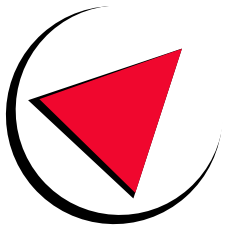

L I F C

Université de Franche-Comté

Mémoire Master 2 Recherche

DST - Distributed Spanning Tree : Une topologie pour la recherche de services de calcul sur la grille

par Christophe Enderlin

le 21 octobre 2012

encadré par

Laurent Philippe ⋆ Professeur à l'Université de Franche-Comté

soutenu le 30 septembre 2011 *devant le jury*

Jean-Marc Nicod ⋆ Professeur à l'École Nationale Supérieure de Mécanique et des Microtechniques de Besançon

David Laiymani ⋆ Maître de Conférence à l'Université de Franche-Comté

Remerciements

Me voici arrivé au terme d'une aventure (le mot n'est pas trop fort !) commencée à Besançon, en octobre 2005. Ces six années ont été riches en expériences, bonnes et moins bonnes, en émotions et en enseignements de toutes sortes, tant dans mes vies familiales et professionnelles que d'étudiant. Pouvoir reprendre des études à mon grand âge (!) et renouer avec le plaisir d'apprendre est une fabuleuse opportunité et je voudrais ici exprimer ma grande reconnaissance envers toutes les personnes qui ont rendu cette reprise d'études possible.

Je veux donc remercier Isabelle Jacques, Eveline Renard et sa remplaçante Stéphanie Jubin, pour leur accueil, leur rigueur et leur professionnalisme. Je voudrais particulièrement remercier Sylvie Damy pour sa compétence, sa gentillesse et son dévouement.

Je veux aussi remercier l'ensemble des enseignants qui ont accepté de jouer le jeu difficile de l'enseignement à distance et qui ont dû supporter nos centaines de questions ! (Pardon à Mmes Greffier et Damy que nous avons traumatisées avec nos questions ! ☺) Étudier seul chez soi n'est pas toujours facile et peut être décourageant par moment ; j'ai donc apprécié à sa juste valeur les efforts déployés pour nous par ces enseignants.

Je profite de cet espace pour remercier mes fidèles compagnons de route, Anne Alvarez et Jean-Luc Joly, pour les heures passées à travailler ensemble, les mails encourageants et les bons restaus ! Un grand merci aussi aux autres étudiants qui ont participé à la très bonne ambiance des forums.

Des remerciements spéciaux pour ma famille qui a dû me supporter pendant ce périple, tout particulièrement ma femme qui n'a jamais cessé de m'encourager et pour sa relecture de ce rapport ! ☺

Un grand merci également à l'équipe de Simgrid, en particulier Arnaud Legrand et Martin Quinson, pour leur sympathie et l'aide qu'ils m'ont apporté en répondant à mes questions.

Pour finir, je souhaite remercier Laurent Philippe, mon encadrant de stage, pour sa gentillesse, sa disponibilité et sa compétence et pour m'avoir, avec Bénédicte Herrmann, proposé un sujet si intéressant. Je considère comme un privilège d'avoir pu travailler avec lui pendant ce stage.

Table des matières

Première partie

Contexte

Chapitre 1

Préambule

1.1 Introduction

Aujourd'hui, il est possible d'accéder aisément via Internet à de grands systèmes distribués tels que les réseaux *peer-to-peer*. Ceux-ci permettent toutes sortes d'applications très utiles telles que le partage de fichiers à grande échelle, le stockage redondant, le nommage hiérarchique ou le calcul scientifique partagé. De telles structures vont bien au-delà des classiques systèmes client/serveur puisque chacun des pairs peut jouer les deux rôles.

Dans le cadre de cette étude, nous nous intéressons plus particulièrement aux réseaux *peer-to-peer* totalement dépourvus de structure de contrôle centralisée, s'organisant eux-mêmes et capables de répartir la charge requise par les requêtes qui leur sont soumises entre les différents pairs. De tels systèmes sont robustes, tolérants aux pannes et peuvent être déployés à très grande échelle. Ce peut être le cas de machines raccordées à Internet comme on vient de le dire, mais aussi de réseaux de capteurs sans fils communiquant de façon *ad-hoc*. Ces réseaux sont également fortement dynamiques : des nœuds y arrivent et en repartent à tout instant.

Pour pouvoir utiliser efficacement les ressources offertes par ces systèmes, il est nécessaire d'y adjoindre une structure dite de *recouvrement* qui doit fournir des services essentiels tels que la recherche de données ou la découverte d'applications. L'efficacité de ces services étant étroitement liée à la topologie de cette structure de recouvrement, il est important de l'étudier et de la choisir avec soin. De plus, différents services peuvent utiliser différentes topologies de recouvrement.

Ici, nous nous focalisons plus précisément sur les services de recherche et de diffusion. Classiquement, lorsqu'on veut effectuer des recherches sur un réseau *peer-to-peer*, on a le choix entre deux grandes familles de solutions : soit on cherche dans un annuaire, soit les différentes ressources sont directement interrogées.

Pouvoir disposer d'un annuaire implique d'une part la présence d'une structure centralisée liée aux données à rechercher, et d'autre part, que les différentes ressources ne soient pas

placées n'importe où dans la structure mais selon certaines règles. Cela ne correspond pas au contexte évoqué plus haut et nous optons pour une solution permettant d'interroger directement les ressources.

Il s'agit donc de mettre en œuvre une topologie permettant d'effectuer des recherches ou des diffusions sur des réseaux *peer-to-peer* tels que nous venons de les décrire. Pour être efficace, supporter le passage à l'échelle et perturber le moins possible les applications et services que les pairs doivent fournir, cette topologie de recouvrement doit utiliser un minimum de ressources – bande passante, puissance de calcul et mémoire. Elle doit en outre bien supporter la forte dynamicité du système ainsi construit, et offrir une bonne tolérance aux pannes.

Les topologies majoritairement utilisées pour cet usage sont les arbres et les graphes, bien adaptés aux algorithmes de recherche par inondation. Ils ont été largement étudiés et leurs propriétés sont bien connues. Les arbres peuvent être parcourus avec peu de messages, mais la charge y est mal répartie (seuls les nœuds intermédiaires participent à la transmission des messages alors qu'ils sont moins nombreux que les feuilles) et leur racine constitue un important goulot d'étranglement. D'un autre côté, les graphes répartissent bien la charge entre leurs différents nœuds, mais nécessitent un grand nombre de messages et consomment donc beaucoup de bande passante.

Le LIFC[1] de Besançon a proposé une nouvelle structure originale, le DST – *Distributed Spanning Tree* – qui combine le meilleur de ces deux familles, ainsi que cela a été montré dans la thèse de Sylvain Dahan. [Dah05] Voici un extrait de son *abstract* :

> *"Afin d'améliorer les performances des découvertes de services utilisant des algorithmes de recherches par voisinage, nous proposons une nouvelle topologie permettant de créer des arbres sans point de contention. Pour y parvenir, nous avons réussi à créer un arbre où tous les nœuds jouent à la fois le rôle de feuille, de racine et de nœuds intermédiaires. [...]*
>
> *La nouvelle topologie d'arbre proposée est dénuée des points de contention typiques des arbres classiques. Si chaque nœud de l'arbre était un ordinateur, des ordinateurs seraient des feuilles se contentant de recevoir des messages et les autres ordinateurs seraient des nœuds intermédiaires acheminant les messages de recherche. Afin de mettre tous les ordinateurs sur un pied d'égalité, nous distribuons le rôle des nœuds intermédiaires entre plusieurs ressources. Ce nouvel arbre que nous proposons est construit de manière récursive de la manière suivante : toutes les ressources sont des feuilles ; un nœud intermédiaire est non pas une ressource mais un graphe complet de ses fils. Nous avons montré qu'un tel arbre était réalisable et qu'il permettait de réaliser des recherches par voisinage avec des algorithmes de parcours d'arbre tout en répartissant la charge d'une manière similaire aux parcours de graphe. Il en résulte des performances concernant la vitesse de recherche et de charge supportée qui sont*

1. Laboratoire d'Informatique de l'université de Franche-Comté

supérieures aux mêmes recherches sur des arbres classiques ou sur des graphes pseudo-aléatoires."

1.2 Présentation du problème

Dans ses travaux, Sylvain Dahan a montré les bonnes propriétés de cette nouvelle structure de façon théorique. En particulier, il les a comparées avec celles des arbres et des graphes conventionnels. Pour la partie pratique, il a mis au point les algorithmes de construction d'un DST et les a soumis à un simulateur qu'il a réalisé, permettant ainsi de vérifier la théorie.

Toutefois, ce simulateur était basé sur une architecture centralisée dans laquelle une horloge globale permettait de séquencer les opérations. Il a été conçu essentiellement pour réaliser des tests de performance – en comptant le nombre de messages échangés, par exemple – mais il n'est pas adapté à l'étude du comportement du DST en situation.

Dans un vrai réseau *peer-to-peer*, il n'y a pas de structure centralisée, donc pas d'horloge globale pour cadencer les différents événements. Du fait de la parallélisation existante dans ce type de structure, ces événements se produisent donc non seulement n'importe quand, mais éventuellement simultanément. Et il faut ajouter à cela l'influence de l'état du réseau utilisé : sur un réseau tel qu'Internet, la bande passante disponible et la latence ne peuvent pas être maîtrisées. On n'y maîtrise donc pas non plus les instants d'arrivée des messages envoyés.

Ces contraintes vont bien sûr avoir des conséquences sur le fonctionnement du DST et font apparaître des problèmes. En particulier, la parallélisation et l'absence d'horloge globale provoquent des problèmes de synchronisation qu'il faut solutionner. Certaines opérations mises en œuvre dans la vie du DST ont besoin d'être ordonnancées, et il faut pouvoir gérer les ajouts et/ou retraits simultanés. De plus, il est maintenant nécessaire de contrôler la cohérence de la structure pendant les phases de construction/destruction. Dans un simulateur séquentiel, l'ordre maîtrisé des opérations assurait cette cohérence, ce qui n'est plus le cas en environnement réel.

Les travaux présentés ici prennent donc la suite de ceux de Sylvain Dahan. Vu la complexité des algorithmes distribués mis en œuvre, les prouver formellement est difficile. Ma démarche sera donc de montrer qu'ils sont justes en effectuant une validation expérimentale. Il s'agit (a) de porter les algorithmes de construction du DST sur un simulateur basé sur des modèles plus proches du monde réel, (b) d'étudier et de proposer des solutions aux problèmes de synchronisation qui se posent alors et enfin, (c) de proposer de nouveaux algorithmes gérant le départ de nœuds du DST, également soumis à ce nouveau simulateur. Ce travail sera donc à compléter par une étude supplémentaire visant à démontrer et prouver ces algorithmes de façon formelle.

Comme on le verra – et avant de parler de problèmes de synchronisation – le fait de porter ces algorithmes sur ce nouveau simulateur a révélé quelques erreurs dans leur conception,

qui ne se produisaient certainement pas avec le simulateur initial du fait que l'ordre des opérations y était toujours identique. Mais la difficulté principale de ce travail réside dans le très grand nombre de cas de figure qui peuvent apparaître lors du passage à l'échelle et ce, de façon aléatoire. La parallélisation massive des tâches, l'importante profondeur de récursivité de certaines fonctions clés, le fait qu'on ne maîtrise pas l'instant d'arrivée des différents événements et le recours à des méthodes de type `RPC` – *Remote Procedure Call* – pour exécuter les différentes fonctions sont les principales difficultés à surmonter.

Une solution possible aurait pu être de mettre en place un mécanisme à base de *mutex* distribués pour n'autoriser l'arrivée que d'un seul nœud à la fois dans le DST. Mais il m'a semblé que cette méthode aurait eu deux inconvénients :

1. si un grand nombre de nœuds cherchent à rejoindre le DST dans un court laps de temps, alors il faut organiser de grandes files d'attentes, le temps que les nœuds précédents aient terminé de rejoindre le DST. Ces files n'étant pas extensibles à l'infini, cela pourrait poser des problèmes d'extensibilité.
2. les performances de construction en seraient probablement très affectées. En ne laissant les nouveaux nœuds rejoindre le DST que l'un après l'autre, il paraît évident que le temps total pour qu'un groupe soit intégré sera plus long que s'ils peuvent tous entrer en même temps.

J'ai donc fait le choix d'étudier une solution qui tente de tirer parti au maximum de la parallélisation. Toutefois, lorsque les arrivées de différents nouveaux nœuds impactent des zones communes du DST, cette solution va alors nécessiter la mise en place de synchronisations plus ou moins complexes et donc, plus ou moins consommatrices de bande passante, ainsi que nous allons le voir.

Dans la suite de cet exposé, nous nous intéresserons au contexte d'utilisation du DST, puis le DST lui-même ainsi que le simulateur *Simgrid* seront présentés. En deuxième partie, nous rentrerons dans le vif du sujet avec le compte-rendu des travaux réalisés. Nous commencerons par le portage des fonctions de construction du DST sur Simgrid et l'étude des problèmes de synchronisation que cela fait apparaître. Après quoi nous présenterons les nouveaux algorithmes de gestion des retraits de nœuds du DST. Nous terminerons en examinant les perspectives qu'ouvre cette étude avant de conclure. Les travaux seront détaillés en annexes.

Chapitre 2

État de l'art

Sommaire

Dans cette partie, nous allons faire un rapide tour d'horizon de ce qui se fait par ailleurs en matière de structures de recouvrement de réseaux *peer-to-peer*. Ainsi que nous le verrons, ces solutions ne sont pas pleinement adaptées au contexte qui nous intéresse.

2.1 Autres travaux

Le DST n'est bien sûr pas la première tentative de réalisation d'une structure de recouvrement de réseaux *peer-to-peer*. On peut citer ici une étude comparative des différentes solutions existantes : "*A survey and comparison of peer-to-peer overlay network schemes*" [LCP+05]. Dans le cadre des structures adaptées aux environnements dynamiques, cette étude classe les différentes solutions proposées selon deux familles principales : les systèmes structurés et les systèmes non structurés.

Dans l'ensemble des systèmes structurés, le principe est d'associer une clé à une donnée particulière (au moyen d'une fonction de hachage), et d'organiser les différents pairs selon un graphe qui permettra le rangement des clés chez des pairs déterminés. Connaissant une clé, il devient ainsi possible de savoir qui en est responsable et de s'adresser à lui pour retrouver les données associées. C'est le principe des *DHT – Distributed Hash Tables*

[WGR05]. Il s'agit d'une amélioration des systèmes basés sur un unique index (comme Napster, par exemple) puisqu'ici, l'index est distribué. Dans cette famille, on peut citer *Chord* [SMK+01], *CAN* [RFH+01] ou *Pastry*. [AD01]

Bien que les systèmes structurés possèdent de bonnes propriétés démontrées et éprouvées, l'étude citée plus haut souligne quelques défauts : consommation inutile de bande passante, goulots d'étranglement possibles, vulnérabilité à la sécurité. Il faut en outre souligner que les recherches effectuées au moyen de ces structures sont des recherches exactes (on parle de *Key Based Routing* [Kta09]). Si les recherches souhaitées utilisent des mots-clés, des expressions ou d'autres critères (ce qui est le cas pour la recherche d'applications ou de services), elles sont susceptibles de retourner un ensemble de réponses, ce que ces protocoles ne savent pas gérer seuls.

2.1.1 Un exemple de système structuré : Chord

À titre d'exemple, nous pouvons examiner ce protocole plus en détails. Chord est une des déclinaisons possible des DHT et en possède donc les propriétés : bon équilibrage de charge, bonne extensibilité et robustesse.

Essentiellement, ce protocole doit spécifier :

1. comment attribuer les clés aux différents nœuds
2. comment retrouver les clés
3. comment intégrer les nouveaux nœuds
4. comment gérer leur départ (ou leur défaillance)

Génération et distribution des clés

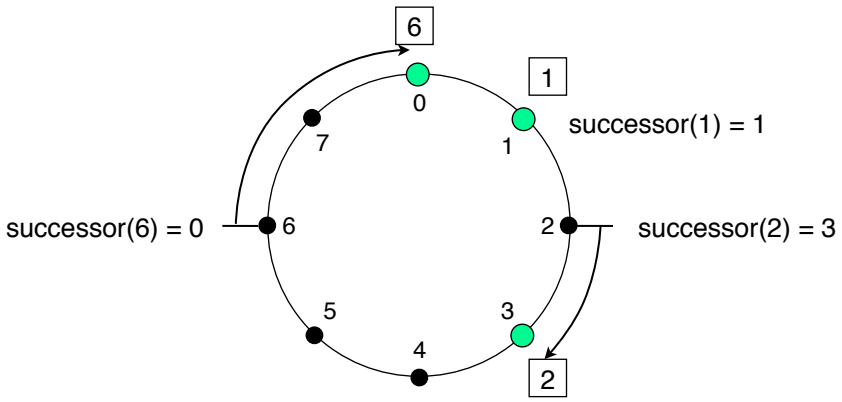

FIGURE 2.1 – Un cercle d'identifiants constitué des 3 nœuds 0, 1 et 3. Sur cet exemple, la clé 1 est située sur le nœud 1, la clé 2 est sur le 3 et la 6 sur le 0.

Pour attribuer les clés aux nœuds, Chord utilise le *consistent hashing* [KLL+97] qui permet deux propriétés essentielles :

- les nœuds reçoivent approximativement le même nombre de clés (nécessaire pour une bonne répartition de charge)
- lorsqu'un nœud rejoint ou quitte le réseau, seule une petite fraction des clés doit être déplacée sur d'autres nœuds. (nécessaire pour limiter le coût de maintenance des tables de routage)

SHA-1 est utilisée comme fonction de hachage de base pour générer des identifiants de m bits. (Il est important de choisir m suffisamment grand pour éviter les collisions.) Chaque nœud et chaque clé se voit ainsi attribuer un identifiant.

Les 2^m identifiants sont ordonnés sur un *cercle d'identifiants* (modulo 2^m) comme on le voit sur la figure 2.1. Une clé k est attribuée au premier nœud qui possède le même identifiant qu'elle, ou à défaut, qui le suit immédiatement sur le cercle. Chaque nœud devient ainsi responsable des clés situées dans l'intervalle compris entre lui-même et son prédécesseur sur le cercle.

Pour assurer un cheminement correct des requêtes de recherche, le minimum requis est que chaque nœud connaisse son successeur et son prédécesseur. Chord fait donc en sorte que cette information soit toujours à jour. Mais pour améliorer les performances, on montre que chaque nœud n'a besoin de connaître que $O(\log N)$ autres nœuds en plus, améliorant ainsi l'extensibilité du *constistent hashing* (qui lui, demande que chaque nœud puisse contacter *tous* les autres). Chord assure la mise à jour de ces tables de routage en $O(\log^2 N)$ messages.

> **Les nœuds virtuels** : La répartition des nœuds dans l'espace d'identification (c'est à dire sur le cercle) n'est pas forcément uniforme, ce qui nuit à la répartition de charge : certains nœuds pourraient se voir attribuer davantage de clés que leurs voisins.
>
> On solutionne ce problème à l'aide de nœuds virtuels. En ajoutant suffisamment de ces nœuds virtuels sur le cercle, l'espace d'identification est partagé plus équitablement. On montre ensuite que si on affecte $\log N$ nœuds virtuels (choisis aléatoirement) à chaque nœud réel, on atteint une répartition de charge bien plus équitable.

Recherche d'une clé

La recherche d'une clé consiste à rechercher le successeur de l'*id* de la clé. Chord s'y prend donc en deux temps : il recherche tout d'abord le prédécesseur n' du nœud recherché n, et le successeur de n' est alors le nœud n recherché.

Pour trouver le prédécesseur n', on approche de la cible par sauts successifs (en utilisant

la plus grande entrée inférieure à l'*id* de la table de routage courante [1]) jusqu'à ce que l'*id* recherché soit compris entre le nœud courant et son successeur ; le nœud courant est donc n'. Ce dernier connaissant son successeur, il ne reste plus qu'à l'interroger pour trouver n.

Avec Chord, tant que le temps mis pour reconstruire les tables de routage ($O(\log^2 N)$) est inférieur au temps pris par le réseau pour doubler sa taille, les recherches continueront à se faire en $O(\log N)$ étapes.

Les recherches peuvent continuer à aboutir, même pendant la mise à jour des tables de routage, au prix toutefois de performances dégradées puisque la recherche ne pourrait alors utiliser que les informations de successeurs.

Lors de l'arrivée d'un nouveau nœud

Pour intégrer un nouveau nœud n sans perturber les recherches, Chord doit réaliser 3 tâches :

1. Initialiser le prédécesseur et la table de routage de n
2. Mettre à jour les prédécesseurs et les tables de routage des nœuds existants pour tenir compte de ce nouveau nœud.
3. Indiquer à l'application construite sur Chord qu'elle peut transférer les données concernées vers ce nouveau nœud.

1 – Pour que n puisse s'insérer dans le système, il doit connaître un nœud quelconque n' appartenant déjà au système. n va donc pouvoir demander à n' de calculer son prédécesseur et sa table de routage. Le successeur de n est donc le premier nœud situé après son *id* sur le cercle, et son prédécesseur est le prédécesseur de ce successeur. (puisqu'il n'est pas encore "au courant" de la présence de n)

Ensuite, on remplit la table de routage de n en recherchant les successeurs de chaque entrée de cette table. Pour améliorer les performances, on n'effectue cette recherche que si on est sûr que l'intervalle considéré n'est pas vide.

2 – L'idée ici est de parcourir le cercle à l'envers, pour trouver les nœuds dont une entrée de la table de routage devrait pointer sur n. Chaque entrée concernée des tables de ces nœuds est mise à jour en conséquence.

3 – n ne peut devenir responsable que des clés auparavant attribuées à son successeur immédiat. Il n'a donc besoin de contacter que ce seul nœud pour connaître les clés impactées et les indiquer à l'application.

1. Dans cette table de routage de m nœuds, la distance entre deux nœuds m_i et m_{i+1} est le double de celle entre m_{i-1} et m_i. Lors de la recherche, chaque saut divise donc au moins par deux la distance restant à parcourir.

Lors du départ (ou d'une défaillance) d'un nœud

Lors de la défaillance d'un nœud n, les nœuds qui le pointaient doivent pouvoir trouver son successeur. À cette fin et en plus des tables de routage, chaque nœud maintient une liste de ses r successeurs. Ainsi, lors de la défaillance de n, son prédécesseur pourra le remplacer par son plus proche successeur vivant.

Ces tâches sont difficiles à réaliser lors d'arrivées et/ou de départs simultanés en grand nombre dans le réseau. Ainsi, Chord a recours à une routine dite de "stabilisation" qui est exécutée périodiquement par chaque nœud du réseau. Cette routine a pour but de maintenir à jour les listes de successeurs et les tables de routage.

Lorsqu'un nœud n exécute cette routine, il cherche à connaître le prédécesseur de son successeur. Si ce n'est pas n, c'est qu'un nouveau nœud s'est inséré et qu'il doit donc être le nouveau successeur de n. De plus, n signale son existence à son successeur, permettant ainsi à ce successeur de mettre à jour son prédécesseur.

Quant aux tables de routage, leurs entrées sont interrogées aléatoirement et mises à jour le cas échéant.

⋆ ⋆ ⋆

Les systèmes non structurés quant à eux, organisent leurs pairs selon des graphes semi-aléatoires plats ou hiérarchiques. Ils sont bien adaptés à la recherche par voisinage par inondation ou par vagues et sont davantage extensibles. Dans cette famille, on peut citer *Gnutella* [SR04], *KaZaA* ou *eDonkey*.

2.1.2 Un exemple de système non structuré : Gnutella

Gnutella est un protocole décentralisé permettant d'effectuer des recherches distribuées sur une topologie à plat de pairs appelés "servants". Bien que Gnutella permette la recherche client/serveur centralisé classique, il se distingue par le modèle *peer-to-peer*, décentralisé mis en œuvre pour le rangement et la recherche de documents, comme le montre la figure 2.2.

Dans ce modèle, chaque pair peut être à la fois serveur et client. Ce système n'est pas un annuaire centralisé et il n'a pas de contrôle précis sur la topologie, pas plus que sur la façon dont les données y sont rangées. Le réseau est formé de pairs qui le rejoignent selon des règles peu contraignantes. Le réseau qui en résulte possède certaines propriétés, mais le placement des différentes données se fait sans avoir de connaissance particulière de la topologie comme c'est le cas dans les systèmes structurés. Pour situer une donnée sur le réseau, un pair doit interroger ses voisins et la méthode de recherche typique dans ce cadre est la recherche par inondation. La requête est ainsi propagée de proche en proche sur un périmètre donné. Cette conception supporte très bien la forte dynamicité du réseau (des pairs le rejoignent ou le quittent à tout instant). Cependant, les mécanismes de recherche actuels ne sont pas très extensibles du fait de la grande charge qu'ils génèrent sur le réseau.

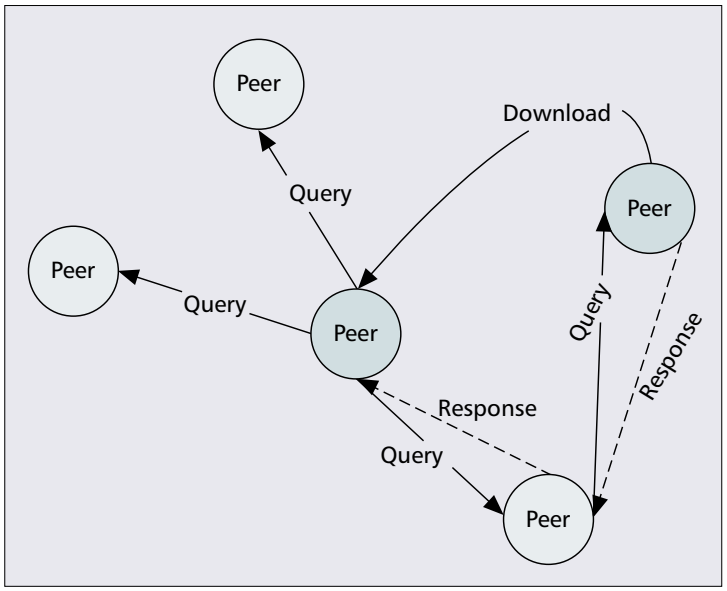

FIGURE 2.2 – Gnutella utilise une architecture décentralisée pour ranger et rechercher des documents (source : [LCP+05])

Les servants accomplissent des tâches normalement dévolues à la fois à des clients et à des serveurs. Ils fournissent une interface client permettant aux utilisateurs de lancer leurs recherches et recevoir leurs réponses. En même temps, ils acceptent des requêtes provenant d'autres servants, les traitent et leur répondent. Ces pairs sont chargés de gérer tout le trafic permettant le maintient de l'intégrité du réseau. De part sa nature distribuée, un tel réseau de servants est très tolérant aux pannes : un ensemble de pairs qui quitteraient le réseau ne provoquerait pas l'arrêt des opérations en cours.

Pour rejoindre le système, un nouveau servant (un pair, donc) contacte l'un de ses hôtes presque toujours disponibles dont la liste est disponible sur Internet. Dès qu'ils sont raccordés au réseau, les pairs envoient des messages pour interagir entre eux. Ces messages peuvent être diffusés vers les autres pairs dont on a connaissance, ou retournés aux pairs dont on vient de recevoir une diffusion, en suivant son chemin dans le sens inverse (c'est la propagation arrière). Trois règles sont respectées pour ces échanges :

1. chaque message possède un identifiant généré aléatoirement
2. chaque pair garde en mémoire les messages récemment routés, afin d'éviter de nouvelles diffusions inutiles ; cela lui permet également de conserver les informations utiles à la propagation arrière
3. les sauts sont comptés pour limiter les inondations à un périmètre donné (c'est le principe appelé *TTL* pour *Time To Live*)

Il y a trois familles de messages sur le réseau :

– pour l'appartenance aux groupes : un pair qui veut signaler sa présence diffuse un message *PING* à son entourage qui répond – au moyen de la propagation arrière –

un message *PONG* contenant des informations telles que l'adresse IP, la taille et le nombre de données stockées, etc.

- pour les recherches : un pair qui reçoit un message *QUERY* contenant des critères de recherche, effectue une recherche localement puis le propage à ses voisins. En réponse, un message *QUERY RESPONSE* est propagé dans le sens inverse pour fournir au demandeur les informations nécessaires au téléchargement des données recherchées.
- pour le transfert de données : les téléchargements se déroulent directement entre deux pairs utilisant les messages *GET* et *PUSH*.

Ainsi, un pair qui souhaite faire partie du réseau doit ouvrir un certain nombre de liens avec d'autres. Cet environnement étant très dynamique, chaque pair doit régulièrement mettre à jour la connaissance qu'il a de son voisinage au moyen de messages *PING*. Ce système constitué de servants reliés entre eux par des connections TCP forme donc un réseau auto-organisé et dynamique d'entités indépendantes.

Pour améliorer les performances de routage sur ce type de réseau, les dernières évolutions de Gnutella utilisent des *super-pairs* fournissant une plus grande bande passante. Un nouveau pair souhaitant rejoindre le réseau va alors se raccorder à l'un de ces super-pairs pour lui fournir sa liste de données partagées. Une requête émise par un autre pair va alors être adressée à son super-pair, qui va la propager de super-pair en super-pair, jusqu'à celui auquel est rattaché le pair possédant la ressource. Toutefois, la recherche par inondation mise en œuvre pour la recherche entre les super-pairs continue de limiter l'extensibilité du système. De plus, cette approche pose le problème du choix des super-pairs (quel pair peut ou pas être considéré comme un super-pair, selon quels critères), ainsi que de son évolution dans le temps.

⋆⋆⋆

Dans le cadre qui nous intéresse, nous souhaitons utiliser une structure qui privilégie la robustesse, l'extensibilité et la tolérance aux pannes. Nous souhaitons donc qu'aucun pair ne joue de rôle prépondérant, ce qui exclut les solutions basées sur des index ou les *super-nœuds* (comme *KaZaA* et *Gnutella* que nous venons de voir). Nous souhaitons également nous affranchir des points de contentions possibles et limiter la bande passante requise par les recherches. Le DST a été conçu pour répondre à ces besoins.

2.2 Présentation du DST

La topologie originale de recouvrement étudiée ici se nomme donc *DST* pour *Distributed Spanning Tree*. Ainsi que cela a été dit plus haut, l'idée du DST est de réunir les qualités des graphes et des arbres : limiter le nombre de messages échangés comme le ferait un arbre tout en répartissant la charge comme un graphe. Dans cette topologie, chaque serveur agit à la fois comme feuille, comme nœud intermédiaire, ou comme racine selon l'étage sollicité ; chaque ordinateur est la racine de son propre arbre de recouvrement.

Le principe général de cette structure est qu'un nœud parent est formé du graphe complet de ses enfants. Le DST se comporte alors comme un arbre en ce qui concerne le nombre de messages échangés, mais évite ses goulots d'étranglement en répartissant la charge de chaque nœud entre chacun de ses fils. Du fait que dans un DST, chaque ordinateur n'a besoin de mémoriser qu'une faible quantité de données (en logarithme du nombre de nœuds) pour pouvoir contacter les autres nœuds de la structure, il supporte bien le passage à l'échelle. Cette structure est bien adaptée aux algorithmes de recherche et de diffusion. Il permet également la découverte de services.

2.2.1 Les trois niveaux de description du DST

Le niveau logique

Le niveau logique est une vue abstraite de la structure dans laquelle les nœuds – dont on rappelle qu'ils sont des groupes d'ordinateurs – sont reliés entre eux par des liens abstraits respectant deux règles de base :

1. tout nœud parent est le graphe complet de ses enfants
2. tout nœud qui n'est ni une feuille ni la racine possède entre a et b enfants (la racine peut avoir moins de a enfants et une feuille n'a bien sûr pas d'enfant)

Le DST étant un arbre, c'est une structure récursive dans laquelle chaque étage se construit avec des éléments de l'étage inférieur.

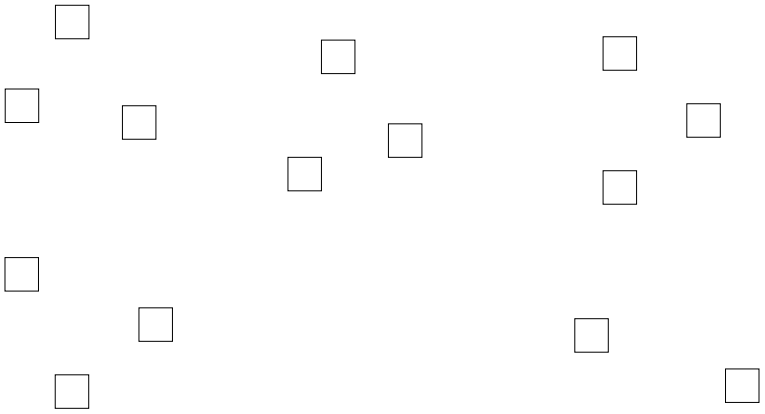

FIGURE 2.3 – Niveau logique – étage 0

L'étage 0 (figure 2.3) contient les feuilles du DST. Chaque feuille est un ordinateur et chaque ordinateur participant au DST doit être une feuille.

L'étage $n + 1$ ($n \geq 0$) (figures 2.4 et 2.5) contient les parents des nœuds de l'étage n. Comme indiqué plus haut, chaque parent est le graphe complet de ses enfants. Ici, $a = 2$ et $b = 3$.

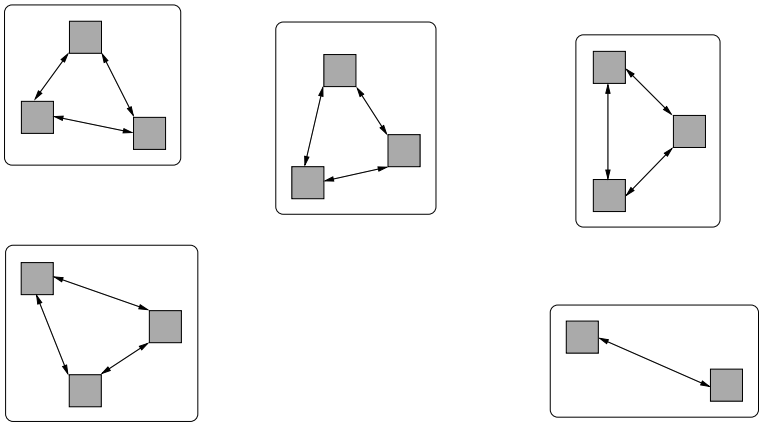

FIGURE 2.4 – Niveau logique – étage 1

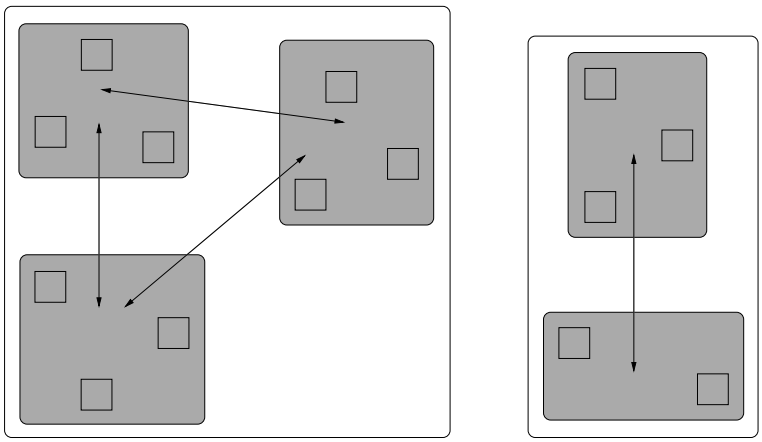

FIGURE 2.5 – Niveau logique – étage 2

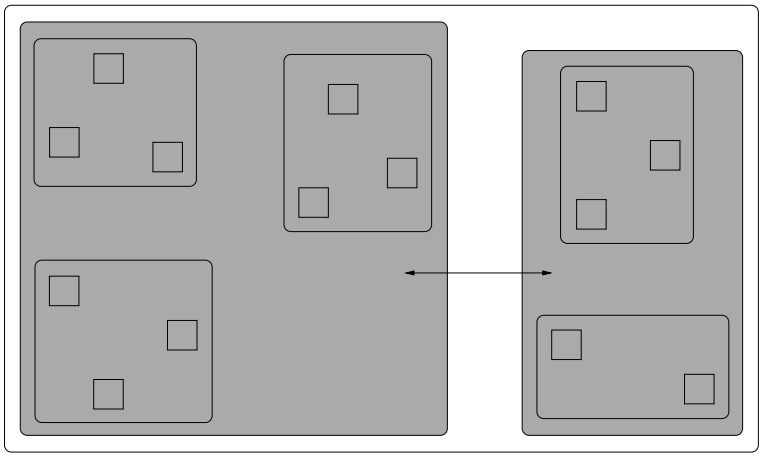

FIGURE 2.6 – Niveau logique – étage 3

L'étage h (si h est la hauteur du DST) (figure 2.6) est la racine du DST et peut donc avoir moins de a enfants.

Le niveau d'interconnexion

Ce niveau est la mise en œuvre concrète de la vue logique. Il décrit comment les nœuds sont distribués entre les ordinateurs et comment les liens logiques inter-nœuds sont réalisés avec des liens physiques.

Il faut introduire ici la notion de représentant d'un nœud : pour un étage donné, un représentant d'un nœud est un des descendants de ce nœud, dont le rôle est de servir de contact à certains de ceux qui ont besoin de communiquer avec le nœud qu'il représente. Concrètement, trois règles doivent être ici respectées :

1. si un lien logique existe entre deux nœuds A et B, alors :
   (a) chaque descendant de A doit posséder un lien physique vers un des descendants de B (qui devient alors le représentant de B pour ce descendant de A)
   (b) chaque descendant de B doit posséder un lien physique vers un des descendants de A (qui devient alors le représentant de A pour ce descendant de B)
2. pour des raisons de répartition de charge et de tolérance de panne, les descendants d'un nœud A ne doivent pas tous être liés avec le même descendant de B et vice-versa (Autrement dit, les descendants de A ne doivent pas utiliser le même représentant de B).
3. à chaque étage du DST, tout ordinateur s'utilise lui-même comme représentant des nœuds auxquels il appartient

Les figures 2.7, 2.8 et 2.9 montrent ces liens TCP/IP à chaque étage. [2]

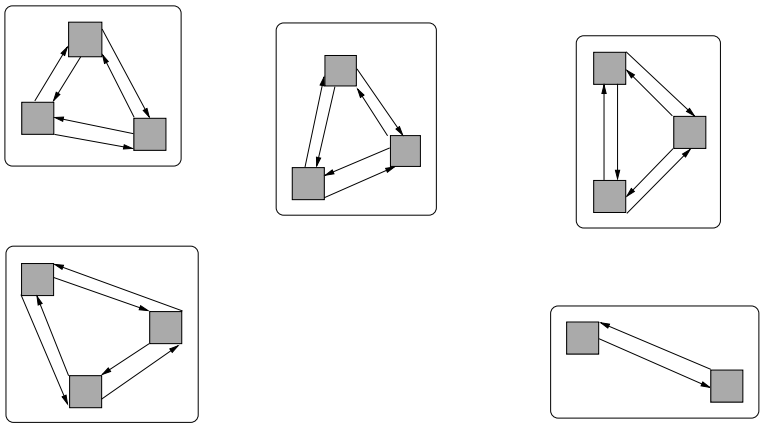

FIGURE 2.7 – Niveau d'interconnexion – étage 1

Comme on peut le voir, chaque membre d'un nœud est capable de joindre chacun de ses nœuds frères via l'un de ses membres et ce, à chaque étage.

2. Pour ces raisons de clarté, le lien qu'un ordinateur possède avec lui-même n'y figure pas.

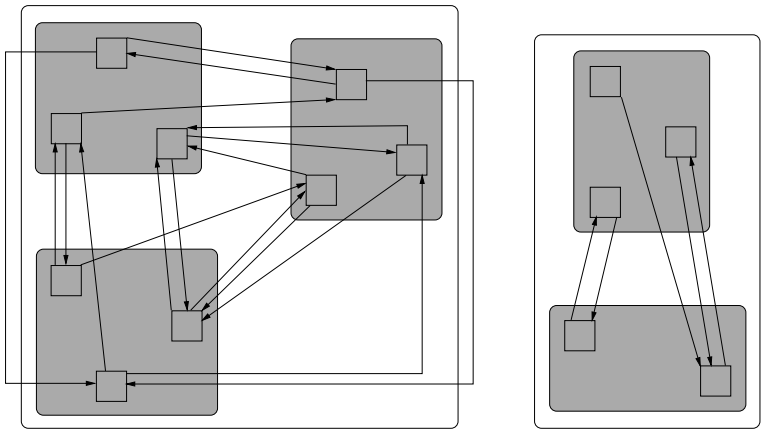

FIGURE 2.8 – Niveau d'interconnexion – étage 2

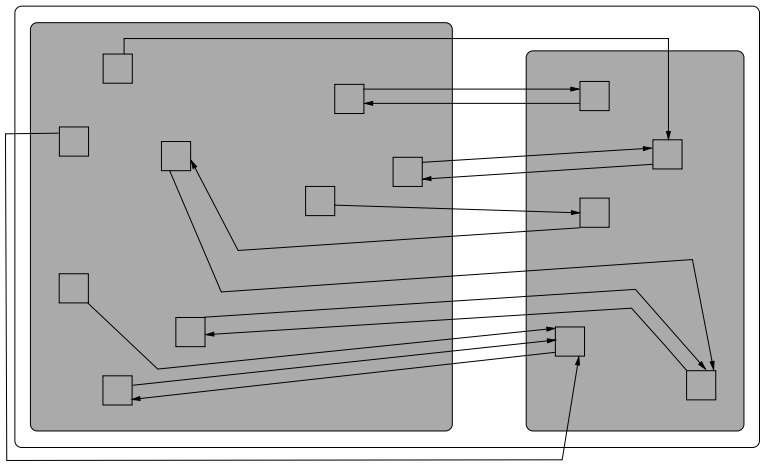

FIGURE 2.9 – Niveau d'interconnexion – étage 3

Concrètement, chaque ordinateur possède une *table de routage* qui contient les adresses des membres qu'il peut joindre – ses frères, donc – à chaque étage. Il possède également une *table des prédécesseurs* qui lui permet de connaître l'ensemble des ordinateurs qui l'utilisent comme représentant du nœud courant. Le nombre de prédécesseurs pour un étage constitue donc la *charge* de cet ordinateur pour cet étage.

Le niveau topologique

Dans ce niveau, on s'intéresse à la manière de placer cette structure sur un réseau réel. Deux exemples :

- sur Internet : en transposant la structure du DST sur celle, hiérarchique, d'Internet, on s'assure que la majorité des échanges a lieu dans les LANs où les performances sont les meilleures.
- sur un réseau de type sémantique : il est possible de réaliser les groupes par centres

d'intérêts communs, là encore pour concentrer les recherches – et donc les échanges – dans les parties où elles ont le plus de chances d'aboutir.

À noter que ce niveau constitue un domaine de recherche récent.

2.3 Le simulateur Simgrid

2.3.1 Pourquoi Simgrid

En matière d'applications distribuées, un des principaux problèmes auxquels on se trouve confronté est de pouvoir scientifiquement comparer différentes solutions entre elles, en fonctions de métriques données. (temps de réponse moyen de recherches, probabilité de disponibilité de certains services, etc ...) Dans de très rares cas, il est possible de le faire sur la seule base d'études théoriques, mais la plupart du temps, on est obligé d'avoir recours à des évaluations empiriques obtenues lors d'expérimentations pratiques.

Outre le fait que cela demande beaucoup de travail, réaliser des tests sur des plates-formes réelles ne permet pas d'obtenir des résultats reproductibles, principalement à cause de la non maîtrise de paramètres comme la charge du réseau à un moment donné. Les émulateurs eux aussi sont difficiles à utiliser et le sont donc rarement par les chercheurs du domaine. Il existe également plusieurs simulateurs de réseaux [ns2, CNO99, Ril03], mais ils sont mal adaptés à la simulation d'applications distribuées telles que celles qui nous intéressent.

Il existe un certain nombre de simulateurs très spécialisés, développés isolément pour les besoins de communautés précises. Comme on peut s'y attendre, ils deviennent difficiles à utiliser dès lors qu'on sort du domaine pour lequel ils ont été conçus. De plus, leur nombre est un obstacle à la comparaison des différents résultats obtenus.

Simgrid est un *framework* qui a été conçu pour la simulation d'un large éventail d'applications distribuées sur toutes sortes de plates-formes distribuées. Il possède trois caractéristiques essentielles pour notre usage : (1) le moteur du simulateur utilise des modèles de réseau connus et validés. Il est ainsi capable de simuler différentes topologies de réseau, ainsi que les ressources fournies par les plates-formes (puissance de calcul, état du réseau) de façon dynamique. Il peut aussi simuler les défaillances. (2) Simgrid possède des interfaces évoluées aisément utilisables par les chercheurs qui peuvent réaliser leurs prototypes de simulation en C ou en Java. (3) Simgrid met à disposition des développeurs d'applications distribuées les APIs nécessaires leur permettant de passer facilement du mode "simulation" au mode "monde réel".

2.3.2 Présentation de Simgrid

Simgrid 3.6.1 (la version utilisée pour cette étude) est présenté ici :
http://simgrid.gforge.inria.fr/3.6.1/doc/ (voir aussi [CLQ08])

Il y est défini comme :

> *... une boîte à outils mettant à disposition l'ensemble des fonctionnalités nécessaires à la simulation d'applications distribuées en environnements distribués hétérogènes.*

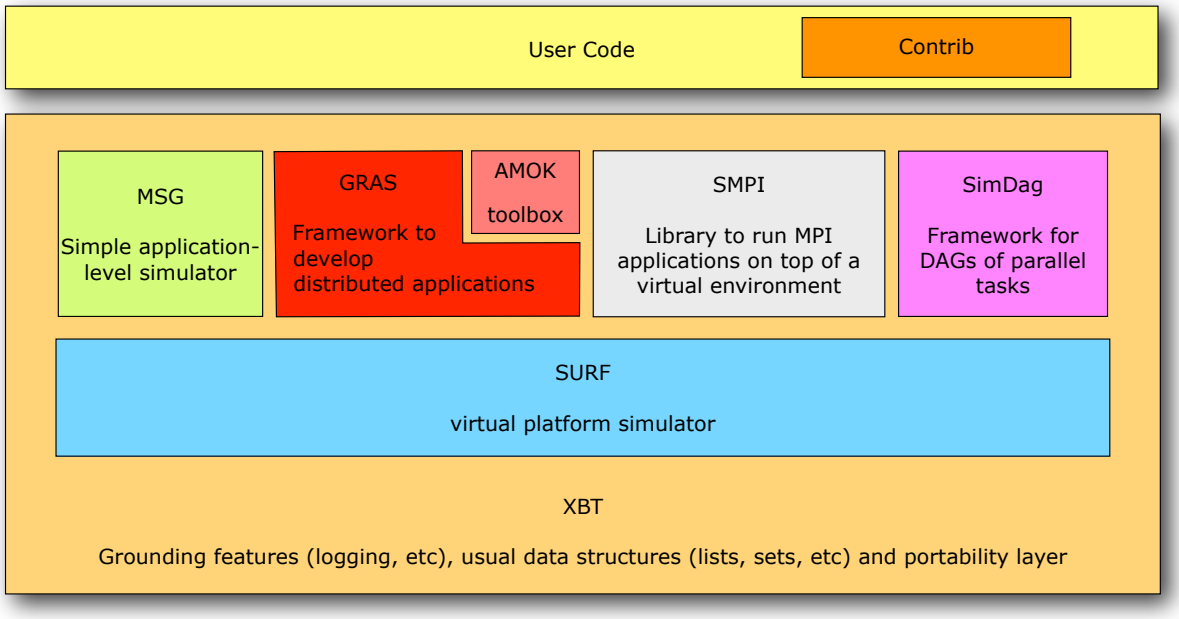

FIGURE 2.10 – Les différents composants de Simgrid

Comme le montre la figure 2.10, l'architecture de Simgrid est modulaire et multi-couches :

En haut On trouve la couche des interfaces utilisateurs qu'on peut scinder en deux catégories : *SimDag* est destiné aux chercheurs désireux d'étudier leurs algorithmes, alors que *GRAS* s'adresse plutôt aux développeurs souhaitant éprouver leurs applications. *MSG* peut être utilisé dans ces deux contextes.

MSG est l'interface la plus utilisée. C'est l'API de choix pour étudier des problèmes théoriques et comparer des heuristiques. Elle peut travailler sur les algorithmes d'ordonnancement, de *peer-to-peer* ou des grilles de calcul. Elle a été conçue pour être utilisée en C, mais Java est également proposé.

GRAS (*Grid Reality And Simulation*) permet le développement d'applications distribuées. Cette API est implémentée deux fois : une fois pour la simulation et une fois pour le monde réel. Il est ainsi possible de facilement passer d'un mode à l'autre sans modification de code. Il suffit d'utiliser la bibliothèque souhaitée lors de la compilation.

SMPI (*simulated MPI*) est utilisée pour directement exécuter des applications MPI [3] sur le simulateur sans modification de leur code, en interceptant les primitives MPI.

3. *Message Passing Interface*

SimDag est conçue pour travailler sur des heuristiques d'ordonnancement d'applications vues comme des graphes de tâches. Elle permet de créer les tâches et leurs dépendances, de planifier leur exécution sur certaines ressources et finalement de calculer le temps d'exécution globale du graphe.

Au milieu **SURF** est le moteur de simulation proprement dit. Il est hautement modulaire, de façon à pouvoir utiliser différents modèles de réseaux et de plates-formes, rendant ainsi leur comparaison possible. De plus, il est conçu pour permettre aux chercheurs d'y inclure librement de nouveaux modèles. SURF étant la base de Simgrid, il a été très optimisé de sorte qu'il ne vienne pas perturber les performances de la simulation.

Au dessus de *SURF*, il faut également mentionner le module *SimIX* qui permet la mise en œuvre de multiples process concurrents.

En bas On trouve la couche **XBT** (pour *eXtended Bundle of Tools*). Il s'agit d'une collection d'outils génériques de base tels que : types de collections de données, gestion des logs, des exceptions, etc ...

Enfin, il faut également préciser que Simgrid est fourni gratuitement sous licence LGPL et qu'il a été porté sous Linux, Windows, Mac OS X and AIX.

Les algorithmes du DST étudiés ici ont été implémentés avec l'API MSG.

Deuxième partie

Réalisations

Chapitre 3

Ajout de nouveaux sommets à un DST

Sommaire

3.1 Mise en œuvre dans Simgrid

Dans sa thèse [Dah05], Sylvain Dahan a conçu un algorithme distribué destiné à intégrer de nouveaux sommets dans un DST. Cet algorithme se compose d'un ensemble de fonctions exécutées par les différents sommets concernés selon la méthode *RPC*[1]. Il a décrit

1. *Remote Procedure Call*

ces fonctions et les a implémentées dans un simulateur qu'il a également réalisé (en Python) pour pouvoir en étudier le comportement et les performances. Ce simulateur était séquentiel – les opérations se déroulant les unes à la suite des autres, sans parallélisation possible – et possédait une architecture centralisée offrant à un nœud un accès direct aux structures de données décrivant les autres nœuds.

Comme expliqué page 5, notre but dans cette étude est de poursuivre ces travaux en étudiant le comportement du DST dans un environnement plus proche de la réalité des réseaux *peer-to-peer* – sans centralisation et avec parallélisation – d'où l'utilisation cette fois, du simulateur *Simgrid*. (voir section 2.3 page 18)

La première partie du travail a donc consisté à implémenter ces fonctions dans *Simgrid* – en utilisant l'API *MSG* en langage *C* – pour y faire fonctionner le DST. Ce travail, ainsi que les quelques modifications d'algorithmes qu'il a impliquées, est présenté en détails en **annexe A** (page 79). Il a permis la construction de plates-formes jusqu'à 4 000 nœuds.

3.1.1 Quelques notions de base au sujet de Simgrid et MSG

> **Remarque préalable** : Dans Simgrid – tout comme dans le domaine des topologies de recouvrement en général – ce qu'on appelle des nœuds sont des ordinateurs. Dans le DST, les nœuds sont des *groupes* d'ordinateurs. Pour éviter toute confusion lorsqu'on parle de DST, les ordinateurs seront plutôt appelés *sommets*. Le terme *nœud* a toutefois été conservé lorsqu'il n'y a pas d'ambiguïté ; par exemple, lorsqu'on dit qu'un "nouveau nœud joint le DST", on comprend bien qu'il s'agit d'un ordinateur.

La plate-forme

La plate-forme souhaitée est décrite dans un fichier XML. Deux formats sont possibles. [2]

Une première façon de faire est de décrire chaque ressource de la plate-forme en indiquant ses caractéristiques individuellement. On a donc :

des machines (`host`) auxquelles on attribue un identifiant et une puissance :
```
<host id="Gaston" power="98095000"/>
```

des liens réseau (`link`) qui reçoivent un identifiant, une latence et une bande passante :
```
<link id="127" bandwidth="3430125" latency="0.000536941"/>
```

des routes (`route`) entre les hôtes, utilisant les liens définis :
```
<route src="Gaston" dst="Marcel" symmetrical="NO"><link_ctn id="153"/>
<link_ctn id="155"/></route>
```

L'autre possibilité est d'indiquer qu'on souhaite utiliser un *cluster* de `x` machines identiques, toutes reliées entre elles par des liens identiques.

2. J'ai volontairement utilisé des descriptions simples, mais il est possible d'indiquer davantage de caractéristiques, comme le nombre de CPU par hôte, par exemple.

L'application

Les hôtes exécutent des *process*, appelés également *agents*[3]. Ceux-ci sont décrits et affectés aux hôtes, également dans un fichier XML, dit de "déploiement". En voici un exemple :

```
1  <process host="Gaston" function="node">
2      <argument value="57"/>         <!-- my id -->
3      <argument value="1"/>          <!-- known id -->
4      <argument value="1200"/>       <!-- time to sleep before it starts -->
5      <argument value ="20000"/>     <!-- deadline -->
6  </process>
```

Ici, on choisit d'avoir un *process* par hôte, chargé de se comporter comme un nœud qui voudrait rejoindre le DST. Dans *MSG*, ces *process* sont des fonctions écrites en C. ''node'' (ligne 1) est donc le nom de cette fonction, et les quatre valeurs qui suivent sont les arguments qui lui seront passés : l'identifiant de ce nouveau nœud, l'identifiant du contact connu par ce nouveau nœud et son heure d'arrivée. Lorsque la *deadline* est atteinte, il demande à quitter le DST.

Simgrid fournit des scripts en Python destinés à générer ces fichiers XML.

Le déroulement du programme

En environnement distribué, l'exécution des fonctions se fait au moyen d'échanges de messages. Dans Simgrid, on parle de *tâches*. Lorsqu'un *agent* souhaite qu'un autre exécute une fonction, il lui envoie une *tâche*. Celle-ci est caractérisée par la puissance de calcul nécessaire à son exécution, elle peut également être nommée et contenir des données.

Dans notre application, nous n'avons pas besoin de simuler le temps passé à exécuter une tâche et nous n'utilisons pas cette caractéristique de puissance de calcul. Par contre, le nom de la tâche permettra de différencier les messages (requêtes, réponses, etc ...) et les données attachées à une tâche seront le nom et les arguments de la fonction souhaitée. À réception de cette tâche, l'agent destinataire exécute la fonction demandée et retourne éventuellement une réponse, toujours sous la forme d'une tâche (dont les données sont alors toujours le nom de la fonction à laquelle on répond et les données de réponse proprement dites).

> *Remarque :* Dans Simgrid, plusieurs nœuds peuvent être simultanément en écoute puisqu'il y a un *thread* par nœud. Pour un nœud donné, j'ai choisi de n'utiliser qu'un seul *thread* qui traite les opérations de façon séquentielle : une fois un message reçu, le *thread* du nœud traite ce message puis revient se mettre en écoute. Simgrid offre la possibilité de créer plusieurs *threads* par nœud, auquel cas il serait possible d'avoir un *thread* chargé de l'écoute et un

3. Dans *Simgrid*, il y a un fil d'exécution par *process*.

autre chargé du traitement. J'ai manqué de temps pour tester cette autre façon de faire et les comparer.

L'API *MSG* de Simgrid fournit les primitives nécessaires à la gestion des tâches, ainsi qu'à leur transmission sur le réseau. Ces fonctions de communications sont fournies sous plusieurs versions dont nous retiendrons les trois principales :

bloquante : pour une réception, par exemple, on attend jusqu'à ce qu'on ait reçu quelque chose.
(par exemple, `MSG_task_send/MSG_task_recv`)

non-bloquante : toujours pour une réception, on se met en écoute et on vient voir de temps en temps si on a reçu quelque chose (on peut faire d'autres choses en attendant).
(par exemple, `MSG_task_isend/MSG_task_irecv`)

détachée : pour une émission, il s'agit d'une émission de type *best-effort* : on envoie et on passe à la suite sans se soucier de savoir si la tâche a été reçue ou pas.
(`MSG_task_dsend`)

Par la suite, on appellera "requête **synchrone**" une requête qui attend une réponse et "requête **asynchrone**" une requête qui n'en attend pas. Par exemple, une requête qui demanderait à un sommet quelle est sa charge est une requête synchrone ; et une requête demandant à un autre sommet d'ajouter un étage à ses tables est une requête asynchrone. Les deux fonctions `send_msg_sync` et `send_msg_async` ont été écrites pour gérer l'envoi de ces deux types de requêtes. `send_msg_sync` doit donc également gérer la réception de la réponse, comme nous le verrons en détails en 3.8.1, page 46.

Ce portage sur Simgrid étant réalisé, nous pouvons maintenant examiner les conséquences de l'introduction de la parallélisation et de la décentralisation sur les algorithmes de construction du DST. Quelques erreurs apparaissent, entraînant les modifications citées plus haut et détaillées dans l'**annexe A**. Mais pour l'essentiel, on a maintenant affaire à un certain nombre de problèmes de synchronisation. La suite de cette étude se propose donc de les étudier en détails pour tenter de leur apporter des solutions.

3.2 Exposé du problème

Comme cela a été dit, les ordres à exécuter peuvent être émis de deux façons : synchrone ou asynchrone (avec ou sans attente de réponse, respectivement). Dans le cas des requêtes asynchrones, l'émetteur ne connaît pas l'état d'avancement de l'ordre qu'il a envoyé et cela peut éventuellement poser problème pour la suite des opérations.

Une première solution qui vient à l'esprit serait de remplacer systématiquement toutes les requêtes asynchrones qui posent problème par de simples requêtes synchrones. Mais après expérimentation, il apparaît que ce n'est pas une bonne solution pour deux raisons : (1) seules les requêtes asynchrones permettent de profiter pleinement de la parallélisation

et on perd alors en performances et (2) des *deadlocks*[4] apparaissent très fréquemment. On en conclut que les requêtes synchrones ne doivent être utilisées que lorsque c'est vraiment nécessaire et que d'autres solutions doivent être trouvées pour synchroniser les tâches asynchrones qui le nécessitent.

Dans le cas de tâches diffusées – les diffusions étant réalisées de façon asynchrone – le problème se complique davantage. Si de la synchronisation est nécessaire, il faut pouvoir s'assurer que tous les nœuds contactés ont bien terminé l'exécution de la tâche diffusée avant de passer à la suite. Cela peut être requis au sein même de la diffusion, lorsqu'on ne veut descendre d'un étage que si l'étage courant a bien été traité. C'est ce que nous ferons à l'aide d'un mécanisme d'attente basé sur des accusés réception, détaillé plus loin. (voir 3.8.2 page 50) De plus, il faut aussi tenir compte du fait que ce n'est pas parce qu'une tâche est terminée que les opérations qu'elle a lancées le sont, toujours à cause de ces envois asynchrones.

Enfin, lors de toutes ces opérations, il faut s'assurer que la taille des groupes impactés reste toujours bien comprise entre les bornes a et b, ce qui constitue une difficulté supplémentaire.

Nous examinons donc le déroulement des opérations lors de l'arrivée d'un nouveau nœud, afin de déterminer les endroits où de la synchronisation est nécessaire. Autrement dit, il s'agit d'examiner chaque endroit où une requête asynchrone est émise pour voir si cela pose problème ou pas.

> *Remarque* : Tous les exemples donnés dans les explications qui suivent sont issus des traces générées lors de mes expérimentations.

3.3 Déroulement des opérations

Pour rappel, voici un exemple de l'arrivée d'un nouveau nœud dans un DST avec ajout d'étage :

La figure 3.1 montre l'arrivée du nœud `52` dans le groupe `AA`.

- Le groupe `AA` a 4 membres et doit se scinder pour donner les groupes `AA0` et `AA1`.
- Le groupe `A` doit alors également se scinder puisqu'il avait aussi déjà 4 membres. Cette scission donne les groupes `A0` et `A1`.
- Le groupe `A` était la racine. Un DST ne pouvant avoir qu'une seule racine, il est donc nécessaire d'ajouter un étage : c'est le groupe `A'`.

⋆⋆⋆

Dans l'exposé qui suit, chacun des endroits pouvant nécessiter de la synchronisation est repéré de cette façon : ①

4. Des boucles sans fin d'attentes mutuelles entre deux ou plusieurs nœuds.

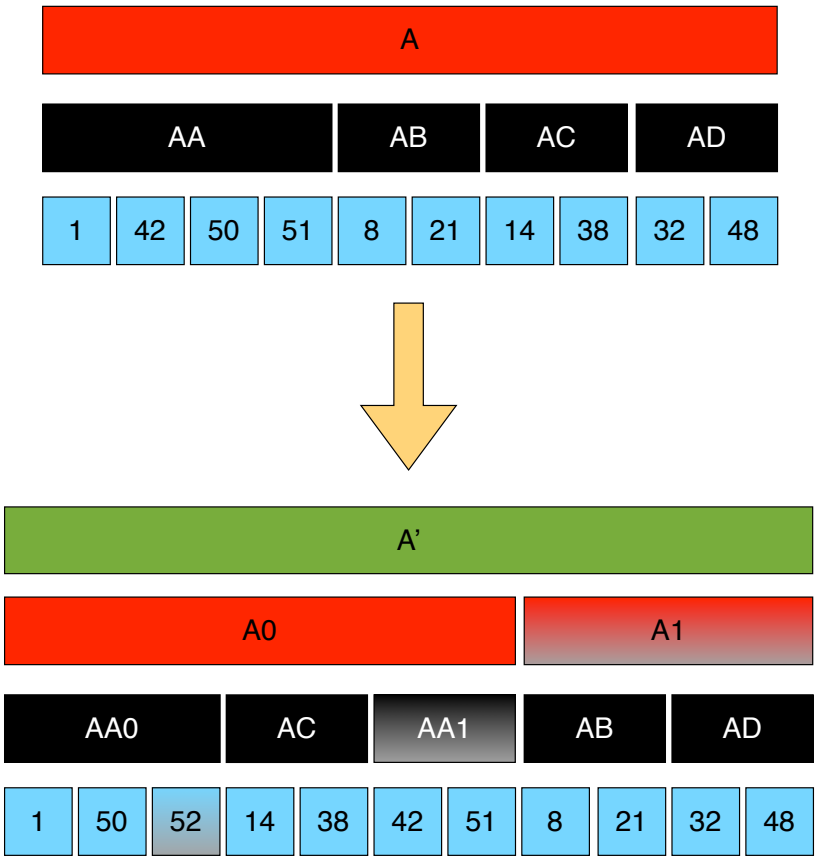

FIGURE 3.1 – Arrivée d'un nœud : scissions avec ajout d'étage

Lors de l'arrivée d'un nœud, on a la séquence d'appels de fonctions suivante [5] :

Algorithme 1 : Algorithme général de l'ajout d'un nœud

1: un nouvel arrivant se présente : `joindre`
2: son contact lui fait de la place : `demander_connexion`
3: des scissions de groupes sont requises : `demander_scission*`
4: ajoute un étage : `ajouter_étage`
5: effectue les scissions : `scission`
6: informe les groupes pères qu'ils ont des nouveaux fils : `connecter_les_groupes_scindés*`

Les * indiquent les boucles : `demander_scission` s'exécute sur chaque étage à scinder et `connecter_les_groupes_scindés` est appelée sur chaque prédécesseur de l'étage supérieur à celui de la scission.

5. On se place bien sûr dans le cas le plus complet, c'est à dire le cas où des scissions qui provoquent un ajout d'étage sont nécessaires.

LES APPELS

`joindre :` L'appel à `demander_connexion` se fait de façon synchrone. Le nœud qui attend est le nouveau nœud qui, à se stade, ne fait pas encore partie du DST et ne peut donc pas être sollicité par d'autres. Il n'y a donc pas de risque de *deadlock*.

`demander_connexion :` Ici, les appels à `demander_scission` se font en local. Pas de souci non plus.

`demander_scission :` On réalise ici essentiellement deux tâches : ajouter un étage si besoin et effectuer les scissions proprement dites. Ces deux tâches sont diffusées.

L'ajout d'étage doit être entièrement réalisé sur l'ensemble des nœuds *avant* de passer aux scissions sans quoi il y aurait un risque de travailler sur un étage qui n'existe pas. ①

`ajouter_étage :` On n'effectue ici que des opérations locales et il n'y a pas de problème : les opérations sont bien terminées lorsque la diffusion l'est.

`scission :` Ici, des envois asynchrones sont utilisés pour mettre à jour des prédécesseurs ② et des tables de routage ③ (avec `connecter_les_groupes_scindés`). Ces prédécesseurs étant différents de ceux qui sont utilisés pour les appels à `connecter_les_groupes_scindés` (ils ne sont pas situés au même étage), il n'y a pas de souci de synchronisation locale ici.

`connecter_les_groupes_scindés :` Des envois asynchrones sont utilisés pour la mise à jour de certains prédécesseurs. ④

LES RETOURS

`connecter_les_groupes_scindés :` Conséquence de ④ : au retour de cette fonction, des prédécesseurs ne sont pas à jour. Pas de problème pour `scission` qui ne les utilise pas.

`scission :` À cause des points ② et ④, des prédécesseurs ne sont pas à jour au retour de cette fonction. Le point ③ fait que des tables de routage ne sont pas à jour non plus, mais seulement aux étages supérieurs à 0 du fait que `connecter_les_groupes_scindés` ne touche pas aux nœuds de l'étage 0.

`demander_scission :` La fonction de diffusion de `scission` est maintenant considérée comme terminée et `demander_scission` rend la main à `demander_connexion`. À ce moment, tous les nœuds qui devaient se scinder l'ont fait – le mécanisme d'attente mis en œuvre dans la diffusion l'assure [6] – mais des prédécesseurs et des tables de routage peuvent encore ne pas être à jour à cause des points ②, ③ et ④ déjà mentionnés.

`demander_connexion :` Au retour ici, on peut éventuellement lancer un nouvel appel à `demander_scission` pour un autre étage et donc, à `scission`. Celle-ci ne fonctionnera pas correctement si les tables de routage et les prédécesseurs concernés ne sont

6. Voir détails page 50

pas à jour. Ici, les points ②, ③ et ④ posent donc problème.

Au retour des appels à `demander_scission`, l'arrivée du nouveau nœud est prise en compte par l'ensemble des sommets de l'étage 0 au moyen d'appels asynchrones à la fonction `nouveau_frère_reçu` ⑤. Pour que cette fonction puisse s'exécuter, il doit y avoir suffisamment de place à l'étage 0. C'est bien le cas puisqu'à ce stade, tous les nœuds concernés se sont bien scindés, comme expliqué plus haut.

Du fait de ⑤, tous les frères ne seront donc pas à jour lors du retour de cette fonction, mais le sommet local transmis à `joindre` l'est forcément puisque sa mise à jour est réalisée en local et pas au moyen d'appels de fonction.

`joindre :` Ici, on doit recevoir de `demander_connexion` la table de routage de son sommet courant. Il est bien sûr important que cette table soit bien à jour puisqu'elle va devenir la table du nouveau nœud. C'est le cas, comme expliqué ci-dessus.

Ensuite, l'équilibrage de charge est réalisé. Pour qu'il puisse fonctionner correctement, il est important que les prédécesseurs soient tous à jour puisque c'est en les comptant qu'on calcule la charge. C'est le cas si les points ② et ④ sont déjà solutionnés.

L'équilibrage de charge s'effectue au moyen d'appels synchrones [7] pour la table de routage et asynchrones pour les prédécesseurs.⑥ Il y a donc un risque que `joindre` rende la main avant que tous les prédécesseurs ne soient à jour.

En conséquence de ⑤ et ⑥, `joindre` peut donc rendre la main alors que les tables de routage et les prédécesseurs ne sont pas encore à jour, ce qui peut poser problème pour d'autres ajouts de nœuds, si ce nœud nouvellement intégré sert tout de suite de contact à un autre nouveau nœud.

Il y a donc deux problèmes à résoudre : [8] les points ②, ③ et ④ d'une part et les points ⑤ et ⑥ d'autre part. On peut les résoudre en mettant en œuvre le mécanisme d'attente déjà mentionné, excepté pour le point ⑥ qui présente une particularité. Pour mémoire, l'algorithme de l'équilibrage de charge est présenté page 31.

On peut y voir que pour un même étage, on peut avoir successivement un envoi asynchrone de `ajouter_prédécesseur` (ligne 10) puis un envoi synchrone de `demander_nouveau_rep` (ligne 8) au tour suivant. Il est donc possible de recevoir l'accusé réception de `ajouter_prédécesseur` pendant l'attente de la réponse de `demander_nouveau_rep`. Si la fonction d'attente est placée entre chaque étage, alors elle ne serait pas encore lancée et cet accusé réception ne sera pas reçu, aboutissant à un blocage. La solution est donc d'attendre entre chaque frère, mais la fonction d'attente n'est alors plus utile ; il suffit d'utiliser un envoi synchrone pour l'ajout de prédécesseur à la ligne 10.

7. Il y a donc un risque de *deadlock* dans la fonction d'équilibrage de charge qui devra être pris en compte. (voir page 46)

8. Le point ① est résolu par la fonction de diffusion synchronisée détaillée plus loin. (voir page 54)

Algorithme 2 : L'équilibrage de charge

```
1: pour étage ← 1, moi.hauteur − 1 faire
2:     nouv_nuds ← []
3:     pour tout f ∈ moi.frères[étage] faire
4:         si (f = contact) alors
5:             nouv_noeuds.ajouter(moi.nom)
6:             ajouter_predecesseur(moi, étage, moi.nom)
7:         sinon
8:             nouveau_rep ← send_msg_sync(moi.nom, f, demander_nouveau_rep,
                (étage))
9:             nouv_noeuds.ajouter(nouveau_rep)
10:            send_msg_async(moi.nom, nouveau_rep, ajouter_prédécesseur,
                (étage, moi.nom))
11:        fin si
12:    fin pour
13:    moi.frères[étage] ← nouv_noeuds
14: fin pour
```

3.4 La gestion des synchronisations

Comme nous venons de le dire, dans une structure totalement décentralisée comme celle recouverte par le DST, on utilise des appels distants pour exécuter les différentes tâches. Ces appels sont de préférence asynchrones, tant pour des raisons de performance (on profite ainsi de la parallélisation possible dans un tel environnement) que de stabilité (il n'y a pas de risque de *deadlocks* avec des communications asynchrones).

Mais il y a essentiellement deux cas de figure pour lesquels de tels échanges asynchrones ne sont pas adaptés : (a) quand une requête attend une réponse (b) quand il y a besoin d'ordonner ou de synchroniser certaines séquences d'opérations entre elles. Dans ces deux cas, les nœuds concernés doivent attendre de recevoir certaines réponses avant de pouvoir continuer leur travail et c'est cette attente qui peut poser problème. Pour le cas (a), on crée la fonction `send_msg_sync` qui est chargée de gérer l'envoi d'une requête et la réception de sa réponse. Pour le cas (b), c'est la fonction `send_msg_async` [9] qui est chargée de l'envoi du message. On lui adjoint la fonction `attente_terminaison` dont le rôle sera de placer des points de synchronisation aux endroits mentionnés précédemment (en 3.3). Dans cette partie, nous allons détailler ce que doivent faire `send_msg_sync` et `attente_terminaison` qui comportent toutes deux des attentes. Nous verrons que chacune de ces deux fonctions doit être capable de recevoir et traiter les réponses attendues par l'autre.

Si on se contente d'une simple attente, on aboutit à des blocages. Prenons un exemple : un nœud `a` qui attend une réponse d'un nœud `b` peut très bien être sollicité par un nœud `c` pendant ce temps. Si `b` attend sur `c` pour donner sa réponse à `a` et que `c` attend sur

9. pour `send_msg_sync` et `send_msg_async`, voir remarque page 26

`a`, il y a blocage. L'expérience montre que plus il y a d'échanges synchrones et plus on rencontre ce genre problème.

Le problème général est alors celui-ci :

1. insérer les points de synchronisation déjà mentionnés (ce qui implique d'introduire de nouvelles attentes)
2. identifier les problèmes que ces attentes peuvent poser
3. trouver des solutions à ces problèmes

> Remarque : Dans ce qui suit, ce que je présente s'appuie sur la notion de *leaders* introduite par Sylvain Dahan dans sa thèse : chaque nœud (chaque groupe de sommets, donc) doit posséder un *leader* qu'il faut consulter pour effectuer certaines opérations sur le groupe. Cette façon de faire va permettre de détecter d'éventuels conflits et d'agir en conséquence pour maintenir la cohérence du groupe.

⋆ ⋆ ⋆

Comme nous l'avons dit, en introduisant des attentes, on augmente le risque de *deadlock*. Pour remédier à cela, il faut donc trouver un moyen de rendre ces attentes non bloquantes. On peut agir à trois niveaux :

1. L'**émission** de la requête synchrone (utilisée dans `send_msg_sync`) utilise une fonction d'envoi asynchrone. De la sorte, on n'attend pas que la requête soit reçue pour se mettre à attendre une réponse. [10]
2. Pour l'**attente** de la réponse, j'ai expérimenté deux solutions : synchrone simple ou synchrone avec *timeout*. Cette dernière se déroule comme ceci : on écoute jusqu'à ce qu'on reçoive une réponse *ou* pendant un certain temps (*timeout*). À l'issue de chaque *timeout*, on endort le process en cours un peu de temps pour passer la main à d'autres process avant de se remettre à écouter .

   Cette méthode présente l'avantage de ne pas bloquer tout le système si un nœud ne répond pas, ce qui sera intéressant pour la tolérance aux pannes. Mais j'ai rencontré des cas de messages perdus (messages jamais reçus) avec cette méthode, et il semble que le temps de "sommeil" ait une importance, ce que je ne comprends pas. L'équipe de Simgrid contactée à ce sujet ne confirme pas ces observations et il faudra donc revoir cette méthode pour corriger le problème.

   La méthode synchrone simple fonctionne bien et elle n'entraîne pas de blocage puisque toutes les tâches reçues dans l'intervalle sont traitées. J'ai donc choisi cette solution, bien qu'elle ne prenne pas en compte la tolérance aux pannes. [11] On se concentre ici sur les problèmes de synchronisation.

10. Dans ce contexte, il est important de ne choisir une communication synchrone que lorsque c'est indispensable pour ne pas risquer d'ajouter de nouveaux problèmes.

11. En effet, telle que l'attente est actuellement réalisée, il est possible ici d'attendre indéfiniment la réponse d'un nœud qui serait en panne.

3. Le **traitement** du message reçu pendant l'attente, qu'il s'agisse d'une requête, d'*une* réponse ou de *la* réponse attendue, doit lui aussi être conçu de manière à éviter les blocages.
   Voyons ce point dans le détail.

3.5 Gestion des messages reçus pendant l'attente

3.5.1 Le message reçu pendant l'attente est une réponse

Pour qu'un message reçu soit identifié comme *la* réponse attendue, 3 conditions doivent être remplies :

1. ce message doit *être* une réponse. (il s'agit d'éviter que des requêtes puissent être prises pour des réponses)
2. ce message doit provenir du destinataire de la requête à laquelle il répond.
3. le type de ce message doit être le même que celui de la requête à laquelle il répond. [12]

Si cette réponse n'est pas *la* réponse attendue, mais *une* réponse attendue, il va falloir la prendre en compte. (voir détails plus loin)

Si le message est une réponse non attendue, on peut simplement l'ignorer. En effet, dès lors qu'une tâche peut être appelée de façon synchrone, elle est conçue pour retourner une réponse. Mais il est possible qu'on appelle aussi cette même tâche de façon asynchrone, auquel cas la réponse qu'elle retourne doit simplement être ignorée.

Pour distinguer les messages-requêtes des messages-réponses, on utilise une possibilité offerte par Simgrid : le nommage des tâches. Les réponses seront ainsi nommées `'ans'`.

3.5.2 Le message reçu pendant l'attente est une requête

Trois cas de figure peuvent se présenter :

requête asynchrone : S'il s'agit d'une simple requête asynchrone, on exécute la tâche requise et on se remet en écoute. Mais si cette tâche utilise elle-même la fonction `attente_terminaison`, d'autres messages peuvent être reçus par cette nouvelle attente. Il va falloir en tenir compte.

requête synchrone : La tâche qu'on va exécuter va alors utiliser `send_msg_sync` et se mettre à son tour en écoute, rendant ainsi possible des attentes en cascade. Comme précédemment, il faut donc trouver un moyen de faire en sorte que la dernière écoute accepte les réponses attendues par les "écoutes parentes", ainsi que tous les autres messages.

12. S'il s'agit d'une diffusion, il faut aussi que les types diffusés correspondent.

requête non autorisée : Au moment où cette requête est reçue, on ne veut pas l'exécuter. Par exemple, lors de l'équilibrage de charge effectué à la fin de la fonction `joindre`, il faut refuser d'exécuter une scission requise par un nouvel arrivant. En effet, à ce stade, le nœud courant n'est pas encore utilisé comme représentant de son groupe à chaque étage et de plus, la scission changerait la table de routage qu'on est en train de parcourir. Il y a ici deux problèmes à résoudre : trouver un moyen de différer l'exécution de la requête et décider quand l'exécuter.

Comme pour les réponses, l'identification des requêtes se fait en les nommant : `'async'` et `'sync'`. Reste à identifier les requêtes non autorisées.

3.6 Reconnaissance et traitement des requêtes non autorisées

La reconnaissance de ces requêtes est plus complexe. Un même type de requête sera accepté ou refusé selon le contexte et ce n'est pas parce qu'une requête est synchrone ou asynchrone qu'on va toujours la refuser ou toujours l'accepter. Par exemple, la scission déjà mentionnée – qu'on souhaite donc refuser – est diffusée et les diffusions sont asynchrones. Les ajouts/suppressions de prédécesseurs aussi, et on peut les accepter la plupart du temps. De plus, il faut aussi tenir compte du fait qu'une requête de diffusion peut être lancée de façon synchrone ou asynchrone [13]. Pour établir des règles permettant de déterminer si une requête doit être refusée ou pas, il faut donc examiner chaque cas. (voir tableau 3.2)

Puisque la décision d'accepter ou de refuser une requête dépend – entre autres – de l'état actuel du nœud courant, il faut le définir. Il va s'agir d'une structure comportant deux champs : un *code état* et l'*id* du nouveau nœud arrivant qui a provoqué le dernier changement d'état.

3.6.1 Les codes état

Voici les codes état utilisés :

- Les états au cours de l'arrivée d'un nouveau nœud :
  1. `'b'` → *en construction* : attribué à l'arrivée d'un nouveau nœud
  2. `'n'` → *non actif* : si le nœud a échoué à rejoindre le DST
  3. `'l'` → *équilibrage de charge* en cours : une fois le DST rejoint, le nouveau nœud procède à un équilibrage de charge
  4. `'a'` → *actif* : le nouveau nœud est intégré

13. La propagation d'une diffusion est toujours faite de façon asynchrone, mais le lancement peut être synchrone si on souhaite attendre que la diffusion soit terminée avant de poursuivre.

– Les états au cours de la vie d'un nœud actif :
  – `'o'` → *ok* pour mise à jour (optionnel – voir explications)
  – `'u'` → *mise à jour* en cours : permet d'empêcher les modifications concurrentes des tables du nœud
  – `'g'` → `demander_nouveau_rep` en cours : on demande au nœud quel est le frère de niveau 0 le moins chargé, en vue de l'utiliser comme représentant du nœud.

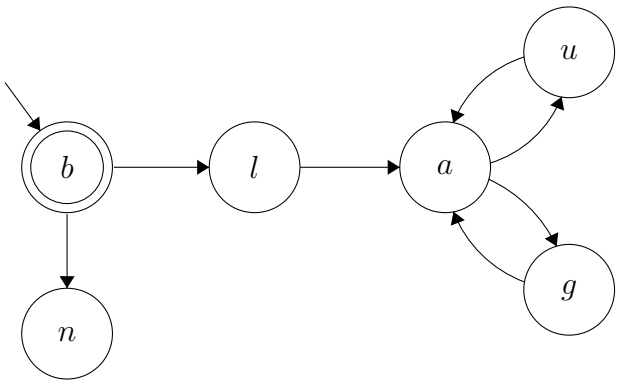

FIGURE 3.2 – Les différents états des sommets du DST

Voici comment se passent les choses en détail :

- Un nouvel arrivant reçoit l'état `'b'`. S'il échoue à rejoindre le DST, son état devient `'n'`.
- Il passe de `'b'` à `'l'` au début de l'équilibrage de charge.

  Voici un exemple tiré des traces générées par le programme sous Simgrid, montrant la nécessité de cet état `'l'` :
  – 109 envoie `ajouter_prédécesseur` à 142 et attend sa réponse (un accusé réception)
  – 142 est en train de faire l'équilibrage de charge : il est donc en `'b'` et diffère alors l'exécution de `ajouter_prédécesseur`
  – 142 continue son équilibrage de charge et envoie des `demander_nouveau_rep` à toute sa table de routage, dont 42
  – 42 envoie des `demander_nb_prédécesseurs` à tous ses frères, dont 569
  – 569 étant aussi en équilibrage de charge à ce moment – en `'b'`, donc – diffère l'exécution de `demander_nb_prédécesseurs`
  – 569 ne passe pas en `'a'` parce qu'il attend lui-même sur 109 (indirectement)

  Donc 569 ne répond pas à 42, qui ne répond pas à 142 qui ne répond pas à 109 : on est bloqué.

  Pour corriger ce type de problème, on peut intervenir à deux niveaux :

1. Faire en sorte que 142 ne diffère pas `ajouter_prédécesseur`, c'est à dire qu'il ne soit pas à l'état `'b'`. Une solution pourrait être de passer ce nœud à `'a'` dès le début de l'équilibrage de charge, mais il y aurait un risque d'accepter des requêtes telles qu'une scission. Passer à l'état `'u'` ne conviendrait pas non plus puisqu'il doit laisser passer `connecter_les_groupes_scindés` (voir détails plus loin). Il faut donc bien créer un nouvel état : `'l'`.
2. Accepter un message `demander_nb_prédécesseurs` sur un sommet en `'b'`. La valeur retournée – qui est la charge du sommet – peut être fausse puisque sur un sommet en construction, la table des prédécesseurs peut ne pas être à jour ; mais si c'est le cas, elle ne peut être que trop basse. Cela aurait pour conséquence de favoriser le choix de ce sommet comme représentant, en le considérant comme moins chargé qu'il ne l'est réellement. Ce nouveau sommet n'est pas encore très utilisé par les autres et il y a de fortes chances pour qu'il soit de toutes façons le moins chargé. Admettre cette erreur paraît donc une meilleure solution que de risquer des *deadlocks*.

J'ai donc choisi de faire les deux : la solution 1 – passer à l'état `'l'` – permet de répondre plus tôt et la 2 intervient même si le premier report de tâche a lieu alors qu'on n'est pas dans l'équilibrage de charge.

- Cet état passe à `'a'` lorsque le nœud est effectivement intégré au DST et opérationnel, c'est à dire après cet équilibrage de charge. [14]
- Un nœud passe à l'état `'g'` lorsqu'il exécute la fonction `demander_nouveau_rep`. Dans celle-ci, il interroge chacun des frères de l'étage 0 pour connaître leur nombre de prédécesseurs et il est important que cette phase ne soit pas interrompue par l'arrivée d'un nouveau nœud. En effet, la fonction `demander_connexion` peut engendrer des scissions et donc modifier la liste des frères en train d'être parcourue, ce qui aboutirait à des erreurs. Prenons un exemple :
  - Le nœud 1570 demande à 2739 d'exécuter `demander_nouveau_rep` pour obtenir un nouveau représentant de l'étage courant.
  - 2739 envoie des `demander_nb_prédécesseurs` à ses frères de l'étage 0 : 1445, 4093 et 1847 (dans cet ordre)
  - Alors qu'il attend la réponse de 1847, 2739 reçoit `demander_connexion` de 1847 (c'est une coïncidence, il se trouve que 1847 est leader pour le contact d'un nouveau nœud).
  - 2739 lance donc les opérations pour faire de la place au nouvel arrivant.
  - Ceci fait, il revient regarder s'il a reçu la réponse de 1847 (le nombre de ses prédécesseurs) : oui, c'est bien le cas et il se trouve que c'est précisément 1847, le frère n°3, qui est le moins chargé.

14. Un nœud peut donc être actif alors que certains des nœuds pointés dans sa table de routage ne sont pas encore forcément prêts. Les nœuds n'arrivent pas à l'état actif dans le même ordre que leur ordre d'arrivée mais d'après mes expérimentations, cela ne semble pas être un problème du fait que d'éventuelles requêtes seraient alors mises en attente jusqu'à l'état `'a'` effectif.

- 2739 répond donc à 1570 que le frère n°3 peut le remplacer. Ce frère n° 3 devrait être 1847 mais puisque des scissions ont eu lieu, il est maintenant vide. Au lieu de répondre '1847', il répond donc '0' (les cases vides de la table de routage sont à 0 par défaut)
- 1570 remplace donc 2739 par 0 et lui envoie `ajouter_prédécesseurs`. Si le nœud 0 existe, le remplaçant de 2739 n'est simplement pas le bon, mais s'il n'existe pas, alors cet envoi échoue et provoque une erreur.

Pour corriger ce problème, il faut faire en sorte de ne pas accepter de `demander_connexion` pendant l'exécution de `demander_nouveau_rep`. L'idée est donc de changer l'état du nœud courant au début et de restaurer son état initial à la fin.

Aucun des états existants `'b'`, `'l'` ou `'u'` ne convient, on obtient toujours des *deadlocks* dûs au fait que ces états reportent trop de types de tâches à ce stade. En effet, un nœud qui peut exécuter `demander_nouveau_rep` est généralement un nœud actif qui ne bloque rien. On va donc créer un nouvel état pour ce cas qu'on va nommer `'g'` (pour `get_rep`). Au début de la fonction, on mémorise l'état courant, puis on le passe à `'g'`. A la fin, si l'état n'est pas passé à `'a'` entre temps – parce que le nœud aurait reçu une diffusion d'un tel changement d'état – on restaure l'état initial. Il suffit alors de faire en sorte qu'un nœud à l'état `'g'` diffère `demander_connexion`.

> **Remarque :** Par manque de temps, je ne suis pas allé plus loin dans la résolution de ce problème, mais je pense que cette solution est incomplète. En effet, comme je l'ai mentionné, il est possible que le nœud à `'g'` passe à `'a'` avant la fin de la fonction. Dès qu'il est à `'a'`, il peut accepter une requête `demander_connexion` et le problème se pose à nouveau. Il faudrait donc mettre en place un mécanisme qui ne passe le nœud à `'a'` qu'à la fin de la fonction, s'il a reçu un tel ordre entre temps, bien sûr. Sinon, on restaure l'état initial de début de fonction.

- L'état `'u'` indique que le nœud est en cours de mise à jour, c'est à dire que ses tables de routage ou de prédécesseurs sont susceptibles d'être modifiées. Les différentes expérimentations que j'ai menées ont montré que la gestion correcte de cet état par la fonction `demander_connexion` constitue le principal problème de synchronisation à résoudre. Il s'agit ici de protéger des sections critiques tout en s'assurant de ne pas provoquer de *deadlocks*. Voyons donc cela dans le détail.

3.6.2 La gestion de l'état 'u'

Cet état a pour but de verrouiller un nœud en cours de mise à jour, de sorte qu'aucun autre événement ne puisse provoquer une mise à jour concurrente de ses tables pendant ce temps : le modifier ferait échouer la mise à jour et l'interroger risque de donner des réponses inconsistantes puisque ses informations ne seront fiables qu'une fois la mise à jour terminée.

S'il y a suffisamment de place pour accueillir un nouveau nœud dans le DST, seul le nœud exécutant `demander_connexion` – c'est à dire le contact (ou son leader) du nouvel arrivant – va être modifié. Il suffit donc de le passer à `'u'` au début de la fonction et de le repasser à `'a'` à la fin [15].

S'il faut faire de la place, `demander_connexion` et `demander_scission` se chargent de ce travail et c'est tout un ensemble de nœuds qui vont devoir être scindés. L'idée est alors de rendre cette phase de scissions atomique : sur le même modèle qu'une transaction dans une base de données, soit on exécute la totalité des scissions, soit on n'en fait aucune, mais il faut absolument éviter de n'en réaliser qu'une partie. Le DST serait alors incohérent et ne fonctionnerait plus.

On passe donc tous ces nœuds à l'état `'u'` au moyen d'une diffusion synchrone : on attend que tout le monde ait retourné son accusé réception pour passer aux scissions. Cette diffusion peut échouer, comme on le verra plus loin (on peut rencontrer une autre phase de scissions, déclenchée par l'arrivée d'un autre nœud, par exemple). Comment faire alors pour s'arrêter là et remettre cette phase à plus tard ?

Si on se contente d'arrêter la diffusion en cours, des nœuds déjà passés à l'état `'u'` vont y rester et demeureront ainsi verrouillés. Puisqu'il est également possible que plusieurs arrivées de nœuds soient dans ce cas, on peut avoir beaucoup de nœuds à `'u'` pendant une durée non négligeable (le temps de recommencer les intégrations des nouveaux nœuds qui ont échoué), ce qui constitue un risque élevé de *deadlocks*. Les expérimentations que j'ai réalisées sur ce problème aboutissent presque toujours à des *deadlocks*, confirmant l'intuition.

Une autre idée pourrait être de relancer une diffusion qui remettrait les nœuds qu'on vient de passer à `'u'` à leur état d'origine, c'est à dire `'a'` [16]. Dans ce cas, il faut pouvoir distinguer deux sortes de nœuds à `'u'` : ceux qu'on veut effectivement remettre à `'a'` et ceux qu'il ne faut pas toucher parce qu'ils ont été mis à `'u'` suite à l'arrivée d'un autre nœud [17]. C'est la raison de la présence du champ *id* dans la structure représentant l'état courant d'un nœud, comme on l'a indiqué plus haut. Ainsi, un nouveau nœud donné ne repassera à `'a'` que les nœuds qu'il a lui-même passé à `'u'` et ne touchera pas aux autres. De plus, il faut faire en sorte que cette diffusion supplémentaire n'augmente pas les risques de blocage.

De prime abord, cette solution paraît gourmande en messages. De plus, si la diffusion des `'u'` a été rapidement arrêtée et qu'on diffuse un "*rollback*" sur l'ensemble des nœuds potentiellement impactés – le mécanisme à mettre en œuvre pour limiter cette diffusion aux seuls nœuds passés à `'u'` paraît complexe –, il semble qu'on gaspille beaucoup de messages. J'ai donc cherché un moyen de limiter ce nombre de messages.

15. Il est forcément à `'a'` au début puisque s'il avait été dans un autre état, on n'aurait pas accepté de lancer cette fonction.

16. J'appelle cette diffusion un *rollback*, toujours par analogie avec ce qui se pratique sur les bases de données.

17. Il peut s'agir de ceux-là même qui ont fait échouer la diffusion qu'on est en train d'annuler.

Pour essai, j'ai donc mis en œuvre une autre façon de faire qui consiste à procéder en deux temps : on commence par diffuser un état supplémentaire 'o' vers tous les nœuds à scinder. Si cette diffusion se passe bien, on diffuse l'état 'u'. Un nœud passé à 'o' suite à l'arrivée d'un nouveau nœud x n'acceptera de passer à 'u' que si l'ordre de le faire est déclenché par l'arrivée du même nœud x.

L'idée est de faire en sorte que l'état 'o' bloque moins de choses que l'état 'u' pour pouvoir se permettre, en cas d'échec de la diffusion des 'o', de ne pas revenir en arrière. On se contenterait alors d'arrêter la diffusion pour la reprendre un peu plus tard. (Ce temps de reprise étant aléatoire) Le nombre d'échecs est surveillé : au-delà d'un certain seuil, on se résout finalement à repasser les nœuds en question à 'a' comme on l'aurait fait avec la solution précédente.

Pour comparer les deux approches, il faut examiner la fréquence à laquelle ce seuil est atteint. S'il l'est trop souvent, alors cette solution est finalement plus coûteuse que la précédente. En effet, plaçons-nous dans le cas où deux tentatives ratées sont suivies d'une réussie. La première solution réalise alors 6 diffusions : 'u', 'a' deux fois puis à nouveau 'u' et 'a' (pour se remettre comme on était avant le premier 'u'). Avant le seuil, la deuxième approche n'en réalise que 5 : 'o' deux fois puis 'o', 'u' et 'a'. Mais si on est au-delà du seuil, on en fait 7 : 'o' 'a' deux fois puis 'o', 'u' et 'a'. Le résultat d'une exécution de chacune de ces deux solutions sur un DST de 4 000 nœuds est exposé dans le tableau 3.1.

	Nombre total de messages	Durée d'exécution
Avec 'o'	1 170 762	2435.821
Sans 'o'	1 076 592	2761.161

TABLE 3.1 – Exécutions comparées des deux solutions

Comme on le voit, la solution avec 'o' (le seuil a été fixé à 2 tentatives) ne donne pas le résultat escompté. La durée d'exécution est un peu meilleure mais le nombre de messages est finalement supérieur. J'ai donc opté pour la solution sans 'o' parce que le critère de bande passante me paraît plus important.

> J'ai également tenté une autre approche qui consiste à ne diffuser l'état 'u' qu'aux leaders de chaque étage concerné, mais cela ne fonctionne pas correctement : le contact d'un nouvel arrivant peut obtenir l'autorisation de procéder aux scissions alors qu'il ne le faudrait pas. Ces cas de figure étant complexes à analyser – les cas d'erreur ne commencent à apparaître qu'à partir de 2 500 - 3 000 nœuds – j'ai manqué de temps pour le faire et je n'ai pas d'explication claire sur ce phénomène. Pour cette raison, j'ai choisi de changer l'état de tous les nœuds concernés et pas seulement celui des leaders.

3.6.3 Les règles de changement d'état

Il faut considérer deux problèmes : comment gérer la diffusion des changements d'état d'une part, et comment gérer le changement d'état d'un nœud d'autre part. Intéressons-nous tout d'abord aux problèmes de diffusion.

À l'état `'b'` : un nœud ne peut transmettre aucune diffusion. Sa table de routage n'est pas encore utilisable.

À l'état `'u'` : il faut laisser passer la diffusion des opérations de construction – c'est pour réaliser cette construction qu'on a mis le nœud à `'u'` – c'est à dire `ajouter_étage` et `scission`. Il faut également laisser passer les tâches de changement d'état déclenchées par les bons nouveaux nœuds : vers `'u'` pour permettre à un nœud de recevoir plusieurs fois le même ordre de changement d'état (ce qui peut arriver lors d'une diffusion) et vers `'a'` lorsqu'on fait marche arrière lors d'un échec, ou à la fin des opérations d'intégration.

Si les diffusions refusées sont des changements d'état vers `'a'` ou `'u'`, les stocker – pour différer leur exécution – aboutit à des *deadlocks* lorsque deux nouveaux nœuds utilisent des contacts proches. À la place, on peut répondre à l'appelant : pour un changement d'état vers `'u'`, on répond `'NOK'` pour indiquer l'échec de la diffusion – nécessaire pour pouvoir la relancer plus tard – et pour un changement vers `'a'`, on répond `'OK'`. En effet, on refuse un passage à `'a'` si l'ordre ne provient pas du bon nœud et ce bon ordre finira par arriver de toutes façons. Il n'est donc pas utile d'informer l'appelant de cet échec.

Les autres états laissent passer les diffusions normalement.

Ensuite, il faut définir comment se passe le changement d'état d'un nœud.

Passage à `'u'` : Un nœud accepte de passer à `'u'` s'il est à `'a'` ou déjà à `'u'` suite à l'arrivée du même nouveau nœud. Si le nœud est à `'l'` – en équilibrage de charge – on ne change pas l'état, mais on répond tout de même `'OK'` puisque dans ce cas, l'ordre de changement d'état est stocké et interviendra plus tard. Dans tous les autres cas, on répond `'NOK'` pour indiquer l'échec de l'opération.

Passage à `'a'` : Un nœud accepte de passer à `'a'` s'il est à `'u'` suite à l'arrivée du même nouveau nœud ou s'il n'est pas à `'u'`. S'il est déjà à `'a'` suite à l'arrivée d'un autre nouveau nœud, il peut passer à `'a'` avec le nouveau nœud courant, ça ne pose pas de problème.

3.6.4 Les règles d'autorisation des requêtes

L'ensemble des requêtes mises en œuvre pour la construction d'un DST figure dans le tableau 3.2. Les colonnes *Modif. routage* et *Modif. préd.* indiquent si les tables de routage ou des prédécesseurs du nœud courant peuvent être modifiées par les fonctions.[18] Les

18. Il n'y a pas d'indication pour la diffusion puisque cela dépend de la fonction diffusée.

colonnes ... *diffère* indiquent si un nœud à l'état indiqué diffère ou pas l'exécution de la fonction reçue. L'état `'o'` n'y figure pas puisque j'ai finalement choisi de ne pas l'utiliser. Pour ne pas alourdir le tableau, l'état `'g'` n'y figure pas non plus puisqu'on a dit qu'il ne différait que `demander_connexion`.

	SYNC	ASYNC	Modif. routage	Modif. préd.	`'u'` diffère	`'b'` diffère	`'l'` diffère
`demander_nouveau_rep`	X		NON	NON		X	
`demander_connexion`	X		OUI	OUI	X	X	X
`nouveau_frère_reçu`		X	OUI	OUI			
`demander_scission`		X	OUI	OUI		X	
`ajouter_étage`		X	OUI	OUI			?
`connecter_grp_scindés`		X	OUI	NON		X	X
`scission`		X	OUI	NON		X	X
`demander_nb_prédéc.`	X		NON	NON	?	?	
`ajouter_prédécesseur`		X	NON	OUI		X	
`effacer_prédécesseur`		X	NON	OUI			
`diffusion`	X	X	-	-		X	

TABLE 3.2 – Les requêtes de construction

L'état `'u'` diffère :

- `demander_connexion` : un nœud en cours de mise à jour ne peut pas fournir sa table de routage à un nouvel arrivant dont il serait le contact, avant qu'elle ne soit stabilisée. Mais il faut tout de même laisser passer cette requête si elle est lancée par le même nouveau nœud que celui qui a mis le nœud courant à `'u'`. Ce cas peut se produire si une première tentative a échoué et qu'on n'a pas re-diffusé des `'a'` après l'échec.

  La question qui se pose alors est de savoir qui, du contact ou de son leader, va être chargé de mettre cette requête en attente. Si on choisit la deuxième solution (c'est le leader qui s'en charge), il faut être sûr que ce leader choisi par un nœud contact non leader à `'u'` soit le bon. Pour cette fonction `demander_connexion`, le leader est le plus ancien nœud du niveau 0. Celui-ci ne risque pas d'être changé parce que d'une part, si une scission devait avoir lieu à cet étage, les nœuds aux places paires

– dont le leader, donc – sont les nœuds qui restent et d'autre part, `connecter_les_groupes_scindés` ne touche pas au niveau 0. Il ne semble donc pas y avoir de souci.

Mais à l'expérience, on se rend compte que cette solution amène à des *deadlocks*. Voici un exemple issu des traces d'exécution du programme sous Simgrid :

1. 958 diffuse une scission vers 514 : il attend sa réponse (`attente_terminaison`)
2. 514 exécute cette scission et envoie `ajouter_prédécesseur` à 563 : il attend sa réponse (`attente_terminaison`)
3. pendant cette attente, 514 reçoit `demander_connexion` de 314. Il transfert au leader 958 et attend sa réponse (`send_msg_sync`)
4. 958 reçoit `demander_connexion` de 514 mais il la met de côté puisqu'il est à `'u'`. 958 continue ensuite l'attente commencée en 1. (`attente_terminaison`)
5. 563 exécute `ajouter_prédécesseur` reçu de 514 et lui répond.
6. 514 reçoit cette réponse de 563 mais il la met de côté parce qu'il s'agit d'une réponse asynchrone (un accusé réception) reçue pendant une attente synchrone commencée en 3 (`send_msg_sync`).

À ce stade, on est effectivement bloqué : 514 ne peut répondre à 958 que s'il a la réponse de 563 (point 2). Or il l'a mise de côté (point 6) parce qu'il attend une réponse de 958 (point 3).

On voit sur cet exemple qu'au point 3, si 514 mettait la requête `demander_connexion` de côté au lieu de la transférer au leader 958, il resterait dans l'attente commencée au point 2. Il ne mettrait alors pas la réponse de 563 de côté au point 6 et il n'y aurait plus de blocage.

C'est donc le **contact** qui est chargé de différer une requête `demander_connexion` quand il est à l'état `'u'`.

- `demander_nb_prédécesseurs` : cette fonction pourrait retourner une mauvaise valeur si la table des prédécesseurs n'est pas à jour. Mais cela ne porte pas atteinte à l'intégrité du DST – cette fonction intervient dans l'équilibrage de charge – et il semble préférable de la laisser passer – toujours dans l'idée d'avoir le moins de risques de blocages possibles à un moment donné.

Les autres fonctions doivent être exécutées tout de suite puisqu'il s'agit précisément des fonctions qui mettent le nœud à jour.

L'état `'b'` diffère :

- `demander_nouveau_rep` : la charge d'un nœud en construction n'est pas fiable.
- `demander_connexion` : on ne peut pas fournir une table de routage en construction à un nouvel arrivant. Cette fonction doit être différée, que le nœud courant soit le leader ou pas puisque le choix du leader n'est pas fiable sur un nœud en construction.
- `demander_scission` : on ne peut pas exécuter cette fonction avant que la table de routage ne soit stable, sans quoi on ne sait pas combien d'étages doivent être scindés, par exemple.

- `connecter_les_groupes_scindés` : il est évident qu'on ne peut exécuter cette fonction que si on est stable.
- `scission` : on ne lance pas de scissions en même temps qu'on construit.
- `demander_nb_prédécesseurs` : la table des prédécesseurs n'est pas forcément à jour à ce stade. Mais puisqu'on est en construction, si le nombre de prédécesseurs est incorrect, il ne peut qu'être inférieur à ce qu'il devrait être. Comme déjà mentionné, cela aurait pour conséquence de favoriser le choix de ce nouvel arrivant comme nouveau représentant, ce qui ne paraît pas aberrant puisqu'un nouvel arrivant est à priori encore peu chargé. J'ai donc choisi de ne pas bloquer cette requête dans ce cas.
- `ajout_prédécesseur` : il faut attendre la fin de la construction pour ne pas la perturber en y ajoutant un prédécesseur.
- `diffusion` : comme déjà mentionné, une diffusion ne serait pas transmise correctement avec une table de routage en construction.

Là encore, les autres fonctions ne doivent pas être bloquées puisque ce sont elles qui permettent de construire un nœud.

L'état `'l'` diffère :

- `demander_connexion` : Tant que l'équilibrage de charge n'est pas terminé, la table de routage n'est pas en état d'être fournie à un nouvel arrivant.
- `ajouter_étage` : Si on ajoute un étage à la table qui est en train d'être parcourue, on risque de perturber les opérations d'équilibrage de charge et il paraît nécessaire de différer cette requête. Toutefois, on se rend compte qu'il est possible de la laisser passer en prenant quelques précautions, ce qui permettrait d'éviter d'ajouter de nouveaux risques de *deadlocks*,

  Lorsqu'un étage est ajouté, le nœud courant est inscrit dans les tables de routage et de prédécesseurs. Pendant l'équilibrage de charge, on envoie `demander_nouveau_rep` à chacun des nœuds de la table de routage. Du coup, en arrivant sur cet étage supplémentaire, le nœud courant risque de s'envoyer un message à lui-même, perturbant du même coup la réception de son accusé réception. Il faut donc prévoir ce cas pour ne pas envoyer de message et conserver le nœud courant sans le remplacer.

  En procédant de la sorte, il n'est plus nécessaire de différer `ajouter_étage` à ce stade.
- `connecter_les_groupes_scindés` : C'est lors de l'équilibrage de charge que le nœud courant est inscrit à chaque étage comme représentant du groupe auquel il appartient. Si ce n'est pas encore fait, `connecter_les_groupes_scindés` va échouer.
- `scission` : Il ne faut pas modifier la table de routage qui est en train d'être parcourue en totalité par l'équilibrage de charge.

3.7 Solutions proposées

Nous venons de voir les problèmes posés et les moyens qu'il est possible d'utiliser pour les solutionner. Dans la suite, nous allons montrer comment mettre en œuvre concrètement ces concepts, afin d'arriver à des solutions pouvant être testées sur Simgrid.

3.7.1 Requêtes croisées et en cascade

Les requêtes croisées se produisent lorsque `attente_terminaison` est lancée par une requête reçue dans `send_msg_sync` et réciproquement.
Les requêtes en cascade se produisent lorsque `send_msg_sync` est appelée depuis `send_msg_sync` (de même pour `attente_terminaison`).

Aux données de chaque sommet, on ajoute deux structures : une pile pour stocker les requêtes synchrones et un tableau pour les requêtes asynchrones. Les requêtes y sont ajoutées lors de leur envoi. Lorsqu'une réponse est reçue, on y cherche la requête correspondante : s'il s'agit de la réponse attendue par la requête courante, celle-ci est simplement supprimée de la structure. S'il s'agit d'une autre réponse attendue, on l'inscrit dans la structure aux côtés de sa requête et on continue d'attendre la réponse à la requête courante. Dès qu'elle est reçue, il suffira d'examiner l'enregistrement suivant de la structure pour savoir si les réponses attendues ont été reçues ou pas et ainsi décider de la suite à donner. (voir algorithmes en 3.8 pour plus de détails)

Il faut noter ici que l'ordre des enregistrements dans ces structures est important : les requêtes y sont enregistrées dans l'ordre de leur appel et le sommet de la pile (ou le dernier enregistrement du tableau) contient donc la requête courante.

3.7.2 Requêtes non autorisées

J'ai expérimenté deux solutions pour différer l'exécution d'une requête : renvoyer un message à l'émetteur pour lui demander de réitérer sa demande un peu plus tard, ou stocker ces requêtes chez le destinataire en vue de les exécuter au moment opportun.

Avec la première solution, deux problèmes apparaissent. On rencontre des cas de boucles causées par des nœuds qui se répondent tous entre eux qu'ils sont occupés. Ce problème se produit même si le temps d'attente avant de réitérer la requête est aléatoire. Le deuxième problème est que le nœud émetteur d'une requête ne peut tenir compte d'un message d'occupation que si la requête est synchrone. Si elle était asynchrone, il n'attendrait pas de réponse et un message d'occupation serait alors ignoré.

Avec la deuxième solution, on peut aussi obtenir quelques *deadlocks* lors des diffusions de certains changements d'état.

J'ai donc choisi d'utiliser la deuxième solution – le stockage des requêtes – la majeure partie du temps et de réserver la première solution aux seules diffusions de changements

d'état qui peuvent échouer. Lorsqu'il faut faire de la place pour un nouvel arrivant, voici comment cette première solution est mise en œuvre :

Algorithme 3 : Faire de la place pour un nouveau nœud (vue globale)

```
1:  un nouveau nœud N veut rejoindre le DST via son contact C

2:  u_rep ← OK                          ▷ u_rep est la réponse retournée par la diffusion de 'u'
3:  cpt ← 0                             ▷ cpt est le compteur de diffusions

4:  faire
5:      N demande à C de lui faire de la place
6:      C diffuse 'u' à l'ensemble des nœuds à modifier
7:      si (u_rep = NOK) alors
8:          si (cpt ≥ SEUIL) alors
                                        ▷ En dessous de SEUIL tentatives, on ne repasse pas à 'a'
9:              C diffuse 'a' vers les nœuds qu'il a passés à 'u' sur ordre de N
10:         fin si
11:         C informe N de l'échec de la diffusion
12:         cpt + +
13:         N attend un certain temps aléatoire borné
14:     sinon
15:         break
16:     fin si
17: tant que (u_rep = NOK  &&  cpt < MAX)

18: si (u_rep ≠ NOK) alors
19:     suite des opérations ...
20: sinon
21:     N renonce à joindre le DST
22: fin si
```

La diffusion de `'a'` (ligne 9) se fait avec le mécanisme des accusés réception pour être certain que chaque nœud a bien reçu l'ordre de repasser à `'a'`, mais toujours dans l'idée de diminuer le risque de blocage, on ne se soucie pas de savoir si l'ordre a pu ou pas être exécuté. En fait, il n'y a qu'un cas où ce passage à `'a'` pourrait être refusé : si le destinataire a été mis à `'u'` suite à l'arrivée d'un autre nœud que N. Faire remonter cette information à N n'a pas d'intérêt ici puisqu'il va de toutes façons refaire une tentative de diffusion de `'u'` un peu plus tard.

Dans cet algorithme, il n'y a plus de risque de `deadlock` puisque le seul nœud qui attend est le nouvel arrivant N qui, n'étant pas encore intégré au DST, ne participe pas encore aux échanges. Il ne peut donc bloquer personne. À titre de précaution, j'ai tout de même choisi de surveiller le nombre de tentatives de rejoindre le DST : au-delà de `MAX` tentatives, on abandonne. Avec 4 000 nœuds, des bornes a et b fixées à 2 et 4, et `MAX` à 10, je n'ai pas rencontré de cas d'abandon.

Le problème qui se pose maintenant est de choisir le moment d'exécution des requêtes stockées, dès lors que le nœud qui les a stockées devient actif. Il y en a trois :

- dès qu'un nouveau nœud a effectivement intégré le DST, c'est à dire au retour de la fonction `joindre`. En effet, les requêtes qui pourraient lui être adressées doivent attendre – état `'b'` – que toutes les opérations d'arrivée soient terminées – état `'a'` – pour s'exécuter.
- dans `send_msg_sync`, dès que la réponse attendue à la requête courante est reçue.
- enfin, pour un nœud quelconque actif du DST, on peut exécuter ces requêtes après chaque exécution de tâche (avant de se remettre en écoute)

De plus, il ne faut pas oublier le fait que l'exécution d'une de ces tâches stockées peut provoquer un nouveau stockage. Une fois les tâches stockées exécutées, il faut donc s'assurer que le stockage est bien vide avant de poursuivre.

3.8 Algorithmes des fonctions de synchronisation

Toute la problématique de synchronisation ayant été posée, nous présentons ici en détails les algorithmes des deux fonctions `send_msg_sync` et `attente_terminaison` mentionnées en début de chapitre. (voir page 26)

Pour rappel, `send_msg_sync` est utilisée pour envoyer une requête synchrone et attendre sa réponse. Et `attente_terminaison` est utilisée chaque fois qu'on a besoin de s'assurer qu'un ensemble de requêtes asynchrones a bien été exécuté par ses destinataires. Ces deux fonctions impliquent donc des attentes qui doivent être conçues de telle sorte qu'elles ne présentent pas de risque de blocage.

3.8.1 Fonction `send_msg_sync`

Les arguments de cette fonction sont `type` (le type de requête à envoyer), `dest` (le destinataire de l'envoi) et `args` (les arguments de la requête).
`réponse` contient la réponse qui sera finalement retournée par `send_msg_sync`.

Comme déjà indiqué, les caractéristiques de chaque requête sont empilées dès leur envoi. On peut remarquer que la requête courante se trouve bien au sommet de la pile. (algorithme 4 ligne 4)

Lors de mes expérimentations, j'ai remarqué que l'équilibrage de charge pouvait poser problème. En effet, il parcourt la totalité d'une table de routage en interrogeant chacun de ses nœuds. Ces échanges synchrones interférant avec d'éventuelles arrivées de nouveaux nœuds, le passage à l'état `'a'` du nœud en cours d'équilibrage peut être considérablement retardé, ce qui peut amener à des blocages.
L'équilibrage de charge n'étant pas une fonction indispensable au fonctionnement du DST, une solution simple à ce problème est de limiter le temps d'attente d'une requête

Algorithme 4 : Fonction `send_msg_sync`

```
1: Def send_msg_sync(type, dest, args)
2:     créer requête
3:     envoi_async(requête, dest)
4:     empiler infos requêtes sur pile_infos_requêtes

5:     timeout ← get_clock()
6:     tant que (get_clock() − timeout ≤ MAX ||
                 type ≠ demander_nouveau_rep) faire
7:         tâche_reçue ← reçoit_sync()
8:         réponse ← tâche_reçue.données
```

`demander_nouveau_rep`. Si la réponse n'est pas reçue au bout d'un certain temps `MAX` [19], on garde le représentant actuel et on passe au suivant, d'où la condition de la boucle de réception (ligne 6).
L'attente d'une réponse se fait de façon synchrone (ligne 7). Il n'y aura pas de blocage puisque toutes les tâches reçues sont traitées.

Algorithme 4 : partie 2

```
9:         si (tâche_reçue est une requête) alors
10:            exécute tâche_reçue
11:            si (sommet(pile_infos_requêtes) contient la réponse) alors
                                          ▷ réponse reçue entre temps ?
12:                réponse ← sommet(pile_infos_requêtes).réponse
13:                dépile pile_infos_requêtes
14:                si (moi.actif = 'a' && pile_tâches pas vide) alors
15:                    pour tout tâche ∈ pile_tâches faire
16:                        exécute tâche
17:                        dépile pile_tâches
18:                    fin pour
19:                fin si
20:                break
21:            fin si
```

La tâche reçue est une requête qu'il faut exécuter. [20] (ligne 10) Lorsqu'elle est terminée, il faut regarder si la réponse attendue est arrivée entre temps (lignes 11 – 13). En effet, si la tâche qui vient d'être exécutée a appelé `send_msg_sync` ou la fonction d'attente décrite plus loin, elle peut très bien avoir reçu la réponse attendue ici et l'aurait donc stockée dans `pile_infos_requêtes`.

19. `get_clock()` est une fonction qui retourne l'heure courante.
20. C'est la fonction chargée d'exécuter la requête qui vérifie si la requête est autorisée ou pas et qui agit en conséquence.

La réponse attendue ayant été reçue, on peut exécuter les tâches éventuellement stockées si le nœud courant est actif (lignes 14 – 19).

Algorithme 4 : partie 3

```
22:     sinon                                   ▷ tâche_reçue est une réponse
23:         index ← position requête dans pile_infos_requêtes
                                                ▷ index = -1 si requête pas trouvée
```

La tâche reçue est une réponse. On commence par vérifier si elle répond à une des requêtes synchrones stockées (ligne 23).
Comme indiqué plus haut, on cherche dans la pile une requête adressée à l'émetteur de la réponse reçue et ayant le même type qu'elle. Parmi l'ensemble des requêtes stockées satisfaisant ces critères, il faut en choisir une qui n'a pas encore reçu de réponse. `index` indique la position de cette requête dans la pile.

Algorithme 4 : partie 4

```
24:     si (index = taille(pile_infos_requêtes)) alors
                                                ▷ c'est la réponse attendue
25:         dépile pile_infos_requêtes

26:         si (moi.actif = 'a' && pile_tâches pas vide) alors
27:             pour tout tâche ∈ pile_tâches faire
28:                 exécute tâche
29:                 dépile pile_tâches
30:             fin pour
31:         fin si
32:         break
```

Si `index` désigne le sommet de la pile (ligne 24), alors cette réponse est la réponse à la requête courante. Puisqu'elle a déjà été récupérée (ligne 8), il suffit de dépiler `pile_infos_requêtes`.
Si le nœud courant est actif, on peut exécuter l'ensemble des tâches stockées, le cas échéant (lignes 26 – 31).

Algorithme 4 : partie 5

```
33:             sinon
                                          ▷ tâche_reçue n'est pas la réponse attendue
34:                 si (index > −1) alors
35:                     stocke cette réponse à la position index dans pile_infos_requêtes
36:                 sinon
                                          ▷ s'agit-il d'un accusé réception de requêtes asynchrones ?
37:                     idx ← position requête dans tableau_infos_requêtes
38:                     si (idx ≠ −1) alors
39:                         marque tableau_infos_requêtes[idx] comme reçue
40:                         enregistre réponse dans tableau_infos_requêtes[idx]
41:                     fin si
42:                 fin si
43:             fin si
44:         fin si
45:     fin tant que
```

Si `index` désigne une autre position que le sommet, on stocke la réponse à cette position, à condition qu'elle soit valide (lignes 34 – 35). Sinon, peut-être que cette réponse est un accusé réception de requêtes asynchrones (lignes 37 – 38). Si c'est le cas, la requête correspondante est marquée comme "reçue", on y ajoute la réponse reçue et on se remet en écoute.

Si la requête n'est trouvée dans aucune des deux structures, cela signifie que la réponse reçue n'était pas attendue et on peut simplement l'ignorer.

Algorithme 4 : partie 6

```
46:     si (get_clock() − timeout ≥ MAX  &&  type = demander_nouveau_rep) alors
47:         si (taille(pile_infos_requêtes) > 0) alors
48:             si (sommet(pile_infos_requêtes).type = demander_nouveau_rep) alors
49:                 dépile pile_infos_requêtes
50:             fin si
51:         fin si
52:     fin si
53:     retourne réponse
54: Fin
```

Si on est sorti de la boucle de réception pour cause de *timeout*, il faut dépiler la requête dont la réponse ne sera donc jamais reçue.

À la fin, la réponse est retournée.

3.8.2 Fonction `attente_terminaison`

Comme expliqué précédemment, il est parfois nécessaire de s'assurer qu'un ensemble de tâches asynchrones est bien terminé avant de passer à la suite. Par exemple, lorsqu'on réalise les scissions d'un étage donné, il faut qu'elles soient toutes terminées avant de passer aux scissions de l'étage inférieur.

Pour réaliser cela, on va mettre en œuvre une fonction d'attente dont voici l'idée : après avoir lancé toutes les tâches souhaitées de façon asynchrone, on attend que tous les nœuds destinataires de ces messages signalent qu'ils ont terminé leur travail. Ce n'est que lorsque l'ensemble des réponses a été reçu qu'on peut quitter cette attente pour passer à la suite. [21]

La problématique ici est semblable à celle de `send_msg_sync` : l'attente ne doit pas être bloquante. Donc pendant cette attente :

- parmi les requêtes reçues, certaines doivent être différées et d'autres exécutées de suite
- il faut gérer les possibles appels en cascade de cette fonction d'attente
- il faut pouvoir recevoir et traiter des réponses destinées à des requêtes synchrones

On met donc en œuvre des mécanismes semblables à ceux que nous venons de voir.

⋆ ⋆ ⋆

Lorsqu'il est nécessaire d'attendre qu'un ensemble de tâches asynchrones soit terminé avant de poursuivre, cette fonction d'attente est appelée. `cpt` est le nombre d'accusés de réception attendus, `dest_tab` est un tableau contenant des structures {*id, type, réponse*} – les nœuds dont on attend des réponses, le type de réponse attendu et la réponse – et `taille_tab` est la taille de ce tableau.

La fonction retourne la valeur `ret` : '`NOK`' si au moins l'un des destinataires renvoie cela au lieu de son accusé de réception normal, ou '`OK`' si tout s'est bien passé. Cette fonction d'attente étant utilisée par la fonction de diffusion, il s'agit de faire remonter le fait qu'une diffusion se passe mal pour agir en conséquence (*cf.* explications sur l'état `'u'` en 3.6.2 page 37).

Voici l'algorithme de cette fonction :

21. Les derniers essais de simulation de construction du DST que j'ai pu effectuer remettent partiellement en cause ce principe. Voir page 66 pour plus d'explications.

Algorithme 5 : Fonction `attente_terminaison`

```
1: Def attente_terminaison(cpt, dest_tab, taille_tab)
2:     ret ← OK

3:     si (moi.dest_tab n'existe pas) alors
4:         crée moi.dest_tab
5:         y recopie dest_tab                          ▷ ne doit pas contenir moi
6:     sinon
7:         y ajoute dest_tab
8:     fin si
```

Comme déjà mentionné, chaque sommet possède un tableau contenant l'ensemble des sommets dont on attend l'accusé réception. Pour gérer correctement les possibles appels en cascade de `attente_terminaison` (ligne 31), on commence par ajouter la liste des sommets fournie en argument (`dest_tab`) à ce tableau global (`moi.dest_tab`). On s'assure ainsi de ne manquer aucune réponse.

Algorithme 5 : partie 2

```
9:     tant que (durée < attente_max  &&  cpt > 0) faire
10:        tâche_reçue ← reçoit_sync()

11:        si (tâche_reçue est une réponse) alors
12:            pos_rep ← index_tab(tâche_reçue.émetteur, tâche_reçue.type,
                                   moi.dest_tab)
13:            si (pos_rep ≠ −1) alors
14:                si (pos_rep ≤ moi.taille_tab − 1  &&
                       pos_rep ≥ moi.taille_tab − cpt) alors

                                ▷ tâche_reçue est une des réponses attendues par l'appel courant

15:                    si (ret ≠ NOK) alors
16:                        ret ← tâche_reçue.réponse
17:                    fin si
18:                    cpt − −
19:                    ôte l'entrée pos_rep de moi.dest_tab
20:                    moi.taille_tab − −
```

La boucle d'attente (synchrone) des réponses se déroule tant que toutes les réponses n'ont pas été reçues. (ligne 9)

On utilise `attente_max` pour arrêter le programme si on n'a pas reçu tous les accusés

réception dans le temps imparti, ce qui permet de détecter d'éventuelles anomalies. [22]

Le message reçu est une **réponse.** (ligne 11)

S'il s'agit d'une des réponses asynchrones attendues [23] (lignes 12–13), on regarde s'il s'agit d'une de celles attendues par l'appel courant de cette fonction d'attente (ligne 14).

Si c'est le cas, on commence par positionner la valeur de retour `ret` [24] (lignes 15–17), puis on décrémente le compteur de réponses et on ôte l'entrée correspondante du tableau global (lignes 18–20) : l'un des destinataires a répondu.

Algorithme 5 : partie 3

```
21:         sinon                ▷ tâche reçue est une réponse attendue par un appel parent
22:             marque moi.dest_tab[pos_rep] comme reçue
23:             moi.dest_tab[pos_rep].réponse ← tâche_reçue.réponse
24:         fin si
```

Si ce n'est pas une des réponses à l'appel courant, c'est une de celles attendues par un appel parent. Il faut alors marquer cette requête comme étant reçue et y enregistrer la réponse. (toujours pour pouvoir gérer les `'OK'` et `'NOK'`)

Algorithme 5 : partie 4

```
25:         sinon                ▷ tâche_reçue est-elle attendue par send_msg_sync ?
26:             pos_pile ← position de {tâche_reçue.émetteur, tâche_reçue.type} dans
                           pile_infos_requêtes
27:             si (pos_pile > −1) alors
28:                 enregistre tâche_reçue.données
                    dans pile_infos_requêtes à la position pos_pile
29:             fin si
30:         fin si
```

Si ce n'est pas une réponse asynchrone, alors c'est peut-être une des réponses attendues par `send_msg_sync` (ligne 26). Si oui, (ligne 27) il faut alors enregistrer cette réponse au bon endroit de la pile des requêtes synchrones (ligne 28).

Si on n'est dans aucun de ces cas, on peut ignorer cette réponse qui n'était donc pas attendue. (voir explications à la section 3.5.1, page 33)

22. A ce stade, je ne me suis pas intéressé à la tolérance aux pannes, il s'agit plutôt de détecter des dysfonctionnements en marche normale.
23. accusé réception ou `'NOK'`, donc
24. On rappelle que `ret` doit valoir `'OK'`, à moins qu'un des destinataires réponde `'NOK'`.

Algorithme 5 : partie 5

```
31:         sinon                                   ▷ tâche_reçue est une requête
32:             exécute tâche_reçue
                                                    ▷ réponses reçues entre temps ?
33:             pour idx ← moi.taille_tab − 1, moi.taille_tab − cpt faire
34:                 si (moi.dest_tab[idx] marqué comme reçu) alors

                                                    ▷ regarde si l'une des réponses est 'NOK'
35:                     si (moi.dest_tab[idx].réponse existe && ret ≠ 'NOK') alors
36:                         ret ← moi.dest_tab[idx].réponse
37:                     fin si

38:                     cpt − −
39:                     ôte l'entrée idx de moi.dest_tab
40:                     moi.taille_tab − −
41:                 fin si
42:             fin pour
43:         fin si
44:     fin tant que

45:     si (cpt ≠ 0) alors
46:         affiche "Erreur : attente trop longue"
47:         arrête le programme
48:     fin si
49: Fin
```

Le message reçu est une **requête** qui est immédiatement exécutée (ligne 32). [25]

Ceci fait, on regarde si des réponses sont arrivées entre temps (lors d'une autre exécution de `attente_terminaison` — lignes 21–24) pour éventuellement les récupèrer (lignes 35–37) et mettre le tableau `dest_tab` et le compteur de réponses à jour en conséquence (lignes 38–40).

En sortie de la boucle d'attente (ligne 45), si le compteur de réponses attendues par l'appel courant n'est pas nul, cela signifie que le délai d'attente `attente_max` (ligne 9) a été dépassé, auquel cas le programme est arrêté : c'est probablement le signe d'un *deadlock*.

25. On rappelle que c'est la fonction d'exécution des tâches qui se charge de gérer les éventuelles requêtes non autorisées.

Exemple d'utilisation de `attente_terminaison` dans la diffusion

Voici l'algorithme utilisé pour la diffusion :

Algorithme 6 : Diffusion simple

```
1: Def diffuser(étage, req)                    ▷ diffuse la requête req depuis l'étage étaget
2:     si (étage = 0) alors
3:         traiter req localement
4:     sinon
5:         pour tout f dans moi.frères[étage][:] faire
6:             si (f ≠ moi) alors
7:                 send_msg_async(moi.nom, f, diffuser, (étage − 1, req))
8:             sinon
9:                 diffuser(étage − 1, req)
10:            fin si
11:        fin pour
12:    fin si
13: Fin
```

Si on ne prend pas de précaution dans le déroulement de cet algo, voici ce qu'il se passe pour un étage donné : un message de diffusion est envoyé à chaque frère, la diffusion locale a lieu et on rend la main. A ce stade, on n'a aucune garantie que les diffusions soient achevées, les envois étant asynchrones (ligne 7). Cela pose problème par exemple dans `demander_scission` (A.2.6) : on utilise cette fonction de diffusion pour tout d'abord ajouter un étage, puis pour scinder. Il va de soi que l'ajout d'étage doit être terminé avant que la diffusion de la scission ne puisse commencer.

C'est donc ici que `attente_terminaison` entre en jeu : (voir algorithme page 55)

Après avoir envoyé la tâche de diffusion de façon asynchrone, le type de la requête et son destinataire sont enregistrés dans le tableau `dest` (ligne 12) transmis à `attente_terminaison` (ligne 19).

À noter que la diffusion locale est réalisée en tout dernier (ligne 21), une fois les autres diffusions terminées. Si elle est réalisée avant, il y a un risque de descendre trop tôt à des étages inférieurs pas encore à jour et d'aboutir à des incohérences de construction. Il faut s'assurer de travailler étage par étage.

Il faut aussi remarquer (ligne 15) qu'on décrémente le compteur de réponses, puisqu'on n'attend pas de réponse de la diffusion locale.

Algorithme 7 : Diffusion synchronisée

```
1: Def diffuser(étage, req)
2:     si (étage = 0) alors
3:         traite req localement
4:     sinon
5:         cpt ← taille(moi.frères[étage])
6:         dest ← []
7:         taille ← 0
8:         diffusion_locale ← 0
9:         pour tout f dans moi.frères[étage] faire
10:            si (f ≠ moi) alors
11:                send_msg_async(moi.nom, f, diffuser, (étage − 1, req))
12:                dest.ajoute(req.type, f.nom)
13:                taille + +
14:            sinon
15:                cpt − −
16:                diffusion_locale ← 1
17:            fin si
18:        fin pour
19:        moi.attente_terminaison(cpt, dest, taille)
20:        si (diffusion_locale = 1) alors
21:            moi.diffuser(étage − 1, req)
22:        fin si
23:    fin si
24: Fin
```

Chapitre 4

Retrait de sommets d'un DST

4.1 Introduction

Pour cette partie, les algorithmes n'avaient pas encore été précisément décrits, seules des idées avaient été proposées ([Dob07, DDNP07, DDNP08]). Cette étude se poursuit donc par la conception de ces algorithmes et leur mise en œuvre dans *Simgrid*. Voici une présentation globale de ce travail dont les détails figurent dans l'**annexe B** page 93.

4.2 Conception des algorithmes

Tout comme l'arrivée d'un sommet dans le DST peut nécessiter des scissions de groupes pour que leur taille reste comprise entre les bornes a et b, le départ d'un sommet peut entraîner des fusions pour la même raison, comme nous le voyons sur la figure 4.1. Celle-ci montre un exemple de ce qu'il se passe lors du retrait d'un sommet d'un DST : le départ du sommet `8` laisse un sommet *orphelin*[1] : le `21`.

Étage 0 : Le groupe `0B` ayant de la place pour accueillir `21`, il va pouvoir fusionner avec `0A` pour donner le groupe `0AB`.

Étage 1 : En conséquence, le groupe `1A` n'a plus qu'un seul membre et va devoir fusionner avec `1B` pour donner le groupe `1AB`.

Étage 2 : À son tour, le groupe `2B` n'a plus qu'un membre et doit fusionner avec `2A` pour obtenir le groupe `2AB`

Dans cet exemple, le DST ne perd pas de niveau.

Un point important de cette partie est donc la gestion de ces sommets orphelins (c'est à dire en nombre inférieur à la borne a du DST). Il y a deux façons de faire : soit ils peuvent

1. J'appelle sommets *orphelins* des sommets qui se retrouvent en trop petit nombre pour constituer un groupe, suite au départ d'un sommet.

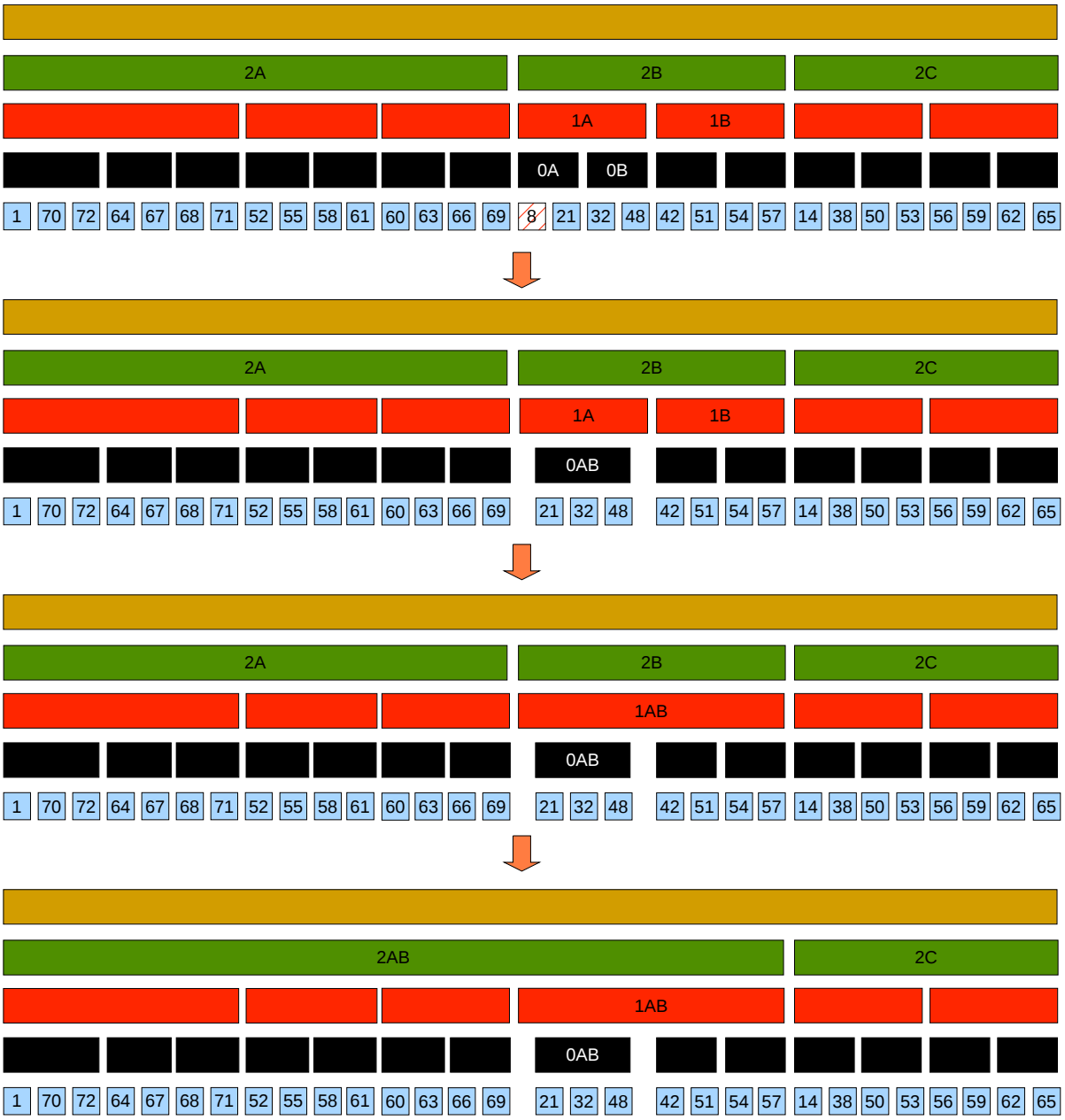

FIGURE 4.1 – Exemple de fusion : le sommet 8 quitte le DST

être répartis dans un ou plusieurs groupes voisins – on parle alors de *fusion* – soit il est possible que les groupes dont ils sont membres puissent eux-mêmes être complétés par des sommets pris aux voisins – auquel cas on parle de *transfert*. Tout ceci en respectant les bornes a et b du DST, bien sûr.

Par manque de temps, le seul cas présenté dans cette étude est celui où la fusion dans un seul groupe accueillant est possible et les autres cas restent donc à étudier. (voir remarque à ce sujet page 107)

Selon le même principe que pour les ajouts de sommets, l'algorithme de retrait se compose d'un ensemble de fonctions mises en œuvre comme indiqué dans l'algorithme général 8.

Algorithme 8 : Algorithme général du retrait d'un nœud

```
1: un nœud souhaite se retirer :                    quitte
2: gère les demandes de fusions :                     demande_fusion
3: réalise les fusions de chaque étage :                fusion*
4: ôte le membre en trop de l'étage supérieur : nettoie_étage_sup
5: supprime la racine :                                 supprimer_racine
```

Dans la suite, ces fonctions sont sommairement présentées.

4.2.1 Fonction `quitte`

Il s'agit de la fonction principale, celle qui est appelée par le sommet qui souhaite quitter le DST. Elle se compose de deux parties :

1. Tout d'abord, puisque le sommet courant s'en va, il ne peut plus être utilisé comme représentant de son groupe. Cette partie est donc chargée de remplacer ce sommet courant – voir `remplace_frère` – par un autre de ses frères de niveau 0 (choisi aléatoirement), partout où cela est nécessaire. Dans l'exemple, `8` est remplacé par `21`.
2. Ensuite, si des fusions sont nécessaires (c'est à dire s'il y a moins de a frères au niveau 0 du groupe courant), cette fonction charge un des frères restants de déclencher les opérations de fusion. (voir `demande_fusion`) Dans l'exemple, c'est donc `21` qui en sera chargé.

4.2.2 Fonction `remplace_frère`

Cette fonction remplace un frère par un autre à un étage et un emplacement donnés et met à jour les prédécesseurs en conséquence.

4.2.3 Fonction `demande_fusion`

Cette fonction est chargée de gérer les fusions. Elle parcourt le DST de bas en haut tant que des orphelins subsistent, comme on le voit sur l'exemple.

Si c'est le cas, on regarde à chaque étage s'il faut réaliser une fusion ou un transfert. Si une fusion est possible, on dispose d'un représentant du groupe à rejoindre. Ce représentant est alors chargé de réaliser une première fusion pour lui-même. Sa table de routage étant alors à jour, on peut lui demander de diffuser une tâche de fusion à l'ensemble des sommets concernés par la fusion, c'est à dire les membres des deux groupes qui fusionnent.

À la fin, il faut s'occuper de la racine si les fusions successives l'ont atteinte :

1. Si elle doit s'en aller, il faut en informer l'ensemble du DST au moyen d'une diffusion globale.
2. Si elle reste, elle comporte un membre de moins à cause des fusions. Les sommets impactés par les opérations de fusion en ont été informés et leur table de routage est à jour, mais ce n'est pas le cas des autres sommets du DST. Sur notre exemple, à la dernière étape, tous les sommets membres du groupe `2AB` sont à jour puisqu'ils sont concernés par les opérations de fusion. Mais la partie droite – les sommets membres du groupe `2C` – n'a pas eu connaissance que des fusions se produisaient. Il faut donc diffuser la tâche `nettoie_étage_sup` chargée de cette mise à jour dans cette partie droite.

4.2.4 Fonction `fusion_ou_transfert`

Cette fonction détermine si une fusion est possible. Ce sera le cas si l'un des groupes voisins de celui qui a perdu un membre possède suffisamment de place pour accueillir les membres restants. Chaque membre de l'étage parent de celui où se trouvent les orphelins représente un groupe susceptible de les accueillir. Ces membres sont donc successivement interrogés jusqu'à ce que l'un d'entre eux réponde que son groupe a de la place. Dans ce cas, cette fonction retourne l'identifiant de ce membre qui servira donc de représentant du groupe à rejoindre. Sinon, elle retourne -1.

4.2.5 Fonction `fusion`

C'est cette fonction qui réalise la fusion proprement dite de deux groupes, à un étage donné. L'opération consiste essentiellement à prendre un membre dans chaque groupe, puis à "concaténer" leurs tables de routage à l'étage en question. Il faut également modifier les tables de prédécesseurs en conséquence. Cette fonction réalise cela tout en maintenant l'ordre chronologique d'arrivée des sommets dans les tables. Cela est important pour la cohérence du DST, ainsi que pour le choix du leader (qui est toujours le plus ancien sommet du nœud).

4.2.6 Fonction `nettoie_étage_sup`

Cette fonction est chargée de supprimer le membre en trop de l'étage parent de celui qui vient de subir une fusion. En effet, lorsque les fusions sont terminées, cet étage doit comporter un membre de moins. C'est aussi cette fonction qui est appelée pour corriger la racine, comme cela a été vu plus haut.

Reprenons l'exemple 4.1. Voici la table de routage du nœud `32` avant l'arrivée de `21` au niveau 0 : [2]

Nœud 32 :

E0	32	48		
E1	21	32		
E2	32	42		
E3	1	32	14	

Et voici la même table après l'arrivée de `21` :

Nœud 32 :

E0	21	32	48	
E1	21	32		
E2	32	42		
E3	1	32	14	

On peut voir qu'il y a un problème puisque l'étage 1 pointe sur 2 représentants du même étage 0. Cette fonction va donc corriger le problème en supprimant `21` à l'étage 1.

4.2.7 Fonction `diffuse_fusion`

Cette fonction est simplement chargée de diffuser la tâche de fusion à tous les nœuds concernés. J'en ai fait une fonction parce que `demande_fusion` doit demander à son contact – le représentant du groupe à rejoindre – de lancer cette fonction.

4.2.8 Fonction `supprimer_racine`

Cette fonction supprime simplement l'étage racine des tables courantes. C'est elle qui est diffusée à l'ensemble du DST en cas de disparition de la racine suite aux opérations de fusion.

2. On rappelle que `8` a déjà été remplacé par `21` partout où il se trouvait par la première partie de la fonction `quitte`.

4.3 Étude des problèmes de synchronisation

Dans cette partie, il faut s'intéresser à deux aspects du problème : (a) les problèmes de synchronisation dûs aux seules fonctions de gestion des retraits (b) et ceux posés par la simultanéité de départs et d'arrivées.

Plusieurs des solutions déjà éprouvées dans le cas des fonctions d'ajout pourront certainement être ré-utilisées ici – en particulier les mécanismes d'attente – mais il convient de mener ici le même travail d'analyse que précédemment. Par manque de temps, cela n'a pas pu être fait.

Troisième partie

Perspectives

Chapitre 5

Synthèse

5.1 Observations

Au début de ces travaux et comme expliqué en section 1.2 (page 6), j'ai fait le choix d'étudier une solution qui, en tentant de tirer le meilleur parti de la parallélisation présente dans ce contexte, devait présenter les meilleures performances. A l'issue de cette étude, je remarque les points suivants :

1. les problèmes de synchronisation sont complexes à détecter et à analyser.
2. il est difficile de montrer que tous les cas de figure ont bien été couverts par les solutions proposées.
3. ces solutions génèrent un nombre de messages supplémentaires non négligeable.
4. l'affirmation posée au début de l'étude qui disait qu'intégrer un groupe de nouveaux nœuds les uns après les autres allait prendre plus de temps que de les accepter tous en même temps, ne paraît maintenant plus si évidente. En effet, quand on voit la progression du nombre de messages nécessaires à la construction du DST en fonction de sa taille (voir figure 5.1 page 70), il semble nécessaire d'effectuer davantage de mesures avec et sans synchronisations pour les comparer et ainsi vérifier cette affirmation.

5.2 Suite des travaux

5.2.1 Concernant le DST lui-même

Il reste encore du travail concernant l'étude de la vie du DST sur Simgrid. En voici quelques exemples.

Problèmes de synchronisation

Synchronisation des fonctions de départ Par manque de temps, je n'ai pas étudié les synchronisations à mettre en place sur les algorithmes gérant le départ d'un nœud. Il s'agit d'une part, de réaliser le même travail sur ces fonctions de départ que celui qui a été fait sur les fonctions d'arrivée, et d'autre part, d'étudier le problème de la synchronisation entre les départs et les arrivées. Pour ce dernier point, on se retrouve dans une problématique semblable à celles déjà étudiées : faut-il simplement empêcher toute simultanéité de départ et d'arrivée ou peut on en autoriser certaines (si elles ont lieu dans des parties éloignées du DST, par exemple) ?

La fonction `attente_terminaison` Lors d'ultimes tests de ces solutions de synchronisation, j'ai trouvé des cas de figure où cette fonction d'attente a été prise en défaut. Je n'ai pas eu le temps de la reprendre, mais voici quelques pistes de réflexion issues de ces observations, pour corriger le problème.

Pour mémoire, voici l'usage général qui est fait de cette fonction :

- un ensemble d'envois asynchrones a lieu
- on constitue un tableau de l'ensemble des destinataires de ces envois ...
- ... qu'on passe en argument à `attente_terminaison`
- `attente_terminaison` se met à l'écoute des réponses de ces destinataires

Prenons par exemple, la fin de la fonction `scission` (A.2.9) à laquelle la synchronisation est ajoutée (voir algorithme 9, page 67) :

Si on augmente la borne b [1] du DST pour allonger la durée de la boucle d'envoi, on constate que des accusés de réception peuvent être renvoyés avant que la fonction d'attente ne soit lancée (et donc, avant que le tableau des destinataires ne soit constitué).
Or, il est possible qu'au même moment, d'autres instances d'`attente_terminaison` – lancées par le même nœud – soient en cours d'exécution. Du coup, ces accusés de réception sont reçus par ces attentes et rejetés, puisque leur émetteur n'est alors pas encore connu. L'instance courante d'`attente_terminaison` ne reçoit donc jamais ces réponses et le programme s'arrête en *timeout*. [2]

Idée de solution Comme expliqué plus haut (3.8.2), `attente_terminaison` s'appuie sur le tableau global des destinataires pour savoir de qui on doit attendre une réponse. Ce tableau global est constitué de l'agrégation des groupes de destinataires successivement passés en argument à la fonction d'attente. Puisque cet ajout a lieu trop tard dans notre exemple, il faudrait procéder différemment : ne plus passer les destinataires en argument de

1. on peut constater le phénomène à $b = 15$
2. Il ne s'agit pas ici d'un *deadlock* à proprement parler puisque nous ne sommes pas dans le cas où deux nœuds attendent l'un sur l'autre. Il s'agit plutôt d'un nœud qui attend une réponse qui a été "consommée" par un autre et qui ne la recevra donc jamais.

Algorithme 9 : Scission (extrait)

```
cpt ← taille(moi.pred[étage + 1][:])

pour tout p dans moi.pred[étage + 1][:] faire
    si (p = moi) alors
        cpt − −
        connecter_les_groupes_scindés(étage + 1, init_idx, init_rep, nouv_idx,
                                      nouv_rep)
    sinon
        send_msg_async(moi.nom, p, connecter_les_groupes_scindés,
                       (étage + 1, init_idx, init_rep, nouv_idx, nouv_rep))
    fin si
fin pour

si (cpt > 0) alors
    pour tout idx ← 0, taille(moi.pred[étage + 1][:]) − 1 faire
        dest_tab[idx] ← moi.pred[étage + 1][idx]
    fin pour
    attente_terminaison(cpt, dest_tab, taille(moi.pred[étage + 1]))
fin si
```

la fonction, mais les inscrire directement dans le tableau global dès que l'envoi de message a eu lieu. Ainsi, n'importe quelle instance de `attente_terminaison` a connaissance de l'ensemble des destinataires dont elle peut recevoir une réponse et elle ne risque pas d'en rejeter par erreur.

Solutions de synchronisation

Il serait intéressant de développer la solution de synchronisation qui consiste à ne laisser entrer qu'un seul nouveau nœud à la fois. Le but serait double : (a) vérifier si la gestion des files d'attentes pose bien un problème d'extensibilité ainsi qu'on le pressent, (b) comparer les résultats obtenus avec la solution présentée dans cette étude, en termes de nombre de messages et temps d'exécution.

On peut remarquer que cette solution revient à rendre les arrivées de nœuds séquentielles.

Simgrid et les traces Pour étudier ce qu'il se passe lors de la vie du DST, le programme génère des traces sous forme de texte qu'il faut ensuite examiner. Dès que la plate-forme mise en œuvre atteint quelques milliers de nœuds, cette trace devient vite très volumineuse et difficile à exploiter. Simgrid propose d'autres solutions de générations de traces (dont une graphique) que je n'ai pas eu le temps d'étudier. Il serait certainement intéressant de le faire pour pouvoir ensuite augmenter la taille des plates-formes de test.

Tolérance aux pannes

Il s'agit d'un point important qui conditionne, entre autres, la robustesse du DST. Pour cela, il faut faire en sorte de détecter un sommet qui ne répond plus et définir ce qu'il faut faire dans ce cas. En particulier, il faut étudier le moyen de déclencher les opérations de retrait d'un sommet dont l'arrêt a été détecté.

En outre, il faut examiner les conséquences d'une panne d'un leader. Comment se passerait son remplacement ? Dans sa thèse, ([Dah05], page 82) Sylvain Dahan évoque la possibilité de distribuer le rôle de leader à plusieurs sommets en vue d'améliorer la tolérance aux pannes. Il est possible de tester tout cela avec Simgrid.

Conception des algorithmes

Lors de la conception des algorithmes de construction du DST, différents choix ont été faits et là encore, il serait intéressant de pouvoir construire et mettre en œuvre d'autres options pour en comparer les performances.

1. Si le contact utilisé par un nouveau nœud représente un groupe déjà plein, on va lancer des scissions. C'est ce qui a été fait ici. Une autre solution pourrait être de chercher un autre nœud qui aurait de la place, évitant ainsi des scissions, mais générant bien sûr des messages pour trouver le bon groupe. (L'étendue de cette recherche pourrait d'ailleurs être bornée.) Cela reviendrait à équilibrer le DST au maximum avant de procéder aux scissions et il faut en vérifier l'intérêt.
2. Dans ce même ordre d'idée, les regroupements de nœuds réalisés par les algorithmes présentés ici se font sans respecter de critères précis concernant les nœuds joints. Il serait donc intéressant d'y ajouter cette possibilité pour ensuite pouvoir grouper les nœuds par centre d'intérêt commun, ou par distance géographique, par exemple.
3. Il me semble utile de vérifier les conséquences des choix de leaders, en particulier sur leur charge par rapport aux autres nœuds.
4. Pour réaliser l'équilibrage de charge, les nouveaux représentants sont choisis en fonction de leur charge. On peut les choisir selon d'autres critères ; aléatoirement ou en fonction de leur distance sur le réseau, par exemple.

Charge du réseau

Comme cela a déjà été dit, il est possible lors des diffusions, que le même message parvienne plusieurs fois à un même nœud. Ce trafic représentant une occupation du réseau inutile, il faudrait étudier la possibilité d'optimiser ces algorithmes pour éviter cette charge.

5.2.2 Utilisation du DST

Une fois le DST pleinement opérationnel sur Simgrid, il devient possible d'étudier son utilisation dans ce nouvel environnement.

Simgrid vs simulateur séquentiel

Les études menées sur le DST jusqu'ici – [Dah05], [DDNP07], [DDNP08] – ont montré que tant en utilisation stable que soumis à une forte dynamicité du réseau (jusqu'à un certain seuil), le DST tenait ses promesses par rapport aux arbres et aux graphes. Mais que deviendraient ces résultats avec le surcoût des solutions de synchronisation ? Les bonnes propriétés du DST seraient-elles impactées par ces surcoûts ou seraient-ils négligeables ?

Il me paraît essentiel de pouvoir refaire ces mêmes études de performances avec les solutions exposées ici. Il s'agit de comparer les résultats avec ceux obtenus avec le simulateur séquentiel, pour valider ou pas les solutions de synchronisation proposées.

Un exemple : le tableau 5.1 et les figures 5.1 et 5.2 montrent le nombre de messages émis pour construire un DST du nombre de nœuds indiqué, à l'aide des fonctions utilisant les synchronisations.

> *Remarque* : Les nouveaux nœuds utilisent des contacts choisis aléatoirement dans le DST et se présentent toutes les 10 unités de temps. Mais du fait des fonctions de synchronisations, les nœuds peuvent se voir refuser l'entrée et tentent à nouveau de se présenter un certain temps – aléatoire et borné – après. À partir d'une certaine taille de DST, la première tentative est majoritairement refusée et on peut donc considérer qu'ils se présentent finalement de façon aléatoire.

Ces relevés ne sont pas comparables en l'état avec ceux présentés dans la thèse (pages 83 et 85). En effet, dans celle-ci, un étage est ajouté au DST dès qu'il est plein alors que dans mes expérimentations, les contacts sont choisis aléatoirement et un étage peut donc très bien être ajouté alors qu'il restait de la place ailleurs dans le DST. De plus, je n'ai pas regardé la distribution des intervalles de nombres de messages, mais leur nombre total absolu. Enfin, mon DST a pour bornes 2 et 4 alors que dans la thèse elles valent 4 et 8.

Pour pouvoir effectuer ces comparaisons, il faudrait d'une part, ajouter toutes les fonctions et données nécessaires à ces études statistiques dans le code proposé dans cette étude, et d'autre part, arriver à faire fonctionner le simulateur séquentiel sur les mêmes plates-formes que celles utilisées avec Simgrid. (En particulier, il faut travailler sur les mêmes modèles de réseau et de plates-formes si on veut que les données obtenues soient comparables entre elles.)

Nb Nœuds	Nb Msg	Nb Moy	Nb Max	Nb Étages
10	352	35,2	119	2
50	4 228	84,56	811	4
100	13 016	130,16	1 389	4
200	28 720	143,6	2 876	5
500	91 770	183,54	8 009	6
1 000	215 608	215,61	13 912	6
1 500	361 486	240,99	27 107	7
2 000	517 620	258,81	34 602	7
2 500	662 382	264,95	34 668	7
3 000	824 214	274,74	41 898	7
3 500	943 086	269,45	41 701	7
4 000	1 177 142	294,29	41 718	7

TABLE 5.1 – Construction d'un DST ($a = 2,\ b = 4$)

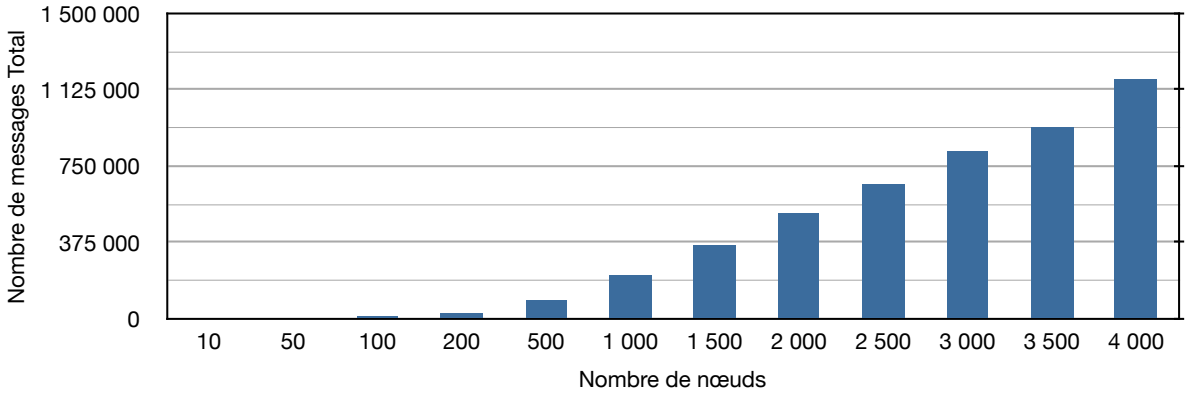

FIGURE 5.1 – Nombre de messages utilisés lors de la construction du DST

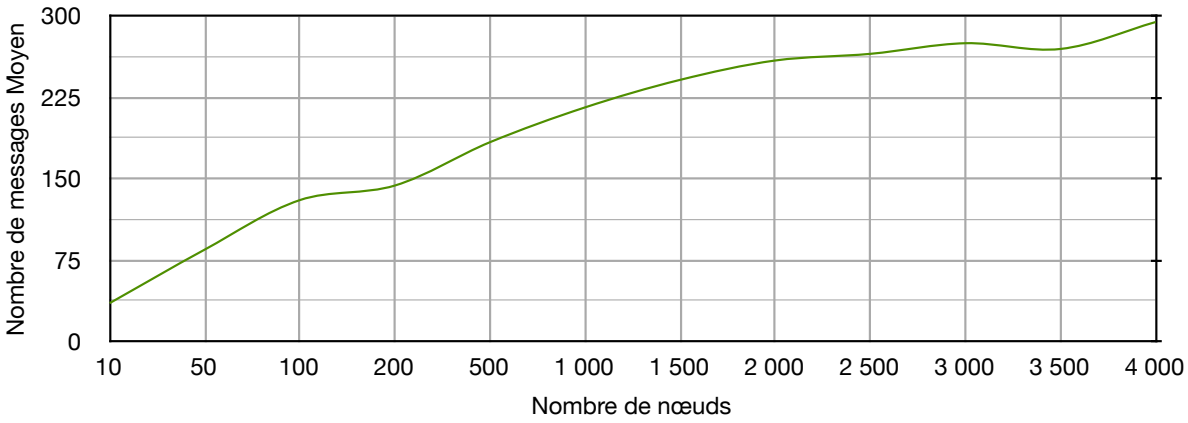

FIGURE 5.2 – Nombre moyen de messages utilisés lors de la construction du DST

Parcours du DST

Dans cette étude, je n'ai utilisé qu'une seule façon de parcourir un DST pour réaliser des diffusions. Dans la thèse [Dah05], pages 84 à 94, plusieurs parcours sont présentés et peuvent donc être étudiés avec Simgrid.

Chapitre 6

Conclusion

La structure de recouvrement DST a jusqu'ici été étudiée sur des simulateurs basés sur des architectures centralisées n'autorisant pas de parallélisation.

Le DST étant conçu pour s'appliquer à des réseaux *peer-to-peer* totalement décentralisés, le but de cette étude était de poursuivre ces travaux en le faisant fonctionner sur un simulateur basé sur des modèles plus proches de cet environnement et permettant la parallélisation. C'est le simulateur *Simgrid* qui a été choisi à cette fin.

La complexité des algorithmes présentés dans cette étude rend difficile la mise en place de preuves formelles. Ma démarche a donc été d'effectuer une validation expérimentale de ces algorithmes pour vérifier qu'ils étaient justes. Cela a été réalisé en portant l'ensemble des fonctions d'ajout et de retrait de sommets au DST sur *Simgrid*. Ce travail a également permis d'identifier et de proposer des solutions aux problèmes de synchronisation posés par la parallélisation et l'absence d'horloge globale. Ce travail est donc à compléter par une étude supplémentaire visant à démontrer et prouver ces algorithmes de façon formelle.

Une fois ces travaux achevés, on disposera alors d'un outil performant et réaliste pour poursuivre l'étude du comportement de cette nouvelle structure. Comme indiqué plus haut, il serait intéressant et utile de recommencer les études de performances déjà réalisées avec l'ancien simulateur pour comparer les résultats entre eux. Il sera ainsi possible de mesurer le coût des solutions de synchronisation proposées ici et donc de les valider ou pas.

Bien que beaucoup de travail reste à fournir pour exploiter pleinement le potentiel de cette nouvelle structure, elle semble prometteuse et le portage sur *Simgrid* était une étape essentielle pour pouvoir continuer ces travaux. Il faut également souligner le fait que puisque *Simgrid* autorise l'emploi de différents modèles de réseaux et qu'il peut même en simuler les défaillances, il sera possible d'étudier le comportement du DST dans tous ces environnements.

Bibliographie

[AD01] Rowstron Antony and Peter Druschel. Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. *International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany, November, 2001*, 2001.

[CLQ08] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid : a Generic Framework for Large-Scale Distributed Experiments. In *10th IEEE International Conference on Computer Modeling and Simulation*, March 2008.

[CNO99] James H. Cowie, David M. Nicol, and Andy T. Ogielski. Modeling the global internet. *Computing in Science & Engineering*, 1(1) :42–50, 1999.

[Dah05] S. Dahan. *Mécanismes de recherche de services extensibles pour les environnements de grilles de calcul.* Thèse de Doctorat, LIFC, Université de Franche-Comté, 13 décembre 2005. Rapporteurs : J.-F. Mèhaut, Professeur (IMAG, Univ. J. Fourier, Grenoble 1), P. Sens, Professeur (LIP6, Univ. Paris 6). Examinateurs : E. Caron, MdC (LIP, ENS Lyon), H. Guyennet, Professeur (LIFC, Univ. Franche-Comté), J.-M. Nicod, MdC (LIFC, Univ. Franche-Comté). Directeur : L. Philippe, Professeur (LIFC, Univ. Franche-Comté).

[DDNP07] Sylvain Dahan, Alexandru Dobrila, Jean-Marc Nicod, and Laurent Philippe. Performances study of management and traversal algorithms on the DST overlay network. Research Report RR2007-02, LIFC - Laboratoire d'Informatique de l'Université de Franche Comté, November 2007.

[DDNP08] Sylvain Dahan, Alexandru Dobrila, Jean-Marc Nicod, and Laurent Philippe. Étude des performances du distributed spanning tree : un overlay network pour la recherche de services. In *CFSE'6, 6ème Conf. Française en Systèmes d'Exploitation*, page (6 pages), Fribourg, Switzerland, February 2008. Publication électronique.

[DNP05] Sylvain Dahan, Jean-Marc Nicod, and Laurent Philippe. The Distributed Spanning Tree : A scalable interconnection topology for efficient and equitable traversal. In *5th Int. Symposium on Cluster Computing and the Grid (CCGrid 2005), workshop on Global and Peer-to-Peer Computing, GP2PC 2005*, Cardiff, United Kingdom, May 2005. IEEE Computer Society Press. CD-ROM.

[Dob07] Alexandru Dobrila. *Distributed Spanning Tree : une topologie pour la recherche de services de calcul sur la grille.* Mémoire de Master Recherche, LIFC, Uni-

versité de Franche-Comté, 28 août 2007. Jury : J.-M. Nicod, F. Bouquet, L. Philippe.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees : distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[Kta09] Salma Ktari. *Interconnexion et routage dans les systèmes pair à pair*. PhD thesis, Télécom ParisTech, 12 2009.

[LCP+05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, University Of Cambridge, Ravi Sharma, Nanyang Technological University, Steven Lim, and Microsoft Asia. A survey and comparison of peer-to-peer overlay network schemes, 2005.

[ns2] The network simulator (ns2) http://nsnam.isi.edu/nsnam/.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[Ril03] G.F. Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, Methods and Tools for Reproducible Network Research*, pages 5–12. ACM, 2003.

[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord : A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31 :149–160, August 2001.

[SR04] D. Stutzbach and R. Rejaie. Characterizing today's gnutella topology. Technical Report CIS-TR-04-02, University of Oregon, November 2004.

[WGR05] Klaus Wehrle, Stefan Götz, and Simon Rieche. 7. distributed hash tables. In Ralf Steinmetz and Klaus Wehrle, editors, *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*, pages 79–93. Springer Berlin / Heidelberg, 2005. 10.1007/11530657_7.

Annexes

Annexe A

Gestion des ajouts de sommets à un DST

Sommaire

Dans cette partie, j'ai repris les algorithmes proposés par Sylvain Dahan pour les implémenter en langage C dans *Simgrid*, au moyen de l'API *MSG*. Ces expérimentations sur simulateur ont permis de tester et de valider ces algorithmes, moyennant parfois quelques modifications. Je les présente donc à nouveau avec modifications et commentaires, le cas échéant. Par souci de clarté, je n'ai pas fait figurer les fonctions de synchronisation dans ces algorithmes. Elles seront détaillées plus loin.

⋆ ⋆ ⋆

Voici des exemples d'arrivées de sommets dans un DST ($a = 2,\ b = 4$) :

La figure A.1 montre l'arrivée du sommet `54` via un des membres du groupe `AAA`.

– Le groupe `AAA` ayant déjà 4 membres, il doit se scinder pour faire de la place à `54`. Cette scission donne les groupes `AAA0` et `AAA1`.

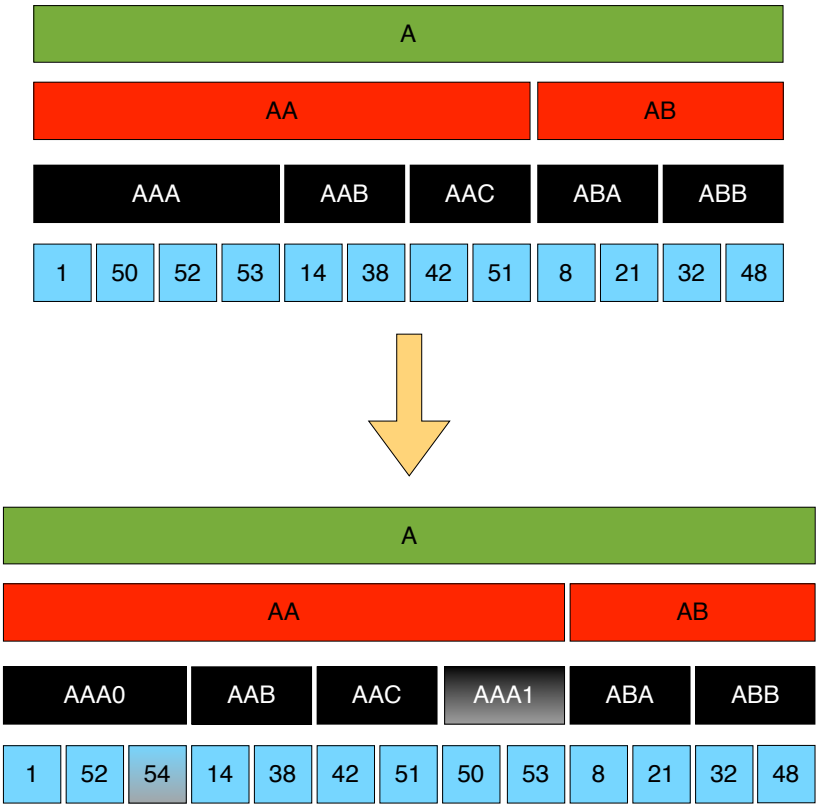

FIGURE A.1 – Arrivée d'un nœud : scission simple

- Le groupe `AA` passe de 3 à 4 membres, il n'a donc pas besoin de se scinder.
- `54` peut rejoindre le groupe `AAA` qui a maintenant de la place pour l'accueillir.

La figure A.2 montre l'arrivée du sommet `52` dans le groupe `AA`.

- Le groupe `AA` a 4 membres et doit se scinder pour donner les groupes `AA0` et `AA1`.
- Le groupe `A` doit alors également se scinder puisqu'il avait aussi déjà 4 membres. Cette scission donne les groupes `A0` et `A1`.
- Le groupe `A` était la racine. Un DST ne pouvant avoir qu'une seule racine, il est donc nécessaire d'ajouter un étage : c'est le groupe `A'`.

L'algorithme distribué permettant d'insérer un sommet à un DST est défini par un ensemble de fonctions. Ces fonctions sont exécutées lorsqu'un message correspondant est reçu.

A.1 Données enregistrées sur chaque sommet

Les sommets possèdent un ensemble de variables liées à la structure du DST. Chacun des sommets possède les variables suivantes :

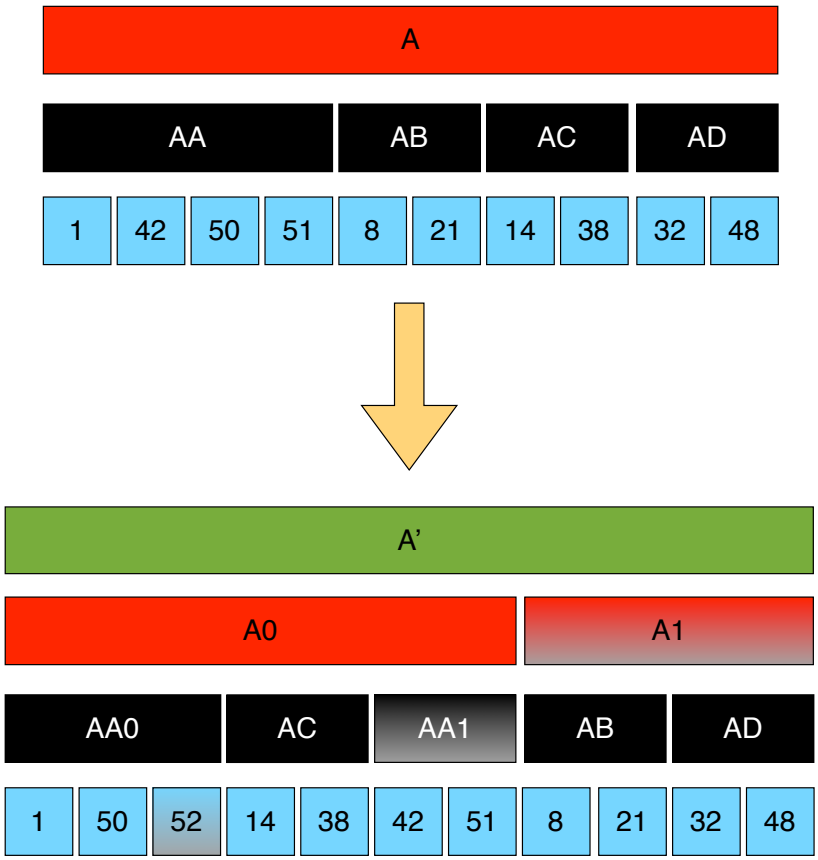

FIGURE A.2 – Arrivée d'un nœud : scissions avec ajout d'étage

nom Nom ou adresse d'un sommet. Cela peut correspondre à un couple ‹adresse IP, port›, à un IOR CORBA, etc. L'important est d'avoir la possibilité de contacter le sommet grâce à son `nom` et que chaque `nom` soit unique.

frères Liste triée de listes des représentants des nœuds frères pour chaque étage. La première liste correspondant aux feuilles contenues par le nœud du premier étage, et la dernière liste correspond aux représentants des fils de la racine pour le sommet courant. Chaque nœud est composé de nœuds frères, qui sont les fils du nœud. Les frères sont triés par ordre chronologique de leur arrivée. Chaque frère correspond au nom d'un représentant du nœud frère.

pred Liste triée contenant une liste de prédécesseurs par étage du DST. La $n^{ème}$ liste contient le nom de tous les sommets qui utilisent le sommet courant comme représentant du nœud de l'étage $n+1$ auquel il appartient.

moi Cette variable fait référence au sommet courant.

hauteur La hauteur courante du DST.

Il existe aussi deux constantes qui sont :

a **:** Nombre minimum de frères pouvant être contenus dans un nœud à l'exception du nœud racine.

b : Nombre maximum de frères pouvant être contenus dans un nœud.

Nous définissons deux fonctions de communication : `send_msg_sync` et `send_msg_async` qui permettent respectivement l'envoi synchrone et asynchrone de messages. Ces fonctions permettent l'appel de fonctions à distance. Les paramètres des fonctions sont le nom de l'émetteur, le nom du récepteur, la fonction appelée avec la liste de ses arguments.

On définit également une fonction `diffuser` qui comprend 4 arguments : le nom de l'initiateur de la diffusion, le numéro de l'étage (`1...moi.hauteur`) d'où part cette diffusion (la *racine* de la diffusion), le nom de la fonction diffusée et ses arguments. Cette fonction permet de diffuser un message à l'ensemble des feuilles accessibles depuis la racine de la diffusion.

A.2 Fonctions gérant la construction d'un DST

A.2.1 Fonction `init` d'initialisation d'un sommet

La fonction `init` est activée à l'initialisation d'un sommet. L'initialisation d'un sommet consiste à créer un DST ne contenant qu'un seul sommet, lui-même. Si le nœud veut se rattacher à un autre DST, il devra utiliser la fonction `joindre`.

```
1   def init (nom):
2     moi.pred = [[nom]]
3     moi.frères = [[nom]]
4     moi.nom = nom
5     moi.hauteur = 1
```

Lors de l'initialisation, le sommet a son propre DST d'un étage contenant une seule feuille. Ainsi, `frères` n'est constitué que d'un unique étage contenant une feuille (ligne 3). Le sommet sert de représentant à un seul sommet, lui-même (ligne 2).

A.2.2 Fonction `joindre`

La fonction `joindre` permet à un sommet isolé de se rattacher à un DST. Pour cela, il doit connaître un membre du DST que nous appelons `contact`. À la fin de l'appel, le sommet fait partie du DST.

```
1   def joindre (contact):
2     (moi.frères[][],contact) = send_msg_sync(moi.nom, contact,
3                                              demander_connexion, moi.nom)
```

La fonction demande au sommet `contact` l'autorisation d'entrer dans le DST. La manière dont le sommet connaît ce contact n'est pas prise en compte par l'algorithme. [1] Le sommet contacté retourne la liste `frères` du sommet qui sera son nouveau contact (lignes 2–3). Ce nouveau `contact` a préparé une place pour le sommet et attend son arrivée qui doit être imminente. Si le sommet prend trop de temps à arriver, le contact considérera que le sommet est mort et annulera sa demande d'entrée dans le DST.

```
4     moi.pred[0] = moi.frères[0][:]
5     pour tout etage dans 1...moi.hauteur-1:
6       moi.pred[etage].ajouter([])
```

L'ensemble des fils d'un nœud du premier étage forme un graphe complet. Ainsi, tous les fils du nœud du premier étage pointent sur notre sommet. La fonction copie donc la liste des frères du premier étage dans la liste des prédécesseurs (ligne 4). Comme notre sommet n'est pas encore intégré à notre DST, aucun sommet ne peut utiliser le sommet comme représentant d'un nœud. Ainsi, la liste des prédécesseurs de tous les étages sont vides à l'exception du premier (lignes 5–6).

```
7     pour tout étage dans 1...moi.hauteur-1:
8       nouv_noeuds = []
9       pour tout f dans moi.frères[étage]:
10        si f == contact:
11          nouv_noeuds.ajouter(moi.nom)
12          ajouter_predecesseur(moi, étage, moi.nom)
13        sinon:
14          nouveau_rep = send_msg_sync(moi.nom, f,
15                                                  demander_nouveau_rep,
16                                                  (étage))
17          nouv_noeuds.ajouter(nouveau_rep)
18          send_msg_async(moi.nom, nouveau_rep,
19                                      ajouter_predecesseur,
20                                      (étage, moi.nom))
21      moi.frères[étage] = nouv_noeuds
```

Le fait qu'un sommet possède les mêmes représentants que `contact` (lignes 2–3) est incompatible avec la notion de distribution de la charge entre les sommets car tous les sommets utiliseraient les mêmes représentants. Nous cherchons donc d'autres représentants équivalents de manière à distribuer la charge entre tous les représentants possibles.

Ainsi, nous modifions tout les représentants des nœuds (ligne 9) de chaque étage à l'exception du premier (ligne 7) qui est toujours un graphe complet de sommets.

D'après la définition du DST, chaque sommet doit être le représentant des nœuds dont il fait partie. Ainsi, le sommet `contact` s'utilise lui-même comme représentant des nœuds auxquels il appartient. Notre sommet contient une copie de la variable `frères` de `contact`

1. Dans la simulation, le contact est choisi aléatoirement parmi les sommets plus anciens que le nouvel arrivant.

(lignes 2–3). Il utilise donc à cet instant `contact` comme représentant de tous les nœuds auxquels il appartient car le sommet `contact` et le sommet courant font partie des mêmes nœuds. Comme notre sommet doit être le représentant des nœuds auxquels il appartient, il suffit de remplacer les références sur `contact` par des références sur le sommet (lignes 10–11).

Pour les autres représentants, le sommet leur demande le nom d'un représentant qui pourrait être utilisé pour représenter son nœud de l'étage `étage` (ligne 14). C'est le représentant qui s'occupe de la répartition de la charge entre tous les représentants possibles. Pour finir, nous prévenons les représentants choisis afin qu'ils puissent mettre le nom du sommet dans leur liste de prédécesseurs (lignes 12–18).

A.2.3 Fonction `demander_nouveau_rep`

La fonction `demander_nouveau_rep` permet d'obtenir le nom d'un nouveau représentant d'un nœud, pour un étage donné.

```
1   def demander_nouveau_rep(étage):
2     nouveau_rep = moi.nom
3     charge = demander_nb_prédécesseurs(moi, étage)
4     pour tout f dans moi.frères[0]:
5       si f == moi:
6         charge_de_f = demander_nb_prédécesseurs(étage)
7       sinon
8         charge_de_f = send_msg_sync(moi.nom, f,
9                                                 demander_nb_prédécesseurs,
10                                                (étage))
11      si charge_de_f < charge:
12        charge = charge_de_f
13        nouveau_rep = f
14    retourner nouveau_rep
```

Toutes les feuilles d'un nœud du premier étage sont capables de représenter les mêmes nœuds. Cette fonction retourne un fils du nœud du premier étage qui remplacera le sommet courant en tant que représentant. Afin d'équilibrer la charge entre les divers représentants, l'algorithme recherche le sommet ayant le moins de prédécesseurs pour un étage donné (lignes 4–8) pour qu'il soit utilisé comme représentant.

Il est possible d'utiliser d'autres heuristiques de choix de représentant afin d'optimiser d'autres paramètres comme la distance en termes de liens réseaux ou la résistance aux pannes.

A.2.4 Fonction demander_connexion

La fonction `demander_connexion` a pour rôle de préparer l'arrivée d'un nouveau sommet dans le DST. Pour cela, elle fait en sorte qu'il y ait une place disponible dans le nœud en le divisant si nécessaire.

Afin d'avoir une cohésion dans le nœud, il est nécessaire d'avoir un leader qui gère les entrées et les sorties des éléments du nœud. Afin de simplifier le choix du leader, nous considérons que c'est le plus ancien sommet du nœud qui se charge de cette tâche.

```
1   def demander_connexion(nouv_sommet):
2     si moi.nom != moi.frères[0][0]:
3       retourner send_msg_sync(moi.nom, moi.frères[0][0],
4                               demander_connexion, (nouv_sommet))
```

Si le sommet devant introduire le nouveau sommet n'est pas le leader (ligne 2), alors l'action d'accepter un nouveau membre est transférée au leader du nœud du premier étage (lignes 3–4).

```
5     n = 0
6     tant que n < moi.hauteur et \
7              taille(moi.frères[n]) ==  b:
8       n = n + 1
9     si n > 0:
10      pour tout étage dans n...1:
11        moi.demander_scission(étage)
```

Afin d'accepter un nouveau membre, il faut qu'il y ait de la place dans le nœud. Si le nœud du premier étage est plein, il faut alors le diviser. Mais si le nœud du deuxième étage contenant le nœud du premier étage est déjà plein, il faudra aussi le diviser afin de laisser de la place au nouveau nœud créé par la division du nœud du premier étage. Et ainsi de suite jusqu'à ce que nous ayons atteint la racine du DST (ligne 6) ou qu'il y ait assez de place pour accepter la division d'un nœud de niveau inférieur (ligne 7).

A ce niveau, l'algorithme sait combien de nœuds vont devoir être divisés, c'est à dire `n` (ligne 8). Ensuite, l'algorithme demande la scission de ces nœuds par ordre d'étage décroissant (lignes 9–11), ce qui permet à chaque nœud d'avoir la place nécessaire à sa division.

```
12    pour tout f dans moi.frères[0][:]:
13      si f == moi:
14        nouveau_frère_reçu(nouv_sommet)
15      sinon
16        send_msg_async(moi.nom, f, nouveau_frère_reçu, (nouv_sommet))
```

Finalement, la fonction indique à tous les fils du nœud du premier étage de prendre en compte l'arrivée d'un nouveau sommet (lignes 12–16). La commande `moi.frères[0][:]` indique que la fonction travaille sur une copie de `moi.frères[0]` plutôt que sur `moi.frères[0]`

directement. Cela est nécessaire car son contenu va être modifié par la boucle en y ajoutant le nouveau fils du nœud du premier étage.

A.2.5 Fonction `nouveau_frère_reçu`

La fonction `nouveau_frère_reçu` gère la prise en compte de l'arrivée d'un nouveau sommet dans le nœud du premier étage.

```
1   def nouveau_frère_reçu(nouv_sommet):
2     moi.frères[0].ajouter(nouv_sommet)
3     moi.pred[0].ajouter(nouv_sommet)
```

Comme le sommet est implicitement le dernier arrivé dans le nœud, il est ajouté à la fin de la liste des fils du nœud du premier étage (ligne 2). Comme le nouveau sommet va lui aussi pointer sur le sommet courant, il est ajouté à la liste des prédécesseurs du premier étage (ligne 3).

A.2.6 Fonction `demander_scission`

La fonction `demander_scission` permet d'envoyer une demande de scission d'un nœud, ainsi que la scission de ses pères si nécessaire. La fonction gère l'envoi de la commande au leader, responsable de la division, ainsi que l'envoi de l'ordre à l'ensemble des sommets concernés.

```
1   def demander_scission(n):
2     étage = n-1
3     si (moi.frères[étage][0] != moi.nom):
4
5       // transfert au leader
6       send_msg_sync(moi.nom, moi.frères[étage][0],
7                               demander_scission, (n))
8     sinon
9       si n == moi.hauteur:
10        diffuser(moi, moi.hauteur-1, ajouter_étage, ())
11      fsi
12      diffuser(moi, étage, scission, (étage) )
13    fsi
14  fin
```

Si le sommet n'est pas le leader du nœud (ligne 3), alors la fonction le décharge de la gestion de la division à un sommet plus proche du leader (lignes 6–7) jusqu'à ce que le leader soit atteint.

On constate que les sommets accessibles par une diffusion depuis un étage `n` sont toujours inclus dans ceux accessibles depuis un étage `n+1`. Il n'y a donc pas de risque de changer

de groupe pendant ces transferts au leader.

Si la racine doit être divisée (ligne 9), l'algorithme ajoute alors un nouvel étage (ligne 10) car un DST ne peut avoir qu'une racine. Pour cela, il informe tous les sommets du DST qu'un nouvel étage a été ajouté. Cette simple information suffit pour ajouter un niveau au DST.

A cet instant, le leader peut envoyer l'ordre de division à tous les sommets membres du nœud qui sera divisé (ligne 12).

A.2.7 Fonction `ajouter_étage`

Afin d'ajouter un nouvel étage, il suffit que la fonction suivante soit appelée par tous les sommets du DST.

```
1   def ajouter_étage():
2     moi.hauteur++
3     moi.frères.ajouter([moi.nom])
4     moi.pred.ajouter([moi.nom])
```

Pour un sommet, ajouter un étage signifie ajouter un élément dans sa liste de nœud. Le nouveau nœud ajouté, qui est le nœud de plus haut niveau, devient alors la nouvelle racine. C'est le sommet courant qui est choisi comme représentant local de ce nouveau nœud racine (lignes 2–3) car tous les sommets peuvent représenter la racine.

A.2.8 Fonction `connecter_les_groupes_scindés`

Cette fonction permet de prévenir un nœud qu'un de ses fils de l'étage `étage` s'est divisé en deux et qu'il doit en tant que père avoir un lien sur ces deux fils.

```
1   def connecter_les_groupes_scindés(étage, init_idx, init_rep,
2                                     nouv_idx, nouv_rep)
3     si nouv_idx < taille(moi.frères[étage]):
4       retourner
```

`étage` indique l'étage qui possède un nouveau fils.

`init_idx` et `nouv_idx` sont les positions du fils originel et du nouveau fils, dans la table de routage du nœud appelant (le nœud qui s'est scindé, voir fonction `scission`), à l'étage `étage`. Donc `nouv_idx` désigne en fait la taille de `moi.frères[étage]` du nœud appelant, c'est à dire le prochain emplacement libre pour un nouveau frère à cet étage.

`init_rep` et `nouv_rep` sont respectivement un représentant du nœud originel et un représentant du nouveau nœud.

Il est possible que la fonction `connecter_les_groupes_scindés` soit appelée plusieurs fois sur un sommet pour une même division. Si le sommet courant a déjà accès au nouveau

nœud (ligne 3), cela signifie qu'il a déjà été informé de la division, il est alors inutile de prendre en compte ce nouvel appel (ligne 4).

```
5     si (init_idx < taille(moi.frères[étage])):
6       rep = moi.frères[étage][init_idx]
7     sinon
8       rep = init_rep
9     fsi
```

Si `init_idx` pointe à la fin de la liste de frères, alors aucun nœud n'est remplacé puisqu'on en ajoute un. (voir ligne 19)

```
10    pos = moi.frères[étage].index(moi.nom)
11    si pos == init_idx:
12      si nouv_rep != moi.nom:
13        init_rep = moi.nom
```

Le sommet doit toujours être sa propre référence sur les nœuds auxquels il appartient. Si le sommet fait partie du nœud restant (ligne 11), alors il doit être utilisé comme référence de son nouveau nœud plutôt que d'utiliser la référence qui a été fournie par l'appel de la fonction.

L'algorithme est écrit de telle manière que si le sommet fait partie du nouveau nœud, alors la fonction `connecter_les_groupes_scindés` est uniquement appelée par le sommet courant qui passe son propre nom en paramètre `nouv_rep`. Si `nouv_rep` n'est pas le nom du sommet courant, alors le sommet courant fait partie du nœud originel (ligne 12) et le nom du sommet courant doit être utilisé comme référence au nœud originel (ligne 13).

```
14    si init_rep et nouv_rep sont déjà connus:
15      retourner
16
```

On ajoute ces deux lignes par précaution, pour être certain de ne pas ajouter de frères en double.

```
17    moi.frères[étage][init_idx] = init_rep
18    moi.frères[étage].ajouter(nouv_rep)
```

L'ancienne référence sur le nœud originel ayant été mémorisée (lignes 5–9), l'algorithme peut remplacer cette ancienne référence par la nouvelle (ligne 17) et il ajoute la référence du nouveau nœud (ligne 18).

```
19    si rep != init_rep:
20      si rep != moi.nom:
21        send_msg_async(moi.nom, rep, effacer_predecesseur,
22                                     (étage, moi.nom));
23      fsi
24      send_msg_async(moi.nom, init_rep, ajouter_predecesseur,
25                                        (étage, moi.nom));
```

```
26      fsi
27
28      si nouv_rep != moi.nom:
29        send_msg_async(moi.nom, nouv_rep, ajouter_predecesseur,
30                                          (étage, moi.nom));
```

Si l'ancienne référence sur le nœud originel a été remplacée (ligne 19), alors l'algorithme doit indiquer à cette ancienne référence qu'elle n'est plus utilisée (ligne 21) et à la nouvelle référence qu'elle l'est (ligne 24).

L'algorithme indique à `nouv_rep` qu'il est utilisé par le nœud courant (lignes 28–29). Cela n'est bien sûr nécessaire que si `nouv_rep` n'est pas le nœud courant.

⋆⋆⋆

Ici, l'ordre des opérations est inversé par rapport à la proposition initiale de Sylvain. (je parle des blocs 19-26 et 28-30) En effet, voici un cas de figure dans lequel l'ancien ordre pose problème :

Le nœud 677 dont voici la table de routage :

E0	42	677	387	
E1	677	467		

reçoit la tâche `connecter_les_groupes_scindés` depuis le nœud 467, avec les arguments suivants :
`étage = 1, init_rep = 556, init_idx = 1, nouv_idx = 2, nouv_rep = 467`

Au final, on doit donc obtenir la table de routage suivante :

E0	42	677	387	
E1	677	556	467	

A l'issue du bloc 5-9, `rep` vaut 467. La ligne 21 ôte donc 677 des prédécesseurs de 467 et la ligne 29 l'y ajoute. Comme on peut donc le voir, si on laisse l'ordre des opérations tel qu'il était, 677 ne figurera plus dans les prédécesseurs de 467 alors qu'il continue à pointer dessus.

A.2.9 Fonction `scission`

Lorsqu'un nœud se divise, tous les sommets appartenant à ce nœud exécutent la fonction `scission`. La moitié des sommets décideront en fonction de l'algorithme de quitter le nœud pour en former un nouveau. Ceci scinde le nœud en deux nœuds de tailles équivalentes.

```
1   def scission(étage):
2     pos = moi.frères[étage].index(moi.nom)
3     reste = (pos+1) % 2
```

Le nœud de l'étage `étage` va être divisé. Nous avons choisi comme heuristique de division que les frères ayant une place paire dans la liste `frères[étage]` restent et que les autres

sortent du nœud pour en former un autre (lignes 2–3).

```
4     si non reste:
5       l_autre_noeud = moi.frères[étage][pos - 1]
6     sinon si pos == taille(moi.frères[étage]) - 1:
7       l_autre_noeud = moi.frères[étage][pos - 1]
8     sinon:
9       l_autre_noeud = moi.frères[étage][pos + 1]
```

La variable `l_autre_noeud` contient le nom d'un sommet qui aura un statut privilégié avec le sommet courant lors de la division. Ce sommet doit être un sommet qui appartiendra à l'autre nœud. À chaque fois que le sommet courant aura besoin d'un représentant de l'autre nœud, il utilisera `l_autre_noeud` plutôt que de rechercher à chaque fois un autre représentant. L'attribution de `l_autre_noeud` (lignes 5–9) est faite de façon à répartir la charge entre les membres du nœud.

```
10    si reste:
11      nouv_grp = []
12      pour tout i dans 0...taille(moi.frères[étage])-1:
13      si i%2 = 0:
14        nouv_grp.ajout(moi.frères[étage][i])
15      sinon:
16        send_msg_async(moi.nom, moi.frères[étage][i],
17                       effacer_prédécesseur, (étage, moi.nom))
18      moi.frères[étage] = nouv_grp
```

La suite de la fonction forme les deux nœuds. Les actions entreprises pour créer ces nœuds ne sont pas identiques selon que le sommet reste dans le nœud ou part rejoindre le nouveau.

S'il reste dans le nœud, c'est qu'il est sur une place paire. Lorsqu'une division est réalisée, seuls les fils du nœud ayant une place paire restent dans le nœud (lignes 13-14). Les autres membres sont rejetés et le lien entre le sommet et ce membre est coupé (lignes 16–17).

```
19      init_rep=moi.nom
20      nouv_rep=l_autre_noeud
```

Il faut choisir qui représentera le nœud initial et qui représentera le nouveau nœud issu de la division. Le sommet courant reste le représentant du nœud initial (ligne 19) et `l_autre_noeud` devient le représentant du nouveau nœud (ligne 20)

```
21    sinon:
22      nouv_grp = []
23      pour tout i dans 0..taille(moi.frères[étage])-1:
24        si i%2 = 1:
25          nouv_grp.ajout(moi.frères[étage][i])
26        sinon:
27          send_msg_async(moi.nom, moi.frères[étage][i],
28                         effacer_prédécesseur, (étage, moi.nom))
```

```
29        moi.frères[étage] = nouv_grp
```

Si le sommet doit quitter le nœud pour en former un autre, c'est qu'il est à une place impaire. Le nouveau nœud est formé de tous les fils situés à une place impaire (lignes 24–25). Les liens avec les membres qui resteront dans le nœud originel sont coupés (lignes 27–28) car ils ne font plus partie du nœud de notre sommet.

```
30        init_rep=l_autre_noeud
31        nouv_rep=moi.nom
```

Ces deux lignes choisissent à nouveau qui deviendra le représentant du nœud originel et qui sera le représentant du nouveau nœud. `l_autre_noeud` faisant partie du nœud originel est utilisé pour représenter le nœud originel (ligne 30). Quant au nouveau nœud, il est représenté par le sommet courant (ligne 31).

```
32        init_idx = moi.frères[étage+1].index(moi.nom)
33        nouv_idx = taille(moi.frères[étage+1])
34
35        pour tout p dans moi.pred[étage+1][:]:
36          si p == moi:
37            connecter_les_groupes_scindés(étage+1, init_idx, init_rep,
38                                                   nouv_idx, nouv_rep)
39          sinon:
40            send_msg_async(moi.nom, p, connecter_les_groupes_scindés,
41                           (étage+1, init_idx, init_rep, nouv_idx, nouv_rep))
```

L'ensemble des prédécesseurs de l'étage supérieur à celui qui s'est scindé (`moi.pred[étage+1]`) forme l'ensemble des sommets du nœud père. Pour prévenir ce nœud père qu'il vient d'avoir un nouveau fils, il faut donc tous les contacter. Il s'agit pour chacun d'eux d'acquérir un nouveau frère (le représentant du nouveau nœud (ligne 33)) tout en s'assurant que l'ordre d'arrivée des nœuds est toujours maintenu. (ligne 32)

C'est ce qui est finalement fait ici.

Annexe B

Gestion des retraits de sommets du DST

Sommaire

Cette partie est nouvelle. Il s'agit d'une proposition d'algorithmes permettant de gérer les départs de sommets du DST, y compris les fusions éventuellement rendues nécessaires par ces départs. L'équilibrage de charge n'est pas réalisé, cela reste à faire.

Je n'ai étudié que le cas de départs "volontaires". C'est à dire que les sommets souhaitant quitter le DST doivent le signaler en exécutant la fonction `quitte`. En cas de panne d'un sommet – détectée par un moyen restant à définir – il ne pourrait pas exécuter lui-même cette fonction puisqu'il serait déjà arrêté. Il faudrait alors étudier la possibilité de la faire exécuter par l'un de ses frères.

Ces fonctions également écrites en C ont pu être testées et validées sous Simgrid. Toutefois et contrairement au travail réalisé sur l'ajout de sommets, j'estime que les fonctions présentées ici n'ont pas été soumises à suffisamment de cas de figure pour pouvoir dire avec certitude qu'elles sont validées, bien que les différents tests se soient bien déroulés. De plus, toujours par manque de temps, les problèmes de synchronisation n'ont pas été étudiés ici.

⋆⋆⋆

Pour décrire ces algorithmes, je propose un exemple de DST – voir figure B.1 – qui servira de support tout au long de ces explications.

Dans cet exemple, le départ du sommet `8` laisse un sommet "orphelin" : le `21`.

Étage 0 : Le groupe `0B` ayant de la place pour accueillir `21`, il va pouvoir fusionner avec `0A` pour donner le groupe `0AB`.

Étage 1 : En conséquence, le groupe `1A` n'a plus qu'un seul membre et va devoir fusionner avec `1B` pour donner le groupe `1AB`.

Étage 2 : À son tour, le groupe `2B` n'a plus qu'un membre et doit fusionner avec `2A` pour obtenir le groupe `2AB`

Ici, le DST ne perd pas de niveau.

B.1 Fonctions gérant le départ d'un sommet

B.1.1 Fonction `quitte`

Cette fonction est appelée par un sommet qui souhaite quitter le DST.

```
1   def quitte()
2
3     // informe mes prédécesseurs de mon départ
4     pour tout étage dans 0 ... moi.hauteur-1:
5       pour tout pred dans moi.pred[étage]:
6         si (pred != moi):
7           si (étage == 0):
8             send_msg_async(moi, pred, efface_frère, (étage, moi.nom))
9           sinon:
10            nouveau_rep = choisir_frère_aléatoirement
11            send_msg_async(moi, pred, remplace_frère, (étage, nouv_rep))
12           fsi
13        fsi
14      fpour
15
16      // informe mes frères de mon départ
17      pour chaque frère dans moi.frère[étage]:
18        si (frère != moi):
19          send_msg_async(moi, frère, efface_pred, (étage, moi.nom))
20        fsi
21      fpour
22    fpour
```

Puisque le sommet courant (`moi`) s'en va, il ne peut plus être utilisé comme représentant du groupe auquel il appartient. Il va donc être remplacé par un autre de ses frères de

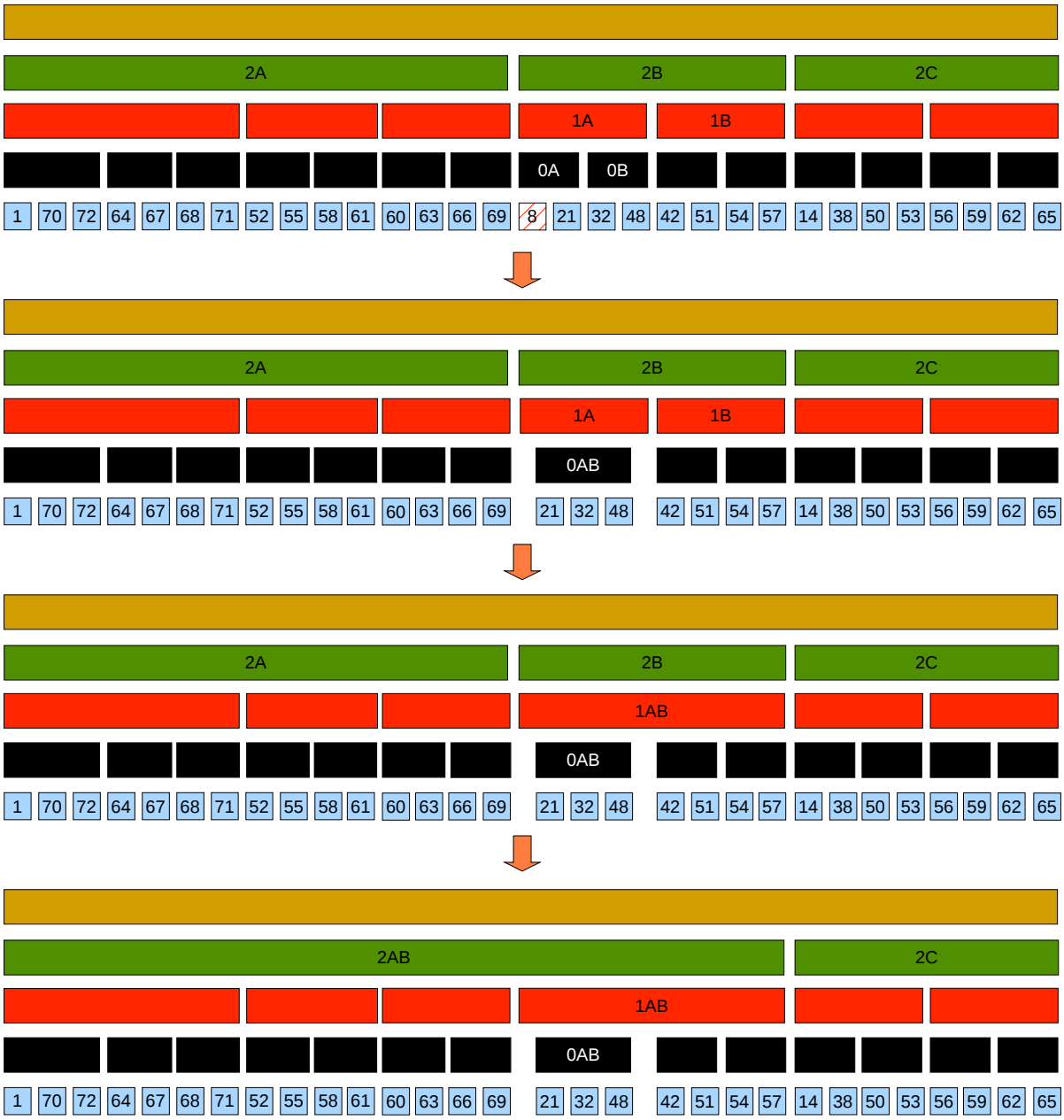

FIGURE B.1 – Exemple de fusion : le sommet 8 quitte le DST

niveau 0 partout où cela est nécessaire. Dans l'exemple, `8` va donc être remplacé par `21`.

`choisir_frère_aléatoirement` est une fonction qui retourne n'importe quel frère de `moi` de niveau 0, autre que `moi` bien sûr.

`remplace_frère` est une fonction détaillée plus loin qui permet de remplacer un frère par un autre, à un étage donné.

```
23      // gère les éventuelles fusions
24      si (taille(moi.frères[0]) <= a):
25
26        prend le premier idx tel que moi.frères[0][idx] != moi
27        send_msg_sync(moi,
28                      moi.frères[0][idx],
29                      demande_fusion,
30                      ())
31      fsi
32    fin
```

Si les opérations précédentes laissent des sommets *orphelins*[1], des fusions vont être nécessaires. On demande ici à l'un des frères du sommet sortant de démarrer ce processus. Dans l'exemple, c'est donc le sommet `21` qui va lancer la fonction `demande_fusion`.

B.1.2 Fonction `remplace_frère`

Cette fonction est chargée de remplacer un frère par un autre à l'étage `étage`. `init_idx` désigne l'emplacement où ce remplacement doit se produire et `nouv_sommet` est le remplaçant.

```
1   def remplace_frère(étage, init_idx, nouv_sommet)
2
3     // rien à faire
4     si (moi.frères[étage][init_idx] == nouv_sommet) retourner
```

Si l'emplacement désigné par `init_idx` contient déjà `nouv_sommet`, c'est que la fonction a déjà été exécutée et il ne faut donc rien faire.

```
5     si (init_idx < taille(moi.frères[étage])):
6
7       // remplace frère
8       ancien_sommet = moi.frères[étage][init_idx]
9       moi.frères[étage][init_idx] = nouv_sommet
10    sinon:
11
```

1. c'est à dire des sommets en nombre insuffisant pour qu'ils se regroupent – donc inférieur à la borne `a` du DST

```
12        // ajoute nouveau frère
13        ajoute_frère(étage, nouv_sommet);
14      fsi
```

Si `init_idx` pointe au-delà de la liste courante, alors on ne procède pas à un remplacement, mais à un ajout.

À noter qu'on mémorise l'ancien sommet (ligne 8) parce qu'on en a besoin pour la mise à jour des prédécesseurs (voir ligne 19).

```
15      // efface ancien prédécesseur
16      si ((ancien_sommet != moi) &&
17          (init_idx < taille(moi.frères[étage]))):
18            send_msg_async(moi,
19                           ancien_sommet.nom,
20                           efface_pred,
21                           (étage,
22                            moi.nom))
23      fsi
24
25      // ajoute nouveau predecesseur
26      si (nouv_sommet != moi):
27            send_msg_async(moi,
28                           nouv_sommet.nom,
29                           ajoute_pred,
30                           (étage,
31                            moi.nom))
32      fsi
33    fin
```

Les prédécesseurs sont mis à jour en conséquence.

B.1.3 Fonction `demande_fusion`

Cette fonction est chargée de gérer les fusions suite au départ d'un sommet. (voir fonction `quitte` plus haut) Elle parcourt le DST de bas en haut, tant que des fusions sont requises (ligne 4), comme on le voit sur l'exemple.

```
1   def demande_fusion()
2     étage = 0
3     recp = -2
4     tant que ((taille(moi.frères[étage]) <= a) && (étage < moi.hauteur - 1)):
5
6       /* fusion_ou_transfert(étage) retourne l'indice du représentant
7          à l'étage étage + 1 du groupe à rejoindre.
8          Retourne -1 si aucun groupe ne peut accueillir le groupe courant */
```

```
 9
10      pos_contact = moi.fusion_ou_transfert(étage)
11      si (pos_contact > - 1):
12        pos_moi = moi.index(étage + 1, moi.nom)
13
14        // demande à contact d'effectuer une première fusion
15        send_msg_sync(moi,
16                      moi.frères[étage + 1][pos_contact],
17                      fusion,
18                        (moi.frères[étage],
19                         taille(moi.frères[étage]),
20                         étage,
21                         pos_moi,
22                         pos_contact))
23
24        // demande à contact de lancer la tâche diffuse_fusion
25        send_msg_sync(moi,
26                      moi.frères[étage + 1][pos_contact],
27                      diffuse_fusion,
28                        (étage,
29                         pos_moi,
30                         pos_contact))
31      sinon
32        // transfert plutôt que fusion
33      fsi
```

`fusion_ou_transfert(étage)` retourne l'indice d'un représentant à l'étage `étage + 1` du groupe à rejoindre. (Retourne -1 si aucun groupe ne peut accueillir le groupe courant, auquel cas il ne faut pas faire une fusion, mais un transfert. (voir plus loin))

Dans notre exemple B.1, cette fonction retourne les indices des sommets 32, 42 et 1 pour les fusions des étages 0, 1 et 2, respectivement. Ce seront les sommets *contacts*.

Après la ligne 12, `pos_moi` représente donc le groupe courant et `pos_contact`, le groupe à rejoindre.

Si une fusion est possible (ligne 11), on commence par demander à `contact` d'effectuer une première fusion (ligne 15). Sa table de routage étant alors correcte, on peut demander à ce sommet de lancer la tâche `diffuse_fusion` (ligne 25) qui va permettre à tous les sommets concernés de prendre cette fusion en compte.

On peut remarquer qu'en procédant de la sorte, le sommet `contact` exécute au moins deux fois la fonction `fusion`. Ce n'est pas gênant dans la mesure où la deuxième exécution ne fera rien – puisque sa table de routage comportera déjà les nouveaux sommets – mais il peut être utile d'empêcher la deuxième exécution pour des raisons de performance.

Il faut aussi noter qu'on utilise à chaque fois des envois synchrones (lignes 15, 25) parce que l'ordre des étapes est important : il faut que la première tâche de fusion soit terminée avant de pouvoir lancer `diffuse_fusion` puisque celle-ci utilise la table de routage à jour de `contact`. De même, `diffuse_fusion` doit être terminée avant de passer à la suite : par exemple, le test (ligne 4) et la lecture de la taille de la racine (ligne 52) doivent avoir lieu sur des données à jour.

Dans le cas où aucun groupe n'a suffisamment de place pour accueillir les nouveaux venus, on a recours à la fonction `transfert` qui travaille dans le sens contraire de `fusion` : cette fois, c'est le groupe des sommets orphelins qui va accueillir d'autres sommets des groupes voisins de telle sorte que les bornes `a` et `b` restent respectées pour tout le monde. Par manque de temps, je n'ai pas écrit d'algorithme pour cette fonction. (voir remarque page 107)

```
34      /* recherche le noeud qui va diffuser la tâche de mise
35         à jour de la racine sur la partie du DST non concernée
36         par les fusions */
37
38      si (étage == moi.hauteur - 2):
39
40        recp = -1
41        si (pos_moi <= pos_contact):
42
43          recp = moi.frères[étage + 1][pos_moi - 1]
44        sinon:
45
46          recp = moi_frères[étage + 1][pos_contact + 1]
47        fsi
48      fsi
49      étage ++
50    ftq
```

Une fois les fusions réalisées et si elles ont atteint la racine (ligne 38), celle-ci doit comporter un membre de moins (à moins qu'elle ne s'en aille – voir plus loin). Les opérations de `fusion` réalisent cela, mais seulement dans les sommets impactés par ces opérations. Il faut donc mettre à jour la racine dans les autres sommets en diffusant la fonction chargée de cela dans la bonne partie du DST. Ici (lignes 38–48), on détermine le sommet `recp` d'où devra partir cette diffusion qui sera réalisée plus loin. (ligne 59)

Si les fusions n'atteignent pas la racine, `recp` restera égal à -2 (ligne 3).

À la fin de l'exemple B.1, tous les membres du groupe 2AB sont à jour puisqu'ils sont concernés par les opérations de fusion. Mais la partie droite – les membres du groupe 2C – n'a pas eu connaissance que des fusions se produisaient. C'est donc dans cette partie que doit être diffusée la tâche `nettoie_etage_sup` et `recp` sera ici le sommet 14.

```
51    choisir i tel que moi.frères[0][i] != moi
```

```
52     taille_dernier_étage = send_msg_sync(moi,
53                                          moi.frères[0][i],
54                                          lit_taille,
55                                           (moi.hauteur-1))
56
57     si (taille_dernier_étage > 1):
58       si (recp > -1):
59         send_msg_sync(moi, recp, diffuse, (moi,
60                                            moi.hauteur - 2,
61                                            nettoie_etage_sup,
62                                             (moi.hauteur - 2,
63                                              pos_moi,
64                                              pos_contact)))
65       fsi
66     fsi
```

Si la racine ne s'en va pas (ligne 57), on demande donc au sommet `recp` de diffuser la fonction de mise à jour de cette racine.

```
67     si (taille_dernier_étage == 1):
68       moi.diffuser(moi, moi.hauteur - 1, supprimer_racine, (moi.hauteur))
69     fsi
70   fin
```

Si la racine doit s'en aller (ligne 67), alors on diffuse une fonction de suppression de la racine sur la totalité du DST.

B.1.4 Fonction `fusion_ou_transfert`

Cette fonction sert à déterminer si une fusion est possible ou s'il faut plutôt réaliser un transfert. La fusion sera possible si l'un des groupes voisins de celui qui a perdu un membre possède suffisamment de place pour accueillir les membres restants. Le transfert consiste au contraire à remplir le groupe devenu trop petit avec des membres pris aux groupes voisins.

```
1   def fusion_ou_transfert(étage)
2     idx_frère = 0
3     fusion = 0
4
5     tant que ((fusion == 0) && (idx_frère < taille(moi.frères[étage + 1]))):
6       si (moi.frères[étage + 1][idx_frère] != moi):
7
8         taille_frères = send_msg_sync(moi,
9                                       moi.frères[étage + 1][idx_frère].nom,
```

```
10                                        taille,
11                                        (étage))
12          si (taille_frères <= b - taille(moi.frères[étage])):
13
14            fusion = 1
15          fsi
16        fsi
17        idx_frère ++
18      ftq
```

Chacun des frères de l'étage parent de `étage` représentant un groupe susceptible d'accueillir les sommets restants, on les interroge pour connaître leur taille. Dès qu'on en a trouvé un avec une taille suffisamment petite (lignes 12-14), on arrête, la fusion étant alors possible.

```
19    si (fusion == 1):
20        retourne idx_frère - 1
21      else
22        retourne -1
23    fin
```

Dans ce cas, on retourne l'indice du représentant de ce groupe, sinon on retourne -1.

B.1.5 Fonction `fusion`

Cette fonction se charge de fusionner deux groupes. `pos_moi` et `pos_contact` représentent les groupes "arrivant" et "accueillant", respectivement. La fusion est réalisée à l'étage `étage`. `liste_frères` contient la liste des frères à accueillir et `nb_frères` est la taille de cette liste.

```
1   def fusion(liste_frères, nb_frères, étage, pos_moi, pos_contact)
2
3     loc_pos_moi = moi.index(étage + 1, moi.nom)
4     contact_nom = moi.frères[étage + 1][pos_contact].nom
5
6     si (moi.index(moi.hauteur - 1, moi.nom) != 0 &&
7         ((loc_pos_moi > pos_contact) ||
8          ((loc_pos_moi == pos_contact) &&
9           ((moi.nom == contact_nom) ||
10           (index(étage, moi.nom) > index(étage, contact_nom))))))
11
12      droite = 1
13    sinon:
14      droite = 0
15    fsi
```

Ici, il s'agit d'inclure les nouveaux frères dans le sommet courant. La liste de frères fournie en argument est celle du premier sommet qui a réalisé une fusion – le sommet `contact` (voir ligne 15 de l'algo `demande_fusion`) – et il ne faut donc pas forcément prendre la totalité de ces frères sous peine d'avoir des doublons. (deux représentants d'un même groupe)

Voici donc l'idée :

- Si le sommet courant `moi` est à droite du contact, on prend la partie gauche de la liste de frères et on l'insère à gauche de la liste courante.
- Si le sommet courant est à gauche du contact, on prend la partie droite de la liste de frères et on l'ajoute à droite de la liste courante

En procédant de la sorte, l'ordre chronologique des frères est maintenu. Cette première partie de l'algo consiste donc à déterminer où se situe le sommet courant par rapport à `contact`.

On rappelle que `loc_pos_moi` [2] et `pos_contact` sont les indices des sommets courant et contact (respectivement) à l'étage supérieur. Si donc ces deux indices sont égaux, il faut regarder ce qui se passe à l'étage en dessous pour déterminer l'ordre de ces frères. C'est ce qui est fait aux lignes 8 à 10.

Au début (ligne 6), on s'assure de ne pas être sur le plus vieux sommet du DST. Si c'est le cas, nous sommes donc sur le sommet le plus à gauche et on ne peut qu'ajouter les nouveaux frères en fin de liste. (voir aussi ligne 55)

```
16      si (étage > 0):
17        si (droite == 1):
18
19            // prend la partie gauche de la liste de frères
20            si (nb_frères >= taille(moi.frères[étage])):
21              loc_nb_frères = nb_frères - taille(moi.frères[étage]);
22            sinon:
23              loc_nb_frères = nb_frères;
24            fsi
25
26            si (loc_nb_frères > 0):
27              pour i de 0 à loc_nb_frères - 1:
28                loc_liste_frères[i] = liste_frères[i]
29              fpour
30            fsi
31        sinon:
32
33          // prend la partie droite de la liste de frères
```

2. `pos_moi` est l'indice à l'étage parent de l'étage courant du sommet qui exécute `demande_fusion`, c'est à dire le point de départ des fusions. (voir `demande_fusion` (ligne 12)) On en a besoin plus loin (ligne 78)

```
34          si (nb_frères < taille(moi.frères[étage]):
35            loc_nb_frères = nb_frères;
36          sinon:
37            loc_nb_frères = nb_frères - taille(moi.frères[étage]);
38          fsi
39
40          si (loc_nb_frères > 0):
41            pour i de (nb_frères - loc_nb_frères) à nb_frères - 1:
42              loc_liste_frères[i - (nb_frères - loc_nb_frères)] = liste_frères[i]
43            fpour
44          fsi
45        fsi
46      sinon:
47
48        // étage 0
49        loc_nb_frères = nb_frères;
50        loc_liste_frères = liste_frères;
51      fsi
```

Dans cette partie, on constitue la liste de frères `loc_liste_frères` qui sera effectivement ajoutée au sommet courant.

```
52      si (loc_nb_frères > 0):
53
54        si (droite == 1 &&
55            moi.index(moi.hauteur - 1, moi.nom) > 0):
56
57          // insère les nouveaux frères en tête de liste
58          pour (i dans loc_nb_frères - 1 ... 0):
59            insère_frère(étage, loc_liste_frères[i])
60            send_msg_async(moi,
61                           loc_liste_frères[i],
62                           ajoute_pred,
63                           (étage,
64                            moi.nom))
65          fpour
66        sinon:
67
68          // ajoute les nouveaux frères en fin de liste
69          pour (i dans 0 ... loc_nb_frères-1):
70            ajoute_frère(étage, loc_liste_frères[i])
71            send_msg_async(moi,
72                           loc_liste_frères[i],
73                           ajoute_pred,
74                           (étage,
```

```
75                              moi.nom))
76          fpour
77        fsi
```

C'est dans cette partie qu'on insère ces nouveaux frères au début ou qu'on les ajoute en fin de liste. Les prédécesseurs sont mis à jour en conséquence.

```
78        nettoie_étage_sup(étage, pos_moi, pos_contact);
79      fsi
80    fin
```

Cette fusion qui vient d'avoir lieu entraîne la perte d'un membre dans le groupe parent. La fonction `nettoie_étage_sup` se charge donc de la mise à jour de l'étage supérieur.

B.1.6 Fonction `nettoie_étage_sup`

Comme indiqué plus haut, cette fonction est chargée de supprimer le membre en trop de l'étage parent de celui qui vient de subir une fusion.

Reprenons l'exemple B.1. Voici la table de routage du sommet 32 avant l'arrivée de 21 au niveau 0 : [3]

Sommet 32 :

E0	32	48		
E1	21	32		
E2	32	42		
E3	1	32	14	

Et voici la même table après l'arrivée de 21 :

Sommet 32 :

E0	21	32	48	
E1	21	32		
E2	32	42		
E3	1	32	14	

On peut voir qu'il y a un problème puisque l'étage 1 pointe sur 2 représentants du même étage 0. Cette fonction va donc corriger le problème en supprimant 21 à l'étage 1.

```
1   def nettoie_etage_sup(étage, pos_moi, pos_contact)
2
3     recp = -1;
4     si (moi.frères[étage + 1][pos_moi].nom == moi.nom &&
```

3. On rappelle que 8 a déjà été remplacé par 21 partout où il se trouvait par la première partie de la fonction `quitte`.

```
5         pos_contact < taille(moi.frères[étage + 1])):
6
7       recp = moi.frères[étage + 1][pos_contact].nom
8     sinon:
9       si (pos_moi < taille(moi.frères[étage + 1])):
10
11        recp = moi.frères[étage + 1][pos_moi].nom
12      fsi
13    fsi
```

Dans cette partie, on cherche qui doit être supprimé. Ce sera `recp`. Si on se trouve sur `pos_moi`, il faut détruire `pos_contact` et réciproquement.

```
14    si (recp > -1):
15      moi.efface_frère(étage + 1, recp);
16      send_msg_async(moi,
17                     recp,
18                     efface_pred,
19                     (étage + 1,
20                      moi.nom))
21    fsi
22  fin
```

Ici, on efface simplement le frère désigné par `recp` en mettant bien sûr le prédécesseur concerné à jour.

B.1.7 Fonction `diffuse_fusion`

Cette fonction est chargée de diffuser la tâche de fusion à tous les sommets concernés. La fonction `demande_fusion` demande au sommet `contact` de lancer cette fonction. (voir `demande_fusion` (ligne 25))

`pos_moi` est l'indice du sommet qui exécute `demande_fusion` à l'étage parent de l'étage courant, c'est à dire le point de départ des fusions. (voir `demande_fusion (ligne 12)`) `pos_contact` est l'indice – toujours à l'étage supérieur – du représentant du groupe à rejoindre.

```
1   def diffuse_fusion(étage, pos_moi, pos_contact)
2
3     moi.diffuse(moi, étage, fusion, (moi.frères[étage],
4                                      taille(moi.frères[étage]),
5                                      étage,
6                                      pos_moi,
7                                      pos_contact))
8   fin
```

On réalise ici simplement une diffusion de la tâche de fusion à partir de l'étage courant.

B.1.8 Fonction `supprimer_racine`

Cette fonction supprime simplement le niveau le plus haut dans la table de routage courante.

```
1  def supprimer_racine(hauteur_init)
2    si (moi.hauteur == hauteur_init):
3      moi.hauteur--
4    fsi
5  fin
```

Cette fonction étant diffusée, un même sommet peut l'exécuter plusieurs fois. L'argument `hauteur_init` permet de s'assurer qu'on n'exécute qu'une fois cette fonction : si la hauteur courante est la même que la hauteur initiale, c'est que la fonction n'a pas encore été exécutée.

Annexe C

Reste à faire (code)

- Compléter la gestion de l'état `'g'` comme indiqué dans la remarque de la page 37.
- Étudier et implémenter les fonctions de transfert lors du départ d'un sommet (lorsqu'une fusion n'est pas possible).
- Étudier l'ensemble des cas de figure pouvant survenir lorsqu'une fusion ou un transfert doivent avoir lieu (que deviennent les orphelins, qui accueille qui, etc ...), puis concevoir et implémenter les fonctions permettant de gérer ces cas. Pour cela, il est possible de partir des indications données dans la publication "*The Distributed Spanning Tree : A Scalable Interconnection Topology for Efficient and Equitable Traversal*" [DNP05], section 3.2 page 6 :

  > *"Merging two groups together is a little more difficult than splitting a group into two parts. If we suppose that the first group has a elements and the second has x elements with* $a \leq x \leq b$*, three cases appear. In the first case* $a + x < 2a$*, then the two groups must be merged in one. In the second case* $a + x > b$*, then the two groups must not be merged in one. Instead, some nodes of the bigger group are transfered to the smaller group. In the third case* $2a \leq a + x \leq b$*, then the two previous actions are possible."*

- Toujours pour les retraits de sommets, je pense qu'il n'y a pas eu assez de cas de figure testés. Il faut donc poursuivre les tests pour achever de valider ces algorithmes.
- Ajouter des fonctions d'équilibrage de charge suite au départ d'un sommet.
- Comme expliqué en page 66, la fonction `attente_terminaison` présente un problème qui doit être corrigé. Pour mettre en œuvre la solution proposée, il faut gérer dynamiquement des tableaux dont on ne peut connaître la taille à l'avance. Il serait alors certainement mieux d'utiliser une autre structure comme les *dynar* (*generic dynamic array*) mis à disposition dans Simgrid. [1]
- En ayant recours à *Valgrind*[2], on peut constater des fuites mémoire lors de l'utilisation des *dynars* dans les fonctions de communication. Leurs éléments ne semblent

1. http://simgrid.gforge.inria.fr/3.6.1/doc/group__XBT__dynar.html
2. http://valgrind.org

pas libérés correctement. Ce point est à corriger si on veut pouvoir travailler sur de grandes plates-formes.

- Il me paraît utile d'inclure dans le code une fonction chargée de vérifier la cohérence du DST en cours de construction. J'ai écrit pour cela une fonction nommée `check()`, mais je n'ai pas trouvé quand ni comment la lancer. En effet, ce n'est pas parce qu'un sommet a terminé ses opérations d'intégration dans le DST que tous les sommets figurant dans sa table de routage en ont fait autant. Vu que la fonction `check()` interroge ces sommets, on pourrait obtenir des réponses erronées si on le fait trop tôt. Ce point est donc également à étudier.
- Les auteurs de Simgrid fournissent un script en Python nommé `generate.py` destiné à générer des descriptions de plates-formes en XML. Ce script se plante dès qu'on dépasse un peu plus de 4 000 nœuds. Il faut corriger ce problème pour pouvoir faire des tests à plus grande échelle.
- Il y a un projet (déjà bien avancé, semble-t-il) d'ajouter un *model-checker* dans Simgrid. Cet outil pourrait permettre de vérifier la pertinence de certaines options choisies lors de la conception de l'application. La seule API qui ne pourra pas l'utiliser sera GRAS, donc il ne devrait pas y avoir de problème avec MSG. Ce serait donc intéressant de soumettre mon programme à cet outil pour voir qu'il en ressortirait.