N° d'ordre : 3081

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

Gwendal Simon

Équipe d'accueil : Adept
École Doctorale : Matisse
Composante universitaire : IRISA

Titre de la thèse :

Conception et réalisation d'un système pour environnement virtuel massivement partagé

soutenue le 10 décembre 2004 devant la commission d'examen

M. :	Daniel	Herman	Président
MM. :	Pierre	Fraigniaud	Rapporteur
	Pascal	Felber	
MM. :	Michel	Raynal	Examinateur
	Joaquìn	Keller	
	Gerardo	Rubino	

Remerciements

Ce travail a été réalisé à France Telecom R&D à Issy les Moulineaux et à l'IRISA, au sein de l'équipe ADEPT. Je remercie mes responsables à France Telecom R&D qui m'ont fait confiance et m'ont accordé les moyens matériels et financiers requis pour la réalisation de ce travail.

Je tiens tout d'abord à remercier M. Daniel Herman qui m'a fait l'honneur de présider le jury de cette thèse. Je remercie également M. Pascal Felber et M. Pierre Fraigniaud d'avoir bien voulu accepter la charge de rapporteur. Je remercie M. Gerardo Rubino qui a été à l'initiative de ma vocation. Il m'a permis de démarrer cette thèse et a gracieusement accepté de juger ce travail.

J'adresse mes remerciements à M. Michel Raynal dont les cours ont éveillé ma curiosité et qui a accepté d'être mon directeur de recherche. Je remercie également M. Joaquin Keller, à l'initiative de ce projet à France Telecom R&D. Son enthousiasme, ses compétences scientifiques et sa créativité sans limite ont grandement contribué à cette thèse.

Je remercie particulièrement Mme Emmanuelle Anceaume et Mlle Maria Gradinariu, qui m'ont encadré et guidé avec clairvoyance. Elles ont été les instigatrices de nombreux travaux décrits dans cette thèse.

Je n'aurais pas eu la chance d'être là où j'en suis si mes parents n'avaient pas fait preuve d'une grance patience à mon égard. Leur soutien permanent est exemplaire.

Je remercie mes amis Sophie, Hervé, Nicolas et Nolwenn qui ont rendu mes déplacements à Rennes beaucoup plus agréables. Je tiens à remercier plus spécialement Julien pour son hospitalité et ses discussions variées mais toujours rigoureuses.

Enfin, je n'aurais certainement pas pu réaliser cette thèse sans le soutien de Géraldine, qui m'a notamment offert la stabilité indispensable à la réalisation de ce travail.

Table des matières

Je me doutais que le style néo-extravagant de ce quartier était stupéfiant, mais je ne m'attendais pas à une telle foule. Dommage que ce foutu monde soit toujours aussi plat.

Entre deux saltimbanques géants, j'aperçois une connaissance qui, machinalement, me fait signe. Le dialogue s'établit sans mal et, rapidement, il me propose de le suivre : un musicien célèbre est en représentation exceptionnelle non loin d'ici.

Pressé, il rejoint ses amis et, subitement, je le perds de vue dans la foule. Grâce au marqueur qu'il a pris soin de me confier, je le rattrape aisément mais son groupe d'amis semble s'être étiolé. La liaison transatlantique a des ratés paraît-t-il. Ils ne tarderont pas à revenir.

Il n'est pas rare de voir des personnages se superposer. Certains semblent ne pas se rendre compte qu'ils ont deux à trois ans de retard et que ce retard devient gênant.

La foule se densifie encore. Mon ami propose de se téléporter directement à l'endroit du concert. Malheureusement, je ne dispose pas de la dernière version du logiciel. Mon ordinateur me le fait comprendre. Dommage, cet artiste n'est pas souvent en concert virtuel...

Chapitre 1

Introduction

Recréer, virtuellement, la réalité ou, du moins, un environnement qui ressemblerait au monde réel, est un défi qui passionne des informaticiens depuis longtemps. Les capacités des machines permettent maintenant de représenter graphiquement des mondes en trois dimensions qui sont des copies conformes de notre monde réel. Malheureusement, les expériences d'immersion demeurent rares et ces mondes virtuels sont peu peuplés, voire inhabités.

Un nouveau défi passionnant consiste à élaborer un environnement virtuel qui pourrait être habité par un grand nombre d'utilisateurs. Dans "*Snowcrash*" [91], Neal Stephenson dépeint le *Metaverse*, monde virtuel massivement partagé, et souligne les nombreuses nouvelles formes d'interaction que ces environnements peuvent susciter.

A ce jour, aucun système ne permet à un nombre *illimité* de participants de s'immerger simultanément dans un tel univers virtuel. Dans cette thèse, nous décrivons un système qui résout ce problème.

1.1 Contexte

Un environnement virtuel partagé est un espace généré par ordinateur(s) dans lequel plusieurs participants peuvent se rencontrer et interagir de telle manière que l'expérience vécue ressemble à celle qu'ils pourraient vivre dans le monde réel.

On appelle **entité** n'importe quel objet indépendant dans le monde virtuel. Il s'agit d'un programme exécuté par une machine. Quand une entité est contrôlée par un humain, on dit que la représentation de l'entité dans le monde virtuel est l'avatar de l'humain qui la contrôle. On parle plutôt d'objet virtuel pour une entité contrôlée par l'ordinateur.

Chaque entité possède une position dans le monde virtuel. C'est à cet endroit, virtuel, que sa représentation est observable par les autres entités. Traditionnellement, les mouvements sont autorisés : une entité peut bouger d'un point virtuel du monde à un autre. Par ailleurs, une entité peut apparaître dans un monde virtuel et en disparaître, aussi facilement que la machine qui l'héberge peut se mettre en marche ou s'arrêter. Un monde virtuel est donc dynamique : il se modifie au gré des événements qui l'affectent.

Les environnements partagés impliquent plusieurs machines hébergeant une ou plusieurs entités susceptibles de modifier le monde virtuel à chaque instant. Ces machines disposent de la capacité de communiquer entre elles par des messages circulant sur un réseau, typiquement l'Internet.

La définition d'un système de communication pour une application telle qu'un monde virtuel massivement partagé comprend la gestion des liens de communication entre les entités et la gestion des messages échangés. Le système mis en place doit être conforme aux contraintes du réseau sur lequel repose l'application et doit respecter les spécifications de l'application.

Deux spécifications nous semblent fondamentales pour un monde virtuel massivement partagé : l'aptitude à passer à l'échelle et la cohérence.

Facteur d'échelle : On considère qu'une application passe à l'échelle si son comportement n'est pas altéré lorsque le nombre de participants varie de plusieurs ordres de grandeurs (d'une dizaine à plusieurs millions d'utilisateurs). On utilisera par la suite le néologisme *scalable* pour définir une application qui passe à l'échelle. Une application scalable se caractérise, entre autres, par une consommation des ressources physiques individuelles (le débit d'arrivée des messages, l'occupation de mémoire ou la charge de l'unité de calcul) indépendante du nombre total de participants.

Cohérence : On dit qu'un environnement virtuel partagé est cohérent lorsqu'une scène virtuelle est perçue de manière identique par tous les participants qui l'observent. L'architecture d'un environnement virtuel partagé cohérent doit permettre à une entité désirant relater un événement de transmettre un message de telle manière que toutes les entités intéressées par cet événement le traitent simultanément.

A notre connaissance, aucun système de réalité virtuel partagé ne garantit, à la fois, la cohérence et le passage à l'échelle.

Les architectures les plus courantes reposent sur des serveurs centraux. En relation avec toutes les entités, ils disposent d'une connaissance globale du monde. A la réception d'un message relatant un événement, ils doivent déterminer les entités qui sont intéressées par cet événement puis leur transmettre le message. Ce mécanisme garantit une cohérence forte. Malheureusement, le nombre de messages que ces serveurs doivent traiter dépend du nombre de participants. De plus, ils peuvent être victime de défaillances susceptibles d'interrompre l'environnement virtuel.

Les architectures décentralisées sont destinées à corriger ces défauts. Chaque entité est directement connectée aux entités susceptibles de générer un événement dans la scène virtuelle observée. Comme la taille de cette zone est limitée, il est admis qu'une entité s'intéresse à un petit nombre d'entités, généralement indépendant du nombre total de participants. En conséquence, le nombre de messages simultanés que chaque entité doit traiter est à peu près constant, ce qui permet le passage à l'échelle. Malheureusement, la gestion de la mise en relation des entités reste un problème. Le plus fréquemment, une infrastructure de serveurs est chargée de détecter les entités les plus proches virtuellement et de permettre l'établissement de connexions entre elles. Même si le nombre de participants peut être élevé, le système n'est plus scalable. En outre, le déploiement et la maintenance de cette infrastructure génèrent un coût financier important tandis que le système est toujours soumis aux risques de défaillances.

Dans cette thèse, nous proposons que la responsabilité de la mise en relation des entités soit partagée par les participants eux-mêmes, collaborant dans un système appelé réseau

de pairs.

Un réseau de pairs est un moyen de structurer une application distribuée de telle manière que chaque participant possède un rôle identique : un **nœud** agit à la fois comme client et comme serveur. Il possède des connexions directes avec d'autres nœuds qui lui permettent de récupérer des informations et d'en offrir.

Le principal avantage des réseaux de pairs est leur autonomie : aucun contrôle centralisé n'est nécessaire à son fonctionnement. De fait, ces systèmes ne dépendent pas des capacités physiques et de tolérance aux défaillances d'une seule machine. L'organisation d'un réseau de pairs émerge de l'action coordonnée de tous les pairs qui le composent. Les comportements individuels ont un impact local qui produit une organisation globale.

1.2 Axe de recherche

Associer les mondes virtuels partagés et les réseaux de pairs est une démarche novatrice dans le prolongement de notre volonté de regrouper plusieurs communautés disjointes au sein d'une seule et même communauté sur le thème des systèmes distribués dynamiques [47]. Sont inclus les réseaux basés sur la communication sans-fil, les réseaux de pairs, les nano-systèmes ou encore les systèmes de réalité virtuelle.

Actuellement, les antennes incorporées dans de petites unités de calcul permettent d'établir une communication hertzienne entre deux machines si la distance qui les sépare n'est pas trop importante. La démocratisation de ces objets communicants provoque un engouement pour les réseaux dits sans-fil. Pour permettre à n'importe quels participants d'établir une communication, deux voies sont explorées : les réseaux cellulaires et les réseaux ad-hoc.

Dans les réseaux cellulaires, une station gère une partie du monde appelée *cellule*. Les stations interconnectées se chargent de relayer tous les messages émis par les machines se trouvant dans leur cellule vers le destinataire, directement s'il se trouve dans la cellule, ou via d'autres stations.

Les réseaux ad-hoc se caractérisent par l'absence d'infrastructure spécifique. A la différence des réseaux cellulaires, les communications ne dépendent pas de stations interconnectées. Deux participants peuvent communiquer, soit par une liaison sans-fil directe, soit par une séquence de liaisons sans-fil mettant en jeu un ou plusieurs nœuds intermédiaires. Chaque participant peut relayer un message, agissant ainsi en routeur et permettant à un réseau de se former.

Un système coopératif de robots et de nano-robots consiste en une collection de robots communicants amenés à former une équipe organisée de manière cohérente. Le déplacement des participants dépend des communications. Les nano-systèmes se caractérisent, entre autres, par le fait que les nano-robots peuvent avoir la capacité de se reproduire.

Les réseaux de pairs (ou réseaux *peer-to-peer*) visent à distribuer massivement le coût du partage de données. A la différence des systèmes précédents, ils sont basés sur Internet, réseau permettant à n'importe quelle paire de participants de communiquer à la condition que l'un d'eux connaisse l'adresse de l'autre.

Les systèmes de réalité virtuelle possèdent des caractéristiques qu'on retrouve dans tous ces systèmes distribués.

Le nombre de participants : Tous ces systèmes visent un très grand nombre d'utilisateurs. Une préoccupation majeure est donc le passage à l'échelle.

Les positions : Un participant se caractérise par une position dans un espace. Pour les réseaux sans-fil, la position correspond à la position physique de la machine tandis que, dans la réalité virtuelle, on associe la position virtuelle de l'avatar à l'entité. Dans ces deux cas, les coordonnées ont trois dimensions. Les systèmes de réseaux de pairs n'admettent pas ces contraintes. Pourtant, ils possèdent généralement une notion de position qui dépend de l'application. Ainsi, les applications les plus récentes attribuent à chaque participant une clé dans un espace virtuel à d dimensions.

L'organisation : Une contrainte physique (la distance limitée des communications hertziennes) ou la volonté de créer une application scalable sont à l'origine d'une caractéristique importante : chaque participant est directement connecté avec un nombre restreint d'autres participants. De fait, un nœud ne possède qu'une vue locale du système. Dans les réseaux sans-fil et la réalité virtuelle, les liaisons directes dépendent de la position des nœuds alors que les réseaux de pairs n'admettent aucune contrainte dans le choix des liaisons.

La mobilité : Un utilisateur peut décider de se déconnecter d'un système à chaque instant, comme un nouveau participant peut décider de se connecter à n'importe quel moment. Le nombre de participants évolue dans le temps. De plus, la position des utilisateurs peut changer, bouleversant l'organisation du réseau. Ainsi, lorsqu'un objet communicant bouge physiquement, certaines liaisons avec ces voisins peuvent disparaître et d'autres se former.

Les problèmes communs : Les concepteurs de réseaux sans-fil, de systèmes de réalité virtuelle ou de réseaux de pairs sont confrontés à quelques problèmes fondamentaux identiques. Ainsi, la découverte de ressources permet d'initialiser une communication dans un réseau sans-fil, mais également de rechercher une donnée dans un réseau de pairs. Ou, encore, le risque de partitionnement du réseau risque de faire échouer une communication entre deux participants ou de rendre une partie des données partagées inaccessibles à certains utilisateurs. Généralement, les solutions proposées reposent sur les mêmes principes.

Le système de réalité virtuelle que nous avons élaboré se situe dans la communauté des réseaux mobiles massivement distribués.

1.3 Contributions

Pour concevoir un système de réalité virtuelle basé sur un réseau de pairs, nous avons tout d'abord cherché à définir, à partir des spécifications de l'application, les propriétés globales du réseau de pairs. Puis, nous avons conçu des comportements individuels qui, reproduits par l'ensemble des participants, permettent d'aboutir aux propriétés attendues.

Le chapitre 2 décrira cette modélisation et présentera les algorithmes permettant à une machine d'adopter le comportement désiré et de participer au monde virtuel. Le réseau

ainsi formé constitue SOLIPSIS, un système de communication pour un monde virtuel massivement partagé pouvant accueillir un nombre *a priori* illimité de participants.

Il est d'ores et déjà possible de participer à SOLIPSIS [55]. Dans le chapitre 3, nous décrivons la réalisation de SOLIPSIS. En particulier, nous présentons les principaux algorithmes utilisés, l'architecture des programmes ainsi que le format des messages échangés. De plus, la proximité virtuelle entre deux participants peut déclencher différentes interactions multimédias gérées par d'autres applications utilisant SOLIPSIS. Nous décrivons dans ce chapitre les relations entre une entité de SOLIPSIS et ces applications.

Il est également possible d'imaginer des applications plus complexes qui utiliseraient le système de mise en relation de SOLIPSIS pour concevoir d'autres organisations répondant à des besoins différents ou complémentaires de l'application de réalité virtuelle partagée. Il nous a paru intéressant d'étudier deux applications qui nous semblent primordiales dans un système de communication à grande échelle.

Service de recherche : Dans un système utilisé par un très grand nombre de participants et dans lequel chaque utilisateur ne dispose que d'une vue partielle du monde, il n'est pas aisé de retrouver une entité particulière. Il s'agit d'offrir à chaque participant la capacité de détecter des entités répondant à certains critères. Ce problème a des spécifications formelles auxquelles répondent plusieurs algorithmes plus ou moins coûteux en ressources. Dans le chapitre 4, nous étudions ce problème, nous présentons les algorithmes existants, puis nous proposons de nouveaux algorithmes permettant une recherche efficace et moins coûteuse grâce à une organisation en réseau de pairs adaptée au problème.

Publication/Abonnement d'informations : Il s'agit de permettre aux entités de disséminer une information à toutes les entités ayant exprimé un intérêt pour des informations de ce type. Pour cela, il est nécessaire de définir un mécanisme permettant aux entités d'exprimer un intérêt (l'abonnement), puis de mettre en place des mécanismes afin que les informations répondant à cet intérêt soient transmises à toutes les entités abonnées. Dans le chapitre 5, nous étudions formellement ce problème, puis nous proposons des solutions basées sur une organisation en réseau de pairs qui utilise le système de mise en relation de SOLIPSIS.

Enfin, nous conclurons ce document en résumant ces recherches et en proposant de nouvelles orientations pour des travaux futurs autour de SOLIPSIS

1.4 Thèse défendue

Jusqu'à présent, aucun système ne permet à un nombre illimité de participants de se côtoyer dans un environnement virtuel attractif. Les solutions proposées ne permettent pas le passage à l'échelle, car elles se basent sur un ou plusieurs serveurs. Outre le coût de leur maintenance qui nécessite de rendre l'accès à ces mondes payant, ces serveurs sont des limitations physiques - le nombre de participants reste borné - et intellectuelles - le propriétaire des serveurs est le propriétaire du monde virtuel.

Dans ce document, nous montrons qu'il est possible de concevoir et réaliser un monde virtuel dans lequel les participants sont les seuls acteurs du système.

Première partie

Solipsis

Chapitre 2

Principes fondamentaux

Dans ce chapitre, nous présentons les principes fondamentaux de Solipsis [68, 69]. D'un point de vue architectural, ce système possède peu de points communs avec les travaux existants dans ce domaine. C'est pourquoi il nous a paru important de décrire ceux-ci afin de mettre en lumière les difficultés auxquelles ces systèmes sont confrontés. Cette étude est l'objet de la Section 2.1.

Ensuite, nous définirons quelques concepts basiques de la réalité virtuelle, puis nous présenterons les propriétés globales du système que nous cherchons à réaliser.

Enfin, nous introduirons deux mécanismes qui sont à la genèse du système Solipsis. Nous montrerons que ces mécanismes permettent d'obtenir un système proche de celui que nous voulons obtenir. En outre, nous présenterons un algorithme qui permet à une entité désirant participer au monde virtuel de se connecter au réseau de pairs sur lequel repose Solipsis.

2.1 Préambule

Ce préambule survole les différents travaux académiques liés au sujet de la réalité virtuelle partagée. Nous nous sommes particulièrement intéressés aux systèmes basés sur une architecture distribuée. Ces projets poursuivent les mêmes buts que Solipsis et visent notamment la cohabitation d'un nombre illimité de participants.

Ensuite, nous introduirons sommairement la théorie des graphes et montrerons que des travaux issus de ce domaine de recherche ont permis l'élaboration d'algorithmes efficaces pour les réseaux hertziens ad-hoc. Même s'ils ne sont pas immédiatement applicables à Solipsis, ces travaux ont été une source d'inspiration du système que nous avons conçu et ils nous ont offert certains outils mathématiques cruciaux.

2.1.1 Architectures pour environnements virtuels distribués

Les premiers systèmes distribués visant à simuler une réalité virtuelle massivement partagée sont issus de la *Naval Postgraduate School of Monterey* [75, 104].

Le projet SIMNET est un système d'entraînement militaire développé dans les années 1980. Il permet d'apprendre à s'organiser et à se battre en équipe grâce à un environnement virtuel constitué de petites unités identiques. Ce projet a conduit au développement du

protocole DIS [4], un ensemble de normes définissant, entre autres, un format d'échange de données ainsi qu'un certain nombre d'entités et leurs interactions.

Les systèmes développés dans ce projet fonctionnent sur le principe suivant. Le monde virtuel est décrit par un ensemble de données : les caractéristiques des entités, leurs positions et déplacements virtuels, leurs interactions...

Chaque participant possède une copie de cette base de données et dispose donc d'une connaissance totale du monde virtuel. Pour garantir la cohérence du monde virtuel, il est nécessaire que tous les participants possèdent, à un instant donné, la même copie de la base de données du monde virtuel. Il faut donc que les événements virtuels, décrits par des messages, puissent être traités, simultanément, par tous les participants du monde virtuel. La diffusion de ces messages est réalisée par une *communication multipoint*. Un mode de communication multipoint assure qu'un message, émis par une entité, est reçu, simultanément, par toutes les autres entités. L'émetteur et les récepteurs forment un groupe de communication. La communication multipoint utilisée dans leurs systèmes fonctionnent grâce à l'action des routeurs, les ordinateurs chargés du transfert des messages dans les réseaux.

Cette architecture distribuée novatrice a l'inconvénient de forcer les entités à traiter tous les événements se produisant dans le monde virtuel. Naturellement, le nombre de messages croît avec le nombre d'événements et, sans doute, avec le nombre de participants. Le traitement des messages est coûteux et les contraintes physiques des machines ne permettent pas la cohabitation d'un grand nombre d'entités.

Par la suite, les scientifiques ont cherché à diminuer la quantité d'informations possédées par les participants et à filtrer les messages de manière à réduire la quantité d'événements à traiter. Deux approches ont été proposées : un découpage du monde et une approche basée sur la perception de chaque entité. Nous décrirons ces deux approches, puis nous présenterons les travaux visant à déployer des jeux en réseaux massivement partagés sans serveurs.

Découpage du monde

La mise au point de l'environnement NPSNET-IV [75] fut une avancée significative pour la création de mondes partagés par un très grand nombre d'utilisateurs. Il incorpore le protocole DIS mais également une communication multipoint par IP Multicast [61]. Le protocole IP Multicast permet de définir un grand nombre de groupes de communication. Les groupes sont dynamiques : une entité peut rejoindre et quitter un nombre quelconque de groupes à la demande. En outre, une distinction est réalisée entre les entités *passives* - elles reçoivent tous les messages de ce groupe mais ne peuvent pas en envoyer - et les entités *actives* - la réception et l'émission de message est possible.

L'idée de NPSNET-IV consiste à découper statiquement le monde virtuel en cellules hexagonales. Un groupe de communication est associé à chaque cellule. Une entité est active dans la cellule dans laquelle elle se situe et passive dans les six cellules adjacentes (cf Figure 2.1). Chaque cellule est un monde virtuel plus petit dans lequel le nombre d'entités en interaction est réduit. En conséquence, le nombre d'événements virtuels et donc le nombre de messages à traiter diminuent.

Au gré de ses déplacements, une entité peut franchir la frontière séparant deux cel-

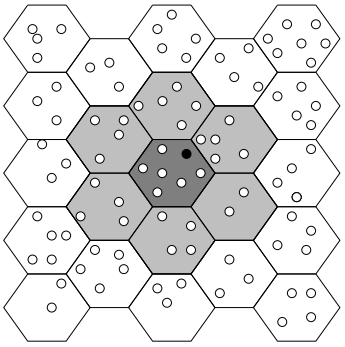

FIG. 2.1 – Partitionnement en cellules hexagonales : l'entité noire reçoit et émet des messages pour le groupe multipoint de la cellule grise foncée, et reçoit des messages des entités situées dans les cellules grises claires. Elle ne reçoit pas de messages des autres entités.

lules adjacentes. Or, elle ne dispose pas des informations lui permettant de se joindre aux groupes associés à ses nouvelles cellules adjacentes. Deux solutions sont avancées pour ce problème. La plus simple se base sur une communication de type client-serveur. Quand une entité passe d'une cellule à une autre, elle interroge un serveur qui lui offre les informations lui permettant de joindre ces groupes. La deuxième solution consiste à nommer un responsable à chaque cellule. Par comparaison avec les autres entités, celui-ci dispose d'informations supplémentaires à propos des cellules adjacentes à la sienne. Par défaut, le responsable est l'entité la plus ancienne de la cellule.

Deux centres de recherche ont, simultanément, enrichi le principe du découpage du monde virtuel et de l'attribution d'un groupe multipoint à chaque cellule.

Le *Mitsubishi Electric Research Laboratory* a élaboré une plate-forme logicielle, nommée Spline, qui permet de développer des applications de réalité virtuelle [16, 15]. Les cellules sont caractérisées par des dimensions, un groupe multipoint et, si elle existe, une cellule englobante vue comme un supérieur hiérarchique. Les auteurs proposent d'ajuster les dimensions des cellules aux éléments du décor virtuel. Une pièce d'un immeuble est une cellule contenue dans un étage qui est, également, une cellule, dont le supérieur hiérarchique est l'immeuble, une autre cellule.

Le *Swedish Institute of Computer Science* a rendu encore plus générique cette démarche avec DIVE [41]. Le monde virtuel est une base de données possédant une organisation hiérarchique correspondant aux arbres de scènes tridimensionnelles utilisés par les infographistes. Des groupes multipoints sont associés à certains nœuds de l'arbre. Tous les éléments dépendants de ce nœud doivent communiquer au sein de ce groupe. L'organisation proposée autorise la duplication de certains sous-arbres : un groupe multipoint peut être englobé par plusieurs groupes multipoints indépendants.

Plus récemment, l'*Université de Nice* a proposé Score, une architecture de communication basée sur un découpage dynamique du monde virtuel [72]. Des serveurs peuvent modifier les dimensions des cellules en fonction de paramètres variés tels que le nombre maximal de groupes de communication disponibles, la vélocité des entités ou encore la densité ou la répartition des entités dans le monde.

Approches basées sur la perception

L'*Université de Nottingham* a été à l'initiative d'une autre approche visant à réduire le flot d'informations entre les entités. Leur système de réalité virtuelle, nommé Massive, a connu trois évolutions majeures.

L'aspect innovant de Massive-1 [48] concerne la gestion du voisinage basé sur la perception des entités [17]. Plusieurs notions issues de l'observation du monde réel ont été définies.

L'**aura** d'une entité est un sous-espace du monde virtuel précisant les limites de la "présence" d'un objet et facilitant les interactions potentielles. Quand les auras de deux objets se chevauchent (même partiellement), le système doit faire rentrer en contact ces deux objets. Cette relation leur permet de s'informer des événements virtuels qu'ils peuvent produire. Une aura peut avoir n'importe quelle taille et n'est pas forcément située autour de son entité.

Le concept de **focus** s'assimile à la zone observée par une entité. Ainsi, l'attention d'une entité se porte sur une région précise du monde virtuel, le focus.

Le **nimbus** peut être considéré comme le sous-espace dans lequel une entité attire l'attention sur elle. Plus une entité est située dans le nimbus d'un objet virtuel, plus elle est intéressée par celui-ci. Par exemple, le nimbus d'un écran de télévision est situé en face de celui-ci.

Ces notions permettent de quantifier précisément l'**intérêt** d'une entité envers une autre. Il dépend de leur focus et de leur nimbus et n'est pas forcément symétrique, comme l'illustre la figure 2.2. Dans la situation de gauche, l'entité A oriente son focus vers B tandis que le nimbus de B contient A. Il en résulte que A est très intéressé par B. Dans la situation au centre, A ne porte pas son attention sur B mais celui-ci essaye d'attirer l'attention de A. L'entité B tente sans doute d'interrompre A mais celui-ci ne l'a pas encore remarquée. Enfin, dans la situation de droite, B est dans le focus de A mais le nimbus de B ne contient pas A. Peut-être que A espionne B ?

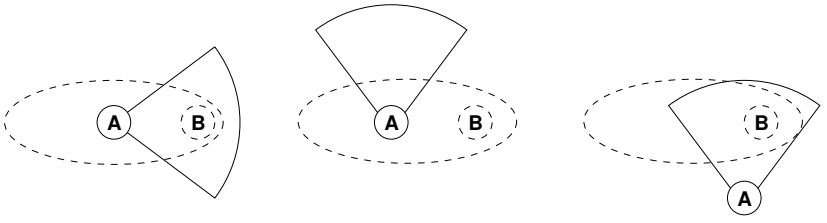

FIG. 2.2 – Illustration du calcul d'intérêt par utilisation du focus de A et du nimbus de B

L'intérêt est une mesure qui permet de quantifier la motivation pour récupérer des informations concernant un certain objet. Les auteurs proposent de différencier les calculs d'intérêts en fonction du média utilisé (textuel, audio, vidéo...).

Dans le projet Continuum de *France Telecom R&D* [45], chaque participant gère des copies de son entité situées sur la machine des entités dont la zone de perception intersecte sa zone d'influence. Les participants possèdent donc un espace de simulation peuplé de représentations locales de ses voisins. Un système de communication multipoint permet aux replicas d'une entité d'être informés des transformations survenues sur celle-ci.

Le caractère le plus innovant de Massive-2 [49] est appelé "*Third Party Object*" [18] : certaines entités ont la capacité de modifier le calcul d'intérêt entre deux autres entités. Ainsi, un objet "fenêtre" peut atténuer le niveau d'intérêt sonore entre deux entités. C'est également à partir de cette plate-forme que le IP Multicast est utilisé : chaque objet est le seul actif dans son groupe de communication, les entités dont les auras chevauchent la sienne sont passifs dans ce groupe.

Massive-3 [84] est la plate-forme pour environnements virtuels partagés la plus aboutie. Elle permet, entre autres, de réaliser un découpage spatial hiérarchisé tout en conservant le chevauchement d'auras pour la mise en relation. Malheureusement, elle dévoile également les difficultés auxquelles restent confrontés ces systèmes.

Tout d'abord, ils se heurtent au faible déploiement de la communication multipoint, et de l'IP Multicast en particulier, sur Internet. Ce type de communication requiert la participation des routeurs, qui, en majorité, n'ont pas adopté les différents protocoles qui ont pu être proposés par la communauté scientifique. Notre volonté de permettre à n'importe quelle machine connectée à Internet de participer au monde virtuel nous a contraints à rejeter ce type d'architectures.

Ensuite, ils nécessitent toujours la présence d'un ou plusieurs serveurs légers pour le maintien des relations. Le découpage du monde nécessite une base de données associant chaque groupe multicast à une cellule tandis qu'un serveur est responsable de déterminer le chevauchement des auras. Au-delà des limitations physiques que ces serveurs entraînent, il est important de noter que leur présence ne permet pas de garantir la persistance du monde et la liberté des participants.

Filtrage d'informations par intérêts

Le récent engouement pour les jeux en réseau a relancé les recherches autour des mondes virtuels massivement partagés. Parmi les nombreuses offres commerciales, quelques systèmes [6, 39, 26, 12] n'utilisent pas de serveurs centraux. La diffusion des événements est assurée par un mécanisme de *publication/abonnement*.

Une entité participant au monde virtuel possède des centres d'intérêts. Elle souhaite être informée de tout événement relatif à cet intérêt. Un événement est publié par une entité lorsqu'elle décide de le rendre public auprès d'autres participants au monde virtuel. Chaque participant est amené à publier les événements qui le concernent (déplacements, actions...) et à notifier ses intérêts. Des mécanismes garantissent que tous les événements ayant trait à un intérêt identifié par une entité sont effectivement reçus par cette entité.

Par exemple, les auteurs proposent que les événements contiennent la position de l'entité émettrice tandis que les intérêts déterminent une zone du monde, typiquement la zone située autour de l'entité. Ainsi, une entité située à la position $(x = 100, y = 200)$ s'abonnerait à l'ensemble des événements contenant une position (x', y') telle que $50 \leq x' \leq 150$ et $150 \leq y' \leq 250$. La difficulté réside dans le déploiement du mécanisme assurant la transmission des messages en rapport avec l'intérêt annoncé.

Ce mécanisme repose sur la collaboration des entités participantes. Chaque entité est responsable d'une collection d'intérêts. Elle gère les abonnements et la diffusion des événements portant sur ces intérêts. Une entité notifiant son intérêt a doit informer au moins un responsable de l'intérêt a. Tous les événements sont également envoyés à ce

responsable qui est chargé de transmettre cet événement à toutes les entités abonnées.

La découverte d'une entité responsable d'un intérêt et la diffusion des événements sont facilitées par une organisation en anneau dans [6]. Les systèmes Gryphon [8, 12] et Siena [26] utilisent la technique de l'inondation pour réaliser ces tâches.

Dans de tels systèmes, il existe toujours un intermédiaire entre une entité désirant annoncer un événement et les entités intéressées par cet événement. Le passage par cet intermédiaire ralentit forcément la diffusion du message. De plus, les risques de défaillance de cet intermédiaire font peser une lourde menace sur la capacité d'une entité à diffuser un événement. Enfin, de nombreux messages sont nécessaires au maintien du système quand il est organisé tandis qu'il est très coûteux quand il n'est pas organisé.

Il apparaît en conclusion que la solution idéale n'a pas encore été dévoilée. Tout d'abord, la communication par groupes multipoints n'offre, pour l'instant, pas les garanties nécessaires à un large déploiement. Ensuite, l'utilisation de serveurs allégés pour mettre en relation les entités par calculs d'intérêt rend le système fragile, tout en contraignant la liberté individuelle des participants. Enfin, les architectures visant à établir cette mise en relation par le travail collaboratif des participants rencontrent encore de nombreuses difficultés.

Il apparaît surtout qu'il semble impératif d'établir des connexions directes entre les entités les plus proches dans le monde virtuel. Ce problème n'est pas nouveau, puisqu'il a fait l'étude de nombreuses études formelles, détaillées dans la suite.

2.1.2 Topologies basées sur les graphes de voisinage

La communication entre les participants d'un réseau se représente par un graphe, parfois appelé graphe de communication. Les nœuds ou sommets modélisent les participants, *i.e.* les machines disposant d'une capacité de communication. La mise en relation des nœuds proches dans un espace a fait l'objet de nombreuses études formelles, récemment reprises pour construire des topologies dans les réseaux ad-hoc.

Les graphes de voisinage

Un graphe G, noté $G(V, E)$ est défini à partir d'un ensemble de sommets V et d'un ensemble E d'éléments $V \times V$. Ces couples sont appelés arcs ou arêtes. Nous noterons (x, y) l'arc reliant le sommet x au sommet y. Selon que la communication est unidirectionnelle ou bidirectionnelle, le graphe est orienté ou non-orienté. Un graphe non-orienté est un graphe $G(V, E)$ tel que l'arc (x, y) est confondu avec l'arc (y, x). On dit qu'un nœud y est voisin de x si $(x, y) \in E$.

Un chemin de longueur q dans un graphe $G(V, E)$ est une séquence de q arcs de G consécutifs $ch = ((x, x_1), (x_1, x_2), \ldots, (x_q, y))$ où x est l'extrémité initiale et y l'extrémité finale du chemin ch. Le chemin le plus court est le chemin de longueur minimale parmi l'ensemble des chemins de x à y.

Un graphe $G(V, E)$ est connexe si, pour toutes les paires $x, y \in V$, il existe un chemin de x à y. Il est planaire lorsqu'il admet une représentation sur un plan ne présentant aucune intersection d'arcs.

Il est possible d'associer à chaque sommet $x \in V$ d'un graphe $G(V, E)$, une position $pos(x)$ dans l'espace R^d. On note $\delta(x, y)$ la distance euclidienne dans R^d entre les sommets x et y et $D(x, r)$ le disque centré en x de rayon r. Une lune entre deux sommets x et y est définie par $\Delta_{xy} = D(x, \delta(x, y)) \cap D(y, \delta(x, y))$.

Les graphes de voisinage [63] sont des constructions issues de la géométrie calculatoire [83] qui permettent de rendre compte de la proximité des sommets dans un espace R^d. Ce sont des structures connexes et planaires qui ont été introduites sous cette dénomination en 1980 [95]. La Figure 2.3 offre une illustration des principaux graphes de voisinage.

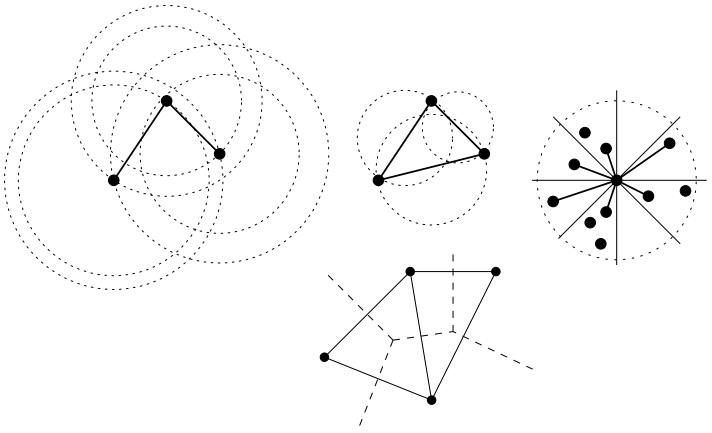

FIG. 2.3 – Exemple de graphes de voisinage : à gauche un graphe de voisinage relatif, au centre haut un graphe de Gabriel, au centre bas un diagramme de Voronoï et une triangulation de Delaunay et à droite un graphe de Yao

Le **graphe de voisinage relatif** [95] d'un graphe $G(V, E)$ vérifie que, pour tous les arcs $(x, y) \in E$, il n'existe pas de sommet $z \in V$ tel que z appartienne à la lune $\Delta_{x,y}$:

$$(x, y) \in E \Leftrightarrow \Delta_{x,y} \cap V = \emptyset$$

Le **graphe de Gabriel** [44] d'un graphe $G(V, E)$ consiste en tous les arcs $(x, y) \in E$ tel qu'il n'existe pas de nœud $z \in V$ localisé dans le disque de diamètre $\delta(x, y)$:

$$(x, y) \in E \Leftrightarrow \forall z \in V, z \notin D(\frac{x + y}{2}, \frac{\delta(x, y)}{2})$$

Construire un **Graphe de Yao** [108] nécessite de découper l'espace autour de chaque nœud en k secteurs d'angle $2\pi/k$. Dans ce graphe, chaque nœud est connecté avec son plus proche voisin dans chaque secteur.

On appelle région de Voronoï [100] d'un sommet $p \in V$ l'ensemble des points $(x, y) \in R^d$ les plus proches de p. Le diagramme de Voronoï est décrit par l'union des régions de Voronoï de tous les points. Une **Triangulation de Delaunay** [34] d'un ensemble de sommets V relie les paires de sommets dont les régions de Voronoï sont adjacentes.

La construction de ces graphes repose sur une connaissance de la position de tous les nœuds participants. De nombreux algorithmes proposés visent à réduire le coût de ces constructions, en particulier le temps de calcul ou le nombre de messages échangés.

Topologies pour les réseaux ad-hoc

Dans un réseau ad-hoc, un nœud x peut envoyer un message à un nœud y si la distance séparant x et y est inférieure à la portée de l'antenne de x. Généralement, les modélisations des réseaux ad-hoc partent de l'hypothèse que toutes les portées sont identiques. Par conséquent, le graphe qui modélise un réseau ad-hoc est constitué d'un ensemble de nœuds V et d'un ensemble d'arcs E tels que $(x, y) \in E$ si $\delta(x, y) < r$ avec r la portée des antennes. Ce type de graphe est communément appelé graphe de disque unitaire. Un message envoyé par un nœud $p \in V$ est reçu simultanément par tous les nœuds q tels que l'arc (p, q) existe dans le graphe de disque unitaire.

La mise en place de certaines fonctions de base du réseau (la communication entre deux nœuds distants, la diffusion d'un message à tous les participants, la communication de groupe...) a rapidement mis en lumière certaines difficultés. Les batteries et les mémoires sont des ressources rares, or chaque émission de message et chaque nouveau stockage d'informations consomment ces ressources. Afin de réduire cette consommation, de nombreux chercheurs ont proposé de construire une structure logique superposée à l'infrastructure du réseau ad-hoc [73].

Ainsi, pour réduire le nombre d'émissions lors de la diffusion d'un message à l'ensemble des participants, les auteurs de [90] proposent de construire une topologie basée sur un graphe de voisinage relatif. Il a été montré dans [21, 66] qu'une méthode efficace de routage pour établir une communication entre deux nœuds distants peut être obtenue en utilisant les liens de communication d'une topologie basée sur des graphes de Gabriel. Ou, encore, une topologie issue de la construction d'un graphe de Yao est susceptible d'offrir une substantielle économie d'énergie [74].

Dans ces propositions, chaque nœud détermine ses voisins grâce à des règles simples et à la connaissance des positions des nœuds les plus proches. Ces positions sont obtenues par l'envoi périodique de messages de signalisation et par la nature hertzienne des communications permettant à deux machines de communiquer même si elles ne se connaissent pas. Les caractéristiques de communication sur Internet ne permettent pas cette découverte spontanée et il est, justement, nécessaire de mettre au point des mécanismes permettant de mettre en relation les entités les plus proches.

2.2 Définitions et propriétés du système

Les difficultés rencontrées par les architectures existantes nous ont poussés à envisager une nouvelle voie pour aborder le problème. Dans un premier temps, nous avons cherché à définir formellement les éléments à considérer dans un environnement virtuel partagé, puis nous avons tenté d'identifier les propriétés que notre système devait garantir. Dans la suite, nous décrivons ces deux étapes.

2.2.1 Définitions

Quand il s'agit de définir les éléments d'un environnement virtuel partagé, il est naturel de prendre le monde réel pour exemple. De fait, nous avons distingué le monde, les entités du monde et les relations entre ces entités.

Le monde virtuel Dans le monde réel, nous évoluons sur la surface d'une sphère. Cette surface est à la fois finie - elle possède un nombre fini de coordonnées - et sans limite - il est possible de se déplacer dans une même direction infiniment longtemps.

Dans le monde virtuel de Solipsis, les entités évoluent sur la surface d'un *tore*. Une surface sur un tore possède les mêmes caractéristiques qu'une surface sur une sphère, *i.e.* un ensemble de coordonnées fini et une surface sans limite. Il nous a semblé que les calculs de position et de distance étaient plus simples et moins coûteux sur la surface d'un tore que sur la surface d'une sphère. Les lecteurs soucieux de comprendre les différences entre ces deux surfaces sont invités à consulter l'ouvrage de Grima et Márquez [50].

Le monde virtuel de Solipsis est un tore $T = \{(x, y) \in \mathbb{N}_{size_x} \times \mathbb{N}_{size_x}\}$ dans lequel $size_x$ et $size_y$ délimitent la taille du monde de Solipsis et la notation $\mathbb{N}_k$ représente l'ensemble des entiers positifs modulo k. Nous avons volontairement choisi $size_x$ et $size_y$ très grands afin de disposer d'un vaste monde pouvant accueillir un grand nombre de participants. Ainsi, les coordonnées x et y sont représentées par 128 bits, ce qui produit environ 10^{75} positions différentes.

Un point e est déterminé dans le tore par une infinité de positions $\{(x_e+(m*size_x), y_e+(n*size_y)); m, n \in \mathbb{Z}\}$. De fait, il existe une infinité de lignes géodésique joignant deux points dans un tore. Pour simplifier, il est courant de noter sa position $(x_e, y_e) \in T$ et, dans la suite, la plus courte ligne géodésique entre deux points e et e' sera notée $[e, e']$ et la longueur de cette ligne géodésique sera $d(e, e')$.

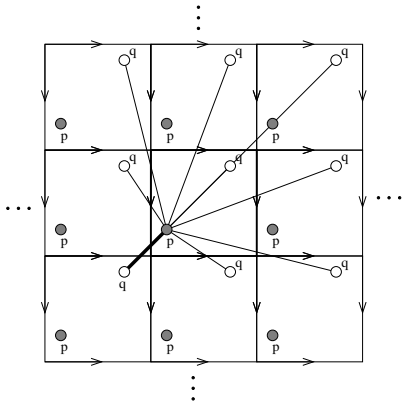

FIG. 2.4 – Un tore plat, les différentes copies de deux points p et q et quelques lignes géodésiques entre ces points

Il existe une représentation simple du tore sur un plan. Un rectangle peut être obtenu en "déroulant" le tore. Le *tore plat* est construit par les translations horizontales et verticales de ce rectangle. La figure 2.4 illustre un tore plat représentant un tore.

Une entité Dans le monde réel, chaque objet est unique - il n'existe qu'un seul exemplaire de cet objet - et dispose d'une position dans le monde - une position sur la surface de la sphère. Tous les objets sont mobiles, *i.e.* leur position varie avec le temps.

Dans Solipsis, une entité est une instance d'un programme exécutée par un ordinateur connecté au réseau Internet. Cette entité dispose d'un identifiant unique.

Une entité e est également caractérisée par une position $(x(e), y(e)) \in T$ dans le monde

virtuel. Les entités étant mobiles, la position d'une entité e à l'instant t peut être différente de sa position à l'instant $t', t' \neq t$.

Des relations Dans le monde réel, une entité perçoit les entités situées dans son voisinage immédiat. Dans Solipsis, nous attribuons à chaque entité un *environnement virtuel local*. L'environnement virtuel local de l'entité e située à la position $pos(e)$ est le disque $A(e)$ de rayon $r(e)$ centré en $pos(e)$. Ce disque représente la zone du monde virtuel située à proximité de l'entité e.

Le rayon $r(e)$ est appelé le *rayon de connaissance*. Dans le monde réel, la capacité de perception d'un homme peut atteindre plusieurs centaines de mètres dans un endroit désert, tandis qu'elle se réduit quand la foule se densifie. De même, le rayon de connaissance d'une entité dans Solipsis peut varier avec les capacités physiques de l'entité et la densité des entités aux alentours.

La perception dans le monde réel signifie qu'il y a un échange d'informations permettant à une entité de situer et se représenter un autre objet. La perception dans le monde virtuel de Solipsis implique un canal de communication entre deux entités. Les informations circulent via une *connexion* dans le réseau. Une connexion peut être initiée dès qu'une des deux entités parvient à envoyer un message à l'autre entité. Lors de l'initialisation d'une connexion, les entités e et e' s'échangent leurs caractéristiques. Cet échange leur offre une connaissance mutuelle de leur position et des dimensions de leur environnement virtuel local. Par la suite, et à moins d'une fermeture de la connexion, les entités e et e' s'échangent régulièrement des informations.

Deux entités partageant une même connexion sont connectées. On appelle *voisin* d'une entité e toute entité e' connectée à e. L'ensemble des voisins d'une entité e est noté $k(e)$.

A partir de ces définitions, il est possible de définir le graphe de communication du monde virtuel. Soit $V(t)$ l'ensemble des entités participant au monde virtuel à l'instant t. Soit $E(t)$ l'ensemble des connexions existantes à l'instant t entre les entités de $V(t)$. Le graphe du monde virtuel $G(t)$ est le graphe $(V(t), E(t))$.

2.2.2 Propriétés

Nous cherchons maintenant à définir les propriétés globales que doit posséder le graphe $G(t)$ pour être conforme à nos objectifs. Nous avons déterminé deux propriétés globales : la connaissance locale et la connexité.

Connaissance locale

Chaque participant au monde virtuel doit être réellement *présent* dans le monde virtuel. La présence d'une entité e à une position dans le monde virtuel est assurée par le fait que (1) les entités situées à proximité de e connaissent e et (2) e connaît les entités situées à proximité de lui.

Soit $\mathcal{A}(e)$ l'ensemble des entités situées dans l'environnement virtuel local de l'entité $e \in V$. Formellement, $\mathcal{A}(e) = \{p \in V : pos(p) \in A(e)\}$.

Le graphe de communication doit, dans la mesure du possible, faire en sorte que, pour toutes les entités $e \in V$, toutes les entités dans $\mathcal{A}(e)$ soient des voisins de e. Nous appelons

cette propriété la *Connaissance Locale*.

Définition 2.2.1 (Connaissance locale) *Le graphe de communication $G(V, E)$ respecte la propriété de connaissance locale, lorsque, $\forall e \in V, \mathcal{A}(e) \subseteq k(e)$.*

Il est important de noter qu'une entité e a la liberté de connaître des entités en dehors de son environnement virtuel local. En revanche, cette propriété impose que e connaisse *toutes* les entités situées à l'intérieur.

Connexité

Le graphe du monde virtuel doit être unique. Si le monde se partitionne en deux sous-mondes, il est probable que les mouvements des entités de ces deux mondes produisent des incohérences virtuelles. Il est donc primordial que le graphe $G(t)$ soit connexe.

Définition 2.2.2 (Connexité) *Le graphe $G(V, E)$ est connexe lorsque, pour toute paire d'entités dans V il existe entre elles un chemin dans G.*

2.3 Comportement local

Le graphe de communication du monde virtuel doit, dans la mesure du possible, respecter les deux propriétés énoncées précédemment. Dans Solipsis, il n'existe aucune autorité centrale. Par conséquent, ces propriétés globales doivent émerger du comportement de chaque entité. Or, les entités ne disposent que d'une vue locale du système. Elles connaissent leur position et les caractéristiques de leurs voisins.

Dans la suite, nous avons cherché à définir deux mécanismes collaboratifs qui, appliquées par toutes les entités, doivent aboutir au respect de ces propriétés globales.

2.3.1 Collaboration spontanée

Le but du premier mécanisme que nous avons imaginé est le respect de la propriété de connaissance locale dans un environnement mobile. A n'importe quel instant, une entité se déplaçant peut pénétrer les environnements virtuels locaux d'autres entités. Or, il est nécessaire qu'une entité soit connectée avec toutes les entités situées dans son environnement virtuel local. Il faut donc doter les entités d'un moyen d'initier de nouvelles connexions avec ces nouveaux voisins.

Un mécanisme de requêtes - réponses est envisageable. Chaque entité pourrait demander, à intervalle régulier, à ses voisins s'ils n'ont pas détecté une nouvelle entité dans son environnement virtuel local. Ce mécanisme souffre de défauts majeurs. Tout d'abord, il produirait de nombreuses requêtes peu pertinentes, particulièrement quand les entités bougent peu. Ensuite, il y aurait toujours un temps de décalage entre l'événement et la détection de celui-ci. Enfin, il est difficile de prévoir la fréquence d'émission de ces requêtes. En choisissant une fréquence trop élevée, le risque de requêtes non pertinentes augmente, tandis que le décalage temporel augmente avec une fréquence plus faible.

Nous avons donc opté pour un mécanisme basé sur la collaboration spontanée des entités. Le principe est le suivant. Une entité a alerte une entité b dès que a détecte qu'une

entité c est entrée dans l'environnement virtuel local de b. Cet événement peut survenir lorsque c se déplace, mais aussi lorsque le rayon de connaissance de b est modifié.

Par ailleurs, une entité est toujours intéressée par une entité qui se dirige vers elle, même lorsqu'elle n'est pas encore entrée dans son environnement virtuel local. En conséquence, une entité a alerte une entité b lorsque a détecte une entité c qui, se dirigeant vers b, est maintenant plus près de b que a ne l'est.

Il est intéressant de constater que a n'agit pas pour son intérêt mais plutôt pour l'intérêt de b et de c. D'un autre côté, a peut espérer que b et c agissent de la même manière. Ainsi, le système se base effectivement sur la coopération entre les participants.

L'Algorithme 1 décrit le travail que chaque entité doit accomplir à la réception d'un message relatant un événement. Pour plus de lisibilité, nous considérons que cet algorithme est exécuté par une entité e, à la réception d'un message émis par une entité a.

La règle $\mathcal{R}_1$ décrit le traitement lorsque l'événement concerne un déplacement. L'ancienne position de a est alors notée $\varrho(a)$ tandis que sa nouvelle position est $p(a)$. La variable booléenne b_1 est vraie lorsque a est rentrée dans l'environnement virtuel local de l'entité i, tandis que b_2 est vraie lorsque le déplacement de a a fait entrer i dans l'environnement virtuel local de a. Enfin, la variable booléenne b_3 est vraie quand a est maintenant plus près de i que e ne l'est alors que b_4 signifie que i est maintenant plus près de a que e ne l'est.

La règle $\mathcal{R}_2$ présente la tâche accomplie lorsque l'événement notifie une modification du rayon de connaissance de l'entité a. L'ancien rayon de connaissance de a est noté $\tau(a)$ et son nouveau rayon $r(a)$.

Ce mécanisme collaboratif doit permettre à chaque entité d'être connectée avec ses plus proches voisins, qu'ils appartiennent à son environnement virtuel local ou pas. De plus, une entité qui se déplace dans une direction connaît à l'avance les entités dans cette direction. En conséquence, cette règle contribue fortement au respect de la propriété de connaissance locale.

2.3.2 Collaboration par requêtes récursives

Dans le monde Solipsis, une entité ne peut compter que sur ses voisins et sur elle-même. Or, si elle ne connaît personne dans un large secteur autour d'elle, il lui sera difficile d'être informée de l'arrivée de nouvelles entités de ce secteur. A l'inverse, si elle se déplace dans ce secteur, elle ne pourra pas être détectée par les entités s'y trouvant.

Nous avons donc proposé une autre règle pour éviter qu'une entité ne "tourne le dos" à une partie du monde. Il s'agit de faire en sorte qu'une entité connaisse au moins un voisin dans chaque direction autour d'elle. Nous nous sommes basés sur la notion d'*enveloppe convexe*, issue de la géométrie calculatoire [83, 50].

L'enveloppe convexe d'un ensemble de points (ici entités) S est le plus petit polygone convexe contenant tous les points de S. Nous notons $CH(V')$ l'ensemble des positions englobées dans l'enveloppe convexe de l'ensemble d'entités $V' \subseteq V$. La règle illustrée dans la figure 2.5 s'énonce comme suit.

Règle 2.3.1 *$\forall e \in V$, e doit se situer à l'intérieur de $CH(k(e))$.*

Algorithme 1 Réception d'un message de déplacement d'une entité a pour une entité e

Données disponibles :

$k(e)$: voisins de e ;
$r(i)$: rayon de connaissance de l'entité $i \in k(e)$;
$p(i)$: position de l'entité $i \in k(e)$;

Fonctions disponibles :

$dist(p, p')$: calcule la position entre deux positions p et p' ;
$alert(i, j)$: envoie un message d'alerte à $i \in k(e)$ concernant $j \in k(e)$;

Actions :

$\mathcal{R}_1$: déplacement de l'entité a

pour tout $i \in k(e) \setminus \{a\}$:
$b_1 = dist(p(a), p(i)) \leq r(i) < dist(\varrho(a), p(i))$
$b_2 = dist(p(a), p(i)) \leq r(a) < dist(\varrho(a), p(i))$
$b_3 = dist(p(a), p(i) \leq dist(p(e), p(i)) < dist(\varrho(a), p(i))$
$b_4 = dist(p(a), p(i) \leq dist(p(e), p(a)) < dist(\varrho(a), p(i))$
si $b_1 \vee b_3$:
$alert(i, a)$
si $b_2 \vee b_4$:
$alert(a, i)$

$\mathcal{R}_2$: modification du rayon de connaissance de l'entité a

pour tout $i \in k(e) \setminus \{a\}$:
si $\tau(a) \leq dist(p(a), p(i)) < r(a)$:
$alert(a, i)$

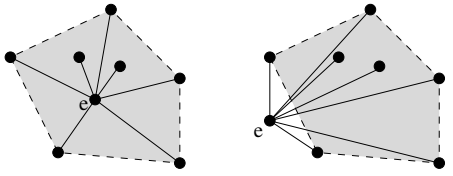

FIG. 2.5 – Les zones grisées représentent l'enveloppe convexe des voisins de e. A gauche, e respecte la règle 2.3.1 alors qu'il ne la respecte pas à droite.

Une entité e peut facilement vérifier qu'elle respecte la règle. Pour cela, nous introduisons quelques notations. L'angle dirigé $\angle(e'\ e\ e'')$ est défini par l'angle dans le sens trigonométrique formé à e par $[e, e']$ et $[e, e'']$. Une entité e_i appartient au secteur $\nabla(e'\ e\ e'')$ quand $\angle(e'\ e\ e'') = \angle(e'\ e\ e_i) + \angle(e_i\ e\ e'')$. Le successeur d'une entité e' pour l'entité e_0 est défini par :

$$s_{e_0}e' = e'' \Leftrightarrow \forall e \in k(e_0) : e \notin \nabla(e'\ e_0\ e'')$$

En généralisant, le $p^{ième}$ successeur de e' pour e est noté $s_e^p e'$, le prédécesseur de e' est $s_e^{-1}e'$ et, si e connaît n entités, $s_e^n e' = e'$.

Une entité e respecte la règle 2.3.1 quand elle vérifie :

$$\forall e' \in k(e) : s_e e' = e'' \Rightarrow \angle(e'\ e\ e'') < \pi$$

Les caractéristiques de Solipsis peuvent conduire à des situations dans lesquelles une entité e respecte la règle 2.3.1 à l'instant t et ne la respecte plus à $t + \delta$. Par exemple

lorsqu'un voisin se déplace ou disparaît. Il faut donc prévoir un mécanisme permettant de recouvrer la règle en détectant un nouveau voisin. L'Algorithme 2 présente le mécanisme que nous proposons.

Quand une entité e détecte deux entités consécutives e_1 et e_f avec $\angle(e_1\ e\ e_f) \geq \pi$, elle se lance immédiatement à la recherche d'une ou plusieurs entités dans le secteur $\nabla(e_1\ e\ e_f)$. Cette situation est illustrée dans la figure 2.6. Si e_1 respecte la règle 2.3.1, elle est connectée avec une entité e_2 dans le demi-plan délimité par la droite Δ_1. Cette entité e_2 vérifie $\angle(e_2\ e\ e_f) < \angle(e_1\ e\ e_f)$.

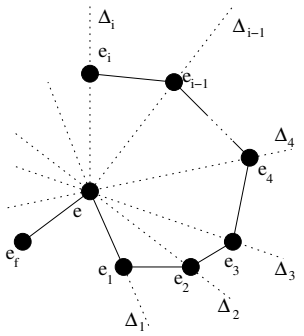

FIG. 2.6 – Collaboration récursive pour le respect de la règle 2.3.1

Algorithme 2 Récupération de la règle 2.3.1 (entité e)

Données disponibles :

$k(e)$: voisins de e ;

Fonctions disponibles :

respect() : retourne vrai si e respecte la règle 2.3.1 ;
message(i) : envoie une requête à l'entité i et retourne l'entité j issue de la réponse de i ;
mauvais() : retourne l'entité i telle que $\angle(i\ e\ s_e i) \geq \pi$;

Actions :

tant que non *respect*()
$i \leftarrow$ *mauvais*()
$j \leftarrow$ *message*(i)
$k(e) = k(e) \cup \{j\}$

Si $\angle(e_2\ e\ e_f) < \pi$, l'entité e respecte de nouveau la règle 2.3.1. Mais, si l'entité e_2 n'est pas suffisante, e peut réitérer sa demande d'entités dans le secteur $\nabla(e_2\ e\ e_f)$. De la même manière, si e_2 respecte la règle, elle connaît une entité e_3 dans le demi-plan délimité par la droite Δ_2.

Récursivement, e finit par découvrir une entité e_i telle que $\angle(e_i\ e\ e_f) < \angle(e_{i-1}\ e\ e_f)$. Cette recherche récursive se termine dès que $\angle(e_i\ e\ e_f) < \pi$.

2.4 Propriétés topologiques

Nous avons précédemment présenté deux propriétés globales - la connaissance locale et la connexité - et deux algorithmes visant au respect de ces propriétés. Il nous reste maintenant à montrer que ces algorithmes permettent effectivement de construire un graphe de communication respectant les propriétés globales.

2.4.1 Connaissance locale

L'exécution des deux algorithmes présentés permet de garantir la connaissance locale dans la majorité des cas. Il existe néanmoins des situations pour lesquelles deux entités ne sont pas connectées alors que leur distance est inférieure au rayon de connaissance de l'une d'elles. Par exemple, la figure 2.7 représente une entité e_f qui n'est pas connectée avec une entité e alors que e_f se situe dans l'environnement virtuel local de e.

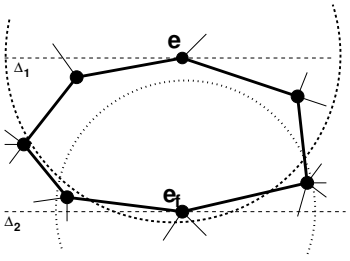

FIG. 2.7 – Les entités e et e_f ne sont pas connectées alors que e_f se situe dans l'environnement virtuel local de e

Le respect de la règle 2.3.1 impose que e et e_f connaissent au moins une entité, respectivement e_1 et e_i, dans les demi-plans respectivement délimités par les droites Δ_1 et Δ_2. Il est évident que $e_1 \neq e_i$. En effet, l'entité e_1 serait alors voisine à la fois de e et de e_f. Dans ce cas, cette entité, détectant la présence de e_f dans l'environnement virtuel local de e, aurait alerté e suivant l'Algorithme 1 et e aurait été connectée à e_f.

De fait, le chemin entre e et e_f contient au moins deux entités intermédiaires. Le chemin entre e et e_f est noté $(e, e_f) = ((e, e_1), \ldots, (e_{i-1}, e_i), (e_i, e_f))$. Les entités e_1 et e_i sont censées informer e et e_f dès qu'elles pénètrent l'environnement virtuel local d'une autre entité. Or, ni e_1 ni e_i n'agit en ce sens, alors que toutes les entités $e_j, \forall j \leq i$ respectent les règles énoncées.

Les conditions d'existence de telles situations sont liées au fait qu'il n'existe pas de plus court chemin que (e, e_f). Ainsi, il est sûr que e_f n'est pas en relation avec e_{i-1}. Cela signifie que $e_{i-1} \notin \mathcal{A}(e_f)$, que $e_f \notin \mathcal{A}(e_{i-1})$ et que $d(e_f, e_{i-1}) > d(e_i, e_{i-1})$.

De la même manière, l'entité e et l'entité e_2 ne sont pas connectées. Forcément, on obtient $e_2 \notin \mathcal{A}(e)$, puis $e \notin \mathcal{A}(e_2)$, et enfin $d(e_1, e_2) < d(e, e_2)$.

Enfin, ni l'entité e, ni l'entité e_f ne soit voisins d'une entité sur le chemin. Formellement, $\forall j \in [2 \ldots (i-1)], e \notin k(e_j)$ et $e_f \notin k(e_j)$.

Cette situation *peut* survenir mais ses conditions d'existence semblent peu probables. Il semble possible de contrevenir à cette situation en mettant au point des mécanismes coûteux basés sur une inondation du réseau. Nous avons préféré laisser le système in-

changé pour deux raisons. Tout d'abord, durant la phase d'élaboration de Solipsis, nous avons toujours privilégié le passage à l'échelle à la cohérence. De fait, nous n'avons pas souhaité ajouter à notre système des mécanismes risquant de limiter le passage d'échelle. D'autant plus que les expériences nous ont montré que cette situation survient très rarement. Ensuite, il est intéressant de noter que si e, par exemple, décide de se diriger vers e_f, elle finira par ne plus respecter la règle 2.3.1. En appliquant l'algorithme permettant de recouvrer la règle, l'entité e_1 l'informera sans doute de la présence de e_2. Récursivement, si e continue de se déplacer dans la direction de e_f, elle découvrira, peu à peu, toutes les entités sur le chemin (e, e_f) jusqu'à rencontrer e_f. En conséquence, les déplacements des entités tendent à éviter cette situation et contribuent donc au respect de la propriété de connaissance locale.

2.4.2 Connexité

Le principal effet de la règle 2.3.1 sur la connexité concerne l'absence d'*île* dans le graphe. On dit qu'un ensemble d'entités $S \subset V$ forme une île lorsqu'il existe un chemin entre toutes les paires d'entités de S, mais qu'il n'existe aucun chemin entre une entité de S et une entité de $V \setminus S$. En théorie des graphes, une île est appelée un composant connexe de V.

La simple collaboration spontanée entre voisins n'est pas suffisante pour éviter la création d'îles. En effet, lorsqu'un ensemble d'entités très proches les unes des autres est isolé du reste du monde, elles peuvent avoir tendance à se satisfaire des connexions avec leurs plus proches voisins. La figure 2.8 est un exemple de cette situation.

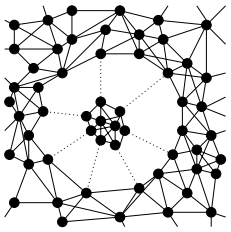

FIG. 2.8 – Un composant connexe isolé et les connexions qui évitent la création d'une île

Soit e un des entités formant l'enveloppe convexe d'un composant connexe S. Par définition de l'enveloppe convexe, les connexions avec les entités situées dans S ne lui permettent pas de respecter la règle 2.3.1. Si e respecte la règle, il est impératif qu'elle soit connectée avec une entité e' située dans $V \setminus S$. De fait, l'entité e devient un intermédiaire entre les entités dans S et une entité dans $V \setminus S$. Il existe alors un chemin entre chaque entité de S et une entité dans $V \setminus S$. Donc, si l'enveloppe convexe d'un ensemble connexe d'entités S_1 est englobée dans l'enveloppe convexe d'un autre ensemble connexe d'entités S_2 alors S_1 et S_2 forment un unique composant connexe $S = S_1 \cup S_2$.

On appelle *méridien* du tore la ligne droite définie par $\{(x, y_0); x \in \mathbb{R}\}$ et *parallèle* du tore la ligne définie par $\{(x_0, y); y \in \mathbb{R}\}$. On dit que les entités sur le tore sont dans une *position cylindrique* lorsque l'ensemble de leurs positions et l'ensemble des lignes géodésiques les joignant sont contenus entre deux méridiens ou deux parallèles. Un lemme bien connu

en géométrie calculatoire indique que l'enveloppe convexe d'un ensemble de points dans un tore est le tore entier si les points ne sont pas dans une position cylindrique [50].

Soit S_1 et S_2 deux sous-ensembles connexes de V. Si les entités de S_1 ne sont pas dans une position cylindrique, l'enveloppe convexe de S_1 englobe le tore entier. Dans ce cas, l'enveloppe convexe de S_2 est englobée dans l'enveloppe convexe de S_1 et, comme précédemment, S_1 et S_2 forment alors un unique composant connexe $S_1 \cup S_2$.

Admettons maintenant que les entités de S_1 et les entités de S_2 se trouvent dans une position cylindrique. Soit Δ_1 et Δ'_1 les deux méridiens de S_1 et Δ_2 et Δ'_2 les deux méridiens de S_2. Admettons que la coordonnée y de Δ_1 est inférieure à celle de Δ'_1 et, de la même manière, la coordonnée y de Δ_2 est inférieure à celle de Δ'_2. Si les entités se situant sur le méridien Δ_1 ne peuvent pas respecter la règle, cela signifie que la ligne géodésique joignant Δ_1 et $\Delta'2$ n'est pas contenue dans $[\Delta_1, \Delta'_2]$, mais plutôt dans $[\Delta_1, \Delta'_1] \cup [\Delta'_1, \Delta_2] \cup [\Delta_2, \Delta'_2]$. Dans ce cas, une entité $e \in S_1$ sur le méridien Δ'_1 peut respecter la règle car il existe une ligne géodésique joignant e_1 et une entité $e_2 \in S_2$ située sur le méridien Δ_2. Donc, e_1 et e_2 sont connectés. En conséquence, il existe un chemin entre S_1 et S_2 et les deux ensembles S_1 et S_2 forment un unique composant connexe $S = S_1 \cup S_2$.

2.5 Connexion au monde

Nous avons montré précédemment qu'un comportement global satisfaisant résulte de l'exécution des deux algorithmes distribués. Il nous reste maintenant à décrire le mécanisme permettant à une entité qui se connecte au système de respecter la règle 2.3.1 et d'obtenir la connaissance locale nécessaire à la présence dans le monde virtuel.

Les réseaux de pairs partent toujours de l'hypothèse qu'une entité désirant intégrer le système connaît une entité déjà connectée. A l'initialisation, la nouvelle entité choisit une position dans le monde virtuel. C'est à cette position que cette entité désire être présente dans Solipsis. Nous avons donc besoin d'un algorithme permettant à une entité de connaître ses plus proches voisins et de respecter la règle 2.3.1 alors qu'elle ne connaît qu'une seule entité ainsi que sa position. En quelques sortes, cet algorithme peut s'apparenter à de la *localisation inverse* [67] dans le sens qu'il prend en argument une position et qu'il répond par les entités les plus proches de cette position. Sur beaucoup de points, cet algorithme est similaire à certaines stratégies de routage dans les réseaux ad-hoc [93, 14, 66]. La Figure 2.9 offre une illustration.

Soit e la nouvelle entité, (x_e, y_e) sa position et e_0 l'entité connue. Pour commencer l'algorithme, un message contenant l'identifiant de e et sa position est envoyé par e à l'entité e_0. Ce message est transmis par e_0 vers une entité e_1 qui est l'entité la plus proche de (x_e, y_e) parmi les entités $k(e_0) \cup \{e_0\}$. Récursivement, ce message atteint une entité e_n qui ne connaît pas de voisin plus proche de la position de e qu'elle-même.

Comme e_n est considéré comme le plus proche voisin de e, il ouvre immédiatement une connexion avec e. Malheureusement, e_n n'est pas forcément l'entité la plus proche mais la seconde étape va le révéler.

L'algorithme rentre alors dans une seconde phase dont le but est de collecter les entités tout autour de la position. L'entité e_n initie ce processus récursif en envoyant un message à l'entité e_m, l'entité la plus proche de la position parmi ses voisins. On peut noter que le

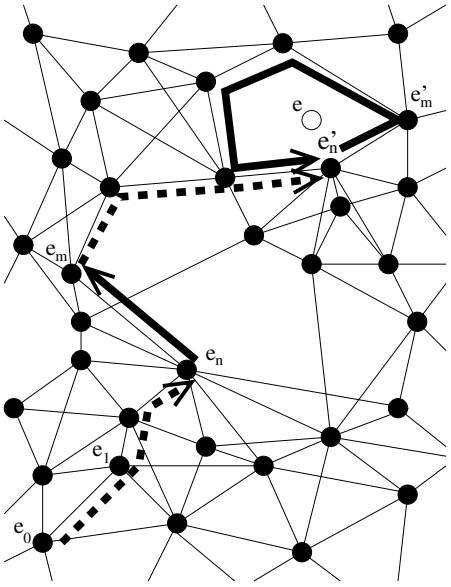

FIG. 2.9 – Algorithme de connexion d'une entité e au monde Solipsis

sens de rotation autour de la cible dépend de la position de e_m. Si l'entité e_m est à droite de la droite (e_n, e), le message va circuler dans le sens trigonométrique et inversement.

Immédiatement, e_m ouvre une connexion avec e. Toutes les entités sur le chemin suivi par ce message génèrent le polygone convexe requis pour le respect de la règle 2.3.1. Ensuite, elle détermine l'entité e_{m+1}, l'entité la plus proche de la position de e parmi $k(e_m) \setminus \{e_n\}$. Puis, elle compare la distance $d(e, e_{m+1})$ avec $d(e, e_n)$. Si l'entité e_{m+1} est plus proche de e que e_n, il est certain que e_n n'est pas l'entité la plus proche de e. Du coup, l'algorithme revient à la première phase en remplaçant l'entité e_0 par l'entité e_{m+1}.

S'il s'avère que e_n est l'entité la plus proche de e, l'entité e_m transmet le message à l'entité la plus proche de e dans le demi-plan délimité par la ligne (e, e_m) et tournant dans le sens initié par e_n. Cette opération se répète récursivement jusqu'à ce que le message traverse une nouvelle fois la ligne (e, e_n).

Des entités tout autour de e ont ouvert des connexions avec e. Ces entités sont les plus proches de e. Elle peut alors considérer qu'elle est connectée car elle respecte la règle 2.3.1 et, en ajustant correctement son rayon de connaissance, elle peut respecter également la propriété de connaissance globale.

2.6 Conclusion

Dans ce chapitre, nous avons présenté les principes fondamentaux de Solipsis. Nous avons tenté de montrer les raisons qui nous ont poussés à choisir une architecture basée sur un réseau de pairs. Puis, nous avons présenté les éléments du système et les propriétés globales visées.

Ensuite, nous avons présenté deux algorithmes simples que chaque utilisateur doit exécuter. Ces algorithmes permettent d'approcher un comportement global satisfaisant. Nous avons montré que ces algorithmes sont adaptés à un contexte de mobilité des entités.

Enfin, nous avons présenté un algorithme permettant à une nouvelle entité d'entrer dans le monde virtuel.

Dans la liste des travaux futurs figure en bonne place le problème de la détection de collisions et des mécanismes pour les éviter. Il s'agit d'éviter que deux entités se déplacent simultanément vers la même position, ou, si cela s'avère impossible, à mettre en place des algorithmes forçant les entités à se déplacer de nouveau pour revenir en un état cohérent virtuellement. Afin d'éviter les abus d'entités malicieuses, ce mécanisme pourrait reposer sur une action des observateurs de la scène.

Un autre sujet d'étude concerne l'algorithme de connexion. Pour l'instant, la découverte de l'entité la plus proche repose sur un grand nombre de petits sauts de proche en proche. Ce mécanisme n'est pas le plus efficace. Ainsi, si on considère n entités dans le monde, il se pourrait que $\sqrt{n}$ messages soient nécessaires pour atteindre l'entité la plus proche. Il serait possible de forcer les entités à conserver quelques connexions avec des entités distantes pour réduire le nombre d'entités intermédiaires. Il est sûr que des résultats spectaculaires peuvent être obtenus en choisissant une connexion au hasard dans le monde virtuel [71]. Mais, les nombreux travaux réalisés autour des tables de hachage distribuées [92, 65, 40] offrent une piste encore plus prometteuse vers une organisation optimale des liens distants.

Chapitre 3

Fonctionnement général

Ce chapitre détaille le fonctionnement d'une entité connectée au monde virtuel de Solipsis. Un lecteur désirant obtenir davantage d'informations est invité à analyser directement le programme accessible sur le site Internet du projet [55].

Nous considérons dans la suite un ordinateur connecté à Internet. L'usage du mot Internet englobe une collection de protocoles permettant à des ordinateurs d'échanger des messages par le biais d'un réseau. Parmi ces protocoles, IP [62], UDP [97] et TCP [94] assurent l'essentiel du transfert des informations. Nous considérons donc qu'un ordinateur connecté est capable de récupérer les informations contenues dans les messages qui lui sont destinés et, inversement, qu'il est capable d'envoyer des messages à un autre ordinateur connecté.

Une entité est un programme exécuté par un ordinateur connecté. Ce programme se décompose en plusieurs sous-programmes assumant des fonctionnalités différentes. Il existe deux grandes familles de sous-programmes.

Nœud Chaque entité possède un et un seul module appelé *nœud*. Ce sous-programme a la charge de garantir la présence de l'entité dans le monde virtuel et de coopérer avec les autres nœuds pour maintenir la topologie du réseau de pairs. Il agit conformément aux principes fondamentaux du système Solipsis.

Ce module est obligatoire. Il fournit un ensemble d'algorithmes permettant au monde virtuel d'être partagé grâce à l'échange de messages avec les autres nœuds. Par ailleurs, ce module est responsable des principales caractéristiques de l'entité. Il possède également des informations concernant les caractéristiques de ses voisins. Il s'agit donc du module central de l'entité.

La Section 3.1 détaille le fonctionnement du nœud. Ces modules communiquent entre eux selon un protocole décrit dans la Section 3.2.

Navigateur La communication entre utilisateurs dans le monde virtuel est la *raison d'être* du système Solipsis. Le module nœud permet à une entité de rencontrer d'autres entités, mais nous avons besoin de modules supplémentaires pour la communication et les interactions entre les entités. Nous appelons ces modules des *navigateurs*.

Il peut exister autant de navigateurs différents qu'il existe de moyens de jouir d'une expérience virtuelle. Dans cette thèse, nous nous sommes plus spécifiquement intéressés à concevoir une architecture de communication entre les divers éléments d'une entité et

non aux différentes possibilités multimédias de s'immerger dans un monde virtuel. C'est pourquoi nous nous sommes essentiellement concentrés sur l'organisation de ces modules et sur la communication entre un navigateur et un nœud.

Un navigateur fournit à l'utilisateur (1) une illustration du monde virtuel, (2) une interface entre l'homme et l'entité et (3) le moyen de communiquer avec d'autres navigateurs. Un navigateur permet de flâner virtuellement et de se déplacer d'un endroit virtuel à un autre. Il dépend de l'existence d'un nœud avec lequel il communique grâce à une interface de commande décrit dans la Section 3.3. Cette interface permet au nœud d'informer le navigateur des événements virtuels et au navigateur de contrôler le nœud.

Nous appelons un *service*, le sous-programme fournissant au navigateur le moyen de communiquer selon un média particulier. Un service est une partie indépendante du module navigateur, capable de communiquer directement avec un service similaire selon un protocole personnel. Par exemple, on peut imaginer des services tels que la visioconférence, l'échange de fichiers ou la communication textuelle...

Un service utilise les informations possédées par le navigateur. Il est optionnel et il n'y a aucune garantie sur son fonctionnement durant la vie de l'entité.

La figure 3.1 illustre l'ensemble des relations qui peuvent exister entre les différents éléments de Solipsis.

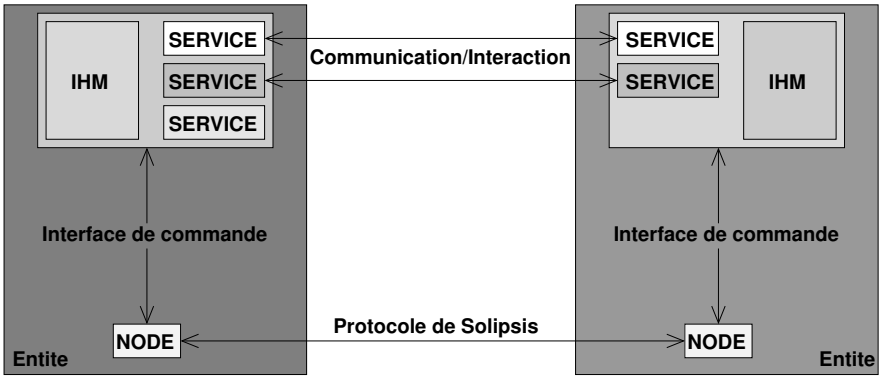

FIG. 3.1 – Vue générale de deux entités en communication dans Solipsis

3.1 Le nœud

Dans la suite, nous décrivons le module central de Solipsis : le nœud. Ses tâches sont dans la lignée des principes généraux décrits dans le chapitre 2.

3.1.1 Caractéristiques

Le nœud contient l'ensemble des caractéristiques de l'entité qu'il décrit. Certaines caractéristiques sont constantes tandis que d'autres sont amenées à évoluer dans le temps.

Les caractéristiques constantes de l'entité sont celles qui la représentent tout au long de son existence virtuelle. Ces caractéristiques, définies lors de la création de l'entité, ne peuvent pas être modifiées.

Identifiant Chaque entité est identifiée par une chaîne de caractères unique et constante id. Cette chaîne peut être générée par l'adresse Internet du lien de communication du nœud $(@IP, @PORT)$:

$$id = @IP : @PORT$$

Pseudo Dans la réalité, un humain est connu de manière usuelle par son nom et non par son identifiant. Dans Solipsis, chaque entité choisit une chaîne de caractères qui est reconnue comme un pseudo. L'unicité du pseudo n'est pas garantie puisque chaque entité peut choisir un pseudo à sa guise.

Calibre Une entité e n'est pas seulement un point dans le monde virtuel. Son corps - sa forme - est contenue dans un disque de rayon $ca(e)$ centré en $pos(e)$. Le calibre $ca(e)$ est une valeur constante qui impacte sur l'inertie de l'entité. Pour éviter les entités trop volumineuses, nous avons imposé une valeur maximale pour ca :

$$ca \in [1 \dots 1024]$$

Nombre de voisins espérés Chaque entité spécifie, dès l'initialisation, le nombre de voisins avec qui elle souhaite communiquer dans son environnement virtuel local. Le nombre de voisins espérés dépend du choix de l'utilisateur qui crée l'entité. Il va influer sur le nombre de connexions simultanées et sur le nombre d'événements qu'une entité doit traiter. Il est donc conseillé de tenir compte des capacités physiques de la machine qui héberge l'entité.

$$exp \in \mathbb{N}$$

Les caractéristiques suivantes sont amenées à être modifiées au gré des souhaits de l'utilisateur et en fonction des événements qui surviennent dans le monde virtuel.

Position Comme nous l'avons vu dans le chapitre 2, une entité possède une position dans le monde virtuel torique.

$$pos \in [0 \dots 2^{128}] \times [0 \dots 2^{128}]$$

Une entité connaît sa position absolue dans le tore. Nous verrons par la suite qu'elle connaît également la position absolue de ses voisins. Mais, nous savons que la ligne géodésique joignant deux entités dans le tore utilise la copie de la position de ce voisin la plus proche de l'entité. Ainsi, pour une entité e, la position d'un voisin e' n'est pas la position absolue de e', mais plutôt la copie de la position de e' telle que la ligne géodésique (e, e') est la plus courte. On appelle cette copie de la position de e', la position de e' *relativement* à e.

Nous considérons une entité e désirant connaître la position d'un voisin e' relativement à sa position $(x(e), y(e))$. L'Algorithme 3 décrit le mécanisme détaillé dans la suite. La position absolue de e' est $(x(e'), y(e'))$ et la position relative est $(x_e(e'), y_e(e'))$. L'entité e commence par calculer la valeur absolue de la différence entre chaque coordonnée de sa position et celle de e'. Ainsi, nous notons $\delta_x = |x(e) - x(e')|$ et $\delta_y = |y(e) - y(e')|$.

Pour chaque coordonnée, si cette valeur est supérieure à la moitié de la taille du monde, il existe une copie de cette coordonnée qui est plus proche de celle de e. Par exemple, pour

Algorithme 3 Calcule la position relative de e' pour une entité e

Paramètres d'entrée :
$(x(e'), y(e'))$: position absolue de l'entité e' ;

Paramètres de sortie :
$(x_e(e'), y_e(e'))$: position relative de e' (par défaut $x_e(e') = x(e')$ et $y_e(e') = y(e')$) ;

Actions :
si $x(e') > x(e)$:
  si $x(e') - x(e) > size_x/2$:
    $x_e(e') = x(e') - size_x$
sinon
  si $x(e) - x(e') > size_x/2$:
    $x_e(e') = x(e') + size_x$

si $y(e') > y(e)$:
  si $y(e') - y(e) > size_y/2$:
    $y_e(e') = y(e') - size_y$
sinon
  si $y(e) - y(e') > size_y/2$:
    $y_e(e') = y(e') + size_y$

$\delta_x > size_x/2$ et $x(e) > x(e')$, la copie de la coordonnée de e' la plus proche de $x(e)$ est $x_e(e') = x(e') + size_x$.

A chaque réception d'une position dans le tore, les entités sont tenues de déterminer la position relative en exécutant l'algorithme 3.

Orientation Une entité possède une orientation *ori* qui est déterminée par un entier positif :

$$ori \in \mathbb{N}$$

Rayon de connaissance Le rayon de connaissance $ar(e)$ d'une entité correspond au rayon du disque représentant son environnement virtuel local. Le rayon de connaissance évolue en fonction de la proximité des entités voisines et du nombre espéré de voisins. En fonction de la densité d'entités dans son voisinage, une entité ajustera son rayon de manière à faire correspondre son environnement virtuel local avec le nombre désiré de voisins.

$$ar \in [1 \dots 2^{127}]$$

3.1.2 Structure des données

Une entité partage des connexions avec un certain nombre de voisins. Cela implique la connaissance de certaines de leurs caractéristiques. Tant que les connexions demeurent, ces caractéristiques sont utilisées par les différents algorithmes agissant sur le comportement de l'entité. Ces informations doivent donc être stockées de telle façon que la récupération de ces données permette une réduction du coût de ces algorithmes. D'un autre côté, il ne faut pas que le maintien d'une structure optimisant ces algorithmes nécessite des calculs trop

lourds. Le choix d'une structure des données trouvant un compromis entre la réduction des coûts des algorithmes et le coût du maintien de ces structures est donc crucial au moment de concevoir un programme informatique.

Nous avons opté pour trois structures de données visant des objectifs différents. La première est une classique table de hachage. Une table de hachage est une généralisation de la notion de tableau destinée à stocker des informations indexées par des clés. La recherche d'une donnée stockée, l'insertion d'une nouvelle donnée et la suppression d'une donnée ont une complexité $o(1)$, ce qui fait des tables de hachage des structures de données d'autant plus idéale que les entités de Solipsis disposent d'un identifiant unique que nous utilisons comme clé de stockage.

La deuxième structure de données doit permettre, à une entité, de déterminer rapidement le nombre de voisins localisés dans son environnement virtuel local. Celui-ci est un disque de rayon $r(e)$. Il suffit donc de disposer d'une structure permettant de connaître le nombre d'entités e' dont la distance $d(e, e')$ est inférieure à $r(e)$. Nous avons décidé de maintenir une liste $\mathcal{D}(e)$ triée en fonction de la distance avec e. La $i^{ème}$ entité de la liste est plus proche de e que la $(i+1)^{ème}$ entité. Les opérations effectuées sur une liste triée sont connues pour admettre une complexité $o(\log n)$. Nous avons ajouté une fonctionnalité qui permet de déterminer le nombre d'entités dont la distance est inférieure à un entier positif. Cette fonction peut également être mise en œuvre par un algorithme dichotomique de complexité $o(\log n)$.

La troisième structure de données cherche à minimiser les calculs nécessaires à la vérification de la règle 2.3.1. Une entité e doit être en mesure de vérifier à moindre coût que $\forall e' \in k(e) : s_e e' = e'' \Rightarrow \angle(e'\ e\ e'') < \pi$. Nous avons décidé de maintenir une liste $\mathcal{K}(e)$ triée dans le sens trigonométrique autour de l'entité e. Cette liste vise à faciliter la récupération du successeur de n'importe quel voisin.

Nous avons tout d'abord choisi une ligne de référence afin de disposer d'un moyen de déterminer le premier élément de la liste. Naturellement, le premier élément de la liste est le successeur du dernier élément de la liste car la liste est circulaire. Dans notre première mise en œuvre de Solipsis, nous avons défini la ligne de référence comme étant la demi-droite $\Delta = \{(x, y(e)) : x > x(e)\}$.

Nous notons $p(e)$ un point imaginaire sur la ligne de référence, puis nous définissons une relation $\succ$ de la manière suivante :

$$e_1 \succ_e e_2 \Leftrightarrow \angle(p(e)\ e\ e_1) > \angle(p(e)\ e\ e_2)$$

Quand $e_1 \succ_e e_2$, l'entité e_1 doit se situer *après* l'entité e_2 dans la liste triée. Dans la Figure 3.2, nous avons $e_3 \succ_e e_2 \succ_e e_1$ et $\mathcal{K}(e) = \{e_1, e_2, e_3\}$.

Pour réduire le coût de la fonction de comparaison $\succ$, nous proposons un algorithme décrit dans la suite. Nous définissons deux sous-ensembles $\mathcal{U}(e)$ et $\overline{\mathcal{U}}(e)$ par $\mathcal{U}(e) = \{p \in k(e) : y(p) < y(e)\}$ et $\overline{\mathcal{U}}(e) = \{p \in k(e) : y(p) \geq y(e)\}$. Naturellement, $\mathcal{U}(e) \cup \overline{\mathcal{U}}(e) = k(e)$.

Il est évident que $\forall p \in \overline{\mathcal{U}}(e), \forall q \in \mathcal{U}(e)$, nous avons $p \succ_e q$. Ainsi, dans la Figure 3.2, l'entité e_3 est placée après les entités e_2 et e_1 dans la liste $\mathcal{K}(e)$.

La difficulté réside lorsque les deux entités sont situées dans le même ensemble $\mathcal{U}(e)$ ou $\overline{\mathcal{U}}(e)$. Il faut alors déterminer si l'entité e_2 est située dans le demi-plan délimité par la droite (e, e_1) et suivant le vecteur $\vec{n}$, vecteur orthogonal à (e, e_1) dans le sens trigonométrique. Dans la Figure 3.2, ce demi-plan apparaît grisé.

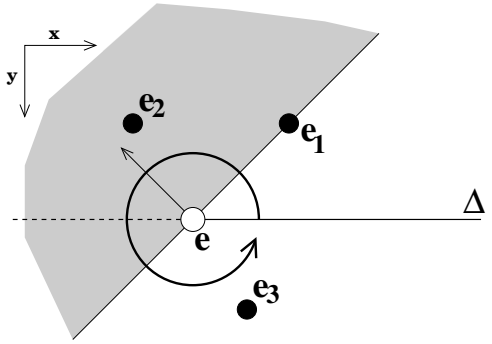

FIG. 3.2 – Trois entités e_1, e_2 et e_3 qu'une entité e doit trier.

Il existe une méthode pour déterminer aisément si un point est dans un demi-plan [83]. Il suffit de vérifier le signe du produit scalaire entre le vecteur $\overrightarrow{ee_1}$ et le vecteur $\overrightarrow{ee_2}$:

$$\overrightarrow{ee_2} \cdot \overrightarrow{ee_1} = (y(e_1) - y(e))(x(e_2) - x(e)) + (x(e) - x(e_1))(y(e_2) - y(e_1))$$

Si ce signe est positif et que les deux entités appartiennent à l'ensemble $\mathcal{U}(e)$ ou si le signe est négatif avec les entités dans $\overline{\mathcal{U}}(e)$, alors e_2 est dans le demi-plan et, naturellement, $e_2 \succ e_1$. Dans les autres cas, $e_1 \succ e_2$.

Ce calcul permet de comparer deux entités. Il est alors aisé de construire et maintenir une liste triée avec la fonction de comparaison $\succ$. Toutes les opérations effectuées sur cette liste ont une complexité $o(\log n)$.

3.1.3 Tâches périodiques

Le nœud est un programme qui utilise ses caractéristiques et les données dont il dispose pour accomplir un certain nombre de tâches permettant à l'entité de participer au réseau de pairs, et donc au monde virtuel. Dans la suite, nous décrivons les tâches qui sont accomplies à intervalle régulier.

Détection de fermeture de connexion Les entités sont susceptibles de quitter le système et donc de fermer toutes leurs connexions à chaque instant. Or, les algorithmes de Solipsis reposent sur les voisins dont on suppose la présence dans le monde virtuel. Nous avons besoin d'un mécanisme permettant à une entité de détecter ses voisins qui ont quitté le système, c'est-à-dire les connexions qui ne sont plus valides.

Pour s'assurer qu'une entité est toujours connectée au monde virtuel, nous nous basons sur sa capacité à émettre des messages. Pour chaque connexion, les deux entités sont tenues d'émettre au moins un message dans un intervalle de temps δ. Nous fournissons un mécanisme simple qui permet à une entité e à l'instant t de (1) détecter que ses voisins sont encore actifs, (2) notifier à ses voisins que e est toujours active.

En premier lieu, l'entité e vérifie qu'elle a effectivement reçu au moins un message de la part de tous ses voisins. Si un voisin n'a pas émis de message dans un temps δ, ce voisin est suspecté. Si un voisin déjà suspecté n'a pas émis de message, il est considéré comme ayant quitté le système et la connexion est fermée.

L'ensemble des entités suspectées à t est noté $\mathcal{S}(t)$ et l'ensemble des voisins qui n'ont pas émis de message depuis $t-\delta$ est noté $\mathcal{I}(t)$. Les entités considérées comme ayant quitté le système appartiennent à l'ensemble $\mathcal{S}(t) \cap \mathcal{I}(t)$. Les entités suspectées, qui appartiendront à $\mathcal{S}(t+\delta)$, sont $\mathcal{I}(t) \setminus \mathcal{S}(t)$.

Dans un second temps, l'entité e doit émettre un message qui permettra à ses voisins de savoir qu'elle n'a pas quitté le monde virtuel. Pour cela, il suffit que e sélectionne, parmi ses voisins, ceux à qui elle n'a pas envoyé de messages, puis elle leur envoie un message de notification de présence. De fait, l'entité e a émis un message dans l'intervalle $[t-\delta, t]$.

Cet algorithme simple garantit qu'une entité ayant quitté le monde Solipsis à l'instant t ne peut plus être considérée comme un voisin au pire à l'instant $t + 2 * \delta$ et au mieux à l'instant $t + \delta + \epsilon$. En choisissant δ très faible, la cohérence entre la réalité physique des connexions et la supposition de l'existence de cette connexion est très importante. D'un autre côté, cela provoquerait beaucoup de messages de notification de présence. Il faut donc trouver un compromis entre la cohérence et le coût induit pas ces échanges de messages. Dans l'actuelle version du programme Solipsis, nous avons choisi, par défaut, $\delta = 10$ secondes.

Respect de la règle 2.3.1 En principe, une entité doit respecter la règle 2.3.1 à chaque instant, et, notamment, à l'issue de chaque événement virtuel. Cette obligation présente des risques de fortes charges de calcul si l'entité se situe dans une partie du monde virtuelle riche en événements. C'est pourquoi nous avons préféré réaliser cette vérification à intervalle régulier et uniquement si un événement est survenu dans le monde virtuel.

La méthode la plus simple consiste à sélectionner chaque voisin itérativement dans la liste triée $\mathcal{K}(e)$ et de vérifier que son successeur est situé dans le demi-plan délimité par la droite formé par le voisin et l'entité. La liste étant circulaire, le dernier élément de la liste doit établir cette vérification avec le premier élément de la liste. Cette méthode nécessite de parcourir l'ensemble de la liste et a donc une complexité $o(n)$.

Nous proposons un autre algorithme réduisant la complexité de cette tâche (cf Algorithme 4). Le principe est le suivant. Nous considérons un sous-ensemble d'entités $\Sigma \subseteq \mathcal{K}(e)$. Soit e_0, e_i et e_k respectivement la première entité, l'entité médiane et la dernière entité de Σ. L'algorithme analyse les angles $\angle(e_0\ e\ e_i)$ et $\angle(e_i\ e\ e_k)$. Si l'un des deux angles est supérieur à π, l'algorithme sélectionne alors l'ensemble des entités contenues dans cet angle. Récursivement, il applique la même démarche avec Σ réinitialisé avec cet ensemble.

Lors de l'exécution de l'algorithme, l'ensemble Σ est initialisé avec $\mathcal{K}(e)$ auquel on ajoute la première entité e_1 de $\mathcal{K}(e)$ en dernière position. Forcément, il existe au moins un angle $\angle(e_1\ e\ e_i)$ ou $\angle(e_i\ e\ e_1)$ qui est supérieur à π. Admettons par exemple que $\angle(e_i\ e\ e_1) > \pi$. Cela signifie que les entités $e_i \dots e_n, e_1$ sont susceptibles de ne pas respecter la règle. L'ensemble Σ est alors reinitialisé avec $e_i \dots e_n, e_1$ et l'algorithme effectue récursivement la même analyse.

L'algorithme peut terminer de deux manières. Si Σ ne contient que deux éléments e_j et e_{j+1}, alors $\angle(e_j\ e\ e_{j+1}) > \pi$ et e ne respecte pas la règle. En revanche, si les deux angles à considérer sont inférieurs à π alors e respecte la règle. La complexité de cet algorithme est $o(\log n)$.

Algorithme 4 Vérifie qu'une entité e respecte la règle 2.3.1

Paramètres d'entrée :

$\mathcal{K}$: ensemble des voisins triés dans l'ordre trigonométrique ;
$\mathcal{K}[i]$: sélectionne le i^{ème} élément de la liste $\mathcal{K}$;
$dsDemiPlan(e_1, e_2)$: fonction booléenne `vrai` si $\angle(e_1\ e\ e_2) < \pi$;

Actions :

$begin = 0$
$end = (|\mathcal{K}| + 1) \bmod |\mathcal{K}|$
tant que $begin + 1 \neq end$:
  $e_1 = \mathcal{K}[begin]$
  $i = begin + \frac{end - begin}{2}$
  $e_2 = \mathcal{K}[i]$
  $e_3 = \mathcal{K}[end]$
  si $dsDemiPlan(e_1, e_2)$:
    si $dsDemiPlan(e_2, e_3)$:
      retourne `vrai`
    sinon :
      $begin = i$
  sinon :
    $end = i$
retourne `faux`

Gestion du rayon de connaissance Le rayon de connaissance est le rayon du disque représentant l'environnement virtuel local d'une entité. Cet environnement virtuel local doit contenir un nombre constant de voisins correspondant à la caractéristique exp. Cette contrainte est forte car elle oblige chaque entité à ajuster son rayon de connaissance à chaque déplacement de la $exp(e)^{ième}$ entité la plus proche.

Nous avons décidé d'admettre que le nombre de voisins pouvait être compris dans un intervalle autour de exp. Ainsi, nous associons à chaque entité e une constante réelle $f(e) \in [0, 1]$. Le but de la tâche périodique décrite dans la suite (cf Algorithme 5) est de garantir qu'à intervalle régulier, une entité e est connectée avec δ voisins situés dans son environnement virtuel local, de telle manière que

$$\delta \in [exp - (f(e) * exp(e)),\ exp(e) + (f(e) * exp(e))]$$

La constante $f(e)$ dépend du choix de l'utilisateur. Plus cette constante est faible, plus l'entité cherchera à ajuster son rayon de connaissance de manière à avoir un nombre d'entités dans son environnement virtuel local proche de la valeur espérée. Plus $f(e)$ se rapproche de 1, plus l'entité est tolérante vis-à-vis de ce nombre d'entités.

Régulièrement, il s'agit de compter le nombre δ d'entités situées dans l'environnement virtuel local. Cette opération est obtenue grâce à la liste triée $\mathcal{D}(e)$. Ensuite, nous vérifions que δ est bien compris dans l'intervalle $[exp(e) - (f(e) * exp(e)),\ exp(e) + (f(e) * exp(e))]$. Si c'est le cas, la valeur du rayon de connaissance est appropriée à la situation.

Si δ est supérieur à $exp(e) + (f(e) * exp(e))$, cela signifie que le rayon de connaissance est trop grand par rapport aux possibilités de la machine, aux souhaits de l'utilisateur et à la densité d'entités dans cette région virtuelle. Il faut alors réduire le rayon de connaissance de telle manière qu'à l'issue de l'opération, $\delta = exp(e)$. Il suffit de choisir l'entité e' située

à la $exp(e)^{ème}$ position dans la liste $\mathcal{D}(e)$, puis d'ajuster le rayon de connaissance $r(e)$ tel que $r(e) = d(e, e')$.

Algorithme 5 Gestion du rayon de connaissance (entité e)

Paramètres en entrée :
$\mathcal{D}$: ensemble des voisins triés dans l'ordre de distance avec e ;
$exp(e)$: nombre de voisins espérés dans l'environnement virtuel local ;
$f(e)$: constante réelle comprise entre 0 et 1 ;
$r(e)$: rayon de connaissance de l'entité e ;

Fonctions :
$dist(i)$: retourne la distance de la $i^{ème}$ entité de $\mathcal{D}(e)$;
$count()$: retourne le nombre d'entités situées dans l'environnement virtuel local de e ;

Actions :
$min = exp(e) - (exp(e) * f(e))$
$max = exp(e) + (exp(e) * f(e))$
si $count() > max$:
  $r(e) = dist(exp(e))$
si $count() < min$:
  si $|\mathcal{D}(e)| > min$:
    si $|\mathcal{D}(e)| > exp(e)$:
      $r(e) = dist(exp(e))$
    sinon
      $r(e) = dist(|\mathcal{D}(e)|)$
  sinon
    $r(e) = r(e) * \sqrt{exp(e)/count()}$

Si δ est inférieur à $exp(e) - (f(e) * exp(e))$, le rayon de connaissance est trop petit. Il faut alors l'augmenter de manière à retrouver δ compris dans l'intervalle toléré. Il est possible que e soit connecté à des entités en dehors de son environnement virtuel local. Si le nombre total de voisins de e est supérieur à $exp(e) - (f(e) * exp(e))$, il suffit d'ajuster le rayon de connaissance de manière à englober dans l'environnement virtuel local un nombre d'entités le plus proche du nombre espéré. Mais, si e ne connaît pas suffisamment d'entités, il faut alors choisir d'augmenter le rayon de connaissance sans savoir si la valeur choisie permettra de revenir à une situation correcte.

De nombreuses techniques sont utilisables. La plus simple consiste à multiplier le rayon actuel par un facteur déterminé (par exemple, 1.25) et espérer que cette augmentation du rayon permettra de rencontrer de nouveaux voisins. Il faudra alors ajuster convenablement le rayon lors de la prochaine exécution de cette tâche périodique.

Plus subtilement, on peut imaginer que la densité des entités sera uniforme dans la région virtuel. Pour une entité e, une approximation de cette densité $\gamma(e)$ peut être obtenue par la connaissance de l'aire de l'environnement virtuel local et le nombre de voisins δ :

$$\gamma(e) = \frac{\delta}{\pi * r(e)^2}$$

Il faut extrapoler de cette connaissance une valeur $r'(e)$ telle que, si la densité est uniforme, le nombre d'entités dans le disque de rayon $r'(e)$ sera égal à $exp(e)$. On obtient

alors :

$$\gamma(e) = \frac{exp(e)}{\pi * r'(e)^2} \Rightarrow r'(e) = r(e) * \sqrt{\frac{exp(e)}{\delta}}$$

En utilisant la valeur du rayon ainsi extrapolée, l'entité a de bonnes chances d'obtenir un nombre de voisins situés dans son environnement virtuel local proche de la valeur espérée.

Politique de voisinage Une entité peut parfois posséder un nombre trop important de connexions. Le nombre de messages reçus dépasse la capacité de la machine et il est nécessaire de fermer quelques connexions. Il lui faut donc choisir une entité parmi ses voisins de telle manière que la fermeture de la connexion les liant ne bouleverse pas les propriétés du réseau de pairs. La tâche périodique décrite dans l'Algorithme 6 met en place une politique de sélection du voisin dont la connaissance n'est plus jugée nécessaire.

Bien entendu, l'entité e ne peut pas fermer la connexion avec un voisin situé dans son environnement virtuel local. De même, elle ne va pas choisir une entité cruciale pour le maintien de la règle 2.3.1. Il serait, par ailleurs, malvenu de couper la connexion avec une entité e' telle que e appartienne à l'environnement virtuel local de e' ou telle que e' ne respecterait pas la règle 2.3.1. Formellement, nous obtenons :

$$e \notin \mathcal{A}(e') \tag{3.1}$$

$$e' \notin \mathcal{A}(e) \tag{3.2}$$

$$pos(e) \notin CH(k(e) \setminus \{e'\}) \tag{3.3}$$

$$pos(e') \notin CH(k(e') \setminus \{e\}) \tag{3.4}$$

L'entité e ne peut pas savoir si la fermeture de la connexion garantit la quatrième propriété. En revanche, elle est capable de déterminer les entités e' respectant les trois premières. Nous notons $\mathcal{H}(e)$ ces entités.

Pour obtenir la liste $\mathcal{H}(e)$, il suffit de sélectionner dans la liste $\mathcal{D}(e)$ le premier voisin dont la distance avec e est supérieur à $r(e)$ puis de tester, sur tous les voisins suivants dans la liste, les propriétés énoncées.

Algorithme 6 Sélection du voisin à supprimer (entité e ayant trop de voisins)

Fonctions :

$checkGC(l)$: retourne vrai si la liste de voisins l permet à e de respecter la règle 2.3.1 ;
$choice(l)$: choisit une entité parmi la liste l ;

Actions :

$\mathcal{H}(e) = \{i \in \mathcal{D}(e) : d(e,i) > r(e)\}$
pour tout $i \in \mathcal{H}(e)$:
  si $(d(i,e) \leq r(i))$ **ou** (**non** $checkGC(k(e) \setminus \{i\})$)
    $\mathcal{H}(e) = \mathcal{H}(e) \setminus \{i\}$
retourne $choice(\mathcal{H}(e))$

Si $\mathcal{H}(e)$ contient plus d'un élément, l'entité e est libre de choisir de couper la connexion avec n'importe quel élément de cet ensemble. Nous proposons dans la suite plusieurs possibilités de faire ce choix :

- le choix le plus facile consiste à sélectionner une entité $e' \in \mathcal{H}(e)$ au hasard
- il est envisageable que e puisse se baser sur des statistiques pour faire le choix. Par exemple, il est possible de trier les éléments de $\mathcal{H}(e)$ grâce au temps de présence dans le monde virtuel. Une entité qui est connectée depuis longtemps à Solipsis a une plus grande probabilité de rester présente dans le monde qu'une entité qui est connectée depuis peu de temps. Ainsi, l'entité e peut choisir l'entité e' telle que e' est l'entité la plus récente dans Solipsis parmi les entités de $\mathcal{H}(e)$
- une entité qui possède un rayon de connaissance très grand a la connaissance d'une vaste région du monde. Il est plus intéressant de rester connecté avec ce voisin plutôt qu'avec une entité dont le rayon de connaissance est plus petit. Il est donc possible de trier l'ensemble $\mathcal{H}(e)$ en fonction de la taille du rayon de connaissance et de choisir l'entité ayant le rayon de connaissance le plus petit
- une entité dont l'environnement virtuel local couvre une grande partie de la frontière de l'environnement virtuel local de e a plus de chance de détecter une entité entrant dans cet environnement virtuel local qu'une entité dont la couverture est faible. De fait, il est possible de trier les entités de $\mathcal{H}(e)$ en fonction de leur couverture de la frontière de l'environnement virtuel local de e
- il est également possible de calculer le périmètre de l'enveloppe convexe de l'ensemble des voisins privé de e'. Il semble intéressant de couper la connexion avec un voisin de telle manière que ce périmètre soit le plus petit
- enfin, la solution que, par défaut, nous avons choisi, consiste à sélectionner, parmi les entités de $\mathcal{H}(e)$, l'entité e' qui est la plus éloignée de e.

Rafraîchissement du fichier d'initialisation Les lecteurs attentifs auront remarqué qu'une entité désirant se connecter au système Solipsis doit, au préalable, connaître une entité présente dans le monde virtuel. Quand un utilisateur télécharge le programme [55], il récupère un fichier `entities.met` contenant un certain nombre d'entités qui ont de bonnes chances d'exister dans le monde virtuel.

Mais, il est important que l'entrée dans le monde virtuel ne dépende pas de la présence de ces entités. Pour ce faire, chaque participant au monde conserve des informations à propos de quelques entités croisées dans le monde virtuel. En ajoutant ces informations au fichier `entities.met`, il enrichit sa capacité à se connecter au monde virtuel lors de la prochaine connexion. Nous avons décidé d'ajouter ces informations au fichier à intervalle régulier.

Le mécanisme que nous avons mis en œuvre a le fonctionnement suivant. A chaque fois qu'une entité croise un nouveau voisin, il ajoute les caractéristiques de ce voisin à un liste $\mathcal{M}(e)$. A intervalle régulier (choisi suffisamment grand, typiquement une dizaine de minutes), l'entité e récupère toutes les informations contenues dans le fichier `entities.met`, ainsi que toutes les informations de la liste $\mathcal{M}(e)$.

Si le nombre d'entités que e connaît est inférieur à une variable fixée γ (par défaut $\gamma = 1000$), alors toutes les informations sont sauvegardées dans le fichier `entities.met`. Mais, il faut éviter que la taille du fichier ne croisse avec le nombre de voisins rencontrés. De fait, e choisit alors, au hasard, γ entités et sauvegarde les informations de ces entités dans `entities.met`. Puis, il vide la liste $\mathcal{M}(e)$.

Le fait de sélectionner au hasard parmi les entités du fichier et les entités rencontrés

depuis la dernière exécution de l'algorithme permet à l'entité de rafraîchir son fichier avec les dernières entités rencontrées sans pour autant perdre la connaissance des "anciennes" entités. En effet, il est important de connaître les entités les plus récentes dans le monde, mais, d'un autre côté, il peut être utile de conserver trace de l'existence de quelques entités stables dans le monde.

Divers algorithmes sont envisageables. Par exemple, il pourrait être intéressant de sauvegarder prioritairement les entités qui ne bougent pas dans le monde, car il y a de fortes probabilités que ces entités soient des "objets" virtuels, entités stables présentant peu de risques de déconnexion.

3.1.4 Gestion des événements

Les algorithmes que nous avons présenté jusqu'à présent sont exécutés à intervalle régulier et assurent que le comportement général de l'entité respecte les principes généraux de Solipsis. Mais, parallèlement, il faut traiter les différents événements intervenant dans le monde virtuel. Nous présentons brièvement ceux-ci dans la suite.

Les interventions de l'utilisateur Lorsqu'un navigateur est connecté au nœud, un utilisateur peut, à tout moment, décider de modifier certaines caractéristiques de l'entité. Il utilise alors l'interface de commande décrite dans la Section 3.3.

Plusieurs types d'événements sont envisageables. Tout d'abord, l'utilisateur peut décider de quitter le monde virtuel. Il faut alors que le nœud cesse toute activité. Il peut également décider de se déplacer dans le monde virtuel. Il faut alors modifier la position de l'entité. Enfin, l'utilisateur peut choisir de se téléporter à une position distante.

Les événements du monde virtuel Un nœud est connecté à d'autres nœuds qui ont un comportement qui n'est pas prévisible. Notamment, ces voisins sont susceptibles de disparaître du monde virtuel, mais aussi de se déplacer virtuellement, de se téléporter à une autre position.

Dans la Section 3.2, nous décrivons les messages permettant de notifier ces événements et le traitement qui est effectué à la réception de ces messages.

3.2 Le protocole Solipsis

Dans cette Section, nous décrivons les messages que deux nœuds peuvent s'échanger lorsqu'ils partagent un lien de communication. Nous détaillons également le traitement de ces messages. Tout d'abord, nous présentons les actions effectuées par une entité lors de la phase de connexion. Puis, nous décrivons les événements liés à la mobilité des intervenants.

3.2.1 Connexion

La phase de connexion d'une entité au monde virtuel se déroule en deux étapes distinctes. Il faut d'abord que l'entité récupère un certain nombre d'informations, en particulier les informations concernant une entité connectée au monde virtuel. Ensuite, l'entité peut utiliser l'algorithme présenté dans la Section 2.5. Dans la suite, nous décrivons la

méthode utilisée pour récupérer ces informations, puis la mise en œuvre de l'algorithme de connexion.

Récupération des informations nécessaires

Quand une nouvelle entité veut se joindre au monde virtuel, elle doit tout d'abord (1) déterminer son adresse Internet et (2) découvrir une entité connectée à Solipsis.

Cette première tâche n'est pas aussi simple qu'il n'en parait. L'adresse Internet est une donnée qui caractérise une machine sur le réseau. Par exemple, l'adresse Internet est insérée dans un message, ce qui permet aux routeurs de transmettre le message et au destinataire de connaître l'émetteur de ce message et, éventuellement, de lui répondre. De nos jours, une machine peut disposer de plusieurs éléments physiques (cartes réseaux) lui permettant de se connecter à Internet. Les systèmes d'exploitation sont libres de commander n'importe quel élément pour communiquer, mais il arrive bien souvent que l'information concernant l'élément effectivement utilisé ne soit pas accessible. Par ailleurs, une machine a appartenant à un réseau local peut accéder à Internet via une machine b utilisant un mécanisme appelé NAT (*Network Address Translator*) [77]. Ce mécanisme permet de multiplier les adresses à l'intérieur d'un réseau local, mais, en contrepartie, il ne permet pas à la machine a de connaître l'adresse qu'elle utilise réellement pour accéder à Internet. Il est donc indispensable de fournir aux machines désirant se connecter à Solipsis un mécanisme leur permettant de déterminer leur véritable adresse Internet.

Dans un premier temps, nous considérons que la machine e désirant créer une nouvelle entité possède un fichier `entities.met` contenant une liste d'entités potentiellement connectées au monde virtuel. La machine e envoie à ces entités un message indiquant qu'elle désire créer une nouvelle entité.

Quand une entité e' reçoit le message de connexion de e, elle observe d'abord l'adresse réelle de l'émetteur du message $i(e)$. Cet émetteur peut être la machine e elle-même ou une machine agissant selon le principe du NAT. L'entité e' envoie alors un message de réponse contenant l'information $i(e)$. Quand la machine e reçoit la réponse de e', elle découvre une entité connectée (l'entité e') et son adresse Internet (l'information $i(e)$). L'adresse Internet d'une entité est de la forme *host* et *port*.

Algorithme de connexion

L'algorithme de connexion consiste à découvrir la plus proche entité de la position $pos(e)$ désirée, puis à tourner autour de cette position de manière à collecter les entités permettant à e de respecter la règle 2.3.1. Cet algorithme peut être utilisé lors de la connexion au monde virtuel mais également lorsque l'entité e désire se téléporter à une position distante.

Trouver l'entité la plus proche L'entité e connaît son identifiant $id(e)$, sa position désirée $pos(e)$ et son adresse Internet *host* et *port*. Le premier message envoyé à e' contient ces informations.

$$< FindNearest, host(e), port(e), pos_x(e), pos_y(e) >$$

A la réception de ce message, l'entité e' choisit une entité $e'' \in k(e')$ qu'il considère comme l'entité la plus proche de la position $pos(e)$. Elle répond par un message $Nearest$ contenant les informations permettant à e de communiquer avec e'' ou par un message $Best$ si elle se considère, elle-même, comme l'entité la plus proche de la position.

Un message $Nearest$ est la réponse classique à un message $FindNearest$. Il contient des informations à propos de l'entité la plus proche de la position spécifiée dans le message $FindNearest$:

$$< Nearest, id(e''), host(e''), port(e''), pos_x(e''), pos_y(e'') >$$

Ce message est envoyé par une entité e' après réception d'un message $FindNearest$ issu de e et les informations qu'il contient concernent une entité $e'' \in k(e)$.

Quand l'entité e reçoit ce message, elle sait qu'elle connait maintenant, par e'', une entité qui la rapproche de l'entité la plus proche de la position $pos(e)$. Elle est alors capable d'envoyer un nouveau message $FindNearest$ à e'' et, ainsi, récursivement jusqu'à ce que e reçoive un message $Best$.

Un message $Best$ est une autre réponse à un message $FindNearest$. Il permet à son émetteur de notifier à l'entité à l'initiative du message $FindNearest$ qu'il se considère comme l'entité la plus proche de la position demandée.

$$< Best, id(e'), host(e'), port(e'), pos_x(e'), pos_y(e') >$$

Ce message est émis par une entité e' qui, recevant un message $FindNearest$, observe qu'elle est plus proche de $pos(e)$ que l'entité e'', l'entité qui, parmi ses voisins, est la plus proche de $pos(e)$. Les informations contenues dans ce message sont les propres informations de e'. Par ce biais, elle informe e de la fin (potentielle) de la première phase de l'algorithme de connexion.

Quand l'entité e reçoit un message $Best$, elle prétend avoir terminé la première phase et peut donc entamer la seconde phase de l'algorithme consistant à tourner autour de $pos(e)$.

Tourner autour de la position L'entité e connaît une entité e_c considérée comme la plus proche de $pos(e)$ dans Solipsis. Elle calcule alors la distance, notée $d(e, e_c)$, séparant l'entité e_c de $pos(e)$. Puis, elle émet un message $Queryaround$ afin de découvrir les entités autour de sa position.

$$< Queryaround, id(e), host(e), port(e), pos_x(e), pos_y(e), id(e_c), d(e, e_c) >$$

Une entité e' recevant un message $Queryaround$ vérifie d'abord que l'entité e_c est effectivement plus proche de la position $pos(e)$ que tous ses voisins dans $k(e')$. Si e' connaît une entité $e'' \in k(e')$ dont la distance avec $pos(e)$ est inférieure à $d(e, e_c)$, elle envoie immédiatement à e un message $Nearest$ contenant les informations sur e''. L'algorithme revient alors à la première phase.

Si e_c est effectivement la plus proche, l'entité e' choisit alors, parmi ses voisins $k(e')$ l'entité e'' la plus proche de $pos(e)$ telle que e'' est située dans le demi-plan, délimité par la droite joignant e et lui-même, dans le sens trigonométrique. L'entité e' répond alors au message $Queryaround$ par un message $Around$.

Le message *Around* est une réponse à un message *Queryaround*. La réception de ce message, outre les informations qu'il contient, permet à l'entité réceptrice de confirmer son hypothèse que l'entité e_c est la plus proche de $pos(e)$.

$$< Around, id(e''), host(e''), port(e''), pos_x(e''), pos_y(e'') >$$

A la réception d'un message *Around* contenant les informations à propos d'une entité e'', une entité e doit vérifier que la connaissance de cette entité ne lui permet pas de terminer la seconde phase. Si l'entité e'' est l'entité e_c ou si e' et e'' sont de part et d'autres de la demi-droite $[e, e_c)$, alors le tour est achevé, e respecte la règle 2.3.1 et peut alors se connecter avec toutes les entités rencontrées durant la seconde phase. Si le tour n'est pas achevé, l'entité e envoie, de nouveau, un message *Queryaround* à l'entité e''.

3.2.2 Préservation de la topologie

Une fois connectée, une entité doit participer au maintien du réseau de pairs tel qu'il a été présenté dans le chapitre 2. Dans la suite, nous décrivons les messages qui permettent à une entité d'adopter un comportement en adéquation avec les principes de Solipsis.

Gestion des connexions

Tout d'abord, nous nous intéressons aux messages permettant d'initier, de fermer et de maintenir une connexion entre deux entités.

Ouverture/fermeture Le message *hello* permet à une entité e de notifier à une entité e' la volonté d'ouvrir une nouvelle connexion. Ce message contient les informations de e.

$$< hello, id, host, port, pos_x, pos_y, ar, ca, pseudo, ori >$$

Ce message est envoyé par e sur la base d'un renseignement qui lui a été fourni par une autre entité. L'entité e connaît l'adresse Internet de e', ce qui lui permet d'émettre ce message.

L'entité e' qui reçoit ce message ne peut pas refuser la connexion. Elle répond par un message *connect* validant l'ouverture de la connexion, puis ajoute immédiatement e à ses structures de données.

$$< connect, id, host, port, pos_x, pos_y, ar, ca, pseudo, ori >$$

A la réception d'un message *connect*, l'entité e vérifie d'abord qu'elle a effectivement émis un message *hello*, puis valide l'existence de la connexion. Ce message lui donne, en outre, la connaissance des informations de e'. L'entité e peut donc intégrer l'entité e' à ses structures de données.

Une entité peut couper une connexion en émettant le message *close*.

$$< close, id >$$

Dès réception d'un message *close*, une entité e' considère que l'entité e n'est plus un voisin et ôte immédiatement e de ses structures de données.

Maintien des connexions Afin de notifier à ses voisins qu'elle est connectée au monde virtuel, une entité e doit envoyer, à intervalle régulier, un message. Quand aucune information n'est échangée, il est nécessaire qu'elle envoie le message de notification de présence suivant :

$$< heartbeat, id >$$

Une entité e' qui reçoit un tel message est informée que e participe toujours au monde virtuel.

Gestion de la mobilité

Quand une entité e modifie une de ses caractéristiques, il informe ses voisins en émettant un message *delta* contenant son identifiant, la variable modifiée et la nouvelle valeur de cette caractéristique. Si la caractéristique modifiée est la position, alors la variable est `POS` et la nouvelle valeur est la nouvelle position. De même, si la modification concerne le rayon de connaissance, alors la variable est `AR`.

$$< delta, id, var, newvalue >$$

Une entité e' recevant ce message doit, si nécessaire, rafraîchir ses structures de données. Notamment, les listes triées peuvent être affectées par cette modification. Ensuite, l'entité e' doit vérifier que cette modification n'a pas d'impact sur la topologie de Solipsis et ne doit pas entraîner de message de détection (cf Algorithme 1).

Collaboration spontanée Un message de détection *detect* permet à une entité e' de fournir à une entité e le moyen de se connecter à une entité e'', suivant le mécanisme décrit dans la Section 2.3.1.

$$< detect, id(e''), host(e''), port(e'') >$$

Ce message est envoyé par l'entité e' après réception d'un message *hello* ou d'un message *delta* de l'entité e. Les informations qu'il contient concerne l'entité e''. Si l'entité e' s'aperçoit que e est maintenant plus près de e'' que lui-même ou si e est maintenant dans l'environnement virtuel local de e'', alors le message *detect* doit être envoyé.

Lorsqu'une entité e reçoit un message *detect*, elle doit faire confiance à e'. Après vérification qu'elle n'est pas déjà connectée à e'', elle ouvre immédiatement une connexion avec e'' en émettant un message *hello* grâce aux informations contenues dans le message *detect*.

Respect de la règle 2.3.1 Un message *search* émis par une entité e à une entité e' permet à e de requérir la collaboration de e' pour récupérer un nouveau voisin e'' dans le demi-plan délimité par la droite (e, e') dans le sens trigonométrique si *wise* vaut 1 ou dans le sens des aiguilles d'une montre si *wise* vaut 0. Ce message agit conformément au principe de la Section 2.3.2.

$$< search, id, wise >$$

Quand une entité e' reçoit un message *search*, elle recherche parmi ses voisins une entité e'' telle que e'' est l'entité la plus proche de e dans le demi-plan délimité par la droite (e, e') dans le sens précisé par *wise*. Pour accomplir cette tâche, l'entité e' peut s'aider de la liste triée dans le sens trigonométrique en restreignant sa recherche aux entités situées dans le demi-plan considérée.

La réponse de l'entité e' est un message $found$ contenant les informations qui vont permettre à l'entité e de rentrer en communication avec une entité e'' sélectionnée.

$$< found, id(e''), host(e''), port(e''), pos_x(e''), pos_y(e'') >$$

Quand l'entité e reçoit un tel message, il peut ouvrir une connexion avec e'' et, si elle le juge nécessaire, renvoyer un message *search* à e'' en espérant que l'entité e'' lui communique une nouvelle entité qui lui permettra de respecter de nouveau la règle 2.3.1.

3.3 L'interface de commande

Dans cette Section, nous nous intéressons aux informations échangées entre un navigateur et un nœud. Avant de détailler cette interface de commande, il est important de comprendre le fonctionnement de telles interfaces.

Une interface de commande permet à un programme a d'interopérer avec un programme b. Les programmes a et b peuvent être exécutés sur des ordinateurs différents, munis de systèmes d'exploitation différents, et peuvent être écrits dans un langage de programmation différent. Le programme b permet à a d'exécuter une procédure incluse dans le code de b. Cette opération est possible grâce à une description de cette procédure dans un langage commun à a et à b. Parmi les langages de description, on peut citer CORBA [56], XML-RPC [59] ou SOAP [58].

Nous n'avons pas voulu réaliser un choix parmi ces différentes techniques de création d'interface de commande. Nous avons donc préféré élaborer une phase de connexion qui permettra aux deux modules de déterminer l'interface de commande qu'ils utilisent.

Ensuite, nous détaillerons les procédures, situées dans le nœud et auxquelles le navigateur peut accéder. Enfin, nous présenterons les procédures qui appartiennent au navigateur, et accessibles au nœud

3.3.1 Connexion

Nous considérons au préalable que le navigateur possède la connaissance de l'adresse Internet du nœud avec lequel il veut se connecter. De plus, nous considérons que ce nœud existe et qu'il est capable de traiter les messages qui lui sont destinés.

Le navigateur initie la connexion avec le nœud en lui envoyant le message suivant :

$$< openGui, protocol, host, port >$$

Ce message indique qu'un utilisateur veut prendre le contrôle de l'entité. Il s'agit donc de la notification de la création d'une interface entre un homme et une entité. Nous avons décidé de n'autoriser qu'une seule interface homme-machine par entité.

Le second message possible est :

$$< openMedia, idMedia, protocol, host, port, pushOrPull >$$

Ce message permet au navigateur de spécifier au nœud qu'il désire recevoir toutes les informations virtuelles relatives à l'immersion de l'entité dans le monde virtuel. De cette manière, le navigateur devrait être capable de rendre compte de l'expérience. Nous ne limitons pas le nombre de navigateurs simultanés par nœud.

Dans ces deux messages, le *protocol* est une description de l'interface de commande que le navigateur propose d'utiliser. La communication par une interface de commande ne contraint pas les navigateurs et les nœuds à coexister sur une machine, ni même à être programmé dans le même langage.

Le *host* et le *port* sont les données qui vont permettre au nœud de communiquer avec le navigateur.

La donnée $idMedia$ est un identifiant que le navigateur choisit et qui lui permet de se différencier des autres navigateurs.

Enfin, la donnée $pushOrPull$ est une variable booléenne. Une valeur vrai signifie que le navigateur désire être informé, immédiatement, de chaque événement virtuel survenu dans l'environnement virtuel. Dans le cas contraire, le navigateur indique au nœud qu'il demandera, parfois, l'exécution de procédures pour récupérer les informations du monde virtuel. Cette variable permet de créer des navigateurs poursuivant des buts variables. Par exemple, un navigateur désirant retranscrire graphiquement le monde virtuel a intérêt à être réactif aux événements virtuels. Typiquement, un tel navigateur initialisera sa variable à la valeur vrai. Inversement, on peut imaginer qu'un utilisateur désire créer un objet virtuel comme une maison. Cet objet n'a pas de perception et n'est pas intéressé par les mouvements des entités aux alentours. En revanche, cet objet veut fournir à ses voisins une description graphique particulière. Le navigateur peut donc se contenter de récupérer, à intervalle régulier, des informations concernant les entités les plus proches disposées à recevoir cette description graphique.

A l'issue de ces deux messages, le nœud sait qu'un navigateur, disposant éventuellement d'une interface homme-machine, désire communiquer avec lui via une interface qui sera accessible à une adresse réseau spécifique. Le nœud peut alors accepter cette "connexion" avec un message *accept* dans lequel il précise une adresse réseau spécifique pour l'interfaçage, ou la refuser par un message *refuse*.

3.3.2 Procédures à disposition du Navigateur

Les procédures décrites dans la suite sont exécutées par le nœud. Elles sont accessibles par le navigateur via l'interface de commande. La bonne exécution de ces procédures est validée par une réponse "1" tandis qu'une erreur, due par exemple à une mauvaise initialisation d'un des paramètres, produit un "0".

Fermeture de connexion

Quand le navigateur désire achever la connexion avec le nœud, il doit, au préalable, prévenir le nœud. Il peut le faire grâce à l'appel de la procédure :

$$closeMedia(idMedia)$$

Si l'utilisateur décide, de surcroît, de perdre le contrôle du nœud, il invoque alors la

procédure :

$$closeGui()$$

Enfin, le navigateur peut décider de supprimer l'entité du monde virtuel. Dans ce cas, dès l'exécution de la procédure, le nœud fermera toutes ses connexions et achèvera son exécution. Ceci peut être réalisé grâce à la procédure :

$$kill()$$

Récupération d'informations

Le navigateur peut, à n'importe quel instant, décider de récupérer certaines informations sur le monde virtuel. Il peut le faire en faisant exécuter par le nœud la procédure :

$$getInfos(idMedia, var, addVar)$$

Il existe différents traitements en fonction de la valeur de la variable *var*.

- si *var* est `ME`, alors la donnée $addVar$ est inopérante. L'exécution de cette procédure avec ces paramètres indique que le navigateur désire connaître les caractéristiques actuelles de l'entité ;
- quand *var* vaut `ADJ`, la valeur de $addVar$ est l'identifiant d'un voisin de l'entité. Le navigateur notifie sa volonté de récupérer les caractéristiques à propos du voisin identifié par $addVar$;
- la variable *var* peut également valoir `SER` avec $addVar$ décrivant l'identifiant d'un service. Dans ce cas, le nœud va indiquer au navigateur toutes les informations qu'il possède à propos du service considéré ;
- si *var* est `ALL`, alors la donnée $addVar$ est inopérante. Le navigateur indique qu'il désire récupérer toutes les informations du monde virtuel. Cela comprend aussi bien les informations sur l'entité que les caractéristiques des voisins et les services dont ils disposent.

Contrôle de l'entité

Quand un navigateur a pris le contrôle d'une entité, il peut modifier certaines de ses caractéristiques variables. Il le réalise avec l'appel d'une procédure :

$$modSelf(var, delta)$$

Quand *var* est égal à `POS`, la variable que le navigateur désire modifier est la position de l'entité. Dans ce cas, la variable *delta* est représentée par deux entiers séparés par une virgule correspondant au déplacement dans les deux coordonnées du monde virtuel.

Le navigateur peut également modifier l'orientation de l'entité en appelant la procédure avec *var* valant `ORI` et *delta* signifiant la nouvelle orientation de l'entité.

Une autre modification de l'entité peut consister en une *téléportation*, *i.e.* un déplacement dans une position lointaine. Cette action, qui nécessite de fermer toutes les connexions et de relancer l'algorithme de connexion du monde avec la nouvelle position est accessible par le biais de la procédure :

$$jump(pos_x, pos_y)$$

Les variables pos_x et pos_y représentent la nouvelle position de l'entité.

3.3.3 Procédures à disposition du Nœud

Lors de l'élaboration de Solipsis, nous nous sommes concentrés sur la communication entre les différents modules et sur le programme construisant le réseau de pairs. Il nous semble que les navigateurs sont des programmes qui ne doivent pas être figés car ils sont liés à l'évolution des technologies, que ce soit pour les aspects multimédia que pour l'interface entre l'homme et la machine.

Nous nous contentons donc de considérer qu'un navigateur possède une structure des données et un comportement qui lui permet de réagir aux événements que le nœud lui transmet par les procédures décrites dans la suite.

Informations sur l'entité

Le navigateur permet au nœud d'initialiser un certain nombre de variables en lui fournissant une procédure *init* :

$$init(id, pos_x, pos_y, ar, calibre, pseudo)$$

Les paramètres correspondent aux caractéristiques de l'entité. Dès la création de l'interface, cette procédure est immédiatement appelée par le nœud car elle permet au navigateur de déterminer l'entité contrôlée.

Il existe également des procédures destinées à notifier les événements virtuels relatifs à l'entité.

$$modSelf(var, deltaVar)$$

Le nœud peut agir sur deux caractéristiques de l'entité, la position et le rayon de connaissance. Le premier cas survient à chaque déplacement de l'entité. Un déplacement peut survenir dans le cadre de la gestion de collision, mais, plus traditionnellement, il fait suite à une initiative d'un navigateur. L'appel de cette fonction permet au nœud de confirmer que le déplacement a été validé. Le paramètre var vaut `POS` et $deltaVar$ correspond au déplacement.

Dans le cas d'une modification du rayon de connaissance, le paramètre var vaut `AR` et $deltaVar$ est la différence entre la nouvelle et l'ancienne valeur.

Informations sur le voisinage

Les procédures qui suivent permettent au nœud d'informer le navigateur des événements virtuels captés par l'entité.

Tout d'abord, le nœud peut notifier au navigateur un nouveau voisin en invoquant la procédure $newNode$:

$$newNode(id, pos_x, pos_y, calibre, ori, pseudo)$$

Les paramètres correspondent aux caractéristiques du voisin id. Ces caractéristiques peuvent varier. Pour indiquer un déplacement d'un voisin, le nœud appelle la procédure :

$$modNode(id, var, deltaVar)$$

Le paramètre var peut être `POS` ou `ORI`. La modification est indiquée dans le paramètre $deltaVar$.

Enfin, quand le nœud décide de fermer une connexion avec un voisin, il le signale au navigateur par :

$$deadNode(id)$$

3.4 La gestion des services

Nous avons vu qu'un navigateur peut posséder un certain nombre de services. Ces sous-programmes permettent au navigateur d'échanger des flux de données avec d'autres services selon un protocole dédié.

3.4.1 Notification d'un Service

Lors de la phase de connexion entre le navigateur et le nœud, le navigateur signale la présence de ses services en appelant la procédure *addService* du nœud :

$$service(idMedia, idService, descService, host, port)$$

Le premier paramètre est l'identifiant du navigateur possédant le service dont l'identifiant est précisé par *idService*. Pour que deux services puissent déterminer les protocoles en commun, nous avons choisi d'ajouter un paramètre *descService* qui est une description des fonctionnalités du service *idService*. Il peut exister autant de description de services que de services eux-mêmes. Ce paramètre pourrait, par exemple, être l'identifiant du protocole utilisé par ce service. Enfin, ce service dispose d'une adresse Internet propre qui est précisée par les paramètres *host* et *port*.

A n'importe quel instant, un navigateur peut choisir de désactiver un service. Il signale alors au nœud que ce service est inopérant par la procédure :

$$closeService(idMedia, idService)$$

Les paramètres permettent au nœud de déterminer le service désactivé.

3.4.2 Diffusion de l'information

Les nœuds sont chargés de diffuser l'information concernant leurs services. Deux entités qui se rencontrent échangent leurs caractéristiques comme détaillé dans la Section 3.2. Lors de cette phase de connexion, ils échangent également les services dont ils disposent ainsi que les paramètres réseaux qui vont permettre à d'autres services de se mettre immédiatement en relation avec ceux-ci.

Les messages échangés sont alors de la forme :

$$< service, id, idService, descService, host, port >$$

Par ce message, l'entité *id* signale à son voisin qu'il possède un service *idService* décrit par *descService* possédant une adresse Internet *host* : *port*.

Le nœud envoie ce message dès qu'il rencontre une nouvelle entité. Il peut également décider d'envoyer ce message à tous ses voisins quand la fonction *addService* est appelée par le navigateur.

Inversement, dès que le navigateur signale la fermeture d'un service, le nœud retransmet l'information à ses voisins en émettant un message :

$$< endService, id, idService >$$

Ainsi, à chaque instant, une entité connaît tous les services possédés par ses voisins.

3.4.3 Mise en relation

La connaissance des services des voisins permet à un nœud de mettre en relation deux services compatibles.

Dès réception d'un message $< service >$, un nœud vérifie que la description de ce service n'est pas identique aux descriptions des services qu'il possède. Quand il découvre qu'un service s_1 d'un de ses voisins pourrait communiquer avec un des services s_2 appartenant au navigateur n_1, il informe immédiatement n_1 en appelant la procédure :

$$service(id, idService, descService, host, port)$$

Les paramètres de cette procédure sont l'identifiant de l'entité possédant s_1, l'identifiant de s_1, sa description et son adresse Internet.

Le navigateur peut alors décider d'ouvrir une connexion entre s_1 et s_2 pour les échanges de flux de données.

Un mécanisme similaire permet au nœud d'informer un navigateur de la fermeture d'un service par la procédure :

$$closeService(id, idService)$$

3.5 Conclusion et travaux futurs

Dans ce chapitre, nous avons détaillé le fonctionnement d'une entité connectée au monde Solipsis. Nous avons présenté l'architecture générale du programme et les moyens de communication entre les différentes parties de ce programme. Notre volonté est de proposer une architecture flexible et indépendante d'un quelconque langage de programmation ou d'un système d'exploitation. Ainsi, nous pouvons espérer que de multiples versions différentes du nœud puissent collaborer, tant qu'elles sont conformes au protocole que nous avons décrit. De la même manière, seule l'imagination peut restreindre le nombre de navigateurs et de services qui peuvent se connecter à un nœud.

Nous avons précisé, à chaque fois que cela était nécessaire, les choix que nous avons fait et les raisons qui nous ont poussé à faire ces choix. Quand c'était possible, nous avons également proposé un certain nombre d'alternatives à ces choix. Il est évident que nous avons peut-être omis des alternatives qui présenteraient des avantages décisifs. Un travail futur consisterait à réaliser une analyse comparative complète pour chacun de ses problèmes afin de valider, ou infirmer, les choix que nous avons pu faire.

D'autres problèmes passionnants pourraient faire l'objet d'une étude et de nouveaux algorithmes pourraient sans doute rendre le programme plus efficace. Par exemple, une entité se déplaçant à une vitesse constante, doit, dans la version que nous avons présentée, déclarer son déplacement à chaque instant. Dans d'autres mondes virtuels, un mécanisme

appelé *dead-reckoning* est mis en place pour gérer ce type de déplacement en minimisant le nombre de messages échangés. Dans le cadre de Solipsis, le problème devient plus ardu, du fait de l'absence d'autorité centrale. Néanmoins, une solution pourrait être envisagée, testée et, éventuellement, validée.

Dans cette première partie, nous nous sommes attachés à présenter Solipsis, un système de réalité virtuelle basé sur un réseau de pairs. Les principes généraux de Solipsis permettent la cohabitation d'un nombre illimité de participants simultanés.

Solipsis n'est pas seulement un ensemble de documentation, mais aussi un programme disponible sur Internet [55]. Pour réaliser ce programme, nous avons dû réaliser quelques choix.

Le langage de programmation que nous avons utilisé est le Python [57]. Ce langage nous a séduits sur plusieurs points. Tout d'abord, on peut noter que Python est un langage qui fonctionne sur de nombreux systèmes d'exploitation. Ensuite, c'est un langage gratuit et ouvert. Puis, sa syntaxe est simple et, combinée à des types de données évolués, il conduit à des programmes lisibles et compacts. Enfin, Python est un langage qui continue à évoluer.

Nous avons, ensuite, opté pour le protocole UDP pour la communication entre les nœuds. Il nous semble que ce protocole est particulièrement adapté aux réseaux de pairs dans lesquels les défaillances et les coupures de connexion peuvent survenir à n'importe quel instant. De plus, il autorise chaque élément à envoyer un message à n'importe quel élément sans phase d'initialisation.

Enfin, nous avons choisi d'utiliser XML-RPC pour l'interface de commande entre le nœud et le navigateur. Ce choix se fait par défaut, puisque les alternatives nous ont paru plus coûteuses ou dépendantes d'un protocole propriétaire.

Le projet Solipsis a fait l'objet de deux articles scientifiques [47, 68], d'une présentation à un colloque regroupant des programmeurs [69] et d'un brevet. Depuis que Solipsis est accessible sur Internet, plusieurs centaines d'internautes se sont intéressés au programme, ainsi qu'à la documentation du protocole. En outre, une communauté de programmeurs intéressés par Solipsis est née et des améliorations du programme, notamment l'aspect graphique du navigateur, ont déjà été proposées.

Cela nous permet de croire que le projet continuera à évoluer et que les fondations que nous avons pu creuser au cours de cette thèse permettront l'érection d'un monde virtuel partagé par plusieurs millions (milliards) d'utilisateurs.

Deuxième partie

Auto-organisation de systèmes mobiles distribués

Dans cette seconde partie, nous nous sommes intéressés à deux problèmes qui pourraient constituer des points de blocage au moment de concevoir des services dédiés aux utilisateurs de Solipsis. Le premier est un service de recherche. Il vise à offrir à un utilisateur le moyen de communiquer avec l'entité la plus proche de lui répondant à un critère donné. Le second permet de diffuser une information uniquement à certaines entités qui se sont déclarées intéressées par ce type d'information.

Bien entendu, notre volonté d'intégrer ces services à Solipsis nous a contraint à refuser toute solution basée sur une autorité centrale. Les algorithmes que nous proposons sont destinés à être intégrés à Solipsis et, en conséquence, ils ne doivent pas perturber la capacité de passer à l'échelle de notre système. De plus, ils doivent pouvoir fonctionner dans un système mobile dans lequel les participants peuvent, à chaque instant, disparaître et apparaître.

Dans [47], nous avons proposé l'association des communautés de recherche étudiant les réseaux de pairs, les réseaux ad-hoc et les environnements virtuels partagés. Afin d'appuyer nos arguments, nous avons décidé d'appliquer les deux services envisagés dans le cadre d'un réseau mobile distribué différent. Ainsi, nous proposons dans la suite un service de localisation pour un réseau ad-hoc et un service de publication et abonnement pour un réseau de pairs logique.

Les deux chapitres de cette partie sont organisés de la même manière. Dans un premier temps, nous présentons sommairement les caractéristiques du système distribué étudié. Puis, nous décrivons le problème auquel nous sommes confrontés. Enfin, nous détaillons la solution que nous proposons.

Chapitre 4

Service de recherche

4.1 Introduction

4.1.1 Les réseaux ad-hoc

Un réseau mobile ad-hoc est un réseau d'ordinateurs mobiles qui, comme le nom le suggère, se constitue selon un processus ad-hoc. Les participants à ce réseau disposent de la capacité de communiquer par voie hertzienne avec d'autres participants situés à proximité (physiquement). Quelques uns peuvent, occasionnellement, se porter volontaires pour retransmettre un message reçu, et ainsi agir comme un routeur. Ce comportement permet la formation d'un réseau qui ne possède pas d'infrastructure prédéterminée. Le réseau est continuellement en mutation et les routeurs sont choisis à la volée. En quelque sorte, les fonctionnalités de ce réseau sont offertes par un mécanisme ad-hoc.

Il est probable que les réseaux ad-hoc vont devenir un important médium de communication dans un futur proche. La technologie sur laquelle ils reposent (cartes réseaux, protocoles...) est maintenant incontournable dans les ordinateurs portables. De plus, les utilisateurs sont attirés par la perspective de profiter d'une bande-passante très importante gratuitement puisque l'utilisation des antennes hertziennes n'admet aucun coût financier. Dans la pratique, tant que les communications demeurent dans le réseau ad-hoc, l'utilisation du réseau est totalement libre.

Dans un premier temps, la communauté scientifique s'est intéressée aux stratégies de routage pour de tels réseaux. Il s'agit de permettre à deux entités de communiquer, malgré l'absence d'organisation du réseau. Brièvement, il existe trois types de stratégies qui ont conduit aux protocoles de routage *proactifs*, *réactifs* et *hybrides*. Les protocoles proactifs tels que OLSR [31] supposent que le voisinage d'une entité est connue à l'avance, par exemple grâce à un échange régulier de messages de signalisation. Ainsi, les entités construisent et maintiennent une table de routage. Celle-ci leur permet de connaître le chemin dans le graphe de communication avec n'importe quelle autre entité. Seules les entités sur ce chemin doivent agir comme routeurs. Au contraire, les protocoles réactifs comme AODV [81] et DSR [64] découvrent le chemin quand il est demandé. Il n'y a pas d'utilisation de ressources pour conserver une trace des routes. En revanche, le temps de réponse peut être important et la découverte des chemins les plus courts est rarement efficace. Les protocoles hybrides tels que ZRP [52] ont un comportement proactif dans le voisinage des entités jusqu'à k hops de distance. Puis, pour des communications plus

lointaines, le protocole adopte un comportement réactif classique.

La diffusion totale, connue par le terme anglais *broadcast*, constitue un autre défi difficile dans les réseaux ad-hoc. Dans [103], les auteurs analysent plusieurs techniques pour la diffusion totale : l'inondation simple [54], une méthode basée sur les probabilités [78], une stratégie utilisant la localisation des entités [78], un algorithme se concentrant sur le passage à l'échelle [79] et un protocole spécifique au réseau ad-hoc [80]. Les auteurs de [103] proposent une série de simulations mesurant les performances telles que le pourcentage d'entités recevant le message, le nombre d'entités participants à l'algorithme, le délai et le nombre total de paquets transmis par entité et par diffusion.

Un troisième axe de recherche fondamentale dans les réseaux ad-hoc concerne la contrôle de la topologie. Ces études se basent sur l'hypothèse que chaque entité possède la capacité de faire varier la puissance d'émission de ses messages. De nos jours, la majorité des antennes hertziennes possèdent cette fonctionnalité. A chaque émission de message, l'entité consomme de l'énergie. Or, l'énergie est une ressource rare. En diminuant la puissance d'émission du message, il est possible de réduire la consommation d'énergie et, ainsi, de rallonger la durée de vie de l'entité participant au réseau. Mais, cette réduction a un impact fort sur la topologie du réseau puisque les liens entre les entités dépendent de cette puissance d'émission. La réduction de la puissance d'émission s'accompagne d'une diminution du nombre de liens dans le graphe de communication, ce qui peut provoquer le partitionnement du graphe. Un algorithme de contrôle de topologie vise à déterminer, pour chaque entité du réseau, la puissance d'émission qu'elle doit utilisée, de manière à ce que le graphe de communication possède les propriétés désirées et, dans le même temps, que la réduction de l'énergie nécessaire au bon fonctionnement du réseau soit maximisée. Comme décrit dans le chapitre 2, les travaux issus de la Géométrie Calculatoire ont inspiré de nombreuses propositions [73, 21, 66, 74]. Il est également possible de construire une topologie en restreignant le nombre de voisins de chaque entité à un paramètre k fixé [19, 36, 20, 85]. Les recherches dans ce domaine sont très actives, comme le prouvent certaines publications récentes [102, 23].

4.1.2 Service de recherche

Nous sommes intéressés par offrir aux utilisateurs de Solipsis un service de recherche d'entités dont le principe général est décrit ci-dessous. Un utilisateur peut, à chaque instant, désirer communiquer avec une autre entité, parfois distante, répondant à un certain nombre de critères. L'idée est de permettre à une entité de lancer une recherche afin de découvrir l'entité la plus proche d'elle qui vérifie un prédicat donné. Le service de recherche se compose donc en premier lieu d'un outil permettant de définir un prédicat basé sur des critères communs à toutes les entités. Nous considérons dans la suite que cet outil, lié à une interface entre l'utilisateur et le système, existe et que chaque entité est capable de déterminer si elle vérifie ce prédicat. Ensuite, nous admettons qu'une nouvelle interface entre le système et l'utilisateur permet à ce dernier de lancer sa recherche et d'afficher le résultat obtenu. Ce résultat peut consister, selon le choix de la mise en œuvre, en l'identifiant de l'entité découverte, sa position ou, encore, en la mise en communication directe avec cette entité selon un médium fixé.

Un tel service de recherche pourrait avoir beaucoup d'utilités dans le cadre d'un réseau

ad-hoc. La principale application que nous avons identifiée concerne un mécanisme de copie temporaire de fichiers, autrement appelé *stratégie de cache*. Un stratégie de cache est une technique utilisée dans de nombreux domaines de l'informatique pour améliorer les performances et la qualité de service quand il existe une notion de localisation. Imaginons par exemple que plusieurs utilisateurs proches désirent accéder à une même donnée. Il suffit que l'un des utilisateurs accède à cette donnée puis qu'il la sauvegarde pour que les autres requêtes pour cette donnée puissent être traitées par cet utilisateur. Il se substitue alors au possesseur de cette donnée, souvent un serveur. Les deux avantages majeurs sont la réduction du nombre de requêtes traitées par le serveur, et une réduction du trafic global, due à la proximité des copies de la donnée. Le travail présenté dans [42] donne un argumentaire pour l'utilisation de cache dans les réseaux ad-hoc.

La mise au point d'un mécanisme de cache dans les réseaux ad-hoc est fondamentalement différente de semblables expériences dans les réseaux filaires. La mobilité des entités apporte de fréquentes déconnections et la topologie du réseau évolue continuellement. De plus, l'absence d'organisation hiérarchique rend impossible l'utilisation des techniques les plus éprouvées sur Internet, comme par exemple [7, 22, 29, 38, 76, 98, 109].

L'utilisation de la technique du cache sur le Web a dévoilé deux principaux problèmes : le placement des copies des données et la recherche de la copie la plus proche. Le premier consiste à sélectionner les entités qui vont conserver une copie d'une donnée particulière. Naturellement, optimiser le placement des copies des données impacte fortement sur les performances du mécanisme mis en place [42]. Par exemple, le nombre de copies pour chaque donnée dans un réseau ad-hoc doit être ajusté de manière à minimiser les communications en dehors du réseau ad-hoc, à minimiser les communications dans le réseau et à maximiser le nombre de données qui peuvent être accessibles directement dans le réseau. De plus, dans le cadre de réseaux mobiles, une solution peut être optimale à un instant donné et médiocre quelques instants plus tard. Il est probable que ce problème fera l'objet d'études approfondies dans un futur proche.

Le second problème identifié concerne la *recherche des copies d'une donnée dans les réseaux ad-hoc*. Il est similaire au service de recherche que nous désirons mettre en œuvre dans Solipsis. Les solutions que nous cherchons à obtenir doivent être complètement décentralisées, efficaces en terme de coût de communications et aussi insensibles que possible à la mobilité des participants. Nous avons donc rapidement écarté les mécanismes basés sur le maintien de bases de données pour les mêmes raisons que lors de l'élaboration de Solipsis.

Certains travaux ne sont pas directement liés au problème que nous avons identifié, mais il est possible de s'en inspirer largement. Ainsi, la découverte de la topologie est un service proche de la recherche dans les réseaux ad-hoc. Le principal problème consiste ici à collecter toutes les informations relatives à la topologie et à distribuer ces informations à toutes les entités du réseau. Des techniques telles que celle présentée dans [88] utilise des agents mobiles pour la diffusion de l'information et des notions de stabilité des liens pour prédire la topologie actuelle du réseau.

Enfin, les récents développements des logiciels basés sur des réseaux de pairs pourraient être adaptés dans les réseaux ad-hoc. En particulier, les logiciels existants comme Pastry [87], Tapestry [110] ou Chord [92] sont décentralisés et efficaces en terme de besoins de communication. Malheureusement, ils ne réagissent pas très bien aux modifications

fréquentes de la topologie du réseau et de l'attribution de données aux entités. Pourtant, certains concepts comme le regroupement des entités en fonction de leur similarité pourrait être appliqué dans les réseaux ad-hoc. Ainsi, dans [89], les auteurs proposent une étude d'une stratégie de cache mêlant les réseaux ad-hoc et une approche par réseaux de pairs. L'algorithme de recherche est hybride : les entités dans une même zone sont contactés par une diffusion totale tandis que les recherches au-delà de cette zone est assurée par un algorithme basé sur un réseau de pairs. L'étude expérimentale présentée s'intéresse particulièrement à la consommation d'énergie.

4.1.3 Contributions

Les travaux que nous avons effectués sur ce sujet sont détaillés dans [43]. Dans un premier temps, nous avons cherché à offrir une définition formelle du service de recherche que nous désirons mettre en œuvre. La partie 4.2 décrit le modèle du réseau et les principales définitions utilisées dans ce chapitre.

Un service de recherche nécessite la création d'une requête et sa dissémination dans le réseau. Nous avons exploré plusieurs techniques de dissémination de requêtes, incluant l'inondation, l'inondation dans un sous-graphe du réseau et une recherche en largeur. De plus, nous avons étendu une technique basée sur un calcul de probabilité présentée dans [78, 96]. Ces quatre mécanismes, décrits dans la partie 4.3, ont été choisis car ils représentent les techniques de recherche distribuée les plus courantes.

Parallèlement, nous proposons plusieurs techniques permettant de réduire le nombre d'entités assurant la dissémination des requêtes. L'idée consiste à assigner à un sous-ensemble des entités la tâche de retransmettre les messages de requête. On distingue donc deux types d'entités : celles qui n'agissent pas dans la tâche de dissémination et celles qui assurent la fonctionnalité de retransmission. Ces dernières forment ce qu'on appelle une *super-couche*, ou en anglais *overlay*.

La construction d'une super-couche n'est pas aisée. Elle doit contenir peu d'éléments, mais garantir que la diffusion des messages couvre totalement le réseau ad-hoc physique. Parmi les super-couches étudiés, nous notons l'*Ensemble Indépendant Minimal* (MIS) [10], l'*Ensemble Dominant Connexe* (DS) [51] et une nouvelle construction basée sur les arbres de diffusion pour la recherche en largeur. Les algorithmes que nous proposons sont décentralisés et se reconfigurent lors de défaillances ou d'événements dus à la mobilité [35].

Nous avons comparé plusieurs combinaisons de ces techniques de dissémination avec les super-couches proposées grâce à un ensemble de simulations. Nous avons cherché à mesurer le pourcentage de requêtes satisfaites, le temps moyen pour l'obtention de ces réponses et le nombre de messages que ces requêtes engendrent. Pour cela, nous avons fait varier plusieurs paramètres du système tels que la distance maximale de transmission de données, la vitesse des entités, la densité d'entités répondant à la requête et une variable permettant de faire varier la densité d'entités de manière réaliste. Le pourcentage de requêtes satisfaites est une indication de la probabilité qu'une réponse soit produite, si cela est possible. Le temps moyen d'obtention de réponse est un des principaux objectifs de la mise en place de mécanisme de cache. Enfin, le nombre de messages engendrés par une méthode mesure le coût imposé en terme de ressources (bande-passante, énergie...), ce qui impacte sur la

capacité de passage à l'échelle.

Les résultats de ces simulations sont détaillés dans la partie 4.5. Ils mettent en lumière les qualités des différentes méthodes et ouvrent des voies pour le développement de techniques de cache pour les réseaux ad-hoc, ainsi, bien entendu, que des pistes intéressantes pour la mise en œuvre de tels algorithmes pour Solipsis.

4.2 Modèle et définitions

Pour ce travail, nous nous situons dans le contexte des systèmes mobiles sans-fil. Une entité est un ordinateur possédant une antenne omnidirectionnelle permettant la communication sans-fil. Un message transmis par une entité p est reçu par toutes les entités situées dans le disque centré en p et dont le rayon dépend de la puissance de transmission. Nous appellerons ce disque le *disque de transmission* tandis que le rayon correspond à la *portée de transmission*. Nous considérons donc une seule primitive de communication `send(m)` permettant à une entité d'envoyer le message m à toutes les entités situées dans son disque de transmission. La combinaison des entités et la fermeture transitive de leurs disques de transmission forment un réseau ad-hoc sans-fil.

Comme dans les chapitres précédents, le réseau que nous décrivons peut être modélisé par un graphe $G = (V, E)$ avec V l'ensemble des entités et E les relations de voisinage. Une entité q est la voisine d'une entité p si q est située dans le disque de transmission de p. Dans la suite, $\mathcal{N}(p)$ représente l'ensemble des voisines de l'entité p.

Les entités sont des objets qui peuvent se déplacer dans un plan. Il n'existe donc aucune garantie qu'un voisin à l'instant t soit encore dans le disque de transmission à l'instant $t + \Delta$. Nous ne pouvons pas garantir non plus que les messages seront effectivement reçus mais nous supposons qu'ils le seront avec une grande probabilité. De plus, les ordinateurs peuvent être éteints et allumés à chaque instant. L'ensemble V des entités peut donc varier dans le temps et n'a pas une taille fixe.

Certaines entités vérifient à un instant donné un prédicat φ. On dit qu'une entité est une *cible* pour le prédicat φ à un instant t lorsqu'elle vérifie le prédicat φ à t.

Enfin, nous considérons une structure abstraite nommée *super-couche* consistant en une collection d'entités appartenant à V. Les entités appartenant à la super-couche sont appelés des *entités élues*. Une partie du travail présenté ici explore des protocoles d'élection de ces entités.

4.2.1 Service de recherche

Les caractéristiques des entités permettent à un ensemble $V^t(\varphi) \subseteq V$ de vérifier le prédicat φ à un instant t. Pour une entité p donnée, nous définissons $V_p^t(\varphi) \subseteq V^t(\varphi)$ comme étant l'ensemble des entités $q \in V^t(\varphi)$ telles qu'il existe un chemin entre p et q à l'instant t. Pour une entité p à la recherche d'une cible pour le prédicat φ, chaque entité de $V_p^t(\varphi)$ est une *alternative*.

Un service de recherche est initialisé par une requête, caractérisée par un *demandeur* $p \in V$, un prédicat φ et l'instant t auquel la requête est émise. En réponse à une requête, le service de recherche est supposé générer une *réponse*, qui peut consister en au moins une alternative, si elle existe, ou $\perp$.

Le comportement désiré pour le service de recherche peut être restreint à deux propriétés : la *validité* et l'*efficacité*. La validité indique que les résultats du service de recherche ne sont pas des "mensonges" tandis que l'efficacité spécifie la capacité du service à produire un résultat. De plus, comme nous aimerions limiter le temps passé pour une recherche, nous proposons deux définitions de l'efficacité : l'efficacité temporelle et l'efficacité ultérieure.

Validité Si la réponse à une requête (p, φ, t) est générée à l'instant t' et inclut l'entité q, alors il existe un instant t'' tel que $t \leq t'' \leq t'$ et $q \in V_p^{t''}(\varphi)$.

Efficacité ultérieure Pour toute requête (p, φ, t) pour laquelle il existe une alternative à un instant $t' \geq t$, le service génère une réponse contenant une alternative.

Efficacité temporelle Il existe une durée T telle que, quelle que soit la requête (p, φ, t) pour laquelle il existe une alternative à l'instant t' avec $t \leq t' \leq t + T$, le service génère une réponse contenant une alternative à un instant t'' avec $t \leq t'' \leq t + T$.

Les entités peuvent se déplacer et leurs caractéristiques peuvent, à n'importe quel instant, être modifiées. De fait, la validité assure seulement qu'une cible existait au moment où la réponse a été générée, mais il est possible que cette alternative ne soit plus existante lorsque la réponse parvient au demandeur.

De même, assurer l'efficacité est coûteux et impossible dans la pratique. Nous proposons donc une version probabiliste des propriétés.

Efficacité ultérieure approximative Pour toute requête (p, φ, t) pour laquelle il existe une alternative à un instant $t' \geq t$, le service génère une réponse contenant une alternative avec une grande probabilité.

Efficacité temporelle approximative Il existe une durée T telle que, quelle que soit la requête (p, φ, t) pour laquelle il existe une alternative à l'instant t' avec $t \leq t' \leq t + T$, le service génère une réponse contenant une alternative à un instant t'' avec $t \leq t'' \leq t + T$ avec une grande probabilité.

4.3 Technique de dissémination de requêtes

4.3.1 Inondation contrainte

La technique de l'inondation est le moyen le plus simple de diffuser une requête dans un réseau [70]. Le demandeur envoie un message *search* à tous ses voisins dans le réseau. Considérons une entité qui reçoit ce message pour la première fois. Si elle ne vérifie pas le prédicat contenu dans le message, elle retransmet la requête à tous ses voisines et inscrit le chemin suivi par la requête pour arriver jusqu'à elle. Si une entité vérifie le prédicat, elle envoie immédiatement un message de réponse qui prendra le chemin indiqué dans la requête. Il est possible que l'entité reçoive le même message *search* plusieurs fois. Ces copies du même message, appelés messages redondants, ne sont pas traités par l'entité.

Quand il existe une structure de type super-couche, seules les entités élues retransmettent le message. Les autres entités se contentent de répondre si elles vérifient le prédicat. Ce type d'inondation utilisant une super-couche est appelé *inondation contrainte*.

L'Algorithme 7 décrit l'algorithme d'inondation. Nous n'avons pas fait figurer le traitement réalisé pour les messages de réponse.

Parmi les avantages de l'inondation, on peut noter que le temps de réponse est minimal. C'est un mécanisme très robuste qui a la plus grande probabilité d'atteindre toutes les entités du réseau et, par conséquent, de trouver une alternative si elle existe. Malheureusement, cette méthode génère un très grand nombre de messages. Dans notre cas, cela signifie que la consommation d'énergie va être très importante. Enfin, les performances globales du système peuvent être altérées par le grand risque de collisions des messages, *i.e.* plusieurs messages qui arrivent simultanément à une entité.

Il existe deux manières de réduire les désagréments de l'inondation. La première consiste à restreindre la diffusion par un paramètre "*Temps avant de mourir*" (TTL), qui spécifie le nombre de fois qu'une requête peut être retransmise. Cette technique permet de résoudre le problème du passage à l'échelle [5], mais elle perd en efficacité. La seconde méthode repose sur les super-couches. Le nombre de messages est réduit car seules les entités élues envoient les messages mais cela impose de pouvoir être en mesure de construire une super-couche qui garantit que *toutes* les entités du réseau reçoivent, au moins une fois, la requête. De plus, sa construction et son maintien peuvent nécessiter une importante consommation de ressources. Il ne faut pas que les gains apportés par la super-couche soient inférieurs aux coûts de sa mise en place, particulièrement dans les réseaux ad-hoc soumis à une mobilité intense.

Algorithme 7 un service de recherche basé sur l'inondation (entité i)

Paramètres d'entrée :
$estElu()$: prédicat vrai si i appartient à la super-couche ;
$enregistre(m)$: conserve une trace du passage du message m ;
$estEnregistre(m)$: prédicat vrai si le message m a déjà été traité ;

Actions :
$\mathcal{R}_1$: l'entité i initie une recherche
envoie $< req, i, \varphi, \emptyset >$ à tous les voisins
réceptionne $< resp, j, presence_j, \mathcal{P} >$ pendant un délai ou jusqu'à $presence_j =$ `true`
si réception d'un message $< resp, j, \texttt{true}, \mathcal{P} >$ **alors**
retourne l'entité j
sinon
retourne $\perp$

$\mathcal{R}_2$: à la réception d'un message $< req, j, \varphi, \mathcal{P} >$
si non $estEnregistre(< req, j, \varphi, \mathcal{P} >)$ **alors**
$enregistre(< req, j, \varphi, \mathcal{P} >)$
si φ est vérifié **alors**
envoie $< resp, i, \texttt{true}, \mathcal{P} \cup \{i\} >$
sinon
si $estElu()$ **alors**
envoie $< req, j, \varphi, \mathcal{P} \cup \{i\} >$

4.3.2 Inondation probabiliste

L'inondation probabiliste est similaire à l'inondation, hormis le fait que les entités retransmettent les messages entrants de façon probabiliste. La décision de retransmission de la requête est prise en fonction du résultat d'une fonction qui retourne `vrai` selon une probabilité p.

Dans un réseau dense, quand une entité décide de retransmettre un message, la plupart de ses voisines a de grandes chances d'avoir déjà reçu ce message. Décider selon un calcul de probabilité de ne pas retransmettre le message est susceptible d'éviter que des messages redondants soient envoyés, sans nuire à l'efficacité de l'algorithme. Au contraire, dans les réseaux faiblement peuplés, certaines entités peuvent ne pas recevoir tous les messages si la probabilité est trop faible.

Une solution consiste donc à ajuster la probabilité p au nombre de voisines. Une autre idée se base sur le nombre de messages redondants qu'une entité a pu recevoir [78, 96]. Ainsi, chaque entité compte le nombre de messages redondants qu'elle reçoit durant un intervalle de temps fixé à partir de la réception du premier message. A la fin de cette durée, si le nombre de messages redondants est inférieur à un seuil fixé, le message est retransmis. Avec cette technique, certaines entités ne vont pas retransmettre la requête dans les zones fortement peuplées tandis que toutes les entités vont le retransmettre dans les zones plus clairsemées. Malheureusement, cette solution produit une augmentation du temps de latence puisqu'à chaque hop, il est nécessaire d'attendre l'écoulement de l'intervalle temporel.

Nous proposons une nouvelle technique qui consiste à déterminer le paramètre de probabilité p en se basant sur le nombre moyen de messages redondants mesuré sur une période temporelle plus longue. Nous notons $\overline{r}$ le nombre moyen de messages redondants reçus sur une période de temps fixée, R le nombre idéal de messages redondants que chaque entité désire recevoir et p le paramètre de probabilité, initialisé à une valeur fixée.

A chaque réception de message, chaque entité calcule son paramètre de probabilité $p \leftarrow (1+d) * p$ avec $d = k * (R - \overline{r})$. Dans cette fonction, k est un paramètre qui permet une évolution rapide ou lente du paramètre de probabilité. Si $\overline{r}$ est inférieur à R, cela signifie que l'entité ne reçoit pas suffisamment de messages redondants. Dans ce cas, $R - \overline{r}$ est positif et p croît. Dans le cas contraire, $R - \overline{r}$ est négatif et, naturellement, p décroît.

Dans nos expérimentations (cf Section 4.5), nous avons fixé $R = 5$ et $k = 0.05$. La valeur de R est basée sur le travail présenté dans [78, 96], dans lequel les auteurs étudient la valeur optimale de ce paramètre. La valeur de k a été fixé après plusieurs expérimentations.

Le principal défaut de cet algorithme vient du calcul du paramètre de probabilité. Ce calcul dépend du nombre de messages circulant dans le réseau, et donc du nombre de requêtes émises. De fait, cet algorithme est efficace quand les entités produisent beaucoup de requêtes. Dans le cas contraire, la qualité du calcul est pauvre et cet algorithme n'assure pas l'efficacité du service.

4.3.3 Recherche en largeur

Un autre mécanisme consiste en une recherche en largeur (BFS) [33]. Cet algorithme peut être vu comme une série d'instantiations successives de l'inondation avec un périmètre de retransmission (TTL) croissant de 1 jusqu'au diamètre du graphe. Plus spécifiquement,

un demandeur $p \in V$ initie tout d'abord la recherche en envoyant un message à tous ses voisins $\mathcal{N}(p)$. A la réception de cette requête, l'entité répond par un message positif si elle vérifie le prédicat ou par un message négatif sinon. Si le demandeur ne reçoit aucune réponse positive, il émet une nouvelle requête à toutes les entités de $\mathcal{N}^2(p)$. Toutes les entités qui n'ont pas reçu le message durant la première phase utilisent le chemin inverse pour faire parvenir au demandeur leur réponse. Ce mécanisme agît de manière récursive jusqu'au moment où (1) le demandeur reçoit une réponse positive ou (2) il pense que sa requête a été reçue par toutes les entités du réseau.

Dans ce protocole, la propagation du message est contrôlée par le demandeur. Dès qu'une alternative est détectée, la propagation s'arrête, ce qui permet d'éviter de nombreux message inutiles, particulièrement quand l'alternative est à proximité du demandeur. A noter que cet algorithme peut être appliqué à tout type de réseau, mais aussi à un réseau disposant d'une super-couche. Dans ce cas, seules les entités élues retransmettent les requêtes, tout en préservant le TTL de l'itération.

Le principal défaut de cette technique est le risque de temps de latence très important. En effet, une nouvelle requête est envoyée aux entités à distance i après la fin de l'expiration du délai d'attente permettant à toutes les entités à distance $i-1$ d'envoyer leurs réponses. De plus, cet algorithme peut générer de nombreux messages quand la cible est loin du demandeur car plusieurs requêtes et de nombreuses réponses sont nécessaires avant de détecter une alternative.

L'Algorithme 8 décrit l'algorithme du BFS. Le service de recherche opère récursivement. A chaque itération i, le demandeur initie une requête qui doit être propagée au plus à i hops de lui et, ensuite, il attend les réponses. Si, à l'expiration d'un délai d'attente, il n'a reçu aucune réponse positive, il entre dans une nouvelle itération du programme. Chaque entité recevant une requête vérifie d'abord qu'elle ne possède pas une copie de la donnée. Si elle est effectivement une cible pour ce prédicat, elle émet immédiatement une réponse positive. Dans le cas contraire, elle retransmet le message, si elle fait partie d'une éventuelle super-couche et si elle se situe à moins de i hops du demandeur. Dans les autres cas, le message est ignoré.

4.4 Maintien et construction d'une super-couche

4.4.1 Ensemble indépendant minimal

Tout d'abord, nous donnons une définition formelle d'un ensemble maximal indépendant (MIS), puis, nous expliquons comment construire une super-couche basée sur un MIS.

Soit $G = (V, E)$ un graphe de communication. Deux entités i et j dans G sont *indépendantes* si $(i, j) \notin E$. Un sous-ensemble $S \subseteq V$ d'entités est indépendant si chaque paire d'entités dans S est indépendante. Un ensemble S est un *ensemble indépendant maximal* si S est indépendant et, quelle que soit l'entité $k \in V \setminus S$, l'ensemble $S \cup \{k\}$ n'est pas indépendant.

La construction de la super-couche basée sur un MIS est réalisée par deux tâches exécutées en parallèle. Dans la première phase, le MIS est calculé. Par définition, les entités du MIS ne peuvent pas communiquer directement les unes avec les autres. La seconde phase cherche justement à identifier des "*ponts*" entre les entités du MIS. Naturellement, le but

Algorithme 8 un service de recherche basé sur une recherche en largeur (entité i)

Paramètres d'entrée :
$\mathcal{N}(i)$: ensemble des voisins de i ;
$estElu(i)$: prédicat `vrai` si i appartient à la super-couche ;

Actions :
$\mathcal{R}_1$: l'entité i initie une recherche
$round = 1$
répète
envoie $< req, i, \varphi, round, 1, \emptyset >$ à tous les voisins
réceptionne $< resp, j, presence_j, \mathcal{P} >$ pour tout $j \in \mathcal{N}(i)$ ou pendant un délai
si réception d'au moins un message $< resp, j, \texttt{true}, \mathcal{P} >$ **alors**
retourne une entité j parmi les réponses positives
si aucune réponse reçue
retourne $\perp$
$round = round + 1$
fin de la boucle répète

$\mathcal{R}_2$: à la réception d'un message $< req, j, \varphi, round, dist, \mathcal{P} >$
si $dist = round$ et $i \notin \mathcal{P}$ **alors**
si φ est vérifié **alors**
envoie $< resp, i, \texttt{true}, \mathcal{P} \cup \{i\} >$
sinon
envoie $< resp, i, \texttt{false}, \mathcal{P} \cup \{i\} >$
sinon
si $estElu(i)$ **alors**
envoie $< req, j, \varphi, round, dist + 1, \mathcal{P} \cup \{i\} >$

est de trouver aussi peu de ponts que possible et, bien évidemment, de réaliser ces deux tâches selon un mécanisme totalement décentralisé.

Nous sommes donc intéressés par un algorithme qui permette la construction d'un MIS en se basant uniquement sur la connaissance que chaque entité possède de ses voisines. De plus, nous aimerions influencer la construction de la super-couche de telle manière que les membres de cette super-couche soient les meilleurs possibles selon une certaine métrique. Par exemple, en partant de la constatation que les systèmes mobiles sont souvent contraints par les batteries des participants, nous pourrions utiliser l'énergie comme métrique, de manière à former une super-couche constituée des entités qui possèdent le plus d'énergie. Mais, nous pourrions également utiliser le nombre de données qu'une entité possède pour réduire le temps de latence, ou encore la bande-passante disponible, la portée de transmission ou la capacité de stockage, ainsi, bien évidemment, que n'importe quelle combinaison de ces critères.

Nous définissons donc une fonction générique qui associe à chaque entité une valeur d'un ensemble ordonné. Cette valeur mesure l'intérêt pour le réseau que cette entité soit membre de la super-couche. Nous appelons cette valeur le *nombre d'adéquation*. Grâce à cette valeur, il est possible de comparer deux entités, et, ainsi, d'élire la meilleure pour faire partie de la super-couche. Par exemple, il est facile d'évaluer et comparer deux niveaux d'énergie ou de comparer le nombre de données stockées dans une mémoire tampon.

L'algorithme de construction du MIS consiste en plusieurs étapes de calcul qui sont effectuées de manière périodique par toutes les entités. A chaque étape de calcul, chaque entité réalise un calcul local pour déterminer si elle pense qu'elle peut prétendre appartenir au MIS. Puis, elle échange ces informations locales avec ses voisines.

L'état d'une entité inclut une variable *statut*, qui peut être *passive*, *bridge* ou *active*, son *nombre d'adéquation*, sa connaissance des variables d'état de ses voisines (basé sur les dernières informations reçues) et, pour chaque voisine, la liste de ses voisines actives. Le statut $actif$ signifie que l'entité se considère comme membre du MIS, *bridge* indique qu'il agit comme pont et *passive* est le complémentaire.

L'exécution locale de l'algorithme de construction du MIS inclue les règles $\mathcal{R}_1$, $\mathcal{R}_2$ et $\mathcal{R}_3$ de l'Algorithme 9. L'algorithme que nous proposons est plus efficace que celui proposé dans les réseaux de capteurs dans [53] puisqu'il recalcule dynamiquement la fonction du MIS en se basant sur le nombre d'adéquation.

La première règle $\mathcal{R}_1$ est utilisée pour élire les entités possédant le meilleur nombre d'adéquation dans leur voisinage. La deuxième règle $\mathcal{R}_2$ assure qu'il n'existe aucune situation dans laquelle deux entités voisines appartiennent simultanément au MIS. Cela pourrait arriver, par exemple, dans le cas d'un déplacement. Ainsi, si une entité *active* détecte une voisine *active* ayant un nombre d'adéquation supérieur au sien, elle devient immédiatement *passive*. La troisième règle $\mathcal{R}_3$ est exécutée par une entité *passive* ou *bridge* n'ayant aucune voisine *active*, même si elle ne possède pas le meilleur nombre d'adéquation. Une telle situation peut survenir, par exemple, lorsque toutes les voisines ayant un nombre d'adéquation supérieur au sien ne peuvent pas devenir *active* parce qu'ils possèdent des voisines ayant un meilleur nombre d'adéquation que le leur.

Les règles $\mathcal{R}_1$ et $\mathcal{R}_3$ sont basées sur l'évaluation du meilleur nombre d'adéquation dans le voisinage. Dans le cas où deux entités ont le même nombre d'adéquation, la symétrie est cassée par les identifiants des machines. Dans la suite, e_i représente le nombre d'adéquation

de l'entité i. Pour simplifier la présentation de l'algorithme, nous introduisons la relation $\prec$. L'entité j est meilleure que l'entité i selon la relation $\prec$ si, soit le nombre d'adéquation de j est supérieur à celui de i, soit leurs nombres d'adéquation sont égaux et l'identifiant de j est supérieur à celui de i. Formellement, nous avons $i \prec j$ si et seulement si $e_i < e_j \vee (ei = e_j \wedge id_i < id_j)$. A noter que la relation $\prec$ définit un ordre total sur les entités dès que leurs nombres d'adéquation deviennent comparables. Enfin, $MAXE(i)$ est un prédicat booléen qui vaut `vrai` si et seulement si i est maximal dans son voisinage, selon la relation $\prec$.

Algorithme 9 Algorithme de construction de MIS exécuté par l'entité i

Paramètres d'entrée :
$\mathcal{N}(i)$: ensemble des voisins de i ;
$status_j, \forall j \in \mathcal{N}(i)$: le statut des voisins de i ;
$e_j, \forall j \in \mathcal{N}(i)$: le nombre d'adéquation des voisins de i ;

Paramètres d'entrée/sortie :
$statut_i$: statut de l'entité i ;

Prédicats :
$i \prec j \equiv e_i < e_j \vee (ei = e_j \wedge id_i < id_j)$;
$MAXE(i) \equiv \forall j \in \mathcal{N}(i), j \prec i$;

Actions :
$\mathcal{R}_1 : MAXE(i) \wedge statut_i \neq statut_j \rightarrow statut_i = active$;
$\mathcal{R}_2 : statut_i = active \wedge \exists j \in \mathcal{N}(i), statut_j = active \wedge i \prec j \rightarrow statut_i = passive$;
$\mathcal{R}_3 : \neg MAXE(i) \wedge \forall j \in \mathcal{N}(i), statut_j \neq active \rightarrow statut_i = active$;

L'Algorithme 10 permet de déterminer les ponts entre les entités constituant le MIS. Cet algorithme possède trois règles et partage avec l'Algorithme 9 l'ensemble des voisins et des variables de statut. La première règle $\mathcal{R}_1$ est utilisée pour connecter deux entités du MIS. Si une entité i *passive* possède deux voisines *active*, alors i devient *bridge*. La seconde règle $\mathcal{R}_2$ permet d'éliminer des ponts qui ne sont pas nécessaires. Si une entité *bridge* ne possède plus deux voisines *active*, alors elle revient à un statut *passive*. Enfin, la règle $\mathcal{R}_3$ permet de supprimer des ponts redondants. Considérons deux ponts i et j. Si l'ensemble des voisines *active* de i est inclus dans l'ensemble des voisines *active* de j, ou, si leurs deux ensembles sont identiques mais j a un meilleur nombre d'adéquation que i, alors i doit revenir à un statut *passive*.

4.4.2 Ensemble dominant connexe

Nous commençons par donner une définition formelle des ensembles dominants connexes (DS), puis nous présentons l'algorithme que nous proposons.

Soit $G = (V, E)$ un graphe de communication. Un ensemble d'entités $S \subset V$ est *dominant* si toutes les entités de V, soit sont membres de S, soit possèdent une voisine dans S. L'ensemble S est *connexe* si toute entité x de S possède une voisine dans S.

Dans la suite, nous présentons une version auto-stabilisante de l'algorithme de construction d'un DS présenté dans [105, 106]. Pour cet algorithme proactif, il est impératif que chaque entité connaisse toutes ses voisines de niveau 2, *i.e.* les voisines de ses voisines

Algorithme 10 Algorithme de construction de ponts exécuté par l'entité i

Paramètres d'entrée :

$\mathcal{N}(i)$: ensemble des voisins de i ;
$status_j, \forall j \in \mathcal{N}(i)$: le statut des voisins de i ;
$e_j, \forall j \in \mathcal{N}(i)$: le nombre d'adéquation des voisins de i ;
$\mathcal{AN}(i)$: l'ensemble des voisins *active* de i ;

Paramètres d'entrée/sortie :

$statut_i$: statut de l'entité i ;

Prédicats :

$i \prec j \equiv e_i < e_j \vee (ei = e_j \wedge id_i < id_j)$;

Actions :

$\mathcal{R}_1 : statut_i = passive \wedge \exists j, k \in \mathcal{N}(i), statut_j = statut_k = active \rightarrow statut_i = bridge$;
$\mathcal{R}_2 : statut_i = bridge \wedge \nexists j, k \in \mathcal{N}(i), j \neq k, statut_j = statut_k = active \rightarrow statut_i = passive$;
$\mathcal{R}_3 : statut_i = bridge \wedge \exists j \in \mathcal{N}(i), statut_j = bridge \wedge (\mathcal{AN}(i) \subset \mathcal{AN}(j) \vee \mathcal{AN}(i) = \mathcal{AN}(j) \wedge i \prec j) \rightarrow statut_i = passive$;

directs.

On dit qu'une entité i est *indépendante* lorsque $\exists j, k \in \mathcal{N}(i), j \neq k \neq i, k \notin \mathcal{N}(j) \wedge j \notin \mathcal{N}(k)$. Intuitivement, une entité i est indépendante lorsque i possède deux voisines qui ne sont pas directement voisines les uns des autres. Une entité indépendante doit être en priorité dans l'ensemble dominant, à moins qu'une entité i' soit également voisine de j et k et qu'elle possède un meilleur nombre d'adéquation que i.

Dans l'Algorithme 11, une entité i exécute la règle $\mathcal{R}_1$ si elle est indépendante. Elle devient alors *active*. En revanche, par la règle $\mathcal{R}_2$, si une entité *active* détecte une voisine *active*, qu'ils partagent les mêmes voisines et que cette voisine possède un meilleur nombre d'adéquation, alors elle revient à un statut *passive*.

A noter qu'au début de l'algorithme, toutes les entités peuvent être *passives*. Si le graphe est une clique, alors aucune entité ne peut devenir *active*. La règle $\mathcal{R}_3$ permet d'éviter cette situation. Si toutes les voisines d'une entité i sont *passive* et que i possède le meilleur nombre d'adéquation, alors elle devient *active*. Enfin, la règle $\mathcal{R}_4$ permet d'éliminer les entités *active* redondantes. Si une entité i est *active*, mais qu'elle possède une voisine j *active* et que l'ensemble des voisines de i est inclus dans l'ensemble des voisines de j alors l'entité i revient à un statut *passive*.

4.4.3 Arbre de diffusion pour la recherche en largeur

L'idée de cet algorithme consiste à exploiter le comportement de la recherche en largeur (BFS) et les particularités des communications hertziennes pour produire un arbre de dissémination à chaque requête. De manière intéressante, cet arbre ne requiert aucun délai supplémentaire, ni de messages spécifiques. Rappelons qu'avec l'algorithme de recherche en largeur, les requêtes se propagent par étapes successives de telle manière que, lors de la phase i, le message est disséminé aux entités jusqu'à une distance i du demandeur. Celui-ci attend les réponses de la phase i pour, éventuellement, passer à la phase $i+1$. Ainsi, quand le demandeur a reçu toutes les réponses pour la phase i, il connaît les plus courts chemins entre lui et toutes les entités à distance i. Grâce à ces informations, il est en mesure de

Algorithme 11 Algorithme de construction d'un DS par l'entité i

Paramètres d'entrée :

$\mathcal{N}(i)$: ensemble des voisins de i ;
$statut_j, \forall j \in \mathcal{N}(i)$: statut des voisins de i ;
$e_j, \forall j \in \mathcal{N}(i)$: nombre d'adéquation des voisins de i ;
$\mathcal{N}(j), \forall j \in \mathcal{N}(i)$: ensemble des voisins des voisins de i ;

Paramètres d'entrée/sortie :

$statut_i$: statut de l'entité i

Predicates :

$Independent\_Neighbors(i) \equiv \exists j, k \in \mathcal{N}(i), j \neq k \neq i, k \notin \mathcal{N}(j) \wedge j \notin \mathcal{N}(k)$;
$i \prec j \equiv e_i < e_j \vee (e_i = e_j \wedge id_i < id_j)$
$MAXE(i) \equiv \forall j \in \mathcal{N}(i), j \neq i, j \prec i$

Actions :

$\mathcal{R}_1 : statut_i = passive \wedge Independent\_Neighbors(i) \rightarrow statut_i = active$
$\mathcal{R}_2 : \exists j \in \mathcal{N}(i), \mathcal{N}(j) = \mathcal{N}(i) \wedge i \prec j \wedge statut_i = statut_j = active \rightarrow statut_i = passive$
$\mathcal{R}_3 : \forall j \in \mathcal{N}(i), \mathcal{N}(j) = \mathcal{N}(i) \wedge MAXE(i) \wedge statut_i = passive \rightarrow statut_i = active$
$\mathcal{R}_4 : \exists j \in \mathcal{N}(i), \mathcal{N}(i) \subset \mathcal{N}(j) \wedge statut_i = statut_j = active \rightarrow statut_i = passive$

calculer le plus petit ensemble d'entités nécessaires à la dissémination de la requête émise à la phase $i + 1$ pour que le message atteigne les entités situées à une distance $i + 1$.

Pour obtenir ce comportement, le demandeur spécifie dans son message de requête les entités chargées de la retransmission. Ces entités élues forment une super-couche. Le message de requête a alors le format suivant : $< req, demandeur, d, phase, dist, \mathcal{R}, \mathcal{P} >$ dans lequel d est la donnée demandée par *demandeur*, *phase* représente la distance des entités desquelles le demandeur attend un message, *dist* est le nombre de retransmissions effectuées, $\mathcal{R}$ est l'ensemble des entités élues et $\mathcal{P}$ est l'ensemble des entités qui ont participé à la dissémination. Une entité retransmet un message si sa distance avec le demandeur est inférieure à *phase* et si elle appartient à l'ensemble $\mathcal{R}$. Dans ce cas, elle s'ajoute à l'ensemble $\mathcal{P}$.

4.5 Simulations

4.5.1 Environnement de simulations

Nous avons conçu un espace de simulation de 500×500 m^2. Les entités sont initialement placées à des positions choisies au hasard dans cet espace. Ensuite, durant les simulations, les entités se déplacent selon une version augmentée du modèle appelé *Random-Waypoint* [24, 64]. Celui-ci spécifie que chaque entité choisit une nouvelle destination au hasard et se dirige vers elle à une vitesse choisie inférieure à une vitesse maximale fixée. Une fois la destination atteinte, l'entité demeure à sa position pendant un certain temps, qui, dans nos simulations, est choisi entre 1 à 2 minutes.

Dans des environnements plus réalistes, les entités ne se déplacent généralement pas vers des positions choisies totalement au hasard, mais sont plutôt attirées par certaines zones plus populaires. Par exemple, dans un salon commercial, les visiteurs errent de stands en stands. Nous avons donc décidé d'inclure au modèle traditionnel quelques zones pri-

Algorithme 12 un service de recherche basé sur une recherche en largeur utilisant un arbre de diffusion (entité i)

Paramètres d'entrée :

$\mathcal{N}(i)$: ensemble des voisins de i ;
$construitArbre(\Lambda)$: retourne un arbre minimal à partir de l'ensemble des chemins Λ ;

Actions :

$\mathcal{R}_1$: l'entité i initie une recherche
$round = 1; \mathcal{R} = \mathcal{N}(i)$
répète
envoie $< req, i, d, round, 1, \mathcal{R}, \emptyset >$ à tous les voisins
réceptionne $< resp, j, presence_j, \mathcal{P}_j >$ pour tout $j \in \mathcal{N}(i) \cap \mathcal{R}$ ou pendant un délai
si réception d'au moins un message $< resp, j, \texttt{true}, \mathcal{P} >$ **alors**
retourne une entité j parmi les réponses positives
si aucune réponse reçue
retourne $\perp$
$round = round + 1$
$\mathcal{R} = construitArbre(\bigcup_j \mathcal{P}_j)$
fin de la boucle répète

$\mathcal{R}_2$: à la réception d'un message $< req, j, d, round, dist, \mathcal{R}, \mathcal{P} >$
si $dist = round$ et $i \notin \mathcal{P}$ **alors**
si $d \in$ mémoire tampon **alors**
envoie $< resp, i, \texttt{true}, \mathcal{P} \cup \{i\}$
sinon
envoie $< resp, i, \texttt{false}, \mathcal{P} \cup \{i\}$
sinon
si $i \in \mathcal{R}$ **alors**
envoie $< req, j, d, round, dist + 1, \mathcal{R}, \mathcal{P} \cup \{i\} >$

vilégiées, appelées *lieux chauds*. Quand une entité doit choisir une position, elle préférera une position dans un lieu chaud selon une probabilité fixée. En faisant varier cette probabilité, nous pouvons modifier l'homogénéité de la densité d'entités dans le réseau. Quand cette probabilité vaut 0, la densité est uniforme. Quand la probabilité augmente, les lieux chauds deviennent plus denses. Dans nos simulations, nous avons définis trois lieux chauds de taille 50×50 m^2, 62×62 m^2 et 83×83 m^2.

Paramètres	**Valeurs**	***Paramètres***	**Valeurs**
Nombres d'entités	200	*Puissance de transmission*	70 mètres
Vélocité des entités	2 mètres par sec.	*Période entre requêtes*	700 msec.
Durée de transmission	$[1 \dots 4]$ msec.	*Nombre de cibles*	8
Probabilité lieux chauds	0.6	*Perte de paquets*	non

TAB. 4.1 – Paramètres par défaut pour les simulations

La durée d'une simulation est 8 minutes. Pour la mise en œuvre des algorithmes de construction d'une super-couche proactive, la période entre deux calculs consécutifs est uniformément choisi entre 2 et 3 secondes. Le Tableau 4.1 dresse une liste des différents paramètres utilisés par défaut pendant les simulations. Nous avons fait varier la vélocité des entités, la probabilité de lieux chauds, la puissance de transmission et le nombre de cibles. Les critères de performance mesurés sont le *nombre de messages par requête*, le *temps de latence* et le *pourcentage de réussite des requêtes*. Nous avons étudié les schémas suivants : `flooding` - inondation sans contrainte, `MIS flooding` - l'inondation sur une super-couche MIS, `DS flooding` - l'inondation sur une super-couche DS, `probabilistic` - inondation probabiliste, `BFS-tree` - la recherche en largeur avec construction d'un arbre de dissémination et `MIS-tree` - la recherche en largeur utilisant une super-couche MIS.

4.5.2 Pourcentage de requêtes satisfaites

La figure 4.1 présente le pourcentage de requêtes satisfaites pour tous les mécanismes lorsque les principaux paramètres du système varient.

Dans la figure en haut à gauche, le paramètre qui varie est le nombre de cibles dans le réseau.

Comme espéré, le pourcentage de requêtes satisfaites augmente avec le nombre de cibles et converge vers 100%. Nous pouvons particulièrement noter que les courbes produisent un "coude" lorsque le réseau contient entre 5 et 10 cibles, *i.e.* quand le nombre de cibles est compris entre 2.5% et 5% de l'ensemble des entités. Ce résultat est important puisqu'il indique le nombre optimal de cibles nécessaires pour qu'au moins une cible soit détectée avec une grande probabilité. Ce nombre pourrait être pris en compte par une stratégie de cache.

Naturellement, la stratégie appelée flooding admet le plus grand pourcentage de requêtes satisfaites, mais, à partir de 5 cibles, le BFS-tree et le DS-flooding présentent des résultats à peu près équivalents. C'est intéressant puisque le BFS-tree est une solution totalement réactive. Enfin, nous notons que la stratégie probabiliste fonctionne bien à partir de 10 cibles.

Nous nous intéressons brièvement à la figure en haut à droite, qui présente le pourcentage de requêtes satisfaites lorsque le rayon de transmission évolue. Nous observons que les

évolutions des courbes sont similaires aux courbes de la figure précédente même si le "coude" est moins important. Nous souhaitons particulièrement noter le bon comportement de la stratégie probabiliste quand le rayon de transmission est faible, *i.e.* quand le réseau est plus clairsemé. Malheureusement, lorsque le rayon de transmission augmente, les performances de la stratégie probabiliste augmentent moins vite que les autres mécanismes.

La figure en bas à gauche propose une observation de l'impact de la vitesse maximale des entités sur le pourcentage de requêtes satisfaites. Un premier coup d'œil indique que l'augmentation de la vitesse tend à réduire le pourcentage de réussite des requêtes. Une explication simple vient du fait que les messages de retour deviennent invalides beaucoup plus souvent lorsque les entités se déplacent rapidement. De plus, les techniques proactives admettent des difficultés à gérer les déplacements des entités et ne peuvent, bien souvent, que réagir aux événements. Ainsi, on observe que les mécanismes utilisant une construction de MIS souffrent lorsque la vitesse des entités croît. En revanche, les mécanismes utilisant le DS fonctionnent plutôt correctement, en grande partie grâce aux redondances supplémentaires qui existent dans sa construction. La super-couche DS est donc naturellement moins vulnérable qu'une super-couche MIS.

Enfin, la figure en bas à droite montre l'évolution du pourcentage de requêtes satisfaites lorsque la probabilité de choisir une destination dans un lieu chaud augmente. On observe immédiatement que le nombre de requêtes satisfaites décroît lentement lorsque la densité d'entités dans les lieux chauds augmente. Ceci peut être expliqué par le fait que le risque de partitionnement du graphe augmente puisque les entités se concentrent dans les lieux chauds. Ainsi, certaines recherches ne trouvent pas de cibles. Ici encore la super-couche MIS semble vulnérable. Une nouvelle fois, cela semble naturel puisque cette super-couche contient moins d'entités que ses concurrentes. D'un autre côté, la super-couche DS confirme ses bonnes propriétés, puisqu'elle semble réussir à inclure des entités connectant les lieux chauds.

4.5.3 Temps de latence

La figure 4.2 présente les variations du temps de latence, pour chaque mécanisme, alors que les différents paramètres évoluent.

Dans la figure en haut à gauche, le nombre de cibles dans le réseau varie. Naturellement, le temps de latence diminue quand le nombre de cibles augmente, puisque la probabilité d'avoir une cible près d'un demandeur augmente. De manière très intéressante, nous observons une nouvelle fois la présence d'un "coude" entre 5 et 10 cibles, ou entre 2.5% et 5%, ce qui correspond aux observations sur le nombre de cibles nécessaires pour une recherche fructueuse. Sans surprise, les techniques utilisant une recherche basée sur un BFS nécessitent davantage de temps pour donner une réponse, particulièrement lorsque le nombre de cibles est faible puisque la probabilité d'avoir une cible lointaine augmente.

Dans la figure en haut à droite, le rayon de transmission varie. Comme on peut l'observer, les mécanismes basés sur l'inondation semblent insensibles à cette évolution, voire même voient leur performances s'améliorer avec l'augmentation du rayon. Pour les mécanismes basés sur le BFS, les gains de performance sont plus significatifs. L'augmentation du rayon réduit le nombre d'itérations nécessaires pour la découverte d'une cible. Comme le temps de latence est très sensible au nombre d'itérations pour ce mécanisme,

les résultats progressent nettement.

Dans la figure en bas à gauche, la vitesse maximale des entités varie. Il apparaît que la vitesse des entités n'a quasiment pas d'impact sur le temps de latence.

Enfin, la figure en bas à droite montre l'évolution du temps de latence lorsque la probabilité de choisir une destination dans un lieu chaud augmente. Ici, nous observons une nette progression pour les mécanismes basés sur le BFS et une progression plus faible pour les techniques utilisant l'inondation. Une nouvelle fois, la raison est à chercher du côté de la diminution du nombre de hops entre le demandeur et la cible. Dans des régions plus denses, une cible découverte est plus proche du demandeur que dans un réseau possédant une densité homogène.

4.5.4 Besoins de communication

La figure 4.3 présente le nombre de messages générés, par requête, pour chacun des mécanismes avec des paramètres du réseau qui varient. Cette série de courbes est sans doute la plus importante puisque l'objectif des super-couches consiste, justement, à réduire le nombre de messages.

Dans la première figure (en haut à gauche), le nombre de cibles varie. Or, lorsque le nombre de cibles augmente, le nombre de messages par requête augmente pour les mécanismes basés sur l'inondation alors qu'il diminue pour les propagations utilisant BFS. En particulier, à partir de 10 cibles, soit 5% des entitiés qui sont cibles, le BFS nécessite moins de messages que l'inondation. Une nouvelle fois, ce phénomène n'est pas surprenant. Le fonctionnement du BFS provoque des gains en performance lorsque les cibles sont proches des demandeurs. Le contrôle de la propagation produit indéniablement des résultats intéressants quand la densité des cibles dans le réseau est importante. Au contraire, l'inondation est de moins en moins efficace lorsque le nombre de cibles augmente. En effet, chaque cible génère un message de réponses et si le nombre de cibles augmente, le nombre de messages de réponse augmente.

Parmi les mécanismes basés sur l'inondation, nous notons sans surprise que l'inondation totale produit les pires résultats. Le nombre de messages générés est une indication du nombre d'entités impliquées dans une super-couche. Dans cette série de simulations, la super-couche la plus efficace est le MIS. De même, le BFS avec le MIS produit de meilleurs résultats que le BFS avec la technique de l'arbre de diffusion. Enfin, nous observons que, dans ce contexte, l'inondation probabiliste produit des performances supérieures à celles de la super-couche DS, et pas trop éloignées de la super-couche MIS.

Dans la figure en haut à droite, nous poursuivons notre exploration en faisant varier le rayon de transmission. Naturellement, le nombre de messages augmente avec le rayon de transmission puisqu'un message est reçu, et donc éventuellement retransmis, par davantage d'entités. Nous pouvons tout de même constater que la propagation par BFS subit moins l'impact de cette variation. L'augmentation du nombre de messages lors de la propagation est annulée par le fait que la distance entre le demandeur et la cible se réduit.

Nous avons observé précédemment que l'inondation probabiliste produit des résultats dont l'évolution diffère des autres mécanismes. Cette observation est confirmée, puisque l'inondation probabiliste fonctionne plutôt mal lorsque le rayon de transmission est faible, *i.e.* quand le réseau est faiblement connecté. En revanche, son comportement s'améliore

notablement lorsque le rayon de transmission augmente. Ces observations militent pour une étude plus approfondie de ce mécanisme, afin de le rendre aussi efficace dans toutes les situations.

La figure en bas à droite montre l'évolution du nombre de messages lorsque la vitesse maximale des entités varie. Nous observons que ce nombre diminue lorsque la vitesse augmente. Ce résultat est plutôt négatif puisqu'il peut s'expliquer par le fait que les messages de réponse et de recherche sont davantage perdus lorsque la vitesse augmente.

Enfin, dans la figure en bas à droite, la probabilité de choisir une destination dans un lieu chaud augmente. Quel que soit le mécanisme, le nombre de messages diminue fortement. Lorsque les entités se situent dans un lieu chaud, il est probable qu'une cible soit découverte au sein de ce lieu chaud, et que le message de recherche ne parvienne pas jusqu'aux autres lieux chauds. D'où, en moyenne, une diminution du nombre de messages à mettre en corrélation avec la diminution du nombre de requêtes satisfaites. Les propagations par BFS sont moins influencées par ce paramètre que les mécanismes basés sur l'inondation. En effet, si la cible n'est pas dans un lieu chaud, les multiples retransmissions pour atteindre les autres lieux chauds sont très coûteuses.

4.6 Discussion

Nos simulations ont mis en lumière un phénomène peu surprenant : en général, il y a un compromis à trouver entre l'efficacité et les performances. Plus un mécanisme est efficace en terme de pourcentage de réussite, plus il est gourmand en ressources.

Nous avons montré qu'il existe un nombre optimal de cibles, dans notre cas entre 2.5% et 5% des entités, qui permet à nos algorithmes de découvrir au moins une cible avec une forte probabilité tout en garantissant des performances correctes. Nous avons vu que l'augmentation du nombre de cibles n'apportait pas de gain pour le pourcentage de réussite du service. Comme ce même nombre de cibles produit également les meilleurs résultats pour le temps de latence, une piste intéressante est dévoilée. Les architectes de stratégies de cache pour réseau ad-hoc savent que la réplication des données dans le réseau doit se rapprocher de cette valeur tandis que, pour Solipsis, la définition du prédicat doit faire en sorte que le nombre de cibles appartienne à cet intervalle.

Nous pouvons constater également que les mécanismes basés sur la propagation en largeur sont les plus performants lorsque le nombre de cibles est optimal. Le contrôle de la propagation permet d'économiser de nombreux messages dès que la cible est découverte. De plus, lorsque le nombre de cibles augmente considérablement, cette technique de propagation devient encore plus attractive. Nous avons néanmoins identifié que le temps de latence n'est pas le meilleur. De même, nous avons mis en lumière que l'inondation probabiliste offre un bon compromis en terme d'efficacité et de performance, d'autant plus que l'algorithme est simple et totalement réactif.

Au contraire, les techniques se basant sur la construction d'une super-couche offre des résultats décevants. Dans ces simulations, nous n'avons pas tenu compte des messages nécessaires à leur maintien. Si nous ajoutons ces messages, il semble que les bénéfices issus de leur utilisation ne justifient pas le coût induit par leur construction. Malgré tout, ces algorithmes, particulièrement le MIS, sont intéressants dans le contexte de réseaux ad-hoc quasiment statiques ou de réseaux de capteurs. En effet, dans ces réseaux, le coût de la

maintenance de la super-couche est faible puisque les entités ne se déplacent pas ou peu.

4.7 Conclusion

Dans ce chapitre, nous avons étudié un service de recherche pour réseaux mobiles distribués, typiquement Solipsis ou des réseaux ad-hoc. Nous avons d'abord cherché à définir formellement notre service, puis nous avons proposé plusieurs techniques permettant de le réaliser. Les simulations visent à observer le comportement de ces techniques dans un réseau ad-hoc dans lequel les paramètres varient. Ce réseau possède de nombreuses similitudes avec la topologie de Solipsis et cette étude offre des pistes pour la mise en œuvre d'un tel service dans notre environnement virtuel partagé.

Les principaux résultats montrent que, dans le cadre de Solipsis, une solution de recherche réactive, notamment probabiliste, peut suffire à mettre en œuvre un service efficace et performant. De plus, il est probable que le graphe de communication de Solipsis admette un diamètre très important. Dans ce cadre, la propagation en largeur semble être la meilleure méthode permettant de diffuser les requêtes.

Dans un futur immédiat, nous chercherons à proposer des algorithmes permettant de construire une super-couche de manière réactive en utilisant les messages circulant dans le réseau. Enfin, nous explorerons l'inondation probabiliste qui reste encore une solution largement empirique.

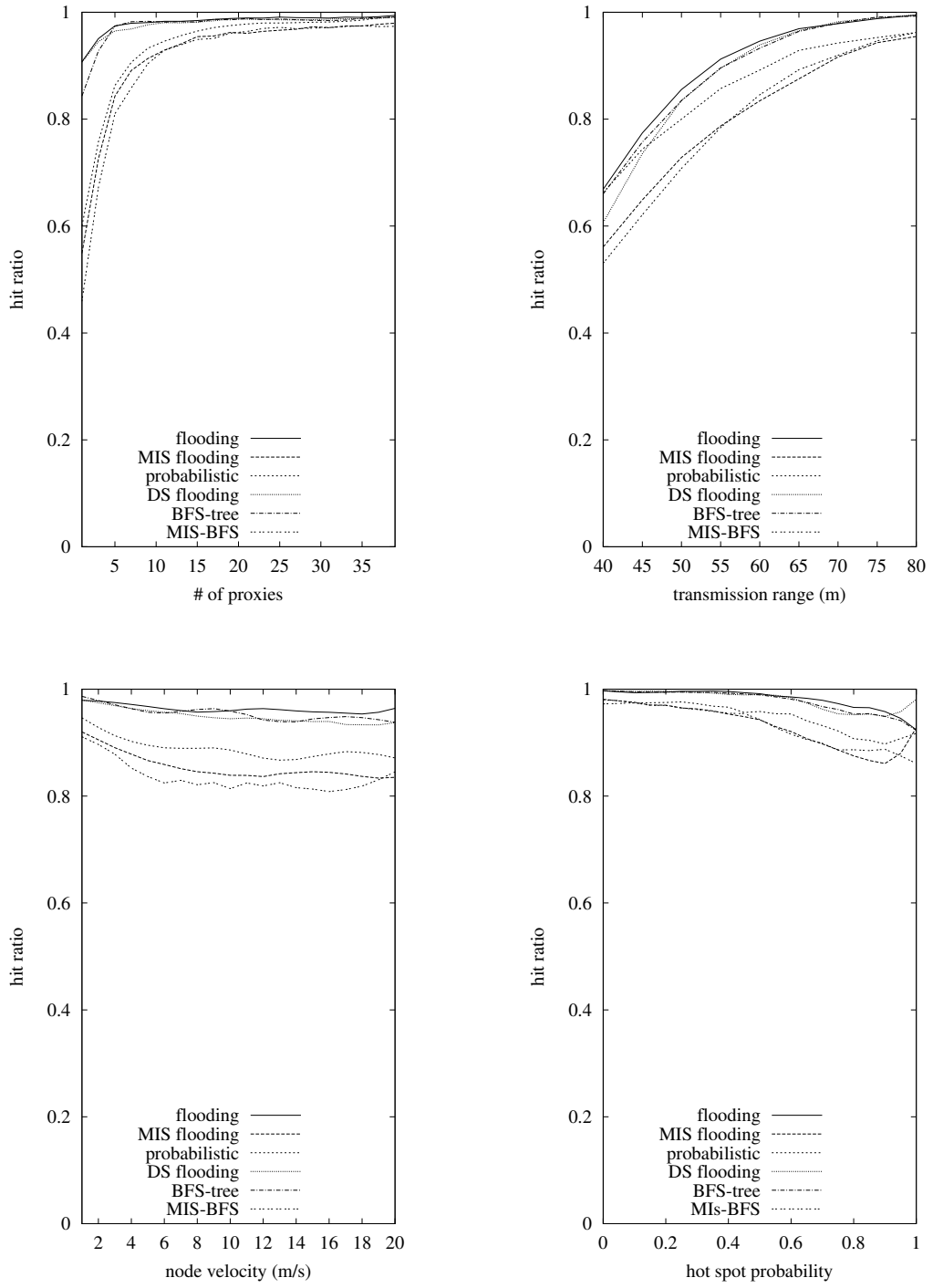

FIG. 4.1 – Évolution du pourcentage de requêtes satisfaites lorsque le nombre de cibles varie (en haut à gauche), lorsque le rayon de transmission varie (en haut à droite), lorsque la vitesse maximale des entités varie (en bas à gauche) et lorsque la probabilité de choisir une destination dans un lieu chaud varie (en bas à droite)

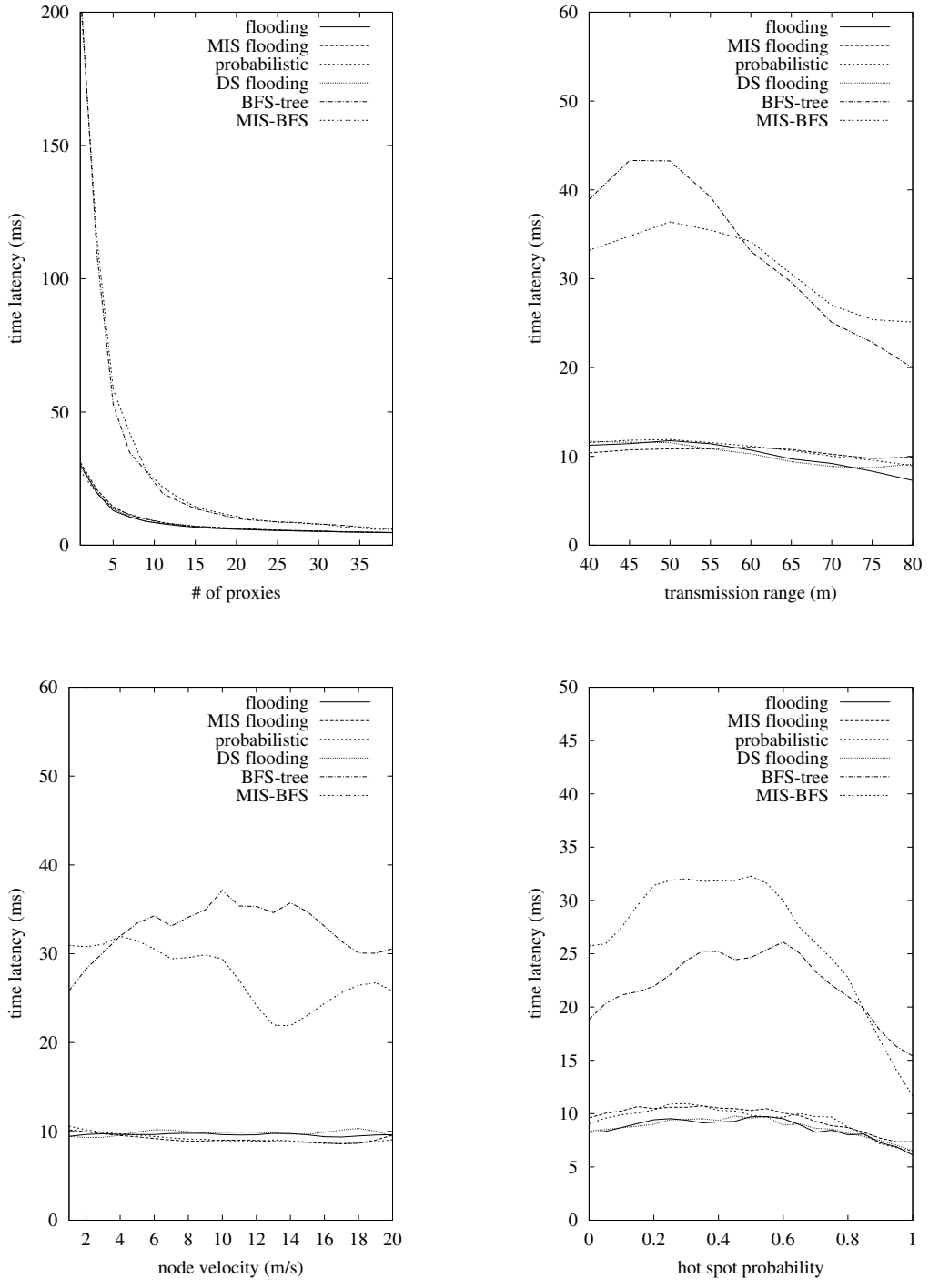

FIG. 4.2 – Évolution du temps de latence lorsque le nombre de cibles varie (en haut à gauche), lorsque le rayon de transmission varie (en haut à droite), lorsque la vitesse maximale des entités varie (en bas à gauche) et lorsque la probabilité de choisir une destination dans un lieu chaud varie (en bas à droite)

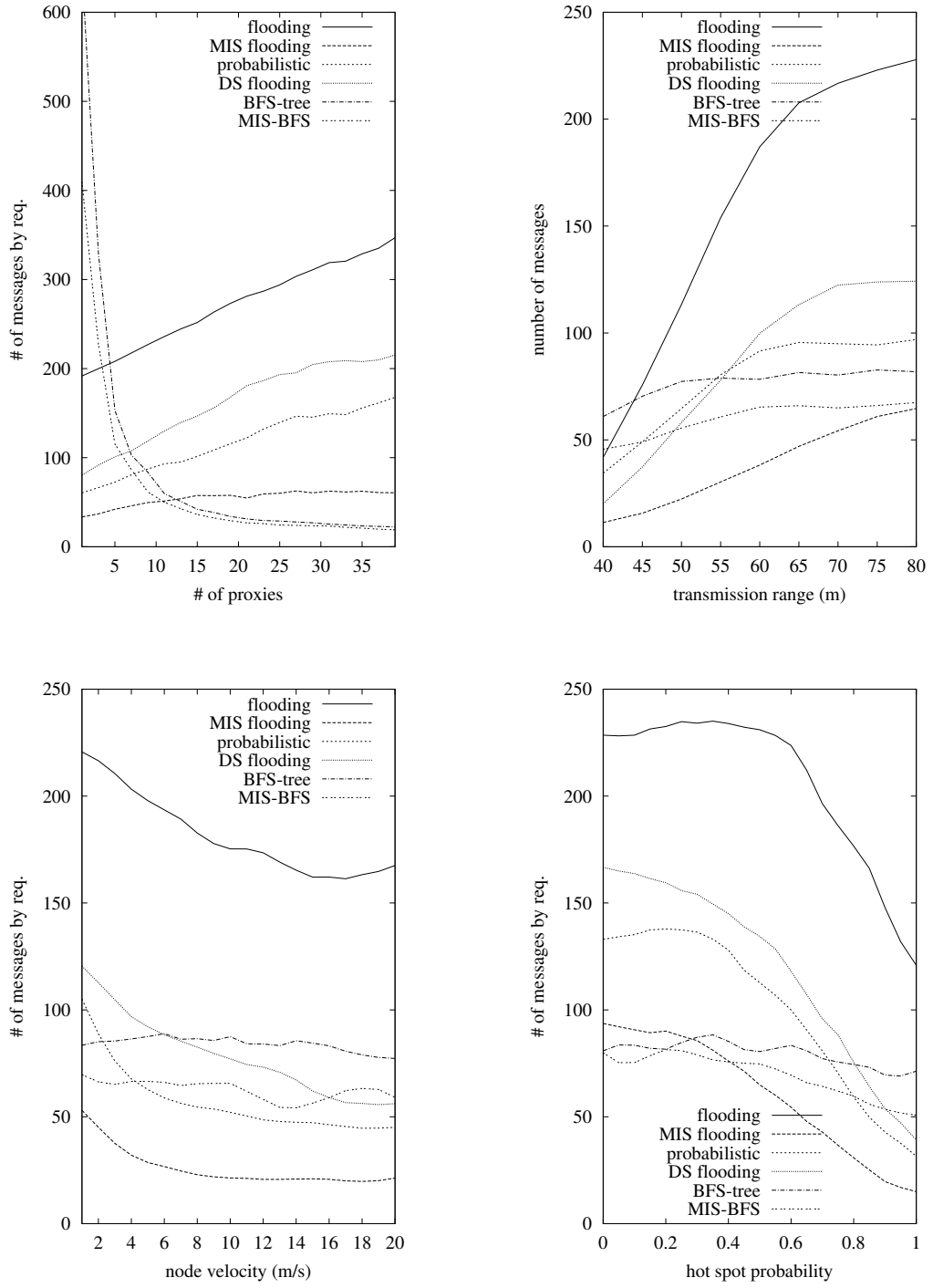

FIG. 4.3 – Évolution du nombre de messages par requête lorsque le nombre de cibles varie (en haut à gauche), lorsque le rayon de transmission varie (en haut à droite), lorsque la vitesse maximale des entités varie (en bas à gauche) et lorsque la probabilité de choisir une destination dans un lieu chaud varie (en bas à droite)

Chapitre 5

Publication/Abonnement

5.1 Introduction

5.1.1 Réseaux de pairs

Un réseau de pairs est constitué d'un ensemble dynamique et scalable de processeurs, appelés *pairs*. Cette dénomination englobe le principe même de ces systèmes : les participants ont originellement les mêmes responsabilités. Les principales caractéristiques de ces organisations logiques de réseaux résident dans leur capacité à regrouper et gérer d'importantes quantités de ressources, à s'organiser sans aide extérieure, à répartir équitablement la charge, à s'adapter aux situations et à tolérer les défaillances. Les applications basées sur des réseaux de pairs visent à permettre le partage des ressources individuelles par des échanges directs entre les participants. Ces ressources partagées incluent l'espace mémoire pour le stockage de fichiers ou la bande-passante utilisée pour réaliser des transferts d'informations.

Nous distinguons deux catégories de réseaux de pairs selon leur degré de centralisation. Dans les systèmes que nous appelons *purs*, tous les pairs jouent exactement le même rôle [5, 30]. Toutes les communications entre les pairs sont symétriques. Chaque pair joue à la fois le rôle de client et de serveur.

Au contraire, dans les réseaux de pairs appelés *hybrides*, seuls quelques pairs se regroupent au sein d'un réseau de pairs pur [107]. On appelle ces pairs des *super-pairs*. Ce sont des machines répondant à un certain nombre de critères tels qu'une large bande-passante disponible, une puissance de calcul importante ou encore la disponibilité sur le réseau. Les autres participants sont directement connectés à un super-pair comme dans un modèle traditionnel client-serveur. Il existe donc deux niveaux dans l'architecture comme l'illustre la figure 5.1. Le fonctionnement des sous-réseaux associés à chaque super-pair est similaire à celui de réseaux dits centralisés. Un pair lance une requête à un super-pair qui se charge, ensuite, de diffuser cette requête dans le réseau des super-pairs.

La récente popularité des applications basées sur des réseaux de pairs provient essentiellement de la démocratisation des moyens d'accès au réseau Internet, mais aussi à quelques initiatives individuelles qui ont conduit à l'élaboration d'applications telles que Napster, Gnutella [5] ou Kazaa [2]. Plusieurs dizaines de millions d'utilisateurs ont ainsi utilisé ces programmes pour s'échanger des fichiers. A leur manière, les créateurs de ces applications ont révolutionné la manière d'appréhender Internet et ont ouvert un nouvel

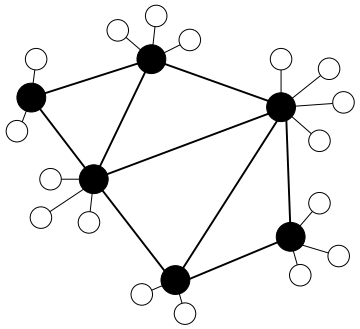

FIG. 5.1 – Réseau de pairs hybride : les disques noirs représentent les super-pairs tandis que les disques blancs sont les pairs.

axe de recherche fondamental proche de l'algorithmique distribuée.

Les systèmes de *tables de hachage distribuées* tentent de résoudre le principal problème rencontré par ces applications. Alors que celles-ci utilisent majoritairement des mécanismes de routage simplistes basés sur une diffusion massive des messages de requêtes, les systèmes de table de hachage distribuée tentent de minimiser le trafic en cherchant à localiser efficacement le ou les destinataires d'un message.

Dans de tels systèmes, les mécanismes de routage et de localisation sont mêlés. Localiser un objet dans le système revient à transmettre un message à l'entité responsable de cet objet. Les pairs formant le réseau et les objets qui s'y trouvent se voient attribuer un identifiant unique issu d'un seul espace de noms. Cet identifiant est utilisé lors du routage d'un message à destination d'un pair ou d'un objet.

Un processus distribué attribue à chaque participant la responsabilité d'un sous-ensemble d'identifiants de l'espace de noms global. Une entité est ainsi responsable des objets dont les identifiants appartiennent au sous-ensemble dont il a la responsabilité. Pour transmettre un message à destination d'un identifiant, un mécanisme de transmission de proche en proche est utilisé. Une entité recevant une requête pour un objet la retransmet au voisin dont l'indentifiant est le plus proche, dans l'espace des noms, de l'identifiant de l'objet.

Ces systèmes sont typiquement utilisés pour rendre accessible des objets. La récupération de ces objets utilise deux primitives de base : l'insertion d'un objet, identifié par une clef, et la récupération d'un objet via sa clef identifiante. Les systèmes existants [92, 86, 87, 40, 65] diffèrent principalement par les méthodes d'apprentissage et de maintien du voisinage.

Un autre axe de recherche important se situe autour de la diffusion de contenus grâce aux ressources offertes par les utilisateurs de réseaux de pairs. Plusieurs axes d'optimisation ont été proposés. L'application BitTorrent [32] permet à un grand nombre de participants de télécharger simultanément un même fichier. Dans Freenet [30], plusieurs mécanismes complexes garantissent l'anonymat, aussi bien de la source du contenu que des participants qui désirent la récupérer. Enfin, le système SplitStream [27] permet à un grand nombre d'utilisateurs de jouir d'un flux vidéo grâce à la construction de plusieurs arbres de diffusion.

5.1.2 Abonnés et éditeurs

Nous pensons que les utilisateurs de Solipsis seront intéressés par un service consistant à publier des informations et à recevoir des informations quand elles sont pertinentes. Nous décrivons ci-dessous les principes généraux d'un tel service appelé *Abonné/Editeur*.

Chaque participant peut jouer le rôle d'*éditeur* ou d'*abonné*. Les éditeurs produisent des informations, généralement appelées *notifications*, qui sont ensuite consommées par les abonnés. La principale caractéristique des mécanismes d'abonné/éditeur réside dans le fait que les abonnés reçoivent les notifications dont le contenu est relatif à certains critères. Préalablement, ils ont exprimé leur *intérêt* pour certains types de notifications.

On distingue généralement deux catégories de systèmes d'abonnés/éditeurs : ceux qui se basent sur des *sujets* et ceux qui se basent sur le *contenu*. Les premiers considèrent qu'il est possible de déterminer, pour chaque notification, le sujet sur lequel la notification porte. Il est alors possible d'exprimer un intérêt par un ou plusieurs sujets et, ainsi, de faciliter le travail consistant à vérifier l'adéquation entre une notification et un intérêt. Dans les seconds, les intérêts sont décrits par un langage complexe se présentant sous la forme d'une collection de prédicats de la forme (t, op, v), dans lequel t est un attribut, op est un opérateur et v est une valeur. Ces prédicats sont reliés par un opérateur logique. Par exemple, un intérêt relatif à toutes les notifications concernant un pays dont le nom commence par M et évoquant une température inférieure à 3° s'exprime par (Pays = M*) $\wedge$ $(T < 3)$. A noter que les systèmes d'abonnés/éditeurs basés sur les sujets sont une simplification des systèmes basés sur le contenu.

Il est impossible d'envisager que tous les éditeurs puissent connaître tous les intérêts de tous les abonnés. Traditionnellement, les éditeurs et les abonnés sont reliés à un intermédiaire qui se charge de stocker tous les intérêts exprimés, de recevoir toutes les notifications, puis de transmettre les notifications aux abonnés quand celles-ci sont conformes à leurs intérêts. La réalisation d'un intermédiaire par un réseau de pairs vise à permettre au service de passer à l'échelle. Le défi que cette réalisation soulève est décrit dans [37].

Les systèmes proposés [1, 3, 28, 82] se basent sur des réseaux de pairs hybrides dans lesquels les intermédiaires sont des super-pairs. Dans les systèmes Scribe [28] et Hermes [82], les entités ayant les mêmes intérêts sont regroupées. Elles dépendent du même super-pair, qui est responsable de tous les intérêts relatifs à certains attributs. Au contraire, dans les systèmes comme Gryphon [1] ou Siena [3], une entité détermine son super-pair dès son entrée dans le système.

Ces deux méthodes d'attribution d'un pair à un super-pair ont un impact majeur sur l'organisation des systèmes. Dans le premier cas, lors de la publication d'une notification, l'éditeur doit découvrir le ou les super-pairs responsables des attributs de la notification et lui envoyer la notification. A la réception, le super-pair peut déterminer les éventuels abonnés intéressés. Dans le second cas, l'éditeur envoie sa notification à son super-pair qui diffuse cette notification dans le réseau, chaque super-pair devant déterminer s'il possède des abonnés intéressés.

Nous décrivons dans la suite Scribe et Siena.

Scribe est une interface d'éditeur/abonné utilisant Pastry [87], un système de table de hachage distribué. Chaque intérêt est associé à un identifiant déterminé aléatoirement. L'entité dont l'identifiant dans Pastry est le plus proche de l'identifiant d'un intérêt est

responsable de celui-ci. Un arbre de diffusion multipoint est construit pour chaque intérêt par l'entité responsable agissant comme racine. Lors d'un nouvel abonnement, le message est transmis au responsable de cet intérêt qui rafraîchit l'arbre de diffusion. De même, une notification est envoyée jusqu'au responsable de l'intérêt sur lequel porte cette notification. Il se charge ensuite de transmettre cette notification grâce à l'arbre de diffusion.

Dans Siena, chaque entité est tenue de déclarer par un message spécifique les principaux attributs des notifications qu'elle compte éditer dans le futur. Ces notifications sont diffusées dans le réseau de super-pairs.x Quand un super-pair possède un client ayant émis un intérêt pour un attribut appraissant dans cette notification, ce super-pair répond au message en précisant qu'il est intéressé. Le super-pair qui gère le client construit, grâce à ces réponses, un arbre de diffusion qu'il utilisera lors de la publication d'une nouvelle notification.

De nombreuses optimisations ont été proposées, notamment pour calculer l'adéquation entre les notifications et les intérêts [9, 46, 101] ou pour délivrer les notifications [12, 25]. Les algorithmes proposés dépendent fortement des caractéristiques du système sur lequel ils s'appuient. Une étude récente et complète de ces systèmes peut se trouver dans [99].

5.1.3 Contributions

Il nous semble difficile d'utiliser les solutions existantes pour élaborer un tel service dans Solipsis. D'une part, construire et maintenir une topologie basée sur une DHT comme le propose Scribe se heurte au contexte de grande mobilité des entités dans Solipsis. Malgré l'efficacité des algorithmes utilisés, la rigidité de la topologie implique des échanges de messages à chaque événement : apparition d'un nouveau participant, disparition d'une entité, émission d'un nouvel intérêt... D'autre part, l'implication d'un grand nombre de participants à chaque nouvelle notification ou à chaque nouvel intérêt émis contrevient à notre volonté de garantir le passage à l'échelle du service. Enfin, il est courant que, dans de tels services, un nombre important d'entités émettent une notification, relative à un événement, simultanément. La publication d'une notification engendre l'utilisation de ressources et ces comportements peuvent provoquer des pics d'activités.

Nous avons voulu élaborer un service plus flexible dans lequel les conséquences de la mobilité ne contraignent pas les entités à des traitements trop lourds. Il nous a paru important de permettre aux entités de se connecter au système le plus rapidement possible, notamment avec des voisins virtuels, même s'ils ne possèdent pas les mêmes intérêts. En outre, les algorithmes mis en œuvre doivent être peu coûteux, simples et offrir une grande robustesse aux événements qui peuvent survenir. Enfin, le service doit, dans la mesure du possible, passer à l'échelle et pouvoir faire face aux pics d'activités. Nous avons partiellement décrit ce système dans [11] et [47].

Nous sommes partis d'une modélisation originale en couches. Chaque couche est un sous-réseau des participants indépendante des autres couches du système. Ce système offre le double avantage de permettre un nombre illimité d'émissions d'intérêts et d'impliquer dans la diffusion d'une notification uniquement les entités potentiellement intéressées par cette notification. Une présentation de cette organisation se trouve dans la partie 5.2.

Afin de contrevenir aux pics d'activités, nous avons élaboré des règles qui brisent la symétrie des réseaux de pairs associés à chaque couche. Nous proposons dans la par-

tie 5.3 une collection d'algorithmes permettant d'orienter logiquement les connexions entre les participants de manière à obtenir une organisation en graphe orienté acycliquement. Ces graphes présentent l'avantage de privilégier localement certaines entités. Nous utilisons cette caractéristique naturelle pour proposer un mécanisme de diffusion de messages résistant aux pics d'activités du système.

Enfin, nous avons conçu une architecture hybride pour l'implémentation de ce service, décrit dans la partie 5.4. Il nous a paru intéressant de laisser beaucoup de liberté aux entités. Notamment, une entité peut se connecter à n'importe quelle super-entité et peut aisément en changer, voire même accéder au statut de super-entité. De plus, le réseau de super-entités, qui est le cœur du système, peut être soumis à d'importantes modifications, offrant beaucoup de flexibilité au service.

Nous terminons cette présentation par une série de simulations destinées à vérifier la robustesse des algorithmes de diffusion des messages dans un réseau soumis à une forte mobilité. Les résultats confortent notre conviction que ce système, qui n'est pas forcément le plus efficace, offre de grandes garanties de fonctionnement quelle que soit la stabilité de l'environnement.

5.2 Organisation logique multi-couches

Nous avons choisi d'organiser le service en un système constitué de plusieurs couches logiques. Dans un premier temps, nous apportons une définition au concept de couche logique. Puis, nous présentons les caractéristiques du système multi-couches.

5.2.1 Définition d'une couche logique

Un réseau de pairs est un réseau asynchrone soumis à des variations topologiques. Les entités peuvent intégrer ou quitter le système à chaque instant. Une communication entre deux entités admet des délais imprévisibles et peut même être défaillante : les messages peuvent se perdre, être répliqués ou retardés et le lien de communication lui-même peut être rompu à chaque instant.

Nous modélisons l'organisation d'un réseau de pairs par un système logique multi-couches dans lequel chaque couche logique l est un graphe de communication dont les sommets sont certaines entités participant au réseau de pairs. Une entité p est dite *active* à une couche l si au moins une autre entité q, active dans l, connaît p.

Un lien logique entre deux entités p et q dans la couche l peut être dans un des états suivants :

- *up* : les deux entités se connaissent mutuellement dans l ;
- *down* : les deux entités ne se connaissent pas dans l ;
- *forming* : au moins une des deux entités a lancé le processus de création du lien.

Les voisins logiques d'une entité p dans l, notés $\mathcal{N}^l(p)$, sont l'ensemble des entités q telles que le lien logique (p, q) est *up* dans l. Les relations de voisinage dans la couche l définissent la topologie du graphe de la couche l. Typiquement, le voisinage d'une entité dans une couche est un petit sous-ensemble des entités actives dans cette couche.

En conséquence, une couche logique l est caractérisée à l'instant t par :

- les entités actives dans l à l'instant t, noté $V^l(t)$;

– les liens logiques *up* dans l à t, noté $E^l(t)$;
– dans notre modèle, l'orientation logique du graphe de communication $G^l(t) = (V^l(t), E^l(t))$, noté $\mathcal{R}^l(t)$.

5.2.2 Système logique multi-couches

Une entité p peut appartenir à plusieurs couches simultanément et peut donc avoir plusieurs ensembles de voisins. Ainsi, le réseau présenté dans la figure 5.2 possède deux couches logiques. Dans cette figure, les voisines logiques de l'entité 5 à la couche 1 sont les entités 2, 3 et 4, alors que son unique voisine dans la couche 2 est l'entité 1.

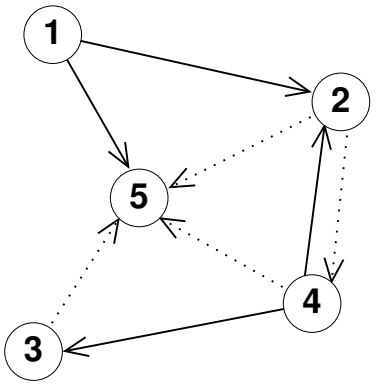

FIG. 5.2 – Réseau logique multi-couches : les liens des couches 1 et 2 sont représentés, respectivement, par des lignes pointillées et pleines.

Les entités exécutent des algorithmes différents sur les différentes couches. L'*état* d'une entité p à la couche l à l'instant t est donné par les valeurs des variables de p et l'état de ses liens logiques. L'état de la couche l, noté $State(V^l(t))$, correspond à l'ensemble des états des entités actives dans l. La *configuration* d'une couche l à l'instant t consiste en son état, le graphe de communication et son orientation logique. Formellement, $c_t^l = \Big(State(V^l(t)), G^l(t), \mathcal{R}^l(t)\Big)$.

La configuration d'un réseau multi-couches à l'instant t est représenté par l'ensemble des configurations des couches existant dans le réseau. On note $\mathcal{L}$ l'ensemble des couches du système. On obtient alors une définition de la configuration du système par $c_t = (c_t^{l_1}, \ldots, c_t^{l_k})$, dans lequel $c_t^{l_i}$ est la configuration de la couche $l_i \in \mathcal{L}$ à l'instant t.

Une transition du système correspond à une modification de la topologie ou à une action interne effectuée par une entité. Une exécution du système est une séquence maximale de transitions.

Nous montrons maintenant que, si chaque couche du système satisfait une propriété $\mathcal{P}$ et si les actions exécutées dans une couche n'ont pas d'influence sur les autres couches, alors le système multi-couches satisfait $\mathcal{P}$.

Définition 5.2.1 (Couches indépendantes deux à deux) *Soit $\mathcal{S}$ un système logiquement organisé par un ensemble $\mathcal{L}$ de couches logiques. Deux couches logiques l_1 et l_2 dans $\mathcal{L}$ sont dites indépendantes deux à deux si aucune action exécutée par une entité dans la couche l_1 n'a d'incidence dans la couche l_2.*

Theorem 5.2.2 *Soit $\mathcal{S}$ un système logique multi-couches constitué d'un ensemble de couches indépendantes deux à deux $\mathcal{L}$. Soit SP^l la spécification de l'algorithme exécuté à la couche $l \in \mathcal{L}$.*

$\mathcal{S}$ satisfait $\wedge_{l \in \mathcal{L}} SP^l$.

Preuve: Soit e une exécution de $\mathcal{S}$. Partons de l'hypothèse que e ne satisfait pas $\wedge_{l \in \mathcal{L}} SP^l$. Alors, il existe au moins une couche $k \in \mathcal{L}$ telle que e ne satisfait pas SP^k.

Soit e_k la projection de l'exécution e sur la couche k. Si e ne satisfait pas SP^k alors e_k ne satisfait pas SP^k, ce qui contredit les hypothèses du théorème. ■

Corollaire 5.2.3 *Soit $\mathcal{S}$ un système logique multi-couches constitué de couches indépendantes deux à deux. Soit SP la spécification des algorithmes exécutés sur chacune des couches du système. Alors, $\mathcal{S}$ satisfait SP.*

L'organisation multi-couches ainsi définie constitue le principal élément du modèle que nous proposons pour le service d'abonné/éditeur. Avant de rentrer dans les détails de ce modèle, nous présentons plusieurs algorithmes nous permettant de construire une orientation de nos graphes de communication. A partir de ces orientations, il sera possible d'offrir des algorithmes de communication pour le service visé.

5.3 Construction et maintien d'un graphe orienté acyclique

Les algorithmes de communication que nous proposons reposent sur des graphes de communication orientés acycliquement. Dans cette partie, nous décrivons les algorithmes qui permettent de construire et de maintenir un graphe de communication acyclique.

L'idée consiste à casser la symétrie des réseaux de pairs en sélectionnant, à chaque instant, quelques entités privilégiées. Ces entités sont autorisées à accomplir des tâches, notamment la publication de nouvelles notifications et la diffusion de notifications. Le but des algorithmes présentés dans la suite est de permettre à chaque entité d'être sélectionnée infiniment souvent.

Chaque entité possède deux variables : une variable entière *lid* et une variable $val \in \{0, 1, 2\}$. Les définitions suivantes visent à créer une orientation pour le graphe de communication.

Définition 5.3.1 *La relation $\prec$ est définie par :*

$$x \prec y \;\Leftrightarrow\; y = (x + 1) \bmod 3$$

Définition 5.3.2 *Soit $G(V, E)$ un graphe de communication. Un arc $(p, q) \in E$ est logiquement orienté de q vers p $(q \rightarrow p)$ si et seulement si :*

$$(val_p \prec val_q) \text{ ou } (val_p = val_q \wedge lid_p < lid_q)$$

Dans un graphe acyclique, il est possible de distinguer deux catégories d'entités : les puits et les entités qui ne sont pas puits.

Définition 5.3.3 (Entité puit) *Soit $G(t)$ un graphe de communication à l'instant t. Une entité p est un puits à l'instant t si :*

$$\forall q \in \mathcal{N}_t(p) : \ q \rightarrow p$$

Les lecteurs attentifs ont remarqué que ces définitions ne garantissent pas que le graphe est acyclique. Considérons, par exemple, trois entités p, q et r et $val_p = 0$, $val_q = 1$ et $val_r = 2$. Dans ce cas-là, les arcs entre ces trois entités sont orientés de telle manière que $p \rightarrow q \rightarrow r \rightarrow p$, ce qui forme un cycle. Dans la suite, nous proposons des algorithmes qui, partant d'une configuration dans laquelle le graphe ne possède pas de cycle, ne cassent pas l'acyclicité du graphe. Nous veillerons donc à ne pas créer de situations dans lesquelles une entité est connectée avec deux entités possédant des valeurs *val* différentes de la sienne et différentes deux à deux.

Le reste de la partie s'attache à permettre à toutes les entités de devenir puits infiniment souvent, dans un contexte dans lequel les entités peuvent, à chaque instant, apparaître, disparaître, ainsi qu'ouvrir et fermer des liens de communication.

5.3.1 Renversement d'orientation des arcs

Une entité puits est une entité privilégiée qui est autorisée à accomplir certaines tâches. Pour des raisons de vivacité, il est impératif que chaque entité puisse devenir puits infiniment souvent. Il faut donc que, dès l'accomplissement de ses tâches, une entité puits cesse d'être privilégiée et permette à d'autres entités de devenir puits. Pour cela, nous reprenons le principe décrit dans [13] en proposant un algorithme dans lequel un puits renverse l'orientation de ses arcs de manière à cesser d'être puits.

Nous présentons tout d'abord l'algorithme pour une seule couche (Algorithme 13), puis nous étendons le mécanisme à un système multi-couches (Algorithme 14).

L'idée de l'algorithme 13 est simple. L'ensemble des voisins $\mathcal{N}(p)$ d'un puits p est constitué de l'union de (1) l'ensemble $\mathcal{G}(p)$ des entités $q \in \mathcal{N}(p)$ telle que $val_p = val_q \wedge lid_p < lid_q$ et (2) l'ensemble $\mathcal{F}(p)$ des entités $q \in \mathcal{N}(p)$ vérifiant $val_p \prec val_q$.

Pour renverser l'orientation des arcs reliant p aux entités de $\mathcal{G}(p)$, il suffit à p de modifier val_p de manière à obtenir $val_p = (val_p + 1) \bmod 3$. En revanche, cette modification peut s'avérer insuffisante pour renverser les arcs avec les entités dans $\mathcal{F}(p)$. En effet, si une entité $q \in \mathcal{F}(p)$ vérifie $lid_p < lid_q$, l'arc demeurera orienté vers p. Il est alors nécessaire de modifier également lid_p de manière à ce que lid_p soit supérieur à $\max_{q \in \mathcal{F}(p)}(lid_q)$. Cette modification peut avoir un impact important sur le système puisque cela peut provoquer une augmentation ininterrompue de la variable *lid*. Pour cette raison, nous réajustons la variable *lid* à 0 dès que c'est possible, c'est-à-dire dès que l'ensemble $\mathcal{F}(p)$ est vide.

L'Algorithme 13 décrit ce mécanisme. Après l'exécution de l'algorithme, le puits p possède une variable val_p qui vérifie $\forall q \in \mathcal{G}(p), var_q \prec var_p$, donc tous les arcs avec les entités de $\mathcal{G}(p)$ sont orientés vers ces entités. Et, s'il possède des voisins dans $\mathcal{F}(p)$, il partage la même valeur pour la variable *val* mais il possède une variable lid_p qui est supérieure aux *lid* des entités de $\mathcal{F}(p)$, donc tous les arcs vers les entités de $\mathcal{F}(p)$ sont orientés vers ces entités.

Nous proposons dans la suite une analyse de la principale caractéristique de cet algorithme.

Algorithme 13 Renversement d'orientation (puits p)

Paramètres :

$\mathcal{N}(p)$: ensemble des voisins de p ;
$val_i \in \{0, 1, 2\}$: valeur de la variable val de l'entité $i \in \mathcal{N}(p)$;
lid_i : valeur de la variable entière lid de l'entité $i \in \mathcal{N}(p)$;

Fonctions :

$choose\_id()$: retourne $\max_{j \in \mathcal{N}(p): val_i \prec val_j}(lid_j) + 1$

Actions :

$val_p = (val_p + 1) \bmod 3$;
si $(\exists q \in \mathcal{N}(p), val_p \prec val_q)$ **alors**
    $lid_p = choose\_id()$;
sinon
    $lid_p = 0$

Lemma 5.3.4 *Soit $G(t)$ un graphe de communication à l'instant t et $G(t+1)$ le graphe de communication après l'exécution de l'algorithme 13 par une entité puits. Si $G(t)$ est acyclique, alors $G(t+1)$ est acyclique.*

Preuve: Si le graphe $G(t)$ est acyclique et le graphe $G(t+1)$ possède un cycle, cela signifie que l'exécution de l'algorithme 13 par un puits p a créé un cycle. Autrement dit, le renversement d'arcs de p a permis la construction d'un cycle. Comme les autres arcs n'ont pas subi de renversement, il est sûr que p participe à ce cycle. Soit $p \to p_1 \to \ldots \to p_i \to p$ ce cycle.

Après exécution de l'algorithme, tous les arcs concernant p sont orientés vers les voisins de p. Il n'existe donc pas d'arcs $p_i \to p$. Par conséquent, aucun cycle ne peut apparaître. ∎

L'algorithme 14 généralise le mécanisme de renversement des arcs pour un système multi-couches.

Algorithme 14 Renversement d'orientation dans un système multi-couches (entité p)

Paramètres :

$\mathcal{L}$: l'ensemble des couches auxquelles l'entité p appartient ;
$Val = \{val_p^l \in \{0, 1, 2\} \mid l \in \mathcal{L}\}$
$Lid = \{lid_p^l \mid l \in \mathcal{L}\}$
$\mathcal{N} = \{\mathcal{N}^l(p) \mid l \in \mathcal{L}\}$: l'ensemble des voisins de p pour chaque couche dans $\mathcal{L}$;

Fonctions :

$sink^l(p)$: retourne **vrai** si p est puits à la couche l ;

Actions :

si $\exists\, l \in \mathcal{L}, sink^l(p)$ **alors**
    exécute l'Algorithme 13 avec les paramètres val_p^l, lid_p^l et $\mathcal{N}^l(p)$

5.3.2 Création d'un nouvel arc

De manière évidente, la disparition d'un arc n'a aucun impact sur l'acyclicité d'un graphe. En revanche, lorsqu'un nouvel arc est créé, il existe de forts risques que ce nouvel arc engendre un cycle. Il est donc nécessaire de disposer d'un algorithme qui assure que la création d'un nouvel arc ne brise pas l'acyclicité.

Les entités appartenant à une même couche l peuvent se transmettre, à tout instant, trois types de messages :

- $<$*request_link*$>$: ce message permet à une entité p de notifier à une autre entité q le désir de créer un arc entre elles deux. Les deux entités considèrent l'arc (p, q) comme étant dans l'état *forming*. Les entités s'échangent les informations sur leurs variables respectives, mais ne prennent pas en compte cet arc pour les autres tâches, notamment pour déterminer si elles sont devenues puits ;
- $<$*link_up*$>$: ce message est l'une des deux réponses possibles à un message $<$*request_link*$>$. Il permet à l'une des deux entités p ou q, liées par un arc dans l'état *forming*, de spécifier qu'elle considère maintenant l'arc comme étant *up* ;
- $<$*link_down*$>$: ce message est l'autre réponse possible à un message $<$*request_link*$>$. Dans ce cas, l'une des deux entités considère qu'il n'est pas possible de créer l'arc sans créer de cycles et décide donc de ne pas donner suite à la demande de création d'arc.

A noter que nous utilisons le verbe *répondre* abusivement puisque l'entité p qui a initiée la demande de création de l'arc peut être l'émettrice d'un des messages $<$*link_up*$>$ ou $<$*link_down*$>$.

L'idée du mécanisme de création d'arc est très simple. Seuls les puits sont en mesure de répondre, positivement ou négativement, à un message $<$*request_link*$>$. Un puits p connaît l'état des variables val_q et lid_q de l'entité q s'il existe un arc (p, q) dans l'état *forming*. Il peut donc aisément déterminer l'orientation de (p, q). Si l'arc (p, q) est effectivement orienté vers p, le puits p répond par un message $<$*link_up*$>$, sinon, il répond par un message $<$*link_down*$>$.

Algorithme 15 Création de nouveaux arcs (puits p)

Paramètres :
$\mathcal{N}(p)$: ensemble des voisins de p ;
$\mathcal{O}(p)$: ensemble des entités q avec (p, q) dans l'état *forming* ;

Fonctions :
$send(msg, q)$: envoie le message msg à l'entité q ;

Actions :
pour tout $q \in \mathcal{O}(p)$:
  si $(val_p \prec val_q$ **ou** $(val_p = val_q \wedge lid_p < lid_q))$ **alors**
    $send(<$*link_up*$>, q)$
  sinon
    $send(<$*link_down*$>, q)$
$\mathcal{O}(p) = \emptyset$

L'algorithme 15 décrit le comportement d'un puits p. Cette entité peut avoir reçu

plusieurs messages <*request_link*> depuis son dernier renversement d'arcs. Toutes les entités qui ont émis ces messages sont stockées dans l'ensemble $\mathcal{O}(p)$. Quand p devient puits, il vérifie, pour toutes les entités dans $\mathcal{O}(p)$, la possibilité de créer un arc et répond immédiatement.

Lemma 5.3.5 *Soit $G(t)$ un graphe de communication à l'instant t et $G(t+1)$ le graphe après l'exécution de l'Algorithme 15 par une entité puits. Si $G(t)$ est acyclique, alors $G(t+1)$ est acyclique.*

Preuve: Si le graphe $G(t)$ est acyclique et $G(t+1)$ possède un cycle, cela signifie que l'arc (p, q), créé par le puits p, a engendré un cycle.

Deux cycles différents peuvent apparaître. Le premier se représente par $p \to q \to p_1 \to \ldots \to p_i \to p$. Mais, l'arc $p \to q$ ne peut pas exister puisque p a répondu par un message <*link_up*> uniquement si (p, q) est orienté vers p. Ce cycle est donc impossible.

Le deuxième cycle possible est $q \to p \to p_1 \to \ldots \to p_i \to q$. Mais, l'arc $p \to p_i$ ne peut pas exister puisque p est puits. Ce cycle est donc impossible également. ■

A noter qu'il est possible d'imaginer une variante de cet algorithme dans lequel le puits ne répondrait pas de messages <*link_down*> si la création du lien n'est pas autorisée. En effet, il est possible que, quand p est puits, la création du lien soit impossible, mais que cette opération soit envisageable lorsque q devient puits.

5.3.3 Connexion à une couche

Dans cette partie, nous nous intéressons au mécanisme permettant à une entité d'intégrer une couche logique. Il s'agit de permettre à cette entité de se connecter au graphe de communication sans briser l'acyclicité de son orientation. Nous considérons une entité p désirant se connecter à une couche $l \in \mathcal{L}$. Comme dans la plupart des algorithmes de connexion pour les réseaux de pairs, nous admettons que l'entité p connaît une entité p_0 active dans la couche l.

Le but de l'algorithme 16 consiste à permettre à l'entité p de déterminer les valeurs de ses variables val_p^l et lid_p^l et de posséder au moins un lien logique *up* avec une entité active dans l. L'entité p lance l'algorithme en envoyant un message <*request_val*> à l'entité p_0. L'algorithme se termine quand p reçoit un message <*respond_val, val, lid*> et un message <*link_up*>.

L'idée de l'algorithme est la suivante. Le message <*request_val*> est transmis, de proche en proche, jusqu'à un puits p_i. Quand p_i reçoit le message, il est capable de déterminer des valeurs val_p et lid_p telles que l'arc (p, p_i) est orienté vers lui. Par exemple, il peut choisir $val_p = (val_{p_i} + 1) \bmod 3$ et $lid_p = 0$. Il peut alors spécifier, dans son message de réponse, ces valeurs ainsi que le message de création du lien.

Pour faire suivre le message jusqu'à une entité puits, une méthode consiste à sélectionner au hasard une entité parmi les voisins vers lesquels l'arc est orienté et à lui transmettre le message. L'algorithme 16 décrit le traitement effectué par une entité i recevant un message <*request_val*> initialement transmis par une entité p.

Il serait malvenu de ne proposer qu'un seul arc à l'entité qui se connecte. En effet, si cet arc disparaît alors que p n'a pas eu le temps d'ouvrir de nouveaux arcs, il serait contraint de relancer l'algorithme de connexion. Pour remédier à ce problème, nous

Algorithme 16 Réception d'un message $m =$<*request_val*> issu d'une entité p (entité i)

Paramètres :
$\mathcal{N}(i)$: ensemble des voisins de i ;

Fonctions :
choose_neighbor() : retourne une entité $q \in \mathcal{N}(i)$ telle que $i \rightarrow q$;
$send(msg, q)$: envoie le message msg à une entité q ;
puit() : fonction booléenne vraie si i est puit ;

Actions :
si *puit*() **alors** :
$send(<respond\_val,\ (val_i + 1) \bmod 3,\ 0>, p)$
sinon
$q = choose\_neighbor()$
$send(m, q)$
$send(<request\_link>, p)$

proposons que toutes les entités par lesquelles le message <*request_val*> transite envoie un message <*request_link*> à p. Ce mécanisme apparaît dans la dernière ligne de l'algorithme 16.

Ce principe présente deux avantages. Tout d'abord, nous observons qu'il renforce la connexité du graphe de communication et qu'il réduit le diamètre du graphe. En effet, il est probable que p parvienne à activer quelques arcs *forming* récupérés sur le chemin. Considérons par exemple qu'un arc (p, p_j) passe rapidement de l'état *forming* à l'état *up*. Si l'arc (p, p_j) était dans l'état *forming*, cela signifie que le message <*request_val*> est passé par p_j avant d'atteindre p_i - l'entité qui permet à p de devenir active dans la couche -. Il existait donc déjà un chemin $p_j \rightarrow p_{j+1} \rightarrow \ldots \rightarrow p_i$. Or, l'entité p est maintenant voisine de l'entité p_i et de l'entité p_j : l'arrivée de p a donc permis la création d'un chemin redondant entre p_j et p_i. Les nouvelles connexions ont donc tendance à renforcer la connexité du graphe de communication et à réduire le diamètre du graphe.

Le second avantage vient du fait que le seul arc que possède l'entité p après la connexion est orienté vers l'entité puits p_i. Dès que p_i a exécuté ses tâches, il renverse ses arcs. Comme p ne possède qu'un seul arc, il est immédiatement puits. Il est donc en mesure, rapidement, d'ouvrir de nouveaux arcs avec les entités qu'il a découvertes. En conséquence, l'entité p a de fortes probabilités de posséder rapidement plusieurs arcs dans cette couche.

A noter, enfin, que cet algorithme de création ne peut pas briser l'acyclicité du graphe puisque les arcs créés lors de la connexion suivent les spécifications de l'algorithme 15.

5.3.4 Équité entre entités

Dans les parties précédentes, nous avons présenté des algorithmes qui permettent à un ensemble d'entités de s'organiser dans un graphe orienté acycliquement. Nous avons décrit un algorithme qui permet aux entités puits de renverser l'orientation de leurs arcs. Il a été prouvé que, dans un réseau synchrone dans lequel tous les puits renversent leurs arcs simultanément, quelle que soit la configuration initiale du graphe, toutes les entités

deviennent puits *le même nombre de fois sur une période* [13].

Dans le contexte que nous considérons, les communications peuvent admettre des délais variables et certaines entités sont plus rapides que d'autres. Un élément d'un sous-ensemble connecté d'entités très rapides peut constater des renversements d'arcs très rapides dans son entourage, et donc une période temporelle réduite entre deux moments où il est puits. En revanche, une entité proche d'une entité très lente doit attendre que cette entité très lente renverse ses arcs avant de devenir puits de nouveau. Cela pose un problème d'équité car la situation de puits est privilégiée puisqu'elle permet de réaliser les tâches.

Soient n_p et n_q le nombre de fois que les entités, respectivement p et q renversent leurs arcs dans une période. Nous montrons dans la suite que, dans un réseau asynchrone, $n_p \leq n_q \leq k * n_p$ avec k fini.

Lemma 5.3.6 *Soit p et q deux entités puits, distants de d hops, dans la configuration c_i. L'entité p ne peut pas être puits plus de $d-1$ fois avant que q ne renverse ses arcs.*

Preuve: Nous prouvons ce lemme par récursivité sur le nombre de hops (sauts).

Admettons que les deux entités sont distantes de 2 hops. Soit p_0 l'entité telle que, à c_i, nous avons $p \leftarrow p_0 \rightarrow q$. L'entité p, très rapide, renverse ses arcs. Dans la configuration c_{i+1}, nous avons $p \rightarrow p_0 \rightarrow q$. Pour que p puisse devenir puits de nouveau, il doit attendre que p_0 soit puits. Or, p_0 ne sera pas puits tant que q n'aura pas renversé ses arcs. En conséquence, p ne peut pas être puits plus d'une fois avant que q ne renverse ses arcs. Le lemme est montré pour $d = 2$.

Supposons maintenant que le lemme soit prouvé pour $d = n$. Plaçons-nous dans la situation dans laquelle p et q sont à $n+1$ hops de distance. Dans la configuration c_i, nous avons $p \leftarrow p_0 \ldots p_i \ldots p_n \rightarrow q$. L'entité p renverse ses arcs et, à la configuration c_{i+1}, nous avons $p \rightarrow p_0 \ldots p_i \ldots p_n \rightarrow q$.

Par l'hypothèse, il est sûr que l'entité p_0 ne peut pas être puits plus de $n-1$ fois avant que q ne renverse ses arcs. A chaque fois que p_0 renverse ses arcs, p devient puits. L'entité p peut donc être puits $1 + (n-1)$ fois. En conséquence, si p et q sont distants de $n+1$ hops, p peut être puits au mieux n fois avant que q renverse ses arcs. ■

Theorem 5.3.7 *Dans un réseau asynchrone, un puits p ne peut pas renverser plus de k fois ses arcs avant qu'un autre puits q ne renverse les siens.*

Preuve: Nous notons δ le diamètre du graphe de communication. Avec le lemme 5.3.6, nous savons que, si deux puits p et q sont à une distance d, l'entité p ne peut pas renverser ses arcs plus de $d-1$ fois avant que q ne renverse les siens. Le pire cas consiste en un éloignement maximal des entités p et q, c'est-à-dire une distance δ. Il est impossible qu'un puits p puisse renverser ses arcs plus de $k = \delta - 1$ fois avant que q ne renverse ses arcs. ■

Ce théorème prouve deux propriétés importantes : chaque entité sera puits à un instant futur et les entités seront puits infiniment souvent.

Dans cette partie, nous avons décrit des algorithmes distribués qui permettent à des entités de construire un graphe orienté acycliquement. Ce graphe nous permet de sélectionner certaines entités privilégiées, les puits. Nous avons présenté des algorithmes qui permettent

à chaque entité de devenir puits infiniment souvent. Puis, nous avons montré que l'orientation de notre graphe peut conserver ses propriétés lorsque de nouvelles entités apparaissent et de nouveaux liens de communication se forment.

Nous allons utiliser ces algorithmes pour proposer un mécanisme de diffusion de l'information. Un des risques apparaissant lors de la mise en œuvre d'applications d'abonné/éditeur concerne la surcharge ponctuelle du réseau. En effet, les éditeurs peuvent produire des notifications à n'importe quel moment, et notamment, un grand nombre d'éditeurs peut produire des notifications simultanément. Cela peut provoquer des importants pics d'utilisation des ressources du réseau, nuisant aux performances des machines. En forçant les entités à publier leurs notifications lorsqu'elles sont puits, nous garantissons que, dans leurs voisinage, ces entités seront les seules à effectuer une publication. De fait, la charge du réseau reste tolérable à chaque instant.

5.4 Implémentation d'un système abonné/éditeur

Cette partie présente le système d'abonné/éditeur basé sur le contenu. Rappelons tout d'abord les termes qui seront utilisés dans la suite.

Une entité émet des *intérêts* relatifs au contenu d'informations appelées des *notifications*. Les intérêts s'expriment par des *attributs*, des opérateurs et des valeurs. Une entité s'*abonne* à un attribut lorsqu'elle émet un intérêt portant sur un attribut. Quand une entité reçoit une notification qui *vérifie* un intérêt émis, on dit que cette entité *consomme* la notification. Le producteur d'une notification est un *éditeur*.

Après l'émission d'un intérêt, il peut se passer un certain temps avant que le message lié à cet intérêt soit reçu et pris en compte par le système. Un intérêt est *stable* quand les éléments du système sont notifiés de cet intérêt. Nous notons également par T_{diff} le temps mis par le système pour diffuser une information, *i.e* la vérification de l'adéquation entre la notification et les intérêts, l'envoi de la notification aux abonnés et la consommation de la notification.

A partir de ces définitions, les propriétés du service d'abonné/éditeur que nous visons peuvent se réduire à :

Adéquation : si une entité p consomme une notification e, alors p a préalablement émis un intérêt f tel que e vérifie f.

Validité : si une entité p consomme une notification e, alors il existe une entité qui a préalablement publié e.

Équité : chaque entité peut publier infiniment souvent.

Existence : une entité p sera amenée à consommer une notification e, si elle a émis un intérêt qui vérifie e et si cet intérêt est stable T_{diff} unités temporelles après la publication de e.

5.4.1 Aperçu de l'organisation

Le système que nous avons conçu est un système hybride, dans lequel nous distinguons deux types d'entités : les super-entités et les entités.

Chaque entité est connectée à une et une seule super-entité[1]. Les super-entités sont reliées entre elles par un réseau de pairs pur. Elles forment le cœur du service. Une super-entité p est connectée avec un ensemble d'entité $\mathcal{C}(p)$ dans une relation client-serveur. Elle est chargée (1) de recueillir les intérêts des entités de $\mathcal{C}(p)$, (2) de publier les notifications des entités de $\mathcal{C}(p)$ aux autres super-entités et (3) de délivrer aux entités de $\mathcal{C}(p)$ les notifications reçues.

Le réseau des super-entités est formé d'une collection de couches $\mathcal{L}$. Chaque couche logique est un graphe de communication orienté acycliquement. Il existe autant de couches qu'il existe d'attributs dans les intérêts des entités. A chaque couche $l \in \mathcal{L}$ est associé un attribut. Une super-entité p est *active* dans toutes les couches associées aux attributs intéressant les entités de $\mathcal{C}(p)$. On note $\mathcal{L}(p)$ l'ensemble des couches dans lesquelles p est active.

De plus, toutes les super-entités sont actives dans une couche spéciale notée l_s. Cette couche permet aux super-entités de rechercher un point d'accès à une couche existante. Une super-entité désirant intégrer une couche l peut ainsi découvrir une super-entité active dans l. Par ailleurs, les annonces concernant la création d'une nouvelle couche, associée à un nouvel attribut, sont également diffusées dans cette couche l_s à laquelle toutes les super-entités doivent participer.

Dans la suite, nous expliquons le fonctionnement du réseau de pairs pur des super-entités, puis les relations entre les entités et les super-entités.

5.4.2 Communication entre super-entités

Dans cette partie, nous nous attachons à décrire les communications entre les super-entités. Celles-ci utilisent l'organisation multi-couches en graphes orientés acycliquement telle que nous l'avons décrite dans les parties précédentes.

Pour plus de simplicité, nous considérerons qu'une super-entité p et l'ensemble $\mathcal{C}(p)$ des entités qui dépendent de p forment ce que nous appelons un *sommet*. Dans la section 5.4.3, nous décrivons plus en détail les relations dans un sommet. Abusivement, le sommet lié à la super-entité p est noté p et toutes les notifications, issues de p ou de n'importe quelle entité de $\mathcal{C}(p)$, sont attribuées à p.

Dans la suite, nous expliquons le fonctionnement de la diffusion des notifications dans une couche $l \in \mathcal{L}(p)$, l'émission de nouvelles notifications dans le système multi-couches, puis les mécanismes permettant d'intégrer une nouvelle couche et, enfin, certaines optimisations possibles.

Diffusion des notifications dans une couche

L'organisation d'un graphe de communication en graphe orienté acycliquement garantit l'existence d'au moins un sommet puits, à n'importe quel instant. Les sommets puits sont privilégiés. Il leur est accordé l'autorisation d'émettre des notifications et de faire suivre des notifications reçues. Seuls les puits sont des points de diffusion, ce qui permet d'éviter des pics de trafic dans le réseau.

[1] Un mécanisme permettant à une entité de ne pas être exclue du système si sa super-entité se déconnecte subitement est envisageable. Par exemple, la super-entité pourrait fournir aux entités connectées à elle des informations concernant k autres super-entités, où k dépend de la robustesse désirée de l'application.

Un sommet peut à chaque instant éditer une nouvelle notification. Dans cette explication, nous nous restreignons à une seule couche logique, et, par simplicité, nous considérons ici uniquement les notifications éditées dans la couche l.

Un sommet p possède trois mémoires tampon :

- $Old(p)$ contient toutes les notifications émises par le sommet p depuis qu'il est actif dans la couche l ;
- $New(p)$ contient toutes les notifications reçues par le sommet p depuis le dernier moment où p a été puits ;
- $Local(p)$ contient toutes les notifications créées par le sommet p depuis le dernier moment où p a été puits.

Quand un sommet p reçoit une notification m, il vérifie tout d'abord que m n'apparait pas déjà dans la mémoire $New(p)$, puis dans la mémoire $Old(p)$. Si m a déjà été sauvegardé dans une de ces mémoires, la réception de m est redondante et aucun traitement n'est effectué. En revanche, si m est une nouvelle notification, elle est immédiatement sauvegardée dans la mémoire $New(p)$.

L'algorithme 17 décrit le traitement effectué par le sommet p lorsqu'il devient puits. Il doit diffuser les notifications qu'il désire éditer (contenues dans la mémoire $Local(p)$) et les notifications qu'il est en charge de diffuser plus largement (contenues dans la mémoire $New(p)$). Une fois qu'il a émis toutes ces notifications, il les consigne dans la mémoire $Old(p)$ et initialise les mémoires $New(p)$ et $Local(p)$

Algorithme 17 Diffusion de notifications (puits p)

Paramètres :
$\mathcal{N}(p)$: l'ensemble des voisins de p ;

Fonctions :
$send(m)$: envoie la notification m à toutes les entités dans $\mathcal{N}(p)$;

Actions :
pour tout $m \in New(p)$:
$send(m)$
$Old(p) = Old(p) \cup \{m\}$
pour tout $m \in Local(p)$:
$send(m)$
$Old(p) = Old(p) \cup \{m\}$
$New(p) = \emptyset$
$Local(p) = \emptyset$

Avec cet algorithme simple, le sommet p est capable non seulement de participer à la diffusion des notifications, mais aussi d'émettre une nouvelle notification.

Connexion à une nouvelle couche

Lorsqu'un sommet p désire s'abonner à un nouvel intérêt a, il peut intégrer la couche $l \in \mathcal{L}$ s'il existe une couche l associée à l'attribut a, ou créer une nouvelle couche l si l'attribut a est un nouvel attribut.

Le sommet p est informé de l'ensemble des couches existantes, et des attributs associés, grâce aux informations diffusées dans la couche l_s. Il peut donc facilement savoir s'il existe une couche l associée à l'attribut a.

Si, effectivement, il existe une telle couche l, il doit alors détecter un sommet q, actif dans la couche l, afin que q puisse lui servir de point d'accès à la couche l. Il réalise cette opération en émettant une notification spéciale dans la couche l_s. Cette notification s'exprime par un message $<search(p, l)>$ indiquant que p recherche un point d'accès pour la couche l.

Lorsqu'un sommet q reçoit, par la couche l_s, un tel message, il participe à sa diffusion s'il n'est pas actif dans l ou il décide de répondre directement à p par un message spécifique $<respond(l, q)>$ s'il est actif dans cette couche. Ainsi, p reçoit des informations sur les sommets actifs dans la couche l et il peut les utiliser pour intégrer cette couche grâce à l'algorithme de connexion décrit dans la section 5.3.3.

Si p ne reçoit aucune réponse à son message $<search(p, l)>$ ou s'il n'existe pas de couche l associée à l'attribut a, le sommet p crée une nouvelle couche l. Il doit alors informer l'ensemble des sommets du système grâce à un message spécifique $<create(l, a)>$ indiquant qu'une nouvelle couche l est créée et que l'attribut associé à cette couche est a. Ce message est émis est diffusé dans la couche l_s. Ainsi, tous les sommets du système sont informés de l'existence de cette nouvelle couche.

Émission des notifications dans le système multi-couches

Un sommet peut aisément diffuser une notification lorsqu'il est membre de la couche associée à l'attribut de cette notification. Malheureusement, il est possible qu'un sommet p souhaite éditer une nouvelle notification m portant sur un attribut a tel que la couche l associée à l'attribut a n'appartienne pas à $\mathcal{L}(p)$.

Dans ce cas, le sommet p doit trouver un autre sommet q en mesure de diffuser cette notification. Cette recherche est possible grâce à un message spécifique $<search\_sender(p, l)>$ qui est diffusé dans la couche l_s.

Lorsqu'un sommet q reçoit un tel message, il participe à la diffusion de cette information s'il n'est pas actif dans la couche l, ou il répond directement à p par un message $resp\_sender(l, q)$ s'il est actif dans la couche l.

Quand p reçoit un message $<resp\_sender(l, q)>$, il sait que q peut se charger de diffuser la notification m dans la couche associée à l'attribut de m. Il envoie alors directement m à q qui pourra émettre la notification dès qu'il sera puits dans cette couche.

Si p ne reçoit aucune réponse à son message $<search\_sender(p, l)>$, ou si aucune couche n'est associée à l'attribut de la notification m, cela signifie qu'aucun sommet n'est intéressé par cette notification. Naturellement, aucun traitement n'est effectué puisqu'aucun abonné n'a exprimé d'intérêt pour cet attribut.

Optimisations

La couche l_s contient tous les sommets du système, par conséquent les messages émis dans cette couche sont diffusés à tous les sommets. Le nombre de messages circulant peut devenir très important, engendrant un coût de traitement qui peut dépasser les capacités

physiques des sommets. Il est donc primordial de restreindre autant que possible l'émission de messages dans la couche l_s.

Nous pouvons imaginer la méthode proactive suivante : à intervalle régulier, tous les sommets échangent avec leurs voisins dans l_s la liste des couches dans lesquels ils sont actifs. Ainsi, à chaque instant, un sommet p connaît l'ensemble des couches $L(p) = \mathcal{L}(q), \forall q \in \mathcal{N}^{l_s}(p)$. Dès lors, avant d'émettre dans la couche l_s un message $<search(p,l)>$ ou un message $<search\_sender(p,l)>$, le sommet p effectue une recherche dans $L(p)$. Si l apparaît dans $L(p)$, cela signifie qu'un de ses voisins immédiats est actif dans cette couche. L'émission des messages n'a alors plus d'intérêt et p peut contacter directement le sommet actif dans l. Cette méthode permet de réduire le nombre de messages diffusés dans l_s.

De plus, il est possible d'envisager un mécanisme qui augmente la probabilité, pour un sommet p, qu'il existe effectivement un sommet voisin actif dans une couche dans laquelle p n'est pas actif. Pour cela, nous proposons que, régulièrement, chaque sommet décide de supprimer une connexion avec un de ses voisins et récupère un nouveau voisin. Le choix du voisin à supprimer est réalisé de manière à ce que l'ensemble $L(p)$ soit le plus vaste possible. L'algorithme 18 décrit ce mécanisme. L'entité p compte le nombre de couches pour chaque sous-ensemble $\mathcal{N}(p)$ privé d'un voisin. Le voisin choisi est l'entité q tel que $L(p)$ avec $\mathcal{N}(p) \setminus \{q\}$ est l'ensemble contenant le plus de couches. Cela signifie que les couches auxquelles q est abonné sont, soit les mêmes que d'autres voisins, soit peu nombreuses.

Algorithme 18 Politique de suppression d'un voisin (sommet p)

Paramètres :
$\mathcal{N}(p)$: l'ensemble des voisins de p ;
$\mathcal{L}(p)$: l'ensemble des couches dans lesquelles q est actif ;

Fonctions :
$close(q)$: ferme la connexion de p avec le voisin q ;

Actions :
$L(p) = \mathcal{L}(p) \cup \left(\bigcup_{q \in \mathcal{N}(p)} \mathcal{L}(q) \right)$;
$min = 0$;
pour tout $q \in \mathcal{N}(p)$:
    $TempL = L(p) \setminus \mathcal{L}(q)$:
    si $|TempL| > min$ **alors**
        $min = |TempL|$
        $choix = q$
$close(choix)$;

Grâce à cet algorithme, chaque sommet cherche à être connecté avec des voisins de telle manière que le nombre de couches, dans lesquelles eux-mêmes et leurs voisins sont actifs, soit le plus important possible. Ainsi, le nombre de messages à diffuser dans la couche l_s peut être considérablement réduit.

Dans cette section, nous avons présenté le comportement d'un sommet dans l'organisation multi-couches de graphes orientés acycliquement. Nous avons vu qu'un sommet est capable d'éditer de nouvelles notifications, que ces notifications sont largement diffusées

dans les couches correspondant aux attributs sur lesquelles elles portent et, enfin, qu'un sommet reçoit toutes les notifications portant sur des attributs pour lesquels il a émis un intérêt.

5.4.3 Relations entre les super-entités et les entités

Tous les algorithmes que nous avons présentés dans la section 5.4.2 sont exécutés par un sommet que nous avons défini comme étant une super-entité et l'ensemble des entités dépendantes de cette super-entité. Nous allons maintenant décrire la répartition de charge dans un sommet.

Fonctionnement général

Chaque entité est susceptible de devenir éditeur à chaque instant. Une entité $q \in \mathcal{C}(p)$ désirant éditer une notification m se contente d'envoyer m à sa super-entité p. La super-entité p est responsable de la diffusion de cette notification grâce aux algorithmes d'émission de notifications. Si p est actif dans les couches associées aux attributs de m, il ajoute m aux mémoires $Local(p)$ de toutes ces couches. S'il existe une couche l dans laquelle p n'est pas actif, il utilise alors une autre super-entité active dans l pour émettre la notification.

Une entité peut émettre un intérêt pour un nouvel attribut à chaque instant. De la même manière, une entité $q \in \mathcal{C}(p)$ s'intéressant à un nouvel attribut a envoie cet intérêt à sa super-entité p. Si p est déjà actif dans la couche l associée à l'attribut a, aucun traitement n'est réalisé. Si, au contraire, p n'est pas actif dans cette couche, il décide alors d'intégrer cette couche de manière à pouvoir transmettre à q les notifications liées à cet attribut.

Quand une super-entité p reçoit une notification m dans la couche l, il détermine immédiatement l'ensemble $\mathcal{C}_l(p) \subseteq \mathcal{C}(p)$ des entités qui ont émis un intérêt portant sur l'attribut associé à la couche l. Ensuite, il doit vérifier, pour toutes les entités q de $\mathcal{C}_l(p)$, si l'intérêt émis par q est en adéquation avec la notification reçue. Si c'est le cas, p transmet alors m à q.

Optimisations

L'optimisation repose principalement sur deux points. Le premier concerne la sélection des entités amenées à devenir des super-entités. Le second est lié à l'attribution des entités aux super-entités.

Il est préférable qu'une super-entité soit une entité qui a émis des intérêts pour un grand nombre d'attributs. En effet, si la super-entité est active dans un grand nombre de couches du fait même de ses propres intérêts, il est probable que les entités qui dépendent d'elle ont émis des intérêts portant sur les mêmes attributs que leur super-entité. Ainsi, il est plus probable que les nouveaux abonnements n'obligeront pas la super-entité à intégrer de nouvelles couches.

Par ailleurs, autant que possible, une super-entité doit réduire le nombre de couches dans lesquelles elle est active. Il est donc important que les entités d'un même sommet aient émis beaucoup d'intérêts communs. En effet, le coût d'appartenance à une couche est

moins rentable si le nombre d'entités intéressées par les notifications liées à cette couche est faible. De fait, les entités abonnées à des attributs différents des autres entités d'un même sommet doivent, dans la mesure du possible, trouver une super-entité plus adaptée.

Trois scénarios sont possibles : la disparition soudaine d'une super-entité, la promotion d'une entité au rang de super-entité et le transfert d'une entité d'un sommet à un autre. Le premier cas admet une solution simple. Quand une super-entité p quitte le système, il est préférable qu'une des entités de $\mathcal{C}(p)$ devienne super-entité. Naturellement, le choix de l'entité se fera en fonction du nombre d'attributs différents apparaissant dans ses intérêts. Les autres cas de figure apparaissent quand une super-entité est surchargée. Il faut alors réduire le trafic qui lui parvient, soit en sélectionnant une entité afin qu'elle devienne super-entité, soit en transférant une entité à une autre super-entité. Dans ces deux cas, la sélection de l'entité se fait en prenant comme critère le nombre d'attributs qui ne sont pas partagés par d'autres entités du sommet.

5.5 Simulations

Le but des simulations que nous avons effectuées est l'observation de l'impact de la mobilité sur le graphe orienté acycliquement tel que nous l'avons décrit dans la partie 5.3. Pour cela, nous avons conçu un réseau dans lequel, à chaque instant, des nouvelles entités peuvent apparaître tandis que d'autres peuvent disparaître, les connexions entre les entités peuvent être supprimées ou créées. A chaque pas d'exécution i, nous introduisons la mobilité en ajoutant à l'ensemble des entités V_i une nouvelle entité, ou en choisissant au hasard une entité dans V_i et en la forçant soit à disparaître, soit à modifier ses connexions avec son voisinage. Chaque entité p possède un nombre souhaité de voisins $e(p)$. Si le nombre de voisins de p au pas i est inférieur à $e(p)$, elle choisit alors au hasard une entité parmi les voisins de ses voisins et entame la procédure de création d'arcs avec lui. Au contraire, si le nombre de voisins est supérieur à $e(p)$, l'entité p supprime alors un arc avec un de ses voisins choisi aléatoirement.

Nous divisons une exécution en périodes de x pas (par défaut $x = 300$). A chaque pas i, nous mesurons $\Delta^i(p)$, le nombre de fois qu'une entité p est devenue puits pendant la période $[(i-x)\dots i]$. Évidemment, nous ne considérons que les entités qui ont appartenu à V_j pour tout j dans $[(i-x)\dots i]$. Nous sommes plus particulièrement intéressés par les valeurs Δ^i_{max} et Δ^i_{min} définies par $\Delta^i_{max} = \max\{\Delta^i(p) : p \in V_i\}$ et $\Delta^i_{min} = \min\{\Delta^i(p) : p \in V_i\}$.

Nous considérons dans ces simulations que les communications sont synchrones. Un message émis au pas i est reçu par le destinataire au pas $i+1$. Enfin, à chaque pas i, *toutes* les entités qui sont puits exécutent les algorithmes de création d'arcs, de diffusion et de renversement d'arcs. Cela signifie qu'à chaque pas, les entités qui sont puits *savent* qu'elles sont puits et réagissent immédiatement.

5.5.1 Liens de communications volatiles

Nous avons d'abord simulé un réseau dans lequel les liens entre les entités se créent et se ferment fréquemment. Le comportement d'un tel réseau peut être proche d'un réseau ad-hoc mobile avec des relations de voisinage volatiles. A chaque pas i, nous choisissons aléatoirement φ entités dans V_i et nous leur imposons de fermer un arc si elles possèdent

trop de voisins, ou de faire la requête d'un nouveau voisin si leur nombre de voisins est inférieur à la valeur souhaitée.

Le réseau contient initialement 1500 entités possédant 10 voisins chacune. La table 5.1 décrit les différents contextes d'exécution. Pour une meilleure compréhension, nous avons indiqué ϱ le nombre moyen d'arcs affecté par notre procédure de mobilité par entité et par période. Nous avons lentement rendu le réseau de plus en plus mobile, puis nous sommes subitement revenu à un contexte de réseau fixe. La figure 5.3 présente les résultats obtenus.

Pas d'exécution	φ	ϱ
$[0 \dots 2.100]$	0	0
$[2.100 \dots 4.200]$	5	1
$[4.200 \dots 6.300]$	10	2
$[6.300 \dots 8.400]$	20	4
$[8.400 \dots 10.500]$	40	8
$[10.500 \dots 12.600]$	80	16
$[12.600 \dots 14.700]$	120	24
$[14.700 \dots 16.800]$	0	0

TAB. 5.1 – Variation de l'intensité de la volatilité des arcs

Les résultats sont, dans un premier temps, conforme au théorème de [13] : dans un système synchrone statique, toutes les entités sont puits exactement le même nombre de fois sur une période. Ainsi, durant les 2100 premiers et derniers pas, les valeurs de Δ_{min} et Δ_{max} sont strictement égales. Cela démontre que nos algorithmes respectent effectivement cette propriété des graphes orientés acycliquement.

Nous observons ensuite que la mobilité a un impact sur l'équité du système puisque certaines entités sont puits plus souvent que d'autres, comme le montre l'écart entre Δ_{min} et Δ_{max}. Mais, cet écart reste minime puisque la différence entre les deux valeurs reste toujours inférieur à 1, à comparer avec 15, le nombre moyen de pas pendant lesquels les entités ont été puits sur une période. De plus, il est intéressant de constater que ce nombre moyen ($\simeq 15$) reste à peu près constant durant toute l'exécution.

L'augmentation de la volatilité des liens n'a qu'un impact : la différence entre Δ_{min} et Δ_{max} s'accroît avec la croissance de la mobilité.

Naturellement, les courbes retrouvent leur comportement normal lorsque le réseau redevient statique. Nous observons tout de même que la fréquence finale du nombre de fois où les entités sont puits est différente de la fréquence initiale ($\Delta = 13$ au lieu de $\Delta = 15$). Cela indique que le réseau est dans une configuration différente et que la configuration a un impact sur cette fréquence.

Pour conclure cette partie, nous notons que les algorithmes semblent robustes à la volatilité des liens de communications puisque celle-ci semble avoir un impact très limité, même dans un contexte de mobilité très intense.

5.5.2 Entités volatiles

Nous simulons ensuite un réseau dans lequel les entités apparaissent et disparaissent souvent, selon un comportement proche des réseaux de pairs sur Internet. Là encore, nous

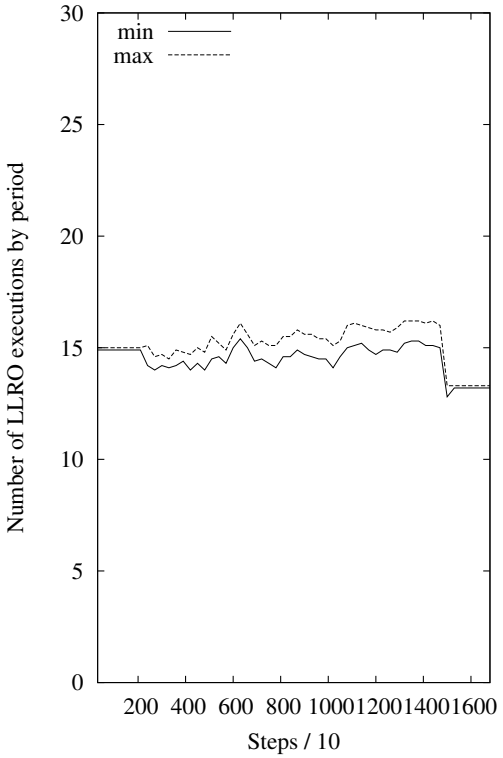

FIG. 5.3 – Évolutions de Δ_{min} et Δ_{max} dans un réseau dans lequel les liens de communications sont volatiles.

observons la fréquence des situations de puits par entité et par période.

Initialement, le réseau contient 1000 entités avec un nombre souhaité de voisins fixé à 10. Nous ajoutons et retirons des entités dans le réseau à une vitesse détaillée dans la table 5.2. La vitesse d'insertion de nouvelles entités est lente au départ (une entité tous les deux pas), puis nous accélérons la vitesse d'insertion. Nous stabilisons alors le système puis nous retirons des entités selon le même principe.

Afin de conserver une bonne connectivité à notre réseau et d'éviter les partitionnements lorsque les entités quittent le système, nous maintenons une mobilité pour les arcs constante à $\varphi = 60$. Les simulations précédentes ont montré que cela n'affecte pas le système. La figure 5.4 présente les variations des valeurs de Δ_{min} et de Δ_{max} durant l'exécution.

Nous observons que ni l'ajout de nouvelles entités, ni le départ d'entités n'affecte la fréquence des situations pour lesquelles les entités sont puits. De plus, la vitesse d'arrivée et de départ des entités n'admet aucun impact.

5.5.3 Regroupement et partitionnement du réseau

Un des principaux risques liés au réseau mobile est son partitionnement. Quand une entité qui est le seul lien entre deux sous-ensembles des entités connectées disparaît, le réseau cesse d'être connexe. Dans ces simulations, nous avons cherché à observer l'impact

Pas d'exécution	Nombre initial d'entités	Nombre final d'entités
[0 . . . 2.100]	1.000	1.000
[2.100 . . . 4.200]	1.000	1.150
[4.200 . . . 6.300]	1.150	1.450
[6.300 . . . 8.400]	1.450	1.450
[8.400 . . . 10.500]	1.450	1.300
[10.500 . . . 12.600]	1.300	1.000
[12.600 . . . 14.700]	1.000	1.000

TAB. 5.2 – Variation du nombre d'entités participant au système

d'un regroupement de plusieurs réseaux initialement dans des configurations différentes, puis, ensuite, à mesurer l'impact d'un partitionnement d'un réseau.

Nous considérons ici le cas de quatre groupes de 300 entités, avec différents nombres souhaités de voisins : 7 voisins pour le groupe 1, 9 voisins pour le groupe 2, 11 pour le groupe 3 et 13 pour le groupe 4. Nous introduisons, dans chaque groupe, une mobilité des arcs constante égale à $\varphi = 50$. Au pas 2100, soudainement, nous provoquons un regroupement des quatre groupes. Cette opération est réalisée en créant artificiellement une connexion entre un membre choisi aléatoirement dans chacun des groupes. Ensuite, nous laissons le système se stabiliser. Puis, brutalement, au pas 6300, nous provoquons un partitionnement du réseau en forçant toutes les entités à fermer leurs connexions avec les entités n'appartenant pas à son groupe initial.

Dans la figure 5.5, nous mesurons la variation de la moyenne des fréquences entre deux situations puits pour les entités de chaque groupe Δ_{avg}.

La première partie nous permet de vérifier l'intuition que le nombre souhaité de voisins, et donc le degré des entités dans le graphe, a un impact sur la fréquence des situations pour lesquelles les entités sont puits. Ainsi, nous observons que cette fréquence est inversement proportionnel au nombre souhaité de voisins. Quand, dans un réseau, le nombre de voisins de chaque entité est faible, ces entités sont souvent puits.

Dans la seconde partie de la simulation, nous observons qu'immédiatement après le pas 2100 toutes les entités ont strictement la même fréquence entre les situations où elles sont puits. De plus, nous remarquons que cette fréquence correspond à un peu moins que la valeur de la fréquence du pire groupe, cela étant dû à un manque de synchronisation entre les entités.

Enfin, les effets du partitionnement apparaissant au pas 6300 sont lents à observer (plus de mille pas). En effet, il faut attendre qu'une meilleure synchronisation s'opère entre les entités. Grâce à la mobilité que nous entretenons, lentement, le réseau se resynchronise plus efficacement et, naturellement, la hiérarchie que nous observions dans les 2100 premiers pas se reforment, mais à des valeurs qui restent éloignées des valeurs initiales.

5.6 Conclusions

Dans ce chapitre, nous avons présenté une solution pour un service d'éditeur/abonné pour un réseau de pairs dans lequel les entités sont très mobiles. Notre proposition comprend une modélisation par un système multi-couches, des graphes orientés acycliquement

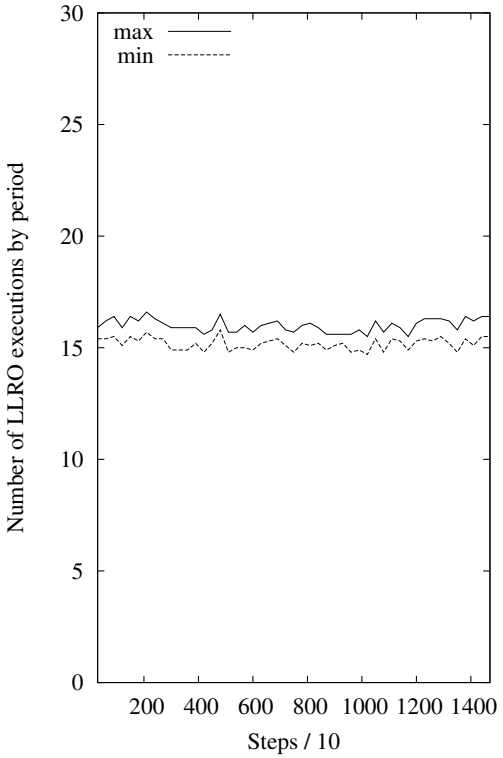

FIG. 5.4 – Évolutions de Δ_{min} et Δ_{max} dans un réseau dans lequel le nombre d'entités varie.

et un organisation basée sur un réseau de pairs hybrides avec des super-pairs.

Le système multi-couches offre de nombreux avantages pour la réalisation d'un service d'éditeur/abonné. Notamment, lorsqu'une notification est émise, seules les entités potentiellement intéressées par cette notification sont affectées par la publication. De plus, comme les couches sont indépendantes deux à deux, le système n'admet aucune limitation sur le nombre de couches existantes. En conséquence, le nombre d'attributs intéressant les entités n'a aucune limitation, offrant beaucoup de flexibilité au service.

Le graphe orienté acycliquement offre la garantie d'une diffusion totale de l'information sans risque de pics d'utilisation des ressources du réseau. En cassant la symétrie du réseau de pairs, nous permettons aux entités d'être les seules émettrices de messages dans leur voisinage. De fait, localement, la charge du réseau est limitée.

Il est important de permettre à chaque entité de pouvoir émettre une notification infiniment souvent. Nous avons proposé des algorithmes simples qui garantissent théoriquement cette propriété, puis nous avons montré par des simulations que la mobilité dans le réseau n'affecte pas ce résultat. Nous avons soumis un réseau à de fortes contraintes de mobilité et les résultats obtenus satisfont toujours nos exigences.

Enfin, nous avons proposé une architecture basée sur un réseau de pairs hybride. Ce type d'organisation a montré, par l'expérience, qu'elle offre toutes les garanties de scalabilité [2]. En outre, elle permet à chacune des entités de participer à la hauteur de ses

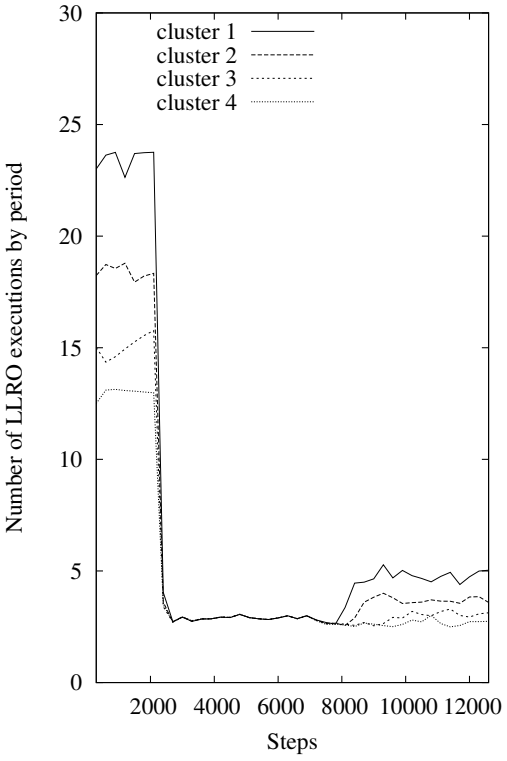

FIG. 5.5 – Évolutions de Δ_{avg} dans quatre groupes d'entités.

possibilités aux tâches de l'application.

Nous avons proposé une organisation simple et flexible. La simplicité provient du fait que seules les super-entités participent à la diffusion des notifications. Elles sont théoriquement plus stables et plus à même de supporter la charge imposée par ces diffusions. La flexibilité apparaît davantage pour les entités qui peuvent se connecter à n'importe quelle super-entité détectée et, immédiatement, profiter du service. Nous n'imposons pas aux entités de détecter la super-entité qui s'accorde à ses intérêts, ni de changer de super-entités à chaque modification d'intérêts.

Les avantages de la solution que nous proposons semblent en parfaite adéquation avec un service pour Solipsis. Une entité peut aisément se connecter avec une super-entité dans son voisinage virtuel. Il est probable que les intérêts émis par des entités proches virtuellement auront une plus grande similitude. Cette hypothèse renforce les performances du système. Enfin, la robustesse à la mobilité est importante car le monde virtuel que nous envisageons est un espace en constante mutation d'idées, d'intérêts, de positions et de participants.

Dans le futur, nous espérons pouvoir expérimenter ces résultats théoriques dans la réalité (virtuelle bien évidemment). Nous n'avons pas voulu effectuer des simulations de ce système car il est impossible de prévoir quelle utilisation les participants en feront. Notamment, il est difficile de simuler la forme et la sémantique des notifications et des

intérêts. Or, ces paramètres sont cruciaux pour l'efficacité de ce service. Nous avons volontairement laissé de nombreux algorithmes ouverts de manière à pouvoir les modifier dès que les premières expérimentations feront apparaître des tendances d'utilisation.

Chapitre 6

Conclusion

Dans cette thèse, nous avons préalablement cherché à mettre en lumière les raisons interdisant aux environnements virtuels partagés actuels d'être utilisés par un nombre illimité de participants. Nous avons montré que l'architecture des systèmes les plus récents amène des contraintes qui rendent impossible l'existence de mondes virtuels massivement partagés. Nous avons donc proposé de concevoir un système qui se base sur une approche différente de celles existantes. Nous présentons ci-après les principales contributions de ce travail ainsi qu'une série de questions qui restent ouvertes.

6.1 Contributions

6.1.1 Un monde virtuel massivement partagé

Nous avons commencé par étudier les principaux concepts fondamentaux de la réalité virtuelle. Nous avons alors remarqué qu'une entité virtuelle, avatar de l'utilisateur, n'est rien d'autre qu'un processus autonome et indépendant. Cette caractéristique rend tout à fait possible la conception d'un monde virtuel basé sur un réseau de pairs, dans lequel un pair est une entité et tous les algorithmes s'appuient sur la collaboration de chacun. Or, les réseaux de pairs permettent d'imaginer des systèmes répartis qui peuvent être utilisées par un nombre *a priori* illimité de machines.

Notre travail a donc consisté à concevoir les algorithmes qui, mis en œuvre par chaque entité, construisent un environnement virtuel se conformant aux propriétés souhaitées. Puis, après avoir apporté des preuves formelles que ce système est suffisamment proche de ce que nous visions, nous avons décidé de passer à une phase de réalisation. Celle-ci a nécessité l'élaboration d'un protocole de communication que nous avons souhaité léger. Le résultat de ce travail est Solipsis, actuellement disponible sur Internet. L'exécution du programme permet à n'importe quelle machine de participer à la naissance d'un monde virtuel qui peut être partagé par des millions (milliards) d'utilisateurs.

6.1.2 Des services pour systèmes distribués mobiles

Nous avons ensuite déterminé deux services qui pourraient être intéressants dans Solipsis, mais dont la conception est rendue difficile par l'architecture en réseaux de pairs. En effet, de nombreuses contraintes apparaissent lorsqu'un système distribué est totalement décentralisé, que le nombre de participants n'est pas limité et que ces participants ont un

comportement dynamique. Les solutions que nous avons proposées prennent en compte ces caractéristiques.

Nous avons tout d'abord cherché à déterminer des algorithmes efficaces permettant de détecter une entité vérifiant un prédicat donné. Nous nous sommes placés dans le contexte des réseaux ad-hoc, qui présentent de nombreuses similitudes avec Solipsis, ainsi que des contraintes supplémentaires. Les algorithmes que nous avons présentés permettent de diffuser une information et de construire un structure logique superposée à la topologie du réseau. Ces algorithmes ont la particularité d'être auto-stabilisants et, ainsi, d'être parfaitement adaptés à la mobilité des entités.

Puis, nous avons proposé une solution pour le problème de l'éditeur/abonné dans le contexte des réseaux de pairs dynamiques. Nous nous sommes concentrés sur une solution flexible apportant de la robustesse dans un réseau fortement mobile. Les algorithmes comprennent une organisation basée sur un réseau de pairs hybride, ainsi que des algorithmes permettant de diffuser une information sans dépasser les limites physiques des machines.

6.1.3 Une communauté autour des systèmes distribués mobiles

Tous les travaux menés durant cette thèse ont été fortement inspiré par les recherches dans les contextes des réseaux ad-hoc, des réseaux de pairs et des systèmes distribués en général. Nous avons remarqué qu'il existe de nombreuses similitudes entre les algorithmes proposées dans une communauté et les solutions présentées dans une autre. Les deux services présentés dans la deuxième partie de ce document illustrent parfaitement notre volonté de mettre en commun ces différents axe de recherche autour des systèmes distribués mobiles.

6.2 Travaux futurs

6.2.1 Principes fondamentaux et choix techniques

Dans [60], les principes fondamentaux de Solipsis sont discutés et comparés avec un autre système, issu des mêmes postulats, mais utilisant une topologie basée sur une triangulation de Delaunay. Néanmoins, une grande partie des principes fondamentaux de Solipsis se retrouvent dans ce travail. Les auteurs sont actuellement en train de mesurer les performances de leur système. Leur proposition semble garantir avec plus de sûreté la cohérence du monde virtuel, mais les algorithmes proposés paraissent plus coûteux que ceux de Solipsis. Il est évident que certains algorithmes de Solipsis peuvent être améliorés et nous pensons que l'architecture du programme que nous avons conçu est suffisamment flexible pour supporter des modifications.

Le protocole de Solipsis et certains choix techniques réalisés seront sans doute amenés à évoluer dans le futur. Nous travaillons actuellement à rendre le protocole plus robuste aux défaillances qui peuvent survenir sur les liens de communications. Là encore, nous avons réalisé Solipsis avec l'idée que de nouveaux protocoles et d'autres choix pourraient être effectués. Nous espérons donc qu'il sera, effectivement, peu coûteux de faire les modifications qui seront jugées nécessaires. En outre, nous espérons pouvoir entrer dans une phase de normalisation du protocole prochainement.

6.2.2 Des services utiles pour les utilisateurs

Pour l'instant, le monde virtuel contient en majorité des objets virtuels inertes. Nous espérons que, dans un futur proche, des utilisateurs décident d'utiliser Solipsis comme un espace de rencontres virtuel. Il est évident que le comportement humain ouvrira de nouvelles pistes de travail, notamment au niveau de la gestion des déplacements.

De nombreux services sont encore à l'idée de projet. Principalement, un mécanisme permettant d'éviter les collisions entre les entités est à prévoir. Mais, d'autres services peuvent être imaginés. Ainsi, la diffusion par un orateur d'un flux de données vers un grand nombre d'entités reste un problème délicat.

Nous espérons surtout que le comportement de participants humains et l'utilisation qu'ils feront du monde virtuel feront apparaître le besoin de nouveaux services.

6.2.3 Les systèmes distribués mobiles

La recherche autour des systèmes distribués mobiles, même si elle est très étudiée actuellement, possède suffisamment de potentiel pour ouvrir de nombreux nouveaux débats. Dans [47], nous avons présenté une collection de questions ouvertes. Un grand nombre d'entre elles sont encore loin d'avoir trouvé une réponse satisfaisante.

Par exemple, de nombreux travaux restent à accomplir sur le sujet de la modélisation de systèmes mobiles, notamment des études sur l'impact du comportement non déterministe des utilisateurs sur le réseau. En outre, les progrès techniques et la commercialisation immédiate de nouvelles technologies permettent d'imaginer de nombreuses hypothèses sur les réseaux futurs. Enfin, le passage à l'échelle est devenue une contrainte très forte qui remet en cause de nombreuses applications réparties.

Nous espérons que nous aurons l'occasion, prochainement, d'évoquer ces nouveaux sujets de recherche au cours d'une discussion dans le monde virtuel.

Bibliographie

[1] Gryphon website. available on http ://www.research.ibm.com/gryphon/.

[2] Kazaa. available on http ://www.kazaa.com.

[3] Siena website. available on http ://www.cs.colorado.edu/users/carzanig/siena/.

[4] ANSI/IEEE standard 1278-1993, Protocols for Distributed Interactive Simulation, March 1993.

[5] The Gnutella Protocol Specification v0.4. available on http ://www.gnutella.co.uk/, September 2000.

[6] A.Bharambe, S. Rao, and S. Seshan. Mercury : a Scalable Publish-Subscribe System for Internet Games. In *Workshop on Network and System Support for Games (Netgames'02)*, 2002.

[7] M. Abrams, C. R. Standridge, G. Abdulla, S. Williams, and E. A. Fox. Caching Proxies : Limitations and Potentials. In *Proceedings of the 4th International WWW Conference*, December 1995.

[8] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching Events in a Content-Based Subscription System. In *ACM Symposium on Principles of Distributed Computing (PODC'99)*, May 1999.

[9] M. Aguilera, R. Strom, D. Sturman, M. Astley, and T. Chandra. Matching Events in a Content-Based Subscription System. In *ACM Symposium on Principles of Distributed Computing (PODC'99)*, 1999.

[10] K. Alzoubi, P-J. Wan, and O. Frieder. Weakly Connected Dominating Sets and Sparse Spanners in Wireless Ad-Hoc Networks. *International Conference on Distributed Computing Systems (ICDCS'03)*, pages 96–104, 2003.

[11] E. Anceaume, A. K. Datta, M. Gradinariu, and G. Simon. Publish/Subscribe Scheme for Mobile Networks. In *Workshop on Principles on Mobile Computing (POMC'02)*, 2002.

[12] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. Strom, and D. Sturman. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In *International Conference on Distributed Computing Systems (ICDCS'99)*, May 1999.

[13] V. C. Barbosa and E. Gafni. Concurrency in Heavily Loaded Neighborhood-Constrained Systems. *ACM Transactions on Programming Languages and Systems*, 11(4) :562–584, 1989.

[14] L. Barriere, P. Fraigniaud, L. Narayanan, and J. Opatrny. Robust Position-Based Routing in Wireless Ad Hoc Networks with Unstable Transmission Ranges. In *Proceedings of DialM*, 2001.

[15] J. W. Barrus, R. C. Waters, and D. B. Anderson. Locales and Beacons : Efficient and Precise Support for Large Multi-User Virtual Environment. Technical report, Mitsubishi Electric Research Laboratory, August 1996.

[16] J. W. Barrus, R. C. Waters, D. B. Anderson, D. Brogan, M. Casey, S. McKeown, T. Nitta, I. Stern, and W. Yerazunis. Diamond Park and Spline : A Social Virtual Reality System with 3D Animation, Spoken Interaction and Runtime Modifiability. Technical report, Mitsubishi Electric Research Laboratory, November 1996.

[17] S. Benford, J. Bowers, L. E. Fahlén, and C. Greenhalgh. Managing Mutual Awareness in Collaborative Virtual Environments. In *VRST'94*, August 1994.

[18] S. D. Benford and C. M. Greenhalgh. Introducing Third Party Objects into the Spatial Model of Interaction. In *Proceedings of ECSCW'97*, pages 189–204, 1997.

[19] C. Bettstetter. On the Minimum Node Degree and Connectivity of a Wireless Multihop Network. In *ACM International Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc'02)*, June 2002.

[20] D. M. Blough, M. Leoncini, G. Resta, and P. Santi. The k-Neighbors Protocol for Symmetric Topology Control in Ad-Hoc Networks. *Proceedings of ACMInternational Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc'03)*, 2003.

[21] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with Guaranteed Delivery in Ad-Hoc Wireless Networks. *ACM/Kluver Wireless Networks*, 7(6), 2001.

[22] C. Bowman, P. Danzig, and D. Hardy. The Harvest Information Discovery and Access System. *Computer Networks*, 28(1-2), 1995.

[23] M. Burkhart, P. von Rickenbach, R. Wattenhofer, and A. Zollinger. Does Topology Control Reduce Interference ? In *ACMInternational Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc'04)*, 2004.

[24] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad-hoc network research. *Wireless Communications and Mobile Computing*, 2002.

[25] A. Carzaniga, D. Rosenblum, and A. Wolf. Challenges for Distributed Event Services : Scalability vs. Expressiveness. In *Engineering Distributed Objects*, 1999.

[26] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3) :332–383, August 2001.

[27] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream : High-Bandwidth Multicast in a Cooperative Environment. In *ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[28] M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstrom. Scribe : A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications*, 2001.

[29] G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. In *Proc. Middleware 2000 : IFIP/ACM International Conference on Distributed Systems Platforms*, pages 1–23, April 2000.

[30] I. Clarke, O. Sandberg, B. Wiley, and T. Hong. Freenet : A distributed anonymous information storage and retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, 2000.

[31] T. Clause, P. Jacquet, and A. Laouti. Optimized Link State Routing Protocol. *INMIC'01, Pakistan*, 2001.

[32] B. Cohen. Incentives Build Robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer*, 2003.

[33] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. MIT Press and McGraw-Hill, July 2001.

[34] B. Delaunay. Sur la sphère vide. *Otdelenie Matematicheskii i Estetven- nyka Nauk*, 7 :793–800, 1934.

[35] E. W. Dijkstra. Self-Stabilizing Systems in Spite of Distributed Control. *Communications of the ACM*, 17(11) :643–644, November 1974.

[36] O. Dousse, P. Thiran, and M. Hasler. Connectivity in ad-hoc and hybrid networks. In *IEEE Infocom*, 2002.

[37] P. Eugster, P. Felber, R. Guerraoui, and A-M. Kermarrec. The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2) :114–131, 2003.

[38] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary Cache : a Scalable Wide Area Web Cache Sharing Protocol. *Transactions on Networking*, 8(3), 2000.

[39] S. Fiedler, M. Wallner, and M. Weber. A Communication Architecture for Massive Multiplayer Games. In *Workshop on Network and systems support for Games (NetGames'02)*, 2002.

[40] P. Fraigniaud and P. Gauron. An Overview of the Content-Adressable Network D2B. In *ACM Symposium on Principles of Distributed Computing (PODC'03)*, 2003.

[41] E. Frécon and M. Stenius. DIVE - A Scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering Journal (special issue on Distributed Virtual Environments)*, Vol. 5, September 1998.

[42] R. Friedman. Caching Web Services in Mobile Ad-hoc Networks. *Proc. of the 2nd Int. Workshop of Principles of Mobile Computing (POMC'02)*, pages 90–96, 2002.

[43] R. Friedman, M. Gradinariu, and G. Simon. Locating Cache Proxies in MANETs. In *ACM International Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc'04)*, 2004.

[44] K. R. Gabriel and R. R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18 :259–278, 1969.

[45] A. Gérodolle, F. Dang Tran, and L. Garcia Bañuelos. A Middleware Approach to Building Large-Scale Open Shared Virtual Worlds. In *TOOLS Europe 2000*, June 2000.

[46] K. Gough and G. Smith. Efficient Recognition of Events in Distributed Systems. In *Proceedings of the ACSC-18*, 1995.

[47] M. Gradinariu, M. Raynal, and G. Simon. Looking for a Common View for Mobile Worlds. In *Future Trends of Distributed Computing Systems (FTDCS'03)*, May 2003.

[48] C. Greenhalgh and S. Benford. MASSIVE, a Distributed Virtual Reality System incorporating Spatial Trading. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, June 1995.

[49] C. Greenhalgh and S. Benford. Supporting Rich and Dynamic Communication in Large-Scale Collaborative Virtual Environments. *MIT Presence*, February 1999.

[50] C. I. Grima and A. Màrquez. *Computational Geometry on Surfaces*. Kluwer Academic Publishers, 2001.

[51] S. Guha and S. Khuller. Approximation Algorithms for Connected Dominating Sets. *Algorithmica*, 1998.

[52] Z. Haas. A New Routing Protocol for the Reconfigurable Wireless Networks. *IEEE International Conference on Universal Personal Communications (ICUP'97)*, 1997.

[53] T. Herman and S. Tixeuil. A Distributed Slot Assignement for Wireless Sensor Networks. *Technical Report no.1370, LRI, Université Paris SUD*, 2003.

[54] C. Ho, K. Obraczka, G.Tsudik, and K.Viswanath. Flooding for Reliable Multicast in Multi-Hop Ad-hoc Networks. *Proc. of the 3th Int. Workshop on Discrete Algothrithms and Methods for Mobile Computing and Communications (DialM'99)*, pages 64–71, 1999.

[55] http ://solipsis.netofpeers.net.

[56] http ://www.corba.org.

[57] http ://www.python.org.

[58] http ://www.w3.org/tr/soap/.

[59] http ://www.xmlrpc.com.

[60] S-Y. Hu and G-M. Liao. Scalable Peer-to-Peer Networked Virtual Environment. In *Network and Systems Support for Games (NetGames'04)*, 2004.

[61] *Request For Comments RFC 2236 Internet Group Management Protocol*.

[62] *Request For Comments RFC 791 Internet Protocol*.

[63] J. W. Jaromczyk and G. T. Toussaint. Relative Neighborhood Graphs and Their Relatives. In *Proc. IEEE*, volume 80, pages 1502–1517, 1992.

[64] D.B. Johnson and D.A. Maltz. Dynamic Source Routing in Ad-hoc Wireless Networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[65] F. Kaashoek and D. Karger. Koorde : a Simple Degree-optimal Hash Table. In *International Workshop on Peer-To-Peer Systems (IPTPS'03)*, 2003.

[66] B. Karp and H. T. Kung. GPSR : Greedy Perimeter Stateless Routing for Wireless Networks. In *6th ACM Conference on Mobile Computing and Networking (Mobicom'00)*, 2000.

[67] J. Keller. *Objets communicants*, chapter 10. Hermes, 2002.

[68] J. Keller and G. Simon. Toward a Peer-to-Peer Shared Virtual Reality. In *IEEE Workshop on Resource Sharing in Massively Distributed Systems (RESH'02)*, July 2002.

[69] J. Keller and G. Simon. Solipsis : A Massively Multi-Participant Virtual World. In *International Conference on Parallel and Distributed Techniques and Applications (PDPTA'03)*, 2003.

[70] S. Keshav. *An Engineering Approach to Computer Networking.* Addison Wesley, April 1997.

[71] J. Kleinberg. The Small-World Phenomenon : An Algorithmic Perspective. In *ACM Symposium on Theory of Computing*, 2000.

[72] E. Léty. *Une Architecture de Communication pour Environnements Virtuels Distribués à Grande-Échelle sur l'Internet.* PhD thesis, Université de Nice-Sophia Antipolis, December 2000.

[73] X-Y. Li. Algorithmic, geometric and graphs issues in wireless networks. *Journals of Wireless Communications and Mobile Computing (WCMC)*, 2002.

[74] X.Y. Li, G. Calinescu, and P-J. Wan. Distributed Construction of a Planar Spanenr and Routing for Ad Hoc Wireless Networks. In *Proceedings of IEEE Infocom*, 2002.

[75] M. Macedonia. *A Network Software Architecture for Large Scale Virtual Environments.* PhD thesis, Naval Postgraduate School, June 1995.

[76] J-M. Menaud, V. Issarny, and M. Banatre. A Scalable and Efficient Cooperative System for Web Caches. *IEEE Concurrency*, 8(3), 2000.

[77] *Request For Comments RFC 1631 Network Address Translator.*

[78] S. Ni, Y. Tseng, Y. Chen, and J. Sheu. The Broadcast Storm Problem in a Mobile Ad-hoc Network. *5th ACM Conference on Mobile Computing and Networking (Mobicom'99)*, pages 151–162, 1999.

[79] W. Peng and X. Lu. On the Reduction of Broadcast Redundancy in Mobile Ad-hoc Networks. *ACM International Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc'00)*, 2000.

[80] W. Peng and X. Lu. An Efficient Broadcast Protocol for Mobile Ad-hoc Networks. *Journal of Science and Technology*, 2002.

[81] C. Perkins. Ad-hoc On Demand Distance Vector (AODV) Routing. *Internet Draft, draft-ietf-manet-aodv-00.txt*, 1997.

[82] P. Pietzuch and J. Bacon. Hermes : A Distributed Event-Based Middleware. In *International Workshop on Distributed Event-Based Systems (DEBS'03)*, 2003.

[83] F. P. Preparata and M. I. Shamos. *Computational Geometry : An Introduction.* Springer-Verlag, 1985.

[84] J. Purbrick and C. Greenhalgh. Extending Locales : Awareness Management in MASSIVE-3. *IEEE Virtual Reality 2000*, March 2000.

[85] R. Ramanathan and R. Rosales Hain. Topology Control of Multihop Wireless Networks using Transmit Power Adjustment. In *IEEE Infocom'00*, 2000.

[86] S. Ratnasamy, M. Handley, P. Francis, and R. Karp. A Scalable Content-Addressable Network. In *ACM SIG/COMM*, pages 161–172, 2001.

[87] A. Rowstrom and P. Druschel. Pastry : Scalable distributed object location and routing for large-scale peer-to-peer systems. In *ACM International Conference on Distributed Systems Platforms (Middleware'01)*, 2001.

[88] R. Roychoundhury, S. Bandyopadhyay, and K. Paul. A Distributed Mechanism for Topology Discovery in Ad-hoc Wireless Networks using Mobile Agents. *ACM International Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc'00)*, 2000.

[89] F. Sailhan and V. Issarny. Cooperative Caching in Ad-hoc Networks. *Mobile data management*, pages 13–28, 2003.

[90] M. Seddigh, J. Solano Gonzales, and I. Stojmenovic. RNG and Internal Node Based Broadcasting Algorithms for Wireless One-to-one Networks. *ACM Mobile Computing and Communications Review*, pages 37–44, 2002.

[91] Neal Stephenson. *Snow Crash*. 1992.

[92] I. Stoica, R.Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord : A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, pages 149–160, 2001.

[93] I. Stojmenovic. Position-Based Routing in Ad Hoc Networks. *IEEE Communications Magazine*, 40(7) :128–134, 2002.

[94] *Request For Comments RFC 793 Transmission Control Protocol*.

[95] G. T. Toussaint. The relative neighborhood graph in a finite planar set. *Pattern Recognition*, pages 261–268, 1980.

[96] Y. Tseng, S. Ni, and E. Shih. Adaptive Approaches to Relieving Broadcast Storms in a Wireless Multihop Mobile Ad-hoc Network. *IEEE Transactions on Computers*, 52(5) :545–557, 2003.

[97] *Request For Comments RFC 768 User Datagram Protocol*.

[98] V. Valloppillil and K. Ross. Cache Array Routing Protocol, internet draft. *http ://ircache.nlanr.net/Cache/ICP/carp.txt*, 1998.

[99] A. Virgilitto. *Publish-Subscribe Communication Systems : From Models to Applications*. PhD thesis, Universita La Sapienza, 2003.

[100] G. Voronoï. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. *Journal für die Reine and Angewandte Mathematik*, 133 :97–178, 1908.

[101] Y. Wang, L. Qiu, D. Achlioptas, G. Das, P. Larson, and H. Wang. Subscription Partitioning and Routing in Content-Based Publish/Subscribe Networks. In *Symposium on Distributed Computing (DISC'02)*, 2002.

[102] R. Wattenhofer and A. Zollinger. XTC : A Practical Topology Control Algorithm for Ad-Hoc Networks. In *4th International Workshop for Wireless Mobile Ad Hoc and Sensor Networks (WMAN'04)*, 2004.

[103] B. Williams and T. Camp. Comparison of Broadcasting Techniques for Mobile Ad-hoc Networks. *ACM International Symposium on Mobile Ad-Hoc Networking and Computing (MobiHoc'02)*, 2002.

[104] J. Williams. *Using Java to construct a Distributed Virtual Environment*. PhD thesis, Curtin University School of Electrical and Computer Engineering, 1998.

[105] J. Wu, M. Gao, and I. Stojmenovic. On Calculating Power-Aware Connected Dominating Sets for Efficient Routing in Ad-hoc Wireless Networks. *Proc. of the 30th International Conference on Parallel Processing (ICPP'01)*, pages 346–353, 2001.

[106] J. Wu and H. Li. On Calculating Connected Dominating Set for Efficient Routing in Ad-hoc Wireless Networks. *Proc. of the 3th Int. Workshop on Discrete Algothrithms and Methods for Mobile Computing and Communications (DialM'99)*, pages 7–14, 1999.

[107] H. Yang and B. Garcia Molina. Designing a Super-Peer Networks. Technical report, Stanford University, 2002.

[108] A. C. Yao. On Constructing Minimum Spanning Trees in k-dimensional Spaces and Related Problems. *SIAM Journal on Computing*, 11 :721–736, 1982.

[109] P. S. Yu and E. A. MacNair. Performance Study of a Collaborative Method for Hierarchical Caching in Proxy Servers. In *Proceedings of the 7th International WWW Conference*, April 1998.

[110] B. Zhao, J. Kubiatoviwicz, and A. Joseph. Tapestry : an Infrastructure for Fault-Tolerant Wide-Area Location and Routing. *Technical Report UCB/CSD-01-1141, Computer Science U.C. Berkeley*, 2001.