Résilience et auto-réparation de processus de décisions multi-agents

P. Rust[a] pierre.rust@orange.com
G. Picard[b] picard@emse.fr
F. Ramparany[a] fano.ramparany@orange.com

[a]Orange Labs, France
[b]Institut Henri Fayol, MINES Saint-Etienne, France

Résumé

Nous définissons la notion de k-résilience de graphes de calculs en support aux décisions d'agents opérées sur des systèmes dynamiques. Nous proposons une méthode d'auto-réparation de la distribution des calculs, DRPM[MGM-2], afin d'assurer la continuité des décisions collectives suite à la disparition d'agents, grâce au déploiement de réplicas de calculs. Nous nous intéressons ici à la réparation de processus d'optimisation sous contraintes, où les calculs sont des variables de décision ou des contraintes distribuées sur l'ensemble des agents. Nous évaluons expérimentalement les performances de DRPM[MGM-2] sur différentes topologies de systèmes opérant des algorithmes (Max-Sum ou A-DSA) pour résoudre des problèmes classiques (aléatoire, coloration de graphe, Ising) alors que des agents disparaissent.

Mots-clés : *DCOP, résilience, auto-réparation*

Abstract

We define the notion of k-resiience for computational graphs in support of agents' decisions on dynamic systems. We propose a method to self-repair the computation distribution, DRPM [MGM-2], as to ensure the continuity of collective decisions following the disappearance of agents, through the deployment of replicas calculations. We are interested here in constraint optimization process repair, where the computations are decision variables or distributed constraints. We experimentally evaluate the performance of DRPM[MGM-2] on different topologies of systems operating algorithms (Max-Sum or A-DSA) to solve classic problems (random, graph coloring, Ising) while agents disappear.

Keywords: *DCOP, résilience, auto-réparation*

1 Introduction

Nous examinons ici le problème de la répartition d'un ensemble de *calculs* étayant les décisions sur un ensemble d'agents incarnés dans des objets physiques (ou *nœuds*), comme des robots, des capteurs ou des véhicules autonomes.

Les décisions collectives et coordonnées sont organisées dans un graphe de calculs, où les sommets représentent des calculs et les arcs représentent une relation de dépendance entre les calculs. Une telle organisation est utilisée dans de nombreux modèles de décision, comme par exemple les graphes de facteurs et les graphes de contraintes utilisés lors de la résolution de problèmes d'optimisation distribuée sous contraintes (DCOP) [5], ou les algorithmes pour graphes de calculs tels que ceux adressés par Pregel ou d'autres frameworks basés sur BSP [8]. Bien que ces dernières structures ciblent généralement l'informatique en grappes hautes performances, nous considérons ici l'Internet des objets (IoT), des scénarios d'informatique embarquée ou des scénarios d'essaims de robots, où les calculs s'exécutent sur des nœuds distribués très hétérogènes et où une coordination centrale pourrait ne pas être souhaitable, voire même impossible [2].

Ici, les systèmes doivent pouvoir gérer les ajouts et les défaillances d'agents : lorsqu'un agent cesse de répondre, les autres agents du système doivent en assumer la responsabilité et exécuter les calculs partagés orphelins. De même, quand un nouvel agent est ajouté dans le système, il peut être utile de reconsidérer la distribution des calculs afin de tirer parti des capacités de calcul du nouveau venu, comme proposé dans [13]. Pour faire face à cette dynamique en maintenant les calculs et les décisions en cours alors que l'infrastructure évolue, une solution inspirée par les bases de données distribuées est la *réplication* [18, 17]. Nous proposons donc de répliquer les définitions de calculs plutôt que les données. Plus précisément, nous définissons la notion de *k-résilience*, qui caractérise les systèmes capables de fournir les mêmes fonctionnalités (ou prendre les mêmes décisions) même lorsque jusqu'à k noeuds disparaissent. L'adaptation de la distribution des calculs a été étudiée par [13], mais n'évalue aucune méthode distribuée en cours d'exécution. De plus, la disparition de plusieurs nœuds en même temps n'a pas non plus été prise en compte. [3] considère le problème distribution pour la résilience aux pannes, mais la méthode proposée s'appuie sur

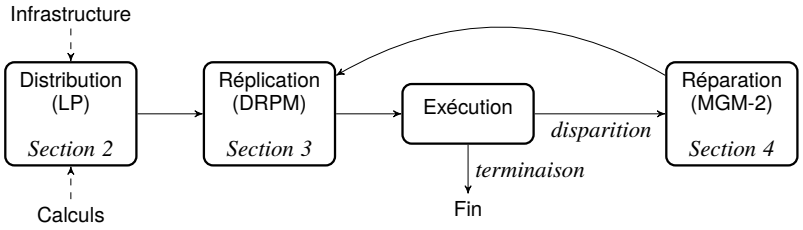


FIGURE 1 – Le cycle de vie de DRPM[MGM-2].

un agent *dispatcher* et reste donc partiellement centralisée. D'autre travaux, comme [6], ont étudié la réplication pour garantir tolérance aux pannes dans un SMA, mais répliquent en général des agents là ou nous proposons de répliquer les décisions, affectées aux agents.

L'article est structuré comme suit. La section 2 définit la notion de distribution optimale de graphes de calculs sur une infrastructure physique. La section 3 expose la notion de k-résilience et présente une méthode de recherche itérative distribuée, DRPM, pour déployer des réplicas afin d'assurer la k-résilience. Nous avons conçu une méthode de réparation distribuée, DRPM[MGM-2], basée sur un modèle DCOP utilisant la réplication pour adapter le déploiement de la décision à la suite de modifications apportées au système multi-agents physique (disparition des agents), dans la section 4. Nous nous intéresserons ensuite au cas particulier de la réparation des méthodes DCOP opérant sur des systèmes dynamiques, et proposons une méthode elle-même basée sur un DCOP pour réparer les solveurs DCOP. Le cadre général de notre approche est résumé dans la figure 1. Nous évaluons expérimentalement nos contributions algorithmiques sur différentes topologies de systèmes dont la fonctionnalité consiste à exécuter des processus de raisonnement sous contraintes distribuées pendant que les agents quittent le système, dans la section 5. Nous exécutons notamment Max-Sum et une version asynchrone de l'algorithme DSA (A-DSA) dans de telles configurations dynamiques. Enfin, nous concluons le document sur certaines perspectives.

2 Distribution des calculs

Le placement de calculs liés à des décisions sur des agents physiques peut voir un impact important sur les performances du système : certaines distributions peuvent améliorer le temps de réponse, d'autres favoriser la charge de communication entre les agents et d'autres peuvent améliorer d'autres critères tels que la qualité de service, les coûts d'exploitation, ou l'impact sur l'environnement.

Soit $G = \langle \mathbf{X}, D \rangle$ un graphe de calculs, où $\mathbf{X}$ est l'ensemble des calculs x_i, et D l'ensemble des arcs (i, j) représentant les dépendances entre calculs (qui impliquent l'envoi de messages). Soit $\mathbf{A}$ l'ensemble des agents pouvant héberger des calculs $x_i \in \mathbf{X}$. Notons $\mu : \mathbf{X} \mapsto \mathbf{A}$ la fonction associant les calculs aux agents et $\mu^{-1}(a_m)$ l'ensemble des calculs hébergés par a_m. Un agent ne peut héberger qu'une quantité limitée de calculs, contrainte par la *capacité* $\mathbf{w_{max}}(a_m)$ de l'agent, et le *poids* du calcul, $\mathbf{w}(x_i)$. Notons x_i^m le booléen spécifiant si x_i est hébergé par a_m.

Définition 1. *Etant donnés un ensemble d'agents* $\mathbf{A}$ *et un ensemble de calculs* $\mathbf{X}$*, une* ***distribution*** *est une fonction* $\mu : \mathbf{X} \mapsto \mathbf{A}$ *qui affecte chaque calcul à un agent exactement et qui satisfait les contraintes de capacité des agents.*

Trouver une distribution est un problème de satisfaction de contraintes avec :

$$\forall x_i \in \mathbf{X}, \quad \sum_{a_m \in \mathbf{A}} x_i^m = 1 \tag{1}$$

$$\forall a_m \in \mathbf{A}, \quad \sum_{x_i \in \mu^{-1}(a_m)} \mathbf{w}(x_i) \leq \mathbf{w_{max}}(a_m) \tag{2}$$

Lorsque les communications sont contraintes (e.g. IoT), la distribution devrait générer aussi peu de charge que possible et favoriser les liens les moins coûteux. Nous supposons que tous les agents peuvent communiquer entre eux, mais avec des coûts différents, représentés par une matrice : $\mathbf{route}(m,n)$ est le coût de communication entre a_m et a_n. Soit $\mathbf{msg}(i,j)$ la taille des messages entre x_i et x_j, le coût entre x_i sur a_m et x_j sur a_n est :

$$\forall x_i, x_j \in \mathbf{X}, \forall a_m, a_n \in \mathbf{A}, \mathbf{c_{com}}(i,j,m,n) = \begin{cases} \mathbf{msg}(i,j) \cdot \mathbf{route}(m,n) & \text{si } (i,j) \in D, m \neq n \\ 0 & \text{sinon} \end{cases} \tag{3}$$

où il n'y a aucun coût pour des calculs hébergés sur le même agent. Les coûts d'hébergement sur un agent sont modélisés par une fonction **host** affectant un coût pour chaque paire (a_m, x_j).

La qualité d'une distribution peut ainsi être évaluée par la fonction suivante, avec $\omega \in [0,1]$:

$$\omega \cdot \sum_{(i,j) \in D} \sum_{(m,n) \in \mathbf{A}^2} \mathbf{c_{com}}(i,j,m,n) \cdot x_i^m \cdot x_j^n \tag{4}$$

$$+ (1-\omega) \cdot \sum_{(x_i,a_m) \in \mathbf{X} \times \mathbf{A}} x_i^m \cdot \mathbf{c_{host}}(a_m, x_i) \quad (5)$$

Définition 2. *Une **distribution optimale** est une distribution μ qui minimise le coût des communications entre agents et le coût d'hébergement des calculs.*

Affecter les calculs aux agents de telle sorte peut se faire par la résolution d'un programme linéaire en linéarisant (4) et (5) comme des objectifs à minimiser, (2) comme des contraintes, et en ajoutant des contraintes afin que chaque calcul ne soit affecté qu'à un unique agent. Ce modèle est plus général que [13], dédié aux graphes de facteurs sans coût d'hébergement. Etant NP-difficile, il peut facilement mettre à mal les solveurs commerciaux modernes. Il peut cependant être utilisé pour initialiser le système pour calculer de manière centralisée la distribution initiale de petites instances. Dans nos expérimentations nous l'utilisons pour évaluer la qualité des distributions obtenues par nos techniques.

3 Résilience et réplication

Dans un contexte centralisé, lorsque certains agents échouent, on peut utiliser le programme linéaire discuté dans la section précédente pour recalculer une nouvelle distribution optimale. Cependant, accéder à un tel calcul centralisé peut ne pas être possible ou souhaitable, en fonction des exigences du scénario d'application. Ainsi, nous étudions des techniques décentralisées pour faire face à une telle dynamique dans l'infrastructure.

3.1 k-résilience

Nous définissons la notion de k-résilience comme la capacité d'un système à se réparer et à fonctionner correctement même lorsque jusqu'à k agents disparaissent. Cela signifie qu'après une période de récupération, tous les calculs doivent être actifs sur exactement un agent et communiquer les uns avec les autres comme spécifié par le graphe $\mathbf{G}$.

Définition 3. *Etant donnés les agents $\mathbf{A}$, les calculs $\mathbf{X}$, et une distribution μ, un système est k-**résilient** si pour tout $F \subset \mathbf{A}, |F| \leq k$, une nouvelle distribution $\mu' : X \rightarrow \mathbf{A} \setminus F$ existe.*

Une condition préalable à la k-résilience est de toujours avoir accès à la définition de chaque calcul après disparition. Une approche consiste à conserver k *réplicas* (copies de définitions) de chaque calcul actif sur différents agents. À condition que les k réplicas soient placés sur différents agents, il y aura toujours au moins un réplica restant après l'échec, comme en bases de données distribuées [8]. Ici, nous appliquons ces idées sauf que nous conservons des réplicas de définitions de calculs au lieu de données, ce qui implique que les calculs doivent être *stateless* ou que leur l'état doit être restaurable. La valeur maximale de k pour laquelle la k-résilience peut être atteinte dépend du système et surtout sur des capacités des agents. De plus, la k-résilience du système réparé devrait être restaurée, tant qu'il y a suffisamment de nœuds disponibles.

3.2 Placement des réplicas

Le problème de l'affectation de réplicas à des hôtes peut être considéré comme un problème d'optimisation, proche de la définition 2. Idéalement, nous devrions optimiser l'emplacement des réplicas pour les coûts de communication et d'hébergement. Cela garantirait que lorsque des agents échouent, des réplicas sont disponibles sur les bons agents candidats. Cependant, l'espace de recherche pour cette optimisation est extrêmement large. Dans un système k-résilient avec n agents, il y a $\sum_{0<i\leq k} \binom{n}{i}$ scénarios de défaillance potentiels (jusqu'à k agents sur n peuvent échouer simultanément). Avec m calculs, le nombre de *configurations de réplications* possibles est $m \cdot \binom{n}{k}$ Le problème de la distribution optimale des réplicas sur un ensemble d'agents ayant des coûts et des capacités différentes peut être transformé en un problème de sac à dos multiple quadratique (QMKP) [14], qui est NP-difficile.

Ensuite, pour chacune de ces configurations de réplicas, il existe m^k *configurations d'activations* (pour chaque calcul orphelin, activer l'un des k réplicas). En supposant que nous puissions calculer le coût de toutes ces configurations, il ne serait toujours pas évident de savoir quel placement de réplicas serait le meilleur : on pourrait envisager celui permettant la meilleure configuration d'activation, ou celui permettant, en moyenne, des configurations d'activation de bonne qualité ou même celle offrant les meilleures configurations d'activation sur l'ensemble des scénarios de défaillance possibles. Évidemment, la définition de l'optimalité pour le placement de réplicas dépend très fortement du problème. Par conséquent, étant donné cette complexité, nous optons pour une approche heuristique distribuée, décrite dans la section suivante.

3.3 Méthode distribuée de placement

Nous proposons ici une méthode distribuée, DRPM, pour déterminer les hôtes des k réplicas d'un calcul x_i donné. DRPM est une version distribuée de *iterative lengthening* (recherche de coût uniforme en fonction du coût des chemins) avec une mémorisation de chemins minimaux pour trouver les k meilleurs chemins. L'idée est d'hé-

berger des réplicas sur les voisins les plus proches en ce qui concerne les coûts de communication et d'hébergement et les contraintes de capacité, en effectuant une recherche dans un graphique induit par des dépendances de calcul. Il génère une distribution de k réplicas (et le coût des chemins d'accès vers leurs hôtes) avec des coûts minimum pour un ensemble d'agents interconnectés. S'il est impossible de placer les réplicas k, en raison de contraintes de mémoire, DRPM place autant de calculs que possible et génère le meilleur niveau de résilience possible. Un agent d'hébergement, appelé *initiateur*, demande *itérativement* à chacun de ses voisins les moins chers, par ordre de coût croissant, jusqu'à ce que tous les réplicas soient placés. Les hôtes candidats sont considérés de manière itérative par ordre croissant de coût, qui comprend à la fois le coût de la communication (tout au long du chemin entre le calcul initial et son réplica) et le coût d'hébergement de l'agent hébergeant le réplica.

Définissons tout d'abord le graphe modélisant les coût de communication qui sera développé lors de la recherche :

Définition 4. *Etant donné un graphe* $\langle \mathbf{X}, D \rangle$*, le* **route***–graphe est un graphe pondéré* $\langle \mathbf{A}, E, w \rangle$ *où* $\mathbf{A}$ *est l'ensemble des nœuds,* E *est l'ensemble des arcs avec* $E = \{(a_m, a_n) | \exists (x_i, x_j) \in D, and\ \mu(x_i) = a_m, \mu(x_j) = a_n\}$ *et* $w : E \rightarrow \mathbb{R}$ *est la fonction de pondération* $w(a_m, a_n) = \mathbf{route}(m,n)$.

Afin de prendre aussi en compte l'hébergement, étendons le **route**–graphe avec des nœuds supplémentaires attachés à chaque agent, excepté l'agent initiateur, par un arc pondéré par le coût d'hébergement, comme dans la figure 2.

Définition 5. *Étant donné le* **route***–graphe* $\langle \mathbf{A}, E, w \rangle$ *et un calcul* x_i*, le* **route+host***–graphe est un graphe pondéré* $\langle \mathbf{A}', E', \mathbf{cost} \rangle$ *où* $\mathbf{A}' = \mathbf{A} \cup \widetilde{\mathbf{A}}$ *est l'ensemble des nœuds,* $\widetilde{\mathbf{A}} = \{\tilde{a}_m | a_m \in \mathbf{A}, a_m \neq \mu(x_i)\}$ *est l'ensemble des nœuds supplémentaires (un pour chaque élément de* $\mathbf{A}$*),* $E' = E \cup \{(a_m, \tilde{a}_m) | \tilde{a}_m \in \mathbf{A}\}$ *est l'ensemble des arcs et* $\mathbf{cost} : E' \rightarrow \mathbb{R}$ *est la fonction de pondération t.q.* $\forall a_m, a_n \in \mathbf{A}$, $\mathbf{cost}(a_m, a_n) = w(a_m, a_n)$, $\forall \tilde{a}_m \in \widetilde{\mathbf{A}}$, $\mathbf{cost}(a_m, \tilde{a}_m) = \mathbf{c_{host}}(a_m, x_i)$.

Un **route+host**–graphe est un graphe de recherche, développé à l'exécution et exploré pour un calcul particulier x_i. Chaque agent exécute autant de DRPM que de calculs à répliquer sur plusieurs **route+host**–graphes. Pour un **route+host**–graphe donné, chaque agent peut encapsuler deux sommets (un dans $\mathbf{A}$ et son image dans $\widetilde{\mathbf{A}}$) et peut recevoir des messages concernant leurs deux sommets, et même des messages réflexifs. En outre,

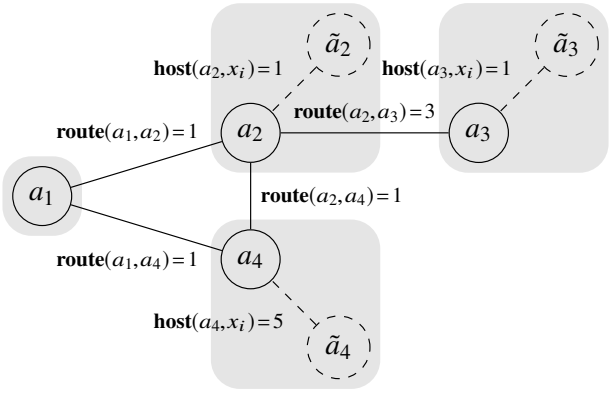


FIGURE 2 – Un **route+host**–graphe avec 4 agents

lors de l'évaluation de la possibilité pour un agent d'héberger un réplica pour x_i, nous nous assurons qu'il accepte uniquement s'il dispose d'une capacité suffisante pour activer tout sous-ensemble de taille k de ses réplicas. Bien sûr, cette contrainte est plus forte que ce qui pourrait être réellement nécessaire, ainsi, cette distribution n'est pas optimale en ce qui concerne les coûts d'hébergement, puisqu'un agent peut refuser d'héberger un calcul alors qu'il peut disposer de suffisamment de mémoire, au final. Comme de multiples instances de DPRM sont exécutées simultanément, une par calcul à répliquer, l'algorithme peut entraîner une distribution sous optimale, y compris en terme de communication. Cependant, si le placement de réplicas ne concerne qu'un seul calcul, la distribution est optimale puisque notre algorithme implémente une stratégie de recherche itérative par rallongement, ou *iterative lengthening*, avec mémorisation des chemins (pour éviter les boucles) [12, p.90].

Exemple 1. *La figure 2 représente un* **route+host***–graphe avec 4 agents (gris), où* a_1 *cherche à répliquer* x_i*. Pour* $k = 2$*, DRMP place des réplicas sur* a_2 *(coût de* $1+1=2$*) et sur* a_3 *(coût de* $1+3+1=5$*) s'il y a assez de capacité sur ces agents, comme le chemin minimal pour héberger sur* a_4 *est plus élevé (*$1+5=6$*).*

DRPM, étant une version distribuée de cet algorithme, utilise deux types de messages (`REQUEST` et `ANSWER`) avec les mêmes champs : (i) `current` : chemin de la requête, i.e. une liste contenant tous les sommets par lesquels les messages ont été transmis depuis l'initiateur jusqu'à celui qui reçoit le message actuel, (ii) `budget`, `spent` : budget restant pour l'exploration du graphe et budget déjà dépensé sur le chemin courant, (iii) `known` : tableau associatif affectant le coût aux chemins déjà découverts à des sommets non visités, qui comptabilise les chemins les moins chers jusqu'à présent, (iv) `visited` : liste des sommets déjà visités, (v) `k` : le nombre restant de réplicas à héberger, (vi) x_i : calcul à répliquer. Les messages

REQUEST descendent le long du graphe depuis l'initiateur, et correspondent au développement du graphe. Les messages ANSWER remontent les solutions (s'il y en a) jusqu'à l'initiateur.

Au début, l'agent nécessitant une réplication de calcul initialise known avec les chemins de ses voisins directs dans le **route+host**–graphe et envoie un message REQUEST avec un budget égal au chemin le moins cher connu. Ensuite, les agents traitent les messages comme expliqué dans le paragraphe suivant. Le protocole se termine lorsque toutes les réplicas possibles ont été placés (au plus k).

A la réception d'un message REQUEST, soit l'agent peut héberger un réplica et diminue ainsi le nombre de réplicas à placer, soit il transmet la demande à d'autres agents voisins. Dans le premier cas, si tous les réplicas ont été placés, l'agent répond à son prédécesseur avec un message ANSWER. Pour rechercher d'autres agents pour héberger des réplicas, s'il existe un chemin de coût minimum connu commençant par le chemin actuellement exploré qui est accessible avec le budget actuel, l'agent transmet la demande à son successeur dans ce chemin (avec un coût et un budget mis à jour). S'il n'y a pas un tel chemin, l'agent remplit le tableau known avec de nouveaux chemins menant à ses voisins dans le **route+host**–graphe, si ces derniers améliorent les chemins connus existants, et renvoie known avec un message ANSWER à son prédécesseur pour qu'il explore ces nouvelles possibilités.

A la réception d'un message ANSWER, le message peut soit notifier que tous les réplicas ont été placés, soit qu'il reste au moins un réplica à placer. Dans le premier cas, si l'agent est l'initiateur, il termine l'algorithme en plaçant tous les réplicas demandés, sinon il renvoie la réponse à son prédécesseur, jusqu'à ce qu'il atteigne l'initiateur. Dans ce dernier cas, si l'agent est l'initiateur, il augmente le budget et envoie une demande au voisin le plus proche le cas échéant ; sinon cela signifie qu'il n'y a plus de chemin à explorer et que tous les réplicas ne peuvent être placés : l'agent termine l'algorithme. Si l'agent n'est pas l'initiateur, mais qu'il existe un chemin accessible dans le budget actuel, il demande la réplication à son successeur dans le meilleur chemin connu, comme lors de la gestion des messages REQUEST. Enfin, s'il n'existe pas de tel chemin, il transmet simplement la réponse à son prédécesseur dans le chemin actuel.

Exemple 2. *Dans le cas de la figure 2, voici la séquence de messages qui sera générée. a_1, l'initiateur, peut envisager deux chemins $[a_1 \to a_2]$ et $[a_1 \to a_4]$ de coût identique, et initialise donc un budget de 1. a_1 envoie un message REQUEST à a_2, son premier successeur dans le premier chemin. a_2 répond avec un message ANSWER contenant deux nouveaux chemins $[a_1 \to a_2 \to \tilde{a}_2]$ de coût 2, et $[a_1 \to a_2 \to a_3]$ de coût 4. Le chemin $[a_1 \to a_2 \to a_4]$ de coût 2 n'est pas considéré car le meilleur chemin connu dans known vers a_4 est de coût 1. a_1, à la réception du message a_2 va explorer le meilleur chemin en envoyant un message REQUEST à a_4. a_4 rajoute un chemin à known, $[a_1 \to a_4 \to \tilde{a}_4]$ de coût 6 et répond à a_1. a_1 va ainsi envoyer un message REQUEST à a_2 pour explorer le meilleur chemin. a_2 peut héberger le réplica, car ce chemin même à une feuille $\tilde{a}_2$. a_2 peut même continuer l'exploration, car le prochain chemin dans known passe par lui et mène à a_3. a_2 envoie donc un message REQUEST à a_3, mais cette fois le nombre de réplicas à placer n'est plus de 2 mais de 1. a_3 peut rajouter le chemin $a_1 \to a_2 \to a_3 \to \tilde{a}_3$, qui est le meilleur actuel, et donc répondre qu'il peut héberger le dernier réplica. a_3 envoie donc un message ANSWER à a_2, qui le transmet à a_1. A la réception, comme tous les réplicas ont été placés, a_1 termine le placement.*

Globalement, chaque agent est chargé de placer k réplicas de tous les calculs actifs qu'il héberge actuellement et exécute donc DRPM une fois pour chacun de ses calculs actifs. Ces multiples exécutions de DRPM peuvent être faites de manière séquentielle ou simultanée mais leur résultat dépend de l'ordre de réception des messages. Notez cependant que même si vous exécutez plusieurs DRPM simultanément, un agent ne dispose que d'une seule file de messages et traite les messages entrants de manière séquentielle, ce qui l'empêche d'accepter des réplicas qui dépasseraient sa capacité.

Théorème 1. *DRPM se termine.*

Démonstration. Pour k = 1, étant donné que les coûts du DRPM sont additifs et monotones et qu'il enregistre les chemins d'accès vers les sommets non consultés, il se termine comme un algorithme *iterative lengthening* classique, avec le chemin de coût minimum ou le chemin vide s'il n'y a pas assez de mémoire dans les agents pour héberger le calcul x_i. Pour $k > 1$, DRPM tente de placer chaque réplica de manière séquentielle. Il recherche d'abord le meilleur chemin (comme pour $k = 1$), puis exécute le même processus pour un deuxième meilleur chemin, et ainsi de suite jusqu'à ce que (i) les k réplicas soient placés ou (ii) il n'y ait pas assez de mémoire pour héberger le $n^{\text{ième}}$ réplica . La mémorisation dans visited garantit que le même chemin ne sera pas pris en compte deux fois. Ainsi, des itérations de recherche consécutives génèrent des chemins différents avec des coûts de chemin augmentant. Ainsi, dans le cas (i), DRPM se termine lorsque k réplicas ont été placés sur les k meilleurs hôtes ; et

dans le cas (ii), il se termine lorsque $k' < k$ réplicas ont été placés, où k' est le nombre maximal de réplicas pouvant être placés. □

4 Méthode d'auto-réparation

Une fois la réplication abordée, nous modélisons maintenant le problème de réparation en tant que problème d'optimisation distribuée sous contraintes (DCOP) [7], à résoudre par les agents eux-mêmes, à la suite de d'apparitions ou disparitions dans le système, afin d'activer des réplicas ou d'améliorer la qualité de la distribution des calculs.

4.1 Formulation DCOP

On note X_c l'ensemble des calculs candidats x_i qui peuvent ou doivent être déplacés lorsque l'ensemble des agents change. Pour chacun de ces calculs, notons A_c^i l'ensemble des agents candidats pour accueillir x_i. L'ensemble de tous les agents candidats, indépendamment des calculs, est noté $A_c = \cup_{x_i \in X_c} A_c^i$ et X_c^m désigne l'ensemble des calculs que l'agent a_m pourrait héberger. Décider quel agent $a_m \in A_c$ héberge chaque calcul $x_i \in X_c$ peut être traduit en un problème d'optimisation similaire à celui présenté dans la section 2, limitée à A_c et X_c. contrainte Pour s'assurer que chaque calcul candidat est hébergé sur exactement un agent, nous réécrivons les contraintes (1) pour chaque $x_i \in X_c$:

$$\sum_{a_m \in A_c^i} x_i^m = 1 \tag{6}$$

De même, les contraintes de capacité (2) peuvent être reformulées comme suit :

$$\sum_{x_i \in X_c^m} \mathbf{w}(x_i) \cdot x_i^m + \sum_{x_j \in \mu^{-1}(a_m) \setminus X_c} \mathbf{w}(x_j) \leq \mathbf{w_{max}}(a_m) \tag{7}$$

L'objectif de coût d'hébergement dans (5) peut être formulé de manière similaire à l'aide d'une contrainte souple pour chaque agent candidat a_m :

$$\sum_{x_i \in X_c^m} \mathbf{c_{host}}(a_m, x_i) \cdot x_i^m \tag{8}$$

Enfin, les coûts de communication dans (4) sont représentés par un ensemble de contraintes souples. Pour un agent a_m, les frais de communication liés à l'hébergement d'un calcul x_i peut être formulé comme la somme des coût des arcs coupés (x_i, x_j) à partir du graphe de calculs $\langle \mathbf{X}, D \rangle$, (c'est-à-dire où $\mu^{-1}(x_j) \neq a_m$). Notons N_i les voisins de x_i dans le graphe de calcul. Quand un voisin x_n n'est pas un calcul candidat (c'est-à-dire qu'il ne sera peut-être pas déplacé et que $x_m \in N_i \setminus X_c$), le coût de communication de l'arc correspondant est simplement donné par $\mathbf{c_{com}}(i, j, m, \mu^{-1}(x_m))$. Pour les voisins qui pourraient être déplacés, le coût de la communication dépend de l'agent candidat choisi pour l'héberger, $\sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{c_{com}}(i, j, m, n)$. Avec cela, nous pouvons écrire la contrainte souple de coût de communication pour l'agent a_m :

$$\sum_{(x_i, x_j) \in X_c^m \times N_i \setminus X_c} x_i^m \cdot \mathbf{c_{com}}(i, j, m, \mu^{-1}(x_j)) + \sum_{(x_i, x_j) \in X_c^m \times N_i \cap X_c} x_i^m \cdot \sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{c_{com}}(i, j, m, n)) \tag{9}$$

Nous pouvons maintenant formuler le problème de réparation en tant que DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ où $\mathcal{A}$ est l'ensemble des agents candidats A_c, $\mathcal{X}$ et $\mathcal{D}$ sont respectivement l'ensemble des variables de décision x_i^m et leur domaine $\{0,1\}$ et $\mathcal{C}$ est composé de contraintes (6), (7), (8) et (9) appliquées pour chaque agent $a_m \in A_c$. (6) et (7) entraînent des coûts infinis en cas de violation, alors que (8) et (9) définissent directement les coûts à minimiser. La fonction μ affecte chaque variable x_i^m à un agent a_m.

4.2 Réparer avec MGM-2

Maintenant que notre problème de réparation a été exprimé en tant que DCOP, nous discutons de sa résolution en utilisant un algorithme DCOP. Plusieurs méthodes existent, telles que les algorithmes de recherche [7, 9] et les algorithmes d'inférence [10, 16, 5], pour en citer quelques-uns. En bref, en utilisant ces protocoles d'envoi de messages (synchrones ou non), les agents se coordonnent pour attribuer des valeurs à leurs variables.

Dans notre cas, nous optons pour une méthode légère, rapide et itérative, à savoir MGM [7]. Chaque agent assigne d'abord des valeurs aléatoires à ses variables et envoie les informations à tous ses voisins. En utilisant toutes les valeurs des voisins, un agent calcule le gain maximal s'il modifie sa valeur et l'envoie à tous ses voisins. Ensuite, en utilisant les gains de tous les voisins, l'agent modifie sa valeur si son gain est le plus grand. Ce processus se répète jusqu'à ce qu'une condition de terminaison soit remplie. Dans notre cas, les décisions nécessitent une coordination entre deux agents : pour déplacer un calcul de l'agent a_m à a_p, la variable binaire x_i^m doit prendre la valeur 0, tandis que *simultanément*, x_i^p doit passer de 0 à 1. Ce besoin de modifications simultanées justifie l'utilisation de MGM-2 [7]. La propriété de monotonie de MGM-2 répond très bien à nos

besoins : une fois les contraintes dures (6) et (7) satisfaites, elles ne seront plus violées, tout en optimisant les contraintes souples (8) et (9).

En combinaison avec notre méthode de placement de réplicas, nous obtenons une méthode de réparation, appelée DRPM[MGM-2], qui peut être utilisée pour s'adapter au départ et à l'arrivée d'agents, en utilisant les définitions appropriées des ensembles A_c et X_c. Discutons le cas de départ d'agent, car c'est la situation la plus stressante. En supposant que le déploiement initial et le placement des réplicas aient été effectués au moment du démarrage du système, celui-ci exécutera le cycle de réparation suivant tout au long de son cycle de vie (voir la figure 1) : (a) Détecter départ / arrivée ; (b) Activer les réplicas des calculs manquants (avec MGM-2) ; (c) Placer les nouveaux réplicas pour les calculs manquants (à l'aide de DRPM) et poursuivre le fonctionnement nominal.

L'étape (a) suppose des mécanismes de découverte et de *keep alive* qui informent automatiquement certains agents de tout événement dans l'infrastructure. Ainsi, lorsqu'un agent a_m échoue ou est supprimé, nous considérons que tous les agents voisins de a_m dans le **route**–graphe sont informés du départ. L'étape (b) déplace les calculs hébergés par les agents disparus A_d à d'autres agents. Les calculs candidats sont les calculs orphelins hébergés sur ces agents : $X_c = \cup_{a_m \in A_d} \mu(a_m)$. Pour éviter tout délai supplémentaire et toute communication lors de la phase de réparation, ces calculs orphelins doivent être affectés à des agents disposant déjà des informations nécessaires pour exécuter le calcul. Cela signifie que l'agent candidat pour un calcul orphelin x_i affecte l'ensemble d'agents encore disponibles hébergeant une réplique pour ce calcul : $A_c^i = \rho(x_i) \backslash A_d$. Dans un système k-résilient, tant que $|A_d| \leq k$, nous sommes sûrs qu'il y aura toujours au moins un agent dans A_c^i. Ainsi, l'étape (b) donne une affectation de chacun des calculs orphelins à l'un des agents hébergeant son réplica. L'étape (c) maintient un bon niveau de résilience dans le système en réparant la distribution des réplicas en utilisant DRPM sur un problème plus petit, car de nombreux réplicas sont déjà placés.

5 Évaluation expérimentale

Analysons maintenant l'impact de la réparation sur les performances de processus de raisonnements sous contraintes – Max-Sum, un algorithme d'inférence approché [5] et A-DSA, une version asynchrone de l'algorithme de recherche locale DSA [19] – ainsi que la qualité des distributions obtenues après réparation.

5.1 Cadre expérimental

Nous exécutons les expériences en utilisant la plateforme multi-thread pyDCOP [1]. Pour évaluer notre cadre de réparation, nous générons des instances définies par trois composants : une définition de problème DCOP, une topologie multi-agents (l'infrastructure), et un scénario de perturbation.

Nous étudions 3 types de DCOPs : (i) coloration de graphes aléatoires avec densité $p = 0.3$, (ii) coloration de graphes *scale free* [1], et (iii) modèles d'Ising. Pour les colorations de graphes, chaque nœud est associé à une variable et chaque arc est associé à une contrainte souple dont le coût de chaque paire de valeurs possibles est tiré aléatoirement et uniformément dans $U[0-9]$. Le modèle d'Ising est une référence largement utilisée en physique statistique ; nous utilisons ici les mêmes paramètres que [15]. Nous générons une grille toroïdale régulière et associons chaque nœud à une variable binaire x_i et chaque arc à une contrainte binaire dont la fonction de coût r_{ij} est déterminée en échantillonnant d'abord une valeur k_{ij} à partir d'une distribution uniforme $U[-1,6,1,6]$, et ensuite assignons $r_{ij}(x_i,x_j) = -k_{ij}$ si $x_i = x_j$, k_{ij} sinon. De plus, chaque variable a une contrainte unaire dont la fonction de coût r_i est déterminée en échantillonnant k_i à partir d'une distribution uniforme $U[-0,05,0,05]$ puis en affectant $r_i(0) = k_i$ et $r_i(1) = -k_i$.

Selon le processus choisi (Max-Sum ou A-DSA), le DCOP est encodé dans un graphe de facteur (FG, pour Max-Sum) ou un graphe de contraintes (CG, pour A-DSA). Pour un même problème, $\langle \mathbf{A}, \mathcal{X}, \mathcal{D}, C, \mu \rangle$, il faut placer soit $|\mathbf{V}| = |\mathcal{X}| + |C|$ calculs pour un FG ou $|\mathbf{V}| = |\mathcal{X}|$ calculs pour un CG.

Une fois le graphe de calcul généré, nous générons deux infrastructures multi-agents différentes : uniforme et dépendante du problème. Une infrastructure est composée d'agents $|\mathbf{A}|$, chacun contenant une variable de décision ($|\mathbf{A}| = |\mathcal{X}|$), et est définie par $\mathbf{c_{host}}$, **route**, $\mathbf{w_{max}}$, $\mathbf{w}$ et **msg**. L'infrastructure uniforme considère les systèmes où les coûts de communication sont uniformes : $\forall a_m, a_n, \mathbf{route}(a_m, a_n) = 1$. Dans le cas dépendant du problème, les coûts $\mathbf{route}(a_m, a_n)$ sont définis de manière à respecter la structure du graphe de calcul : les agents ayant plusieurs voisins ont une faible communication tandis que les agents ayant peu de voisins ont un coût de communication plus élevé, comme dans de nombreuses infrastructures physiques (e.g. l'IoT). Plus précisément, $\mathbf{route}(a_m, a_n) = \frac{1+||N(a_m)|-|N(a_n)||}{|N(a_m)|+|N(a_n)|}$ où $|N(a_i)|$ est le nombre de voisins de a_i dans le graphe de calculs. Dans tous les cas, nous définissons

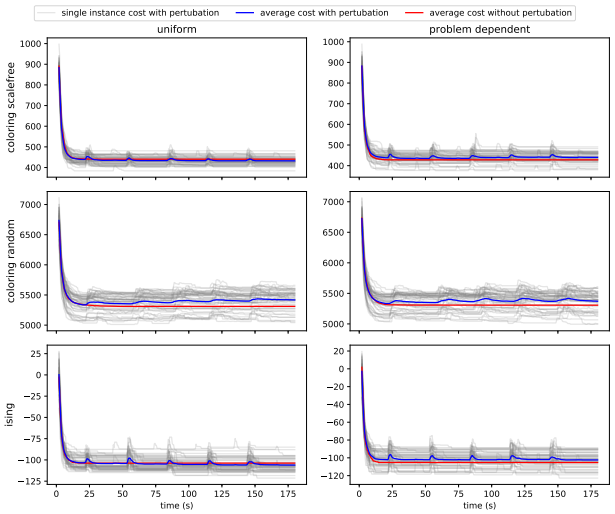
1. https://github.com/Orange-OpenSource/pyDcop

(i) $\mathbf{c_{host}}(a_m, x_j) = 0$ si le calcul x_j est initialement hébergé par l'agent a_m, $\mathbf{c_{host}}(a_m, x_j) = 10$ sinon ; (ii) la capacité de chaque agent dépend du poids de sa variable de décision et est mis à une grande valeur, pour que tous les réplicas puissent être hébergés et que la k-résilience soit possible, même après plusieurs réparations : $\mathbf{w_{max}}(a_i) = 100 * \mathbf{w}(x_i)$; (iii) finalement, $\mathbf{w}$ et $\mathbf{msg}$ dépendent du processus utilisé. Pour Max-Sum, la taille des messages est une fonction directe de la taille de domaine de la variable : $\mathbf{msg}(i, j) = |D_j| = 10$ et le poids des calculs de variables et de facteurs est respectivement proportionnel à la taille du domaine de la variable et la somme de la taille des domaines des variables liées. Pour A-DSA, $\mathbf{msg}(i, j) = 1$ et $\mathbf{w}(x_i) = |N(a_m)|$. Initialement, chaque variable de décision est affectée à un agent, et dans le cas de FG, les facteurs sont placés de manière optimale à l'aide du LP décrit dans la section 2. Nous utilisons $\omega = 0.5$ (les coûts d'hébergement et de communication sont également pris en compte).

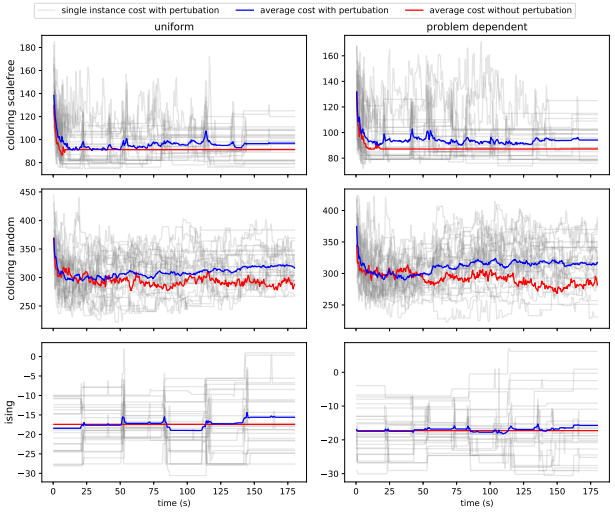
Nous générons des scénarios de perturbation sous forme de séquences d'événements toutes les 30 secondes, à partir de $t = 20s$. A chaque événement, k agents choisis au hasard disparaissent afin d'analyser l'impact sur les processus DCOP, et d'observer la k-résilience de notre système. Nous générons 20 instances (infrastructure et problème), et exécutons le scénario 5 fois pour chaque instance. De plus, nous résolvons les mêmes problèmes (5 exécutions pour chacune des 20 instances) sans aucune perturbation, afin d'évaluer l'impact de nos méthodes de réparation sur la qualité de la solution renvoyée par A-DSA ou Max-Sum. Dans les deux cas, les résultats sont moyennés sur toutes les instances. Les expériences sont effectuées sur un processeur Intel Core i7 avec 16 Go de RAM.

5.2 Impact de la réparation sur A-DSA

Regardons l'état d'exécution d'A-DSA actuel avec et sans perturbations, et analysons comment DRMP[MGM-2] modifie le fonctionnement du processus en cours d'exécution. Nous générons des problèmes avec $|\mathbf{A}| = |\mathcal{X}| = 100$ et utilisons $k = 3$. La figure 3a montre le coût de la solution trouvée par A-DSA au fil du temps. Le coût de chacune des 100 exécutions est affiché en gris transparent, et la forme globale illustre le fait que le comportement du système est cohérent dans les différentes instances. Nous pouvons voir que les solutions avec perturbations se dégradent lorsque les agents sont supprimés, mais rapidement améliorées à nouveau lorsque le système récupère. Ici, les réplicas activés par la réparation, par opposition aux calculs hébergés sur des agents supprimés, n'ont pas besoin des connaissances accumulées pour retrouver un état cohérent, grâce aux messages passés entre voisins, par nature dans A-DSA. En effet, les



(a) A-DSA



(b) Max-Sum

FIGURE 3 – Coût des solutions de Max-Sum (a) et A-DSA (b) en cours d'exécution, avec (bleu) et sans perturbations (rouge), sur des infrastructures uniformes (gauche) et dépendantes du problème (droite), et sur coloration *scale free* (haut), coloration aléatoire (milieu), et Ising (bas).

calculs sont ici `stateless`, comme l'exige notre approche de la k-résilience : ils collectent de nouvelles informations sur les coûts auprès de leurs voisins à chaque échange de messages.

La période de récupération est plus courte sur les modèles *scale free* et Ising. En effet, les problèmes de coloration des graphes sont plus denses et plus difficiles à résoudre, et leur réparation nécessite plus de messages et de temps. Pendant la même durée, les agents qui résolvent et réparent les processus de coloration des graphes doivent gérer davantage de messages spécifiques à la réparation (MGM-2 et DRPM). Ils gèrent donc moins de messages A-DSA pour améliorer le coût de la

solution. C'est la même raison pour laquelle le coût global est plus élevé dans une infrastructure dépendant du problème : les processus d'activation et de placement de réplicas nécessitent plus de messages. Nous avons évalué le temps moyen nécessaire pour réparer une distribution à moins de 3 secondes. Dans certains contextes, le coût moyen avec perturbation est inférieur au coût moyen sans perturbation. En fait, parfois, supprimer certains calculs et les transférer à d'autres agents, oubliant ainsi des informations sur le voisinage passé, peut extraire A-DSA de certains optima locaux. Nous pouvons également observer un décalage dans le temps pour la réparation du système. Cela est principalement dû à la latence de traitement des messages accumulés.

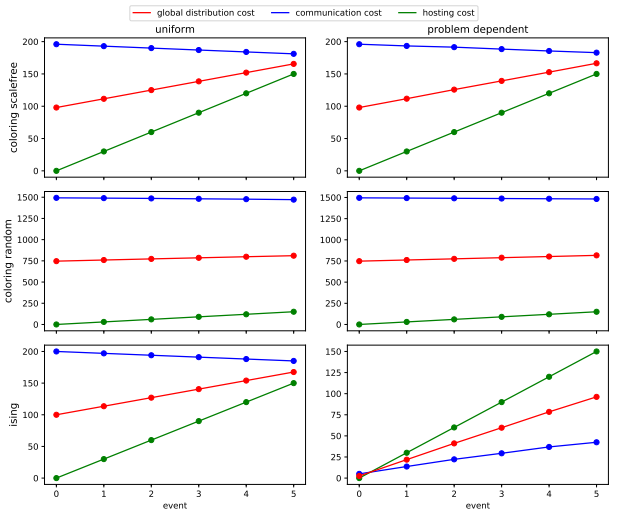
5.3 Impact de la réparation sur Max-Sum

Ici, nous considérons des problèmes plus petits, car Max-Sum fonctionne sur des graphes de facteurs nécessitant plus de calculs (un de plus par arête dans le graphe) à distribuer que les graphes de contraintes utilisés par A-DSA. Par conséquent, nous considérons $|\mathbf{A}| = |\mathcal{X}| = 25$, et $k = 2$. Néanmoins, sur un graphe aléatoire avec une densité de 0.3, de tels problèmes nécessitent en moyenne $25+0.3\frac{25\times24}{2} = 125$ calculs à gérer.

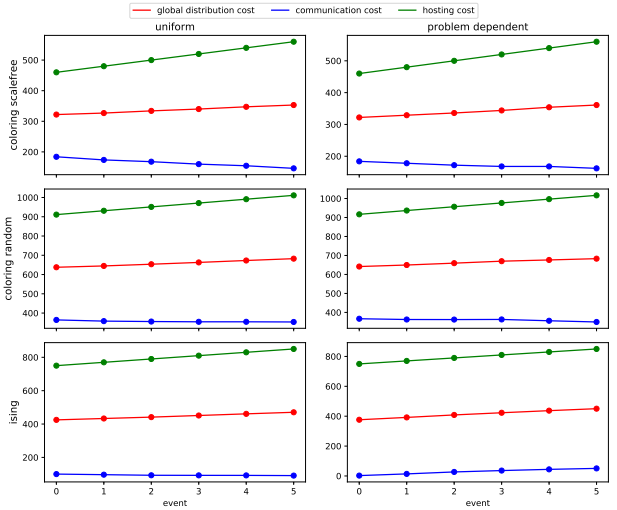
Dans la figure 3b nous pouvons voir que les solutions sur le système perturbé se dégradent lorsque les agents sont supprimés, mais s'améliorent à nouveau lorsque le système récupère, comme pour A-DSA. Cependant, le fonctionnement Max-Sum sur des problèmes très cycliques tels que la coloration aléatoire est connu pour être très bruitée, même en utilisant un facteur d'amortissement élevé (nous utilisons ici 0.8), comme proposé dans [4]. En outre, dans les algorithmes de propagation de croyances comme Max-Sum, les calculs ne sont pas vraiment sans état : ils accumulent des informations sur les contraintes et les préférences de leurs voisins. Lors de l'activation d'un réplica, le nouveau calcul actif recommence à nouveau et un nombre indéterminé de tours de messages sont nécessaires pour restaurer ces informations. Globalement, le fonctionnement Max-Sum est davantage impacté par la procédure de perturbations et de réparation qu'A-DSA.

5.4 Qualité des distributions réparées

Pour évaluer la qualité des distributions de calculs réparées, nous mesurons la dégradation de la distribution tout au long de la vie du système. A chaque événement, nous évaluons le coût de la distribution actuelle des graphes de contraintes (pour A-DSA) et des graphes de facteurs (pour Max-Sum) à l'aide des équations 4 et 5, par rapport au coût de distribution initial (qui est optimal,



(a) A-DSA



(b) Max-Sum

FIGURE 4 – Coût de la distribution des graphes de calculs pour A-DSA et Max-Sum, après chaque événement, sur des infrastructures uniformes (gauche) et dépendantes du problème (droite), et sur coloration *scale free* (haut), coloration aléatoire (milieu), et Ising (bas).

mais ne peut pas être calculé au moment de l'exécution). Les figures 4a et 4b montrent les coûts de distribution pour les 100 instances. Comme le coût de distribution global est constitué de coûts de communication et d'hébergement, nous traçons également ces deux coûts indépendamment. Dans tous les cas, le coût de l'hébergement augmente logiquement de $10 \cdot k$ à chaque événement de perturbation, les k calculs étant déplacés de leur agent initial à un autre (où le coût d'hébergement est de 10). Sur les modèles *scale free* et Ising, les coûts d'hébergement et de communication ont le même ordre de grandeur. Mais, pour les graphes aléatoires, une densité plus élevée implique qu'il y a plus d'arcs dans le graphique et, par conséquent,

le coût de communication global est plus élevé. En général, les coûts de communication diminuent à chaque réparation, sauf dans Ising avec une infrastructure dépendant du problème. Ici, tous les agents sont homogènes et les calculs sont nécessairement transférés vers des agents plus coûteux, en termes de communication (il n'existe aucun agent moins coûteux ni équivalent aux agents manquants) et les coûts de communication augmentent donc progressivement. Dans les autres cas, les calculs peuvent passer à un agent moins coûteux, ce qui entraîne une diminution des coûts de communication.

6 Conclusions

Nous avons étudié la distribution résiliente de calculs sur un ensemble d'agents dynamique, et deux algorithmes distribués ont été proposés : (i) un protocole de placement de réplicas (DRPM) et (ii) un protocole de réparation (DRPM[MGM-2]) basé sur les réplicas placés par DRPM et sur l'algorithme MGM-2. Nos contributions ont été évaluées expérimentalement en exécutant Max-Sum et A-DSA sur des systèmes dynamiques où les agents disparaissent lors du processus d'optimisation. Pour toutes les configurations étudiées, l'exécution de ces algorithmes a été peu impactée par notre méthode de réparation, et le système continue de fournir des solutions, ce qui démontre la résilience du système. Dans nos expérimentations, A-DSA s'est montré le moins impacté par la suppression d'agents et la réparation.

Nous nous sommes focalisés ici sur le pire scénario. Nous étudierons des scénarios moins critiques où d'autres agents peuvent (ré-)apparaître. Nous avons proposé d'utiliser MGM-2 comme algorithme au cœur de la réparation, mais d'autres algorithmes légers de résolution de DCOP pourrait être considérés, voire conçus spécifiquement. Nous visons également à appliquer nos contributions à un cadre plus large de graphes de calculs, comme la virtualisation de fonctions réseaux [11].

Références

[1] A. Barabási. Emergence of scaling in random networks. 286(5439) :509–512.

[2] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[3] Fadoua Chakchouk, Sylvain Piechowiak, René Mandiau, Julien Vion, Makram Soui, and Khaled Ghedira. Fault Tolerance in DisCSPs : Several Failures Case. In Fernando De La Prieta, Sigeru Omatu, and Antonio Fernández-Caballero, editors, *Distributed Computing and Artificial Intelligence, 15th International Conference*, volume 800, pages 204–212. Springer International Publishing.

[4] L. Cohen and R. Zivan. Max-sum revisited : The real power of damping. In Gita Sukthankar and Juan A. Rodriguez-Aguilar, editors, *Autonomous Agents and Multiagent Systems*, pages 111–124, Cham, 2017. Springer International Publishing.

[5] A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08)*, pages 639–646, 2008.

[6] Zahia Guessoum, Jean-Pierre Briot, and Nora Faci. Un mécanisme de réplication adaptative pour des sma tolérants aux pannes. In *JFSMA*, pages 135–148.

[7] R.T. Maheswaran, J.P. Pearce, and M. Tambe. Distributed algorithms for dcop : A graphical-game-based approach. In *Proc. of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS)*, pages 432–439, 2004.

[8] G. Malewicz, M. H. Austern, A. J.C Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel : A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146. ACM, 2010.

[9] P.J. Modi, W. Shen, M. Tambe, and M. Yokoo. ADOPT : Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence Journal*, 2005.

[10] A. Petcu and B. Faltings. A scalable method for multiagent constraint optimization. In *International Joint Conference on Artificial Intelligence (IJCAI'05)*, pages 266–271, 2005.

[11] Q. Pham, K. Singh, A. Bradai, G. Picard, and R. Riggio. Single and multi-domain adaptive allocation algorithms for vnf forwarding graph embedding. *IEEE Transactions on Network and Service Management*, pages 1–1, 2018.

[12] S. Russel and P. Norvig. *Artificial Intelligence : a Modern Approach*. Prentice-Hall, 3rd edition, 2009.

[13] P. Rust, G. Picard, and F. Ramparany. On the deployment of factor graph elements to operate max-sum in dynamic ambient environments. In *8th International Workshop on Optimisation in Multi-Agent Systems (OPTMAS 2017)*, 2017.

[14] T. Saraç and A. Sipahioglu. Generalized quadratic multiple knapsack problem and two solution approaches. *Computers & Operations Research*, 43(Supplement C) :78 – 89, 2014.

[15] M. Vinyals, M. Pujol, J. A. Rodriguez-Aguilar, and J. Cerquides. Divide-and-coordinate : DCOPs by agreement. page 8.

[16] M. Vinyals, J. A. Rodriguez-Aguilar, and J. Cerquides. Constructing a unifying theory of dynamic programming dcop algorithms via the generalized distributive law. *Autonomous Agents and Multi-Agent Systems*, 22(3) :439–464, 2010.

[17] R. Wattenhofer. *Principles of Distributed Computing*. 2015.

[18] O. Wolfson and A. Milo. The multicast policy and its relationship to replicated data placement. *ACM Trans. Database Syst.*, 16(1) :181–205, March 1991.

[19] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. Distributed stochastic search and distributed breakout : properties, comparison and applications to constraint optimization problems in sensor networks. 161(1) :55–87.