Optimisation de requêtes sur des données massives dans un environnement distribué

Noël Gillet

université de BORDEAUX

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ DE BORDEAUX

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

par **Noël Gillet**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Optimisation de requêtes sur des données massives dans un environnement distribué

Date de soutenance : 10 mars 2017

Devant la commission d'examen composée de :

Olivier Beaumont	Directeur de recherche	Président du jury
Nicolas Hanusse	Directeur de recherche	Directeur de thèse
Nicolas Nisse	Chargé de recherche HDR	Rapporteur
Maria Potop-Butucaru	Professeur	Rapporteur

– 2017 –

Résumé Les systèmes de stockage distribués sont massivement utilisés dans le contexte actuel des grandes masses de données. En plus de gérer le stockage de ces données, ces systèmes doivent répondre à une quantité toujours plus importante de requêtes émises par des clients distants afin d'effectuer de la fouille de données ou encore de la visualisation. Une problématique majeure dans ce contexte consiste à répartir efficacement les requêtes entre les différents nœuds qui composent ces systèmes afin de minimiser le temps de traitement des requêtes (temps maximum et en moyenne d'une requête, temps total de traitement pour toutes les requêtes...).

Dans cette thèse nous nous intéressons au problème d'allocation de requêtes dans un environnement distribué. On considère que les données sont répliquées et que les requêtes sont traitées par les nœuds stockant une copie de la donnée concernée. Dans un premier temps, des solutions algorithmiques quasi-optimales sont proposées lorsque les communications entre les différents nœuds du système se font de manière asynchrone. Le cas où certains nœuds du système peuvent être en panne est également considéré. Dans un deuxième temps, nous nous intéressons à l'impact de la réplication des données sur le traitement des requêtes. En particulier, un algorithme qui adapte la réplication des données en fonction de la demande est proposé. Cet algorithme couplé à nos algorithmes d'allocation permet de garantir une répartition des requêtes proche de l'idéal pour toute distribution de requêtes. Enfin, nous nous intéressons à l'impact de la réplication quand les requêtes arrivent en flux sur le système. Nous procédons à une évaluation expérimentale sur la base de données distribuées Apache Cassandra. Les expériences réalisées confirment l'intérêt de la réplication et de nos algorithmes d'allocation vis-à-vis des solutions présentes par défaut dans ce système.

Mots-clés Équilibrage de charge, algorithme distribué, graphe, données massives

Title Optimization of queries over large data in a distributed environment

Abstract Distributed data store are massively used in the actual context of Big Data. In addition to provide data management features, those systems have to deal with an increasing amount of queries sent by distant users in order to process data mining or data visualization operations. One of the main challenge is to evenly distribute the workload of queries between the nodes which compose these systems in order to minimize the treatment time.

In this thesis, we tackle the problem of query allocation in a distributed environment. We consider that data are replicated and a query can be handle only by a node storing the concerning data. First, near-optimal algorithmic proposals are given when communications between nodes are asynchronous. We also consider that some nodes can be faulty. Second, we study more deeply the impact of data replication on the query treatement. Particularly, we present an algorithm which manage the

data replication based on the demand on these data. Combined with our allocation algorithm, we guaranty a near-optimal allocation. Finally, we focus on the impact of data replication when queries are received as a stream by the system. We make an experimental evaluation using the distributed database Apache Cassandra. The experiments confirm the interest of our algorithmic proposals to improve the query treatement compared to the native allocation scheme in Cassandra.

Keywords Load balancing, distributed algorithm, graph, large data

Laboratoire d'accueil LaBRI, UMR 5800, 351, cours de la Libération F-33405 Talence cedex

Remerciements

Le présent manuscrit est l'aboutissement de 3 ans (et demi) de thèse. Cette période fut pour moi très riche à de nombreux niveaux et fut surtout jalonnée par des rencontres qui ont toutes contribuées à la réalisation de ce document. Car une thèse ne s'écrit pas seul (et encore moins toute seule). Par conséquent je voudrais remercier quelques unes des personnes que j'ai eu la chance de rencontrer.

Je tiens tout d'abord à remercier mon directeur de thèse, Nicolas Hanusse, sans qui cette aventure n'aurait jamais pu commencer. Il a toujours fait preuve jusqu'au bout d'une grande disponibilité (les nombreuses heures de discussion dans son bureau en témoignent), de patience (le même argument s'applique ici) et de pédagogie pour me guider au cours de la thèse. Il a toujours crû en moi (souvent plus que moi-même d'ailleurs) et a toujours su me motiver pendant les périodes difficiles. Sa vision de la recherche et son désir de toujours motiver le problème ont grandement contribué à forger le chercheur que je suis devenu et je l'en remercie pour cela. Ce fut un réel plaisir de travailler avec lui et j'espère que les occasions de collaborer à nouveau ensemble se présenteront par la suite.

Je tiens sincèrement à remercier mes deux rapporteurs, Nicolas Nisse et Maria Potop-Butucaru, pour avoir pris le temps de relire mon manuscrit de thèse. Merci de leurs conseils et remarques qui m'ont permis d'améliorer la qualité du manuscrit ainsi que des nombreuses pistes de recherches évoquées lors de la soutenance. Merci également à Olivier Beaumont d'avoir accepter de présider mon jury de thèse et pour ses questions intéressantes au cours de la soutenance.

Je voudrais bien sûr remercier tous les membres du groupe de travail Algorithmique Distribuée du LaBRI. Évoluer parmi vous a toujours été très agréable et les séminaires du Lundi à 14h me manqueront, indéniablement. J'espère avoir prochainement quelque chose d'intéressant à raconter pour avoir l'occasion de revenir. Je voudrais tout particulièrement remercier Arnaud Casteigt (pour m'avoir permis de participer à l'organisation d'Algotel, c'était super), Cyril Gavoille (pour avoir aidé à faire pousser le graphe fleur, entre autres) et David Ilcinkas (pour ses nombreux retours post-exposés, toujours pertinents et bienveillant). Je voudrais également remercier tous les doctorants (dont la plupart sont maintenant docteurs) : Christian, Matthieu, Nesrine, Vincent et Yessin ; pour tous les bons moments partagés ensemble, notamment lors des conférences. J'ai une pensée toute particulière pour David B., dont la connaissance tolkiennesque m'épatera toujours, et Claire C., qui sait à quel point faire un logo est un exercice difficile.

Je voudrais bien sur remercier tous les membres du groupe Big Data du vendredi matin. Assister aux différentes réunions a ouvert mon horizon scientifique. Je voudrais remercier en particulier Sofian Maabout pour s'être intéressé à mon travail, pour ses nombreuses remarques constructives et bien sûr pour sa gentillesse. Les différents séjours à BDA ont toujours été très agréables en sa compagnie. Merci également à Emanuela pour sa bonne humeur et sa gentillesse permanente. Il m'est également impossible de ne pas citer les mangeurs de sandwichs du 3eme étage, qui m'ont gentiment accueilli parmi eux. Merci aux doctorants : Alexandre, Antoine H., Aaron, Gaëlle, Jason, Joris ; et aux permanents : David A., Romain B. et Romain G. ; sans oublier Fred (pour avoir décortiqué Cassandra) ; pour toutes les discussions (pas toujours) scientifiques et les bons moments partagés.

Je tiens également à remercier les membres de l'UF Informatique de l'Université de Bordeaux, avec lesquels j'ai eu la chance de pouvoir interagir. Merci pour leurs conseils, anecdotes et réponses à mes nombreuses questions. Merci également aux doctorants, de l'AFoDIB ou non, de Bordeaux et d'ailleurs, pour les bons moments passés en votre compagnie. Merci en particulier à Claire P. pour sa joie de vivre et à Quentin Bramas, dont l'heureux hasard a voulu qu'il soit à Bordeaux pendant ma soutenance.

J'en viens bien sûr tout naturellement à mes proches. Merci à mes parents et mon (pas si petit que ça) petit frère (à qui je n'avais pas rappelé le protocole de soutenance) pour m'avoir soutenu bien avant la thèse et tout au long de celle-ci. Si j'en suis là aujourd'hui c'est bien entendu grâce à eux. Je voudrais également rendre un immense merci à Eugénie, que je considère un peu comme un membre de ma famille, pour m'avoir hébergé pendant mes exils bordelais et pour toutes les discussions passionnantes et les bons moments passés en sa compagnie. Merci à tous mes amis et ma famille qui ont pu faire le déplacement et à tous les autres qui ont pensés à moi et m'ont chaleureusement félicité à la fin.

J'ouvre un paragraphe spécial pour remercier mon ami de longue date Romaric Duvignau, qui, malgré les grèves des transports, a pu venir assister à ma soutenance et donner un coup de main pour l'organisation du pot (nous sommes quitte maintenant).

Pour terminer, je tiens bien sûr à remercier ma compagne Jessica, qui sait mieux que personne tous les sacrifices que cette thèse représente. Son amour et son soutien sont indissociables de la fin heureuse de cette aventure, et pour cela je ne la remercierai jamais assez.

Table des matières

Chapitre 1

Introduction

Sommaire

1.1 Contexte et motivation

La production de données a littéralement explosé ces dernières années, notamment grâce au développement des réseaux à grandes échelles. Outre ce volume gigantesque, les données proviennent de sources variées allant des réseaux sociaux aux données scientifiques issues de réseaux de capteurs par exemple. Afin de pouvoir gérer cette masse de données toujours croissante et diversifiée, la conception de systèmes de stockage distribués a reçu énormément d'attention ces dernières décennies. En effet, de part leur volume mais également leur taille, un stockage centralisé de ces données n'est pas envisageable. En particulier, les bases de données distribuées ont connu un essor important à la fin des années 2000, avec l'apparition entre autres de systèmes comme Big Table [CDG+08] ou Cassandra [LM10] pouvant se déployer sur des infrastructures distribuées de grande taille et gérer des données de différents types.

Un autre aspect auquel doivent faire face ces systèmes distribués est l'accès à ces données. En effet l'extraction et l'analyse de ces données sont des problématiques majeures et engendrent de nombreuses requêtes *en lecture*, souvent coûteuses. Toutes ces *requêtes* doivent être répondues rapidement, parfois même en temps réel en fonction des applications. Il est donc nécessaire de mettre en place des mécanismes *d'allocation* afin de répartir au mieux les requêtes reçues entre les différents nœuds capables de les traiter.

1.2 Équilibrer la charge des requêtes

Un système de stockage distribué est composé d'un ensemble de noeuds stockant des données, que nous appellerons des *objets*. Chaque nœud peut servir de premier contact à un client souhaitant se connecter à la base afin d'effectuer des *requêtes* sur les objets stockés. Par définition, une requête peut donc être traitée uniquement par un nœud stockant l'objet concerné. On supposera également que les requêtes reçues sont indépendantes les unes des autres, c'est-à-dire que le traitement d'une requête ne dépend pas du traitement d'une autre requête.

On remarque tout d'abord que le placement des objets va avoir un impact important sur le nombre de requêtes que recevra un nœud. En effet, si le stockage est trop déséquilibré et qu'un nœud stocke beaucoup d'objets, alors il pourra potentiellement recevoir beaucoup de requêtes.

Toutefois un équilibrage parfait du stockage ne peut pas nécessairement garantir un bon équilibrage des requêtes. En effet, certains objets peuvent être plus demandés que d'autres, entraînant une surcharge pour les nœuds stockant ces objets. L'objectif de cette thèse est donc de proposer à la fois un placement d'objets mais également une répartition des requêtes entre les nœuds permettant d'optimiser le traitement des requêtes reçues par un système distribué.

1.2.1 Équilibrage du stockage

L'*équilibrage du stockage* a été largement étudié, notamment dans le cas des réseaux dynamiques de type *pair-à-pair*. Les réseaux pair-à-pair sont des réseaux logiques décentralisés fonctionnant comme une surcouche d'un réseau physique et où chaque nœud joue à la fois le rôle du client et de serveur et peuvent ainsi émettre et répondre à des requêtes. Grâce à ce caractère décentralisé, ce type de réseau offre des propriétés intéressantes comme le passage à grande échelle ou encore la tolérance aux fautes.

Nous nous intéressons plus particulièrement aux réseaux structurés, pour lesquels une *table de hachage distribuée* est mise en place afin de faciliter entre autres la recherche d'objets. Informellement, chaque nœud du système se voit attribuer un identifiant dans un espace d'adressage et est responsable d'un *intervalle* de cet espace d'adressage. Les objets que l'on veut stocker se voient également attribuer un identifiant dans le même espace d'adressage et sont stockés par le nœud responsable de leur identifiant. Les nœuds maintiennent également un ensemble de lien entre eux, qui permettrons de faciliter la recherche d'un objet. Un pan important de la littérature traite donc de l'équilibrage de l'espace d'adressage dont sont responsables les nœuds. Plus précisément, on souhaite minimiser le *ratio de régularité* qui correspond au rapport de l'intervalle le plus grand par l'intervalle le plus petit[KLL+97, KM05, BKM05].

Un autre type de solution consiste à utiliser des *serveurs virtuels* [SMK+01, GLS+04, KR06]. Chaque nœud physique va maintenir un ensemble de serveurs vir-

tuels qui sont, chacun, responsables d'une portion de l'espace d'adressage. Dans Chord[SMK+01] par exemple, chaque nœud possède $\Theta(\log n)$ serveurs virtuels, ce qui permet d'obtenir un ratio de régularité constant.

Ces méthodes ont un inconvénient majeur à savoir que si un nœud se retrouve surchargé, on va modifier le partitionnement de l'espace d'adressage, impliquant la *migration* de certains objets sur d'autres nœuds moins chargés. Dans le contexte de données massives, la migration est coûteuse et il est donc naturel de vouloir minimiser, voire de ne pas utiliser ce type d'opération.

1.2.2 Impact de la réplication

Pour limiter la migration, une solution consiste à augmenter le nombre de copies d'objets afin de permettre un meilleur équilibrage des requêtes. La *réplication* d'objets est une des caractéristiques déjà présente dans les bases de données distribuées [CDG+08, LM10, DHJ+07]. Typiquement, chaque objet est répliqué un nombre constant de fois, principalement pour garantir l'accessibilité des objets. Ce nombre de copies, identique pour tous les objets, est appelé le *facteur de réplication*. Nous pouvons cependant constater que ces copies ne sont pas utilisées à des fins d'équilibrage dans les systèmes actuels. HBase par exemple ne propose que des méthodes d'équilibrage basées sur de la migration d'objets. Cassandra propose également uniquement de migrer des objets en cas de surcharge d'un nœud.

Cependant, on peut facilement envisager des scénarios où le système peut se retrouver surchargé. Considérons que les requêtes sont effectuées par un *adversaire* qui connaît le placement de toutes les copies. Il peut donc concentrer les requêtes sur des objets stockés en commun par un même ensemble de nœuds. Dans ce cas de figure, la seule solution pour équilibrer parfaitement la charge est de créer autant de copies que de nœuds, ce qui n'est bien sûr pas envisageable dans un contexte de données massives.

Afin de palier ce comportement adversarial, on peut adapter la réplication en fonction de la demande (le nombre de requêtes reçues) pour un objet. On va ainsi créer des copies additionnelles pour ces objets dit *populaires*. Outre l'intérêt en terme d'accessibilité des données (si un nœud tombe en panne, on a toujours une copie accessible), cette stratégie présente l'avantage de pouvoir s'adapter à toute distribution de requêtes. De plus, on peut ainsi répartir les requêtes pour un objet entre les différents nœuds qui en stockent une copie. Différentes stratégies de placement de copies ont été étudiées dans le cadre des réseaux pair-à-pair : à proximité des serveurs/nœuds émettant les requêtes [GSBK04], sur les chemins fréquemment empruntés par les requêtes [She10] ou encore via l'utilisation de fonctions de hachages multiples [XCCK09].

Comme de nombreuses bases de données distribuées fonctionnent sur un modèle pair-à-pair, on pourrait adapter ces méthodes basées sur la gestion de copies. Cependant on peut noter un défaut principal : elles ne considèrent qu'une estimation *locale* de la popularité des objets, ce qui ne reflètent pas forcément la popularité

globale. De plus, *l'évolution de la popularité* des objets n'est pas toujours prise en compte. Par exemple dans [XCCK09], le nombre de copie peut être augmenté en fonction de la popularité, par contre les copies créees sont conservées même si la popularité diminue. Dans un contexte de données massives, cela impliquerait une consommation mémoire trop importante.

1.3 Contribution de la thèse

L'objectif de cette thèse est de proposer des solutions d'équilibrage pour la charge des requêtes reçues par des systèmes distribués. Nous nous intéressons plus particulièrement aux systèmes de stockage distribués. Comme nous avons pu le voir, des *copies d'objets* sont présentes initialement dans de nombreux systèmes distribués et nous ferons l'hypothèse dans le reste de ce manuscrit qu'il y a toujours *un nombre constant de copies* présentes sur le système pour chaque objet. Nous ne nous intéressons pas à la migration d'objets et considérons que ces copies initiales sont *stockées de manière permanente* par les nœuds.

Le premier axe de recherche abordé dans ce manuscrit consiste à étudier de quelle manière *allouer les requêtes* entre les différents nœuds candidats qui stockent les copies.

Dans le Chapitre 2, nous présentons formellement le problème d'allocation de requêtes qui sera étudié ainsi que les différents modèles utilisés tout au long de la thèse. En particulier, nous montrons que nous pouvons représenter les requêtes à allouer par un *(hyper)graphe de requête*, où les arêtes représentent les requêtes et les extrémités des arêtes représentent les candidats pouvant traiter chaque requête. L'allocation est modélisée ici par une orientation sortante des arêtes du graphe et la charge d'un nœud correspond au nombre d'arcs sortant qui lui sont incidents. On va donc chercher à minimiser le degré sortant maximum du graphe. Cette approche utilisant la théorie des graphes est un élément clé des différents résultats obtenus dans cette thèse. Nous présentons également les modèles de calculs distribués que nous manipulons.

Dans le Chapitre 3, nous nous intéressons à l'allocation de requêtes dans un système distribué utilisant un mode de communication *asynchrone*, c'est-à-dire que le temps d'envoi d'un message entre deux nœuds n'est pas connu et varie d'un lien de communication à l'autre. Ici on s'intéresse à une vision *hors-ligne* où toutes les requêtes sont déjà reçues par le système. Nous étudions le problème d'orientation de graphe équivalent et nous nous concentrons sur le cas où le facteur de réplication est 2, c'est-à-dire au cas des graphes. Nous proposons un algorithme d'orientation qui produit une *approximation constante* de l'optimal pour tout graphe G. Plus précisément, notre algorithme nécessite un nombre de ronde asynchrone de l'ordre de $O(\log^2 n)$, où n est le nombre de nœuds et T_B le temps maximum nécessaire pour envoyer un message à tous les nœuds du graphe G. Notre algorithme pré-

sente également l'avantage d'être *tolérant aux pannes initiales*, ce qui est un atout majeur dans des contextes d'utilisation réels. Nous montrons également que tout algorithme d'orientation doit utiliser au moins $\Omega(diam(G))$ rondes de communication, où $diam(G)$ correspond au nombre maximum d'arêtes à emprunter pour relier deux nœuds.

Nous nous intéressons dans un deuxième temps à *l'impact de la réplication* sur l'équilibrage de charge. Nous avons vu effectivement que le nombre des copies d'objets mais également leur placement avait un rôle important sur l'équilibrage des requêtes.

Dans le Chapitre 4, nous proposons de placer les copies d'objets de manière aléatoire, uniforme et indépendante. Nous montrons que cette stratégie de placement permet d'obtenir une allocation proche de l'optimal si la popularité des objets n'est pas trop hétérogène pour un facteur de réplication donné (Section 4.2). Nos résultats se basent sur l'étude de l'*hypergraphe des requêtes* sous-jacent. Après avoir donné un encadrement de l'allocation optimal lorsque les objets sont placés aléatoirement et les requêtes ne sont pas populaires, nous présentons une combinaison d'algorithmes garantissant une f-approximation de l'optimal quel que soit le facteur de réplication f, et une $O(f)$-approximation de l'idéal si $f = \Omega(\log n)$. L'idéal correspond simplement à une répartition parfaitement équilibrée des requêtes entre les nœuds. Cette garantie tient même si le choix des requêtes est laissé à un *adversaire*. De plus, notre méthode n'ajoute que $\Theta(n)$ copies supplémentaires pour les objets populaires.

Enfin dans le Chapitre 5 nous nous plaçons dans un contexte où les requêtes arrivent *en flux*, à l'inverse des chapitres précédent. Nous adaptons les algorithmes présentés dans le Chapitre 4 et nous procédons à une évaluation expérimentale menée sur une version modifiée de la base de données distribuée Apache Cassandra [Cas]. Nous adaptons nos algorithmes dans ce contexte, nous les comparons avec la stratégie d'allocation proposée par défaut dans Cassandra et évaluons notamment l'impact de notre gestion de copies sur les performances des différents algorithmes. Nous montrons que notre gestion de copies permet d'améliorer sensiblement le temps de réponse aux requêtes quel que soit l'algorithme considéré. Nous montrons également que nos algorithmes d'allocation ont de meilleures performances que celui utilisé dans Cassandra par défaut.

Chapitre 2

Modèles et état de l'art

Sommaire

Nous proposons d'étudier le problème de l'optimisation du traitement des requêtes en se ramenant au problème bien connu de l'allocation de requêtes. En particulier, nous considérons la version du problème où les requêtes arrivent de manière *distribuée* et par conséquence les décisions d'allocations doivent être prises par les noeuds avec une connaissance *partielle* du système. La différence majeure entre le modèle que nous proposons et les modèles plus classiques de la littérature est que nous autorisons une requête à être envoyée à plusieurs noeuds pour être traitée. Cette caractéristique permet notamment de *différer* la prise de décision pour l'allocation. Après avoir formalisé le problème sur lequel nous travaillerons, nous présentons notre angle d'attaque qui consiste à se réduire à un problème de théorie des graphes bien connu appelé l'*orientation minimum* de graphes. Nous présentons également quelques algorithmes majeurs d'orientation de graphes. Nous introduisons enfin les principaux modèles distribués que nous manipulerons dans ce manuscrit.

2.1 Préliminaires

2.1.1 Présentation du modèle d'allocation

On considère un système de stockage distribué composé d'un ensemble V de n noeuds de traitement. Le système va recevoir un ensemble Q de requêtes à traiter et chaque requête $q \in Q$ possède un *coût de traitement* ou *poids* que l'on notera $w(q)$.

On fait l'hypothèse qu'une requête peut être reçue par tout nœud $u \in V$ et sera mise en attente dans la file de requête F_u de u. Une requête q est dite *permanente* si elle n'est jamais supprimée de la file et *temporaire* sinon. Le problème d'allocation que nous étudions dans cette thèse consiste à répartir les requêtes reçues entre les différents nœuds afin de **minimiser** le temps de traitement de ces requêtes.

Considérons un premier scénario où toutes les requêtes ont un poids de 1 et sont reçues par un serveur central ayant une connaissance globale du système. C'est ce serveur qui va s'occuper de l'allocation. Supposons également qu'une requête peut être allouée à n'importe quel nœud du système. Dans ce cas de figure, on peut envisager l'algorithme d'allocation suivant :

Entrée: ensemble de requêtes Q, ensemble de nœuds $V = \{u_1, u_2, \ldots, u_n\}$
1: $i = 1$
2: **pour** toute requête $q \in Q$ **faire**
3: allouer q à u_i
4: **si** i=n **alors** $i \leftarrow 0$
5: $i \leftarrow i + 1$

Algorithme 1 – `Round Robin`

Définissons informellement la charge d'un nœud comme le nombre de requêtes qui lui sont allouées. Cet algorithme très simple, dit du tourniquet ou *round robin* en anglais, permet de garantir que la différence de charge entre deux nœuds quelconques sera d'au plus une requête. Cependant, ce cas de figure n'est pas envisageable dans les systèmes de stockage distribués où une requête concerne typiquement un objet et ne peut donc être traitée que par un nœud stockant cet objet (ou encore une de ses *copies*).

Afin de modéliser cela, on considère un ensemble d'objets O stockés par les nœuds et un *placement d'objets* noté $\mathcal{P}$ qui détermine sur quel nœud sera stocké un objet. Chaque objet o_i possède $f_i \geq 1$ copies (incluant o_i) qui sont stockés sur des nœuds différents et $\mathcal{P}(o_i) \subset V$ correspond à l'ensemble des nœuds stockant une copie de o_i. On supposera donc que le nombre copies f_i pour tout objet o_i est inférieur à n. Sauf mention contraire, on suppose que les nœuds ont le même nombre de copies f et on appelle ce paramètre le *facteur de réplication*.

Une requête q_o sur un objet o ne peut être traitée que par un nœud de $P(o)$ et nous dirons que ces nœuds sont *candidats* pour la requête q_o. Sauf mention contraire, on considère que les requêtes sont permanentes.

Définition 2.1. Un *algorithme d'allocation* $\mathcal{A}_{\mathcal{P}}(q_o)$ pour un placement d'objets $\mathcal{P}$ associe à la requête q_o un nœud candidat $v \in P(o)$.

Définition 2.2. Soit Q un ensemble de requête et $\mathcal{A} = \mathcal{A}_{\mathcal{P}}$ un algorithme d'allocation pour un placement d'objets P. On note $Q_{\mathcal{A}}(u) = \{q \in Q \mid \mathcal{A}(q) = u\}$ l'ensemble des requêtes de Q allouées à u avec l'algorithme $\mathcal{A}$. La *charge* $\ell_{Q,\mathcal{A}}(u)$ d'un nœud u est définie par

$$\ell_{Q,\mathcal{A}}(u) = \begin{cases} \sum_{q \in Q_{\mathcal{A}}(u)} \omega(q) & \text{si } Q_{\mathcal{A}}(u) \neq \emptyset \\ 0 & \text{sinon} \end{cases}$$

Si $w(q) = 1$ pour toute requête $q \in Q$, on remarque que $\ell_{Q,\mathcal{A}}(u) = |Q_{\mathcal{A}}(u)|$.

Définition 2.3. Le *temps de complétion* $T_{\mathcal{A}_{\mathcal{P}}}(Q)$ avec l'algorithme d'allocation $\mathcal{A}_{\mathcal{P}}$ est défini par :

$$T_{\mathcal{A}_{\mathcal{P}}}(Q) = \max_{u \in V} \{\ell_{Q,\mathcal{A}_{\mathcal{P}}}(u)\}$$

Définition 2.4. Soit $\mathbf{P}$ l'ensemble des placements d'objets possibles et $\mathbf{A}_{\mathcal{P}}$ l'ensemble des algorithmes d'allocation possibles en fonction d'un placement d'objets $\mathcal{P}$. Le temps de complétion optimal $T^*(Q)$, est défini par

$$T^*(Q) = \min_{\mathcal{P} \in \mathbf{P}} \left\{ \min_{\mathcal{A} \in \mathbf{A}_{\mathcal{P}}} \{T_{\mathcal{A}}(Q)\} \right\}$$

De même on définit le temps de complétion optimal $T^*_{\mathcal{P}}(Q) = \min_{\mathcal{A} \in \mathbf{A}_{\mathcal{P}}} \{T_{\mathcal{A}}(Q)\}$ vis-à-vis d'un placement $\mathcal{P}$.

Une dernière notion que nous abordons au cours de ce manuscrit est la notion de *temps de complétion idéal*. Le temps de complétion idéal $T^{\diamond}(Q)$ correspond simplement à une répartition parfaitement équitable des requêtes entre les nœuds. Plus formellement on a $T^{\diamond}(Q) = \sum_{q \in Q} w(q)/n$.

Objectif de la thèse Le problème étudié dans cette thèse consiste donc à trouver un placement d'objet $\mathcal{P}$ et un algorithme d'allocation $\mathcal{A}_{\mathcal{P}}$ pour un ensemble de requêtes Q tel que le temps de complétion $T_{\mathcal{A}_{\mathcal{P}}}(Q)$ soit *minimal*. Nous nous intéressons plus particulièrement au cas où $w(q) = 1$ pour toute requête $q \in Q$. Nous mesurons notamment l'approximation temps de complétion optimal mais également du temps de complétion *idéal*, qui correspond tout simplement à $\lceil |Q|/n \rceil$ dans notre contexte.

2.1.2 Les stratégies d'allocation

Concernant les stratégies d'allocation possibles, nous considérerons plusieurs modèles au cours de cette thèse.

— Dans l'allocation **en-ligne**, la requête doit être allouée au moment de sa réception par le système. Informellement, on se base uniquement sur le *passé*, c'est-à-dire sur les requêtes déjà reçues par le système afin de prendre une décision. Cette décision peut toutefois être *immédiate* (on choisit **un** nœud parmi l'ensemble des nœuds capable de traiter la requête concernée) ou bien *différée* (on alloue la requête à **tous** ses nœuds *candidats* et on décide à posteriori lequel d'entre eux va conserver cette requête). On remarque que dans ce deuxième cas de figure, une même requête peut être traitée plusieurs fois.
— Dans l'allocation **hors ligne**, on attend d'avoir reçu toutes les requêtes avant de prendre une décision pour l'affectation. Intuitivement, cette stratégie se base sur une connaissance du passé et du futur pour prendre une décision d'allocation.

La vision en-ligne que nous venons de décrire s'applique bien aux situations concrètes où les requêtes arrivent en *flux* sur le système. Prenons par exemple le cas d'un moteur de recherche qui serait utilisé par un grand nombre d'utilisateurs. En pratique les recherches des utilisateurs s'effectuent en permanence et il n'est pas envisageable de retarder trop longtemps l'allocation des requêtes. De plus, le flux étant potentiellement infini, l'application d'une vision hors-ligne perd son sens. Toutefois, on se convainc sans trop de difficulté que connaître le futur ne peut qu'aider à faire des choix judicieux. En particulier, on fait la remarque suivante.

Remarque 2.1. *Tout algorithme d'allocation en ligne sur une séquence finie de requêtes peut s'exprimer comme un algorithme d'allocation hors-ligne sur cette même séquence de requête.*

En effet il suffit de simuler un algorithme en ligne et d'appliquer les même décisions dans la version hors-ligne. La conséquence de cette remarque est que les algorithmes en ligne ne peuvent pas garantir un meilleur temps de traitement total des requêtes que les algorithmes hors-ligne.

De nombreux travaux tendent donc à mesurer la performance d'algorithme en ligne en les comparant au *meilleur* algorithme hors-ligne pour résoudre le même problème. Le ratio de performance entre les deux algorithmes est appelé *ratio de compétitivité* et fut introduit par Graham [Gra66]. L'auteur montre notamment que l'algorithme très simple appelé `Greedy` qui consiste à choisir le candidat avec la charge la moins grande au moment de la réception d'une requête est $(2 - \frac{1}{n})$-compétitif. Les requêtes sont supposées ici ne jamais être traitées et ne pas avoir de restriction de candidats. On trouve énormément de travaux dans la littérature faisant varier ces deux caractéristiques (permanence et restriction) mais également l'aspect distribué de l'arrivée des requêtes. On peut citer entre autres [ABK94, ANR95, AKP92, DLLX97].

Tous les cas présentés dans la littérature considèrent cependant une prise de décision immédiate. La *prise de décision différée* semble pourtant une piste naturelle

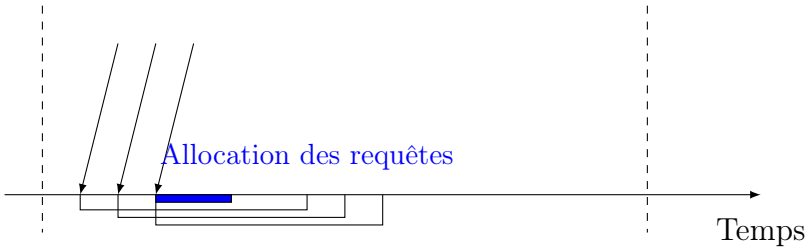


FIGURE 2.1 – Illustration de l'intérêt de la prise de décision différé dans le cas où les requêtes sont coûteuses. Le temps nécessaire à l'allocation de requêtes, représenté en bleu, est nettement inférieur au temps de traitement d'une requête. On retombe pour ces requêtes à une vision hors-ligne.

de réflexion. Cette méthode d'allocation est déjà utilisée en pratique, notamment dans les bases de données distribuées, afin de tester la *cohérence* d'une réponse à une requête. En effet, un objet est typiquement répliqué dans la base mais toutes les copies peuvent ne pas être à jour. Faire traiter la requête par les différents nœuds stockant l'objet concerné permet ainsi à un client de comparer les réponses et de garder par exemple celle correspondant à la donnée la plus récente. Si on ne s'intéresse pas à la cohérence mais seulement à la performance, cette solution peut se révéler avantageuse. Informellement, plus notre connaissance de l'ensemble des requêtes est grande, plus il sera facile de prendre des décisions pertinentes pour l'allocation. Si la mémoire le permet, on pourrait même stocker toutes les requêtes et retomber sur une vision hors-ligne du problème.

Bien que cet exemple extrême ne soit bien sûr pas applicable en pratique, le flux de requêtes pouvant être énorme, une solution peut consister à faire le bilan sur un ensemble de requêtes reçues au cours d'un intervalle de temps. Nous avons illustré ce scénario dans la Figure 2.1. Intuitivement, si le temps de traitement d'une requête est important, un algorithme d'allocation suffisamment *rapide* pourra allouer de manière efficace les requêtes reçues au cours de cet intervalle. Nous présentons notamment dans le chapitre 5 des algorithmes différés d'allocation qui font le bilan à intervalle régulier et se basent sur cette connaissance pour répartir les requêtes.

2.2 Orientation de graphe et allocation de requêtes

Plaçons-nous dans le cas où le facteur de réplication est 2 pour toutes les copies d'objets. Il y a donc deux candidats pour chaque requête. De manière générale, on peut modéliser un ensemble de requêtes et leurs candidats liés à un placement d'objets donné par un *graphe* noté $G = (V, E)$ où l'ensemble des sommets $V(G)$ représente les nœuds et l'ensemble des arêtes $E(G)$ représente les requêtes à allouer. Les extrémités d'une arête encodent les candidats possibles pour l'allocation. On note $d(u)$ le degré non orienté d'un nœud u. Par la suite nous utiliserons l'appellation *graphe des requêtes* pour désigner cette représentation.

Nous allons modéliser l'allocation des requêtes par l'*orientation* des arêtes correspondantes. Une orientation ϕ de G consiste à assigner à toute arête $\{u, v\} \in E$ une orientation (informellement un *sens*). Soit (u, v) un arc sortant de u, alors la requête correspondante à cet arc sera alloué à u. Soit $d^+_\phi(u)$ le degré sortant de u, c'est-à-dire le nombre d'arêtes orientées vers des voisins de u, avec l'orientation ϕ et $D^+_\phi(G) = \max\{d^+(u) \mid u \in V\}$ le degré sortant maximum avec cette même orientation.

Orientation Minimum

— Entrée : un graphe G non-orienté
— Sortie : une orientation ϕ tel que pour toute orientation $\phi' \neq \phi$, alors $D^+_\phi(G) \leq D^+_{\phi'}(G)$

On remarque que D^+ correspond au temps de complétion que l'on obtiendrait en appliquant l'allocation définie par l'orientation à toutes les requêtes.

Définition 2.5. Une orientation ϕ d'un graphe G est une k-orientation si $D^+_\phi(G) \leq k$.

Avant de chercher à concevoir un algorithme, une question naturelle à se poser consiste à déterminer la classe de complexité du problème. Intéressons nous dans un premier temps à la version généralisée du problème de l'orientation minimum. Soit w une fonction de poids qui associe à chaque arête un entier. Le degré sortant pondéré correspond à la somme des poids des arcs sortants de u. L'orientation minimum généralisée consiste à trouver une orientation qui minimise le degré sortant pondéré. Ce problème a été démontré NP-difficile dans [AMOZ06] et dans [AJM$^+$11] les auteurs renforcent ce résultat en montrant que même si les poids des arêtes sont compris dans un intervalle $[1, k]$, le problème est fortement NP-difficile. De plus, il n'existe pas d'approximation inférieure à $(1 + 1/k)$, sauf si $P = NP$. Toutefois la version "simplifiée" que nous considérons, où le poids de chaque arête est identique et vaut 1, est fort heureusement solvable en temps polynomial, comme nous allons le voir par la suite.

Les deux catégories d'algorithmes d'allocation dont nous avons parlé précédemment sont modélisées de manière légèrement différentes. Les algorithmes d'allocation en-ligne sont modélisés par des graphes *dynamiques* où l'arrivée des requêtes est représentée par une suite d'insertions d'arêtes. A chaque nouvelle insertion d'une arête e, une décision d'orientation doit être prise pour e. Les algorithmes d'allocation hors-ligne sont par contre modélisés par des graphes *statiques* où l'ensemble des arêtes n'évolue pas au cours du temps. L'énoncé du problème reste cependant identique, c'est-à-dire que l'on cherche à minimiser le degré sortant maximum après avoir pris une décision d'orientation pour toutes les arêtes.

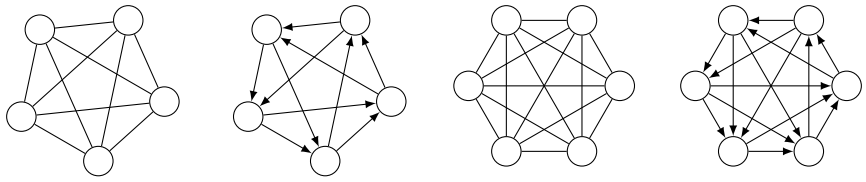


FIGURE 2.2 – Orientation des graphes complets K_5 et K_6

2.2.1 Lien entre la densité de graphe et l'orientation minimum

Dans le reste de cette sous-section nous considérerons le modèle d'allocation hors-ligne et donc l'orientation du graphe statique sous-jacent. Tout d'abord, posons quelques définitions et notations qui seront utilisées dans tout le reste du manuscrit.

Définition 2.6. Soit $S \subseteq V(G)$ un sous-ensemble de nœuds de G. Le sous-graphe induit $G[S]$ de G a pour ensemble de nœuds S et pour ensemble d'arêtes $\{\{u, v\} \mid u, v \in S \wedge \{u, v\} \in E(G)\}$.

Définition 2.7. La densité $\delta(G)$ d'un graphe $G = (V, E)$ est défini par $\frac{|E|}{|V|}$ et la densité maximale $\Delta(G)$ est le maximum des densités de tous les sous-graphes induits.

En guise d'échauffement, nous allons étudier l'orientation d'une famille de graphes connus appelés *graphes complets*. Un graphe complet K_n est un graphe à n sommets tel que chaque sommet est relié à tous les autres par une arête. Dans le cas du graphe complet, la densité maximale est égale à $\frac{|E|}{|V|} = \frac{n(n-1)}{2n} = \frac{n-1}{2}$. On montre la proposition suivante.

Proposition 2.1. *Il existe une orientation de K_n tel que $D^+(K_n) = \lceil \frac{n-1}{2} \rceil$.*

Démonstration. Considérons l'algorithme qui consiste à choisir séquentiellement chaque nœud $u_0, \ldots, u_{n-1}$ de K_n et à orienter au plus $k = \lceil \frac{n-1}{2} \rceil$ arêtes incidentes non orientées vers des nœuds $u_{i+1}, \ldots, u_{i+k \mod n}$. Il est clair qu'un nœud va avoir un degré sortant d'au plus $k = \lceil \frac{n-1}{2} \rceil$. Si $n-1$ est pair, un nœud a au plus $(n-1)/2$ nœuds qui ont orienté des arêtes incidentes, il reste donc exactement $(n-1)/2$. On en déduit qu'à chaque étape, on oriente $(n-1)/2$ arêtes, ce qui fait $n(n-1)/2 = m$ arêtes au total. Dans le cas où $n-1$ est impair, tout nœud $u_{j \leq (n/2)}$ aura un degré sortant de $n/2$ et les autres nœuds auront un degré sortant de $n/2 - 1$ soit $(n/2)^2 + (n/2)(n/2-1) = (n^2-n)/2 = \frac{n(n-1)}{2}$. □

L'illustration de l'orientation du graphe complet est donné sur la figure 2.2. On peut généraliser ce résultat pour tout graphe G et ainsi lier la densité maximum avec une orientation minimum. Tout d'abord posons quelques définitions. Soit D un graphe dirigé. Un **chemin dirigé** P est une séquence $(u_1, u_2), \ldots, (u_{k-1}, u_k)$ d'arcs de D. On dit que u_1 est la *source* de P et que u_k est la *destination* de P. Soit P un chemin dirigé de source u et de destination v. On dit que P est un **chemin**

augmentant si $d^+(u) > d^+(v) + 1$. En combinant les résultats de [AF76] et [FT14], on montre le résultat suivant.

Théorème 2.2 ([AF76, FT14])**.** *Soit $G = (V, E)$ un graphe simple quelconque.*

$$D^+_{opt}(G) = \lceil \Delta(G) \rceil$$

Démonstration. ($\leq$) On commence par choisir une orientation arbitraire pour G et on applique l'algorithme suivant : tant qu'il existe un chemin augmentant P avec une source u tel que $d^+(u) > \lceil \Delta \rceil$, on inverse l'orientation de tous les arcs de P. On va maintenant prouver qu'il existe toujours un tel chemin tant qu'il existe un sommet u avec $d^+(u) > \lceil \Delta \rceil$. Soit $H \subseteq G$ le sous-graphe induit par tous les chemins augmentant ayant pour destination u. Le nombre d'arcs de H est égal à $\sum_{v \in V(H)} d^+_H(v)$. Supposons maintenant qu'aucun de ces chemins ne soit augmentant, alors pour tout nœud $v \in V(H)$ on a $d^+(v) \geq \lceil \Delta \rceil$ par conséquent $|E(H)| > |V(H)| \cdot \lceil \Delta \rceil \leftrightarrow |E(H)|/|V(H)| > \lceil \Delta \rceil$ ce qui est impossible par définition de Δ.

($\geq$) L'argument est simplement une application du principe des tiroirs. Considérons un sous-graphe avec m' arêtes et n' sommets. Pour chaque arête on choisit une orientation donc il y a au moins un nœud avec au moins m'/n' arcs sortants. Comme l'argument peut être appliqué à tout sous-graphe, on en déduit la borne inférieure. □

On constate donc que l'orientation minimum est liée à la densité maximum d'un graphe. L'estimation de cette densité est donc un élément clé de l'orientation. Différents paramètres permettent de mesurer la densité d'un graphe G. L'*arboricité* $arb(G)$ est une notion introduite par Nash-Williams [NW64] et qui est définie par

$$arb(G) = max_{G' \in \mathcal{G}} \left\{ \left\lceil \frac{|E(G')|}{|V(G')| - 1} \right\rceil \right\}$$

où $\mathcal{G}$ représente l'ensemble de tous les sous-graphes induits de G. Une autre définition plus intuitive est que l'arboricité correspond au nombre minimum de forêts arêtes-disjointes $F_1, F_2, \ldots, F_{arb}$ tel que l'union de ces forêts forme le graphe tout entier. Nous prouvons la proposition suivante.

Proposition 2.3. *Soit G un graphe simple non dirigé.*

$$\Delta(G) \leq arb(G) \leq \Delta(G) + 1$$

Démonstration. La borne inférieure est trivial. Pour la borne supérieur, considérons un sous-graphe G' avec m arêtes et n sommets. On a $m/(n-1) - (m/n + 1) = (m - n^2 + n)/(n-1)n \leq (-n+1)/2(n-1) \leq 0$. En appliquant le même raisonnement pour tout sous-graphe de G, on déduit l'énoncé. □

On en déduit facilement que l'arboricité d'un graphe et sa densité maximum sont identiques à une constante près. D'autres paramètres proches sont également

utilisés comme la *pseudo-arboricité*. Une pseudo-forêt est une union d'arbres ayant chacun au plus un cycle. La pseudo-arboricité correspond donc au nombre minimal de pseudo-forêts nécessaire pour partitionner les arêtes d'un graphe.

Une autre mesure de densité est la *dégénérescence*. Considérons l'algorithme d'épluchage qui consiste à retirer un à un les sommets de plus petit degré du graphe. A chaque étape, on considère le degré dans le sous-graphe induit restant. Soit $u_{i_1}, u_{i_2}, \ldots, u_{i_n}$ l'ordre dans lequel les sommets sont retirés, alors on nomme cette séquence un *ordre de dégénérescence*. Un graphe est dit k-dégénéré si tout sommet u_{i_j} a au plus k voisins qui le suivent dans l'ordre. La dégénérescence $deg(G)$ de G est le plus petit entier k tel que G est k-dégénéré. On peut donner une autre définition équivalente : la dégénérescence de G est le plus petit entier k tel que pour tout sous-graphe induit $H \subseteq G$, on a $\delta(H) \leq k$. La proposition suivante est adaptée en partie du Lemme 2.31 de [BE13].

Proposition 2.4. *Soit G un graphe simple non dirigé.*

$$\Delta(G) \leq deg(G) \leq 2 \cdot \Delta(G)$$

Démonstration. Pour prouver la borne supérieure, étudions l'algorithme d'épluchage décrit précédemment. A chaque itération, on supprime un sommet de degré minimum. Soit H le sous-graphe restant, m le nombre d'arêtes restantes et n le nombre de sommets restants. Le degré moyen de H est $2 \cdot (m/n) = 2 \cdot \delta(H)$ donc le degré minimum est inférieur à $2 \cdot \delta(H)$. Comme $\delta(H) \leq \Delta(G)$ par définition, on en déduit qu'un nœud u a toujours au plus $2 \cdot \Delta(G)$ voisins lorsqu'il est supprimé, impliquant que u aura au plus $2 \cdot \Delta(G)$ voisins après lui dans l'ordre de dégénérescence. Pour la borne supérieure, soit $V' \subseteq V$ un sous-ensemble quelconque de i nœuds. On ordonne les nœuds de l'ensemble en fonction de leur apparition dans l'ordre de dégénérescence du graphe. Notons $U = \{u_1, \ldots, u_i\}$ un tel ordre et on note $U_{>j} = \{u_{j+1}, \ldots, u_i\}$ l'ensemble des nœuds de U d'indice supérieur à j. On en déduit l'inégalité suivante

$$|E(G[U])| = \sum_{j=1}^{i} d^+_{U_{>j}}(u_j) = \sum_{j=1}^{i} d^+_{U_{>j}}(u_j) \leq ki$$

ce qui nous permet de conclure que $|E(G[U])|/i \leq ki/i = k$. On peut appliquer le même raisonnement sur tous les sous-graphes induit de G, en particulier sur le sous-graphe de densité maximum. □

La majorité des travaux de la littérature que nous allons présenter dans le reste de ce manuscrit se base sur l'estimation d'un de ces paramètres de densité.

2.2.2 Quelques algorithmes d'orientation de graphes

2.2.2.1 Épluchage de graphe

Considérons l'algorithme d'épluchage suivant : (1) choisir le sommet de plus petit degré dans le graphe et le supprimer (ainsi que toutes les arêtes qui lui sont

1: **tant que** il existe un sommet de degré inférieur à k **faire**
2: trouver un sommet u tel que $d_{G'}(u) \leq k$
3: orienter toutes les arêtes (non orientées) $\{u, v\}$ vers v

Algorithme 2 – `KCoreOri(`G, k`)`

incidentes) (2) répéter l'opération (1) tant que G n'est pas vide. Au lieu de supprimer un sommet u, on va orienter toutes les arêtes non encore orientées $\{u, v\}$ vers v. De plus, on s'autorise à sélectionner un nœud de degré inférieur ou égal à k. On obtient ainsi l'Algorithme 2 nommé `KCoreOri(G,k)`.

On sait que pour tout graphe G de degré moyen $\overline{d}(G)$, il existe un nœud $u \in V(G)$ tel que $d(u) \leq \overline{d}(G) = 2 \cdot \Delta(G)$. On peut donc facilement conclure de la terminaison de l'algorithme `KCoreOri(`$G, 2\lceil\Delta(G)\rceil$`)`. De plus tout nœud u aura un degré sortant d'au plus $2\lceil\Delta(G)\rceil$, c'est-à-dire 2 fois l'optimal.

On en conclut que si la densité maximum du graphe était connue, l'algorithme `KCoreOri` permettrait de calculer une 2-approximation de l'orientation.

Si ce paramètre n'est pas connu, une solution consiste à essayer d'orienter le graphe avec l'Algorithme 2 en faisant varier la valeur de k jusqu'à orienter complètement le graphe. On définit ainsi l'algorithme `SimpleOri(G)` (3) qui produit donc dans le pire cas une 2-approximation de l'orientation

1: $k \leftarrow \lceil m/n \rceil$ ▷ $|V(G)| = n$, $|E(G)| = m$
2: $G' \leftarrow \emptyset$: sous-graphe orienté de G
3: **tant que** $G' \neq G$ **faire**
4: supprimer l'orientation de G
5: KCoreOri(G,K)
6: $k \leftarrow k + 1$

Algorithme 3 – SimpleOri(G)

Intéressons nous maintenant au temps d'exécution de cet algorithme. Le temps d'exécution d'un algorithme est clairement dépendant de la manière dont est représenté le graphe, ce que nous n'avons volontairement pas précisé pour le moment afin de simplifier le discours. On suppose donc que le temps nécessaire pour parcourir tous les sommets du graphe et tester s'ils possèdent une propriété donnée (dans notre cas, un degré ne dépassant pas un certain seuil) prend un temps $\Theta(m)$. De plus, on considère que l'orientation des arêtes incidentes d'un sommet se fait en temps constant. On en déduit que l'Algorithme 2 prend un temps $O(mn)$ dans le pire des cas.

Dans [FT14], les auteurs améliorent cette performance et proposent une $(2+\epsilon)$-approximation qui s'exécute en temps $O(m \cdot \log_s n)$, avec $s = (2+\epsilon)/2$. L'algorithme consiste à sélectionner l'ensemble des nœuds S_0 de G dont le degré est inférieur à $(2+\epsilon) \cdot (m/n)$ puis à orienter toutes les arêtes qui leurs sont incidentes, vers eux. On réitère ainsi l'opération avec le sous-graphe induit par $G_1 = G[V \backslash S_0]$. De manière

FIGURE 2.3 – Illustration de l'inversion d'un chemin augmentant. Les nœuds des extrémités voient leurs degrés modifiés, par contre ceux des autres nœuds du chemin augmentant restent inchangés.

générale, pour tout sous-graphe $G_i = G[V \backslash (\cup_{j=0}^{i-1} S_j)]$, les nœuds sélectionnés sont ceux dont le degré est inférieur à $(2+\epsilon) \cdot (m_i/n_i)$ avec m_i et n_i respectivement le nombre d'arêtes et de nœuds de G_i.

2.2.2.2 Algorithme de flots

Les algorithmes d'épluchage permettent de calculer une approximation de l'orientation minimum. Il est toutefois possible de calculer une orientation optimale en se ramenant au problème bien connu de flot maximum dans les réseaux. Nous présentons très brièvement ce problème mais on pourra trouver des précisions et des références supplémentaires dans [Kow06]. Dans ce problème, on considère un réseau représenté par un graphe dirigé $G = (V, E)$ avec $s, t \in V$ respectivement le nœud source et le nœud destination. On associe également à chaque arc $e \in E(G)$ une capacité $c(e)$. Un *flot* fl est une fonction qui associe à tout arc e un entier entre 0 et $c(e)$ et dont la valeur $|fl|$ correspond à $\sum_{u \in V} fl(s,u)$. La capacité résiduelle $c_{fl}(e) = c(e) - fl(e)$ correspond à la capacité restante après avoir appliqué le flot. On cherche à maximiser le flot que l'on peut acheminer depuis s vers t. Nous présentons la réduction suivante, tirée de [Kow06], entre le problème de l'orientation minimum vers le problème de flot maximum. D'autres réductions similaires sont présentées dans [PQ82, AAR95, AMOZ06]. Tout d'abord on considère une orientation quelconque de G, que l'on note $\overrightarrow{G} = (V, \overrightarrow{E})$ et on construit un réseau $G_k = (V_N, E_N)$ de la manière suivante. On définit $V_N = V \cup \{s, t\}$ et $E_N = \overrightarrow{E} \cup \{(s,u) \mid d^+(u) > k\} \cup \{(u,t) \mid d^+(u) < k\}$. De plus, on définit la fonction de capacité c_k qui associe à chaque arc (s,v) une capacité égale à $d^+(v) - k$, à chaque arc (v,t) une capacité égale à $k - d^+(v)$ et à tous les autres une capacité de 1. On note $c_k(u,S) = \sum_{v \in S} c_k(u,v)$ la somme des capacités des arcs sortants de u dans l'ensemble $S \subseteq V$ de sommets. On note G_k^{fl} le sous-graphe induit par les arcs de capacité résiduelle $c_{fl}(e) = c_k(e) - fl(e)$ non nul.

Théorème 2.5 ([Kow06]). *Soit G un graphe et fl un flot dans le réseau G_k. Il existe une k-orientation de G si et seulement si $|fl| = c_k(s, V-s)$. De plus, quand $|fl| = c_k(s, V-s)$ alors $D^+(G_k^{fl}) = k$.*

Sans rentrer dans les détails, la preuve consiste à montrer que si G est k-orientable mais que $|fl| < c_k(s, V-s)$, alors il existe un chemin augmentant dans G_k^{fl} partant de s et ayant pour destination un nœud de degré supérieur à k, menant à une contradiction. A l'inverse, si $|fl| = c_k(s, V-s)$ alors on peut montrer qu'il

n'existe pas de nœud avec un degré sortant supérieur à k. Afin de trouver facilement une orientation optimale de G, il suffit donc de chercher un flot fl avec pour valeur $|fl| = c_k(s, V-s)$ dans G_k, en testant successivement toutes les valeurs de k. Supposons que l'on dispose d'un algorithme `AugmentPath(G,k)` permettant de détecter un chemin augmentant. On propose ainsi l'Algorithme `OptimalOri(`G`)` (Algorithme 6) qui utilise un autre algorithme appelé `ImproveOri(`G, k`)` (Algorithme 5). Ce dernier consiste à trouver un chemin augmentant P entre $u \in N(s)$ et $v \in N(t)$, s'il existe, puis à inverser l'orientation de ce chemin. Comme illustré sur la Figure 2.3, cette opération a pour conséquence de baisser le degré sortant de u et donc de baisser la valeur de $|fl| = c_k(s, V-s)$. On remarque également que le degré sortant pour les autres nœuds du chemin ne change pas sauf pour v qui augmente son degré sortant d'une unité sans dépasser k. Lorsque `ImproveOri(`G, k`)` s'arrête, il n'y a plus de chemin augmentant et on considère donc 2 situations : (1) on a déjà atteint une orientation optimale et l'algorithme `OptimalOri(`G`)` s'arrête (2) on recommence le même processus avec $k \leftarrow k+1$ jusqu'à trouver l'orientation optimale. En appliquant le Théorème 2.5, on en déduit qu'une telle orientation sera déterminée.

$P \leftarrow$ `AugmentPath(G,k)`
tant que $P \neq \emptyset$ **faire**
    $P \leftarrow$ `AugmentPath(G,k)`
    Inverser l'orientation de P

Algorithme 5 – `ImproveOri(`G, k`)`

$k \leftarrow \lceil m/n \rceil$
$\overrightarrow{G} \leftarrow$ orientation arbitraire de G
`ImproveOri(`$\overrightarrow{G}, k$`)`
tant que $D^+(\overrightarrow{G}) \neq k$ **faire**
    $k \leftarrow k+1$
    `ImproveOri(`$\overrightarrow{G}, k$`)`

Algorithme 6 – `OptimalOri(`G`)`

Cet algorithme est analysé par Venkateswaran [Ven04]. L'auteur montre qu'une orientation optimale est calculée en temps $O(m^2)$. De nombreux articles se basent sur des algorithmes identiques ou similaires pour déterminer l'orientation minimum. Gabow et Westerman [GW92] sont les premiers à proposer un tel algorithme, qui s'exécute en temps $O(m^{\frac{3}{2}} \log n)$. Dans [AMOZ06], les auteurs améliorent légèrement les performances et proposent un algorithme en $O(m^{\frac{3}{2}} \log \lceil \Delta(G) \rceil)$.

On peut également se servir de techniques similaires pour déterminer la pseudo-arboricité d'un graphe, qui est un paramètre étroitement lié à l'orientation. En effet, en combinant des résultats de [AF76] et [PQ82], on en déduit que $pa(G) = \lceil \Delta(G) \rceil$. De plus, on peut facilement passer d'une décomposition en k pseudo-forêts

à une k-orientation, en orientant toutes les arêtes d'une même forêt vers la racine. Comme chaque nœud appartient à au plus k forêts, le degré sortant maximal sera k. Picquard et Queyranne [PQ82] proposent un algorithme pour calculer l'arboricité en $O(nm \log^3 n)$.

En s'autorisant une approximation constante de l'optimal, il est possible de réduire significativement le temps d'exécution. Dans [Kow06], Kowalik propose une $(1+\epsilon)$-approximation en temps $O(\frac{m \cdot \log_2 n}{\epsilon})$. Informellement, l'algorithme consiste à calculer un flot maximum en limitant la longueur des chemins augmentants à rechercher. L'auteur montre notamment que pour tout $k \geq \lceil (1+\epsilon)\Delta(G) \rceil \geq \lceil \Delta(G) \rceil$, l'algorithme va trouver un chemin augmentant de longueur au plus logarithmique.

2.3 Introduction à l'algorithmique distribué

Nous avons vu que les requêtes peuvent arriver sur les nœuds de manière *distribuée*. Intuitivement, chaque nœud peut recevoir des requêtes qui seront mises en attente et devra prendre la décision de l'allocation de ces requêtes à partir d'une connaissance partielle du système et notamment de la charge des autres nœuds. Toutefois les nœuds sont autorisés à *communiquer* à l'aide de message afin d'apprendre des informations sur le système et ainsi prendre une décision pour l'allocation. Les définitions et modèles qui suivent sont tirés de [Pel00].

2.3.1 Modélisation du réseau

Dans cette thèse nous considérons uniquement le mode de communication par passage de messages. On considère un réseau $\mathcal{N}$ modélisé par un graphe $G_{\mathcal{N}} = (V, E_{\mathcal{N}})$, avec $|V| = n$, où les sommets représentent les nœuds du réseau et les arêtes représentent les canaux (ou liens) de communication entre les nœuds. Sauf mention contraire, on considérera que les canaux sont bidirectionnels. On précise que le graphe de communication et le graphe des requêtes ne sont pas nécessairement confondus.

On associe à chaque nœud un identifiant unique u_i $(1 \leq i \leq n)$ tel que $u_i < u_{i+1}$. De plus, on associe à chaque canal de communication incident à un nœud u un numéro de port allant de 1 à $d_G(u)$. Une arête $\{u, v\}$ correspond donc à un couple $((u, i), (v, j))$ où i et j sont les numéros de ports respectifs de l'arête $\{u, v\}$ d'un point de vue local pour u et v. Pour communiquer avec v, le nœud u va donc envoyer un message via son port i, et v va recevoir le message via son port j. Par défaut, les messages sont reçus dans un ordre *First In First Out* (FIFO) *i.e.* dans l'ordre où ils ont été envoyés par le nœud émetteur.

2.3.2 Modèle de calcul distribué

Un *algorithme distribué* est un ensemble de protocoles exécutés de manière local par chaque nœud. Un *protocole* Π_i d'un nœud u_i peut être vu comme une machine

à état avec un ensemble d'états Q_i et un état initial $q_{0,i}$. L'algorithme distribué est une succession d'événements qui vont faire évoluer les protocoles locaux et vont changer l'état courant q_i de u_i. Un événement peut être l'*envoi* d'un message, la *réception* d'un message ou bien le résultat d'un *calcul local*. Une *configuration* $\mathcal{C} = \{q_1, \ldots, q_n\}$ correspond à l'ensemble des états des nœuds à un instant donné. On peut voir une configuration comme une *photographie* de l'état global du système à un instant donné. L'exécution d'un algorithme distribué peut donc être décrit par les configurations successives prises par le système. Afin d'être plus précis, on peut également associer une machine à état à chaque lien de communication et ainsi ajouter à la configuration l'ensemble des états courants de chaque lien. Cependant nous nous intéresserons uniquement à l'état des nœuds dans le reste de ce manuscrit.

2.3.3 Les modèles de communication

On rappelle que les définitions et modèles utilisés ici sont tirés de [Pel00]. On distingue deux modes d'exécution pour les algorithmes distribués : un mode d'exécution *synchrone* et un mode d'exécution *asynchrone*. Dans une exécution synchrone, tous les événements s'effectuent par *ronde d'exécution*. On garantit ainsi que tous les événements qui commencent à une ronde i seront terminés avant le début de la ronde $i + 1$. Dans une exécution asynchrone, l'unique contrainte est qu'un message envoyé par un nœud u via un canal $\{u, v\}$ sera reçu par v en un temps fini. Cependant, on ne fait pas d'hypothèse sur le délai que vont mettre ces messages à arriver. On définit ainsi deux modèles de communication qui dépendent notamment de ces modes d'exécution. Ces deux modèles seront principalement utilisés dans le reste du manuscrit (bien que certaines précisions pourront être apportées).

— Le modèle $\mathcal{LOCAL}$: L'exécution est synchrone et tous les nœuds se "réveillent" en même temps au début de chaque ronde afin d'exécuter leur protocole. De plus la taille des messages n'est pas limitée.
— Le modèle $\mathcal{ASYNC}$: L'exécution est asynchrone. Il n'y a pas de contrainte sur la taille des messages envoyés.

Dans les deux cas, on suppose que les nœuds ne tombent jamais en panne, que les identifiants des nœuds sont de taille $O(\log n)$ et que les calculs locaux sont gratuits (ou plus précisément sont négligeables vis-à-vis de la communication).

2.3.4 Mesures de performances

Les performances d'un algorithme distribué peuvent être évaluées selon plusieurs critères : le *temps d'exécution* de l'algorithme, la *mémoire* utilisée ainsi que le *nombre de messages* échangés.

L'expression du temps de complétion dépend du niveau de synchronisme de l'exécution de l'algorithme à analyser. Dans une exécution asynchrone, si l'on considère qu'un message arrive en une unité de temps, le temps d'exécution asynchrone correspond au nombre d'unité de temps qui se sont passées entre le début et la fin

de l'algorithme. Le temps d'exécution synchrone correspond aux nombres de rondes d'exécution avant la terminaison de l'algorithme.

La mémoire est mesurée par le nombre de bits utilisés par l'algorithme, soit localement vis-à-vis d'un nœud, soit globalement pour tout le système.

Enfin en plus du nombre de messages utilisés par l'algorithme, c'est-à-dire le nombre de messages qui transitent par les canaux de communication durant l'exécution, on peut aussi s'intéresser à la taille des messages utilisés.

Chapitre 3

Allocation de requêtes dans un environnement asynchrone avec pannes

Sommaire

Les systèmes de stockage distribués fonctionnent sur un mode de communication asynchrone, ce qui implique qu'il n'est a priori pas possible de savoir quand un message arrive à destination. On ne peut donc pas distinguer un nœud *en panne* d'un nœud lent à répondre, ce qui peut poser des problèmes majeurs. A titre d'exemple, le problème bien connu du consensus en algorithmique distribué ne peut pas être résolu lorsqu'un nœud est en panne [FLP85].

On peut également citer le cas des *réseaux de capteurs sans fils*. Un réseau de capteurs sans fils est un ensemble d'entités autonomes possédant des capteurs ainsi

qu'une capacité de calcul, de communication et énergétique qui leurs sont propres. Une des problématiques majeures dans ce type de système est de pouvoir limiter la consommation énergique afin d'améliorer la durée de vie des capteurs (on pourra se référer par exemple à [PNV13] pour un survol de la question). Les capteurs doivent, en autre, collecter des données liées à leur environnement, les traiter et les communiquer aux autres nœuds. Trouver une bonne allocation de ces tâches est donc une piste de recherche naturelle pour minimiser la consommation d'énergie. Le problème est que les capteurs sont éloignés géographiquement et instaurer un mode de communication synchrone semble difficile. D'autre part, un capteur est susceptible de tomber en panne, soit car ses ressources énergétiques sont justement épuisées, par exemple, soit car un événement extérieur a causé une défaillance du capteur.

Dans ce chapitre, nous considérons donc le cas spécifique où les communications se font de manière asynchrone et où certains nœuds du système peuvent être en panne. Nous étudions plus spécifiquement le problème équivalent de l'orientation de graphe en distribué. Après avoir fait un état de l'art et présenté quelques algorithmes simples d'orientation distribué (Section 3.1), nous présentons un algorithme d'orientation asynchrone tolérant aux pannes (Section 3.2). Nous prouvons notamment la correction de notre algorithme quelque soit le degré d'asynchronisme considéré. Nous donnons ensuite une borne inférieure dans le modèle LOCAL sur le nombre de rondes nécessaires pour obtenir une orientation optimale d'un graphe (Section 3.3).

3.1 Préliminaires

Dans le reste de ce chapitre, nous considérons que toutes les requêtes sont déjà présentes dans les files des nœuds candidats et que la décision d'affectation se fait de manière hors-ligne. On suppose ici que le nombre de candidat est égal à 2. Nous avons vu dans le Chapitre 2 que le problème d'allocation peut se réduire au problème d'orientation minimum dans les graphes. Nous adoptons dans ce chapitre cet angle d'attaque et étudions le problème de l'orientation de graphe en distribué. Nous considérons un mode de communication par passage de messages et nous supposons que le graphe sous-jacent au réseau de communication et le graphe des requêtes à orienter sont confondus.

3.1.1 Etat de l'art

Orientation distribuée Plusieurs solutions exactes ou approchées du problème de l'orientation minimum ont déjà été présentées dans le Chapitre 2 mais toutes étaient conçues dans un modèle centralisé [Kow06, GW92, AMOZ06]. Chrobak et Eppstein [CE91] s'intéressent à l'orientation des graphes planaires mais ne proposent que des algorithmes séquentielles et parallèles pour cette famille de graphe. On notera qu'ici aussi les algorithmes proposés sont optimaux. Dans le cas spécifiques

des graphes $2k$-réguliers, il est possible de trouver une k-orientation à partir d'un *cycle eulérien*. On rappelle qu'un cycle eulérien est un chemin partant d'un sommet u qui passe par toutes les arêtes du graphe et qui termine à u. Il suffit d'orienter les arêtes du chemin de manière sortante en commençant par u. On peut facilement montrer qu'un tel chemin passe exactement k fois par un même nœud, impliquant que tous les nœuds ont un degré sortant k avec une telle orientation. Il est possible de trouver un cycle eulérien dans un graphe de manière distribué et dans un modèle asynchrone [Mak97]. L'algorithme utilise un nombre linéaire de messages mais par contre ne tolère pas les pannes.

Cependant, on peut facilement imaginer un algorithme distribué optimal dans le modèle LOCAL. A chaque ronde, chaque nœud communique à ses voisins sa connaissance du graphe G (au début son voisinage). Après un nombre de rondes proportionnel à $diam(G)$, tous les nœuds ont une copie de G en local et peuvent ensuite appliquer un algorithme d'orientation optimal. L'inconvénient de cette méthode est que la taille des messages et la mémoire de chaque nœud sont proportionnelles à la taille du graphe, ce qui est énorme. Une solution plus réaliste est proposée par Barenboim et Elkin [BE10], qui s'intéressent aux problèmes du Minimum Independent Set (MIS) et de la coloration pour des graphes d'arboricité bornée. Pour résoudre ces problèmes, ils proposent des algorithmes basés sur une décomposition en forêts (arêtes-disjointes) induite par une *orientation* du graphe. Cette orientation est calculée en $O(\log n)$ rondes et correspond à une $(2+\epsilon)$-approximation du problème dans le cas où l'arboricité est connue à l'avance par les nœuds. Si ce paramètre n'est pas connu, l'approximation passe à $4+2\epsilon$ mais le nombre de ronde reste asymptotiquement inchangé. Ici, la taille des messages et de la mémoire additionnelle ne dépassent pas $O(\log arb(G))$ avec $arb(G)$ l'arboricité du graphe.

Les résultats que nous venons de présenter fonctionnent dans une exécution synchrone. Il est toutefois possible de les adapter dans un environnement asynchrone en utilisant un synchroniseur. L'inconvénient est que les synchroniseurs sont difficilement utilisables dans le cas des *pannes*. Considérons par exemple le problème du consensus. Supposons en effet qu'il existe un algorithme synchrone pour ce problème et qu'il existe un synchroniseur capable de simuler cet algorithme dans une exécution asynchrone et en présence de pannes. On obtiendrait alors un algorithme asynchrone tolérant aux pannes, ce qui a été démontré impossible dans [FLP85].

Orientation de réseau On peut également s'intéresser au problème similaire de *l'orientation de réseau*. Orienter un réseau consiste à associer un label à tous les canaux de communication de manière à induire un *sens* à ces canaux. On remarque qu'ici le problème est plus facile que le notre car on ne cherche pas à faire d'optimisation sur le degré sortant obtenu. Un résultat de Tel [Tel94] montre qu'il n'existe pas d'algorithme asynchrone et déterministe pour calculer une telle orientation si les nœuds ne possèdent pas d'identifiant unique ou s'il n'existe pas de nœud distingué. On remarque également que l'une des deux hypothèses implique l'autre. En effet, il est facile d'attribuer un identifiant à chaque nœud si on a un nœud distingué. Par

exemple, on peut effectuer un parcours de graphe depuis ce nœud et donner comme identifiant l'ordre de visite des nœuds pendant le parcours. De même, si tous les nœuds sont identifiés, on peut appliquer un algorithme d'élection afin de déterminer le nœud distingué, puis diffuser le résultat de l'élection à tout le monde. Toutefois, ces résultats supposent que les nœuds ne tombent pas en panne. Si c'est le cas, l'existence d'un leader ne peut plus être garanti. En effet, Fischer, Lynch et Paterson [FLP85] montrent que le consensus (et par conséquent l'élection) est impossible dans un système asynchrone avec un nœud en panne. Toutefois, cela devient possible si l'on suppose que les nœuds sont soit en panne dès le début de l'algorithme, soit ne tombent jamais en panne.

Estimation de la densité Comme nous avons déjà pu le voir dans le Chapitre 2, l'estimation de la densité maximale est un élément clé de l'orientation de graphe. Le calcul de fonctions d'agrégation sur des données distribuées comme la somme ou la moyenne a déjà été largement étudié (voir par exemple le survol [JBA15]). Dans notre cas, nous voulons calculer le degré moyen des nœuds du graphe. Il existe des algorithmes pour calculer la moyenne de n valeurs stockées de manière distribuée, basés uniquement sur des calculs locaux. Cependant ils nécessitent un nombre de rondes polynomial en n. Par exemple dans [LZ12], les nœuds échangent en permanence leur estimation du degré moyen et mettent à jour leur estimation après réception de celles de leurs voisins. Cet algorithme converge vers une moyenne globale mais nécessite $\Omega(n^3)$ messages. Dans [JMB05], les auteurs proposent un protocole épidémique d'agrégation qui permet d'estimer la moyenne des valeurs de chaque nœud. Leur méthode s'adapte à un contexte de nœuds en panne et plus généralement à la dynamicité des nœuds. L'analyse théorique porte sur une version séquentielle de l'algorithme et considère un mode de communication synchrone et l'accès à une primitive globale permettant de tirer aléatoirement et uniformément un nœud parmi l'ensemble des nœuds du réseau. Avec ces hypothèses simplificatrices, les auteurs montrent que le facteur de convergence de l'algorithme ne dépend pas de la taille du système. Ils présentent également un algorithme plus réaliste asynchrone ainsi que divers simulations sur des topologies de graphes variées.

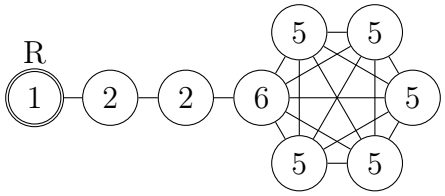
Conclusion de l'état de l'art Nous avons fait un survol de la littérature existante, tant sur l'orientation de graphes en distribué que sur l'estimation de paramètres globaux. Il en ressort qu'à notre connaissance, il n'existe pas d'algorithme asynchrone et tolérant aux pannes permettant de calculer une orientation de graphe. De plus, les adaptations directes d'algorithmes existants via l'utilisation de synchroniseurs ne sont pas utilisables si les nœuds peuvent tomber en panne.

3.1.2 L'algorithme `AvrDeg`

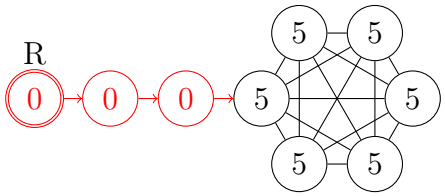
On peut également concevoir un algorithme approché en adaptant l'algorithme de Farach-Colton et Tsai [FT14] présenté dans le Chapitre 2. Nous appellerons cet

algorithme `AvrDeg` dans le reste du manuscrit. Cette adaptation utilise des messages de taille logarithmique et permet de limiter la mémoire et la capacité des nœuds, rendant le modèle plus réaliste et donc plus adapté à des contextes d'application réels. La racine diffuse la densité α du graphe G à orienter à tous les nœuds et attend un réponse de chacun. Lorsqu'un nœud u recoit cette densité, si $d(u) \leq (2+\epsilon)\alpha$ alors il s'active. Pour tout $v \in N(u)$, si v ne s'est pas activé alors l'arête $\{u, v\}$ est orienté de u vers v. Sinon un départage arbitraire est mis en place afin de déterminer qui va effectivement conserver l'arc sortant. Le nœud u informe ensuite la racine du nombre d'arcs conservés, sinon il envoie juste un accusé de réception. Lorsque la racine a reçu une réponse de tous les nœuds, elle calcule la densité du sous-graphe induit par les nœuds non activés et diffuse cette valeur. On réitère ainsi le processus tant que tous les nœuds ne se sont pas activés.

Afin d'illustrer cet algorithme, nous présentons son exécution sur le graphe ci-dessous que nous noterons G_0.



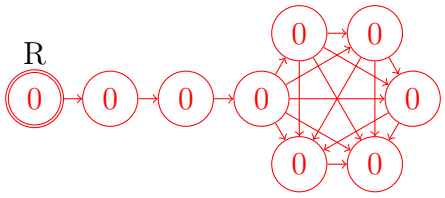
Ici le nœud R correspond à la racine et on fixe ϵ à $1/2$. Comme G_0 possède 18 arêtes et 9 nœuds, on a $\delta(G_0) = \lceil 18/9 \rceil = 2$, donc tous les nœuds avec un degré non orienté (indiqué à l'intérieur des sommets) inférieur à $5/2 \cdot 2 = 5$ vont s'activer. Après la diffusion de $\delta(G_0)$, tous les nœuds du chemin (en rouge sur la figure ci-dessous) vont donc s'activer les uns après les autres, en commencant par R.



On considère maintenant le sous-graphe induit par les sommets non activés (en blanc) que l'on note G_1. Ici le nombre d'arêtes est 15 et le nombre de nœuds est 6, donc $\delta(G_1) = \lceil 15/6 \rceil = 3$. Tous les nœuds avec un degré non orienté inférieur à $5/2 \cdot 3 = 15/2$ s'activent. Comme les nœuds restant ont tous un degré non orienté égal à $5 < 15/2$, ils vont tous s'activer et on choisit un départage arbitraire. Au final, l'orientation obtenue est la suivante.

Nous avons obtenu une 5-orientation du graphe car un des sommets de la clique a 5 arcs sortants. Il est toutefois possible d'améliorer cette orientation afin d'obtenir une 3-orientation. On a donc une 5/3-approximation de l'optimal.

Concernant le temps d'exécution, seulement 2 diffusions de la densité à tous les nœuds du graphe ont suffit afin de produire une orientation complète.



De manière plus générale, cet algorithme donnerait une $(2+\epsilon)$-approximation en $O(T_B \cdot \log_{2+\epsilon} n)$ rondes de communication, où T_B correspond au temps nécessaire à une diffusion. Cependant, comme nous avons pu le voir dans la Section 3.1.1, cette barrière de synchronisation ne peut pas fonctionner si les nœuds peuvent tomber en panne. En effet, il faut que la racine attende la réponse de chaque nœud avant de diffuser une nouvelle valeur de densité. Or si un nœud est en panne, cette contrainte se révèle bloquante pour l'algorithme.

3.1.3 Résumé du chapitre

Il ressort de notre état de l'art et de l'étude d'algorithmes simples adaptés de la littérature qu'il n'existe pas d'algorithme distribué asynchrone tolérant aux pannes garantissant une approximation constante de l'optimal. Notre première contribution présentée en Section 3.2 consiste donc à proposer un algorithme distribué que nous appelons `AvrDegAsync` avec les propriétés suivantes.

— **Asynchronisme.** Notre algorithme fonctionne de manière *totalement* asynchrone. Nous faisons toutefois l'hypothèse que les canaux de communications sont FIFO et qu'un message emprunte toujours le même chemin de routage entre 2 nœuds. Nous montrons que notre algorithme s'exécute en $O(\log \Delta(\log n + T_B))$ rondes asynchrones, où n, Δ et T_b correspondent respectivement au nombre de nœuds, à la densité maximum du graphe et au nombre de rondes pour diffuser un message à tous les nœuds.

— **Approximation.** Notre algorithme calcule une $2(2+\epsilon)$-approximation de l'optimal, avec $\epsilon > 0$. Cette performance est également proche des meilleures approximations existantes dans la littérature.

— **Connaissance locale.** Initialement, chaque nœud du réseau n'a aucune connaissance globale sur le graphe, comme le nombre de nœuds ou encore la densité maximum. Une estimation de ces paramètres va être réalisée par l'algorithme au cours de l'exécution.

— **Tolérance aux pannes.** Dernier point fort de notre proposition, l'algorithme calcule une orientation, même si certains nœuds sont en panne initialement, c'est-à-dire lors du lancement de l'algorithme.

Notre algorithme `AvrDegAsync` est une adaptation de l'algorithme `AvrDeg` présenté dans la section précédente.

Nous présentons également dans la Section 3.3 une **borne inférieure** pour le problème de l'orientation minimum dans un contexte distribué. Nous nous plaçons dans le modèle LOCAL et nous montrons qu'il est impossible de calculer une orien-

tation optimale **sans avoir une connaissance totale du graphe**. Cette borne inférieure vient compléter les bornes existantes présentées dans [BE10] qui lie l'approximation d'un algorithme distribué et son temps d'exécution.

3.2 Orientation asynchrone et tolérante aux pannes

Dans cette section nous présentons la principale contribution du chapitre, à savoir un algorithme d'orientation de graphe fonctionnant de manière distribuée, asynchrone et qui tolère un ensemble de pannes initiales, selon le modèle décrit dans [FLP85].

3.2.1 Voisinage

Nous utiliserons les notations $N(u)$ (resp. $N(S)$) pour définir le voisinage ouvert de u (resp. l'union des voisinages de tout sommet $u \in S$). Plus formellement, on a $N(u) = \{v \in V \mid \{u, v\} \in E\}$ (resp. $N(S) = \bigcup_{u \in S} N(u) \setminus S$). On définit de manière similaire le voisinage fermé $N[u] = N(u) \cup \{u\}$ (resp. $N[S] = N(S) \cup S$). Pour le voisinage sortant (resp. entrant) on utilise la notation N^+ (resp. N^-).

3.2.2 Représentation du graphe

Tout d'abord, rappelons le problème et quelques définitions. Soit $G = (V, E)$ un graphe simple avec n sommets et m arêtes. L'orientation de G consiste à assigner à toute arête $\{u, v\} \in E$ une orientation (informellement un *sens*). Soit $d^+(u)$ le degré sortant de u, c'est-à-dire le nombre d'arcs (u, v), et $D^+(G) = \max\{d^+(u) \mid u \in V\}$ le degré sortant maximum. Nous voulons trouver une orientation du graphe de départ de manière à minimiser le degré sortant maximum. Nous nous intéressons ici à une version modifiée du problème. Soit $\overrightarrow{G} = (V, \overrightarrow{E})$ un graphe bidirigé avec $V(\overrightarrow{G}) = V(G)$ et pour toute arête $\{u, v\} \in E(G)$, on a les arcs $(u, v) \in \overrightarrow{E}(\overrightarrow{G})$ et $(v, u) \in \overrightarrow{E}(\overrightarrow{G})$. Intuitivement, nous représentons chaque arête entre u et v par deux arcs dans les deux sen,s ce qui équivaut à considérer un arc *bidirigé*. On utilisera la notation $[u, v]$ pour définir un tel arc. Le but ici ne sera donc pas de trouver une orientation à des arêtes non orientées mais plutôt de choisir une direction à des arcs bidirigés. Plus formellement, un sous-graphe $\overrightarrow{H}$ est *valide* si pour toute paire de sommets u, v tel que $[u, v] \in \overrightarrow{E}(\overrightarrow{G})$, on a soit $(u, v) \in \overrightarrow{E}(\overrightarrow{H})$, soit $(v, u) \in \overrightarrow{E}(\overrightarrow{H})$ mais pas les deux en même temps. On considère ici que tout sous-graphe valide correspond à une orientation du graphe et un algorithme d'orientation est un algorithme qui prend en entrée un graphe bidirigé et qui produit en sortie un sous-graphe valide. Soit $\mathcal{A}$ un algorithme d'orientation, on note $D^+(\mathcal{A}(\overrightarrow{G}))$ le degré sortant maximum du sous-graphe valide produit par $\mathcal{A}$ avec le graphe $\overrightarrow{G}$ en entrée. Afin de simplifier les notations, on utilisera plutôt $D^+_{\mathcal{A}}(\overrightarrow{G})$. Le degré sortant maximum obtenu avec un algorithme d'orientation optimal sera noté $D^+_{opt}(\overrightarrow{G})$.

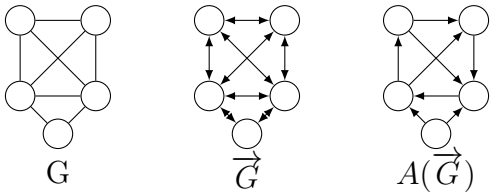


FIGURE 3.1 – Cette figure représente un graphe G non orienté, la version bidirigée de G ainsi qu'un sous-graphe valide produit par un algorithme d'orientation A avec pour degré sortant maximum 2.

Ce problème se décline naturellement dans un environnement *distribué*. En effet, le graphe $\overrightarrow{G}$ va représenter un réseau de communication où les nœuds correspondent aux sommets de $\overrightarrow{G}$ et les canaux de communication correspondent aux arcs de $\overrightarrow{G}$. On s'intéresse ici à concevoir un algorithme distribué dans le modèle par passage de messages qui calcule une orientation du graphe.

Une autre notion que l'on peut capturer avec ce modèle est la notion de *panne*. En effet, certains nœuds du réseau peuvent être en panne au lancement de l'algorithme d'orientation. Un nœud en panne est un nœud à qui il est possible d'envoyer des messages mais qui ne répond jamais. Dans ce cas, et ce quel que soit l'algorithme d'orientation utilisé, tous les arcs (u, v) avec u en panne ne seront pas conservés dans le sous-graphe valide retourné. Plus formellement, on considère un ensemble $F \subset V$ de nœuds en panne et un ensemble de nœuds corrects $C = V \backslash F$. On considère que les arcs de $G[F]$ sont inutilisables et on cherche donc à orienter le sous-graphe $\overrightarrow{G_F} = (V_F, \overrightarrow{E_F})$ défini par $V_F = N[C]$ et $\overrightarrow{E_F} = \{(u, v) \in \overrightarrow{E} \mid u \in C \vee v \in C\}$. On remarque que tout arc (u, v) avec u un nœud correct et v un nœud en panne sera conservé dans tout sous-graphe valide de $\overrightarrow{G}$. Le problème qui nous intéresse peut s'énoncer de la manière suivante.

Orientation Minimum avec pannes

- **Entrée** : un graphe bidirigé $\overrightarrow{G}$ avec un ensemble F de nœuds en panne.
- **Sortie** : un sous-graphe valide de $\overrightarrow{G}$ produit par un algorithme $\mathcal{A}$ tel que pour tout algorithme d'orientation $\mathcal{A}' \neq \mathcal{A}$ on a $D^+_{\mathcal{A}'}(\overrightarrow{G_F}) \geq D^+_{\mathcal{A}}(\overrightarrow{G_F})$.

Nous représentons dans la Figure 3.2 un graphe bidirigé sans pannes et le même graphe avec un ensemble de pannes.

Pour finir, nous adaptons quelques définitions présentées pour le cas des graphes non orientés dans le cadre des graphes bidirigés. On rappelle que pour tout graphe G non orienté, on a $\Delta(G) = \max_{H \subseteq G}\{\delta(H)\}$. On peut définir de la même manière la densité d'un graphe bidirigé $\overrightarrow{G} = (V, \overrightarrow{E})$ par $\delta(\overrightarrow{G}) = \frac{|\overrightarrow{E}|}{2|V|}$. Le degré sortant moyen de $\overrightarrow{G}$ est défini par $2\delta(\overrightarrow{G}) = \frac{|\overrightarrow{E}|}{|V|}$. Un sous-graphe $\overrightarrow{G}[V]$ induit par un ensemble de sommets V est défini par $\overrightarrow{E}(\overrightarrow{G}[V]) = \{[u, v] \mid u, v \in S \wedge [u, v] \in \overrightarrow{E}(\overrightarrow{G})\}$. Soit $S \subseteq V$

(a) graphe sans panne avec $D^+(\overrightarrow{G}) = 4$ et $D^+_{opt}(\overrightarrow{G}) = 2$

(b) graphe avec 2 pannes avec $D^+(\overrightarrow{G}_F) = D^+_{opt}(\overrightarrow{G}_F) = 3$

FIGURE 3.2 – Illustration de l'orientation avec ou sans pannes. Les nœuds en noirs sont en panne.

un ensemble de sommets de $\overrightarrow{G}$.

$$\overrightarrow{G} \backslash S = \overrightarrow{G}[V \backslash S]$$

La densité maximum $\Delta(\overrightarrow{G})$ d'un graphe bidirigé $\overrightarrow{G}$ est définie par

$$\Delta(\overrightarrow{G}) = \max_{\overrightarrow{H} \subseteq \overrightarrow{G}} \{\delta(\overrightarrow{H})\}$$

avec $\overrightarrow{H}$ un sous-graphe induit de $\overrightarrow{G}$. On fait l'observation suivante.

Observation 3.1. *Soit $G = (V, E)$ un graphe non orienté et $\overrightarrow{G} = (V, \overrightarrow{E})$ la version bidirigé de G.*

$$|\overrightarrow{E}| = 2|E|$$

On donne également quelques définitions et propriété sur la dégénérescence des graphes bidirigés qui seront utiles pour prouver certains résultats du chapitre.

Définition 3.1. Un graphe bidirigé est k-dégénéré s'il existe un ordre de dégénérescence des sommets $u_1, u_2, \ldots, u_n$ tel que pour tout i, $d^+_{\{u_{i+1},\ldots,u_n\}}(u_i) \leq k$.

Définition 3.2. La dégénérescence $deg(\overrightarrow{G})$ d'un graphe bidirigé $\overrightarrow{G}$ correspond au plus grand k tel que $\overrightarrow{G}$ est k-dégénéré.

Définition 3.3. Un k-cœur d'un graphe bidirigé $\overrightarrow{G}$ est un sous-graphe induit $H \subseteq \overrightarrow{G}$ tel que tous les sommets de H ont un degré sortant (dans H) au moins égal à k.

Le lemme suivant est une adaptation du Lemme 2.31 de [BE13] pour les graphes bidirigés.

Lemme 3.1. *Pour tout graphe bidirigé $\overrightarrow{G}$, $\Delta(\overrightarrow{G}) \leq deg(\overrightarrow{G})$*

Démonstration. Soit U un sous-ensemble quelconque de i nœuds. On ordonne les nœuds de l'ensemble en fonction de leur apparition dans l'ordre de dégénérescence du graphe. Notons $U = \{u_1, \ldots, u_i\}$ un tel ordre et on note $U_{>j} = \{u_{j+1}, \ldots, u_i\}$ l'ensemble des nœuds de U d'indice supérieur à j. On en déduit l'inégalité suivante

$$|E(\overrightarrow{G}[U])| = \sum_{j=1}^{i} 2d^{+}_{U_{>j}}(u_j) = 2\sum_{j=1}^{i} d^{+}_{U_{>j}}(u_j) \leq 2ki$$

ce qui nous permet de conclure que $|E(\overrightarrow{G}[U])|/2i \leq 2ki/2i = k$. On peut appliquer le même raisonnement sur tous les sous-graphes induit de $\overrightarrow{G}$, en particulier sur le sous-graphe de densité maximum. □

Lemme 3.2. *Il existe toujours un Δ-cœur pour tout graphe bidirigé $\overrightarrow{G}$.*

Démonstration. Supposons qu'il n'existe pas de Δ-cœur. On peut donc trouver un ordre de dégénérescence sur les sommets du graphe tel que tout sommet ait au plus $\Delta - 1$ arcs vers des sommets ayant un indice plus grand dans l'ordre. On en déduit que $deg(\overrightarrow{G}) < \Delta(\overrightarrow{G})$, ce qui contredit le Lemme 3.1. □

3.2.3 Modèle distribué

On considère un réseau de communication statique représenté par le graphe bidirigé connexe $\overrightarrow{G}$ à orienter avec n nœuds qui ont chacun un identifiant unique. Il existe un nœud distingué appelé la racine et noté R. On considère un mode de communication par passage de messages et on suppose que les canaux de communication sont FIFO (les messages sont reçus dans l'ordre d'émission) et qu'il n'y a pas de perte de message. Les communications sont asynchrones, mais pour l'analyse on supposera qu'un message est délivré en au plus une unité de temps. On définit une ronde de communication asynchrone de la manière suivante : soit t_i le temps où une ronde i commence et M_i l'ensemble des messages envoyés mais pas encore reçus au temps t_i. On considère qu'une nouvelle ronde $i+1$ ne peut pas commencer tant que les conditions suivantes ne sont pas remplies :

1. tous les messages de M_i ont été reçus
2. pour chaque message reçu au cours de la ronde, un nœud peut appliquer le traitement correspondant
3. un nœud peut envoyer (resp. recevoir) un message à tous ses voisins (de tous ses voisins)

Finalement, il y a des algorithmes de routage et de diffusion connus de tous les nœuds pour communiquer avec R. On pourra par exemple initier un parcours de graphe depuis la racine afin de construire un arbre de diffusion. Dans ce cas, chaque nœud possède un *père* qui lui permet de faire remonter de l'information vers la racine. On fait l'hypothèse suivante :

Hypothèse 3.1. Le chemin de routage entre deux nœuds u et v est toujours identique, quel que soit le message envoyé.

Nous ne faisons pas d'hypothèse supplémentaire sur ces algorithmes. Bien que le temps de diffusion et de routage peuvent être différents, on suppose afin de simplifier qu'ils sont identiques et on note T_B le nombre de rondes nécessaires pour chacune des deux opérations.

Enfin on définit la *terminaison* de notre algorithme comme le moment où les états des nœuds ne peuvent plus changer et qu'aucun message n'est envoyé. S'il n'y a pas de pannes, la terminaison de l'algorithme correspond au moment où tous les nœuds se sont activés.

3.2.4 Théorème principal

Nous proposons l'algorithme `AvrDegAsync` qui calcule un sous-graphe valide de $\overrightarrow{G}$. A notre connaissance, c'est le premier algorithme d'orientation de graphe fonctionnant de manière totalement asynchrone. De plus l'algorithme tolère les pannes initiales et ne nécessite pas la connaissance d'un paramètre global du graphe, comme le nombre de nœuds ou bien l'arboricité. L'algorithme est similaire à celui de [FT14] et consiste à *activer* des nœuds dont le degré est inférieur à un seuil proche de la *densité* du graphe, *estimée* puis diffusée à travers le réseau tout au long de l'algorithme par un nœud distingué appelé la *racine*. On rappelle que T_B correspond au nombre maximum de rondes nécessaires à une diffusion et on définit M_B comme le nombre maximum de messages échangés pour cette même opération. On prouve le théorème suivant :

Théorème 3.3. *Soit $\overrightarrow{G}$ un graphe bidirigé avec un ensemble de F pannes initiales et $\epsilon > 0$.*

Pour $|F| = 0$, `AvrDegAsync`$(\overrightarrow{G}, \epsilon)$ calcule une $(4+2\epsilon)$-approximation de l'orientation optimale du graphe $\overrightarrow{G}$.

Pour $|F| > 0$, après exécution de `AvrDegAsync`$(\overrightarrow{G}, \epsilon)$ on a $d^+(u) \leq \max\{(2+\epsilon)|F|, (4+2\epsilon)D^+_{opt}(\overrightarrow{G})\}$ pour tout nœud u.

Dans tous les cas, l'algorithme s'exécute en $O((T_B + \log_s n)\log_{1+\epsilon}\Delta(\overrightarrow{G}))$ rondes, avec $\Delta(\overrightarrow{G})$ la densité maximum du graphe et $s = \frac{2+\epsilon}{1+\epsilon}$.

La mémoire additionnelle utilisée par nœud est de taille $O(n)$ bits et l'algorithme nécessite l'échange de $O(m + M_B \log_{1+\epsilon} n)$ messages de taille $O(\log m)$ bits.

Dans la Section 3.2.5, nous décrivons en détails l'algorithme. Dans la Section 3.2.6 nous donnons les principaux lemmes nécessaires afin de prouver le théorème. Informellement, nous faisons dans un premier temps l'hypothèse qu'il n'y a pas de nœuds en panne. Nous montrons que la racine ne diffuse jamais de valeur supérieure à la densité maximum. Nous montrons également que l'algorithme termine toujours et diffuse une valeur proche de la densité, ce qui implique une terminaison rapide. Nous bornons le nombre de ronde nécessaire afin de diffuser cette valeur (Sous-section 3.2.6.2). Nous étudions ensuite le comportement de l'algorithme en présence de pannes (Sous-section 3.2.6.3).

3.2.5 Description de l'algorithme `AvrDegAsync`

3.2.5.1 Résumé de l'algorithme

Nous allons décrire les différentes actions qui peuvent être effectuées par les nœuds du système au cours de notre algorithme. Les détails sur ces actions sont donnés dans la sous-section 3.2.5.2 et nous résumons dans Tableau 3.1 les différents types de messages qui peuvent être échangés par les nœuds.

Structure de données Chaque nœud u maintient la structure de données suivante :

- une liste d'adjacence contenant l'identifiant de tous ses voisins dans $\overrightarrow{G}$
- une variable α_u correspondant à la dernière valeur de densité reçu par u. Initialement on fixe $\alpha_u \leftarrow 0$.
- une variable booléenne `active` qui permet de savoir si le nœud s'est déjà activé ou non
- une variable booléenne `identify` qui permet de savoir si le nœud s'est déjà identifié auprès de R

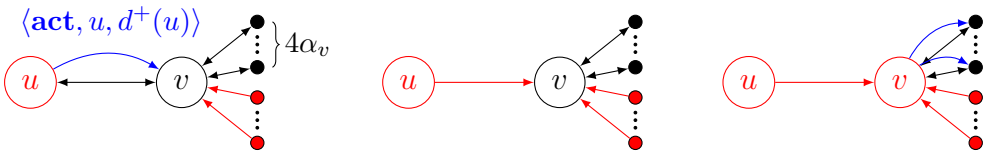
La racine R maintient la structure de données suivante :

- une variable α_R initialisée à 1 qui correspond à la dernière valeur diffusée
- une liste notée `recorded` qui va permettre de savoir quels sont les nœuds déjà activés
- une liste notée `identified` qui va garder en mémoire les nœuds déjà identifiés
- La racine maintient également un compteur m_R correspondant au nombre d'arcs pour lesquels une décision d'orientation n'a pas été prise

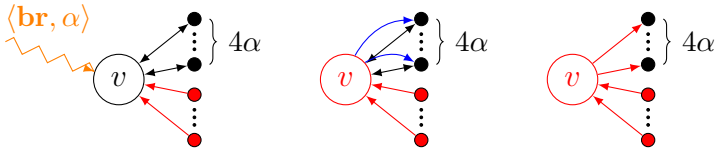
On note n_a la taille de la liste `recorded` et n la taille de la liste `identified`.

Identification des nœuds Au début de l'algorithme, la racine R ne connaît pas le nombre de nœuds du système, ni le nombre d'arcs du graphe. Elle va envoyer un premier message dit de *diffusion*, noté $\langle \mathbf{br}, \alpha_R \rangle$, avec $\alpha_R = 1$, à tous les nœuds de G. A noter que nous ne faisons pas d'hypothèse sur l'algorithme de diffusion utilisé. Lorsqu'un nœud u reçoit pour la première fois un message de diffusion, il envoie un message d'identification $\langle \mathbf{id}, u, N[u] \rangle$ à R. On remarque ici qu'en au plus $2T_B$ rondes, la racine connaît le nombre de nœuds qui ne sont pas en panne et le nombre d'arcs.

Diffusion et activation Lorsqu'un nœud u reçoit un message de diffusion $\langle \mathbf{br}, \alpha \rangle$, si $d^+(u) \leq 2(2+\epsilon)\alpha$, alors u s'*active*. Informellement, un nœud qui s'active va prendre en charge les doubles arcs qui lui sont adjacents. Il va envoyer un message d'activation $\langle \mathbf{act}, u, d^+(u) \rangle$ à la racine R ainsi qu'à chaque voisin $v \in N(u)$. Lorsqu'un voisin v reçoit le message d'activation est qu'il n'est pas lui même activé, il va supprimer l'arc (v, u) correspondant. Son degré sortant va donc diminuer et v va tester s'il est activable. Nous avons représenté sur la Figure 3.3 ces 2 scénarios d'activation.



(a) Activation après la réception d'un message d'activation. Le nœud v a initialement un degré sortant de $4\alpha + 1$. Après l'activation de u, son degré va diminuer d'une unité et v va donc s'activer.



(b) Activation après réception d'un message de diffusion. Le nœud v a un degré sortant suffisamment faible pour prendre en charge les arcs qui lui sont adjacents. Il s'active et envoie un message d'activation à chacun de ses voisins qui ne se sont pas encore activés.

FIGURE 3.3 – Deux scenario d'activation possibles. Ici on a choisi $\epsilon = 0$, donc le seuil d'activation est 4α et on représente en rouge les sommets déjà activés.

A la réception du message $\langle \mathbf{act}, u, d^+(u) \rangle$, R va mettre à jour le nombre d'arcs m_R restants non pris en charge par un nœud ainsi que la liste des nœuds activés. Lorsque la valeur $m_R/2(n - n_a)$ dépasse d'un certain seuil la valeur de α_R courante, alors α_R est mise à jour et cette valeur est diffusée à tous les nœuds. Selon certaines conditions (décrites dans le paragraphe suivant), u est ajouté à la liste `recorded`. On dit que u est *enregistré*.

On remarquera enfin que certains nœuds peuvent être enregistrés alors que d'autres n'ont pas encore été identifiés.

Conflits entre nœuds L'un des principaux problèmes liés à l'asynchronisme et aux pannes est qu'il n'est pas possible d'attendre une réponse d'un nœud avant d'effectuer une action locale. En effet on ne peut distinguer si un nœud est simplement lent à répondre ou bien s'il est défaillant. Supposons maintenant que deux nœuds voisins u et v s'activent à un intervalle de temps réduit. Plus précisément, le nœud u (resp. v) s'active avant d'avoir reçu le message d'activation de v (resp. u). Les deux nœuds vont donc vouloir prendre en charge l'arc bidirigé $[u, v]$ et conserver l'arc sortant qui leur est adjacent. On dit que ces deux nœuds sont *en conflit*. Le sous-graphe ainsi obtenu ne serait pas valide. Pour pallier ce problème, on choisit de manière arbitraire un des nœuds pour conserver l'arc. Dans notre cas, c'est le nœud de plus petit identifiant qui conserve l'arc correspondant. Toutefois, un message d'activation a été envoyé à la racine par chacun d'eux. Intuitivement, R va comptabiliser *deux fois* $[u, v]$ et donc sous-estimer le nombre d'arc restant. Si la densité est trop faible, cela peut stopper les activations des nœuds et donc l'algorithme lui-même. Supposons que v soit le nœud de plus petit identifiant. Afin de corriger cette estimation,

TYPE	DESCRIPTION
$\langle \mathbf{br}, \alpha \rangle$	estimation α de la densité diffusée par R
$\langle \mathbf{id}, N[u] \rangle$	identification du nœud u ainsi que de son degré sortant
$\langle \mathbf{act}, u, d^+(u) \rangle$	activation du nœud u ainsi que le nombre d'arc pris en charge
$\langle \mathbf{patch}, u, d^+(u) \rangle$	conflit avec le nœud u, degré sortant lors de l'activation de u

TABLE 3.1 – Récapitulatif des différents messages échangés par les nœuds lors de l'exécution de l'algorithme.

un message correctif $\langle \mathbf{patch}, u, d^+(u) \rangle$ va être envoyé par v à R afin de l'avertir du conflit entre u et v. Lorsque R reçoit $\langle \mathbf{patch}, u, d^+(u) \rangle$, il va d'abord vérifier si u est déjà enregistré, c'est à dire s'il est présent dans la liste `recorded`. Si ce n'est pas le cas, alors u est enregistré et R va déduire $2d^+(u) - 2$ de m_R, réglant ainsi le conflit. Le message d'activation envoyé par u ne sera pas pris en compte lors de sa réception. Par contre, si u est déjà enregistré, cela signifie que le double arc $[u, v]$ a déjà été comptabilisé 2 fois, m_R étant sous-estimée de 2 unités. Dans ce cas, R va additionner 2 à m_R.

3.2.5.2 Détails de l'algorithme

Tout d'abord, nous définissons dans Algorithme 7 la primitive `activation(u)` qui décrit les actions effectuées par un nœud u lors de son activation.

$active_u \leftarrow true$
envoyer à la racine R un message d'activation $\langle \mathbf{act}, u, d^+(u) \rangle$
envoyer à tout $v \in N(u)$ un message d'activation $\langle \mathbf{act}, u, d^+(u) \rangle$

Algorithme 7 – `activation(u)`

On suppose également que tous les nœuds connaissent les primitives `broadcast` et `forward` qui permettent de diffuser et de transférer un message vers le reste des nœuds ou vers la racine. Nous décrivons ensuite dans Algorithme 8 l'algorithme local exécuté par tout nœud u en fonction des messages reçus de la part de ses voisins. Le paramètre ϵ permet de déterminer le seuil d'activation et est identique pour tous les nœuds. On précise qu'en cas de conflit, c'est le nœud de plus petit identifiant qui va supprimer l'arc concerné. Nous décrivons les actions effectuées par R en fonction des messages reçus dans Algorithme 9.

3.2.6 Analyse de `AvrDegAsync`

On dit qu'un nœud u est *activable* vis-à-vis d'une estimation de densité α si $d^+(u) \leq 2(2 + \epsilon)\alpha$. De même, lorsque la racine R recoit un message d'activation $\langle \mathbf{act}, u, d^+(u) \rangle$, alors u est considéré comme *enregistré*.

pour tout message $\mathcal{M}$ reçu **faire**
  si $\mathcal{M} = \langle \mathbf{br}, \alpha \rangle$ **alors**
    si $identify = false$ **alors**
      envoyer $\langle \mathbf{id}, N(u) \rangle$ to R
      $identify \leftarrow true$
    si $\alpha_u < \alpha$ **et** $active = false$ **alors**
      exécuter `broadcast(`α`)`
      $\alpha_u \leftarrow \alpha$
      si $d^+(u) \leq 2(2+\epsilon) \cdot \alpha_u$ **alors**
        exécuter `activation(u)`
  si $\mathcal{M} = \langle \mathbf{act}, v, d^+(v) \rangle$ **alors**
    exécuter `forward(`$\langle \mathbf{act}, v, d^+(v) \rangle$`)`
    si $v \in N(u)$ **alors**
      si $active = false$ **alors**
        supprimer l'arc (u, v)
        si $d^+(u) \leq 2(2+\epsilon) \cdot \alpha$ **alors**
          exécuter `activation(u)`
      sinon
        si $u < v$ **alors**
          supprimer l'arc (u, v)
          envoyer $\langle \mathbf{patch}, v, d^+(v) \rangle$ à R
  si $\mathcal{M} = \langle \mathbf{patch}, v, d^+(v) \rangle$ **alors**
    exécuter `forward(`$\langle \mathbf{patch}, v, d^+(v) \rangle$`)`
  si $\mathcal{M} = \langle \mathbf{id}, N[v] \rangle$ **alors**
    exécuter `forward(`$\langle \mathbf{id}, N[v] \rangle$`)`

Algorithme 8 – Algorithme local exécuté par tout nœud u

```
pour tout message M reçu faire
    si M = ⟨id, N[u]⟩ alors
        pour tout nœud v ∈ N[u] faire
            si v ∉ identified alors
                identified.add(v)
        m_R ← m_R + d⁺(u)
    si M = ⟨act, u, d⁺(u)⟩ alors
        si u ∉ recorded alors
            recorded.add(u)
            m_R ← m_R − 2d⁺(u)
    si M = ⟨patch, u, d⁺(u)⟩ alors
        si u ∈ recorded alors
            m_R ← m_R + 2
        sinon
            recorded.add(u)
            m_R ← m_R − 2d⁺(u) + 2
si ⌈m_R/2(n − n_a)⌉ > (1 + ε) · α alors
    α ← ⌈m_R/2(n − n_a)⌉
    exécuter broadcast(⟨br, α, ⟩)
```

Algorithme 9 – Algorithme local exécuté uniquement par la racine R. On remarque que la réception d'un message de diffusion n'entraîne aucune action particulière.

3.2.6.1 Schéma de preuve

Nous présentons dans la suite de cette section les principaux lemmes permettant de déduire le Théorème 3.3. Afin de simplifier la présentation, nous présentons les résultats en deux parties.

Tout d'abord nous considérons qu'aucun nœud n'est en panne. Nous montrons dans un premier temps que la racine ne diffuse jamais de valeur supérieure à la densité maximum (*cf.* Lemme 3.4). Nous montrons ensuite que tous les nœuds sont activés en un nombre logarithmique de diffusion (*cf.* Lemme 3.9) ce qui nous permet de déduire l'énoncé du Théorème 3.3 dans le cas sans pannes.

Dans un deuxième temps, nous considérons un ensemble F de nœuds en panne. Nous montrons notamment que si un nœud s'active au cours de l'algorithme, son degré sortant ne dépassera pas l'optimal (*cf.* Lemme 3.12). Si à l'inverse un nœud correct ne s'active pas, nous montrons que son degré sortant sera proportionnel à $|F|$ (*cf.* Lemme 3.13). L'argument pour le temps d'exécution est identique que pour le cas sans panne.

Concernant la mémoire, l'énoncé découle directement de la structure de données où chaque nœud maintient un nombre constant de tableaux de n bits.

Enfin, le nombre de messages échangés se déduit du Lemme 3.8 sur le nombre de diffusion. De plus, un nombre constant de message est envoyé pour chaque double arc.

3.2.6.2 Analyse sans panne

Dans cette sous-section, nous étudions les performances de notre algorithme lorsque aucun nœud n'est en panne. Nous étudions dans la sous-section 3.2.6.3 la version avec un nombre f de pannes. Nous utilisons les notations $V_{rec,k}$ et $V_{id,k}$ pour décrire respectivement l'ensemble des nœuds enregistrés et l'ensemble des nœuds identifiés par la racine après la réception de k messages. On définit ainsi $\overrightarrow{G_k} = \overrightarrow{G}[V_{id,k} \backslash V_{rec,k}]$ le sous-graphe induit par les nœuds identifiés qui ne sont pas encore enregistrés.

Lemme 3.4. *L'algorithme* `AvrDegAsync(`$\overrightarrow{G}, \epsilon$`)` *ne diffuse jamais de valeur supérieure à* $\Delta(\overrightarrow{G})$.

Démonstration. On note $\mu_{i_1}, \ldots, \mu_{i_k}$ les k premiers messages reçus par R. Supposons dans un premier temps qu'il n'y a pas de conflits entre les nœuds de $V_{rec,k}$. On peut prouver par induction que pour tout nouveau message $\mu_{i_{k+1}}$ reçu, on a $\alpha_R \leq \delta(\overrightarrow{G_{k+1}})$. Les canaux de communication étant FIFO et le chemin de routage toujours identiques entre 2 nœuds, le premier message reçu par la racine est forcément un message d'identification $\langle \mathbf{id}, N[u] \rangle$ et il est clair que le nombre d'arcs m_R estimé par la racine sera inférieur à $|\overrightarrow{E}(\overrightarrow{G}[N[u]])|$. Supposons maintenant la propriété vraie, pour les k messages reçus. Si R reçoit un nouveau message $\langle \mathbf{id}, N[u] \rangle$, on montre par un argument similaire à celui de l'initialisation de la récurrence que

$\alpha_R \leq |\overrightarrow{E}(\overrightarrow{G_{k+1}})|$. Considérons le cas où $\mu_{k+1} = \langle \mathbf{act}, u, d^+(u) \rangle$, on remarque que certains arcs (v, u) ont pu ne pas être comptabilisés alors que la racine va quand même les considérer comme supprimés, impliquant une sous-estimation du nombre d'arcs. En effet, nous ne pouvons pas savoir si les messages d'identification des nœuds de $N(u)$ ont déjà été reçus. Supposons qu'il existe un nœud $v \in N(u)$ pour lequel la racine n'a pas reçu de message d'identification. Avant la réception de μ_{k+1}, on a que $|m_R| < \sum_{u \in V_{id,k}} d^+(u) - 2\sum_{u \in V_{rec,k}} d^+(u) = |\overrightarrow{E}(\overrightarrow{G_k})|$. On a donc après réception de μ_{k+1} que $|m_R| < |\overrightarrow{E}(\overrightarrow{G_k})| - 2d^+(u) \leq |\overrightarrow{E}(\overrightarrow{G_{k+1}})|$. Si tous les nœuds sont identifiés avant que l'un d'entre eux soit enregistré, alors on peut simplement vérifier que la racine va estimer exactement la densité $\delta(\overrightarrow{G_{k+1}})$ [1]. On peut donc conclure que quelque soit le type de message, on a $\alpha_R \leq \delta(\overrightarrow{G_{k+1}}) \leq \Delta(\overrightarrow{G})$.

Supposons maintenant qu'il existe un conflit entre deux nœuds $u_{i_a}, u_{i_b} \in V_{rec,k}$ et supposons sans perte de généralité que $u_{i_a} < u_{i_b}$. R va recevoir 3 messages en lien avec ce conflit : 2 messages d'activation $\langle \mathbf{act}, u_{i_a}, d^+(u_{i_a}) \rangle$ et $\langle \mathbf{act}, u_{i_b}, d^+(u_{i_b}) \rangle$ envoyés par u_{i_a} et u_{i_b} et un message correctif $\langle \mathbf{patch}, u_{i_b}, d^+(u_{i_b}) \rangle$ envoyé par u_{i_a}. Comme les canaux de communication sont FIFO et que le chemin de routage est toujours identique, le message correctif ne peut être reçu avant au moins un des deux messages d'activation. Supposons que $\langle \mathbf{act}, u_{i_a}, d^+(u_{i_a}) \rangle$ soit reçu en premier. On doit étudier 2 cas :

1. le message $\langle \mathbf{patch}, u_{i_b}, d^+(u_{i_b}) \rangle$ est reçu en deuxième par R. Dans ce cas, le conflit est réglé immédiatement et la racine ne déduit que $2d^+(u_{i_b}) - 2$ du nombre d'arcs m_R. La variable α_R a donc pour valeur $\delta(\overrightarrow{G} \backslash V_{rec,k})$.
2. le message $\langle \mathbf{act}, u_{i_b}, d^+(u_{i_b}) \rangle$ est reçu en deuxième par R. Dans ce cas, la racine va comptabiliser deux fois le double arc $[u_{i_a}, u_{i_b}]$, ce qui implique que $\alpha_R = \delta(\overrightarrow{G} \backslash \{u_{i_1}, \ldots, u_{i_a}, \ldots u_{i_b}\}) - 2$. Après la réception de $\langle \mathbf{patch}, u_{i_b}, d^+(u_{i_b}) \rangle$, on a $\alpha_R = \delta(\overrightarrow{G} \backslash V_{rec,k})$.

On applique le même raisonnement s'il existe plus de conflits entre les nœuds et on en déduit que $\alpha_R \leq \delta(\overrightarrow{G} \backslash V_{rec,j})$ pour tout $j \leq k$, ce qui implique que R ne diffuse jamais de valeur plus grande que $\Delta(\overrightarrow{G})$. □

Nous montrons maintenant que notre algorithme termine toujours puis nous déterminons le nombre de rondes nécessaires afin que calculer un sous-graphe valide.

Lemme 3.5. *Au cours de l'algorithme* `AvrDegAsync`$(\overrightarrow{G}, \epsilon)$*, tous les nœuds sont activés.*

Démonstration. On note $\mu_{i_1}, \ldots, \mu_{i_k}$ les k premiers messages reçus par R. Supposons qu'il n'y a plus de changement d'états sans qu'un sous-graphe valide ait été calculé et notons α la dernière valeur émise par la racine. Si l'algorithme s'arrête, cela

1. Cela revient à éplucher le graphe séquentiellement et à estimer la densité du sous-graphe restant.

signifie qu'aucun nœud ne peut s'activer, donc $d^+(u) > 2(2+\epsilon)\alpha$ pour tout $u \in \overrightarrow{G_k}$, ce qui implique que $\delta(\overrightarrow{G_k}) > \frac{n \cdot 2(2+\epsilon)\alpha}{2n} = (2+\epsilon)\alpha > (1+\epsilon)\alpha$. Comme la racine a enregistré tous les nœuds qui se sont activés, on a $\alpha = \delta(\overrightarrow{G_k}) > (1+\epsilon)\alpha$, ce qui implique une nouvelle diffusion par R. On rappelle que $2\delta(\overrightarrow{G_k})$ correspond au degré moyen du graphe, donc par définition il y a au moins un nœud u tel que $d^+(u) \leq 2\delta(\overrightarrow{G_k}) \leq 2(2+\epsilon)\delta(\overrightarrow{G_k})$. Il y a donc au moins un nœud activable après cette dernière diffusion, ce qui implique qu'il y a toujours des changements d'état à venir. □

Par soucis de simplicité, on utilise la notation Δ au lieu de $\Delta(\overrightarrow{G})$ dans le reste de cette section.

Lemme 3.6. *La racine diffuse une valeur $\alpha_R \geq \frac{\Delta}{2(2+\epsilon)}$.*

Démonstration. Supposons que la valeur maximum diffusée par la racine soit $\alpha_R < \frac{\Delta}{2(2+\epsilon)}$. Après la dernière activation, seuls les nœuds avec un degré sortant supérieur à $2(2+\epsilon)\frac{\Delta}{2(2+\epsilon)} = \Delta$ peuvent ne pas avoir été activés. Or d'après le Lemme 3.2, on sait qu'il existe un Δ-cœur donc il y a au moins un nœud non activé. Il y a donc contradiction avec le Lemme 3.5. □

Lemme 3.7. *Après la dernière diffusion de la racine, l'algorithme `AvrDegAsync`$(\overrightarrow{G}, \epsilon)$ termine en $O(\log_s n + T_B)$ rondes, avec $s = \frac{2+\epsilon}{1+\epsilon}$.*

Démonstration. Grâce au Lemme 3.6, nous savons que la dernière émission α_R est au moins égale à $\Delta/2(2+\epsilon)$. Tout d'abord, on remarque qu'en au plus T_B rondes, tous les nœuds du graphe ont appris la valeur α_R. On se place donc dans le cas où tous les nœuds connaissent déjà α_R. Nous allons montrer qu'à chaque ronde une proportion constante des nœuds va s'activer. On rappelle qu'un nœud u s'active si $d^+(u) \leq 2(2+\epsilon)\alpha_R$. On note G_i le sous-graphe induit par les nœuds non activés à la ronde i et n_i le nombre de sommets de G_i. Soit N_k le nombre de sommets de degré sortant k et δ_i la densité de G_i. On remarque que s'il n'y a pas de nouvelle diffusion, alors on a $\delta_i < \min\{(1+\epsilon)\alpha_R, \Delta\}$ pour tout i.

$$
\begin{aligned}
\delta_i &= \frac{\sum_{k \geq 1} k \cdot N_k}{2 \cdot n_i} \\
\delta_i &> \frac{\sum_{k \geq 2(2+\epsilon)\alpha_R} k \cdot N_k}{2 \cdot n_i} \\
\delta_i &> 2(2+\epsilon)\alpha_R \frac{\sum_{k \geq 2(2+\epsilon)\alpha_R} N_k}{2 \cdot n_i} \\
\delta_i &> (2+\epsilon)\alpha_R \frac{\sum_{k \geq 2(2+\epsilon)\alpha_R} N_k}{n_i}
\end{aligned}
$$

Nous allons étudier deux cas de figures : $\alpha_R \in [\Delta/2(2+\epsilon), \Delta/(1+\epsilon)[$ et $\alpha_R \geq \Delta/(1+\epsilon)$. Dans le premier cas, puisque par hypothèse il n'y a pas de nouvelle diffusion, on a que $\delta_i \leq (1+\epsilon)\alpha_R$. On en déduit l'inégalité suivante.

$$\begin{aligned}(1+\epsilon)\alpha_R &> (2+\epsilon)\alpha_R \frac{\sum_{k \geq 2(2+\epsilon)\alpha_R} N_k}{n_i} \\ \frac{1+\epsilon}{2+\epsilon} n_i &> \sum_{k \geq 2(2+\epsilon)\alpha_R} N_k\end{aligned}$$

Dans le second cas, on a que $\delta_i \leq \Delta$ et $\alpha_R \geq \Delta/(1+\epsilon)$. On en déduit l'inégalité suivante.

$$\begin{aligned}\Delta &> \frac{2+\epsilon}{1+\epsilon}\Delta \frac{\sum_{k \geq 2(2+\epsilon)\alpha_R} N_k}{n_i} \\ \frac{1+\epsilon}{2+\epsilon} n_i &> \sum_{k \geq 2(2+\epsilon)\alpha_R} N_k\end{aligned}$$

On en déduit que le nombre de sommets de degré sortant supérieur à $2(2+\epsilon)\alpha_R$ est strictement inférieur à $\frac{1+\epsilon}{2+\epsilon} n_i$, ce qui implique que le nombre de sommets de degré inférieur ou égal à $2(2+\epsilon)\alpha_R$ est supérieur ou égal à $\frac{n_i}{2+\epsilon}$. En appliquant un raisonnement similaire pour le cas $\alpha_R \geq \Delta/2$, on en arrive à la même conclusion. Par définition d'une ronde, tous les nœuds activables vont s'activer et une décision d'orientation sera prise pour tous les arcs bidirigés incidents. On obtient ainsi un nouveau sous-graphe sur lequel les arguments ci-dessus restent valables. □

Lemme 3.8. *La racine effectue au plus* $O(\log_{1+\epsilon} \Delta)$ *émissions.*

Démonstration. Grâce au Lemme 3.6, on sait que la dernière valeur émise vaut au moins $\Delta/2(2+\epsilon) \leq \Delta$. Comme une nouvelle diffusion a lieu si le ratio entre l'estimation courante de la densité et la variable α_R correspondant à la dernière valeur diffusée est supérieur à $1+\epsilon$, on en déduit facilement qu'il y a au plus $O(\log_{1+\epsilon} \Delta)$ émissions effectuées par la racine avant de diffuser la dernière valeur. □

Lemme 3.9. *L'algorithme* `AvrDegAsync(`$\overrightarrow{G}, \epsilon$`)` *termine en* $O((T_B + \log_s n) \log_{1+\epsilon} \Delta)$ *rondes, avec* $s = \frac{2+\epsilon}{1+\epsilon}$.

Démonstration. Grâce au Lemme 3.7, on sait que s'il n'y a pas de nouvelles diffusions, une portion constante des nœuds s'activent à chaque ronde et il faut au plus $\log_s n$ rondes pour que tous les nœuds s'activent. On en déduit qu'il y a au plus $2T_B + \log_s n$ rondes entre deux diffusions, $2T_B$ représentant le nombre de rondes

maximum pour que tous les nœuds reçoivent la dernière mesure de densité et soient enregistrés par la racine en cas d'activation. En combinant ce résultat avec le Lemme 3.8, on en déduit qu'il faut au plus $O((T_B + \log_s n) \log_{1+\epsilon} \Delta)$ rondes pour activer tous les nœuds. □

3.2.6.3 Analyse avec des nœuds en panne

Nous supposons maintenant qu'il y a un ensemble $F \subset V$ de nœuds en panne dès le début de l'algorithme et on note C l'ensemble des nœuds corrects. Comme les nœuds en panne ne peuvent pas s'activer, on s'intéresse donc à minimiser le degré sortant des nœuds corrects. Dans ce contexte, le sous-graphe produit par l'algorithme n'est donc pas nécessairement valide.

On rappelle que l'ensemble des nœuds en panne est noté F et l'ensemble des nœuds corrects C. On rappelle qu'on s'intéresse à l'orientation du sous-graphe $\overrightarrow{G_F} \subset \overrightarrow{G}$ (voir sous-section 3.2.2 pour la définition de ce graphe). Comme les nœuds dans $N(C) \cap F$ ne peuvent pas prendre en charge d'arcs, on commence par faire l'observation suivante. On note $N_F^+(u)$ le nombre d'arcs sortant de u vers des nœuds en panne.

Observation 3.2. *Soit $u \in C$ tel que $N(u) \cap F \neq \emptyset$. Quel que soit l'algorithme d'orientation utilisé, on a $d^+(u) \geq N_F^+(u)$.*

On montre le lemme suivant.

Lemme 3.10.

$$\left\lceil \Delta(\overrightarrow{G_F}) \right\rceil \leq \mathcal{D}^+_{opt}(\overrightarrow{G_F}) \leq D^+_{opt}(\overrightarrow{G}[C]) + \max_{u \in C}\{N_F^+(u)\}$$

Démonstration. Supposons que les nœuds en panne soient connus. Considérons l'algorithme suivant : on oriente optimalement le sous-graphe $\overrightarrow{G}[C]$ et on sait que cet optimal correspond à la densité maximum $\Delta(\overrightarrow{G}[C])$ de ce sous-graphe. Comme chaque sommet garde à sa charge tous les arcs sortants vers des nœuds fautifs, on déduit la borne supérieure. Considérons maintenant tout algorithme d'orientation $\mathcal{A}$ appliqué à $\overrightarrow{G_F}$ et notons $\overrightarrow{G_\mathcal{A}} = \mathcal{A}(\overrightarrow{G_F})$ le sous-graphe valide produit par $\mathcal{A}$. Par définition, on sait que pour tous u, v avec $u \in C$ et $v \in F$, on a $(u, v) \in \overrightarrow{E}(\overrightarrow{G_\mathcal{A}})$. Toutefois, si v pouvait s'activer et prendre en charge l'arc double $[u, v]$, on obtiendrait une meilleur orientation. Cette observation nous permet de déduire que l'orientation produite par $\mathcal{A}$ sur $\overrightarrow{G_F}$ ne peut pas être meilleure que l'orientation produite sur ce même graphe sans panne. Soit $\overrightarrow{G'}$ un tel graphe, on sait que $D^+_{opt}(\overrightarrow{G'}) = \left\lceil \Delta(\overrightarrow{G'}) \right\rceil = \left\lceil \Delta(\overrightarrow{G_F}) \right\rceil$. On en déduit que $\left\lceil \Delta(\overrightarrow{G_F}) \right\rceil = D^+_{opt}(\overrightarrow{G'}) \leq \mathcal{D}^+_{opt}(\overrightarrow{G_F})$. □

Lemme 3.11. *Au cours de l'exécution de `AvrDegAsync`$(\overrightarrow{G_F}, \epsilon)$, la valeur de la variable α_R ne dépasse jamais $\left\lceil \Delta(\overrightarrow{G_F}) \right\rceil$.*

Démonstration. Soit $\overrightarrow{G_k}$ le sous-graphe de $\overrightarrow{G}$ induit par les nœuds identifiés qui n'ont pas encore été activés après la réception de k messages par la racine. Supposons qu'il existe au moins un nœud v qui soit en panne, les arcs sortants de v ne seront donc jamais comptabilisés par la racine. Comme les nœuds en panne sont tout de même identifiés [2], la densité α estimée par R sera toujours inférieure à $\delta(\overrightarrow{G_k})$. On remarque que $\overrightarrow{G_k} \subseteq \overrightarrow{G'_F}$, donc α est toujours inférieure ou égale à $\left\lceil \Delta(\overrightarrow{G'_F}) \right\rceil$. □

Lemme 3.12. *Si un nœud $u \in C$ est activé au cours de l'exécution de* `AvrDegAsync`$(\overrightarrow{G_F}, \epsilon)$*, alors* $d^+(u) \leq (4\epsilon + 2)D^+_{opt}(\overrightarrow{G_F})$.

Démonstration. D'après le Lemme 3.11, on sait que R ne diffuse jamais de valeur supérieure à $\Delta(\overrightarrow{G_F})$ et grâce au Lemme 3.10 on sait que $\left\lceil \Delta(\overrightarrow{G_F}) \right\rceil \leq \mathcal{D}^+_{opt}(\overrightarrow{G_F})$. On en déduit facilement l'énoncé. □

Cependant certains nœuds peuvent ne pas être activés au cours de l'algorithme, et ce pour deux raisons. Evidemment un nœud en panne ne peut pas s'activer par définition et ne pourra pas prendre en charge ses arcs. Toutefois, il est également possible qu'un nœud correct ne s'active pas. En effet, R calcule et diffuse une sous-estimation de la densité de $\overrightarrow{G_F}$ parce que R ne peut distinguer si un nœud est fautif ou non. Si la densité estimée puis diffusée est trop faible, aucun nœud ne s'activera. Ce scénario peut notamment arriver si certains nœuds de fort degré ont un nombre important de voisins en panne. Nous montrons dans le lemme suivant que les nœuds corrects non activés lorsqu'il n'y a plus de changement d'états ont un degré sortant ne dépassant pas le nombre de fautes $|F|$, à une constante près.

Lemme 3.13. *Un nœud u qui n'est pas activé durant l'exécution de* `AvrDegAsync` *est soit en panne soit a un degré sortant inférieur où égal à $(1 + \epsilon)|F| + d^+_F(u)$ à la fin de l'algorithme, avec $|F|$ le nombre de pannes.*

Démonstration. Soit α_i la dernière valeur émise par R et α la dernière valeur estimée par R avant qu'il n'y ait plus de changement d'état. On sait que $\alpha \leq (1 + \epsilon)\alpha_i$. On va tout d'abord déterminer le nombre de nœuds activables vis-à-vis de α. Soit n_i le nombre de nœuds au total qui n'ont pas été activés.

2. On rappelle qu'un message d'identification d'un nœud u contient $N[u]$.

$$\begin{aligned}
\alpha &= \frac{\sum_{k\geq 1} k \cdot N_k}{2 \cdot n_i} \\
&> \frac{\sum_{k\geq 2(2+\epsilon)\alpha_i} k \cdot N_k}{2 \cdot n_i} \\
&> 2(2+\epsilon)\alpha_i \frac{\sum_{k\geq 2(2+\epsilon)\alpha_i} N_k}{2 \cdot n_i} \\
&> (2+\epsilon)\alpha_i \frac{\sum_{k\geq 2(2+\epsilon)\alpha_i} N_k}{n_i} \\
&> \frac{2+\epsilon}{1+\epsilon}\alpha \frac{\sum_{k\geq 2(2+\epsilon)\alpha_i} N_k}{n_i}
\end{aligned}$$

On a donc au plus $n_i \frac{1+\epsilon}{2+\epsilon}$ sommets non activables, ce qui implique qu'il y a au moins $n_i - n_i \frac{1+\epsilon}{2+\epsilon} = n_i/(2+\epsilon)$ sommets activables. On remarque maintenant que si aucun sommet ne s'active, cela signifie que tous les sommets activables sont fautifs. On en déduit que $|F| \geq n_i/(2+\epsilon)$ et que $|C| = n_i - |F| \leq (2+\epsilon) - |F| = (1+\epsilon)|F|$. Donc tout sommet u correct non activé a au plus $(1+\epsilon)|F| + d_F^+(u)$ arcs sortants. □

Enfin le temps de complétion du graphe avec des pannes est clairement inférieur à celui du même graphe sans pannes, donc le Lemme 3.9 reste valide dans ce contexte.

3.3 Borne inférieure de l'orientation minimale en distribué

Dans cette section, nous considérons un graphe G non orienté et nous présentons une borne inférieure pour le problème de l'orientation minimum dans le modèle LOCAL. Nous montrons notamment qu'il est impossible de trouver une orientation optimale en utilisant moins que $O(diam(G))$, où $diam(G)$ correspond au diamètre du graphe.

3.3.1 Définitions

La connaissance du graphe Dans le modèle LOCAL, la taille des messages n'est pas bornée et l'on considère que les nœuds ont une capacité de calcul et une mémoire illimitée. Au cours d'une ronde, un nœud u peut recevoir, envoyer des messages et effectuer des calculs locaux. Ainsi à chaque ronde, u va étendre sa *connaissance* de G. Plus formellement, soit $\mathcal{B}_t(u)$ la boule de rayon t correspondant au sous-graphe induit par les nœuds à distance inférieure ou égale à t de u. La vue $\mathcal{V}_t(u)$ d'un nœud u à la ronde t correspond à la boule $\mathcal{B}_t(u)$ de rayon t centré en u. A la ronde 0, on a $\mathcal{V}_0(u) = \mathcal{B}_0(u) = \{u\}$, puis à chaque ronde i on a $\mathcal{V}_i(u) = G[N[\mathcal{V}_{i-1}(u)]]$ de rayon i centré sur lui. On dit qu'un nœud u connaît le graphe à la ronde t si $\mathcal{V}_t(u) = G$.

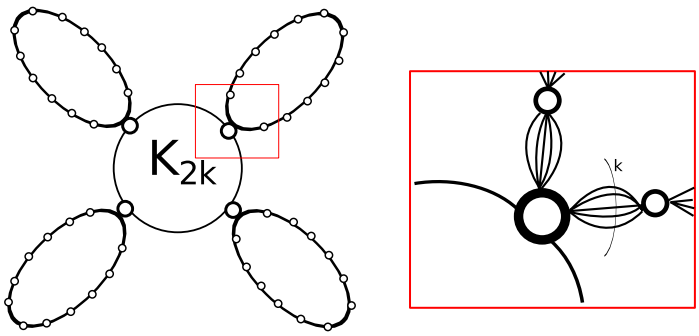


FIGURE 3.4 – Un graphe-fleur composé d'une clique de taille $2k$ avec des pétales d'épaisseur k

Le graphe fleur Un k-cycle de longueur ℓ est un graphe composé d'un ensemble de sommets $u_1, u_2, \ldots, u_\ell$ tel que pour tout $i \in [1, \ell - 1]$, il y a k arêtes entre u_i et u_{i+1}. De plus, il y a k arêtes entre u_ℓ et u_1. Un *graphe-fleur*, noté $F_{\ell,k}$, est un graphe composé d'une clique K_{2k} de taille $2k$ tel que pour tout sommet u de la clique, on a un k-cycle $(u_1 = u, u_2, \ldots, u_\ell)$ de longueur ℓ. Ce cycle est appelé *pétale* de base u. On dit que le pétale a épaisseur k et longueur ℓ (voir Figure 3.4).

Orientation cohérente On définit un algorithme d'orientation comme une collection de fonctions $g_{u_1}, g_{u_2}, \ldots, g_{u_n}$ tel que pour tout i, on définit g_{u_i} de la manière suivante :

$$\begin{array}{rccc} g_{u_i} : & E & \longrightarrow & \{0,1\} \\ & \{u_i, u_j\} & \longrightarrow & g_{u_i}(\{u_i, u_j\}) \end{array}$$

Un algorithme d'orientation est cohérent si pour toute arête $\{u, v\}$, on a $g_u(\{u, v\}) = 1 - g_v(\{u, v\})$.

3.3.2 Présentation de la borne inférieure

Nous allons montrer que l'orientation optimale n'est pas atteignable, même dans le modèle LOCAL, pour tout graphe, sans attendre un nombre de ronde égal au diamètre du graphe. Pour prouver cela, nous allons considérer que le réseau de communication est représenté par un graphe-fleur $F_{\ell,k}$ et on fait l'hypothèse forte que tout nœud du réseau sait qu'il appartient à un graphe de cette famille. On montre tout d'abord le lemme 3.14 sur l'orientabilité d'un tel graphe.

Lemme 3.14. *Soit $F_{\ell,k}$ un graphe-fleur avec une clique de taille $2k$ et dont les pétales sont d'épaisseur k et de longueur ℓ. On choisit k et ℓ tel que $k/\ell \in \mathbb{N}$ et k/ℓ soit pair.*

$$D^+_{opt}(F_{\ell,k}) = k + k/\ell$$

Démonstration. On remarque facilement qu'au moins un des nœuds de la clique doit prendre en charge k arêtes, quel que soit l'algorithme d'orientation utilisé. Nous notons u ce nœud et on considère le pétale $P_u = (u_1 = u, u_2, \ldots, u_\ell)$ de base u. Le nombre d'arêtes de P_u est $k\ell$ et on sait que u devra prendre en charge k arêtes. On en déduit que le nombre d'arêtes qui devront être prises en charge par des nœuds de P est $k + k\ell$. Le degré sortant moyen des nœuds de P est donc $\frac{k+k\ell}{\ell} = k + k/\ell$, ce qui nous permet de déduire qu'il existe au moins un nœud du pétale avec un degré sortant au moins $k+k/\ell$. On va donner une orientation permettant d'atteindre cette borne. On note $d^+_v(u)$ le nombre d'arc sortant de u vers v.

— $d^+_{u_2}(u_1) = d^+_{u_\ell}(u_1) = k/2\ell$
— $d^+_{u_{i+1}}(u_i) = \frac{1+2(i-1)}{2}\frac{k}{\ell}$ et $d^+_{u_i}(u_{i+1}) = k - d^+_{u_{i+1}}(u_i)$ pour $i < \lfloor \ell/2 \rfloor$
— $d^+_{u_i}(u_{i+1}) = \frac{1+2(i-\lfloor \ell/2 \rfloor -1)}{2}\frac{k}{\ell}$ et $d^+_{u_{i+1}}(u_i) = k - d^+_{u_i}(u_{i+1})$ pour $i > \lceil \ell/2 \rceil$
— si ℓ est impair, alors $d^+_{u_{\lfloor \ell/2 \rfloor}}(u_{\lceil \ell/2 \rceil}) = d^+_{u_{\lceil \ell/2 \rceil}}(u_{\lfloor \ell/2 \rfloor}) = k/2$.

Pour tout $i < \lfloor \ell/2 \rfloor$, on a

$$\begin{aligned} d^+(u_i) &= d^+_{u_{i+1}}(u_i) + d^+_{u_{i-1}}(u_i) \\ &= d^+_{u_{i+1}}(u_i) + k - d^+_{u_i}(u_{i-1}) \\ &= \frac{1+2(i-1)}{2}\frac{k}{\ell} + k - \frac{1+2(i-2)}{2}\frac{k}{\ell} \\ &= k + \frac{k(1+2(i-1) - (1+2(i-2)))}{2\ell} \\ &= k + \frac{2k}{2\ell} = k + k/\ell \end{aligned}$$

Pour tout $i > \lceil \ell/2 \rceil$, on a

$$\begin{aligned} d^+(u_i) &= d^+_{u_{i+1}}(u_i) + d^+_{u_{i-1}}(u_i) \\ &= k - d^+_{u_i}(u_{i+1}) + d^+_{u_{i-1}}(u_i) \\ &= k - \frac{1+2(\ell-i-1)}{2}\frac{k}{\ell} + \frac{1+2(\ell-(i-1)-1)}{2}\frac{k}{\ell} \\ &= k + \frac{k(1+2(\ell-i) - (1+2(\ell-i-1)))}{2\ell} \\ &= k + \frac{2k}{2\ell} = k + \frac{k}{\ell} \end{aligned}$$

Si ℓ est impair, on a $d^+(u_{\lceil \ell/2 \rceil}) = k/2 + \frac{\lfloor \ell \rfloor}{4}\frac{k}{\ell} \leq k + \frac{k}{2\ell}$. Enfin si ℓ est pair, on a $d^+(u_{\ell/2}) = 2k - 2(\frac{1+2(\ell/2-1)}{2}\frac{k}{\ell}) = 2k - \frac{k(\ell-1)}{\ell} = k + k(1 + \frac{\ell-1}{\ell}) = k + k/\ell$ □

Si un nœud connait le paramètre ℓ, il sera en mesure de choisir de manière locale la meilleur orientation. Toutefois, on rappelle que dans le modèle que nous utilisons, les nœuds ont une connaissance partielle du graphe qui va toutefois augmenter à

chaque ronde de communication. A chaque ronde t, un nœud possède donc une sous-estimation de la valeur de ℓ qui correspond à $2t$. Il faut donc $\lfloor \ell/2 \rfloor$ rondes pour que tous les nœuds connaissent la valeur de ℓ.

On va considérer tous les algorithmes cohérents utilisant au plus $t < \lfloor \ell/2 \rfloor$ rondes de communication, autrement dit aucun nœud ne connaît la valeur de ℓ. Le cœur de la preuve consiste à remarquer que les décisions prises par tout nœud dépendent uniquement de sa vue, et ces décisions seront identiques pour deux pétales de taille différentes supérieures à ℓ. On ne pourra donc pas orienter optimalement ces 2 graphes en prenant les même décisions. Nous allons donc montrer qu'il existe toujours un graphe tel que pour tout algorithme d'orientation, l'orientation optimale n'est pas atteinte pour ce graphe.

Lemme 3.15. *Soit $\mathcal{A}$ tout algorithme d'orientation cohérent utilisant t rondes de communication. Il existe un graphe $F_{\ell,k}$ avec $\ell \in [2t+1, 4t]$ tel que*

$$D^+_{\mathcal{A}}(F_{\ell,k}) \geq c \cdot D^+_{opt}(F_{\ell,k})$$

où c est un facteur d'approximation strictement supérieur à 1.

Démonstration. Soit l'intervalle $I = [0, 2k]$. On sait qu'il existe un pétale de base u tel que u doit prendre en charge k arêtes de la clique. On s'intéresse ici uniquement aux décisions prises par ce nœud u et on note $i \in I$ le nombre d'arêtes du pétale prises en charge par u. On remarque tout d'abord que $D^+(G) = \max\{k+i, \frac{k\ell-i}{\ell-i}\}$ où $k+i$ correspond au degré sortant de u et $\frac{k\ell-i}{\ell-i}$ correspond au degré sortant d'au moins un nœud du pétale. On peut facilement vérifier que si $k/\ell > i$, alors le nœud de plus haut degré est un nœud du pétale. A l'inverse, si $k/\ell \leq i$ alors c'est u qui a le plus haut degré. Cependant on sait que u ignore la valeur exacte de ℓ. On note $i_{opt} = k/\ell$ le choix optimal à faire pour i, et on va montrer qu'il est toujours possible de choisir $i_{opt} \in [k/4t, k/(2t+1)]$ afin de maximiser l'approximation donnée par $\max\{\frac{k+i}{k+i_{opt}}, \frac{k+i_{opt}}{k+i}\}$. Si $i \geq i_{opt}$ alors $\frac{k+i}{k+i_{opt}}$ est maximisé pour $i_{opt} = k/4t$ et si $i < i_{opt}$ alors $\frac{k+i_{opt}}{k+i}$ est maximisé pour $i_{opt} = k/(2t+1)$. On en déduit donc que l'approximation maximale c_i obtenue en fonction d'un choix i est

$$c_i = \max\left\{\frac{k+i}{k+k/4t}, \frac{k+k/(2t+1)}{k+i}\right\}$$

On en déduit que l'approximation obtenue C, quelque soit l'algorithme utilisé, est $c \geq \min_{i \in I}\{c_i\}$ et on peut facilement vérifier que $c > 1$. □

En choisissant le cas spécifique des algorithmes utilisant 1 ronde de communication, on obtient le corollaire suivant.

Corollaire 3.16. *Soit $\mathcal{A}$ tout algorithme d'orientation cohérent utilisant 1 ronde de communication. Il existe un graphe $F_{\ell,k}$ avec $\ell \in [3, 4]$ tel que*

$$D^+_{\mathcal{A}}(F_{\ell,k}) \geq (16/15) D^+_{opt}(F_{\ell,k})$$

Démonstration. On rappelle que l'approximation obtenue pour tout choix i de la base est $c_i = \max\left\{\frac{k+i}{k+k/4t}, \frac{k+k/(2t+1)}{k+i}\right\}$. On a $t = 1$ donc $c_i = \max\left\{\frac{k+i}{k+k/4}, \frac{k+k/3}{k+i}\right\} = \max\left\{(k+i)\frac{4}{5k}, \frac{4k}{3(k+i)}\right\}$. Les deux choix possibles pour i sont $(4/3)k$ et $(5/4)k$ et dans les 2 cas on a $c_i = \frac{16}{15}$. □

3.4 Conclusion du chapitre

Nous avons présenté un algorithme distribué et asynchrone d'orientation de graphe. Cet algorithme est également tolérant aux pannes initiales et ne nécessite pas de connaissance globale sur le graphe. Cet algorithme fournit une $2(2+\epsilon)$-approximation de l'orientation, ce qui est comparable aux meilleurs algorithmes synchrones connus, et nécessite $O(\log n(1 + diam(G)))$ rondes de communication. Nous montrons également qu'il est impossible d'obtenir une orientation optimale en utilisant moins de $O(diam(G))$ rondes de communication. Cependant plusieurs questions restent en suspens et constituent selon nous des pistes de recherches intéressantes.

Question 3.1. *Peut-on améliorer les performances et se rapprocher d'un temps d'exécution proche du diamètre ?*

Grâce au lemme 3.7, nous savons que l'orientation se terminerait rapidement après la diffusion de la densité maximale Δ. De plus, on sait que cette densité est émise en $O(\log_{1+\epsilon}\Delta) = O(\frac{\log\Delta}{\log(1+\epsilon)})$ rondes. On remarque donc que dans le cas où cette densité est constante, le nombre de rondes nécessaires à notre algorithme passerait à $O(\log_{\frac{2+\epsilon}{1+\epsilon}} n + T_B)$. En choisissant une structure d'arbre couvrant pour le protocole de diffusion, on obtient que T_B est de l'ordre de $diam(\overrightarrow{G})$, ce qui implique que l'algorithme nécessiterait au plus $O(\log_{\frac{2+\epsilon}{1+\epsilon}} n + diam(\overrightarrow{G}))$. Si $diam(\overrightarrow{G}) \geq \log_{\frac{2+\epsilon}{1+\epsilon}} n$, alors cette performance est asymptotiquement optimale. Obtenir une telle garantie pour tout graphe serait un résultat fort. Une piste consiste à estimer efficacement la densité maximale puis à diffuser cette valeur à tous les nœuds.

Notre algorithme pourrait également trouver une application pour le calcul de la dégénérescence de graphe. Un algorithme intéressant est proposé dans [MPM11] qui permet de calculer le *paramètre de cœur* pour les nœuds du graphe. On rappelle qu'un k-cœur est un sous-graphe dont tous les sommets ont un degré au moins k et le paramètre de cœur d'un sommet u est le plus grand k tel que u appartienne au k-cœur mais pas au $(k+1)$-cœur. Les nœuds vont estimer de manière locale leur paramètre de cœur en se basant sur l'estimation de leur voisin. Cet algorithme a cependant une complexité linéaire dans le pire des cas. Notre algorithme permettrait donc aux nœuds d'avoir rapidement une bonne estimation de la densité du graphe et donc de sa dégénérescence, accélérant ainsi l'estimation du paramètre de cœur.

Chapitre 4

Impact de la réplication sur le temps de complétion

Sommaire

Dans le chapitre 3, nous nous sommes intéressés plus particulièrement à la version distribuée du problème d'allocation en proposant un algorithme distribué asynchrone avec une approximation constante de l'optimal lorsque le facteur de réplication est 2 pour toutes les requêtes. Toutefois, un temps de complétion optimal ou quasi-optimal ne garantit pas forcément un traitement rapide des requêtes. En effet, considérons que les requêtes sont émises par un *adversaire* qui a une connaissance complète du système. Par connaissance complète, il faut comprendre que l'adversaire connaît le placement des objets ce qui implique qu'il a connaissance des candidats pour toutes requêtes susceptibles d'être reçues par le système. Un mauvais scénario dans ce contexte consiste donc à faire uniquement des requêtes ayant le même ensemble

de candidats. Clairement, les seuls noeuds pouvant traiter les requêtes sont ces candidats, ce qui conduit à un temps de complétion de $\Omega(m/f)$ où m correspond au nombre de requêtes reçues et f au nombre de candidats pour chaque requête.

Dans ce chapitre nous étudions plus précisément l'impact du facteur de réplication, que nous notons f, sur le temps de complétion. Dans la Section 4.2, nous donnons une borne supérieure, pour un f donné, sur le temps de complétion optimal lorsque le placement d'objet est aléatoire et que les objets ne sont pas trop demandés. Nous donnons également une borne inférieure pour tout placement qui est égale à notre borne supérieure à une constante multiplicative près.

Dans la Section 4.3, nous proposons un algorithme distribué qui, en plus d'allouer les requêtes reçues, adapte le nombre de candidats en fonction de la *popularité des requêtes*. Notre algorithme garantit une f-approximation de l'optimal en ajoutant uniquement un nombre linéaire de copies par rapport au nombre de nœuds du système (Section 4.4). L'hypothèse est que le nombre d'objets est nettement supérieur au nombre de nœuds du système. Ce surcout de copies est donc négligeable par rapport à une augmentation du nombre de copies pour tous les objets.

Les résultats de ce chapitre ont été présentés à la conférence BDA 2016 [GHL16].

4.1 Préliminaires

4.1.1 Adapter le facteur de réplication

L'impact du nombre de copies a été principalement étudié d'un point de vue théorique dans le modèle en-ligne lorsque que *le placement des objets et la distribution des requêtes sont aléatoires et uniformes.*

Pour $f = 1$, Gonnet [Gon81] montre que le nœud le plus chargé va recevoir $\Theta(\frac{\log n}{\log \log n})$ requêtes dans le cas spécifique $m = n$. Si $m < \frac{n}{\log n}$, Mitznmacher [Mit96] montre que la charge maximum sera $\Theta(\frac{\log n}{\log(n/m)})$. Raab et Steger [RS98] analysent la charge maximum pour tout m et montrent en particulier que si $m \gg n$ la charge maximum sera $\Theta(\frac{m}{n} + \sqrt{m \log n/n})$.

Pour $f > 1$, l'algorithme `Greedy` qui consiste à choisir le nœud candidat le moins chargé a été largement étudié. En particulier, il garantit que la charge maximum sera $\Theta(\frac{m}{n} + \frac{\log \log n}{\log f})$ [ABKU94, BCSV00, TW14]. Une variante de `Greedy` est proposée dans [CS97] avec des performances similaires. On peut également citer l'algorithme `AlwaysLeft`[Vöc99] qui garantit une charge maximum de $\Theta(\frac{\log \log n}{f \log \phi_f})$ avec $1.61 \leq \phi_f \leq 2$ dans le cas $m = n$ et de $\Theta(\frac{m}{n} + \frac{\log \log n}{f \log \phi_f})$ pour tout m [BCSV00, TW14].

Dans un modèle hors-ligne, un placement d'objet et une distribution de requêtes aléatoires uniformes reviennent à étudier le *graphe des requêtes aléatoire* sous-jacent.

Plus particulièrement, on retrouve plusieurs travaux sur l'étude des *pseudo-graphes aléatoires*. Un pseudo-graphe aléatoire est obtenu en choisissant pour chaque arête f nœuds de manière aléatoire, uniforme et indépendante. Chaque nouveau tirage se fait avec remise, donc un pseudo-graphe peut contenir des arêtes multiples et

des boucles. Dans [Cai06] (page 45), l'auteur montre que toute propriété vraie pour les pseudo-graphes aléatoires est aussi vraie (asymptotiquement presque sûrement) pour les graphes aléatoires uniformes $G_{n,m}$ [1] introduits dans [ER59]. La preuve est simple et se base sur un résultat de Chvátal qui dit qu'un pseudo-graphe aléatoire est simple avec probabilité (asymptotiquement) constante.

Czumaj et Stemann [CS97] proposent d'utiliser l'algorithme `KCoreOri(`G`,`k`)` (*cf.* Algorithme 2) pour l'orientation des pseudo-graphes aléatoires. En se basant sur une analyse fine du seuil d'apparition d'un k-coeur étudié dans [PSW96], les auteurs concluent que pour $m < 1,675459 \cdot n$, un pseudo-graphe aléatoire uniforme à n sommets et m arêtes est 2-orientable. Dans [SEK03], les auteurs montrent qu'avec grande probabilité, tout pseudo-graphe aléatoire uniforme ne contient pas de (L^*+1)-coeur, avec $L^* = \lceil m/n \rceil + 1$ impliquant qu'une L^*-orientation est possible. Enfin Cain, Sanders et Wormald proposent dans [CSW07] un algorithme linéaire nommé `SelfLess` et montrent qu'il produit une $\lceil m/n \rceil$-orientation ou une $(\lceil m/n \rceil + 1)$-orientation des pseudo-graphes aléatoires uniformes avec grande probabilité. Ce résultat est donc proche de *l'idéal* $\lceil m/n \rceil$ à une constante près. On trouve également plusieurs références pour $f \geq 2$ qui proposent une analyse fine du seuil d'apparition d'un $(k+1)$-cœur pour les hypergraphes aléatoires [Lel12, GW15, FKP16]. Cependant le calcul du seuil résulte de la résolution de systèmes d'équations différentielles complexes et rend difficilement exploitable ces résultats.

L'existant que nous avons présenté précédemment concerne des distributions de requêtes aléatoires et uniformes. De plus, le facteur de réplication était fixé.

Nous nous allons maintenant nous intéresser au modèle d'adversaire suivant.

Définition 4.1. Un *adversaire omniscient* connaît le placement de tous les objets et leur copies. Il choisit les requêtes qui seront reçues par le système.

Ce modèle fort d'adversaire correspond donc à une étude de *pire cas* et permet de déterminer la robustesse de nos algorithmes face à un tel comportement adversarial.

Dans le modèle en-ligne, Tang [Tan14] montre que pour une séquence de requêtes particulière, l'algorithme `Greedy` garantit un temps de complétion d'au moins $\Omega(\log_2 n)$ au lieu de 1 optimalement. Sans rentrer dans les détails, la séquence de requêtes présentée pour cette borne inférieure forme un *arbre binomial*. L'arbre binomial est une structure combinatoire défini récursivement de la manière suivante :

- un arbre de hauteur 1 est composé d'une unique arête dont l'une des deux extrémités est la racine de l'arbre.
- un arbre de hauteur i est l'union de 2 arbres de hauteur $i-1$ dont les racines sont reliées par une arête.

Nous avons illustré cette construction par la Figure 4.1. L'arbre binomial a notamment été utilisé pour concevoir des files de priorité [Vui78, Cha00], pour implémenter des tas de Fibonacci [BLT12] ou encore pour des structures d'arbres

1. Dans cette famille de graphe aléatoire, on choisit de manière aléatoire et uniforme un graphe à n sommets et m arêtes parmi l'ensemble des graphes à n sommets et m arêtes

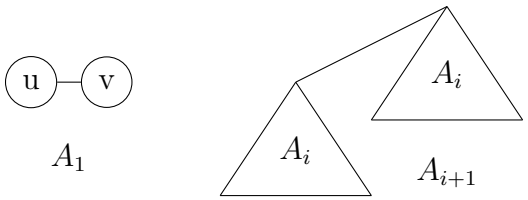


FIGURE 4.1 – Illustration de la construction d'un arbre binomial.

couvrants pour les applications parallèles de type hypercube [WFL98, WLK04]. En choisissant bien l'ordre d'arrivé des requêtes, l'adversaire peut faire en sorte d'allouer à la racine de l'arbre toutes les requêtes reçues dans sa file, ce qui représente un nombre de requêtes logarithmique par rapport au nombre de nœuds. Pourtant il est facile de 1-orienter un arbre. Il suffit pour cela d'orienter toutes les arêtes vers la racine.

Dans un modèle hors-ligne, Liu *et al* [LWW03] étudient le problème d'allocation dans un modèle client/serveur et proposent un algorithme centralisé basé sur la recherche de chemins augmentants qui produit une allocation optimale en temps $O(nm \log n + n^2 log^{3/2} n)$. Dans [FT14], les auteurs généralisent leur résultat obtenu pour les graphes ($f = 2$) aux cas des hypergraphes et montrent que l'algorithme `AvrDeg` est une $(f+\epsilon)$-approximation de l'optimal. Ils montrent notamment l'intérêt de leur solution pour le problème de l'équilibrage de charge dans le modèle client-serveur présenté dans [LWW03]. Cependant aucun de ces algorithmes n'adapte le facteur de réplication et donc ne donne de garantie vis-à-vis de l'idéal.

Enfin dans [FLAK11], les auteurs se placent également sur le modèle client-serveur où les requêtes arrivent au serveur qui va ensuite répartir la charge. Ici aussi les données sont placées de manière aléatoire. Ils proposent de répliquer les objets populaires dans un cache, placé au niveau du serveur, afin de permettre à un client d'y accéder rapidement. Ils montrent qu'un cache de taille $O(n \log n)$ est suffisant pour se prémunir d'un comportement adversarial. Toutefois, ils considèrent un modèle d'adversaire plus faible qui ne connaît pas le placement des objets.

En résumé. Il n'existe pas à notre connaissance d'algorithmes distribués garantissant une approximation de l'idéal dans le modèle d'adversaire omniscient.

4.1.2 Hypothèses de travail

Dans le reste de ce chapitre, nous considérons un modèle hors-ligne où toutes les requêtes ont déjà été reçues par le système. On considère également que les requêtes sont permanentes.

On considère un système distribué composé d'un ensemble de n nœuds qui possèdent une file de requêtes en attente. On considère également un ensemble Q de requêtes à allouer, avec $m = |Q|$, et l'ensemble O des objets stockés par le système.

De la même manière, on définit $\bar{m} = |O|$. Sauf mention contraire, le nombre de copies pour chaque objet est identique et sera noté f. On considère que les requêtes peuvent être choisies par un *adversaire omniscient*, qui connait à la fois Q mais également les candidats pour chaque requête. On distingue trois catégories de requêtes :

- Les requêtes sont dites *unitaires* si pour tout objet $o \in O$, il y a au plus une requête $q_o \in Q$.
- Les requêtes sont *non populaires* s'il existe au plus $\lceil m/n \rceil$ requêtes pour un même objet.
- Si aucune de ces conditions n'est remplie, on dit que les requêtes sont populaires.

4.1.3 Contribution et plan du chapitre

Supposons que $\bar{m} = n^\alpha$, avec $\alpha > 1$ une constante. Notre proposition consiste à utiliser un placement d'objet $\mathcal{P}$ *aléatoire uniforme*. Plus précisément, pour chaque objet o, on choisit aléatoirement un ensemble de f nœuds parmi l'ensemble des ensembles de nœuds de cardinalité f. Le choix du placement d'un objet est indépendant de celui des autres objets.

Notons $T(Q)$ le temps de complétion optimal avec ce placement d'objets. Les résultats présentés dans ce chapitre peuvent se résumer de la manière suivante.

- Nous donnons des bornes théoriques sur $T(Q)$. Nous prouvons notamment que $T(Q) = O(m/n \cdot (\bar{m}/n)^{1/f})$ pour tout ensemble de requêtes non populaires. Nous montrons également que le temps de complétion optimal $T^*(Q)$ est $\Omega(m/n \cdot (\bar{m}/n)^{1/f})$ pour un f donné en considérant tous les placements d'objets possibles, ce qui nous permet de déduire que $T(Q) = \Theta(m/n \cdot (\bar{m}/n)^{1/f})$. Toutefois, cet optimal peut être éloigné de l'idéal d'un facteur logarithmique près (*cf.* Corollaire 4.13).
- Nous donnons un algorithme distribué qui combine une adaptation de l'algorithme `AvrDeg` présenté dans le chapitre 3 et un algorithme de gestion de copies qui adapte le nombre de copies en fonction de la popularité d'une requête. A cela, nous ajoutons également un algorithme de tourniquet pour les requêtes populaires reçues. Nous montrons qu'en choisissant $f = (\alpha - 1) \log n$ et en créant un nombre linéaire de copies additionnelles, notre algorithme garantit un temps de complétion de $O(f \cdot m/n)$, quelle que soit la *distribution de requêtes* (*cf.* Théorème 4.18). Notre algorithme est non seulement une f-approximation du temps de complétion optimal mais également de **l'idéal**.
- Dans le cas spécifique des requêtes non populaire avec $\alpha = \log \log n$, nous montrons que $f = \log \log n$ suffit pour garantir un temps de complétion de $O(m/n \log \log n)$ (*cf.* Lemme 4.15).

Nos résultats théoriques se basent sur l'étude de *l'hypergraphe des requêtes* sous-jacent. En effet, comme nous avons pu le voir dans le chapitre 2 pour le cas $f = 2$, nous pouvons nous ramener à un problème d'orientation minimum et étudier

la *densité* de l'hypergraphe pour $f > 2$, qui est un élément clé pour résoudre ce problème. Les détails de cette analyse de densité sont donnés dans la section 4.2. Nous présentons ensuite plus en détails notre algorithme dans un modèle asynchrone par passage de messages en section 4.3. Enfin nous analysons cet algorithme dans un modèle synchrone en section 4.4.

4.2 Analyse du temps de complétion optimal

Dans cette section, nous présentons les hypergraphes du stockage et des requêtes qui modélisent respectivement le placement des objets et l'ensemble des requêtes reçues. Nous analysons la densité maximum d'un hypergraphe de requêtes et donnons ainsi un encadrement du temps de complétion optimal pour une ensemble de requêtes données.

4.2.1 Notations sur les hypergraphes

Soit $H = (V, E)$ un hypergraphe avec un ensemble de nœuds V et un ensemble d'hyperarêtes E. Un hypergraphe est *k-uniforme* si toutes ses hyperarêtes sont d'arité k. Une orientation de H consiste à désigner pour chaque hyperarête un sommet que l'on nomme la tête. Nous donnons dans la Figure 4.2 une illustration de l'orientation des hyperarêtes.

Le degré $d(u)$ d'un nœud u correspond au nombre d'hyperarêtes auxquelles il appartient et le degré sortant $d^+(u)$ correspond aux nombres d'hyperarêtes pour lesquelles il est la tête.

Une hyperarête $h = \{u_{i_1}, \dots u_{i_f}\}$ est complètement contenue dans un ensemble de nœuds S si $u_{i_j} \in S$ pour tout nœud $u_{i_j} \in h$. Le sous-graphe induit $H[S]$ est défini par l'ensemble des hyperarêtes de H complètement contenues dans S. La densité de H est définie par $\delta(H) = |E|/|V|$ et la *densité maximum* $\Delta(H)$ est définie par :

$$\Delta(H) = \max_{H' \in \mathcal{H}} \{\delta(H')\}$$

où $\mathcal{H}$ est l'ensemble de tous les sous-graphes induits de H.

Remarque 4.1. *La somme des degrés $\sum_{u \in V} d(u)$ des nœuds de H est égale à $k \cdot |E|$ si H est k-uniforme, ce qui implique que le degré moyen de H est égal à $k \cdot \lceil |E|/|V| \rceil$.*

Démonstration. C'est une simple adaptation du lemme des poignées de main au cas des hypergraphes. Une hyperarête est comptée k fois, c'est-à-dire une fois par chacune de ses extrémités. Au total, l'ensemble des arêtes sera compté k fois. Il suffit de diviser cette valeur par le nombre de nœuds pour obtenir la propriété sur le degré moyen. □

Pour finir, la multiplicité maximum $M(H)$ d'un graphe f-uniforme H correspond au nombre maximum d'hyperarêtes qui sont complètement contenues dans un ensemble de f nœuds.

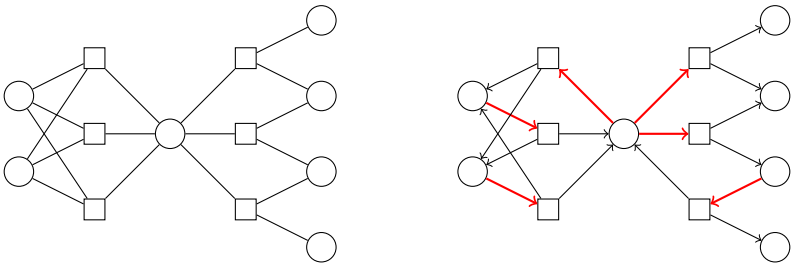


FIGURE 4.2 – Un hypergraphe de requête 3-uniforme. Les sommets dessinés avec un carré représentent les hyperarêtes alors que les nœuds du graphe sont représentés par un cercle. Sur le graphe de droite, on a représenté une orientation arbitraire.

4.2.2 Graphe du stockage et graphe des requêtes

L'hypergraphe du stockage $\bar{H}_{\mathcal{P}} = (V, E_O)$ est un graphe pour lequel l'ensemble V correspond aux nœuds du système et les arêtes de E_O symbolisent les objets de O placés selon le placement $\mathcal{P}$. Les extrémités d'une hyperarête h_o encodent les nœuds stockant une copie de o. Plus formellement, il y a une hyperarête $h_o = \{u_{i_1}, \dots u_{i_f}\} \in E(\bar{H}_C)$ pour tout objet o si $P(o) = \{u_{i_1}, \dots u_{i_f}\}$.

L'hypergraphe des requêtes $H_{\mathcal{P},Q} = (V, E_Q)$ est un hypergraphe ayant le même ensemble de nœuds V que $\bar{H}$ et dont l'ensemble E_Q des arêtes correspond aux requêtes de Q. Les extrémités d'une arête représentent les candidats pouvant traiter la requête, c'est-à-dire les nœuds stockant l'objet demandé. S'il y a une hyperarête $h_{q_o} = \{u_1, \dots, u_f\} \in E_Q$, cela implique que $\mathcal{P}(o) = \{u_1, \dots, u_f\}$. Pour le cas particulier des requêtes unitaires, on remarque que le graphe des requêtes associé est un sous-graphe de l'hypergraphe du stockage.

Si cela n'induit pas d'ambiguïté, nous utiliserons les notations $\bar{H}$ pour le graphe du stockage et H_Q pour le graphe des requêtes associé, correspondant à l'ensemble de requêtes Q.

L'orientation minimum d'hypergraphes Nous pouvons modéliser l'allocation d'une requête q par l'orientation de l'hyperarête h_q correspondante. Une orientation d'un hypergraphe H_Q revient donc à trouver une allocation des requêtes de Q. On note $D^+_{\mathcal{A}}(H)$ le degré sortant maximum avec l'orientation $\mathcal{A}$. Le problème de l'orientation minimum consiste à trouver une orientation qui minimise $D^+(H)$.

On peut relier la densité maximum et l'orientation minimum en utilisant des arguments similaires à ceux utilisés pour le cas $f = 2$ (*cf.* Théorème 2.2). Tout d'abord on doit adapter certaines définitions au cas $f > 2$. Un chemin dirigé est une séquence $u_1 - h_1 - u_2 - h_2 - \ldots - h_{k-1} - u_k$ où tout nœud u_i est la tête de l'hyperarête h_i pour tout $i \in [1, k]$. Un chemin augmentant est un chemin dirigé pour lequel $d^+(u_1) > d^+(u_k) + 1$.

Théorème 4.1. *Soit H un hypergraphe f-uniforme avec $f \geq 2$.*

$$D^+_{opt}(H) = \lceil \Delta(H) \rceil$$

Démonstration. ($\leq$) On commence par orienter l'hypergraphe de manière arbitraire puis on applique l'algorithme suivant. Tant qu'il existe un chemin augmentant $P = u_1 - h_1 - u_2 - h_2 - \ldots - h_{k-1} - u_k$ avec $d^-(u_1) > \lceil \Delta \rceil$, on inverse l'orientation de P, c'est-à-dire que pour toute hyperarête $h_i \in P$ avec $1 \leq i \leq k-1$, on change la tête de h_i, qui va devenir u_{i+1} au lieu de u_i. Nous allons prouver qu'un tel chemin existe toujours tant qu'il y a un nœud de degré sortant au moins $\lceil \Delta \rceil + 1$. Soit u un nœud tel que $d^+(u) > \lceil \Delta \rceil$ et $H' = (V', E')$ un sous-graphe induit par tous les chemins dirigés partant de u. Le nombre d'arêtes dans H' est $\sum_{v \in V'} d^+(v)$ pour tout $v \in V'$. Supposons maintenant qu'aucun de ces chemins dirigés ne soit augmentant, alors pour tout $v \in V'$ on a $d^+(v) \geq \lceil \Delta \rceil$, par conséquent $|E'| > |V'| \cdot \lceil \Delta \rceil \leftrightarrow |E'|/|V'| > \lceil \Delta \rceil$ ce qui est impossible par définition de Δ.

($\geq$) L'argument est simplement une application du principe des tiroirs. Considérons un sous-graphe avec m hyperarêtes et n nœuds. Pour chaque hyperarête, on choisit un nœud comme tête, donc il y a un nœud qui aura un degré sortant d'au moins m/n. Cet argument étant applicable pour tout sous-graphe, on en déduit facilement l'énoncé. □

On en déduit directement le corollaire suivant.

Corollaire 4.2. *Considérons un système avec n nœuds, f copies de $\bar{m}$ objets avec un placement d'objets $\mathcal{P}$ et un ensemble de requêtes Q. Soit $\bar{H}_{\mathcal{P}}$ l'hypergraphe f-uniforme du stockage et $H_{\mathcal{P},Q}$ l'hypergraphe des requêtes basé sur $\bar{H}_{\mathcal{P}}$ et sur l'ensemble Q. Le temps de complétion optimal est $T^*_C(Q) = \lceil \Delta(H_{\mathcal{P},Q}) \rceil$.*

En résumé, le *temps de complétion* d'un ensemble de requête à répartir entre un ensemble de nœuds correspond à la *densité maximum* de l'hypergraphe des requêtes sous-jacent. Dans la sous-section suivante, nous présentons des bornes supérieures et inférieures pour la densité maximum des graphes de requêtes lorsque les requêtes sont soit unitaires soit non populaire (au plus m/n requêtes par objets).

4.2.3 Bornes pour la densité des hypergraphes de requêtes

Tout d'abord, nous présentons une borne supérieure applicable à tout hypergraphe et qui est basé sur la multiplicité maximum. Nous déterminons ensuite cette multiplicité maximum pour les hypergraphes de requêtes unitaires et non populaires, ce qui nous permet de dériver des bornes supérieures sur la densité pour ces deux distributions de requêtes. Nous proposons également une borne inférieure, indépendante de toute distribution de requêtes.

4.2.3.1 Borne supérieure sur la densité maximum

Le premier résultat consiste à déterminer la densité maximum pour n'importe quel graphe. L'argument est combinatoire et consiste à calculer pour tout sous-ensemble de nœuds le nombre de combinaison de f nœuds dans cet ensemble. Ces combinaisons correspondent aux "emplacements" possibles pour une hyperarête de

taille f. En prenant en compte la multiplicité maximum du graphe, qui correspond au nombre d'hyperarêtes ayant les mêmes extrémités, on calcule la densité maximum du graphe.

Lemme 4.3. *Pour tout hypergraphe f-uniforme H avec m (multi-)arêtes, $\Delta(H) \leq m^{1-1/f} \cdot (\frac{M(H)}{f!})^{1/f}$ où $M(H)$ est la multiplicité maximum de H.*

Démonstration. Un sous-graphe $H' \subseteq$ avec n'_H nœuds a au plus $m_{H'}$ hyperarêtes, avec

$$m_{H'} \leq \binom{n_{H'}}{f} \cdot M(H) \leq \frac{(n_{H'})^f}{f!} \cdot M(H)$$

Ce qui suit est équivalent.

$$\begin{aligned} n_{H'}^f &\geq \frac{m_{H'} \cdot f!}{M(H)} \\ n_{H'} &\geq \left(\frac{m_{H'} \cdot f!}{M(H)}\right)^{1/f} \end{aligned}$$

On peut donc borner la densité de tout sous-graphe $H' \subseteq H$ par

$$\begin{aligned} \delta(H') = \frac{m_{H'}}{n_{H'}} &\leq m_{H'} \cdot \left(\frac{M(H)}{m_{H'} \cdot f!}\right)^{1/f} \\ &\leq (m_{H'})^{1-1/f} \cdot \left(\frac{M(H)}{f!}\right)^{1/f} \end{aligned}$$

Considérons maintenant un sous-graphe H_{max} de densité maximum.

$$\begin{aligned} \delta(H_{max}) &\leq (m_{H_{max}})^{1-1/f} \cdot \left(\frac{M(H)}{f!}\right)^{1/f} \\ &\leq (m_H)^{1-1/f} \cdot \left(\frac{M(H)}{f!}\right)^{1/f} \end{aligned}$$

□

Ce lemme nous permet donc de déterminer la densité de tout hypergraphe pour peu que l'on connaisse sa multiplicité maximum. Nous allons donc déterminer la multiplicité maximum des graphes de requêtes pour les distributions unitaires et non populaires en considérant que le graphe du stockage sous-jacent est un *graphe aléatoire*. Ce graphe du stockage est construit selon un processus aléatoire qui consiste

à choisir de manière indépendante et uniforme, pour chaque objet, un ensemble de f nœuds parmi l'ensemble des f nœuds possibles. On ajoute ensuite une hyperarête au graphe, avec comme extrémité les nœuds de l'ensemble. On peut se ramener à un problème d'urne classique dans lequel on alloue $\bar{m}$ balles dans $N = \binom{n}{f}$ urnes en choisissant une urne de manière aléatoire, uniforme et indépendante. Dans ce contexte, la multiplicité maximum est représenté par la charge maximum d'une urne. Des résultats précis pour calculer la valeur de cette charge maximum existent déjà dans la littérature (notamment [RS98]), toutefois nous utiliserons les résultats suivants, plus facile à manipuler.

Lemme 4.4 (Charge maximum d'une urne)**.** *Considérons $\bar{m}$ balles allouées de manière aléatoire, uniforme et indépendante dans N urnes. La charge L maximum est inférieure à :*

1. $3\frac{\log N}{\log(N/\bar{m})}$ *avec probabilité* $1 - 1/N$ *pour* $\bar{m} \leq N/\log N$;
2. $\frac{3\log N}{\log\log N}$ *avec probabilité* $1 - 1/N$ *pour* $N/\log N < \bar{m} \leq N$;
3. $7 \cdot \frac{\bar{m}\log N}{N}$ *avec probabilité* $1 - 1/N$ *pour* $N < \bar{m} \leq N\log N$;
4. $7 \cdot (\bar{m}/N)$ *avec probabilité* $1 - 1/N$ *pour* $\bar{m} > N\log N$.

La preuve, laissée en annexe, utilise des outils probabilistes connus comme les bornes de Chernoff ou encore l'approximation de la charge des urnes par une distribution de Poisson.

On remarque facilement que dans le cas où $\bar{m} \leq N/\log N$, on a $\frac{\log N}{\log(N/\bar{m})} \leq \frac{\log N}{\log\log N} \leq \log N$. On peut donc majorer les trois premiers cas (pour tout $m \leq N\log N$) par $7 \cdot \frac{\bar{m}\log N}{N} + 3\log N$ afin de simplifier les calculs par la suite. On en déduit le lemme suivant.

Lemme 4.5. *La multiplicité maximum d'un hypergraphe du stockage aléatoire f-uniforme est inférieura à :*

— $7 \cdot \frac{\bar{m}\log N}{N} + 3\log N$ *avec probabilité* $1 - 1/N$ *pour* $f \geq \frac{\log \bar{m}}{\log n} - 1$.
— $7 \cdot (\bar{m}/N)$ *avec probabilité* $1 - 1/N$ *pour* $f < \frac{\log \bar{m}}{\log n} - 1$.

Ce résultat sur la multiplicité du graphe aléatoire du stockage va nous servir de base pour étudier la multiplicité du graphe des requêtes.

Requêtes unitaires Nous étudions tout d'abord les graphes de requêtes pour un ensemble de requêtes unitaires. Dans ce cas, l'hypergraphe des requêtes obtenu est simplement un sous-graphe de l'hypergraphe du stockage. La multiplicité maximum d'un tel graphe de requête est donc bornée par celle du graphe de stockage. On en déduit directement le lemme suivant :

Lemme 4.6. *Soit $\bar{H}$ un hypergraphe du stockage aléatoire f-uniforme et soit H_Q un hypergraphe de requêtes basé sur $\bar{H}$ et sur un ensemble de requêtes unitaires Q. La multiplicité maximum $M(H_Q)$ est inférieure à*

— $7 \cdot \frac{\bar{m} \log N}{N} + 3 \log N$ *avec probabilité* $1 - 1/N$ *pour* $f \geq \frac{\log \bar{m}}{\log n} - 1$.
— $7 \cdot (\bar{m}/N)$ *avec probabilité* $1 - 1/N$ *pour* $f < \frac{\log \bar{m}}{\log n} - 1$.
avec $N = \binom{n}{f}$.

En combinant cet énoncé avec le Lemme 4.3 on obtient le lemme suivant :

Lemme 4.7. *Soit* $\bar{H}$ *un hypergraphe du stockage aléatoire* f*-uniforme avec* n *nœuds et* $\bar{m} \leq n^c$ *hyperarêtes, avec* $c \geq 1$. *Soit* H_Q *un hypergraphe de requêtes basé sur* $\bar{H}$ *et sur un ensemble* Q *de* m requêtes unitaires. *La densité maximum* $\Delta(H_Q)$ *est inférieure à :*
— $5 \cdot e \cdot \frac{m}{n} \cdot (\frac{\bar{m} \cdot \log n \cdot f}{m})^{1/f}$ *avec probabilité* $1 - 1/N$ *si* $f \geq \frac{\log \bar{m}}{\log n} - 1$.
— $3 \cdot e \cdot \frac{m}{n} \cdot (\frac{\bar{m}}{m})^{1/f}$ *avec probabilité* $1 - 1/N$ *si* $f < \frac{\log \bar{m}}{\log n} - 1$.
avec $N = \binom{n}{f}$.

Démonstration. Grâce au Lemme 4.3 on sait que

$$\Delta(\bar{H}) \leq \bar{m}^{1-1/f} \cdot (\frac{M(\bar{H})}{f!})^{1/f}$$

On remplace maintenant $M(\bar{H})$ par la valeur donnée dans le Lemme 4.6. On a donc deux cas à étudier.

Cas 1 : $f \geq \frac{\log \bar{m}}{\log n} - 1$

$$\begin{aligned}
\Delta(H_Q) &\leq m^{1-1/f} \cdot \left(\frac{M(\bar{H})}{f!}\right)^{1/f} \\
&\leq m^{1-1/f} \cdot \left(\frac{M(\bar{H}) \cdot e^f}{f^f}\right)^{1/f} \\
&\leq \frac{e}{f} \cdot m^{1-1/f} \cdot \left(7 \cdot \frac{\bar{m}}{N} \log N + 3 \log n\right)^{1/f} \\
&\leq \frac{e}{f} \cdot m^{1-1/f} \left(7 \cdot \frac{\bar{m} \cdot f^{f+1}}{n^f} \log n + 3 \log n\right)^{1/f} \\
&\leq e \cdot \frac{m}{n} \left(21 \frac{\bar{m}}{m} \cdot f \cdot \log n\right)^{1/f} \\
&\leq 5 \cdot e \cdot \frac{m}{n} \left(\frac{\bar{m}}{m} \cdot f \cdot \log n\right)^{1/f}
\end{aligned}$$

Cas 2 : $f < \frac{\log \bar{m}}{\log n} - 1$

$$\begin{aligned}
\Delta(H_Q) &\leq m^{1-1/f} \cdot \left(\frac{M(\bar{H})}{f!}\right)^{1/f} \\
&\leq m^{1-1/f} \cdot \left(\frac{M(\bar{H}) \cdot e^f}{f^f}\right)^{1/f} \\
&\leq \frac{e}{f} \cdot m^{1-1/f} \cdot \left(7 \cdot \frac{\bar{m}}{N}\right)^{1/f} \\
&\leq \frac{e}{f} \cdot m^{1-1/f} \left(7 \cdot \frac{\bar{m} \cdot f^f}{n^f}\right)^{1/f} \\
&\leq 3 \cdot e \cdot \frac{m}{n} \left(\frac{\bar{m}}{m}\right)^{1/f}
\end{aligned}$$

Grâce à ces deux cas, on peut donc déduire l'énoncé. □

Requêtes non populaires Nous considérons maintenant que les requêtes ne sont pas nécessairement unitaires, c'est-à-dire qu'on peut recevoir plus d'une requête pour un même objet. Toutefois on considère qu'il n'est pas possible de recevoir plus de m/n requêtes pour un même objet. On en déduit donc que la multiplicité d'un tel graphe est bornée supérieurement par $M(\bar{H})\frac{m}{n}$, où $\bar{H}$ correspond au graphe des candidats. Plus formellement, on obtient le lemme suivant.

Lemme 4.8. *Soit $\bar{H}$ un hypergraphe aléatoire du stockage f-uniforme et soit H_Q un hypergraphe de requêtes basé sur $\bar{H}$ et sur un ensemble de requêtes non populaires Q. La multiplicité maximum $M(H_Q)$ est inférieure à*

— $(7 \cdot \frac{\bar{m} \log N}{N} + 3 \log N) \cdot \frac{m}{n}$ *avec probabilité* $1 - 1/N$ *pour* $f \geq \frac{\log \bar{m}}{\log n} - 1$.

— $7 \cdot \frac{\bar{m}}{N} \cdot \frac{m}{n}$ *avec probabilité* $1 - 1/N$ *pour* $f < \frac{\log \bar{m}}{\log n} - 1$.

où $N = \binom{n}{f}$.

On peut maintenant déterminer la densité maximum.

Lemme 4.9. *Soit $\bar{H}$ un hypergraphe aléatoire du stockage f-uniforme avec n nœuds et m hyperarêtes. Soit H_Q un hypergraphe de requêtes basé sur $\bar{H}$ et sur un ensemble Q de R* requêtes non populaires. *La densité maximum $\Delta(H_Q)$ est inférieure à :*

— $5 \cdot e \cdot \frac{m}{n} \cdot (\frac{\bar{m}}{n} \cdot f \cdot \log n)^{1/f}$ *avec probabilité* $1 - 1/N$ *si* $f \geq \frac{\log \bar{m}}{\log n} - 1$.

— $3 \cdot e \cdot \frac{m}{n} \cdot (\frac{\bar{m}}{n})^{1/f}$ *avec probabilité* $1 - 1/N$ *si* $f < \frac{\log \bar{m}}{\log n} - 1$.

La preuve est très similaire à celle du Lemme 4.7 et a été laissée en annexe.

4.2.3.2 Borne inférieure sur la densité maximum

Nous donnons dans cette sous-section une borne inférieure sur la densité maximum de tout hypergraphe de requêtes f-uniforme, et ce quel que soit le graphe du

stockage considéré. On suppose que les requêtes sont unitaires, ce qui minimise la multiplicité et donc la densité maximum. Afin de simplifier les notations, on notera $\bar{H}$ le graphe du stockage et H le graphe des requêtes. On définit $\bar{m}_k = \bar{m} \cdot \binom{k}{f} / \binom{n}{f}$.

Lemme 4.10. *Quelle que soit la valeur $k \in [f, n]$, il y a un sous-graphe $H_k \subseteq H$ de k nœuds tel que le nombre d'hyperarêtes complètement contenues dans $H_k$est borné inférieurement par m_k.*

Démonstration. Le nombre d'ensemble de f nœuds est $\binom{n}{f}$ donc le nombre moyen d'hyperarêtes complètement contenues par f nœuds est $m/\binom{n}{f}$. Puisque le nombre d'ensemble de f nœuds dans un ensemble de k nœuds est $\binom{k}{f}$, on en déduit qu'il existe un sous-graphe H' de k nœuds qui contient complètement $m \cdot \frac{\binom{k}{f}}{\binom{n}{f}}$ hyperarêtes. □

Lemme 4.11. *Si $R \leq m_k$, on peut choisir un ensemble Q de m requêtes tel que $\Delta(H_Q) \geq R/k$.*

Démonstration. Supposons qu'un adversaire choisisse les requêtes de telle sorte que H_Q corresponde au sous-graphe $H_k \subset H$ décrit dans le Lemme 4.10. Il est clair que les requêtes ne peuvent être allouées qu'à des nœuds de H_Q, ce qui nous permet de déduire l'énoncé. □

En utilisant ce lemme on en déduit le suivant.

Lemme 4.12. *Pour un m donné, il y a une valeur k tel que $m < \bar{m}_k$ avec $\Delta(H_Q) \geq \frac{R}{n} \cdot (\frac{m}{R})^{1/f}$.*

Démonstration. On veut trouver le plus petit k tel que $m \leq \bar{m}_k$.

$$\begin{aligned} m &\leq \bar{m} \cdot \frac{\binom{k}{f}}{\binom{n}{f}} = \bar{m} \cdot \frac{k! \cdot (n-f)!}{(k-f)! \cdot n!} \\ &\leq \bar{m} \cdot k^f \cdot \frac{(n-f)!}{n!} \\ m \cdot \frac{n!}{(n-f)! \cdot \bar{m}} &\leq k^f \end{aligned}$$

En utilisant l'approximation de Stirling on en déduit que

$$\begin{aligned} k^f &\geq m \cdot n^f \cdot \frac{\sqrt{2 \cdot \pi}}{e^{f+1}} \cdot \frac{1}{\bar{m}} \\ k &\geq \frac{n}{e} \cdot \left(\frac{\sqrt{2 \cdot \pi} \cdot m}{e \cdot \bar{m}} \right)^{1/f} \end{aligned}$$

Donc en choisissant $k = (n/e) \cdot \left(\frac{\sqrt{2\cdot\pi}\cdot m}{e\cdot\bar{m}}\right)^{1/f}$ on en déduit qu'il existe un ensemble Q de requêtes tel que

$$\begin{aligned}\Delta(H_Q) &\geq m/\left(\frac{n}{e}\cdot\left(\frac{\sqrt{2\cdot\pi}\cdot m}{e\cdot\bar{m}}\right)^{1/f}\right)\\ &\geq \frac{m}{n}\cdot e\cdot\left(\frac{e\cdot\bar{m}}{\sqrt{2\cdot\pi}\cdot m}\right)^{1/f} \geq e\cdot\frac{m}{n}\cdot\left(\frac{\bar{m}}{m}\right)^{1/f}\end{aligned}$$

□

Nous pouvons tirer plusieurs conclusions de cette borne inférieure. Tout d'abord, on remarque que cette borne est proche de l'idéal m/n à un facteur $(\bar{m}/m)^{1/f}$ près. En choisissant $f = \Omega(\log(\bar{m}/m))$, ce facteur devient donc constant et on se ramène à l'idéal à une constante près, *quelles que soient les valeurs de m, $\bar{m}$ ou n*. On remarque également que les bornes supérieures présentées précédemment pour un placement aléatoire sont très proches de cette borne inférieure. On en déduit donc qu'en optant pour un placement aléatoire et un nombre de candidat approprié, obtenir une allocation proche de l'optimal nous garantit une bonne approximation de l'idéal.

On applique maintenant cette borne au scénario particulier où d'une part le nombre d'objets est légèrement plus grand que le nombre de nœuds et d'autre part le nombre de requêtes est égale au nombre de nœuds.

Corollaire 4.13. *Considérons un système de n nœuds stockant f copies de $\bar{m} = n^\alpha$ objets ($\alpha \geq 1$) et un ensemble de m requêtes. Pour $m = n$ et $\alpha \geq \log\log n + 1$, le temps de complétion est au moins $\Omega(\log n)$ si on choisit $f \leq \log n$, quel que soit le placement d'objets considéré.*

Démonstration. Grâce au Lemme 4.12, on sait que le temps de complétion est au moins $e \cdot (\frac{m}{n})^{1/f} \geq e \cdot (n^{\log\log n})^{1/f}$. En choisissant $f = \log n$ on obtient la borne suivante.

$$\begin{aligned}e\cdot(n^{\log\log n})^{1/f} &\geq e\cdot n^{\frac{\log\log n}{\log n}}\\ &= e\cdot\exp\left(\log\left(n^{\frac{\log\log n}{\log n}}\right)\right)\\ &= e\cdot\exp\left(\frac{\log\log n}{\log n}\log n\right)\\ &= e\cdot\exp(\log\log n) = e\cdot\log n\end{aligned}$$

On en déduit l'énoncé. □

Ce résultat confirme qu'un nombre logarithmique de copies est nécessaire pour espérer garantir un temps de complétion proche de l'idéal dans un modèle d'adversaire omniscient.

Nous concluons cette section en remarquant que le nombre de candidats requis pour prévenir un comportement adversarial est en fait très faible en pratique. En effet, si on considère que le système est composé de $n = 1000$ nœuds de stockage, alors $\log n \sim 7$ et $\log \log n \sim 2$.

4.3 Algorithmes Distribués

Dans cette section, nous présentons notre algorithme distribué d'allocation de requête et de gestion de copies. Nous présentons tout d'abord les hypothèses de travail considérées puis nous décrivons de manière assez informelle notre algorithme afin de donner les idées de son fonctionnement.

4.3.1 Modèle distribué

Chaque nœud possède une file de requêtes en attente. On suppose que tous les nœuds peuvent communiquer entre eux et connaissent, pour tout objet o qu'ils stockent l'ensemble $P(o)$ des nœuds candidats.

Parmi les nœuds, nous en distinguons un en particulier que nous appelons la *racine* et que nous notons R. La racine joue un rôle prépondérant dans le calcul de la popularité et l'estimation de la charge moyenne.

On rappelle que nous considérons un modèle hors-ligne où toutes les requêtes à allouer sont présentes sur le système. Par contre, une requête peut être initialement reçue par n'importe quel nœud du système de manière distribuée. Ce nœud va donc jouer le rôle de coordinateur et envoyer la requête à un ou plusieurs nœuds candidats (voir sous-section 4.3.1). Une fois la coordination terminée, une requête ne pourra être supprimée de la file que lors de l'exécution de l'algorithme d'allocation.

On précise enfin que les communications se font de manière asynchrone mais qu'aucun nœud n'est en panne. Tous les nœuds peuvent communiquer entre eux. On suppose qu'un message envoyé à un nœud sera reçu en au plus une unité de temps.

Phase 1 : Gestions des copies Notons m_i le nombre de requêtes reçues par le système pour un objet $o_i \in O$ et $f_i \geq f$ le nombre de copies pour cet objet. Initialement, on a donc $f_i = f$. Nous redéfinissons légèrement la notion de popularité en considérant qu'un objet est *populaire* si $\frac{m_i}{f_i} > \frac{m}{n}$ mais non populaire dans le cas contraire. Intuitivement, un objet o est populaire si même en répartissant au mieux les requêtes concernant o entre les différents nœuds candidats, il est impossible d'atteindre l'idéal.

L'évaluation de la popularité sera effectuée par la racine, **avant le début de l'allocation**. La racine va procéder à une première diffusion/agrégation afin de récolter des statistiques sur le nombre de requêtes reçues pour chaque objet. Grâce aux informations de popularité envoyées par les nœuds, la racine est capable de déterminer si un objet est populaire ou non. Lorsqu'un objet o est détecté comme étant populaire, la racine va choisir $\lfloor m_i \cdot n/m \rfloor - f_i$ nouveaux nœuds candidats différents de manière aléatoire et uniforme. Chacun de ces nœuds va recevoir une copie de o.

La racine diffuse ensuite à tous les nœuds les emplacements de ces copies additionnelles.

Nous ferons référence à ce protocole par `Copies-management` dans le reste du manuscrit.

Phase 2 : Coordination des requêtes On suppose maintenant que les copies additionnelles ont été ajoutées et que les nœuds connaissent leurs emplacements. On se place ici du point de vue d'un nœud coordinateur u. La stratégie utilisée pour l'allocation des requêtes reçues pour un objet o dépend de la popularité de o. On distingue deux cas de figure :

1. Si o n'est pas populaire, u va transmettre les requêtes à tous les nœuds candidats.
2. Si o est populaire, pour chaque requête reçue q_o, le nœud u va choisir un nouveau nœud candidat grâce à l'algorithme du tourniquet.

Phase 3 : Allocation des requêtes non populaires Après cette étape de coordination, il faut encore prendre une décision d'allocation pour les requêtes non populaires. Pour cela, nous exécutons une adaptation de l'algorithme `AvrDeg` présenté dans le Chapitre 3. En effet, comme par hypothèse les nœuds ne peuvent pas tomber en panne, il est tout à fait envisageable d'utiliser une barrière de synchronisation.

On précise que les nœuds peuvent distinguer si une requête est populaire ou non. Ils peuvent également savoir si une requête est simplement en attente ou bien déjà allouée de manière permanente.

Nous rappelons brièvement le fonctionnement de l'algorithme. Soit $\ddot{\ell}(u)$ le nombre de requêtes en attente dans la file de u. La racine va estimer la charge moyenne $\ddot{\ell}_{avr}$ des nœuds ayant des requêtes en attente et diffuser cette valeur. Un nœud u qui apprend que sa charge $\ddot{\ell}(u)$ est inférieure à $(1+\epsilon) \cdot \ddot{\ell}_{avr}$ va s'activer et s'affecter de manière permanente *toutes les requêtes de sa file*. Les autres nœuds candidats sont informés de cette décision et suppriment les requêtes correspondantes, sauf s'ils se sont également activés au préalable. Après toute activation, la racine est informée du nombre de requêtes affectées à un nœud. Après avoir reçu une réponse de tous les nœuds, la racine recalcule la charge moyenne $\ddot{\ell}_{avr}$ et diffuse cette valeur à tous les nœuds. On réitère ce processus jusqu'à ce que tous les nœuds aient été activés.

4.4 Analyse du temps de complétion

Nous étudions maintenant les performances de l'algorithme présenté dans la section précédente. On rappelle que la gestion de copies est effectuée *avant de commencer l'allocation* et que les requêtes sont permanentes. Tout d'abord faisons l'observation suivante.

Observation 4.1. *Soit T le temps de complétion pour un ensemble de requête après tout algorithme d'allocation. Soit T_u le temps de complétion en considérant uniquement les requêtes non populaires et T_p le temps de complétion en considérant uniquement les requêtes populaires, alors $T \leq T_u + T_p$.*

Nous faisons donc de manière indépendante l'analyse de `AvrDeg` pour les requêtes non populaires (Lemme 4.15) et de l'algorithme du tourniquet pour les requêtes populaires (Lemme 4.17). Nous en déduisons ensuite une borne supérieure sur le temps de complétion globale de notre algorithme (Théorème 4.18).

4.4.1 Requêtes non populaires

L'algorithme `AvrDeg` a déjà été étudié pour le cas $f = 2$, nous nous proposons de généraliser cette étude pour le cas $f > 2$. Nous nous intéressons au problème de graphe sous-jacent de l'orientation minimum. Soit $H = (V, E)$ l'hypergraphe de requêtes à orienter. Contrairement à la structure de données de graphe manipulée dans le chapitre 3, on considère que le graphe est initialement non orienté et on oriente au fur et à mesure ses hyperarêtes. Le degré non orienté $\ddot{d}(u)$ correspond au nombre de requêtes en attente $\ddot{\ell}(u)$ et on va donc chercher à déterminer le degré moyen $\ddot{d}_{avr}$.

Soit $D^+(H)$ le degré sortant maximum à la fin de l'algorithme, on montre le lemme suivant :

Lemme 4.14. *Soit H un hypergraphe des requêtes f-uniforme représentant un ensemble de requêtes non populaires. Après l'exécution de `AvrDeg`, on obtient la garantie suivante.*

$$D^+(H) \leq \lceil f \cdot (1 + \epsilon) \cdot \Delta(H) \rceil$$

Démonstration. La valeur de $\ddot{d}_{avr}(H_i)$ est $\frac{\sum_{u \in V(H_i)} d(u)}{n_{H_i}} = f \cdot \frac{m_{H_i}}{n_{H_i}} = f \cdot \delta(H_i)$ où H_i représente le sous-graphe f-uniforme non-orienté de H après la diffusion i de la racine. On en déduit que la racine ne diffuse jamais de valeur supérieure à $f \cdot \Delta(H)$, ce qui implique qu'un nœud ne peut pas avoir un degré sortant supérieur à $\lceil (1 + \epsilon) f \Delta \rceil$ après activation. □

En combinant ce lemme avec le Lemme 4.1, on en déduit donc que `AvrDeg` est une f approximation de l'optimal. Concernant le temps d'exécution de l'algorithme,

il est de $O(\log_{2+\epsilon} \cdot T_B)$ rondes, où T_B correspond au nombre de rondes nécessaire afin de diffuser un message à tous les nœuds (voir Section 3.1 pour plus de détails).

On peut maintenant se servir des résultats obtenus dans la Section 4.2 sur la densité des graphes de requêtes afin de déterminer les performances vis-à-vis de l'idéal. Nous avons vu qu'un nombre de candidats de l'ordre de $\Omega(\log(\bar{m}/n))$ permet d'approximer l'idéal à une constante près. Toutefois, il faut également prendre en compte l'approximation de notre algorithme afin de déterminer le nombre de candidats approprié. Nous prouvons le lemme suivant :

Lemme 4.15. *Le temps de complétion obtenu avec* `AvrDeg` *est* $O(f \cdot \frac{m}{n})$ *avec grande probabilité en fixant la valeur de* f *comme suit :*

$$f = \begin{cases} \lceil (\alpha - 1) \log n \rceil & si\ \alpha \geq 2 \\ \left\lceil \frac{\alpha}{2} \log n \right\rceil & si\ 1 + \frac{\log \log n}{\log n} < \alpha < 2 \\ \lceil \log \log n \rceil & sinon \end{cases}$$

Démonstration. Considérons l'hypergraphe des requêtes H sous-jacent. Nous avons prouvé dans le Lemme 4.14 que le degré sortant maximum $D^+(H)$ est borné par $f \cdot (1+\epsilon) \cdot \Delta(H)$ avec `AvrDeg`. Afin de simplifier les calculs, supposons que $\epsilon = 0$. Nous nous concentrons sur le cas où on ne peut pas recevoir plus de m/n requêtes identiques. On remarque tout d'abord que $x^{1/\log x} = \exp(\frac{1}{\log x} \log x) = \exp(1)$ et $x^{1/x} = \exp(\frac{1}{x} \log x) \leq \exp(1)$.

Considérons le cas $\alpha \geq 2$. Si $f \geq \log n$, on obtient la borne suivante :

$$\begin{aligned} \Delta(H) &\leq 5e \frac{m}{n} \left(f \cdot \log n \cdot \frac{\bar{m}}{n} \right)^{1/f} \\ &\leq 5e^3 \frac{m}{n} \left(\frac{\bar{m}}{n} \right)^{1/f} \end{aligned}$$

On conclut facilement, en étudiant la dérivée, que le minimum est atteint pour $f = \log \frac{\bar{m}}{n} = \log \frac{n^\alpha}{n} = (\alpha - 1) \log n$, qui est plus grand que $\log n$ pour le cas considéré. Puisque $(\bar{m}/n)^{1/\log(\bar{m}/n)} =$, on déduit la borne suivante pour le degré sortant maximum avec `AvrDeg`.

$$\begin{aligned} D^+(H) &\leq f \cdot 5e^3 \frac{m}{n} \left(\frac{\bar{m}}{n} \right)^{1/f} \\ &= \log(\bar{m}/n) 5e^3 \frac{m}{n} \left(\frac{\bar{m}}{n} \right)^{1/\log(\bar{m}/n)} \\ &= \log(\bar{m}/n) 5e^4 \frac{m}{n} \\ &= (\alpha - 1) \log n \cdot 5e^4 \frac{m}{n} \end{aligned}$$

Pour $\alpha < 2$, on a $(\log m / \log n) - 1 \leq 1$ donc on considère uniquement la borne $\Delta(H) \leq 3e \frac{m}{n} \left(f \cdot \log n \cdot \frac{\bar{m}}{n} \right)^{1/f}$. En utilisant des arguments similaires à ceux du cas

précédent, on en déduit que si $1+\frac{\log\log n}{\log n} < \alpha < 2$ et si on choisit $f = \lceil(\alpha/2)\log n\rceil \geq \log n$ alors :

$$\begin{aligned}\Delta(H) &\leq 5 \cdot e^3 \cdot (m/n) \cdot (\bar{m}/n)^{1/f} \\ &< 5 \cdot e^3 \cdot (m/n) \cdot (n)^{1/f}\end{aligned}$$

Ce qui implique la propriété suivante pour le degré maximum avec `AvrDeg`.

$$\begin{aligned}D^+(H) &< f \cdot 5 \cdot e^3 \cdot (m/n) \cdot (n)^{1/f} \\ &< (\alpha/2)\log n \cdot 5e^4 \cdot (m/n)\end{aligned}$$

Finalement si $\alpha \leq 1 + \frac{\log\log n}{\log n}$ et $f = \log\log n$, alors

$$\begin{aligned}\Delta(H) &\leq 5 \cdot e^3 \cdot (m/n) \cdot (f \cdot \log n \cdot \bar{m}/n)^{1/f} \\ &< 5 \cdot e^3 \cdot (m/n) \cdot (f \cdot \log n \cdot n)^{\frac{\log\log n}{f \cdot \log n}}\end{aligned}$$

Ce qui implique que

$$\begin{aligned}D^+(H) &< f \cdot 5 \cdot e^3 \cdot (m/n) \cdot n^{(\log\log n)/(f \cdot \log n)} \\ &< (\log\log n) \cdot 5 \cdot e^3 \cdot (m/n) \cdot n^{\frac{\log\log n}{\log\log n \cdot \log n}} \\ &< (\log\log n) \cdot 5 \cdot e^3 \cdot (m/n) \cdot n^{1/\log n} \\ &< (\log\log n) \cdot 5 \cdot e^4 \cdot (m/n)\end{aligned}$$

Pour résumer, on obtient un temps de complétion de $O(f \cdot (R/n))$ en choisissant $f = O(\log n)$, quelle que soit la valeur de α. □

4.4.2 Requêtes populaires

Nous analysons maintenant les performances de l'algorithme du tourniquet utilisé pour répartir les requêtes populaires entre les candidats, y compris ceux ajoutés additionnellement par l'algorithme `Copies-management`. Nous montrons tout d'abord le lemme suivant.

Lemme 4.16. *Au plus n candidats supplémentaires sont ajoutés par l'algorithme `Copies-management`.*

Démonstration. On remarque tout d'abord que la somme de $\sum_{i=1}^{\bar{m}} m_i = m$. On en déduit la borne suivante sur le nombre de candidats additionnels :

$$\begin{aligned}\sum_{i=1}^{\bar{m}} m_i \cdot (n/m) - f_i &\leq \sum_{i=1}^{m} m_i \cdot (n/m) \\ &= n/m \cdot \sum_{i=1}^{\bar{m}} m_i \\ &= n/m \cdot m \\ &= n\end{aligned}$$

Il y a donc au plus n candidats additionnels pour les requêtes populaires. □

On étudie maintenant le temps de complétion de l'algorithme du tourniquet en supposant que les copies additionnelles ont déjà été ajoutées.

Lemme 4.17. *Après avoir appliqué l'algorithme* `Copies-management`*, un nœud reçoit au plus* $12f$ *copies d'objets supplémentaires avec probabilité* $1 - 1/n$*.*

Démonstration. Considérons un nœud u et supposons qu'il y a $k \leq n$ requêtes populaires. Soit X_i une variable aléatoire binaire qui vaut 1 si u est candidat pour une requête populaire q_i et 0 sinon. Puisqu'il y a $f_i > f$ candidats pour q_i, on en déduit que $P(X_i = 1) = f_i/n$. En sommant sur toutes les requêtes populaires, on en déduit que le nombre de copies additionnelles reçues par u est $X = \sum_{i=1}^{k} X_i$. Par la linéarité de l'espérance, on en déduit que $E[X] = \frac{k}{n}(f + 1)$. En utilisant les bornes de Chernoff, on obtient pour toute constante $c > 1$:

$$P(X \geq (1 + c)E[X]) < e^{-cE[X]/3}$$

Si on fixe c tel que $cE[X]/3 > 2 \log n$, on obtient que $P(X \geq (1+c)E[X]) \leq 1/n^2$. Puisque $c > 1$, alors $(1 + c)E[X] < 2cE[X] < 12 \log n \leq 12f$. En appliquant l'inégalité de Boole sur les n variables aléatoires de même espérance, on en déduit qu'aucun nœud n'est candidat pour plus de $12f$ requêtes populaires avec probabilité $1 - 1/n$. □

4.4.3 Théorème principal

Après avoir étudié séparément le temps de complétion pour les requêtes non populaires et les requêtes populaires, en appliquant respectivement les algorithmes `AvrDeg` et du tourniquet, nous prouvons le résultat principal de notre chapitre.

Théorème 4.18. *En combinant* `Copies-management`*,* `AvrDeg` *et l'algorithme du tourniquet avec* $f = \left\lceil \max\{\log n, \log \frac{\bar{m}}{n}\} \right\rceil$*, le temps de complétion est d'au plus* $O(f \cdot \frac{m}{n})$ *avec grande probabilité.*

Démonstration. Le cas des requêtes non populaires ayant déjà été traité, nous allons étudier le nombre de requêtes reçues par un nœud pour des requêtes populaires. On suppose que l'algorithme `Copies-management` a déjà été appliqué. Par définition, on a donc $m_i \leq f_i \cdot m/n$ pour toute requête q_i. Soit $m_{i,j}$ le nombre de requêtes envoyées par un nœud u_j pour la requête q_i. La décision est prise localement par chaque nœud, qui applique l'algorithme du tourniquet parmi les f_i candidats. Donc un nœud candidat pour une requête q_i reçoit au plus $\left\lceil \frac{m_{i,j}}{f_i} \right\rceil \leq \frac{m_{i,j}}{f_i} + 1$ requêtes identiques à q_i envoyées par le nœud u_j. Au total, il doit répondre à au plus $\frac{m_i}{f_i} + n \leq m/n + n$ requêtes identiques à q_i. D'après le Lemme 4.17, on sait que u est candidat pour au plus $12f$ requêtes populaires. Le temps de complétion est donc au plus

$$12f(R/n + n)$$

En prenant $m > n^2$, on en déduit que le temps de complétion est $O((\alpha - 1)\log n\frac{R}{n})$, borne identique au cas des requêtes non populaires. □

4.5 Conclusion du chapitre

Nous avons étudié dans ce chapitre l'impact que le nombre de candidats pouvait avoir sur le temps de complétion d'un algorithme. Nous avons montré qu'un placement d'objet aléatoire permet de garantir un temps de complétion proche de l'optimal, à une constante près.

Nous avons proposé un algorithme distribué d'allocation de requêtes et de gestion de popularité, combiné à un placement d'objets aléatoire. Dans ce contexte, notre algorithme est une $O(f)$-approximation de l'optimal pour tout f et une $O(f)$-approximation de l'idéal lorsque f est logarithmique. Cette performance est garantie *pour toute distribution de requêtes.*

Plusieurs questions naturelles se posent comme perspectives.

Question 4.1. *Que peut-on garantir si les requêtes ont un poids différent ?*

L'extension naturelle de nos travaux consiste à étudier les performances de notre algorithme avec des poids hétérogènes. En effet, concernant `AvrDeg`, seule la définition de charge moyenne serait à réviser pour pouvoir appliquer l'algorithme à ce cas particulier. Si on définit la charge d'un nœud comme la somme des poids des requêtes qu'il doit traiter, un nœud aurait une charge d'au plus $\max\{w_{max}, (1+\epsilon)L_{avr}\}$, où L_{avr} correspond ici à la charge moyenne maximum pour tout ensemble de nœuds et w_{max} au poids maximum d'une requête. En ramenant ce problème à l'étude de la densité de l'hypergraphe *pondéré* sous-jacent, on doit pouvoir garantir que cette borne est une bonne approximation de l'optimal. Il resterait toutefois à déterminer à quel point on est loin de l'idéal.

Question 4.2. *Que peut-on garantir si le choix des requêtes suit une distribution de probabilité particulière ?*

Dans le cas que nous avons considéré ici, les requêtes étaient choisies par un adversaire omniscient qui tentait de surcharger le système. Nous cherchons donc des garanties dans *le pire cas*. Nous avons également présenté des travaux étudiant un choix de requêtes aléatoire *uniforme* en considérant différents nombres de candidats par requêtes ([CSW07], [GW10]). Dans le premier cas, nous avons montré qu'un nombre logarithmique de candidats est nécessaire pour contrer un adversaire. Dans le deuxième cas, à l'inverse, un nombre de candidats égal à 2 est suffisant pour garantir un temps de complétion idéal. Il serait intéressant par exemple de regarder des distributions de probabilité plus représentatives de distributions réelles. Les graphes aléatoires en loi de puissance dit *RPLG* [Lu01] par exemple ont un degré en moyenne constant si le logarithme de la taille du graphe est constant et semblent donc être des candidats naturels pour l'étude de densité. On peut espérer qu'un facteur de réplication sous-logarithmique ou même constant pourrait permettre d'obtenir une bonne approximation de l'idéal.

Chapitre 5

Allocation différée d'un flux de requêtes

Sommaire

A l'inverse des chapitres précédents, nous ne considérons pas dans ce chapitre un modèle hors-ligne où toutes les requêtes ont été reçues par le système avant de prendre une décision d'allocation. Les requêtes arrivent *en flux* et doivent donc être allouées au fur et à mesure de leur arrivée. De plus, il faut aussi prendre en compte que les requêtes sont traitées par les nœuds et ne restent donc pas de façon permanente dans les files d'attente.

Une autre contrainte que nous abordons ici est l'évolution potentielle de la distribution des requêtes. En effet, un objet fortement demandé sur une période de temps donnée peut tout à fait ne plus être demandé du tout sur une autre période de temps. L'application immédiate de notre gestion de copies présenté dans le Chapitre 4 peut donc s'avérer problématique, car elle ne permet dans l'état actuel que d'ajouter des copies.

La solution adoptée dans ce chapitre consiste à faire le bilan des requêtes reçues à intervalles de temps réguliers et de mettre à jour le système afin d'adapter le nombre des copies en fonction de l'évolution de la popularité des objets. De plus, nous prenons en compte les requêtes déjà allouées à un nœud lors de l'allocation de requêtes récentes et procédons à une prise de décision *différée* pour l'allocation.

Les expériences présentées dans ce chapitre ont été réalisées avec Frédérique Lalanne, ingénieur de recherche au LaBRI. Une partie de ces expériences a été présentée à la conférence BDA 2016 [GHL16].

5.1 Préliminaire

Nous avons pu voir dans le chapitre précédent (*cf.* Chapitre 4) que la réplication d'objet permet de garantir de bonnes performances sur le temps de complétion dans un modèle hors-ligne. En particulier, nous avons vu que l'algorithme `Copies-management` qui consiste à adapter le nombre de copies d'objets en fonction de leur popularité permet de garantir un temps de complétion proche de l'optimal.

L'objectif principal de ce chapitre consiste donc à mesurer expérimentalement l'impact de la réplication sur le temps de réponse aux requêtes lorsque celles-ci arrivent en flux. On précise que l'on ne considère pas de migration d'objets et que l'on cherche à minimiser le nombre de suppressions et de créations de copies.

Nous étudions également dans ce chapitre les solutions d'allocation en-ligne mais avec prise de décision *différée*. L'idée consiste à exécuter en parallèle un algorithme d'affectation proche de `AvrDeg` et de mettre à jour les copies d'objets en fonction de l'évolution de la distribution des requêtes.

Nous avons choisi d'utiliser la base de données distribuée Apache Cassandra afin d'implémenter nos algorithmes. L'allocation des requêtes dans Cassandra se faisant de manière en-ligne et avec prise de décision immédiate, on pourra donc mesurer l'apport éventuel de la prise de décision différée. Nous présentons ce système dans la Section 5.2.

Nous présentons ensuite une combinaison d'algorithmes distribués qui sont des adaptations dans le modèle de flux des algorithmes `AvrDeg` et `Copies-management` (*cf.* Section 5.3). Notamment, nous utilisons pour l'allocation une nouvelle notion de charge qui tient compte des requêtes déjà allouées à un nœud. Pour la gestion de copies, un nœud *racine* va faire le bilan de la popularité des objets et adapter le nombre de copies, en prenant en compte les *baisses* de popularité.

Nous menons enfin une étude expérimentale sur une version modifié de Cassandra

(*cf.* Section 5.4). Nous présentons les modifications qui ont été apportées au système afin de pouvoir implémenter nos algorithmes ainsi que le protocole expérimental mis en place, les jeux de données utilisés ainsi que les résultats des expériences. On montre notamment que notre gestion de copies permet un réel gain de performance dans la quasi-totalité des scénarios considérés.

5.2 Présentation de Apache Cassandra

Dans le Chapitre 4, nous avons étudié d'un point de vue théorique l'impact du nombre de candidats sur le temps de complétion pour un ensemble de requêtes donné. Nous avions notamment illustré ce cas d'étude en se plaçant dans le contexte des *bases de données distribuées*. Dans ce chapitre, nous menons une étude expérimentale dans ce type de systèmes afin d'évaluer les performances de nos algorithmes en pratique. Nous avons opté pour le système distribué Apache Cassandra [Cas], initialement développé par Facebook [LM10], qui est massivement utilisé par de nombreuses applications gérant de grandes masses de données et une grande quantité de requêtes. Sans chercher à être exhaustif, on pourra citer par exemple Instagram, Netflix ou encore GitHub (source [Cas]).

Les fonctionnalités décrites dans le reste de cette section correspondent à la version 2.1 de Apache Cassandra [1].

5.2.1 Architecture du système

Les nœuds de stockage dans Cassandra sont organisés comme un réseau pair-à-pair dans lequel un nœud peut à la fois émettre des requêtes et y répondre. Afin de faire des requêtes sur les données stockées, un client peut se connecter à n'importe quel nœud du système. Lorsqu'un nœud reçoit une requête, il va ensuite jouer le rôle de coordinateur pour cette requête et la rediriger vers un nœud pouvant la traiter, c'est-à-dire un nœud stockant l'objet concerné par la requête. Cette redirection va dépendre de la stratégie d'allocation choisie (cf. Figure 5.1).

Chaque nœud envoie à intervalle de temps régulier des informations statistiques aux autres nœuds, comme son statut ou bien le temps de réponse moyen aux requêtes. Au cours de ce protocole de communication appelé *gossip*, un nœud choisit aléatoirement un autre nœud pour envoyer ces informations. De plus, il envoie également un message à un nœud parmi un ensemble de nœuds appelés *seeds*. Ces nœuds ont donc une information très récente de l'état du système.

1. La documentation est disponible à l'adresse `http://docs.datastax.com/en/cassandra/2.1/`

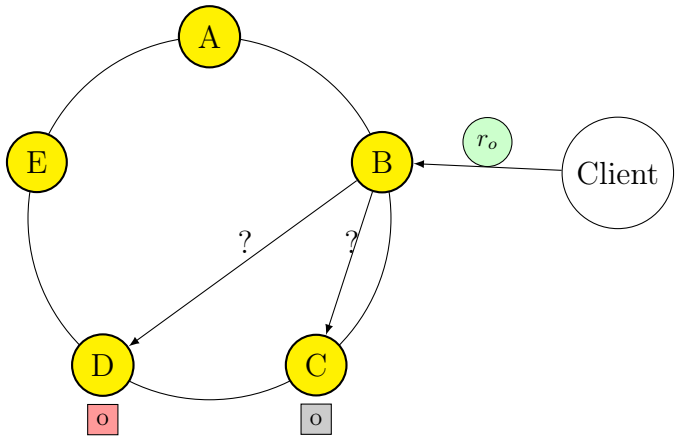


FIGURE 5.1 – Illustration de l'envoi d'une requête à une base de données distribuée par un client. Ici le client fait une requête pour un objet o stocké par les nœuds C et D. C'est le nœud B qui recoit la requête et qui va ensuite la transférer à l'un des deux nœuds candidats.

5.2.2 Les données

Afin de simplifier, on considère que les données sont représentées sous forme d'une table à plusieurs colonnes, qui sont aussi appelées des *attributs*. Cette table est ensuite partitionnée en *lignes* qui sont stockées de manière distribuée par les nœuds. Nous ferons référence à un *objet* pour désigner une ligne ou un ensemble de lignes de la table. Chaque objet est associé à une *clé primaire* permettant de le distinguer du reste des données. Cette clé primaire peut directement être utilisée pour le partitionnement des données. Il existe également des clés primaires composites constituées notamment d'une *clé de partition*, qui permet de déterminer le placement de l'objet. Typiquement, une clé est un ensemble de colonnes de la table. Dans le reste de cette description, on utilisera le terme de clé de partitionnement pour désigner la clé utilisée pour placer les objets.

Chacune des clés de partitionnement va être associée à un identifiant dans un espace d'adressage, que l'on représente généralement sous la forme d'un *anneau*. Cet identifiant est choisi par le *partitioner*. Le partitioner peut utiliser les fonctions de hachage `MurmurHash` (fonction non-cryptographique) ou `MD5` (fonction cryptographique). Dans les deux cas, ces fonctions utilisent le principe du hachage consistant décrit dans [KLL+97] pour répartir équitablement les données entre les nœuds.

Chaque nœud est également associé à un identifiant unique dans l'anneau. Soient $id(u_1), \ldots, id(u_n)$ l'ensemble des identifiant tel que $id(u_i) < id(u_j)$ si $i < j$. Un nœud u_i est responsable de l'intervalle allant de l'identifiant précédant $id(u_{i-1})$ à son identifiant $id(u_i)$. Les identifiants des nœuds sont choisis de manière à ce que ces intervalles soient identiques pour chaque nœuds.

Soit h la fonction de hachage utilisée par le partitioner et soit $h(k_o)$ le résultat de la fonction de hachage sur la clé de partition k_o de l'objet o. Cet objet sera stocké par

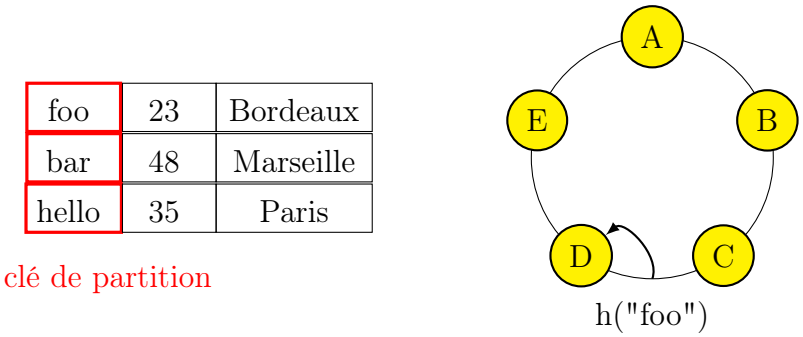


FIGURE 5.2 – Illustration de la répartition des objets entre les nœuds. La figure de gauche représente la table à partitionner. La colonne servant de clé de partition est coloriée en rouge. Dans notre exemple, la valeur de hachage $h(" foo")$ est comprise dans l'intervalle géré par le nœud D. Ce dernier stockera donc cette ligne.

le nœud responsable de l'intervalle d'identifiant contenant $h(k_o)$ (*cf.* Figure 5.2). Les fonctions de hachage utilisées permettent de garantir un bon équilibrage du stockage et un nombre de migrations d'objets limité en cas de panne d'un nœud.

Cassandra propose également un système de serveurs virtuels, afin d'affiner la répartition des objets entre les nœuds.

5.2.3 La réplication

Les objets stockés sont répliqués un nombre constant de fois (3 par défaut) afin de garantir une meilleure disponibilité. Bien que ce facteur de réplication soit paramétrable, il est cependant identique pour tous les objets.

Contrairement au placement de l'objet original, le placement des copies se fait de manière déterministe. La première stratégie appelée `SimpleStrategy` consiste à placer les copies d'objets sur les nœuds qui suivent l'emplacement de l'objet original. Une autre stratégie appelée `NetworkTopologyStrategy` consiste à tenir compte de la topologie du système (rack, data center,...).

5.2.4 Équilibrage de charge

Par défaut, un coordinateur alloue une requête à un des nœuds candidats immédiatement après la réception. Pour choisir le nœud qui va recevoir la requête, un mécanisme appelé *dynamic snitch* est utilisé. Informellement, le dynamic snitch consiste à mesurer la performance des différents nœuds candidats pour répondre à cette requête par le passé et à choisir le plus performant pour l'allocation.

Nous nous référerons à cet algorithme par `Alloc-Cassandra` dans le reste du manuscrit.

5.2.5 Mise en tampon

Il existe des phénomènes de mise en tampon par les nœuds lors de l'étape de coordination. Plus précisément, un nœud coordinateur peut transférer un nombre limité de requêtes aux nœuds candidats et doit attendre la réponse d'un nœud avant de lui envoyer une nouvelle requête. En fonction de la fréquence d'injection des requêtes par le client, le nœud coordinateur va donc conserver un nombre plus ou moins important de requêtes.

5.3 Algorithmes distribués

Nous présentons ici les modifications apportées aux algorithmes `AvrDeg` et `Copies-management` afin de les rendre utilisables avec un flux de données. En parallèle, un nœud va également traiter la première requête de sa file. La principale différence avec les algorithmes précédents est que nous prenons en compte l'évolution de la distribution des requêtes reçues, notamment pour la gestion de copies, et l'évolution de la charge des nœuds au fur et à mesure du traitement des requêtes.

Initialement, il existe f copies de chaque objet. Les objets (et toutes leurs copies) sont placés de manière aléatoire et uniforme et on note $\mathcal{P}$ ce placement d'objets. On rappelle que $\mathcal{P}(o)$ est l'ensemble des nœuds du système ayant une copie de l'objet o. Afin de simplifier la présentation, on pourra utiliser la notation $\mathcal{P}(q)$ pour désigner l'ensemble des nœuds candidats pour une requête q. De manière générale, un nœud u maintient :

- une file de requêtes en attente F_u. On précise qu'un nœud peut distinguer si une requête de la file est déjà allouée ou non.
- une variable $curr_u$ qui correspond à la requête traitée actuellement par u.
- le placement d'objets $\mathcal{P}$

Un des nœuds du système est distingué des autres et s'appelle la *racine*, notée R. La racine maintient en plus un tableau L avec la charge de chaque nœud ainsi qu'un vecteur $\mathbf{m} = \{m_1, \ldots, m_m\}$ où m_i correspond au nombre de requêtes reçues pour un objet o_i.

Nous utilisons également une notion de charge différente, à savoir que nous tenons compte des requêtes déjà allouées à un nœud. Cette notion de charge provient de l'algorithme `Selfless` présenté dans [CSW07]. Plus précisément, si on note $\ell(u)$ le nombre de requêtes déjà allouées à un nœud u et $\ddot{\ell}(u)$ le nombre de requêtes qui ne sont pas allouées dans la file, alors on définit la *charge mixte* $\ell^*(u) = 2\ell(u) + \ddot{\ell}(u)$. On note ℓ^*_{avr} la charge mixte moyenne du système. Cependant, afin de simplifier la présentation, nous utiliserons le terme de charge au lieu de charge mixte dans le reste de ce chapitre.

Enfin on précise que les nœuds communiquent par envoi de messages. Nous décrivons les différents types de messages existants :

— envoi d'un message de suppression $\langle \textbf{delete}, q \rangle$ à tous les nœuds de $\mathcal{P}(q)$.
— envoi d'un message de diffusion $\langle \textbf{broadcast}, \ell^*_{avr} \rangle$ à tous les nœuds si u est la racine.
— envoi de la charge $\langle \textbf{load}, \ell^*(u) \rangle$ par un nœud u à la racine.
— envoi d'un message $\langle \textbf{coord}, q \rangle$ par un nœud coordinateur à la racine.
— envoie d'un message $\langle \textbf{update}, \mathcal{P} \rangle$ par la racine à tous les nœuds lorsque le placement d'objet évolue.

Nous donnons maintenant plus de détails sur l'algorithme et notamment sur les actions à effectuer lors de la réception d'un de ces messages.

5.3.1 Coordination et traitement des requêtes

On considère qu'un client peut se connecter à n'importe quel nœud du système afin de faire des requêtes. Supposons qu'un nœud u reçoit une requête q de la part du client, il va transférer q à tous les nœuds candidats de $\mathcal{P}(q)$ et avertir la racine R de la réception de cette requête. La racine va ensuite mettre à jour son tableau de popularité.

Concernant le traitement, u traite la première requête $q = F_u[0]$ de sa file F_u (si $F_u \neq \emptyset$) puis envoie un message de suppression $\langle \textbf{delete}, q \rangle$ à tous les nœuds de $\mathcal{P}(q)$ afin qu'ils suppriment q de leurs files respectives. Cet algorithme simple de défilement sera appelé `Naive` dans le reste du manuscrit.

5.3.2 Gestion des copies

La gestion des copies sera gérée par la racine qui va, à intervalle régulier, appliquer l'algorithme `Copies-management` décrit dans le chapitre précédant (voir 4.3). Cependant, cet algorithme ne peut qu'ajouter des copies pour des objets populaires et ne prend pas en compte une éventuelle baisse de la popularité. Afin de palier ce problème, nous ajoutons un mécanisme de suppression de copies.

On rappelle que m_i correspond au nombre de requêtes reçues pour un objet o_i et f_i le nombre de copies de o_i déjà présentes sur le système. Si $m_i/f_i < m/n$, alors la racine va supprimer $\max\{0, f_i - 2 - \lfloor m_i \cdot n/m \rfloor\}$. On remarque qu'on garantit ainsi qu'il y a toujours au moins 2 copies de chaque objet sur le système.

On peut également paramétrer le seuil de suppression et de création de copies grâce à deux variables $p \geq 1$ et $p' \leq 1$. Plus précisément si $m_i/f_i \notin [p'(m/n), p(m/n)]$, alors le nombre de copies sera mis à jour. On présente dans l'algorithme 10 les détails de `Copies-management`.

Après avoir effectué les mises à jour, R va avertir les nœuds des nouveaux emplacements des copies en envoyant un message $\langle \textbf{update}, \mathcal{P} \rangle$. Il va ensuite remettre à 0 toutes les valeurs du vecteur $\mathbf{m}$.

Entrée: $p, p' \in \mathbb{Q}$
  pour tout objet o_i **faire**
    si $m_i/f_i > p \cdot m/n$ **alors**
      ajouter $\lfloor m_i \cdot n/m \rfloor - f_i$ copies de manière aléatoire et uniforme
    si $m_i/f_i < p' \cdot m/n$ **alors**
      supprimer $\max\{0, f_i - 2 - \lfloor m_i \cdot n/m \rfloor\}$ copies de manière aléatoire et uniforme.

Algorithme 10 – Algorithme `Copies-management` dans le modèle de flux

5.3.3 Allocation de requêtes

Contrairement à la proposition présentée dans le chapitre 4, nous utilisons un algorithme similaire `AvrDeg` pour l'allocation de toutes les requêtes reçues, quelle que soit la popularité des objets demandés.

Chaque nœud u va communiquer régulièrement sa charge $\ell^*(u)$ à la racine. Après chaque message $\langle \textbf{load}, \ell^*(u) \rangle$ reçu, la racine va recalculer la valeur ℓ^*_{avr} correspondant à la charge moyenne des nœuds. Soit α_R la dernière valeur diffusée par la racine et $c \geq 1$ un paramètre de l'algorithme, si $\ell^*_{avr} \notin [\frac{\alpha_R}{c}, c \cdot \alpha_R]$ alors la racine va diffuser ℓ^*_{avr} et faire $\alpha_R \leftarrow \ell^*_{avr}$.

Lorsqu'un nœud u reçoit un message de diffusion $\langle \textbf{broadcast}, \ell^*_{avr} \rangle$, il teste si $\ell^*(u) \leq (1+\epsilon)\ell^*_{avr}$. Si c'est le cas, il *s'active* et s'alloue toutes les requêtes de sa file F_u et envoie aux autres nœuds candidats un message de suppression $\langle \textbf{delete}, q \rangle$ pour toute requête $q \in F_u$. Ce test d'activation est également effectué lors de la réception d'un message de suppression.

Nous nous référerons à cet algorithme par `AvrDeg-online` dans la suite du manuscrit.

5.3.4 Détails des algorithmes

Nous présentons maintenant le détail des algorithmes en se plaçant respectivement du point de vue d'un nœud quelconque u (Algorithme 11) et de la racine du système (Algorithme 12). Pour l'algorithme de la racine, on ne décrit que les actions supplémentaires qui ne sont pas effectuées par les autres nœuds.

5.4 Évaluation expérimentale

Dans les expériences présentées dans cette section, nous nous intéressons au comportement de nos algorithmes en fonction de la distribution des requêtes émises par le client mais également en fonction de la fréquence à laquelle le client injecte les requêtes sur le système.

La fréquence d'injection joue bien évidemment un rôle primordial sur le temps de réponse aux requêtes. Soit π le temps de traitement d'une requête par un nœud et

Entrée: ϵ, T_ℓ
  `/* Coordination */`
  pour toute requête q_o à coordonner **faire**
    envoyer q_o à chaque nœud $v \in P(o)$
    envoyer $\langle \textbf{received}, q_o \rangle$ à R
  `/* Naive */`
  si $F_u \neq \emptyset$ **alors**
    $curr_u \leftarrow F_u[0]$
    pour tout nœud $v \in P(curr_u)$ **faire**
      envoyer $\langle \textbf{delete}, curr_u \rangle$ à v
    traiter $curr_u$
  `/* Traitement des messages */`
  pour tout message $\mathcal{M}$ reçu **faire**
    si $\mathcal{M} = \langle \textbf{delete}, q \rangle$ **alors**
      si $curr_u \neq q$ **alors**
        supprimer q de F_u
    si $\mathcal{M} = \langle \textbf{update}, \mathcal{P} \rangle$ **alors**
      mettre à jour le placement d'objet
    si $\mathcal{M} = \langle \textbf{broadcast}, \ell^*_{avr} \rangle$ **alors**
      si $\ell^*(u) < (1+\epsilon)\ell^*_{avr}$ **alors**
        pour tout $q \in F_u$ **faire**
          envoyer $\langle delete, q \rangle$ à tout nœud $v \in P(q)$
  `/* Envoi de la charge */`
  chaque T_ℓ unité de temps **faire**
    envoyer $\langle \textbf{load}, \ell^*(u) \rangle$ à la racine.

Algorithme 11 – Algorithme local d'un nœud u

Entrée: ϵ, c, T_{pop}, p, p'
  `/* Initialisation */`
  pour tout objet o_i **faire**
    $m_i = 0$
  pour tout nœud u **faire**
    $L[u] = 0$
  `/* Traitement des messages */`
  pour tout message $\mathcal{M}$ reçu **faire**
    si $\mathcal{M} = \langle \textbf{load}, \ell^*(u) \rangle$ **alors**
      $L[u] \leftarrow \ell^*(u)$
      calculer la charge moyenne ℓ^*_{avr}
      si $\ell^*_{avr} \notin [\frac{\alpha_R}{c}, c \cdot \alpha_R]$ **alors**
        envoyer $\langle \textbf{broadcast}, \ell^*_{avr} \rangle$ à tous les nœuds.
        $\alpha_R \leftarrow \ell^*_{avr}$
    si $\mathcal{M} = \langle \textbf{coord}, q_{o_i} \rangle$ **alors**
      $m_i \leftarrow m_i + 1$
  `/* Gestion des copies */`
  chaque T_{pop} unités de temps **faire**
    exécuter `Copies-management`(p,p')
    réinitialiser le vecteur **m**
    envoyer $\langle update, \mathcal{P} \rangle$ à tous les nœuds

Algorithme 12 – Algorithme local de la racine R

supposons pour simplifier que ce temps de traitement soit identique pour toutes les requêtes indépendamment du nœud qui va les traiter. Un nœud pourra donc traiter au plus $1/\pi$ requêtes par seconde. Si le système a n nœuds, alors on pourra traiter au mieux n/π requêtes par seconde. Supposons maintenant que le client injecte plus de n/π requêtes par seconde, il clair que le système, même dans l'idéal, ne pourra pas tout traiter en une seconde et devra mettre en attente un certain nombre de requêtes. On dit dans ce cas que le système est *saturé*. Cette notion de saturation est fondamentale car elle peut générer des phénomènes de mise en tampon plus ou moins important si les requêtes sont mal réparties entre les nœuds.

Après avoir présenté les modifications apportées à Cassandra et avoir donné quelques précisions sur l'implémentation de nos algorithmes, nous présentons en détails le protocole expérimental mis en place. Nous présentons ensuite les différentes expériences en fonction de la fréquence d'injection et donc de la saturation du système.

5.4.1 Modification de Apache Cassandra

Pour nos expériences, nous avons choisi la version 2.1.10 de Apache Cassandra à laquelle nous avons apporté un certain nombre de modifications.

Paramétrage du partitionnement. Nous utilisons la clé primaire de chaque ligne de la table pour le partitionnement. Un *objet* correspond donc à une seule ligne et chaque objet sera stocké de manière aléatoire grâce à la fonction de hachage `MurmurHash`. C'est la clé primaire qui sera également utilisée pour retrouver un objet dans la base.

Placement aléatoire des copies. Comme nous avons pu le voir, le placement des copies de Cassandra est déterministe et consiste à choisir des nœuds d'identifiants successifs, impliquant une forte probabilité de stocker des objets en commun. Il devient donc aisé pour un adversaire de surcharger le système en concentrant les requêtes sur ces objets communs. Afin d'éviter ce genre de phénomènes, nous avons ajouté un placement aléatoire pour chacune des copies d'un objet en utilisant une fonction de hachage différente. Afin de garantir que les emplacements de chaque copie d'un même objet sont différents, nous utilisons une nouvelle fonction de hachage tant qu'il y a des collisions [2].

File d'attente. Initialement, les nœuds du système n'ont pas de files d'attente de requêtes, indispensable pour l'utilisation des algorithmes présentés dans le Chapitre 4. Nous avons donc rajouté cette fonctionnalité en permettant notamment de mettre

2. Plus précisément on utilise la fonction `MurmurHash` en modifiant la graine du générateur aléatoire.

à jour ces files d'attente. Un nœud est en mesure de distinguer si une requête est simplement en attente ou bien si elle est déjà alloué.

5.4.2 Implémentation des algorithmes

La racine. Dans les algorithmes présentés, un nœud racine doit être choisi afin de mettre à jour les copies et diffuser la charge moyenne courante. Nous avons choisi de laisser ce rôle à un nœud *seed* du système[3]. Ainsi, les nœuds vont communiquer leur charge et le nombre de requêtes reçues par objet.

Pour la gestion des copies (*cf.* Algorithme 10), on fixe le paramètre p d'ajout de copies (resp. le paramètre p' de suppression) à 1 (resp. 1/2) dans la plupart des expériences (tout changement de paramètre sera spécifié). La seed maintient 2 vecteurs, disons $\mathbf{m}_1$ et $\mathbf{m}_2$, qui représente chacun l'historique des requêtes sur 10 secondes. Au bout de 10 secondes, on commence à faire le bilan sur la popularité des requêtes enregistrées par $\mathbf{m}_1$. Lorsque les 10 secondes suivantes sont terminées et que le vecteur $\mathbf{m}_2$ a été rempli, on réinitialise $\mathbf{m}_1$ puis on lance le bilan sur $\mathbf{m}_2$. On réitère ainsi l'opération jusqu'à la fin de l'expérience.

Afin de limiter la mémoire nécessaire pour ces vecteurs, nous utilisons l'algorithme `SpaceSaving` [MAE05] afin de conserver les n objets les plus populaires. Cet algorithme connu de calcul d'items fréquents dans un flux offre de bonnes garanties sur l'estimation des objets populaires, malgré une mémoire limitée.

La charge moyenne ℓ^*_{avr} est mise-à-jour à chaque réception de message et on fixe le paramètre c de diffusion à 2.

Optimisation de `AvrDeg-online` On remarque que plusieurs nœuds peuvent être activés "en même temps" et donc traiter les même requêtes simultanément, ce qui représente un gaspillage de ressources. Lorsqu'un nœud u déjà activé reçoit un message de suppression pour une requête q de la part d'un nœud v, u va utiliser une fonction de hachage h' sur l'identifiant de q. Soit $h'(q)$ cette valeur, u va supprimer la requête q de sa file dans l'un des 2 cas suivants :

— si $h'(q)$ est pair et que $u > v$
— si $h'(q)$ est impair et que $u < v$

Le nombre de messages échangés reste le même mais le nombre de requêtes traitées va sensiblement diminuer.

5.4.3 Protocole expérimental

Nous comparons l'algorithme `Alloc-Cassandra` pour l'équilibrage de charge dans Cassandra avec l'algorithme simple de défilement `Naive` prit séparément et notre algorithme d'allocation `AvrDeg-online` combiné à `Naive`.

3. Dans notre cas, il y aura une seule seed. Cependant, on peut facilement adapter nos algorithmes en utilisant plusieurs seeds afin d'améliorer la tolérance aux pannes.

Les expériences ont été réalisées sur un cluster de 32 nœuds de Amazon EC2. Notre client utilise 4 threads en parallèle afin d'injecter les requêtes sur le système. Le premier nœud de contact est choisi en utilisant l'algorithme du tourniquet.

Dans tous les cas, une requête consiste à chercher un objet selon sa clé primaire et à retourner la valeur d'un des attributs de l'objet.

On considère différents scénarios utilisant soit des jeux de données réelles, soit des jeux de données synthétiques. Un résumé de ces scénarios est donné dans le Tableau 5.1. On précise que le facteur de réplication est 2 par défaut dans tous ces scénarios. Un changement de facteur lors d'une expérience sera précisé. La fréquence d'injection des requêtes par le client sera spécifié au début de chaque expérience.

Dans le premier scénario Power law, le système stocke 256 objets de taille 128 Mo et le client envoie au total 2048 requêtes en lecture. Les requêtes suivent une loi de Zipf[4] de paramètre 2.

Dans le scénario CalgaryL, on utilise une trace réelle d'activité du serveur de l'université de Calgary au Canada. Le jeu de donnée peut être récupéré à l'adresse `http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html`. Ici le nombre d'objets est 8373 et chaque objet est de taille 13 Ko. La trace contient 568346 requêtes avec une distribution légèrement hétérogène (quelques rares objets concentrent une majorité de requêtes, probablement la page d'accueil du site, pour les autres la demande est homogène). Toutefois le temps de traitement ($<$ 1 milliseconde) est ici trop faible pour que l'impact de nos algorithmes soit mesurable. Nous avons donc augmenté la taille des objets à 128 Mo. Afin de limiter l'utilisation mémoire, nous avons gardé les 256 objets les plus populaires, tout en conservant la même distribution de requêtes.

Nous avons simulé un comportement adversarial en générant un jeu de 2048 requêtes qui se concentrent sur des objets en commun entre 2 nœuds. La taille et le nombre des objets sont identiques aux scénarios Power law et CalgaryL. Ce dernier scénario est appelé Adversary.

Nous étudions un autre scénario d'adversaire que l'on nomme AdversaryMedium. Ici les objets sont de plus petites tailles (env. 10 Mo) et le client envoie au total 100000 requêtes. Dans ce scénario, seulement 4 objets stockés par 2 nœuds sont demandés et on applique une distribution en loi de puissance pour les requêtes sur ces 4 objets.

Enfin, nous étudions le comportement de nos algorithmes dans un contexte dynamique grâce au scénario Dynamic. Les objets sont de taille 10 Mo et le client envoie au total 400000 requêtes. L'évolution de la distribution des requêtes est la suivante :

— 80000 requêtes avec une distribution uniforme
— 80000 requêtes avec une loi de puissance de paramètre 2

4. La fréquence des requêtes pour le $i^{\text{ème}}$ objet le plus populaire est $\frac{1/i^2}{H_{\bar{m},2}}$, où $\bar{m}$ correspond au nombre d'objets stockés et $H_{\bar{m},2}$ correspond au $\bar{m}^{\text{ème}}$ nombre harmonique généralisé.

Nom	#objets	taille	#requêtes	temps de traitement
Power law	256	128 Mo	2048	350 ms
CalgaryL	256	128 Mo	2048	350 ms
Adversary	256	128 Mo	2048	350 ms
AdversaryMedium	256	10 Mo	100000	30 ms
Dynamic	256	10 Mo	400000	30 ms

TABLE 5.1 – Récapitulatif des scénarios utilisés lors des expériences. On donne le nom et le symbole de chaque scénario, le nombre d'objets ainsi que leur taille et enfin le nombre de requêtes envoyées au total par le client. Nous précisons également le temps de traitement moyen d'une requête (en ms).

— 80000 requêtes avec une distribution uniforme
— 80000 requêtes avec une loi de puissance de paramètre 2
— 80000 requêtes avec une distribution uniforme

5.4.4 Résultats

Dans les résultats que nous présentons, nous nous intéressons aux mesures de performances suivantes :

— **Charge maximum.** Comme défini précédemment, la charge maximum correspond au nombre de requêtes total allouées au maximum à un nœud avec un algorithme d'allocation .
— **Temps de traitement total.** On mesure l'intervalle de temps entre l'envoie de la première requête par le client et la réception de la dernière réponse.
— **Temps moyen passé dans le système.** Ici on mesure la durée entre le moment où le client envoie sa requête au nœud coordinateur et le moment où il reçoit la réponse à cette requête.
— **Temps moyen passé dans les files.** On mesure la durée entre le moment où une requête est insérée dans une file et le moment où elle est traitée par le nœud.
— **Débit.** Le débit correspond au nombre de requêtes traitées par le système par unité de temps (1 seconde par défaut).

5.4.4.1 Variation de la fréquence d'injection des requêtes

Nous avons fait varier la fréquence d'injection des requêtes par le client dans le scénario DYNAMIC (*cf.* Figure 5.3). Nous avons représenté le temps moyen dans le système et dans les files pour une requête, le temps total de traitement ainsi que le ratio entre le temps total et le temps pour injecter toutes les requêtes. Cette dernière mesure nous donne une indication supplémentaire sur le niveau de saturation du système. On précise que nous avons utilisé une échelle logarithmique (base 10) pour les courbes sur le temps moyen.

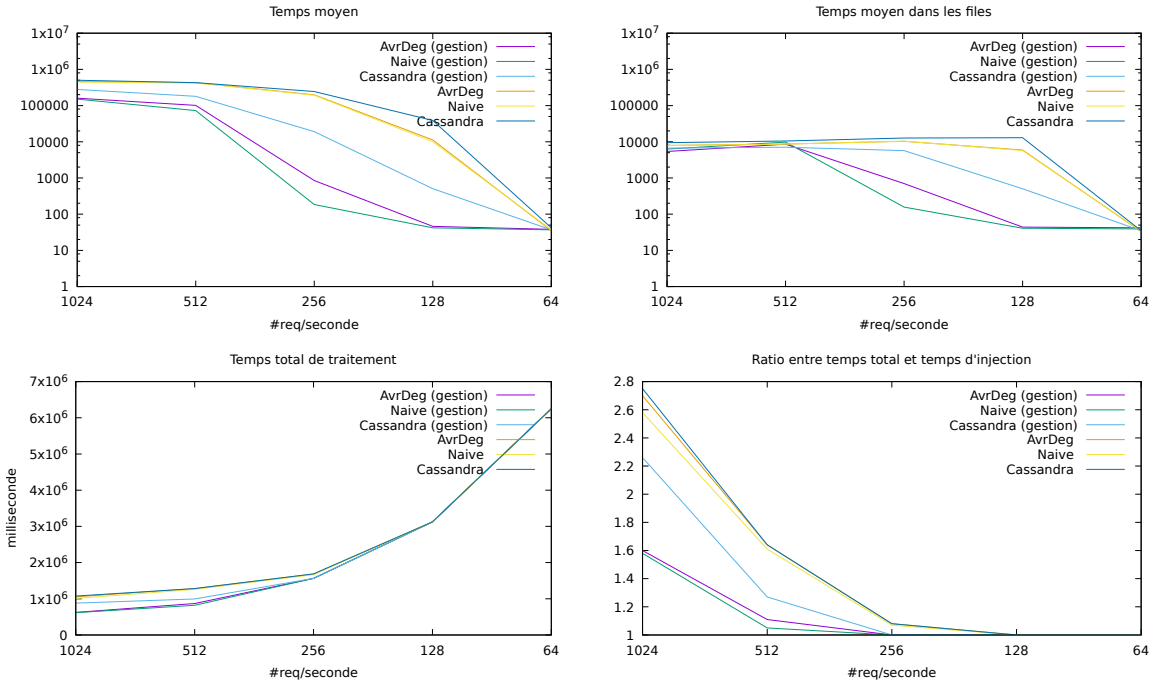


FIGURE 5.3 – Étude de la fréquence d'injection de requêtes sur le temps moyen passé sur le système et dans les files (en milliseconde) par une requête dans le scénario DYNAMIC.

Pour des fréquences d'injection faibles ($\leq$ 64 requêtes par seconde), le temps moyen dans le système et dans les files pour une requête sont similaires. Informellement, les nœuds sont capables de traiter le flux de requêtes à la volée, il n'y a donc pas de mise en tampon au niveau du coordinateur. De même, le temps de traitement total est très proche, car il est dominé par le temps d'injection des requêtes. Ici la différence entre les algorithmes est quasi inexistante.

En augmentant un peu la fréquence d'injection (jusqu'à 256 requêtes par seconde), l'apport de la gestion reste peu visible sur le temps de traitement total. Par contre, les écarts de temps de réponse d'une requête avec ou sans gestion sont énormes. Dans le cas où la fréquence est de 256 requêtes par seconde par exemple, la gestion de copies permet d'obtenir un temps moyen dans le système proche du temps moyen dans les files (env. 1 sec.). Pour les mêmes algorithmes sans gestion le temps moyen dans le système est de l'ordre de 200 secondes contre moins de 10 secondes pour le temps moyen dans les files. Ce cas particulier est traité plus en détails dans la sous-sous-section suivante (*cf.* Sous-sous-section 5.4.4.2).

Enfin, lorsqu'on commence à saturer le système, la gestion de copies permet d'améliorer les performances mais les écarts entre les algorithmes sont moins flagrants. On note par contre une amélioration sur le temps de traitement total avec la gestion. Nous traitons plus en détails le cas du système saturé dans la Sous-sous-section 5.4.4.3 en se plaçant dans différents scénario.

5.4.4.2 Système non saturé

Dans un premier temps, nous nous intéressons à l'impact de la gestion de copies lorsque le système n'est pas saturé. Ici, le temps de traitement total sera peu impacté par les choix des algorithmes, car le temps d'injection est faible et domine cette mesure. Nous mettrons plutôt en avant le temps de réponse d'une requête.

Distribution de requêtes évolutive Focalisons-nous sur le cas plus spécifique où le client injecte 256 requêtes par seconde. Nous avons détaillé sur le Tableau 5.2 le temps moyen et maximum passé sur le système par une requête et le temps moyen passé dans les files. Dans la Figure 5.4, on présente le débit obtenu tout au long de l'expérience pour l'algorithme `AvrDeg-online` ainsi que la taille des files en moyenne et le nombre de files vides. Les résultats pour les autres algorithmes ont été laissés en annexe (*cf.* Annexe B).

On constate tout d'abord que la gestion de copies permet d'obtenir un débit très régulier proche de 256 requêtes/seconde, ce qui correspond d'ailleurs à la fréquence d'injection des requêtes. On remarque toutefois quelques piques de performance pour la version de l'algorithme sans gestion. Ce phénomène est probablement dû à la mise en tampon de requêtes au niveau du coordinateur. On peut avancer les arguments suivants pour justifier cette hypothèse. Rappelons tout d'abord que la distribution des requêtes change après l'injection de 80000 requêtes, soit toutes les 312 secondes environ. Lors de la première période, la moyenne des tailles de files est faible car la

	moyenne (système)		moyenne (files)		temps total	
Gestion	oui	non	oui	non	oui	non
`Alloc-Cassandra`	19082	244361	5716	12714	1562526	1687538
`AvrDeg-online`	850	198368	702	10320	1562538	1685254
`Naive`	185	192928	157	10416	1562526	1667875

TABLE 5.2 – Mesures (en milliseconde) du temps total de traitement et du temps moyen passé dans le système et dans les files par une requête lorsque la fréquence d'injection est de 256 requêtes/seconde.

distribution est uniforme et les requêtes sont bien réparties [5]. Lors de la deuxième période par contre, la taille moyenne est beaucoup plus élevé mais le nombre de files non vides reste faible, car peu de nœuds sont concernés par les requêtes reçues. Le coordinateur va donc accumulé ces requêtes populaires, ne pouvant transmettre aux nœuds qu'un nombre limité de requêtes. Cette accumulation continue probablement pendant la troisième période uniforme tant que les requêtes populaires n'ont pas pu être distribués. Le débit augmente ainsi petit à petit car les coordinateurs distribuent les requêtes uniformes. On remarque également quelques irrégularités pour la version avec gestion qui peuvent être expliquées soit par une instabilité possible du cluster où ont été lancées les expériences, soit par des phénomènes de gestion de files au niveau du coordinateur que nous ne maîtrisons pas. De nouvelles expériences seraient nécessaires afin de déterminer avec précision les causes de ces anomalies.

Le temps de traitement total n'est pas vraiment amélioré par la gestion de copies. En effet, comme il faut 1562 secondes pour injecter toutes les requêtes, c'est ce temps qui va dominer dans les mesures. On constate par contre que le temps moyen pour répondre à une seule requête est nettement amélioré pour tous les algorithmes. A titre d'exemple, le temps moyen sur le système avec `AvrDeg-online` est de 850 millisecondes avec gestion, contre un peu moins de 200000 sans gestion. On remarque enfin que l'algorithme de Cassandra présente de nettement moins bonnes performances que les deux algorithmes, sur toutes les mesures.

Distribution de requêtes unique On présente ici l'impact de notre gestion de copie lorsque la distribution des requêtes ne change pas (*cf.* Figure 5.5). On se focalise en particulier sur des situations où la distribution des requêtes cible un petit ensemble de nœuds. Le facteur de réplication initial est fixé à 2 et la fréquence d'injection est de 24 requêtes par seconde. Dans ce scénario, le temps de traitement d'une requête est en moyenne de 350 ms, ce qui implique que le système peut idéalement traiter $32/0.35 \sim 91$ requêtes par seconde. On se trouve donc bien dans un cas d'étude où le système n'est pas saturé.

Dans POWER LAW, on divise le temps de traitement total de nos algorithmes par

5. On précise qu'une requête en cours de traitement n'est pas comptabilisée dans la file d'un nœud.

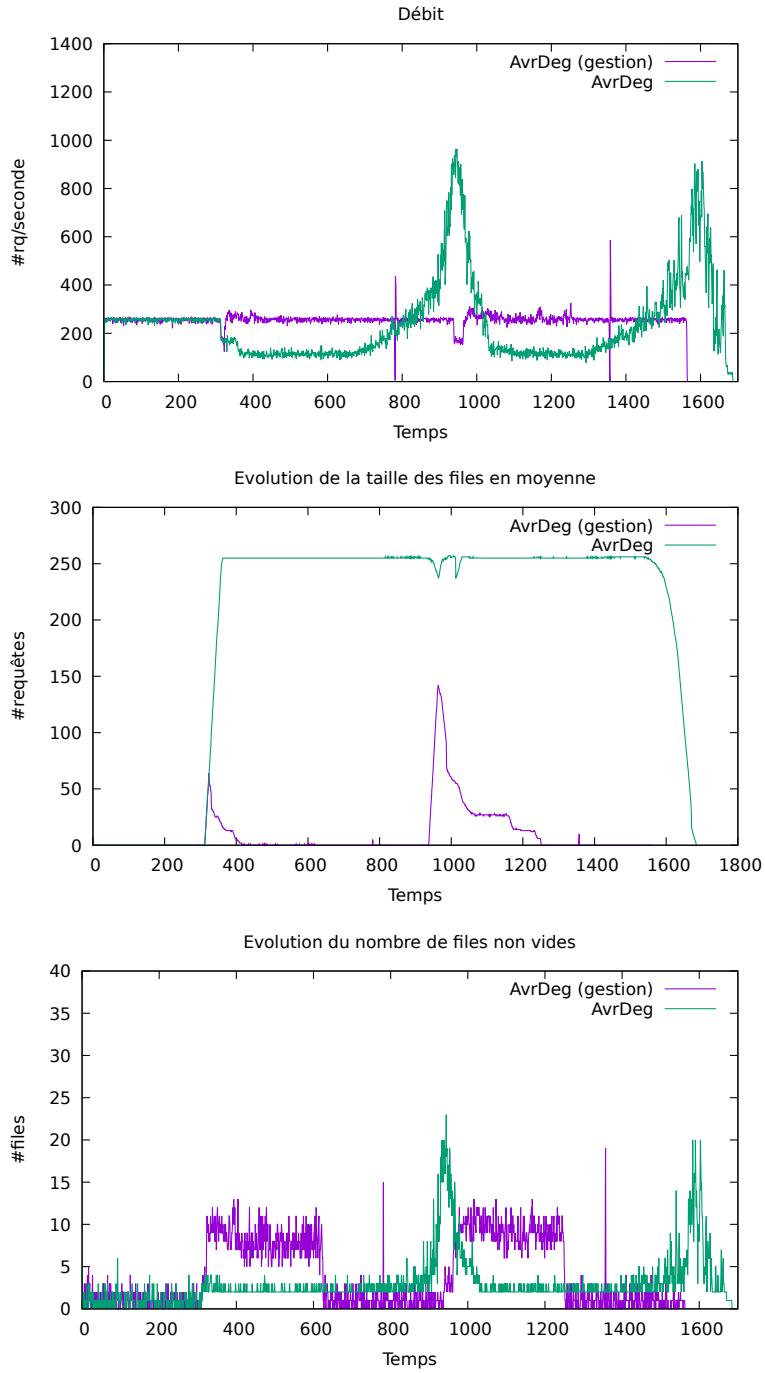


FIGURE 5.4 – Scénario DYNAMIC avec une fréquence d'injection de 256 requêtes/seconde pour l'algorithme `AvrDeg-online`.

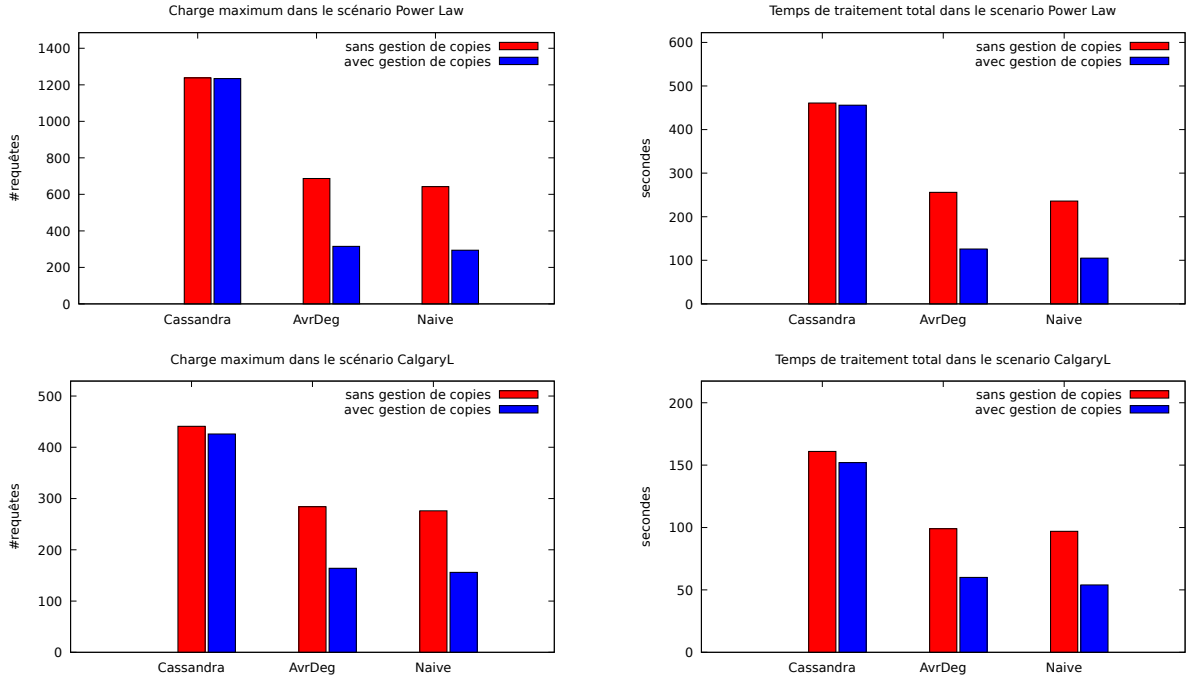


FIGURE 5.5 – Impact de `Copies-management` à la fois sur la charge maximum et sur le temps maximum dans le système.

2 en utilisant la gestion de copies. Le scénario CALGARYL est également favorable à nos algorithmes d'allocation avec des performances similaires à celles obtenues pour le scénario POWER LAW.

Dans les 2 cas, nos algorithmes sont nettement plus performants que `Alloc-Cassandra` (environ 4 fois plus rapide avec la gestion)

On remarque également que la gestion de copies diminue la charge maximum dans les deux scénarios, impliquant une meilleure répartition des requêtes entre les nœuds du système.

5.4.4.3 Saturation du système

Nous avons pu constater sur les résultats présentés que la gestion de copies couplée à nos algorithmes offrait de bonnes performances si le système n'ait pas saturé. Toutefois, les différences semblent s'amenuiser lorsque la fréquence d'injection augmente et que le système sature. Nous avons réalisé un certain nombre d'expériences afin de tester le comportement de nos algorithmes lorsque la fréquence d'injection permet de saturer le système.

Impact du facteur de réplication Nous comparons ici les algorithmes `Naive`, `AvrDeg` et `Alloc-Cassandra` pour un facteur de réplication $f \in [2, 4, 8, 16]$ pour les scénarios ADVERSARY et CALGARYL (voir Figure 5.6). Ici le client envoie 96

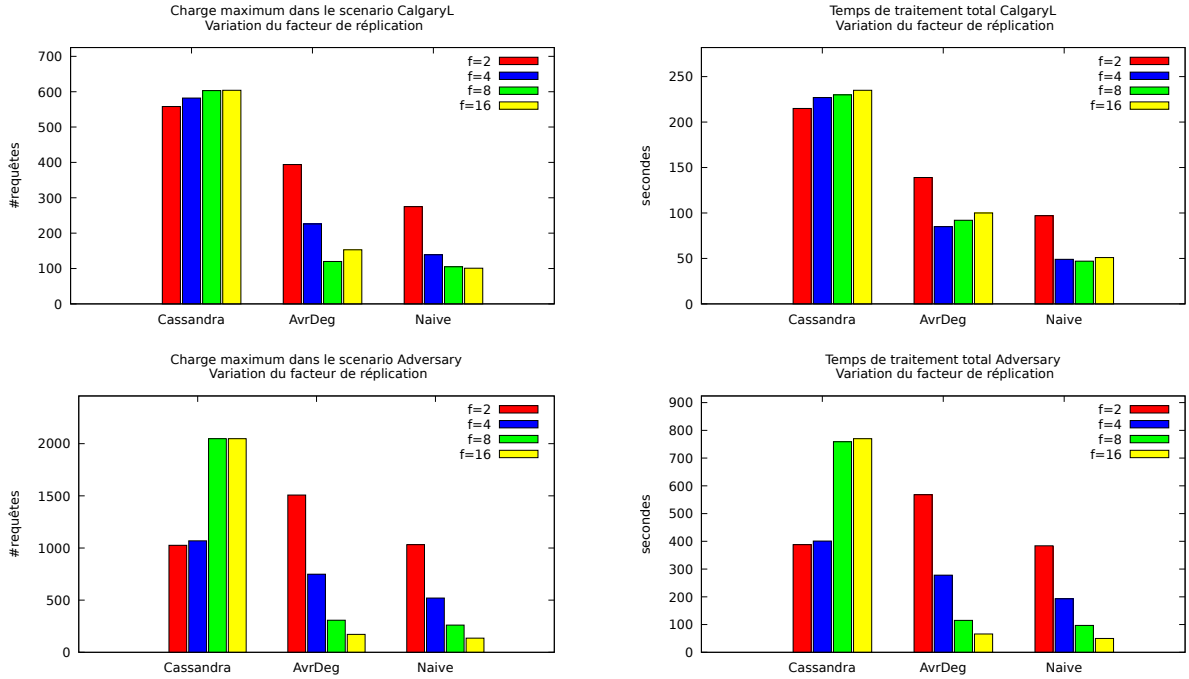


FIGURE 5.6 – Impact du facteur de réplication f sur la charge maximum et le temps maximum.

requêtes par seconde.

Sans surprise, le nombre maximum de requêtes allouées à un nœud diminue proportionnellement à l'augmentation de f dans le scénario ADVERSARY, quelque soit l'algorithme concerné. Il en est de même pour le temps de traitement total.

Pour le scénario CALGARYL, les résultats sont comparables mais on remarque toutefois que l'augmentation du facteur de réplication semble atteindre ses limites, notamment pour `AvrDeg`. En effet, on constate que la charge maximum est plus faible pour $f = 8$ que pour $f = 16$. Pour le temps de traitement total, on atteint la meilleure performance pour $f = 4$. Un phénomène similaire, bien que moins prononcé, peut être constaté pour `Naive`. Une explication possible est que beaucoup de requêtes sont traitées en double lorsque le nombre de copies augmente, car les mises-à-jour des files d'attente n'ont pas le temps d'avoir lieu.

Cette expérience nous permet de mettre en lumière que seulement augmenter le facteur de réplication de manière globale ne garantit pas toujours un meilleur équilibrage des requêtes. De plus, cela représente un surcoût mémoire important, notamment quand les données sont massives.

Modèle d'adversaire Nous présentons une expérience avec le scénario `Adversary-medium` en utilisant notre gestion de copies (*cf.* Figure 5.7 pour les résultats). La fréquence d'injection est de 2400 requêtes par seconde.

On s'intéresse ici à la charge maximum d'un nœud et au temps de traitement

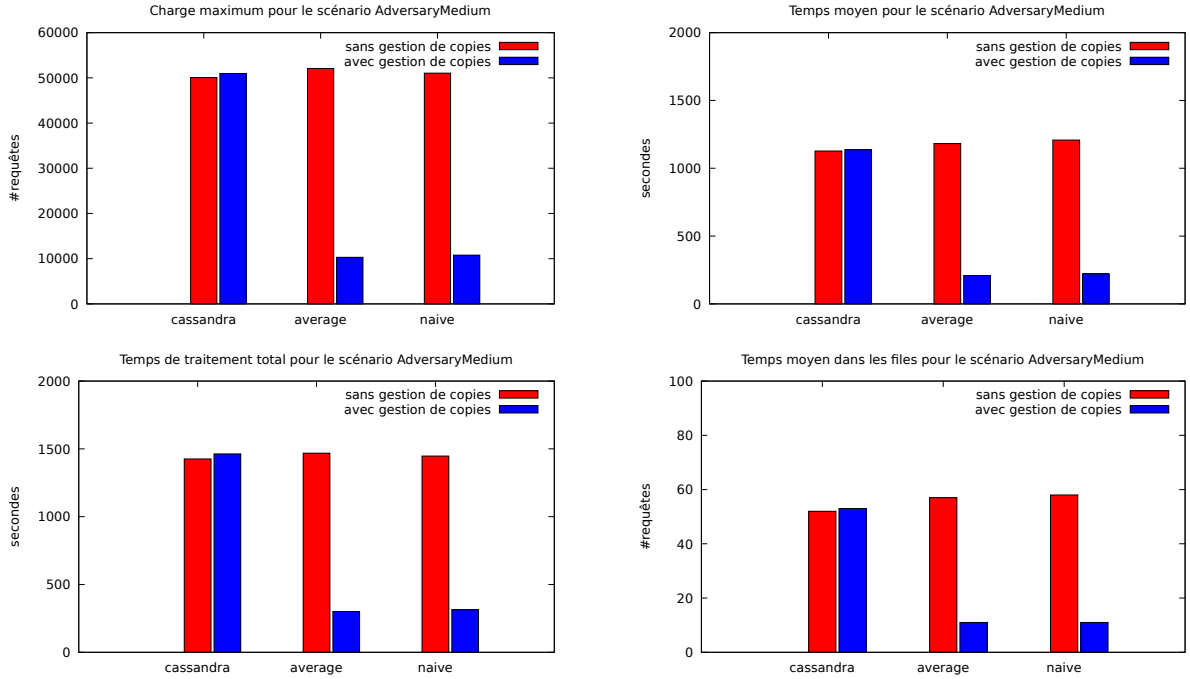


FIGURE 5.7 – Impact de `Copies-management` dans le scénario ADVERSARYMEDIUM.

Algorithmes	**Temps moyen (système)**		**Temps moyen (files)**	
	sans gestion	avec gestion	sans gestion	avec gestion
`Alloc-Cassandra`	1127	1137	52	53
`AvrDeg-online`	1182	209	57	11
`Naive`	1208	221	58	10

TABLE 5.3 – Mesures (en seconde) du temps moyen passé dans le système et dans les files par une requête avec le scénario ADVERSARYMEDIUM.

total comme précédemment. De plus, nous avons mesuré le temps moyen passé dans le système et dans les files par une requête.

Les résultats sont sans appel : avec notre gestion de copies, les algorithmes `AvrDeg` et `Naive` sont environ 5 fois plus rapide et la charge maximum est également divisée par 5. On notera toutefois que le temps moyen passé dans les files d'attente par une requête est très nettement inférieur à celui passé en moyenne dans le système (*cf.* Tableau 5.3). Cet écart surprenant est probablement du à des phénomènes de mise-en-attente au niveau du nœud coordinateur causé par la saturation du système.

Distribution de requêtes évolutive Dans cette section, nous considérons le scénario DYNAMIC. La fréquence d'injection est de 2400 requêtes par seconde. Enfin, nous avons modifié légèrement la gestion de copies. Nous fixons le paramètre p

d'ajout de copies à 2 et le paramètre p' de suppression de copies à 1.

Les résultats sont rassemblés sur la Figure 5.8. On constate que le temps de traitement total, le temps moyen passé sur le système et le temps moyen passé dans les files par une requête est nettement amélioré avec la gestion de copies. Cette tendance est confirmée par l'étude du débit en fonction des différents algorithmes, où le système traite environ 2 fois plus de requête par seconde tout au long de l'expérience. Nous avons également mesuré le nombre de copies additionnelles présentes sur le système. Comme nous l'avions démontré en théorie, très peu de copies additionnelles (au maximum 7) sont ajoutées.

Optimisation de l'algorithme d'allocation Pour finir, nous présentons une dernière expérience permettant d'évaluer l'intérêt de l'optimisation de `AvrDeg-online` présenté dans la sous-section 5.4.2. Cette expérience a été réalisée dans le scénario CalgaryL et nous présentons le nombre de requêtes allouées à chaque nœud, avec ou sans l'optimisation. On note que l'amélioration est sensible avec l'optimisation, ce qui permet d'obtenir des performances similaires à `Naive`. Les résultats détaillés sont présentés dans la Figure 5.9.

Nous mentionnons que le nombre de requêtes mesuré pour le nœud 2 ne correspond pas uniquement à des requêtes de lecture. En effet, certaines requêtes émises par le système ont également été comptabilisées ici, et nous ne sommes pas en mesure de les distinguer. Nous avons toutefois borné leur nombre entre 1 et 4, ce qui est donc négligeable. Les mesures pour les autres nœuds sont exactes.

5.5 Conclusion des expériences

Au travers de ces expériences, nous avons voulu vérifier à la fois l'apport de l'algorithme `Copies-management` mais également de nos algorithmes d'allocation sur le temps total de traitement d'un ensemble de requête mais également sur le temps de réponse d'une requête. Nous avons pris en compte la saturation du système lors des mesures effectuées.

Nous avons pu constater que la gestion de copies permet un gain de performance colossal sur le temps de réponse d'une requête pour les différents algorithmes étudiés lorsque le système n'est pas trop saturé. On obtient également une meilleure régularité du débit même si la distribution des requêtes évolue. On remarque également que les algorithmes `AvrDeg-online` et `Naive` se comportent nettement mieux que `Alloc-Cassandra` dans les différentes expériences présentées.

Nous avons ensuite réalisé plusieurs expériences en saturant le système. Nous avons pu confirmer qu'augmenter le facteur de réplication n'est pas toujours suffisant pour améliorer le temps de complétion. Dans certains cas, on observe même une baisse de performance du à un grand nombre de requêtes traitées plusieurs fois, alors que le surcoût en terme de stockage est important. Notre algorithme de gestion de copies prend donc tout son sens, d'autant plus que le nombre de copies additionnelles

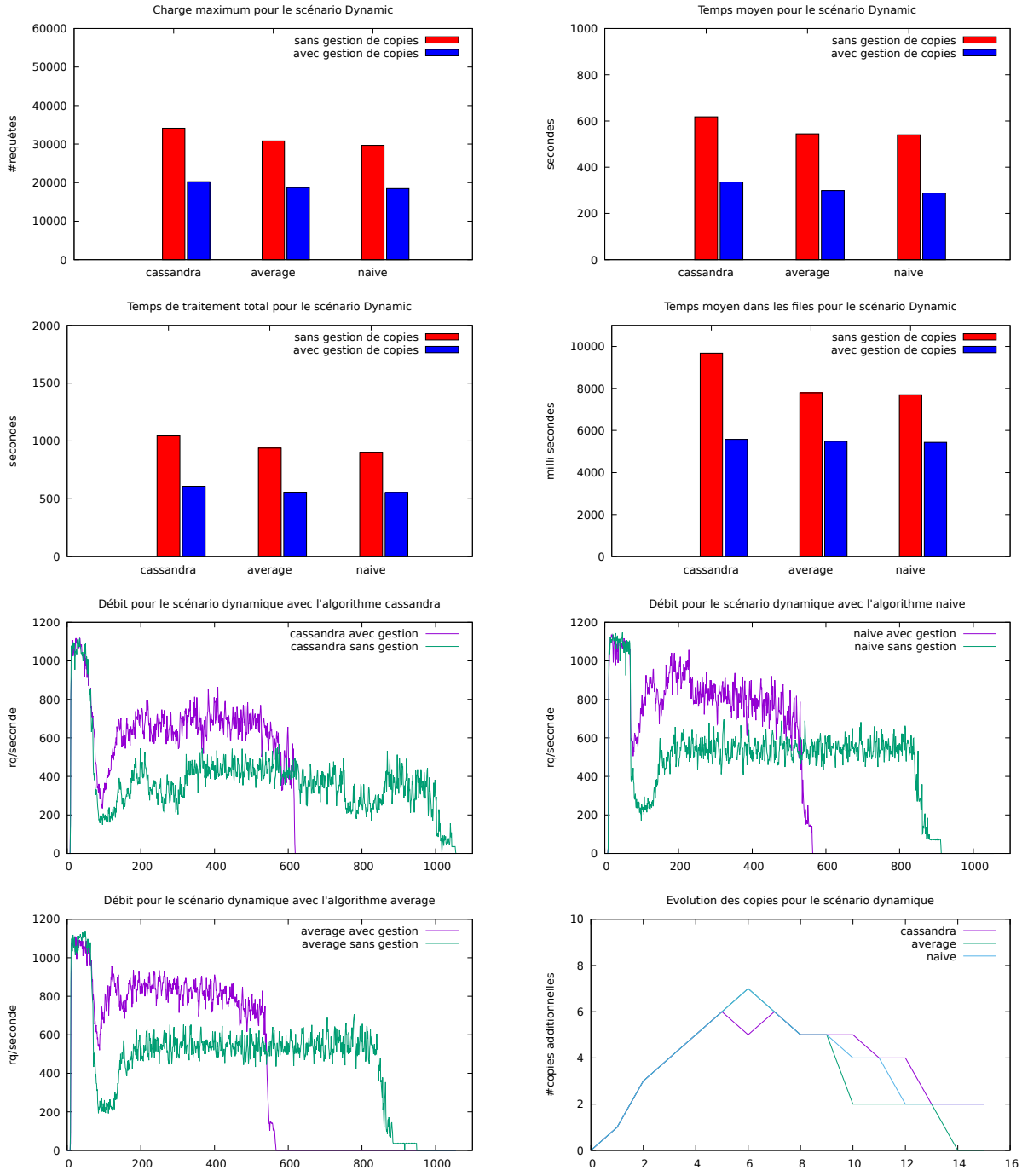


FIGURE 5.8 – Impact de `Copies-management` dans le scénario DYNAMIC.

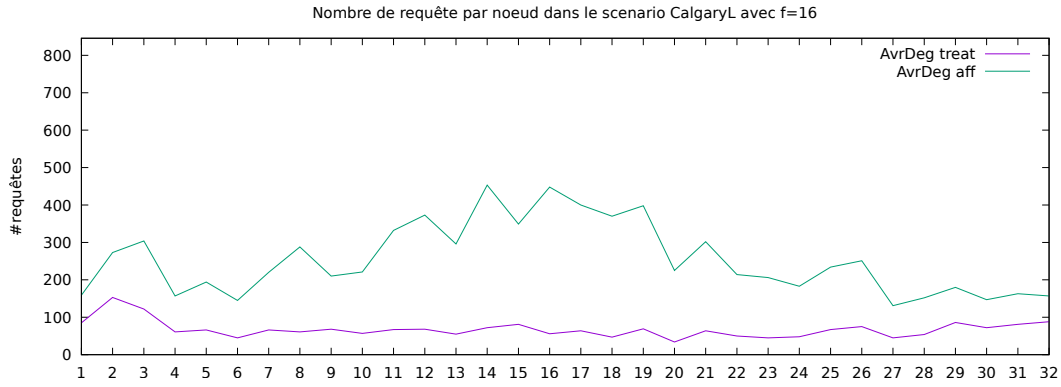


FIGURE 5.9 – Mesure de l'optimisation réalisé lors de l'implémentation de `AvrDeg`. `AvrDeg treat` représente le nombre de requêtes affectivement traitées par un nœud avec `AvrDeg` optimisé et `AvrDeg aff` représente le nombre de requêtes affectées (et qui aurait donc été traitées sans l'optimisation).

est négligeable comparativement au nombre total d'objets stockés par le système. On note une réelle amélioration des performances dans les différents scénarios étudiés lorsque le temps de traitement d'une requête est important. Notre gestion de copies est également intéressante dans un contexte dynamique où la distribution des requêtes évolue avec le temps.

Pour finir, on peut remarquer que l'algorithme `Naive`, bien que très simple, a des performances similaires voir meilleures que `AvrDeg-online` dans certains scénarios. Ce phénomène peut être en parti expliqué par le fait que les requêtes sont transférées de manière asynchrone, ce qui induit que l'ordre d'arrivé dans les files est différent. Le message de mise-à-jour des files d'attente étant envoyé avant traitement, de nombreuses requêtes dupliquées sont donc éliminées.

Chapitre 6

Conclusion

6.1 Bilan de la thèse

Nous nous sommes intéressé au problème de l'équilibrage des requêtes pour les systèmes distribués. Plus précisément, on cherche à répartir les requêtes reçues entre les nœuds du système qui peuvent les traiter. Nous nous sommes intéressés plus particulièrement aux systèmes de stockage distribués où chaque nœud stocke un ensemble d'objets qui représentent les données. Dans ce contexte, une requête concerne un *objet* et ne peut être traitée que par un nœud stockant cet objet.

Afin de modéliser ce contexte, nous avons étudié le problème d'allocation qui consiste à trouver un placement d'objets et un algorithme d'allocation afin de minimiser le nombre maximum de requêtes allouées à un nœud. Nous nous sommes concentrés sur le cas où les requêtes ont un poids identique et arrivent de manière distribuée sur le système.

Dans un modèle hors-ligne, nous avons proposé un algorithme asynchrone et tolérant aux pannes permettant de garantir une approximation constante de l'optimal dans le cas spécifique où il y a $f = 2$ copies de chaque objets. Cet algorithme se base sur l'estimation de la charge moyenne maximum. Lorsqu'il y a absence de pannes et que $f > 2$, nous utilisons un algorithme asynchrone similaire utilisant une barrière de synchronisation couplé à un mécanisme de gestion de copies. La gestion de copies va adapter le facteur de réplication de chaque objet en fonction de la demande pour cette objet. Nous utilisons également un placement de copies aléatoire et uniforme. Cet algorithme fournit une f-approximation de l'optimal et nous montrons que si f est suffisamment grand, on peut garantir une f-approximation de l'idéal pour toute distribution de requêtes.

Enfin, nous nous sommes intéressés au cas des flux de requêtes et nous avons adapté les algorithmes proposés pour le cas sans panne dans ce contexte. Nous avons mené une évaluation expérimentale en utilisant la base de données distribuée Apache Cassandra. Nous montrons notamment que nos algorithmes d'allocation couplés à la gestion de copies permettent de nettement améliorer le temps de traitement total des requêtes (quand le système est saturé) et le temps de réponse moyen d'une requête

(quand le système n'est pas saturé) dans la plupart des scénarios considérés.

6.2 Perspectives

Dans le Chapitre 4, nous avons émis l'idée d'étendre nos travaux au cas des requêtes pondérées, remarque qui peut également s'appliquer au modèle de flux présenté dans le Chapitre 5. Pour cela, nous faisons l'hypothèse que le poids d'une requête, autrement dit son *coût*, est connu à l'avance, ce qui est une hypothèse forte en pratique. Une direction possible de recherche consiste donc à se passer de cette hypothèse en se basant par exemple sur une estimation de ce coût.

Nous avons pu constater dans le Chapitre 5 les bonnes performances de l'algorithme différé `Naive` sur l'allocation des requêtes. Cet algorithme est également intéressant car il ne nécessite aucune connaissance sur le système ni sur le coût des requêtes, ce qui en fait un atout pour les systèmes distribués. Des références théoriques existent dans un modèle où les requêtes arrivent par *lots* [ABS98, BCFV00, San03]. Il semble notamment y avoir un lien avec l'algorithme `Greedy` sous certaines conditions. Par contre, ces références portent uniquement sur des requêtes aléatoires. Essayer de mieux comprendre cet algorithme pour des distributions différentes constitue donc une piste de recherche intéressante.

Enfin, étudier théoriquement l'algorithme `AvrDeg-online` constitue une extension naturelle de notre travail. On peut dans un premier temps considérer le modèle des *graphes dynamiques*. Il existe plusieurs travaux sur l'orientation de graphe étudiant la dynamique des arêtes (voir par exemple [Tan14, BF99, KKPS14]). En distribuée, Parter *et al* [PPS16] sont à notre connaissance les seuls à proposer un algorithme distribué qui maintient une $O(a + \log^* n)$-orientation en temps amorti $O(\log^* n)$ par mise-à-jour lorsque que l'arboricité du graphe dynamique est bornée par une constante a pour toute mise-à-jours. Ces algorithmes s'autorisent cependant une ré-orientation des arêtes, contrairement au notre. Il serait donc intéressant de le comparer à cet existant.

Annexe A

Preuves omises

A.1 Preuve du Lemme 4.4

On rappelle que $\bar{m}$ représente le nombre de balles et N représente le nombre d'urnes. Soit $X_i = \sum_{j=1}^{\bar{m}} x_j$ le nombre de balles allouées à l'urne i, où $x_{i,j}$ représente une variable aléatoire binaire prenant la valeur 1 si la balle j est alloué à l'urne i. L'espérance μ de $x_{i,j}$ est toujours $1/n$, chaque balle étant allouée de manière indépendante et uniforme. Dans le reste de cette preuve, nous utiliserons la borne suivante dérivée de la borne de Chernoff.

$$\Pr(X_i > (1+c) \cdot \mathbb{E}(X_i)) < e^{\frac{-c \cdot \mathbb{E}(X_i)}{3}} \tag{A.1}$$

On étudie 4 cas différents.

Cas 1 : $\bar{\mathbf{m}} \leq \mathbf{N}/\log \mathbf{N}$ Ce premier énoncé est prouvé dans le manuscrit de thèse de Mitzenmacher ([Mit96], Lemme 2.13, page 30) et se base sur une approximation de Poisson. Nous donnons juste quelques éléments de preuve ici et laissons le lecteur intéressé consulter le manuscrit pour les détails.

On peut représenter le nombre X_i de balles allouées à une urne par une variable aléatoire Y_i qui suit une distribution de Poisson de paramètre $\frac{\bar{m}}{N}$. La différence majeure avec le cas initial est que ces variables aléatoires de Poisson sont *indépendantes*, ce qui facilite l'analyse. Nous appelons cette représentation la *version de Poisson* du problème initial.

Un premier énoncé ([Mit96], Corollaire 2.12, page 29) montre que tout événement basé sur la charge qui apparaît avec probabilité p dans la version de Poisson apparaît avec probabilité $p \cdot \sqrt{2\pi e \bar{m}}$ dans le cas initial. L'idée est donc d'analyser la probabilité que la charge maximum d'une urne dépasse un certain seuil dans la version de Poisson et de se servir de ce corollaire pour en déduire la probabilité du même événement dans le cas initial.

Notons p_k la probabilité qu'une variable aléatoire Y_i qui suit une distribution de Poisson ait une charge supérieure ou égale à k. En trouvant les bornes appropriées

pour p_k, on peut montrer que pour $k = \Theta(\frac{\log N}{\log(N/\bar{m})})$ on obtient $p_k \leq O(1/N^2)$. Comme $\sqrt{2\pi e\bar{m}} = O(N)$, cette probabilité est $O(1/N)$ dans le cas initial.

Cas 2 : $\mathbf{N/\log N < \bar{m} < N}$ On remarque que X_i suit une distribution binomiale, on en déduit donc l'équation suivante.

$$\begin{aligned}
\Pr(X_i \geq k) &= \sum_{i=k}^{\bar{m}} \binom{\bar{m}}{i} (1/N)^i (1-1/N)^{m-i} \\
&\leq \sum_{i=k}^{\bar{m}} \binom{\bar{m}}{i} (1/N)^i \\
&\leq \sum_{i=k}^{\bar{m}} (\bar{m}^i/k!)(1/N)^i \\
&\leq \sum_{i=k}^{\bar{m}} (\frac{\bar{m}}{N})^i (1/i!)
\end{aligned}$$

Dans le cas $\bar{m} \leq N$, la dernière ligne est une suite géométrique de raison $\bar{m}/N < 1$. On en déduit la borne suivante.

$$\begin{aligned}
\Pr(X_i \geq k) &\leq 2 \cdot (\frac{\bar{m}}{N})^k (1/k!) \\
&\leq 2/k! \leq 2/k^k
\end{aligned}$$

En choisissant $k = \frac{3\log N}{\log\log N}$, on obtient

$$\begin{aligned}
2/k^k &\leq (\frac{2\log\log N}{3\log N})^{\frac{3\log N}{\log\log N}} \\
&\leq \exp(\log(\frac{\log\log N}{\log N})^{\frac{3\log N}{\log\log N}}) \\
&\leq \exp(\frac{3\log N}{\log\log N}\log(\frac{\log\log N}{\log N})) \\
&\leq \exp(\frac{3\log N}{\log\log N}(\log\log\log N - \log\log N)) \\
&\leq \exp(3\log N(\frac{\log\log\log N}{\log\log N} - 1)) \\
&\leq \exp(2\log N) \\
&\leq \frac{1}{N^2}
\end{aligned}$$

En utilisant l'inégalité de Boole pour tous les X_i, on en déduit qu'il n'existe pas d'urne avec une charge supérieure à $\frac{3\log N}{\log\log N}$ avec probabilité $1 - 1/N$.

Cas 3 : $\mathbf{N \leq \bar{m} \leq N \log N}$ L'espérance de toute variable X_i est $\bar{m}/n \geq 1$.Ici nous appliquons l'inégalité de Chernoff (voir A.1).

$$\begin{aligned}\Pr(X_i > (1+c) \cdot \mathbb{E}(X_i)) &< e^{\frac{-c \cdot \mathbb{E}(X_i)}{3}} \\ &\leq e^{\frac{-c}{3}}\end{aligned}$$

En choisissant $c = 6 \log N$ et en appliquant l'inégalité de Boole, on déduit qu'il n'existe pas d'urne de charge supérieure à $O(\frac{\bar{m} \log N}{N})$ avec probabilité $1 - 1/N$.

Cas 4 : $\mathbf{\bar{m} > N \log N}$ Ici aussi il suffit d'appliquer la borne de Chernoff.

$$\Pr(X_i > (1+c) \cdot \mathbb{E}(X_i)) < e^{\frac{-c \cdot \log N}{3}}$$

En choisissant $c = 6$ et en appliquant l'inégalité de Boole, on en déduit qu'il n'existe pas d'urne avec une charge supérieure à $O(\bar{m}/N)$ avec probabilité $1 - 1/N$.

On remarque également qu'on peut prouver facilement que la charge est $\Omega(\bar{m}/N)$ en appliquant le principe des chaussettes et des tiroirs.

A.2 Preuve du Lemme 4.9

On rappelle que le graphe des requêtes est H, m correspond au nombre de requêtes, $\bar{m}$ correspond aux nombres de requêtes distinctes pouvant être reçues et n correspond au nombre de noeud du graphe. On déduit l'énoncé en combinant les Lemmes 4.3 et 4.8.

Cas 1 : $f \geq \frac{\log \bar{m}}{\log n} - 1$

$$\begin{aligned}\Delta(H) &\leq m^{1-1/f} \cdot \left(\frac{M(H)}{f!}\right)^{1/f} \\ &\leq m^{1-1/f} \cdot \left(\frac{M(H) \cdot e^f}{f^f}\right)^{1/f} \\ &\leq \frac{e}{f} \cdot m^{1-1/f} \cdot \left((7 \cdot \frac{\bar{m}}{N} \cdot \log N + 3 \log n) \cdot \frac{m}{n}\right)^{1/f} \\ &\leq 3 \cdot \frac{e}{f} \cdot m^{1-1/f} \cdot \left((8\frac{\bar{m} \cdot f^f}{n^f} \cdot f \cdot \log n) \cdot \frac{m}{n}\right)^{1/f} \\ &\leq 6 \cdot e \cdot \frac{m}{n} \cdot \left(\frac{\bar{m}}{n} \cdot f \cdot \log n\right)^{1/f}\end{aligned}$$

Cas 2 : $f < \frac{\log \bar{m}}{\log n} - 1$

$$\begin{aligned}
\Delta(H) &\leq m^{1-1/f} \cdot \left(\frac{M(H)}{f!}\right)^{1/f} \\
&\leq m^{1-1/f} \cdot \left(\frac{M(H) \cdot e^f}{f^f}\right)^{1/f} \\
&\leq \frac{e}{f} \cdot m^{1-1/f} \cdot \left(7 \cdot \frac{\bar{m}}{N} \cdot \frac{m}{n}\right)^{1/f} \\
&\leq 3 \cdot \frac{e}{f} \cdot m^{1-1/f} \cdot \left(\frac{\bar{m} \cdot f^f}{n^f} \cdot \frac{m}{n}\right)^{1/f} \\
&\leq 3 \cdot e \cdot \frac{m}{n} \cdot \left(\frac{\bar{m}}{n}\right)^{1/f}
\end{aligned}$$

Annexe B

Résultats d'expérience supplémentaires

Dans cette annexe, nous donnons des résultats supplémentaires afin de mieux comprendre les expériences. Nous nous intéressons plus particulièrement à l'expérience dans le scénario DYNAMIC lorsque le client injecte 256 requêtes/seconde. Nous donnons l'évolution du débit pour les algorithmes `Naive` et `Alloc-Cassandra` ainsi que l'évolution du nombre de files vides et la taille des files en moyenne (*cf.* Figure B.1).

Ici aussi, on constate que la gestion de copies permet d'obtenir une bonne stabilité du débit pour ces deux algorithmes.

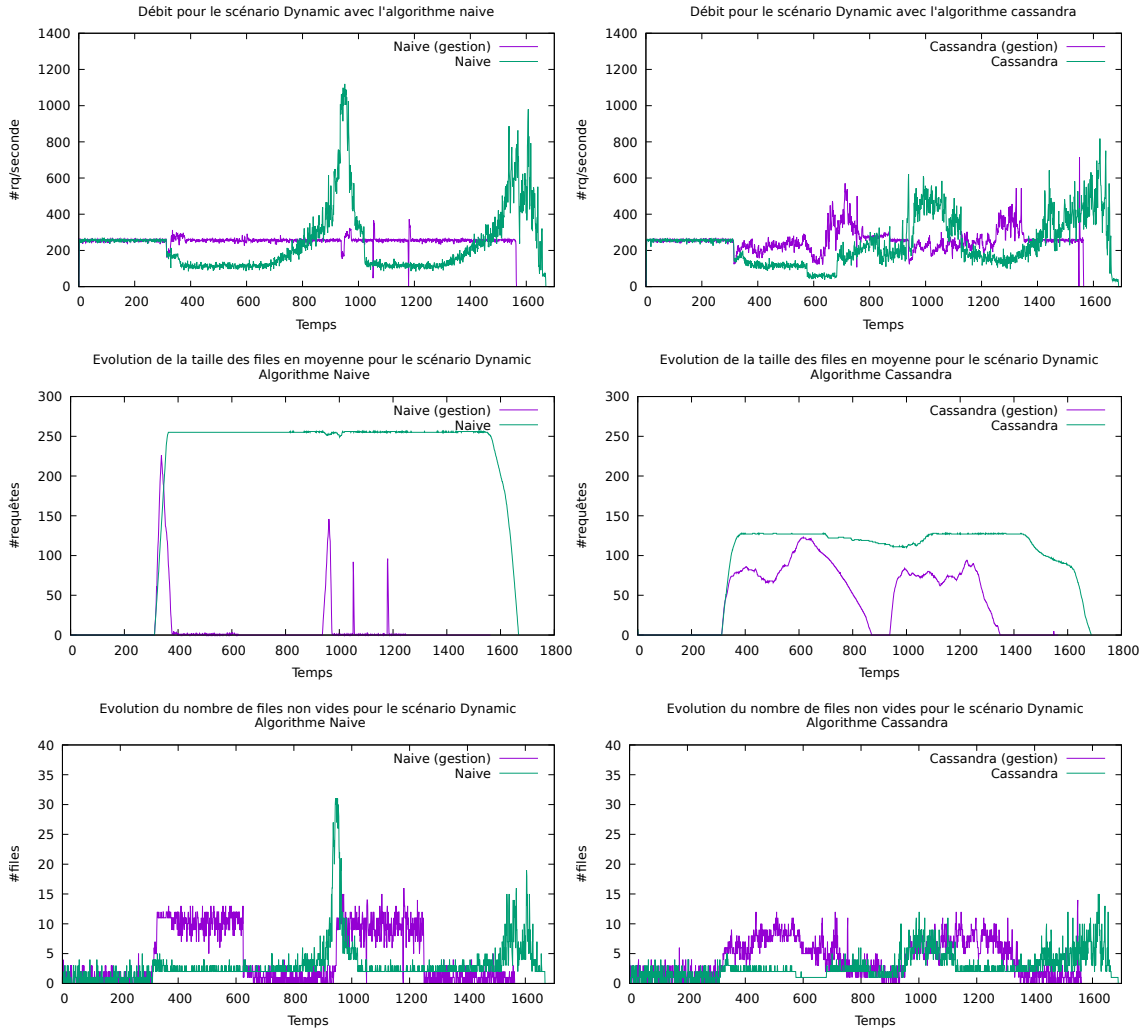


FIGURE B.1 – Évolution du débit et des files pour une fréquence d'injection de 256 requêtes/seconde.

Bibliographie

[AAR95] Oswin Aichholzer, Franz Aurenhammer, and Günter Rote. *Optimal graph orientation with storage applications*. Universität Graz/Technische Universität Graz. SFB F003-Optimierung und Kontrolle, 1995.

[ABK94] Yossi Azar, Andrei Z. Broder, and Anna R. Karlin. On-line load balancing. *Theor. Comput. Sci.*, 130(1) :73–84, 1994.

[ABKU94] Yossi Azar, Andrei Z. Broder, Anna R. Karlin, and Eli Upfal. Balanced allocations (extended abstract). In Frank Thomson Leighton and Michael T. Goodrich, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 593–602. ACM, 1994.

[ABS98] Micah Adler, Petra Berenbrink, and Klaus Schröder. Analyzing an infinite parallel job allocation process. In Gianfranco Bilardi, Giuseppe F. Italiano, Andrea Pietracaprina, and Geppino Pucci, editors, *Algorithms - ESA '98, 6th Annual European Symposium, Venice, Italy, August 24-26, 1998, Proceedings*, volume 1461 of *Lecture Notes in Computer Science*, pages 417–428. Springer, 1998.

[AF76] A. Gyarfas A. Frank. How to orient the edges of a graph ? In *Combinatorics, 18Colloq. Math. Soc. János Bolyai*, pages 352–354, 1976.

[AJM+11] Yuichi Asahiro, Jesper Jansson, Eiji Miyano, Hirotaka Ono, and Kouhei Zenmyo. Approximation algorithms for the graph orientation minimizing the maximum weighted outdegree. *J. Comb. Optim.*, 22(1) :78–96, 2011.

[AKP92] Baruch Awerbuch, Shay Kutten, and David Peleg. Competitive distributed job scheduling (extended abstract). In S. Rao Kosaraju, Mike Fellows, Avi Wigderson, and John A. Ellis, editors, *Proceedings of the 24th Annual ACM Symposium on Theory of Computing, May 4-6, 1992, Victoria, British Columbia, Canada*, pages 571–580. ACM, 1992.

[AMOZ06] Yuichi Asahiro, Eiji Miyano, Hirotaka Ono, and Kouhei Zenmyo. Graph orientation algorithms to minimize the maximum outdegree. In *Proceedings of the 12th Computing : The Australasian Theory Symposium-Volume 51*, pages 11–20. Australian Computer Society, Inc., 2006.

[ANR95] Yossi Azar, Joseph Naor, and Raphael Rom. The competitiveness of on-line assignments. *J. Algorithms*, 18(2) :221–237, 1995.

[BCFV00] Petra Berenbrink, Artur Czumaj, Tom Friedetzky, and Nikita D. Vvedenskaya. Infinite parallel job allocation (extended abstract). In *SPAA*, pages 99–108, 2000.

[BCSV00] Petra Berenbrink, Artur Czumaj, Angelika Steger, and Berthold Vöcking. Balanced allocations : the heavily loaded case. In F. Frances Yao and Eugene M. Luks, editors, *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 745–754. ACM, 2000.

[BE10] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed MIS algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6) :363–379, 2010.

[BE13] Leonid Barenboim and Michael Elkin. *Distributed Graph Coloring : Fundamentals and Recent Developments*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2013.

[BF99] Gerth Stølting Brodal and Rolf Fagerberg. Dynamic representation of sparse graphs. In Frank K. H. A. Dehne, Arvind Gupta, Jörg-Rüdiger Sack, and Roberto Tamassia, editors, *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, volume 1663 of *Lecture Notes in Computer Science*, pages 342–351. Springer, 1999.

[BKM05] Marcin Bienkowski, Miroslaw Korzeniowski, and Friedhelm Meyer auf der Heide. Dynamic load balancing in distributed hash tables. In Miguel Castro and Robbert van Renesse, editors, *Peer-to-Peer Systems IV, 4th International Workshop, IPTPS 2005, Ithaca, NY, USA, February 24-25, 2005, Revised Selected Papers*, volume 3640 of *Lecture Notes in Computer Science*, pages 217–225. Springer, 2005.

[BLT12] Gerth Stølting Brodal, George Lagogiannis, and Robert Endre Tarjan. Strict fibonacci heaps. In Howard J. Karloff and Toniann Pitassi, editors, *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 1177–1184. ACM, 2012.

[Cai06] Julie Anne Cain. *Random graph processes and optimisation*. PhD thesis, Department of Mathematics and Statistics, University of Melbourne, 2006.

[Cas] Apache cassandra. `http://apache.cassandra.org`.

[CDG+08] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable : A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2), 2008.

[CE91] Marek Chrobak and David Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theor. Comput. Sci.*, 86(2) :243–266, 1991.

[Cha00] Bernard Chazelle. The soft heap : an approximate priority queue with optimal error rate. *J. ACM*, 47(6) :1012–1027, 2000.

[CS97] Artur Czumaj and Volker Stemann. Randomized allocation processes. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 194–203. IEEE Computer Society, 1997.

[CSW07] Julie Anne Cain, Peter Sanders, and Nicholas C. Wormald. The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In Nikhil Bansal, Kirk Pruhs, and Clifford Stein, editors, *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, New Orleans, Louisiana, USA, January 7-9, 2007*, pages 469–476. SIAM, 2007.

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo : amazon's highly available key-value store. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 205–220. ACM, 2007.

[DLLX97] Xiaotie Deng, Hai-Ning Liu, Junsheng Long, and Bing Xiao. Competitive analysis of network load balancing. *J. Parallel Distrib. Comput.*, 40(2) :162–172, 1997.

[ER59] Paul Erdös and Alfréd Rényi. On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6 :290–297, 1959.

[FKP16] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs†. *Combinatorics, Probability & Computing*, 25(6) :870–908, 2016.

[FLAK11] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect : provable load balancing for randomly partitioned cluster services. In Jeffrey S. Chase and Amr El Abbadi, editors, *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*, page 23. ACM, 2011.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2) :374–382, 1985.

[FT14] Martin Farach-Colton and Meng-Tsung Tsai. Computing the degeneracy of large graphs. In Alberto Pardo and Alfredo Viola, editors, *LATIN*

2014 : Theoretical Informatics - 11th Latin American Symposium, Montevideo, Uruguay, March 31 - April 4, 2014. Proceedings, volume 8392 of *Lecture Notes in Computer Science*, pages 250–260. Springer, 2014.

[GHL16] Noël Gillet, Nicolas Hanusse, and Frédérique Lalanne. Impact de la réplication sur l'équilibrage de charge dans les bases de données distribuées. In *Actes de la conférence BDA 2016, Poitiers, France*, 2016.

[GLS+04] Brighten Godfrey, Karthik Lakshminarayanan, Sonesh Surana, Richard M. Karp, and Ion Stoica. Load balancing in dynamic structured P2P systems. In *Proceedings IEEE INFOCOM 2004, The 23rd Annual Joint Conference of the IEEE Computer and Communications Societies, Hong Kong, China, March 7-11, 2004*. IEEE, 2004.

[Gon81] Gaston H Gonnet. Expected length of the longest probe sequence in hash code searching. *Journal of the ACM (JACM)*, 28(2) :289–304, 1981.

[Gra66] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9) :1563–1581, 1966.

[GSBK04] Vijay Gopalakrishnan, Bujor Silaghi, Bobby Bhattacharjee, and Pete Keleher. Adaptive replication in peer-to-peer systems. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 360–369. IEEE, 2004.

[GW92] Harold N. Gabow and Herbert H. Westermann. Forests, frames, and games : Algorithms for matroid sums and applications. *Algorithmica*, 7(5&6) :465–497, 1992.

[GW10] Pu Gao and Nicholas C. Wormald. Load balancing and orientability thresholds for random hypergraphs. In Leonard J. Schulman, editor, *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pages 97–104. ACM, 2010.

[GW15] Pu Gao and Nicholas C. Wormald. Orientability thresholds for random hypergraphs. *Combinatorics, Probability & Computing*, 24(5) :774–824, 2015.

[JBA15] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. A survey of distributed data aggregation algorithms. *IEEE Communications Surveys and Tutorials*, 17(1) :381–404, 2015.

[JMB05] Márk Jelasity, Alberto Montresor, and Özalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3) :219–252, 2005.

[KKPS14] Tsvi Kopelowitz, Robert Krauthgamer, Ely Porat, and Shay Solomon. Orienting fully dynamic graphs with worst-case time bounds. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International*

Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II, volume 8573 of *Lecture Notes in Computer Science*, pages 532–543. Springer, 2014.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent hashing and random trees : Distributed caching protocols for relieving hot spots on the world wide web. In Frank Thomson Leighton and Peter W. Shor, editors, *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 654–663. ACM, 1997.

[KM05] Krishnaram Kenthapadi and Gurmeet Singh Manku. Decentralized algorithms using both local and random probes for P2P load balancing. In Phillip B. Gibbons and Paul G. Spirakis, editors, *SPAA 2005 : Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures, July 18-20, 2005, Las Vegas, Nevada, USA*, pages 135–144. ACM, 2005.

[Kow06] Lukasz Kowalik. Approximation scheme for lowest outdegree orientation and graph density measures. In *ISAAC*, volume 4288 of *Lecture Notes in Computer Science*, pages 557–566. Springer, 2006.

[KR06] David R. Karger and Matthias Ruhl. Simple efficient load-balancing algorithms for peer-to-peer systems. *Theory Comput. Syst.*, 39(6) :787–804, 2006.

[Lel12] Marc Lelarge. A new approach to the orientation of random hypergraphs. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 251–264. SIAM, 2012.

[LM10] Avinash Lakshman and Prashant Malik. Cassandra : a decentralized structured storage system. *Operating Systems Review*, 44(2) :35–40, 2010.

[Lu01] Linyuan Lu. The diameter of random massive graphs. In S. Rao Kosaraju, editor, *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 912–921. ACM/SIAM, 2001.

[LWW03] Pangfeng Liu, Da-Wei Wang, and Jan-Jan Wu. Efficient parallel I/O scheduling in the presence of data duplication. In *32nd International Conference on Parallel Processing (ICPP 2003), 6-9 October 2003, Kaohsiung, Taiwan*, pages 231–238. IEEE Computer Society, 2003.

[LZ12] Enrico Lovisari and Sandro Zampieri. Performance metrics in the average consensus problem : A tutorial. *Annual Reviews in Control*, 36(1) :26–41, 2012.

[MAE05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In Thomas Eiter and Leonid Libkin, editors, *Database Theory - ICDT 2005, 10th International Conference, Edinburgh, UK, January 5-7, 2005, Proceedings*, volume 3363 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 2005.

[Mak97] SAM Makki. A distributed algorithm for constructing an eulerian tour. In *Performance, Computing, and Communications Conference, 1997. IPCCC 1997., IEEE International*, pages 94–100. IEEE, 1997.

[Mit96] Michael David Mitzenmacher. *The Power of Two Choices in Randomized Load Balancing*. PhD thesis, University of California at Berkeley, 1996.

[MPM11] Alberto Montresor, Francesco De Pellegrini, and Daniele Miorandi. Distributed k-core decomposition. In Cyril Gavoille and Pierre Fraigniaud, editors, *Proceedings of the 30th Annual ACM Symposium on Principles of Distributed Computing, PODC 2011, San Jose, CA, USA, June 6-8, 2011*, pages 207–208. ACM, 2011.

[NW64] C. St. J. A. Nash-Williams. Decomposition of finite graphs into forests. *Journal of the London Mathematical Society*, 1964.

[Pel00] D. Peleg. *Distributed Computing : A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

[PNV13] Nikolaos A. Pantazis, Stefanos A. Nikolidakis, and Dimitrios D. Vergados. Energy-efficient routing protocols in wireless sensor networks : A survey. *IEEE Communications Surveys and Tutorials*, 15(2) :551–591, 2013.

[PPS16] Merav Parter, David Peleg, and Shay Solomon. Local-on-average distributed tasks. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 220–239. SIAM, 2016.

[PQ82] Jean-Claude Picard and Maurice Queyranne. A network flow solution to some nonlinear 0-1 programming problems, with applications to graph theory. *Networks*, 12(2) :141–159, 1982.

[PSW96] Boris Pittel, Joel Spencer, and Nicholas C. Wormald. Sudden emergence of a giant k-core in a random graph. *J. Comb. Theory, Ser. B*, 67(1) :111–151, 1996.

[RS98] Martin Raab and Angelika Steger. "balls into bins" - A simple and tight analysis. In Michael Luby, José D. P. Rolim, and Maria J. Serna, editors, *Randomization and Approximation Techniques in Computer Science, Second International Workshop, RANDOM'98, Barcelona, Spain, October 8-10, 1998, Proceedings*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 1998.

[San03] Peter Sanders. Asynchronous scheduling of redundant disk arrays. *IEEE Trans. Computers*, 52(9) :1170–1184, 2003.

[SEK03] Peter Sanders, Sebastian Egner, and Jan H. M. Korst. Fast concurrent access to parallel disks. *Algorithmica*, 35(1) :21–55, 2003.

[She10] Haiying Shen. An efficient and adaptive decentralized file replication algorithm in p2p file sharing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 21(6) :827–840, 2010.

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

[Tan14] Ganggui Tang. Fully dynamic graph orientation. Master's thesis, Dalhousie University, Halifax, Canada, 2014.

[Tel94] Gerard Tel. Network orientation. *Int. J. Found. Comput. Sci.*, 5(1) :23–57, 1994.

[TW14] Kunal Talwar and Udi Wieder. Balanced allocations : A simple proof for the heavily loaded case. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 979–990. Springer, 2014.

[Ven04] Venkat Venkateswaran. Minimizing maximum indegree. *Discrete Applied Mathematics*, 143(1-3) :374–378, 2004.

[Vöc99] Berthold Vöcking. How asymmetry helps load balancing. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 131–141. IEEE Computer Society, 1999.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4) :309–315, 1978.

[WFL98] Jie Wu, Eduardo B. Fernández, and Yingqui Lo. Embedding of binomial treas in hypercubes with link faults. *J. Parallel Distrib. Comput.*, 54(1) :49–74, 1998.

[WLK04] Shih-Chang Wang, Yuh-Rong Leu, and Sy-Yen Kuo. Distributed fault-tolerant embedding of several topologies in hypercubes. *J. Inf. Sci. Eng.*, 20(4) :707–732, 2004.

[XCCK09] Ye Xia, Shigang Chen, Chunglae Cho, and Vivekanand Korgaonkar. Algorithms and performance of load-balancing with multiple hash functions in massive content distribution. *Computer Networks*, 53(1) :110–125, 2009.

Liste des symboles

Symbole	Description
$F(u)$	File de requêtes en attente pour le noeud u
$\pi(q)$	temps d'exécution de la requête q
$w(q)$	poids de la requête q
$\ell(u)$	charge de u
ℓ_{avr}	charge moyenne
$\ddot{\ell}(u)$	nombre de requêtes en attente d'un noeud u
$\ddot{\ell}_{avr}$	nombre moyen de requêtes en attente
$\ell^*(u)$	charge mixte d'un nœud u
ℓ^*_{avr}	charge mixte moyenne
$\lambda(q)$	noeud d'arrivée de la requête q
$Q_{A_C}(u)$	ensemble des requêtes allouées à u avec l'algorithme d'allocation A_C
$T_{A_C}(Q)$	temps de complétion pour un algorithme d'allocation A_C
$T^*(Q)$	temps de complétion optimal pour l'ensemble de requêtes Q
n	nombre de noeuds
m	nombre de requêtes
$\bar{m}$	nombre d'objets
f	facteur de réplication
$\delta(G)$	densité d'un (hyper)graphe G
$\Delta(G)$	densité maximum d'un (hyper)graphe G
$d(u)$	degré d'un noeud u dans un graphe non orienté
$d^+(u)$	degré sortant d'un noeud u
$D^+(G)$	degré sortant maximum d'un (hyper)graphe G
$G[S]$	sous-graphe induit par un ensemble de nœuds S
$N(u)$	voisinage ouvert de u
$N[u]$	voisinage fermé de u
$arb(G)$	arboricité de G
$deg(G)$	dégénérescence de G