     net MessageHandler():
3
4         buffer MH : object = ()
5         buffer BF : object = ()
6
7
8         #m thode treatmessage traite le message re   u
9         #cr e un nouveau job de lecture
10        #et execute la continuation
11        net treatresponse():
12            [calls-("call treatresponse ", mh,UID,continstance, FILEMI)
13             ,MH+(mh)]
14            ;[BF+(Instance("peer.ReaderJob"))]
15            ;[BF?(bf),calls+("call ByteToFileMI",bf,continstance,FILEMI)]
16            ; ([MH-(mh),BF-(bf),calls-("return ByteToFileMI",bf,"void")
17                ,calls+("return treatresponse",mh,"void")]
18            + [MH-(mh),BF-(bf),calls-("except ByteToFileMI",bf,"void")
19               ,calls+("except treatresponse",mh,erreur)])
20
21        net treatrequest():
22            #si la requete est autre qu'un GetFragData
23
24            #il execute la continuation fait appel au reader job
25            #et lui passe pour continuation la m thode
26            #finalymessagereceper
27            [calls-("call treatrequest ",fid, src, UID,"Get FileMI"),
28             calls +("call FinalysMessageReceper",fid,src,UID,"GetFileMI"),
29             MH+(mh)]
30            ;([MH-(mh),calls-("return FinalyseMessageReceper","void"),
31             calls+("return treatrequest",mh,"void")]
32            + [MH-(mh),calls-("except FinalyseMessageReceper",erreur),
33             calls+("except treatrequest",mh,erreur)])
34
35        net FinalyseMessageReceper():
36
37            buffer storerID : object = ()
38            buffer UID : object = ()
39
40            [calls-("call FinalysMessageReceper",fid,src,UID,"GetFileMI"),
41             storerID+(src),UID+(UID),
42             calls+("call GetFileMI",fid)]
43            ;([calls-("return GetFileMI",FILEMI),UID-(UID),storerID-(sid),
44              MessageResponse-("FileMI Response",sid,UID,FILEMI),
45               calls+("return FinalyseMessageReceper","void")]
46              +[calls-("except GetFileMI",erreur),UID-(UID),storerID-(sid),
47              MessageResponse-("FileMI Response",sid,UID,erreur),
48               calls+("except FinalyseMessageReceper",erreur)])
49
50        treatresponse() * [False]
51        |treatrequest()*[False]
52        |FinalyseMessageReceper()*[False]
53
54
55    #net du readerjob , lit le message
56    #cr e le FILEMI et excute la continuation sur le FILEMI
57    net ReaderJob():
58
59        buffer RJ : object = ()
60        buffer C : object = ()
61
62        #lis les octets sur le selector les transforme en FILEMI
63        # puis execute la continuation sur le r sultat.
64        net ByteToFileMI():
65
66            #appelle la m thode on result si le filemi est valide
67            #appelle la methode on exception si le filemi est vide
68            #ou pas valide
69            [calls-("call ByteToFileMI ",bf,continstance, FILEMI)
70             ,RJ+(bf),C+(continstance)]
71
72            #appel    onResult si FILEMI PAS VIDE
73            ; ([RJ?(bf),C?(continstance)
74                ,calls+("call onResult", continstance, FILEMI) if(FILEMI)]
75              + [RJ?(bf),C?(continstance),
76                 calls+("call onException",continstance,"FILEMI vide")])
77            ;([C-(c),RJ-(r),calls-("return onResult",c,"void")
78                ,calls+("return ByteToFileMI",r,"void")]
79            + [C-(c),RJ-(r),calls-("except ByteToFileMI",c,"void")
80               ,calls+("except ByteFileMI",r,erreur)])
81
82        ByteToFileMI() * [False]
```

```
1
2       #net message sender, il envoie les messages aux pairs concern s
3       net MessageSender():
4           net send () :
5               buffer objets : object = ()
6               # la m thode messagesender
7               #le message cr e par la m thode getFileMI
8               #dans le r seau et pui il est achemin       son destinataire.
9
10              [objets?(this),
11               calls-("call messagesender",this,Fid)
12               ,SentMessage+(UID,StorerId,Fid,"GetFileMI")
13               if this.cls == "Peer.Client__messagesender" ]
14              #rentourne void      la m thode qui l'a appel e
15              ;([calls+("return messagesender",this,"void")]
16                +[calls+("except messagesender"
17                         ,this,"erreur de reception message")])
18          send() * [False]
19
20
21      #net user
22      net User () :
23          #buffer pour stocker l'instance de l'objet client
24          buffer client : object = ()
25
26          [client+(Instance("Peer.Client"))]
27          #appel    la fonction GetFileMI pour le fileid Fid
28          #pour le client d'id c.
29          ; [client?(c)
30             , calls+("call GetFileMI", c, Fid)]
31          ;([client?(c), calls-("return GetFileMI", c, "void")]
32            +[client?(c), calls-("except GetFileMI",c,erreur)])
33
34
35      #classe Client du net Peer
36      net Client():
37
38          buffer MessageSenderObject : object = ()
39
40          #m thode GetFileMI de la classe client
41          #elle cr e un message et fait appel au message sender
42          #pour envoyer le message et enregistrer une continuation.
43
44          net GetFileMI():
45               buffer objectid : object = ()
46               buffer FileId : object = ()
47               buffer MessageSenderObject : object = ()
48
49               [calls-("call GetFileMI", this, Fid), objectid+(this),
50                FileId+(Fid) if this.cls == "Peer.Client"]
51
52               #appel au messagesender qui va envoyer le message dans
53               #le r saux et sauvegarde de la continuation dans le
54               #buffer des continuations. on ne mod lise pas
55               #GetFileMIMessage mais on passe les infos qu'il contient
56               ;[MessageSenderObject?(msgsnd), FileId-(Fid),
57                 NextUID-(UID), NextUID+(UID+1),
58                 calls+("call send",msgsnd,UID,Fid,"GETFILEMI")]
59
60               # r cup re le retour de la m thode appell e
61               # deux choix de r ponse : soit un return qui redonne la main
62               #       la m thode appellante pour continuer son ex cution
63               # ou bien une exception qui interompt l' xecution   du programme
64               #r cup re la valeur retour de la m thode message sender
65               # et renvoie un valeur retour       la m thode appelante
66               # soit GetFileMI
67               ;([objectid-(this),
68                  MessageSenderObject?(msgsnd),
69                  calls-("except send",msgsnd,error)
70                  , calls+("except GetFileMI",this
71                           ,("Erreur GetFile",) + error)]
72
73                 #met dans le buffer Concitnuations l'idendit  de l'appelant
74                 #et l'identit  de la continuation cr e
75                 #ne pouvant pas passer les continuations comme param tre
76                 #on a d finit les continuations comme une classe      part
77                 #et on cr e une instance      chaque fois qu'on veut les utiliser
78                 #l'identit  de l'objet cr e est ensuite sauvegard e
79                 # dans le buffer FileMIContinuation
80
81                 + [objectid-(this),
82                    MessageSenderObject?(msgsnd),
83                    calls-("return snd",msgsnd,"void"),
84                    continuations+(UID,Instance
85                                   ("Peer.Client__GetFile__FileMIContinuation")),
86                    calls+("return Peer.Client.GetFileMI",this,"void")])
```

```
1
2           # main du net Client , les m thodes tournent ind finiment en attendant
3           #d'  tre    appel es
4           [MessageSenderObject+(Instance("Peer.MessageSender"))]
5           ;(GetFileMI() * [False])
6
7       # Peer
8       (Client()
9        | Client__GetFileMI__FileMIContinuation()
10       | User()
11       | MessageReceiver()
12       | MessageSender()
13       | MessageHandler()
14       | ReaderJob())
15
16  Peer(1,2)|Peer(2,3)|Peer(3,1)
```

Bibliographie

[1] http://www.fing.org/index.php%20?num=3297,2.

[2] Acs - trust ant colony system. http://ants.dif.um.es/~felixgm/research/tacs.

[3] Algorithme de colonies de fourmise. http://fr.wikipedia.org/wiki/Algorithme_de_colonies_de_fourmis.

[4] The gnutella protocol specification v0.4.

[5] Jforum des droits sur l'internet. http://www.foruminternet.org/texte/publications/lire.phtml?id=581.

[6] Mojo nation. http://www.mojonation.net.

[7] Napster. http://fr.wikipedia.org/wiki/Napster.

[8] Résumé sur le routage à vecteur de distance. http://cisco.goffinet.org/s2/vecteur_distance.

[9] SPREADS project. http://www.spreads.fr.

[10] *PAST : A large-scale, persistent peerto-peer storage utility*, IEE proceedings, 2001.

[11] *Making gnutella-like P2P systems scalable*. ACM Press, 2003.

[12] *An analysis of the skype peer-to-peer internel telephony protocol*. IEEE Computer society, 2006.

[13] *Optimising the compilation of Petri net models*, volume 726 of *Workshop proceedings*. CEUR, 2011.

[14] *Tapestry : An infrastructurefor fault-tolerant wide-area location and routing*, Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[15] R. Anderson. The eternity service. In *In Proceedings of the 1st International Conference on the Theory and Applications of Cryptology*, 1996.

[16] Christopher Batten, Kenneth Barr, Arvind Saraf, and Stanley Trepetin. Pstore a secure P2P backup system. Projet Pstore, 2001.

[17] David Bindel and al. Oceanstore : An extremely widearea storage system. Technical Report UCB/CSD-00-1102, Computer Science Division (EECS) ; University of California ; Berkeley, California 94720.

[18] A. Tajeddine ; A. Kayssi ; A. Chehab and H. Artail. Patrol-f - a comprehensive reputation based trust model with fuzzy subsystems. 2006 ; Springer.

[19] Y. Zhang ; H. Chen and Z. Wu. A social network-based trust model for the semantic web. 2006 ; Springer.

[20] Cholez and al. A distributed and adaptive revocation mechanism for P2P networks. In *Proc. of the 7th International Conference on Networking*. IEEE Computer Society, 2008.

[21] M. Kaashoek F. Dabek ; J. Li ; E. Sit ; J. Robertson ; and R. Morris. Designing a dht for low latency and high throughput. In *Symposium on Networked Systems Design and Implementation*, 2004.

[22] F. Gomez and G. Martinez. *State of the Art in Trust and Reputation Models in P2P networks*. Springer.

[23] Y. Wang ; V. Cahill ; E. Gray ; C. Harris and L. Liao. Bayesian network based trust management. 2006 ; Springer.

[24] R. Chen ; X. Chao ; L. Tang ; J. Hu and Z. Chen. Cuboidtrust : A global reputation-based trust model in peer-to-peer networks. 2007 ; Springer.

[25] Laflaquière Julien. Les « autres » applications des technologies peer-to-peer. `news.cnet.com/2100-7344-5169894.html`.

[26] I. Stoica ; R. Morris ; D. Karger ; M. Kaashock ; and H. Balakrishman. Chord : A scalable peer-to-peer lookup protocol for internet applications. 2001.

[27] D. Mazières ; M. Kaminsky ; F. Kaashoek ; and E. Witchel. Separating key management from file system security. In *ACM SOSP*, 1999.

[28] K. Fu ; M. F. Kaashoek ; and D. Mazières. Fast and secure distributed read-only file system. In *In Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, 2000.

[29] Sylvia Ratnasamy ; Paul Francis ; Mark Hetley ; Richard Karp and Scott Shenker. A scalable content addressable network. In *Proceedings ACM SIGCOMM 200*, 2001.

[30] Jian Liang ; Rakesh Kumar ; and Keith W. Ross. Understanding kazaa `http://cis.poly.edu/~ross/papers/UnderstandingKaZaA.pdf` , 2004.

[31] Lesueur and al. Detecting and excluding misbehaving nodes in a P2P network. *Studia Informatica Universalis*, 7(1), 2009.

[32] R. Mahajan, M. Castro, and A. Rowstron. Controlling the cost of reliability in peer-to-peer overlays. In *Proc. of IPTPS'03*, 2003.

[33] F. Gomez ; G. Martinez and A. F. Skarmeta. Tacs, a trust model for P2P networks. 2007 ; Springer.

[34] Oliver Heckmann ; Axel Bock ; Andreas Mauthe ; and Ralf Steinmetz. The edonkey file-sharing network. In Peter Dadam and Manfred Reichert, editors, *Workshop on Algorithms and Protocols for Efficient Peer-to-Peer Applications (Informatik)*, volume volume 51 of LNI, 2004.

[35] Petar Maymounkov and David Mazières. The edonkey file-sharing network. In Peter Druschel ; M. Frans Kaashoek ;, Antony I, and T. Rowstron éditions, editors, *st International Workshop on Peer-to-Peer Systems(IPTPS)*, volume volume 2429 of Lecture Notes in Computer Science. Springer, 2002.

[36] S. Legtchenko ; S. Monnet, P. Sens, and G. Muller. Relaxdht : a churn-resilient replication strategy for peer-to-peer distributed hash-tables. 2011.

[37] F. Dabek ; M. F. Kaashoek ; D. Karger ; R. Morris ; and I. Stoica. Wide-area cooperative storage withcfs. In *In Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[38] Frank Dabek ; M. Frans Kaashoek ; David Karger ; Robert Morris and Ion Stoica. Wide-area cooperative storage with cfs. In *Symposium on Operating Systems Principles*, pages 202–205, 2001.

[39] Michel Morvan and Sylvain Sené. A distributed trust diffusion protocol for ad hoc networks. In *Proc. of ICWMC'06*. IEEE Press, 2006.

[40] N.Six. Jdnet, grid computing : l'union fait la force. `http://solutions.journaldunet.com/0212/021211_grid.shtml`.

[41] James S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. In *Software Practice and Experience3*, 1997.

[42] Franck Pommereau. Quickly prototyping Petri nets tools with SNAKES. *Petri net newsletter*, October 2008.

[43] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *SIAM journal on applied mathematics*, 8(2), 1960.

[44] Antony Rowstron and Peter Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Symposium on Operating Systems Principles*. IFIP/ACM, 2001.

[45] Smith S. Secure coprocessing applications and research issues. Technical Report Unclassified Release LAUR -96-2805, Los Alamos National Laboratory, 2003.

[46] Sam Sanjabi and Franck Pommereau. Modelling, verification, and formal analysis of security properties in a P2P system. In *Proc. of COLSEC'10*, IEEE Digital Library. IEEE, 2010.

[47] O. Kamvar ; M. Schlosser and H. Garcia-Molina. 2003.

[48] Sylvain Sené. Modèle de diffusion de la confiance dans les réseaux ad hoc. Technical report, 2005.

[49] F. Picconi ; J.-M. Busca ; P. Sens. Pastis : a highly-scalable multi-user peer-to-peer file system. EuroPar 2005.

[50] F. Picconi ; J.-M. Busca ; P. Sens. An experimental evaluation of the pastis peer-to-peer file system under churn. INRIA RR-6114, 2007.

[51] Dinesh C. Sharma. Lindows routes os over file-sharing networks. `http://news.cnet.com/2100-7344-5169894.html`.

[52] B. Gassend ; D. Clarke ; M. van Dijk ; S. Devadas and E. Suh. Caches and merkle trees for efficient memory authentication. In *Proc. of the 9th High Performance Computer Architecture Symposium (HPCA'03)*, 2003.

[53] H. Tian ; S. Zou ; W. Wang and S. Cheng. A group based reputation system for P2P networks. 2006 ; Springer.

[54] Atul Adya ; William J. Bolosky ; Miguel Castro ; Gerald Cermak ; Ronnie Chaiken ; John R. Douceur ; Jon Howell ; Jacob R. Lorch ; Marvin Theimer ; Roger P. Wattenhofer. Farsite : Federated, available, and reliable storage for an incompletely trusted environment.

[55] M. Welsh and D. Culler. Jaguar. aguar : Enabling efficient communication and i/o from java. In *Concurrency : Practice and Experience, Special Issue on Java for High-Performance Applications*, 1999.

[56] L. Xiong and L. Liu. Peertrust : Supporting reputation-based trust in peer-to-peer communities. 2004.

[57] Y. Wang ; Y. Tao ; P. Yu ; F Xu and J. Lu. A trust evolution model for P2P networks. 2007 ; Springer.

[58] F. Yu ; H. Zhang ; F. Yan and S. Gao. An improved global trust value computing method in P2P system. 2006 ; Springer.

[59] W. Wang ; G. Zeng and L. Yuan. Ant-based reputation evidence distribution in P2P networks. 2006.

[60] T. Zhuo ; L. Zhengding and L. Kai. Time-based dynamic trust model using ant colony algorithm. 2006.