Université Polytechnique HAUTS-DE-FRANCE

Thèse de doctorat

Pour obtenir le grade de Docteur de

L'Université Polytechnique Hauts-de-France

Et l'Ecole Nationale des Sciences de l'Informatique

Spécialité Informatique

Présentée et soutenue par Fadoua CHAKCHOUK

Le 19/11/2018, à Valenciennes

Ecoles doctorales :

Sciences Pour l'Ingénieur (SPI) - Sciences et Technologies de l'Informatique, des Communications du Design et de l'Environnement (STICODE)

Equipes de recherche, Laboratoires :

Département Informatique.

Laboratoire d'Automatique, de Mécanique et d'Informatique Industrielles et Humaines (LAMIH UMR CNRS 8201) - Thème : Interaction et Agents (InterA)

Laboratoire COSMOS – Thème : Systems for Smart Optimization and Intelligent Engineering (SSOIE)

Contribution à la robustesse dans les CSPs Distribués par réplication locale

JURY

Président du jury

- THOUVENIN Indira Professeur, Université de Technologie de Compiègne, France

Rapporteurs

- OCCELLO Michel Professeur, Université Grenoble Alpes, France
- FARAH Imed Riadh Professeur, Université de Manouba, Tunisie

Examinatrice

- THOUVENIN Indira Professeur, Université de Technologie de Compiègne, France

Co-directeurs de thèse

- MANDIAU René Professeur, Université Polytechnique Hauts-de-France, France
- GHEDIRA Khaled Professeur, Université Centrale de Tunis, Tunisie

Co-encadrant de thèse

- PIECHOWIAK Sylvain Professeur, Université Polytechnique Hauts-de-France, France

Invités

- VION Julien Maître de Conférence, Université Polytechnique Hauts-de-France, France

Remerciement

Je tiens tout d'abord à remercier tous les membres du jury, Occello Michel et Farah Imed Riadh, rapporteurs, Thouvenin Indira, examinatrice, pour le temps consacré à la lecture de ce travail de thèse. Je tiens également à remercier Khaled Ghédira d'avoir accepté de diriger le travail de thèse. Un grand MERCI à René Mandiau et Sylvain Piechowiak qui m'ont encadré pendant ces 4 ans. Merci pour votre disponibilité, pour les échanges d'idées, pour vos précieux conseils, pour le temps passé à corriger le manuscrit, pour votre confiance et pour votre patience. Enfin, merci de m'avoir toujours soutenu et d'avoirs cru en moi et en mes recherches, même dans les périodes difficiles. Je remercie aussi Julien Vion pour ses conseils, sa disponibilité, son encadrement et ses encouragements. Un grand merci aussi à Houcine Ezzeddine qui a suivi cette thèse dès son début, qui a cru en moi jusqu'à la dernière minute et qui n'a jamais arrêté de m'encourager et me soutenir.

Je remercie aussi tous les personnels et les doctorants du LAMIH, spécifiquement Kathia Oliveira, Véronique Delcroix, Sondes Chaabane, Rabie ben Atitallah, Christophe Kolski, qui ont participé à ce doctorat à leur manière. Ensuite, je remercie ma famille de Valenciennes Sandro, Jocelene, Ahmed, Nadya, Ahmad, Taisa, Saleh, Ahlem, José, Wassim, Ali, Shadab et "Pipoka" pour les bons moment, les pauses cafés plus que quotidiennes, les Barbecues qui ont participé à la vie et à la bonne ambiance. Je remercie mes amis en Tunisie, Arij, Khitem, Asma, Ameni, Radhouane, Manel, qui étaient là pendant les moments difficiles et qui n'ont pas arrêté de me supporter et de m'encourager.

Un grand MERCI à ma famille, mon frère, ma belle-soeur, mon oncle, ma tante, et mes cousins d'avoir cru en moi et de m'avoir donné depuis toujours l'opportunité de poursuivre mes objectifs personnels et professionnels et merci pour tout l'amour que vous portez pour moi. MERCI à ma belle-famille d'avoir été à mes côtés. Enfin, je remercie de tout mon cœur mon époux et l'amour de ma vie Hazem. Merci d'être toujours à mes côtés, de m'avoir supporté et soutenu depuis qu'on se connait. Merci d'avoir supporté la distance pour faire cette thèse. Merci pour ta patience. Merci pour les bons moments qu'on a vécu ensemble et pour ta bonne humeur qui m'a donné l'énergie pour finir ce doctorat. Enfin, Merci d'être dans ma vie.

A la mémoire de mes parents.

qui ont cru en moi jusqu'à la dernière minute, qui ont tant rêvé de ce moment,

qui m'ont toujours encouragé et aimé, qui m'ont fait confiance,

Je ne cesserais jamais de vous aimer.

...

A la mémoire de mon grand-père.

ma source de bonheur et de joie, la personne la plus honnête

et la plus aimable, et qui n'a jamais arrêter de me combler d'amour,

Même ton absence ne m'empêchera pas de t'aimer.

...

A mon amour

qui n'a jamais arrêté de me soutenir et me motiver,

qui m'a rendu forte sans qu'il se rend compte,

qui était là sans que je le demande,

qui a su surmonter toutes les difficultés.

A mon âme sœur , mon chéri, mon confident,

mon meilleur ami, mon tout... Je t'aime plus que tout au monde...

surtout ne change pas... tu es parfait tel que tu es

...

A ma famille... mes amis... mes collègues...

Table des matières

Liste des Figures

Liste des Tableaux

Liste des Algorithmes

Introduction Générale

Les caractéristiques et les attentes des applications informatiques évoluent selon les demandes du marché. De nombreuses applications nécessitent des mécanismes de distribution et de partage de données tels que la simulation de trafic routier, la gestion d'un emploi du temps, etc. Les services fournis par ces applications doivent être robuste pour s'adapter à n'importe quel environnement. Pour concevoir de telles applications, les systèmes multi-agents (SMA) ont été proposés comme étant des structures particulières capables d'assurer la décentralisation du traitement des problèmes complexes.

Les systèmes distribués sont caractérisés par la notion de la défaillance partielle : en cas de panne, seulement une partie du système est affectée. Leur nature répartie leur permet d'être robuste. Le concept de la robustesse concerne la tolérance aux fautes et la tolérance aux pannes. Pour ces deux notions, les travaux existants utilisent le terme « fault tolerance » (tolérance aux fautes). Les travaux qui s'intéressent à la tolérance aux fautes des systèmes distribués visent à assurer qu'un système remplit sa fonction en présence de perturbations qui affectent ses composants.

Certains problèmes distribués peuvent être modélisés sous forme d'un DisCSP (problème de satisfaction de contraintes distribué). Ce dernier couvre les domaines des CSPs et des SMAs. Sa résolution est basée sur l'interaction et la coordination entre les agents en décomposant le problème en sous-problèmes (CSP locaux) à résoudre par chaque agent. La tolérance aux fautes au sein d'un DisCSP vise à obtenir une solution (s'il en existe) en présence de perturbations qui peuvent affecter les agents ou la communication entre les agents. Ces perturbations peuvent interrompre le fonctionnement de l'algorithme utilisé pour résoudre le DisCSP global, voir fournir des résultats erronés.

Nous abordons, dans ce travail, la problématique de la résolution d'un DisCSP en présence d'un ou plusieurs agents défaillants. Les travaux qui traitent la tolérance aux fautes dans les SMAs proposent généralement d'augmenter la taille de la population du SMA, soit en répliquant les agents du SMA, soit en utilisant des agents particuliers qui sont chargés de traiter les défaillances. Mais aucun de ces travaux n'a été appliqué pour résoudre des DisCSPs en présence d'agents défaillants.

Dans cette thèse, nous proposons une approche basée sur la réplication des CSPs locaux : chaque CSP local est copié au sein d'un autre agent. De cette façon, en présence d'un agent défaillant, son CSP local sera pris en charge par un autre agent. Ce dernier fusionne le réplicat du CSP de l'agent défaillant avec son propre CSP local. Cette méthode permet de conserver le nombre d'agents et la modélisation initiale du problème. Nous adaptons cette méthode à l'algorithme Multi-ABT

La mémoire sera organisée de la manière suivante :

Chapitre 1 Le premier chapitre présentera les notions de base liées à l'intelligence artificielle appliquée aux systèmes distribués en général, et aux SMAs en particulier. Nous abordons aussi les principes du raisonnement par contraintes en présentant les DisCSPs.

Chapitre 2 Le deuxième chapitre sera dédié aux travaux existants qui s'intéressent à la tolérance aux fautes. Les méthodes de détection des défaillances et les techniques de traitement des défaillances dans les systèmes distribués et les SMAs seront abordées.

Chapitre 3 Dans ce chapitre, nous définirons les différents algorithmes proposés pour détecter, répliquer et fusionner les CSPs locaux. Nous détaillerons aussi les messages échangés entre les agents au cours de la résolution du DisCSP et du traitement des défaillances, ainsi que le comportement adopté par les agents.

Chapitre 4 Ce chapitre est consacré aux expérimentations effectuées afin de valider notre approche. Nous présenterons les résultats obtenus en traitant un ensemble d'instances de DisCSPs générés aléatoirement.

Chapitre 5 Finalement, le cinquième chapitre proposera une amélioration de l'approche

proposée dans le troisième chapitre. Cette amélioration vise à remédier aux limites de l'approche déterminées au cours des expérimentations du chapitre 4.

Chapitre 1

IAD et raisonnement par contraintes

Sommaire

1.1 Introduction

Certains problèmes d'aide à la décision peuvent être résolus via l'intervention d'un ensemble d'entités intelligentes. Ces entités constituent un système distribué dont le but est de remédier aux problèmes des systèmes centralisés. Les Systèmes Multi-Agents (SMAs) présentent une approche dédiée à la résolution des systèmes distribués dans le domaine de l'intelligence artificielle.

Les travaux qui s'intéressent aux SMAs visent à résoudre des problèmes soit physiquement distribués, soit ayant une expertise distribuée. Ces problèmes peuvent être modélisés sous forme de problèmes de satisfaction de contraintes distribués (DisCSPs). La

résolution des DisCSPs est basée sur des échanges de messages entre les agents afin d'obtenir une solution globale à partir de leurs solutions locales. Les types et la façon de traiter ces messages dépendent de l'algorithme de résolution utilisé. Ces algorithmes possèdent tous le même inconvénient : ils ne prennent pas en considération les défaillances qu'un SMA peut subir.

Dans ce chapitre, nous présentons les principes du raisonnement par contraintes dans le domaine de l'intelligence artificielle distribuée. Dans la première section (1.2), nous présentons quelques généralités de l'IA (Intelligence artificielle), de l'IAD (Intelligence Artificielle Distribuée), et des SMAs. Dans la section (1.3), nous présentons les principes et les éléments de base des CSPs distribués.

1.2 L'intelligence Artificielle Distribuée

L'IAD est une extension de l'IA classique. Elle peut être définie comme l'intersection entre l'IA et les systèmes distribués. Dans cette section, nous présentons les notions de l'IA et de l'IAD, en les définissant et en citant les axes de recherche de l'IAD (1.2.1). Nous présentons aussi les SMAs comme exemple type d'un système dédié à résoudre des problèmes d'IAD, en détaillant la notion d'un agent (1.2.2)

1.2.1 De l'IA à l'IAD

L'Intelligence Artificielle (IA)

Plusieurs définitions de l'IA sont proposées dans la littérature. Ces définitions tournent autour d'un seul concept : la modélisation d'un comportement intelligent (un comportement semblable à celui de l'être humain).

> « *L'intelligence artificielle est la construction des programmes capables d'accomplir des tâches nécessitant des processus mentaux de haut niveau. Ces tâches sont mieux accomplies par les êtres humains. En terme informatique, l'IA centralise un problème dans un seul système en* ***modélisant le comportement intelligent d'un seul agent*** » [McCorduck et al., 1977].

« *L'intelligence artificielle peut être définie comme une branche des sciences de l'informatique qui concerne l'**automatisation d'un comportement intelligent*** » [Luger, 2005].

« *L'intelligence artificielle est le domaine qui étudie la synthèse et l'analyse **des agents informatiques qui agissent d'une manière intelligente*** » [Poole and Mackworth, 2010].

[Luger, 2005] a présenté un aperçu sur quelques disciplines d'application de l'IA, tels que les jeux vidéo, les systèmes experts, la compréhension du langage naturel de l'être humain. Mais selon [Labidi and Lejouad, 1993], l'IA est incapable de résoudre des problèmes complexes et hétérogènes, tels que les problèmes physiquement distribués.

L'Intelligence Artificielle Distribuée (IAD)

Le but de l'IA distribuée (IAD) est de remédier aux insuffisances de l'IA en distribuant l'intelligence sur un ensemble d'agents intelligents.

« *L'IAD est **l'étude, la construction, et l'application des systèmes multi-agents**. Ceux sont des systèmes dans lesquels **différents agents** interactifs et intelligents **poursuivent un ensemble de buts ou exécutent un ensemble de tâches*** » [Weiss, 1999].

« *Les systèmes d'IAD sont conçus sous forme d'un **groupe d'entités intelligentes**, dites agents, qui interagissent par coopération, ou par compétition* » [Goyal and Yadav, 2011].

« *L'IAD est **l'étude du comportement d'un groupe d'agents intelligents** pour résoudre un problème global difficile. Le principe est de **diviser le problème original à des problèmes plus simples et de les résoudre indépendamment les uns des autres*** » [Huhns, 2012].

Selon [Goyal and Yadav, 2011, Chaib-Draa et al., 1992], l'IAD vise à : (i) traiter des connaissances distribuées dans des applications géographiquement dispersées, tel que le contrôle du trafic aérien [Amalberti and Deblon, 1992], (ii) étendre la coopération entre l'homme et la machine en proposant une résolution distribuée entre eux. Cette extension

nécessite l'implémentation de machines intelligentes qui peuvent raisonner comme les êtres humains, et (iii) faciliter la représentation des connaissances et la résolution des problèmes physiquement distribués.

Plusieurs travaux tels que ceux présentés par [Goyal and Yadav, 2011] et [Frécon and Kazar, 2009] donnent les différents axes de recherche de l'IAD. Trois axes de base ont ainsi été retenus :

- Intelligence Artificielle Parallèle [Roosta, 2000] : l'IA parallèle s'intéresse aux architectures, aux langages informatiques et aux algorithmes de l'IA, mais non pas aux comportements et aux raisonnements des agents intelligents. L'objectif du parallélisme est d'améliorer la rapidité des traitements des informations.

- Résolution des Problèmes Distribués [Yeoh and Yokoo, 2012] : les problèmes distribués sont résolus en partageant les connaissances sur un ensemble d'entités qui coopèrent ensemble pour avoir une solution. Le principe de la distribution d'un problème particulier, tel que le problème de satisfaction de contraintes (CSP), est de le diviser en sous-problèmes résolus chacun par une entité.

- Système Multi-Agent (SMA) [Ferber, 1995] : un système multi-agents est un ensemble d'agents, situés dans un environnement, ayant un comportement intelligent. Ces agents coopèrent ensemble, en échangeant des messages, pour atteindre un but commun. Plus de détails sont présentés dans la section (1.2.2).

1.2.2 Les Systèmes Multi-Agents (SMA)

Un système multi-agents (SMA) est conçu pour résoudre, soit des problèmes complexes qui ne peuvent pas être résolus d'une manière centralisée, soit des problèmes qui sont de nature distribuée. Un SMA, selon [Briot and Demazeau, 2001], est *un ensemble organisé d'agents*. [Shoham and Leyton-Brown, 2008] l'ont défini comme étant un système qui inclut plusieurs entités autonomes ayant soit des informations différentes, soit des intérêts différents, soit les deux.

Agent : Définition et caractéristiques

Beaucoup de définitions d'un agent existent dans la littérature. Dans ce qui suit, nous présentons quelques unes.

> « *Un agent est une entité physique ou virtuelle* ***capable d'agir dans un environnement****, qui peut* ***communiquer directement avec d'autres agents****, qui possède des ressources propres, et* ***dont le comportement tend à satisfaire ses objectifs****, en tenant compte des ressources et des compétences dont elle dispose, et en fonction de sa perception, de ses représentations et des communications qu'elle reçoit* » [Ferber, 1995].

> « *Un agent est une entité logicielle capable* ***d'agir de manière autonome*** *dans le but d'****accomplir un certain nombre de tâches*** *au nom de son utilisateur et en fonction de son intérêt* » [Paraschiv, 2004]

> « *Un agent est toute entité considérée comme* ***percevant son environnement*** *grâce à des capteurs et qui* ***agit sur cet environnement via des effecteurs*** » [Russell and Norvig, 2010]

Donc pour résoudre un problème distribué décomposé en sous-problèmes, nous définissons un agent comme étant « une entité capable de résoudre un sous-problème, et qui peut interagir avec les autres agents pour résoudre le problème initial. »

Afin d'accomplir ses tâches, un agent doit posséder un ensemble de caractéristiques. Parmi ces caractéristiques, nous trouvons celles présentées par [Briot and Demazeau, 2001] et [Caglayan et al., 1998] :

- Délégation : Chaque agent possède un ensemble de tâches à accomplir, soit à la demande d'un utilisateur, soit d'un autre agent. Nous définissons la délégation comme étant « la prise en charge de la résolution d'un sous-problème affecté à un agent par un autre agent »

- Communication : Chaque agent communique avec soit l'utilisateur, soit un autre agent, afin de recevoir les instructions pour accomplir ses tâches, de l'aider à accomplir les leurs, et d'assurer le suivi de leurs réalisations.

- Autonomie : Un agent peut réaliser ses tâches sans l'intervention d'un élément externe (utilisateur ou autre agent). Il contrôle aussi son comportement et son état interne.

- Situé : Un agent agit avec son environnement composé des autres agents et d'un ensemble d'objets.

- Flexibilité : Un agent doit être capable de répondre aux demandes des utilisateurs ou des autres agents, et de prendre l'initiative au bon moment.

En plus de ces caractéristiques, les agents sont divisés en 2 catégories : agents réactifs [Ferber and Drogoul, 1992], et agents cognitifs [Demazeau and Müller, 1991]. Ces catégories sont obtenues en fonction des capacités et des comportements des agents : (i) Agents réactifs : ils ne possèdent pas de mémoire, ni de représentation globale de leur environnement. Ils ont un comportement simple et sont capables de résoudre un problème complexe sans avoir une vision explicite de leurs buts, ni des mécanismes de planification, et (ii) Agents cognitifs : ils possèdent des connaissances partielles de leur environnement, et des capacités de savoir-faire leur permettant d'interagir avec leur environnement et les autres agents. Ces connaissances et capacités leur permettent de mémoriser certaines situations, les analyser et prévoir les actions à adapter dans le futur afin d'atteindre leurs buts.

[Monier, 2012] a présenté le SMA selon 3 vues (Figure 1.1) : (i) une vue globale qui place le SMA dans un environnement, (ii) une vue sociale composée d'un ensemble d'agents qui interagissent entre eux. Dans notre cas, les agents communiquent les uns avec les autres pour résoudre un DisCSP (les algorithmes de résolution sont définis dans la prochaine section). (iii) une vue locale présentant le mécanisme de raisonnement d'un agent. Dans le cas d'une modélisation d'un DisCSP, cette vue présente chaque CSP à résoudre par un agent (réf. définition 1).

L'utilisation des SMAs pour résoudre un problème distribué, décomposé par les agents ou lié à un système physiquement distribué, conserve l'aspect distribué et les propriétés d'autonomie du problème. Pour exprimer les relations entre les agents du SMA, les travaux existants se sont inspirés des problèmes de satisfaction des contraintes (CSP), en présentant ces relations sous forme de contraintes. D'où l'apparition des CSPs distribués.

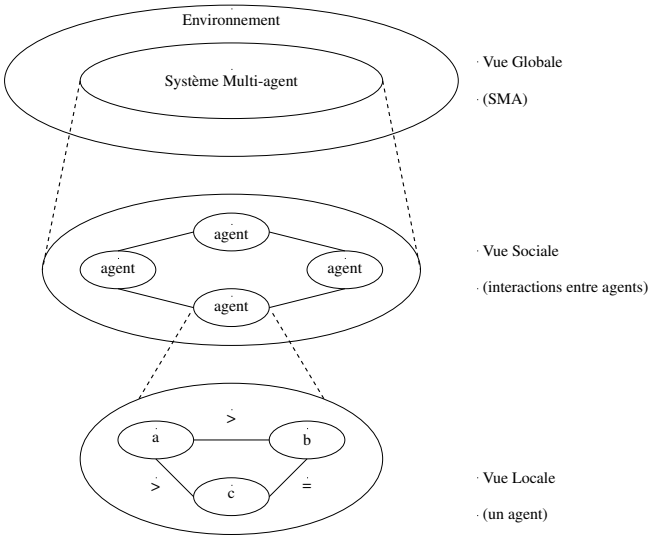


FIGURE 1.1 – Vue Globale, Sociale et Locale du SMA [Monier, 2012]

Dans la section suivante, nous présentons les notions de raisonnement par contraintes en général, et les CSPs distribués en particulier.

1.3 Raisonnement par contraintes

L'utilisation des SMAs pour résoudre des problèmes d'aide à la décision est basée sur l'interaction entre les agents. La relation entre eux a été modélisée sous forme de contraintes liant les agents, générant des problèmes de satisfaction de contraintes distribués (DisCSP).

Dans cette section, nous présentons le formalisme d'un DisCSP, ainsi que les éléments nécessaires pour sa modélisation. Nous définissons aussi les différents termes utilisés dans la littérature pour résoudre un DisCSP (1.3.1). Plusieurs travaux ont proposé des algorithmes de résolution des DisCSPs, parmi lesquels nous présentons l'algorithme ABT, et son extension Multi-ABT (1.3.2).

1.3.1 Définitions

Le DisCSP (Problème de Satisfaction de Contraintes Distribué) est un formalisme de CSP (Problème de Satisfaction de Contraintes) dont les variables sont distribuées entre

les agents d'un SMA constituant ainsi pour chaque agent un CSP local. Ce formalisme a été présenté par [Yokoo et al., 1992].

Définition 1 (CSP)**.** *Un CSP peut être présenté sous forme d'un triplet* $(\mathscr{X}, \mathscr{D}, \mathscr{C})$ *tel que :*

- $\mathscr{X} = \{X_1, X_2, ...X_n\}$ *est l'ensemble de* n *variables qui constitue le CSP.*
- $\mathscr{D} = \{\mathscr{D}_1, \mathscr{D}_2, ...\mathscr{D}_n\}$ *est l'ensemble des domaines de chaque variable* $X_i \in \mathscr{X}$*. Les valeurs de ces domaines sont celles qui peuvent être affectées aux variables.*
- $\mathscr{C} = \{c_1, c_2, ..., c_e\}$ *est l'ensemble de* e *contraintes. Chaque contrainte* c_i *met en relation plusieurs variables de* $\mathscr{X}$*. Les variables d'une seule contrainte* c_i *appartiennent a un ensemble* $vars(c_i)$ *tel que* $vars(c_i) \subseteq \mathscr{X}$*.*

De nombreux problèmes peuvent être modélisés sous forme d'un CSP tels que le problème de coloration de graphe, N-reines, etc. Considérons par exemple la figure 1.2. Elle présente un CSP constitué de 3 variables X_0, X_1 et X_2. Chaque variable possède un domaine contenant les valeurs $\{1, 2, 3\}$. L'affectation des valeurs doit respecter les contraintes $X_0 \neq X_1$, $X_0 \neq X_2$ et $X_1 \neq X_2$. Ce CSP peut avoir comme solution $\{X_0 = 3, X_1 = 1, X_2 = 2\}$. Ces valeurs vérifient bien les contraintes $X_0 \neq X_1$, $X_0 \neq X_2$ et $X_1 \neq X_2$. Si la résolution est effectuée par un SMA, nous parlons des DisCSPs.

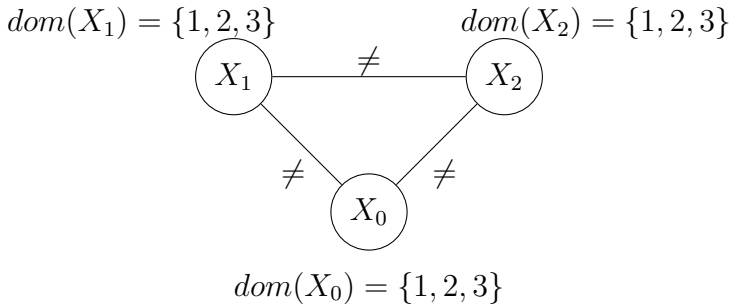


FIGURE 1.2 – Exemple d'un CSP

Définition 2 (DisCSP)**.** *Un DisCSP est défini par un quadruplet* $(\mathscr{X}, \mathscr{A}, \mathscr{D}, \mathscr{C})$ *tel que :*

- $\mathscr{X} = \{X_1, X_2, ...X_n\}$ *est l'ensemble de* n *variables qui constitue le CSP*

- $\mathscr{A} = \{A_1, A_2, ...A_m\}$ *est l'ensemble de* m *agents constituant le Système Multi-agent utilisé pour résoudre le DisCSP. Chaque agent encapsule un ensemble de variables* $(vars(A_i))$ *qui constituent le CSP local à l'agent. Si chaque agent possède une seule variable, le DisCSP est dit* mono-variable*, sinon, le DisCSP est dit* multi-variables*. Chaque variable est affectée à un et un seul agent.*

- $\mathscr{D} = \{\mathscr{D}_1, \mathscr{D}_2, ...\mathscr{D}_n\}$ *est le domaine de chaque variable* $X_i \in \mathscr{X}$

- $\mathscr{C} = \{c_1, c_2, ..., c_e\}$ *est l'ensemble de* e *contraintes. Chaque contrainte associe plusieurs variables. Les contraintes sont divisées en deux catégories : (i) des contraintes intra-agent* $intraC(A_i)$ *qui concernent des variables appartenant à un même agent, et (ii) des contraintes inter-agents* $interC(A_i, A_j)$ *qui concernent des variables appartenant à des agents différents. Dans ce travail, nous nous intéressons aux contraintes binaire, c'est-à-dire, chaque contrainte associe deux variables.*

Définition 3 (Instanciation)**.** *Une instanciation* I *de variables* $vars(I) \subseteq \mathscr{X}$ *est une affectation à chaque variable* $x_i \in vars(I)$ *une valeur prise dans son domaine* $\mathscr{D}_i$*.*

Définition 4 (Solution globale)**.** *Une solution globale est une instanciation de toutes les variables* $\mathscr{X}$ *du DisCSP qui satisfait toutes les contraintes* $\mathscr{C}$ *du problème.*

Définition 5 (Solution partielle)**.** *Une solution partielle* S *concerne une partie des variables* $vars(S) \subset \mathscr{X}$ *du problème. C'est une instanciation qui satisfait les contraintes portant sur les variables* $vars(S)$*.*

Définition 6 (Solution locale)**.** *Une solution locale d'un agent* A_i *est une instanciation de ses variables* $vars(A_i)$*, qui satisfait ses contraintes intra-agent* $intraC(A_i)$*.*

Définition 7 (Nogood)**.** *Un* Nogood *est une instanciation qui ne peut appartenir à aucune solution globale du problème.*

Soit par exemple le CSP distribué à la figure 1.3 [Mandiau et al., 2014] modélisant un problème de coloration de graphe. Le CSP contient 3 agents $\mathscr{A} = \{A_0, A_1, A_2\}$ ayant chacun 3 variables : $vars(A_0) = \{X_0, X_1, X_2\}$, $vars(A_1) = \{X_3, X_4, X_5\}$, et $vars(A_2) = \{X_6, X_7, X_8\}$. Chaque domaine D_i d'une variable X_i contient 2 valeurs :

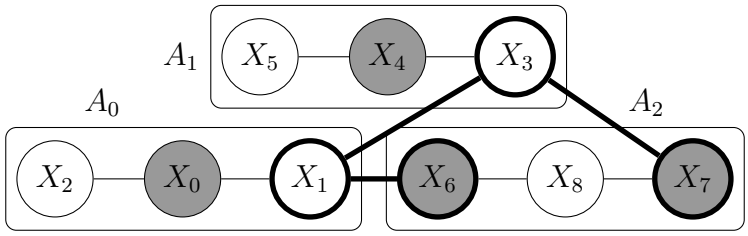


FIGURE 1.3 – Exemple de CSP Distribué [Mandiau et al., 2014]

Blanc ou noir ($D = \{\circ, \bullet\}$). Les arcs présentés en gras présentent les contraintes inter-agents, et le reste des arcs présentent les contraintes intra-agent. Les agents d'un DisCSP sont triés selon un ordre de priorité noté $\succ$ tel que :

Définition 8 ($\succ$)**.** *Soient deux agents A_i et A_j possédant une contrainte en commun. $A_i \succ A_j$ signifie que l'agent A_i possède une priorité supérieure à celle de l'agent A_j. Ce symbole est utilisé aussi pour les variables : si une variable X_i possède une priorité supérieure à celle de X_j, alors $X_i \succ X_j$.*

Définition 9 (Accointances/Voisins)**.** *Les accointances ou voisins d'un agent A_i ($\Gamma(A_i)$) sont les autres agents du système avec lesquels l'agent A_i partage au moins une contrainte c. Les accointances de A_i ayant une prioirté supérieure à la sienne sont dits des accointances supérieures ($\Gamma^+(A_i)$), et ceux ayant une priorité inférieure sont dits des accointances inférieures ($\Gamma^-(A_i)$)*

Définition 10 ($\mathrm{AgentView}$)**.** *Un* $\mathrm{AgentView}$ *est un ensemble associé à chaque agent contenant les différentes instanciations reçues de la part des accointances supérieures de l'agent.*

Les priorités dans la plupart des algorithmes de résolution sont attribuées selon le nombre de contraintes inter-agents de chaque agent : l'agent ayant le plus grand nombre de contraintes inter-agents possède la plus haute priorité, et celui ayant le plus petit nombre de contraintes inter-agents possède la plus petite priorité. Dans l'exemple de la figure 1.3, tous les agents possèdent 3 contraintes inter-agents. Dans ce cas, l'ordre de priorité est affecté selon l'ordre lexicographique des identifiants des agents : par exemple $\Gamma^+(A_0) = \{\}$ et $\Gamma^-(A_0) = \{A_1, A_2\}$.

1.3.2 Asynchronous Backtraking / Multi-Asynchronous Backtracking

Parmi les algorithmes de résolution des DisCSPs proposés dans la littérature, il y a ceux qui sont destinés à la résolution des DisCSPs mono-variables et qui ont été adaptés pour résoudre des DisCSPs multi-variables tels que Asynchronous BackTracking (ABT)/Multi-Asynchronous BackTracking (Multi-ABT), Asynchronous Weak Commitment (AWC)/Multi-Asynchronous Weak Commitment (Multi-AWC) et Distributed Backtracking with Sessions (DBS)/Multi-Distributed Backtracking with Sessions(Multi-DBS). Il y a d'autres algorithmes qui sont destinés à résoudre seulement les DisCSPs multi-variables tel que Asynchronous Forward Checking (AFC). Tous ces algorithmes sont basés sur l'interaction entre les agents, en échangeant leurs solutions locales à travers des messages. Les messages sont traités selon leur ordre de réception. Les algorithmes AWC, DBS et AFC sont présentés dans l'Annexe A.

Asynchronous Backtracking (ABT) [Yokoo, 2001] est le premier algorithme proposé par [Yokoo et al., 1992] pour résoudre des DisCSPs mono-variables d'une façon asynchrone. Il a été utilisé comme référence pour plusieurs travaux dans le domaine des DisCSPs tels que [Meisels and Zivan, 2007, Bessière et al., 2001]. Un ordre de priorité est affecté aux agents. Chaque agent affecte une valeur à sa variable et l'envoie à ses accointances inférieures via un message $\langle \mathrm{OK?} \rangle$. En recevant le message $\langle \mathrm{OK?} \rangle$, l'agent essaye d'affecter à sa variable une valeur de son domaine qui soit cohérente avec le contenu du message reçu. Si aucune valeur du domaine n'est cohérente avec les instances reçues, il en informe ses accointances supérieures via un message $\langle \mathrm{BT} \rangle$. Ces derniers essaieront de modifier les valeurs courantes de leurs variables.

Cet algorithme a été étendu à Multi-ABT [Hirayama et al., 2000, Hirayama et al., 2004] pour l'adapter à la résolution des DisCSPs multi-variables. Il fonctionne de la même façon que ABT, mais dans Multi-ABT, nous parlons d'un ordre de priorité établi entre les agents et aussi entre les variables internes de chaque agent. Afin de trouver une solution au DisCSP, les agents procèdent comme suit :

- Chaque agent propose une solution à son CSP local et la transmet à ses accointances inférieures, via un message $\langle \mathrm{OK?} \rangle$.

- Chaque agent recevant un message $\langle \mathrm{OK?} \rangle$ enregistre son contenu dans son $\mathrm{AgentView}$,

et cherche une solution locale.

- Pour obtenir une solution locale, l'agent affecte à chaque variable de son CSP local une valeur de son domaine. Cette valeur doit être cohérente avec le contenu de son $\mathrm{AgentView}$ et avec les valeurs de ses autres variables de priorité supérieure.

- Si l'agent ne trouve pas de valeur cohérente à une de ses variables, il crée un Nogood. Ce dernier contient les instanciations pour lesquelles l'agent ne peut pas instancier une de ses variables.

- L'agent envoie ce Nogood via un agent $\langle \mathrm{BT} \rangle$ à l'agent de plus petite priorité appartenant au Nogood. L'agent destinataire doit avoir obligatoirement une priorité supérieure à celle de l'agent émetteur du Nogood.

- En recevant le message $\langle \mathrm{BT} \rangle$, l'agent essaye de trouver une solution cohérente avec son $\mathrm{AgentView}$ et différente de sa solution locale courante. Un agent ne traite le contenu d'un message $\langle \mathrm{BT} \rangle$ que si la valeur de sa variable appartenant au Nogood reçu est identique à sa valeur courante. Sinon, le message est considéré comme obsolète. Si aucune valeur n'est cohérente, il transmet le Nogood au prochain agent ayant une faible priorité via un nouveau message de backtracking.

- Si le Nogood contient une instanciation d'une variable d'un agent non voisin, l'agent recevant le message $\langle \mathrm{BT} \rangle$ envoie à cet agent une demande de création de lien.

L'algorithme s'arrête si tous les agents trouvent des solutions locales qui satisfont les contraintes inter-agents du problème, ou lorsqu'il n'y a plus d'échange de messages entre les agents.

Considérons l'exemple présenté par la figure 1.3. Afin de résoudre ce DisCSP, l'échange des messages entre les agents est présenté par la figure 1.4. Les agents A_0 et A_1 envoient leurs solutions $\{X_0 = \bullet, X_1 = \circ, X_2 = \circ\}$ et $\{X_3 = \circ, X_4 = \bullet, X_5 = \circ\}$ aux agents $\{A_1, A_2\}$ et $\{A_2\}$ respectivement. En recevant M_1, l'agent A_1 modifie sa solution et la communique à l'agent A_2 via le message M_4. Lorsqu'il reçoit les messages M_2 et M_3, l'agent A_2 modifie sa solution et attend la réception du prochain message. Mais en recevant le message M_4, l'agent A_2 ne trouve pas de solution cohérente avec son $\mathrm{AgentView} = \{X_1 = \circ, X_3 = \bullet\}$. Donc, il envoie un message $\langle \mathrm{BT} \rangle$ à l'agent A_1 qui crée

le $\text{Nogood} = \{X_1 = \circ\}$ et le transmet à l'agent A_0. Quand l'agent A_0 reçoit le message de backtracking M_6, il cherche une solution dans laquelle $X_1 \neq \circ$. Alors, il modifie sa solution $\{X_0 = \circ, X_1 = \bullet, X_2 = \bullet\}$ et l'envoie aux agents A_1 et A_2 grâce aux messages M_7 et M_8. L'agent A_1 met à jour sa solution $\{X_3 = \circ, X_4 = \bullet, X_5 = \circ\}$ et l'envoie à l'agent A_2 via le message M_9. Ces étapes sont répétées jusqu'à ce que l'agent A_0 se trouve dans une situation où il n'a plus de solution à proposer. Ce DisCSP n'a donc pas de solution et l'algorithme s'arrête.

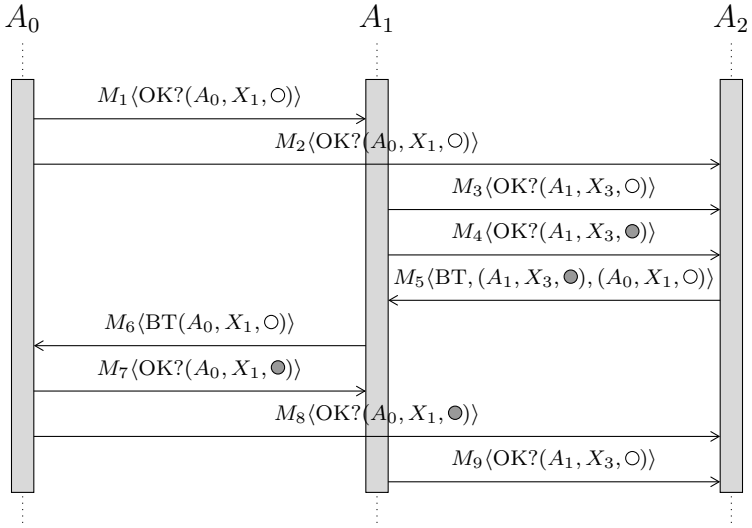


FIGURE 1.4 – Messages échangés entre les agents (Multi-ABT)

Cet algorithme fonctionne correctement et donne les résultats attendus tant qu'il n'y a pas d'événement inattendu qui intervienne. Mais en présence d'un événement perturbant tel que la présence d'un agent malveillant, défaillant, ou d'une rupture de communication entre les agents, les algorithmes continuent à fonctionner mais en fournissant des résultats erronés. La figure 1.5 présente la résolution de l'exemple de la figure 1.3 en présence d'un agent défaillant. Nous avons simulé la défaillance de l'agent A_0 juste après avoir envoyé le message M_2. Nous remarquons que l'échange de messages s'arrête à l'envoi du message M_4. L'algorithme fournit l'instanciation $\{X_0 = \circ, X_1 = \bullet, X_2 = \bullet\}, \{X_3 = \circ, X_4 = \bullet, X_5 = \circ\}, \{X_6 = \bullet, X_7 = \bullet, X_8 = \circ\}$, comme solution au problème, ce qui n'est pas le cas.

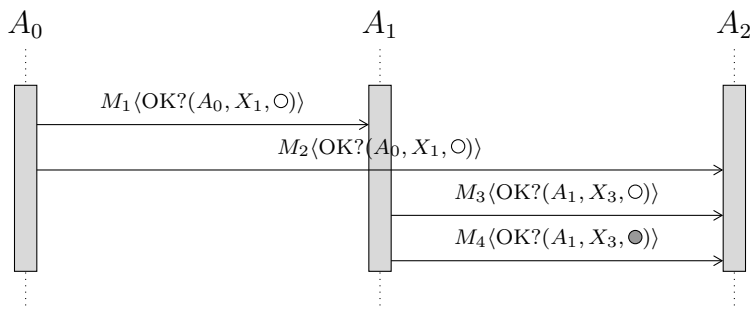


FIGURE 1.5 – Application de Multi-ABT en présence d'un agent défaillant

1.4 Conclusion

Dans ce chapitre, nous avons présenté le contexte auquel appartient la résolution d'un DisCSP : elle est placée dans les domaines des SMAs et de raisonnement par contraintes. Afin de résoudre un DisCSP, les agents interagissent ensemble selon un algorithme de résolution. Les algorithmes de résolution existants dans la littérature sont nombreux, mais possèdent un inconvénient en commun : il ne sont pas tolérants aux fautes en fournissant des résultats erronés lors de l'intervention d'un événement imprévu.

Le but de ce travail est de garantir l'obtention d'une solution d'un DisCSP, s'il en existe, en présence d'agents défaillants. Pour ce faire, nous présentons, dans le prochain chapitre, les différentes méthodes de tolérance aux fautes au sein des systèmes distribués en général, et dans les SMAs en particulier.

Chapitre 2

Robustesse et tolérance aux fautes

Sommaire

2.1 Introduction

Certains problèmes de la vie réelle modélisés sous forme d'un CSP peuvent appartenir à un environnement incertain ce qui peut provoquer une modification de la modélisation initiale du problème. Ces modifications peuvent être de différents types : retrait et/ou ajout de variables, de valeurs dans le domaine des variables, ou de contraintes, d'où l'apparition

des CSPs dynamiques [Dechter and Dechter, 1988]. La méthode de résolution d'un CSP dynamique doit garantir la robustesse de la solution obtenue. En effet, la solution obtenue comme résultat de la résolution du CSP final n'est pas forcément valide pour le CSP initial.

De nombreuses méthodes ont été proposées pour maintenir la robustesse d'un CSP après la modification de sa modélisation. [Hebrard et al., 2004] et [Hebrard, 2004] ont proposé une méthode qui traite la perte de valeurs dans le domaine des variables. Ils proposent d'attribuer d'autres valeurs aux domaines des variables dont le domaine est modifié, ou de modifier les valeurs des domaines des autres variables. De cette façon, ils garantissent que les valeurs des domaines de toutes les variables sont valides pour toutes les solutions possibles du CSP. [Climent et al., 2010] traitent le problème de restriction des contraintes, en associant 2 paramètres à chaque contrainte. À partir de ces paramètres, de nouvelles contraintes apparaissent, l'espace de recherche des solutions est réduit, et de nouveaux CSPs apparaissent. La solution valide avec le maximum des nouveaux CSPs est la plus robuste. [Wallace and Freuder, 1998] ont eu recours à la pénalisation des valeurs qui ne sont plus valides après modification du CSP. L'algorithme de recherche des solutions utilisé essaye de trouver des solutions qui n'impliquent pas ces valeurs. [Climent et al., 2013] introduisent le concept du *covering*. Il s'agit de mesurer la protection de la solution trouvée contre les changements possibles en utilisant la *« topologie d'oignon »* : c'est une structure présentée sous forme de plusieurs couches. Les solutions sont présentées dans l'espace de recherche. La solution la plus robuste est celle trouvée au centre de la structure, puisqu'elle est protégée par les solutions appartenant aux différentes couches qui l'entourent.

Ces méthodes proposées pour maintenir la robustesse d'un CSP en cas de modification de sa modélisation initiale ne maintiennent pas forcément la robustesse d'un DisCSP. Un DisCSP peut être exposé à de différentes perturbations qui affectent les agents. Dans ce cas, nous invoquons la notion de la tolérance aux fautes dans les SMAs. Notre objectif est de résoudre un DisCSP en présence de plusieurs agents défaillants. Un SMA est connu par son aspect distribué, donc nous ne pouvons pas parler des SMAs sans faire intervenir les systèmes distribués. Dans ce contexte, les SMAs se retrouvent dans deux positions : (i) un SMA est tolérant aux fautes par sa nature modulaire ; les failles peuvent être isolées

et ne se propagent pas ; et (ii) un SMA est non sécurisé, car le contrôle est distribué, et difficile de garantir un comportement spécifique, en particulier dans les situations de panne. Dans ce chapitre, nous nous intéressons à la tolérance aux fautes au niveau des systèmes distribués en général, et au niveau des SMAs en particulier. Nous présentons les concepts de base de la tolérance aux fautes (2.2). Ensuite, nous nous intéressons aux différentes techniques de détection des défaillances, et de traitement des pannes au niveau des systèmes distribués (2.3). Finalement, nous présentons les défaillances auxquelles un SMA peut faire face, et les techniques actuelles présentées pour les traiter (2.4).

2.2 Concepts de base

Dans cette section, nous présentons les définitions des éléments de base de la tolérance aux fautes dans les systèmes distribués et dans les SMAs (2.2.1). Ensuite, nous citons les différents modes de défaillance qui peuvent impacter un système distribué (2.2.2).

2.2.1 Définitions

La tolérance aux fautes est une des caractéristiques qui différencie les systèmes distribués des systèmes centralisés. Une perturbation au niveau d'un système centralisé affecte le système en entier. Par contre, au niveau des systèmes distribués, une perturbation peut affecter une seule entité tandis que les autres entités continuent à fonctionner.

Définition 11 (Système Distribué)**.** *« Un système distribué est un ensemble de machines indépendantes qui aparaissent aux utilisateurs comme un seul système cohérent » [Tanenbaum and Steen, 2006].*

« Un système distribué est un ensemble de composants matériels ou logiciels (applications) situés dans des machines connectées en réseau, et qui communiquent et coordonnent leurs actions en échangeant des messages. » [Coulouris et al., 2011]

« Un système distribué est un ensemble d'entités indépendantes qui coopérent pour résoudre un problème qui ne peut pas être résolu individuellement. » [Kshemkalyani and Singhal, 2011].

Définition 12 (Fonction d'un système)**.** *« La fonction d'un système est ce à quoi il est*

destiné. Elle est décrite par la spécification fonctionnelle, qui inclut les performances attendues du système » [Arlat et al., 2006]

Définition 13 (Service d'un système)**.** *« Le service délivré par un système est son comportement tel que perçu par ses utilisateurs (séquence d'états externes) » [Arlat et al., 2006]*

Définition 14 (Tolérance aux fautes)**.** *« La tolérance aux fautes d'un système est sa capacité de se comporter d'une manière bien définie une fois que des erreurs parviennent. » [Gärtner, 1999].*

« La tolérance aux fautes vise à maintenir la délivrance d'un service correct en présence de fautes actives. Elle est généralement mise en œuvre par la détection d'erreur et le recouvrement du système, et éventuellement par le contrôle des erreurs. » [Lussier et al., 2004]

Selon [Avizienis et al., 2004], la tolérance aux fautes vise à éviter la défaillance des services en présence des fautes. Ces fautes peuvent entrainer des erreurs et, éventuellement, une défaillance. D'où les définitions de la défaillance, erreur et faute.

Définition 15 (Défaillance)**.** *La défaillance est un évènement provoqué par la déviation d'un service correct à un service incorrect. Donc une défaillance d'un service signifie qu'au moins un état externe dévie de l'état correct du service. Cette déviation est dite* erreur.

Définition 16 (Erreur)**.** *Une erreur est la partie de l'état général du système qui peut conduire à une défaillance. Elle peut se propager au sein d'un des composants du système. Cette propagation est causée par le processus de calcul qui transforme une erreur en d'autres. Une erreur est dite latente si elle n'a pas été reconnue, et détectée si elle a été détectée par un algorithme ou un mécanisme de détection. L'erreur est causée par une* faute.

Définition 17 (Faute)**.** *Une faute peut intervenir lors de la phase de la création du système, telles que les fautes de développement. Elle peut être interne, ou externe. Elle cause d'abord une erreur dans l'état interne du système, l'état externe n'est pas immédiatement affecté. Une faute peut affecter aussi le matériel, ou les logiciels et les données.*

2.2.2 Modes de défaillance

Le but de notre recherche est de garantir la résolution d'un DisCSP en présence d'un agent défaillant. Ci-dessous nous présentons les différents modes de défaillance qu'un système peut subir. Selon [Avizienis et al., 2004], un service est considéré incorrect à partir de 4 points de vue : (i) défaillance du domaine ; (ii) détection de la défaillance ; (iii) cohérence de la défaillance, et (iv) conséquences de la défaillance.

1. *Défaillance du domaine* : La défaillance du domaine implique (i) la défaillance du contenu ; et/ou (ii) la défaillance temporelle.

   - **La défaillance du contenu :** Le contenu de l'information fournie à l'interface du système ne correspond pas à la fonction implémentée du système.
   - **La défaillance temporelle :** L'instant ou la durée de la délivrance de l'information ne correspond pas à la fonction implémentée du système. La délivrance du service peut être précoce (avant l'instant désiré), ou retardée.

   La combinaison de ces 2 modes de défaillances, c'est-à-dire, si le service est délivré à n'importe quel moment, et contient de fausses informations, la défaillance est dite **byzantine**. Ce cas peut appartenir aussi à 2 classes :

   - **Arrêt du service :** Un service s'arrête si son état externe devient constant. C'est-à-dire l'activité du système n'est plus lisible aux utilisateurs. Ce cas est dit aussi, **crash failure** [Nasreen et al., 2016]. Si aucun service n'est délivré, la défaillance est dite **silencieuse**, ou aussi, **une défaillance d'omission** [Nasreen et al., 2016]. Dans ce cas, un service ne peut plus recevoir ou envoyer des messages. C'est un cas particulier de l'arrêt du service.
   - **La défaillance imprévisible :** Une défaillance est dite imprévisible si le service est délivré mais d'une façon instable (e.g bruit indésirable sur une ligne téléphonique).

2. *Détection de la défaillance* : La défaillance d'un service est détectée par des mécanismes de détection. Si le mécanisme détecte que le service fourni est incorrect, la défaillance est dite **signalée**, sinon elle est **non signalée**. Ces mécanismes peuvent

aussi signaler une défaillance qui n'a pas eu lieu, c'est une **fausse alerte**. Ou aussi, ils peuvent ne pas signaler une défaillance existante (défaillance non signalée).

3. *Cohérence de la défaillance* : La cohérence d'une défaillance est distinguée si le système possède plusieurs utilisateurs. Une défaillance est **consistante** si les utilisateurs perçoivent de la même façon le service incorrect. Sinon, si certains ou tous les utilisateurs le perçoivent différement, la défaillance est dite **inconsistante**.

4. *Conséquences de la défaillance* : Le degré de la gravité de la défaillance est défini à partir de la relation entre les bénéfices fournis par le service en présence et en absence de la défaillance.

    - **Défaillance mineure** : Une défaillance est dite mineure si les conséquences de la délivrance d'un service incorrect ont un coût similaire aux bénéfices de la délivrance d'un service correct.
    - **Défaillance catastrophique** : Une défaillance est dite catastrophique si les conséquences de la défaillance ont un coût plus élevé que celui de la délivrance d'un service correct.

Ils existent aussi des **défaillances de développement**. Une telle défaillance peut causer un arrêt du processus avant même que le système soit utilisé. Une défaillance de développement peut être sous forme de : (i) **une défaillance de budget**, où le budget dédié au développement est épuisé avant que le système ne soit accepté ; et (ii) **une défaillance de planning**, où le développement du système dépasse le délai de livraison, donc le système devient obsolète, ou il ne respecte plus les besoins de l'utilisateur.

Pour faire face à ces défaillances, de nombreux travaux ont proposé des méthodes de tolérance aux fautes. Dans la section qui suit, nous présentons l'aspect de tolérance aux fautes dans les systèmes distribués.

2.3 Tolérance aux fautes dans les systèmes distribués

Le processus de la tolérance aux fautes commence tout d'abord par détecter la défaillance à laquelle le système doit faire face. Après avoir détecté la défaillance, une tech-

nique de tolérance aux fautes est appliquée pour garantir le fonctionnement correct d'un système en présence d'une défaillance.

Dans cette section, nous présentons les différentes propriétés d'un détecteur de défaillance, ainsi que quelques techniques de détection de défaillance adaptées dans les systèmes distribués (2.3.1). Ensuite, nous présentons les différentes méthodes de la tolérance aux fautes proposées dans les travaux existants, et utilisées dans les systèmes distribués (2.3.2).

2.3.1 Détection des défaillances

Le rôle d'un détecteur de défaillances est de surveiller les composants du système en question, et de les informer en cas de défaillance d'un ou plusieurs composants. Mais le comportement d'un détecteur n'est pas forcément fiable. En effet, il peut ajouter, par erreur, un composant actif à sa liste des composants soupçonnés d'être défaillants (*fausse alerte*). Le détecteur peut rectifier cette erreur plus tard en supprimant le composant actif de sa liste des composants défaillants. Pour éviter ces fausses alertes, un détecteur doit être : (i) Complet, qui soupçonne tous les composants défaillants. Si chaque composant défaillant est suspecté par tous les composants non défaillants, la complétude est dite **forte**. Et s'il est suspecté par au moins un seul composant non défaillant, la complétude est dite **faible** ; (ii) Exact, qui ne suspecte aucun composant non défaillant. Si les composants non défaillants ne sont jamais suspectés, l'exactitude est dite **forte**. Et s'il existe au moins un composant non défaillant qui n'est jamais suspecté, l'exactitude est dite **faible**.

[Hayashibara et al., 2002] a introduit deux stratégies d'interaction entre les détecteurs et les composants surveillés. Ces stratégies permettent de comprendre les comportements des protocoles de surveillance utilisés par les détecteurs de défaillance.

Modèle « push » Dans ce modèle, les composants surveillés sont actifs et le détecteur de défaillances est passif. Chaque composant surveillé envoie périodiquement des messages au détecteur qui le surveille. Un détecteur de défaillance suspecte une défaillance d'un composant s'il ne reçoit pas de message de sa part dans un intervalle de temps (timeout).

Modèle « pull » Dans ce modèle, les composants surveillés sont passifs, et le détecteur

de défaillances est actif. Le détecteur envoie périodiquement des messages de vérification de l'état aux composants. A la réception de ce message, chaque composant surveillé envoie une réponse au détecteur. Si le détecteur ne reçoit pas de réponse d'un composant dans un certain délai (timeout), il suspecte la défaillance du composant. Cette méthode sera utilisé pour détecter la présence d'un agent défaillant en appliquant notre approche présentée dans le chapitre 3.

Plusieurs travaux ont été proposés dans la littérature. Tous ces travaux se basent sur l'interaction entre les composants et des détecteurs qui surveillent le comportement des composants.

Service de détection des défaillances [Stelling et al., 1999] ont proposé une architecture comportant deux couches : une couche inférieure comportant des détecteurs locaux et une couche supérieure comportant des collecteurs de données. Le détecteur local est responsable de la surveillance de son propre composant. Chaque détecteur envoie périodiquement des messages aux collecteurs de données contenant des informations sur les composants surveillés. Les collecteurs des données reçoivent ces messages, identifient les composants défaillants et en informent les utilisateurs du système.

Approche hiérarchique [Felber et al., 1999] proposent un service ayant une structure hiérarchique. Cette structure organise les processus sous forme hiérarchique où les messages sont échangés. Chaque sous-réseau possède des détecteurs de défaillance. Chaque détecteur est responsable de surveiller les composants d'un même niveau. Les messages de surveillance entre un détecteur et les composants surveillés sont envoyés uniquement au sein du sous-réseau. Les messages entre les détecteurs sont envoyés à travers les sous-réseaux. Donc, les détecteurs de défaillances obtiennent des informations de soupçon soit par des messages provenant d'autres détecteurs, soit en surveillant directement les composants cibles.

Protocole de détection paresseux [Fetzer et al., 2001] ont proposé un protocole permettant à un composant de surveiller un autre composant en utilisant des messages d'application. Ces derniers permettent aux composants d'obtenir des informations sur la dé-

faillance d'un composant. Les messages sont échangés à travers trois primitives : (i) La primitive SEND, utilisée pour envoyer un message d'application à un autre processus. Elle inclut des informations de contrôle ; (ii) la primitive RECEIVE utilisée pour recevoir un message d'application ; et (iii) la primitive QUERY utilisée pour savoir si un composant est suspecté d'être défaillant.

2.3.2 Techniques de tolérance aux fautes

La tolérance aux fautes est une priorité qui caractérise les systèmes distribués par rapport aux systèmes centralisés. Les techniques de tolérances aux fautes proposées assurent la continuité de fonctionnement des systèmes en présence de défaillances. Ces techniques sont basées sur le concept de *la redondance*, dite aussi *réplication*. Il s'agit d'avoir différentes copies, dites aussi réplicats, d'un objet ou d'une donnée.

La réplication peut être temporelle, physique, ou logicielle : (i) la réplication temporelle vise à remédier contre les défaillances non permanentes, en reprenant à nouveau une action particulière ; (ii) la réplication physique est destinée à traiter les défaillances au niveau d'un composant en lui-même en faisant fonctionner un même programme sur plusieurs machines ; (iii) la réplication logicielle est proposée afin de remédier aux fautes de développement, ou de conception. Dans ce cas, le même programme est conçu et développé de manières différentes sur différentes machines.

[Wiesmann et al., 1999] proposent deux types de réplication : la réplication passive et la réplication active. Ces types de réplication appartiennent à la réplication physique.

Réplication passive (Primary-backup) La réplication passive copie un composant en n exemplaires. Ces exemplaires comportent une copie particulière dite *primaire*, et $n-1$ copies *secondaires*. La copie primaire est la seule copie qui communique avec le processus client, et qui effectue tous les traitements. Les copies secondaires communiquent directement avec la copie primaire. En cas de défaillance de cette dernière, une copie secondaire prend le relève. Après le traitement d'un message reçu de la part du processus client, la copie primaire transmet aux copies secondaires le résultat du traitement, ainsi qu'une mise à jour de son état d'exécution. Chaque copie secondaire enregistre l'état reçu de la copie primaire en constituant un point de reprise. Ce point permet de synchroniser

les états des copies secondaires avec celle de l'état primaire. De cette façon, en cas de défaillance de la copie primaire, la copie secondaire, qui prend le rôle de la copie primaire, reprend son état à partir du dernier point de reprise enregistré. Ce type de réplication a été utilisé par [Almeida et al., 2007].

Réplication active La réplication active copie un composant en n exemplaires identiques interagissant tous avec le processus client. Ces copies possèdent des comportements symétriques : ils reçoivent les mêmes messages, produisent le même résultat, et envoient les mêmes réponses dans le même ordre. Puisque tous les résultats sont identiques, le processus client choisit le premier message reçu comme réponse. En cas de défaillance d'une copie, le service délivré par le composant n'est pas perturbé. En effet, cette défaillance est masquée par le comportement des autres copies puisqu'elles possèdent toutes le même comportement. Ce type de réplication a été utilisé par [Khan et al., 2005].

Reprise et points de reprise (roll back and checkpoints) La technique de reprise (roll back) présente une technique de réplication temporelle. Les informations d'un composant, tels que son état actuel, son environnement, etc, sont stockées dans un périphérique stable distinguées par des points de reprise (checkpoints). Ce stockage facilite le retour en arrière en cas de défaillance : les composants utilisent les informations enregistrées pour restaurer un état global cohérent, et reprennent l'exécution à partir de cet état. Cette technique a fait l'objet d'une approche proposée par [Perumalla and Park, 2014].

Par contre, cette technique n'est pas compatible avec n'importe quelle application. En effet, la taille des points de reprise, ainsi que le coût temporel de la reprise, imposent une structuration des contraintes qui doit être prise en considération lors du développement de l'application. Certaines de ces techniques ont été adaptées pour remédier aux défaillances qu'un SMA peut avoir. Dans la section suivante, nous présentons les techniques de tolérance aux fautes au sein des SMAs.

2.4 Tolérance aux fautes dans les systèmes multi-agents

Plusieurs fautes intervenant dans les SMAs peuvent provoquer des défaillances équivalentes à celles présentées dans les systèmes distribués. Dans cette section, nous présentons les perturbations qu'un SMA peut subir, en les classifiant selon les modes de défaillances des systèmes distribués (2.4.1). Ensuite, nous présentons les différentes techniques proposées dans la littérature pour remédier à ces perturbations (2.4.2).

2.4.1 Les défaillances dans les SMAs

Un SMA est défini comme étant un type particulier de système distribué. Il peut avoir le même type de défaillances qu'un système distribué : il peut avoir des défaillances du domaine (défaillance d'un agent), des défaillances byzantines (la présence d'un agent défectueux), des défaillances d'omission (interruption de la communication entre les agents), etc. Donc, au sein d'un SMA, les défaillances peuvent être au niveau de l'agent, de la plateforme de l'agent, ou de la communication.

La plupart des travaux existants dans la littérature, proposant des techniques de tolérance aux fautes au niveau des SMAs, présentent des approches générales sans spécifier exactement le type de défaillance à traiter. Les défaillances spécifiées par les autres travaux peuvent être classifiées selon les types de défaillances dans les systèmes distribués (Tableau 2.1)

TABLE 2.1 – Types de défaillances

Types	Fautes et Défaillances	Références
« Crash failure »	Fautes de processeur	[Fedoruk and Deters, 2002]
	Défaillance de processeur	[Hägg, 1997]
	Agent ne donne plus de réponse	[Almeida et al., 2007]
« Omission failure »	Fautes de communication	[Fedoruk and Deters, 2002]
	Erreurs de communication	[Hägg, 1997]
	Défaut logiciel et états inattendus	[Fedoruk and Deters, 2002]
« Byzantine failure »	Comportement émergeant non désiré	[Fedoruk and Deters, 2002]
	Présence d'agent malveillant	[Dellarocas and Klein, 2000]

Certaines défaillances sont spécifiques aux systèmes multi-agents. Ces défaillances peuvent toucher la partie environnement du SMA, tel que l'augmentation de la taille de la population, le changement des tâches au fil du temps. Dans ces cas, certains travaux tels que [Frey et al., 2003, Nagi, 2001] utilisent des technologies de base de données pour présenter l'environnement d'un SMA. Ces technologies fournissent des services garantissant la robustesse des tâches exécutées par les agents.

2.4.2 Techniques de tolérance aux fautes

La tolérance aux fautes dans les SMAs commence par la détection des défaillances. Cette détection peut être définie comme étant un composant logique du système, ou un agent spécifique qui prédit les états des agents et des services du système. Une défaillance peut être détectée en appliquant la notion du *timeout* (aucun échange de message pour une longue durée), ou si des accusés de réceptions manquent. La procédure à exécuter pour traiter la défaillance détectée est définie par un protocole de tolérance aux fautes. Ils existent deux types : (a) un protocole d'une et une seule fois, où un agent doit exécuter une action désirée une et une seule fois, (b) un protocole non bloquant, destiné aux problèmes ne pouvant pas être traités par le premier protocole. Dans ce cas, soit le processus défaillant est repris dès le début, soit les agents sont répliqués.

1. Réplication des agents

La technique de réplication des composants est proposée comme une méthode de tolérance aux fautes dans les systèmes distribués, qui a été adapté aux SMAs.

Les agents critiques [Guessoum et al., 2004] proposent une approche basée sur la réplication dynamique des agents. Cette méthode vise à répliquer les agents critiques. La criticité d'un agent définit son importance et son influence, en cas de défaillance, sur le comportement du système. Le nombre de réplicats des agents est adapté dynamiquement en fonction de la criticité des agents. La réplication des agents critiques est appliquée quand la réplication simultanée de tous les composants d'un SMA n'est pas possible. La criticité d'un agent est calculée en fonction de deux types d'informations : (i) des informations du niveau système qui permettent d'évaluer le degré d'activité d'un agent, tel

que les charges de communication, le temps de traitement, etc. ; (ii) des informations au niveau sémantique, qui dépendent de l'agent et du paradigme choisi.

L'approche proposée est basée sur les réseaux d'interdépendance établis à partir d'un mécanisme d'observation. Afin de contrôler et observer les agents, un agent réactif (*moniteur*) est affecté à chaque agent, et un *observateur* est affecté à chaque machine. Cette observation prend en compte le nombre de messages échangés, le temps de traitement du problème à résoudre, le taux d'informations échangées, etc.

Une autre approche, proposée par [Almeida et al., 2006], utilise la notion de la réplication des agents critiques. La criticité des agents est calculée par une approche prédictive d'une manière dynamique. La méthode est basée sur la prédiction du comportement à venir d'un agent. Elle vise à traiter les défaillances d'un agent ou d'une machine.

Réplication transparente Une approche basée sur la notion de proxy est proposée par [Fedoruk and Deters, 2002]. Un proxy est présenté sous forme d'une interface entre les réplicats et le reste du système. Lors de l'interaction entre les agents et les réplicats, un agent ne se rend pas compte qu'il est entrain d'interagir avec un ensemble de réplicats. On parle, dans ce cas, de la transparence de la réplication.

Les proxys gèrent les groupes de réplicats : ils choisissent le type de réplication à adapter. Ils sont responsables aussi des réponses transmises aux réplicats : ils décident, par exemple, s'ils transmettent une synthèse des réponses reçues de la part des réplicats, ou une des réponses selon un ensemble de critères. Ils peuvent aussi gérer la performance des groupes des réplicats : ils décident quel réplicat sera actif au sein d'un groupe à partir d'un ensemble de caractéristiques des réplicats. Un proxy est soit informé de ces caractéristiques, soit il les a appris en observant les comportements de chaque réplicats. Ainsi, les proxys remplissent deux fonctions : ils font apparaître, d'une part, le groupe répliqué comme une entité unique, et, d'autre part, ils contrôlent l'exécution et la gestion d'état d'un groupe répliqué.

2. Les groupes

Certains auteurs utilisent la notion de groupe dans les SMAs. Les membres d'un groupe possèdent un but commun en plus de leurs buts individuels. Ces groupes sont

composés d'un ensemble d'agents pouvant être virtuels, spécifiques, etc.

Les brokers [Kumar and Cohen, 2000, Kumar et al., 2000] ont proposé une approche basée sur des groupes de *brokers*. C'est un agent intermédiaire qui peut offrir divers services (chercher des agents capables d'effectuer une certaine tâche, router des requêtes et des réponses, etc). Ils sont organisés par équipe. Chaque équipe contient un ensemble de brokers et d'agents. Les *brokers* d'une équipe possèdent un but et un état interne communs. Les agents d'une équipe ne réalisent pas tous les mêmes tâches. En cas de défaillance d'un *broker*, un autre de son équipe le remplace. Il peut créer de nouveaux *brokers* afin de maintenir un nombre minimal dans l'équipe. Puis, il informe les membres de son équipe de l'ajout d'un nouveau membre.

Virtual Agent Cluster (VAC) Une architecture décentralisée est proposée par [Khan et al., 2005], adoptant un paradigme peer-to-peer. Cette architecture est basée sur des *clusters* d'agents virtuels (VAC). Un *cluster* englobe toutes les machines sur lesquelles une seule plateforme est déployée. Il fournit une interface à travers laquelle les agents peuvent communiquer (via le langage de communication ACL). Chaque *cluster* possède un identifiant global unique. Chaque machine au sein d'un VAC possède un identifiant unique pour router les messages, et éviter les communications supplémentaires entre les machines. En cas de défaillance d'une machine, les machines du même VAC coopèrent pour remédier à cette défaillance.

3. Les sentinelles

Les sentinelles représentent la structure de contrôle d'un système multi-agents. Elles sont chargées de garantir des fonctionnalités spécifiques : gérer les différents agents du système et surveiller l'interaction entre les agents afin de détecter la défaillance d'un agent, ou l'interruption d'une liaison de communication. Les sentinelles n'interviennent pas pour résoudre le problème. Par contre, elles peuvent intervenir pour exclure des agents défaillants, pour proposer une autre méthode de résolution, ou pour modifier les paramètres des agents.

La notion de sentinelle a été proposée pour la première fois par [Hägg, 1997]. Son

approche repose sur l'association d'une sentinelle à chaque agent afin de surveiller ses fonctionnalités et de le protéger de certains états indésirables. Étant un agent, la sentinelle interagit avec les autres agents, et de cette façon, elle peut construire des modèles d'autres agents. En analysant les messages échangés, les sentinelles peuvent construire et faire évaluer d'autres modèles d'agents. À partir de ces modèles, les sentinelles créent des points de reprise qui lui permettent de juger les agents à partir de leurs comportements et états internes. Cela permet de détecter la présence d'un agent défaillant, ou la présence d'un problème de cohérence entre les agents. Pour ce faire, les sentinelles doivent avoir accès aux connaissances des agents, et peuvent modifier leurs valeurs.

[Klein and Dellarocas, 1999] proposent un service de gestion des exceptions utilisant la notion de sentinelle. Les défaillances sont détectées selon une comparaison entre un comportement normal enregistré pour chaque agent, et le comportement réel de l'agent. Chaque agent fournit un modèle de son fonctionnement au service de gestion des exceptions. À partir de ces modèles, le service essaye de fournir une liste d'exceptions possibles auxquels des scripts sont associés. Les sentinelles sont générées pour faire fonctionner ces scripts, qui contrôlent le comportement des agents, et détecter les erreurs.

Toutes ces méthodes proposent d'introduire des agents supplémentaires, quelque soit pour reprendre le comportement d'un agent défaillant, ou pour contrôler le comportement des agents, ce qui peut augmenter la taille du problème modélisé. Cela limite l'application de ces méthodes de tolérance aux fautes à des problèmes possédant un nombre d'agents non statique.

2.5 Conclusion

Un système tolérant aux fautes est un système qui se comporte d'une manière bien définie en présence d'une faute. Pour concevoir un système tolérant aux fautes, il faut tout d'abord spécifier la classe de faute, puis concevoir les méthodes qui le protègent contre les fautes. Dans ce chapitre, nous avons présenté les différents concepts et bases de la tolérance aux fautes d'un système. Plusieurs comportements inhabituels des composants d'un système peuvent être considérés comme une défaillance tel qu'un délai de réponse dépassé ou précoce, un service non délivré, etc. Les méthodes de détection de ces

défaillances ont engendré l'apparition de 2 modèles d'interaction entre les détecteurs et les composants du système : un modèle *pull*, et un modèle *push*. Pour remédier aux défaillances détectées au sein d'un système distribué, la plupart des méthodes de tolérance aux fautes ont été basées sur le concept de *la réplication*. Elle peut être au niveau des composants du système, des tâches d'un composant, ou du moment d'exécution d'une tâche. La réplication des composants ainsi que d'autres méthodes de tolérance aux fautes ont été adaptés pour traiter les défaillances au sein des SMAs. Ces approches ont été distinguées selon les techniques utilisées telles que : la réplication des agents, la notion de groupe d'agents, et l'utilisation des sentinelles comme agents particuliers.

A partir des méthodes présentées dans la littérature, nous avons remarqué qu'aucune de ces méthodes n'a été adapté pour maintenir la robustesse d'un CSP distribué, ou pour résoudre un CSP distribué en présence d'une défaillance. En effet, le maintien de la robustesse des CSPs dans la littérature concerne l'obtention des résultats dans le cas des CSPs dynamiques (modification au niveau des contraintes ou des variables du CSP) mais sans faire intervenir les SMAs. Dans le chapitre suivant, nous introduisons une méthode basée sur certaines notions déjà existantes pour résoudre un CSP distribué en présence d'un agent défaillant.

Chapitre 3

Réplication des CSPs locaux pour résoudre un DisCSP

Sommaire

3.1 Introduction

La robustesse dans les CSPs distribués peuvent concerner les CSPs ou les SMAs. Dans le cas des CSPs, nous parlons des approches proposées de tolérance aux fautes dans les CSPs dynamiques (chapitre 2). Dans le cas des SMAs, les approches proposées n'ont pas été adaptées pour résoudre des DisCSPs en présence d'agents défectueux.

Dans ce chapitre, nous traitons le cas de la défaillance des agents lors de la résolution d'un DisCSP. L'approche proposée est basée sur le principe de réplication des CSPs locaux : chaque CSP est copié au sein d'un agent différent du sien. Si un agent est défaillant, son CSP sera pris en charge par cet agent. Nous adaptons ce principe à l'algorithme Multi-ABT. La détection d'une défaillance est assurée par un échange de messages différents de ceux utilisés par Multi-ABT. Le but de ces messages est d'identifier l'agent défaillant et d'en informer les autres agents. La prise en charge du CSP de l'agent défaillant entraine la fusion des CSPs locaux.

Dans la première section (3.2), nous citons les différentes hypothèses et les structures de données utilisées lors du traitement de la défaillance. Dans la section d'après (3.3), nous définissons les éléments caractérisant notre approche. Dans la dernière section (3.4), nous expliquons le fonctionnement général de notre approche en présence d'un agent défaillant via un exemple d'illustration.

3.2 Généralités

Dans cette section, nous citons les hypothèses et les notations utilisées (3.2.1), ainsi que les structures de données utilisées durant le traitement de la défaillance de l'agent (3.2.2).

3.2.1 Hypothèses

Nous proposons une approche de résolution d'un DisCSP même en présence d'agents défaillants. La défaillance d'un agent se situe dans le cas de l'arrêt du service (section 2.2.2). Le principe de l'approche proposée est de déléguer le CSP local de l'agent défaillant à un autre agent.

Définition 18 (Agent délégué)**.** *L'agent délégué est l'agent qui prendra en charge la résolution du CSP local de l'agent défaillant.*

Hypothèse 1. Le DisCSP à résoudre est connexe.

Hypothèse 2. Si un agent est défaillant, il ne peut pas reprendre son activité. Il ne peut plus envoyer de messages, ni résoudre son CSP local.

Hypothèse 3. Si un agent est considéré défaillant par un agent, il sera reconnu comme défaillant par tous les autres agents.

L'approche proposée est basée sur la notion de la réplication des CSPs locaux.

Hypothèse 4. Lors de la réplication d'un CSP local, tous ses composants sont répliqués : ses variables ainsi que leurs domaines, et ses contraintes intra et inter-agents.

Le processus de la réplication est effectuée par un agent particulier, appelé *Agent Dispatcher*.

Hypothèse 5. L'*Agent Dispatcher* peut communiquer avec tous les agents du système, et ne risque pas d'être défectueux.

En plus des messages échangés entre les agents lors de l'exécution de Multi-ABT (les messages $\langle \mathrm{OK?} \rangle$ et $\langle \mathrm{BT} \rangle$), nous proposons quatre messages supplémentaires afin de détecter la défaillance d'un agent dans le système :

- Le message $\langle \mathrm{CHECK}(A_i) \rangle$: Ce message est envoyé par un agent A_i à son voisin A_j. [Tanenbaum and Steen, 2006] ont introduit ce type de message au sein des systèmes distribués : si l'agent A_i ne reçoit aucun message de résolution de la part de son voisin A_j pendant un intervalle de temps, il lui envoie ce message afin de vérifier son état d'activité. Si l'agent A_i ne reçoit pas de réponse à ce message, il considère l'agent A_j comme agent défaillant. Ce message est plus prioritaire que les messages de résolution échangés : si un agent reçoit ce message, il suspend son comportement pour en répondre. Ce type de message appartient au modèle « pull » défini précédemment dans la section 2.3.1.

- Le message $\langle \mathrm{ACTIVE}(A_i) \rangle$: Ce message est envoyé par l'agent A_j à l'agent A_i comme une réponse au message $\langle \mathrm{CHECK} \rangle$ reçu de la part de l'agent A_i. Il ne contient aucune information. En recevant ce message, l'agent A_i considère l'émetteur du message comme agent actif (non défaillant).

- Le message $\langle \mathrm{isFailed}(A_j, \Gamma) \rangle$: Ce message est envoyé par l'agent A_i à ses voisins après avoir détecté la défaillance d'un agent A_j. Afin de garantir que les voisins de l'agent A_j soient informés de la défaillance, les voisins de l'agent A_i transmettent ce message à leurs voisins, et ainsi de suite. L'agent qui transmet ce message doit éliminer ses voisins appartenant à la liste Γ de la liste des destinataires. La transmission du message s'arrête quand la liste Γ contient tous les voisins de l'agent émetteur.

- Le message $\langle \mathrm{NewCSP}(A_j, \Gamma(A_j)) \rangle$: Ce message est envoyé par l'agent A_i aux voisins de l'agent A_j ($\Gamma(A_j)$) dans le cas où c'est l'agent A_i qui prend en charge le CSP de l'agent A_j. Le but est de les informer qu'il remplace l'agent A_j afin qu'ils mettent à jour leurs listes de voisins.

Un autre message est envoyé de la part de l'*Agent Dispatcher* à tous les agents :

- Le message $\langle \mathrm{Affectation}(A_i, Affect) \rangle$: Ce message contient la liste des copies des CSPs locaux ($Affect$) affectée à chaque agent A_i. Il est envoyé aux agents après avoir copié tous les CSPs locaux.

3.2.2 Structures de données

Afin de résoudre un DisCSP en présence d'agents défaillants, les agents du système possèdent des structures de données nécessaires pour l'exécution de l'algorithme Multi-ABT (Nogood et $\mathrm{AgentView}$), ainsi que de nouvelles structures de données :

- $Dupp(A_i)$: L'ensemble des copies des CSPs locaux reçu par un agent A_i de la part de l'*Agent Dispatcher*.

- $Failed$: L'ensemble d'agents défaillants. À chaque fois qu'un agent A_i reçoit une information de défaillance d'un agent A_j, il enregistre l'identifiant de l'agent A_j

dans $Failed$. De cette façon, si un agent reçoit la même information de défaillance de la part de deux agents différents, il considère le 2^{eme} message comme obsolète.

Afin de copier chaque CSP local chez un autre agent et envoyer aux agents les copies, l'*Agent Dispatcher* doit posséder les structures de données suivantes :

- $CSPs$: Une liste composée d'un ensemble de triplets sous la forme $\langle Agent, CSP, CSPsVoisins\rangle$. Chaque triplet comporte un agent $Agent$, son CSP local CSP, et les CSPs des voisins de l'agent $Agent$ ($CSPsVoisins$).
- $Affect$: Une liste composée d'un ensemble de couples sous la forme $(A_i, Copies(A_i))$. Chaque couple contient l'agent A_i, et les copies des CSPs $Copies(A_i)$ que l'*Agent Dispatcher* lui a affecté.

Les structures de données et les messages échangés évoluent lors du traitement de la défaillance. Nous détaillerons ce traitement dans les prochaines sections.

3.3 Éléments caractérisant notre approche

Pour trouver une solution globale à un DisCSP en présence d'un agent défaillant, il faut résoudre le CSP local de l'agent défaillant. Pour ce faire, nous proposons une approche basée sur la réplication des CSPs locaux : elle détecte l'agent défaillant, et affecte une copie de son CSP local à un autre agent. Dans cette section, nous présentons les trois étapes de ce processus : (i) le processus de la réplication, (ii) la détection de la défaillance, et (iii) le processus de fusion des CSPs locaux. Dans la section (3.3.1), nous définissons l'*Agent Dispatcher*, comme étant un agent particulier. Ensuite, nous définissons les algorithmes de réplication des CSPs locaux (3.3.2), de la détection de l'agent défaillant (3.3.3), et de la fusion des CSPs locaux (3.3.4).

3.3.1 L'agent Dispatcher

L'*Agent Dispatcher* est un agent particulier chargé de répliquer les CSPs locaux au sein des agents. Considérons par exemple un DisCSP composé de 3 agents A_0, A_1 et A_2, tel que $A_0 \in \Gamma(A_1)$, $A_1 \in \Gamma(A_0, A_2)$, et $A_2 \in$

$\Gamma(A_1)$. La liste $CSPs$ de l'*Agent Dispatcher* contient les éléments suivants : $CSPs = \{(A_0, CSP_0, \{CSP_1\}), (A_1, CSP_1, \{CSP_0, CSP_2\}), (A_2, CSP_2, \{CSP_1\})\}$.

Supposons que le processus de la réplication, détaillé dans la prochaine section et exécuté par l'*Agent Dispatcher*, affecte les copies des CSPs de cette façon : l'agent A_0 aura la copie du CSP_1, et l'agent A_1 aura les copies des CSPs CSP_0 et CSP_2. De cette façon, la liste des affectations contient les éléments suivants : $Affect = \{(A_0, \{CSP_1\}), (A_1, \{CSP_0, CSP_2\}), (A_2, \{\})\}$.

L'agent Dispatcher n'intervient pas pour résoudre un DisCSP ; il intervient avant de commencer la résolution d'un problème, et après la détection d'une défaillance. Donc, son absence n'influence pas le fonctionnement de l'algorithme de résolution. Les CSPs locaux sont répliqués selon un algorithme de réplication présenté dans la prochaine section.

3.3.2 Réplication des CSPs

Dans cette section, nous présentons les détails ainsi que le pseudo-code du processus de réplication exécuté par l'*Agent Dispatcher*.

Ce processus vise à répliquer le CSP local de chaque agent chez un de ses voisins. Ce processus assure qu'un agent peut avoir plusieurs copies de différents CSPs, mais un CSP local n'est répliqué qu'une seule fois chez un et un seul agent. De cette façon, le CSP local de l'agent défaillant n'est pris en charge que par un seul autre agent, et quelque soit l'agent défaillant, il y aura un seul agent délégué appartenant au même système.

Afin de répliquer les CSPs, l'*Agent Dispatcher* utilise sa liste $CSPs$ comme entrée de son processus, pour avoir la liste $Affect$ comme sortie (Algorithme 1). Tout d'abord, il commence par la création d'une liste vide $Copies(A_i)$ pour chaque agent A_i (Lignes 1 et 2). Chaque liste $Copies(A_i)$ sera envoyée à l'agent A_i qui lui correspond, grâce au message $\langle Duplication(A_i, Copies(A_i))\rangle$, à la fin du processus de la réplication. Ensuite, pour chaque voisin A_j d'un agent A_i (Ligne 4), l'*Agent Dispatcher* vérifie si son CSP (CSP_j) est déjà répliqué chez un autre agent (Ligne 5). Si ce n'est pas le cas, il affecte une copie de CSP_j à l'agent A_i en l'ajoutant à $Copies(A_i)$ (Ligne 6). Si les CSPs de tous les voisins

Algorithme 1 : Replication ($CSPs$)

Entrées : $CSPs$: liste des CSPs locaux

Sorties : $Affect$: liste des copies des CSPs

```
1 pour chaque A_i, i ∈ {1, .., m} faire
2     Copies(A_i) ← { } ;
3     répéter
4         pour chaque A_j ∈ Γ(A_i) faire
5             si CSP_{A_j} ∈ CSPs alors
6                 Copies(A_i) ← Copies(A_i) ∪ {CSP_{A_j}} ;
7                 CSPs ← CSPs − {CSP_{A_j}} ;
      jusqu'à CSPs = { };
   retourner Affect ;
```

de l'agent A_i sont répliqués chez des agents autre que l'agent A_i, il n'aura aucune copie de CSP (sa liste $Dupp(A_i)$ reste vide durant la résolution du DisCSP). Après avoir répliqué CSP_j chez l'agent A_i, l'*Agent Dispatcher* le supprime de sa liste $CSPs$ contenant les CSPs non encore répliqués (Ligne 7). Ce processus est répété jusqu'à répliquer tous les CSPs locaux (obtention d'une liste $CSPs$ vide).

Ce processus assure un équilibre de charge entre les agents. En effet, les CSPs locaux ne risquent pas d'être répliqués au sein d'un seul agent. La répartition des CSPs locaux est proportionnelle entre les agents. Cela est démontré lors des expérimentations dans la section 4.3. Un exemple d'un CSP est illustré à la figure 3.1. L'*Agent Dispatcher* a comme en-

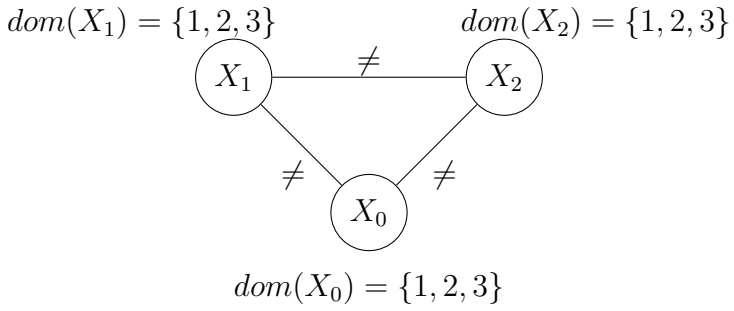


FIGURE 3.1 – Exemple d'un CSP

trée la liste $CSPs = \{(A_0, CSP_0, \{CSP_1, CSP_2\}), (A_1, CSP_1, \{CSP_0, CSP_2\}), (A_2, CSP_2, \{CSP_1, CSP_0\})\}$. La liste des voisins des agents A_0, A_1 et A_2 est $\{A_1, A_2\}$, $\{A_0, A_2\}$ et $\{A_1, A_0\}$ respectivement. L'*Agent Dispatcher* commence par affecter à l'agent A_0 une copie du CSP de son premier voisin alors $Copies(A_0) = \{CSP_1\}$, et $CSPs = \{(A_0, CSP_0, \{CSP_2\}), (A_1, CSP_1, \{CSP_0, CSP_2\}), (A_2, CSP_2, \{CSP_0\})\}$. Ensuite, il passe à l'agent A_1, il lui affecte une copie du CSP de son voisin A_0 : $Copies(A_1) = \{CSP_0\}$ et $CSPs = \{(A_0, CSP_0, \{CSP_2\}), (A_1, CSP_1, \{CSP_2\}), (A_2, CSP_2, \{\})\}$. En passant à l'agent d'après (A_2), l'*Agent Dispatcher* vérifie si une copie du CSP de son premier voisin (CSP_0) est déjà affectée à un autre agent. Dans ce cas, puisque $\{CSP_0\} \in Copies(A_1)$, il passe au CSP du voisin d'après (CSP_1), qui est répliqué déjà chez l'agent A_0 : $\{CSP_1\} \in Copies(A_0)$.

Dans ce cas, l'*Agent Dispatcher* revient au premier agent de sa liste (A_0). Puisque le CSP de son premier voisin est déjà répliqué, il passe au CSP de l'agent A_2. Ce dernier n'est répliqué chez aucun agent, donc il le réplique chez l'agent A_0 : $Copies(A_0) = \{CSP_1, CSP_2\}$, et la liste des CSPs des voisins dans $CSPs$ devient vide ($CSPs = \{(A_0, CSP_0, \{\}), (A_1, CSP_1, \{\}), (A_2, CSP_2, \{\})\}$). A la fin de ce processus, L'*Agent Dispatcher* a comme résultat final : $Affect = \{(A_0, \{\{X_1\}, \{X_2\}\}); (A_1, \{X_0\}); (A_2, \{\})\}$.

Dans ce qui suit, nous présentons la méthode de détection d'une défaillance d'un agent en utilisant les messages mentionnés précédemment (section 3.2.1).

3.3.3 Détection d'un agent défaillant

Dans cette section, nous présentons comment détecter un agent défaillant lors de la résolution d'un DisCSP. Cette détection est effectuée grâce aux message $\langle$CHECK$\rangle$ et $\langle$ACTIVE$\rangle$.

Lors de la résolution du DisCSP, chaque agent $self$ attribue à chacun de ses voisins A_j deux délais de réponse $Delai(A_j)$ et $DelaiActif(A_j)$.

Définition 19 ($Delai(A_j)$)**.** *Temps écoulé depuis la réception du dernier message de la part d'un agent A_j.*

Définition 20 ($DelaiActif$)**.** *L'intervalle de temps pendant lequel un agent doit répondre à un message* $\langle$ACTIVE$\rangle$.

L'agent défaillant est détecté selon l'algorithme 2. Si un agent tombe en panne, c'est un de ses voisins qui détectera cette défaillance. Donc, un agent prend sa liste des voisins comme entrée. Le résultat obtenu à la fin de l'algorithme est l'identifiant de l'agent défaillant.

Algorithme 2 : Détection d'une défaillance

Entrées : Γ : liste des voisins

Output : A_j : l'agent défaillant

1 **si** $\exists A_j \in \Gamma(self)/Delai(A_j) > 10$ **alors**

2 $\quad self$ envoie $\langle \text{CHECK} \rangle$ à A_j;

3 $\quad$ **si** $DelaiActif > 10$ **alors**

4 $\quad\quad failed(A_j) \leftarrow True$;

retourner A_j;

Supposons qu'un agent $self$ détecte la panne (c'est lui qui exécute l'algorithme de détection). Cet agent commence par transmettre un message $\langle \text{CHECK} \rangle$ à un agent A_j s'il ne reçoit aucun message de sa part pendant 10 secondes ($Delai(A_j) > 10sec$) (Lignes 1 et 2). En effet, la valeur de 10 secondes est suffisante pour un agent pour trouver une solution locale à son CSP et la transmettre à ses voisins. L'agent $self$ reste en attente d'une réponse à son message $\langle \text{CHECK} \rangle$ pour confirmer l'activité de son voisin A_j. Si ce temps d'attente dépasse 10 secondes ($DelaiActif > 10$ secondes) et il n'a pas encore reçu le message $\langle \text{ACTIVE} \rangle$ (Ligne 3), il considère que l'agent A_j est défaillant. La valeur de 10 secondes permet à un agent d'arrêter son activité et répondre au message $\langle \text{CHECK} \rangle$. La fixation de la valeur de $DelaiActif$ est justifiée dans la partie des expérimentations (section 4.4.2). Supposons que l'agent $self$ reçoit un message de résolution ($\langle \text{OK?} \rangle$ ou $\langle \text{BT} \rangle$) de la part de l'agent A_j avant de recevoir un message $\langle \text{ACTIVE} \rangle$. Dans ce cas, l'agent $self$ considère l'agent A_j comme agent actif même s'il n'a pas encore reçu le message $\langle \text{ACTIVE} \rangle$

Après la détection de la défaillance, l'agent qui possède une copie du CSP de l'agent défaillant fusionne cette copie avec son propre CSP. Dans ce qui suit, nous présentons le processus de fusion des CSPs locaux.

3.3.4 Fusion des CSPs

Le principe de la fusion est de créer un nouveau CSP local à partir de plusieurs CSPs locaux. Pour avoir un seul CSP local, l'agent fusionne l'ensemble des variables ainsi que l'ensemble des contraintes. Le but de cette fusion est de garder la forme initiale du DisCSP globale, et de trouver une solution au CSP de l'agent défaillant. L'agent exécutant ce processus aura comme résultat un nouveau CSP obtenu à partir de son propre CSP local, et de la copie du CSP de l'agent défaillant (Algorithme 3).

Algorithme 3 : Fusion des CSPs (CSP_{A_j})

Entrées : CSP_{A_j} : CSP de l'agent défaillant

Output : CSP_{A_i} : nouveau CSP local de A_i

1 $Variables(A_i) \leftarrow Variables(A_i) \cup Variables(A_j)$;

2 $IntraC(A_i) \leftarrow IntraC(A_i) \cup IntraC(A_j) \cup InterC(A_i, A_j)$;

3 $\forall C \in InterC(A_j, A_k), A_k \neq A_i, InterC(A_i, A_k) \leftarrow InterC(A_i, A_k) \cup \{C\}$;

retourner CSP_{A_i} ;

La fusion des CSPs commence par fusionner les variables (Ligne 1) : l'agent A_i ajoute les variables et leurs domaines, de la copie du CSP de l'agent défaillant A_j, à l'ensemble de ses propres variables. La fusion des contraintes se fait de la manière suivante :

- L'ensemble des contraintes intra-agents du nouveau CSP est composé de l'ensemble des contraintes intra-agents de l'agent A_i, et de celles de l'agent défaillant A_j. Les contraintes inter-agents qui relient les agents A_i et A_j sont ajoutées aussi à l'ensemble des contraintes intra-agents du nouveau CSP de l'agent A_i (Ligne 2).
- Les contraintes inter-agents de l'agent défaillant A_j qui le relient aux agents autre que l'agent A_i, sont ajoutées à l'ensemble des contraintes inter-agents de l'agent A_i (Ligne 3).

Cette méthode de fusion des contraintes permet de conserver la modélisation intiale du CSP après la fusion des CSPs. Reprenons par exemple le CSP présenté dans la figure 3.1. On a supposé dans la section 3.3.2 que le résultat de la réplication est :
$Affect = \{(A_0, \{\{X_1\}, \{X_2\}\}); (A_1, \{X_0\}); (A_2, \{\})\}$.

Maintenant, supposons que l'agent A_1 est défaillant. L'agent délégué est l'agent A_0, puisque c'est lui qui possède une copie de son CSP (X_1). Avant la fusion, l'agent A_0 possède le CSP suivant :

- $Variables(A_0) = \{X_0\}$
- $intraC(A_0) = \{\}$
- $interC(A_0, A_1) = \{C_{01}\}$: la contrainte qui relie les variables X_0 et X_1.
- $interC(A_0, A_2) = \{C_{02}\}$: la contrainte qui relie les variables X_0 et X_2.

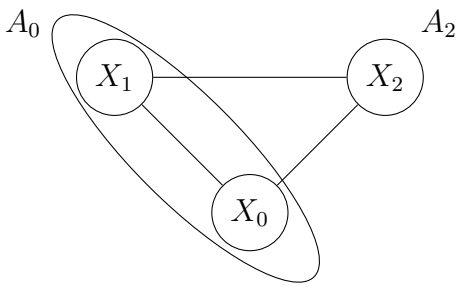


FIGURE 3.2 – Fusion de deux CSPs

Après la fusion, le nouveau CSP de l'agent A_0 est composé des éléments suivants (Figure 3.2) :

- $Variables(A_0) = \{X_0, X_1\}$
- $intraC(A_0) = \{C_{01}\}$
- $interC(A_0, A_2) = \{C_{02}, C_{12}\}$: les contraintes qui relient les variables X_0 et X_1 à la variable X_2.

Dans la section qui suit, nous présentons le fonctionnement général de notre approche en utilisant les algorithmes détaillés précédemment.

3.4 Traitement général d'une défaillance

Afin de résoudre un DisCSP en présence d'un agent défaillant, les agents sont appelés à exécuter l'ensemble des algorithmes présentés dans la section précédente dans un ordre

bien défini. Dans cette section, nous présentons le comportement général des agents afin de traiter une défaillance (3.4.1), une illustration de notre approche via son application à un exemple de coloration de graphe (3.4.2), et les différentes propriétés de notre approche (3.4.3).

3.4.1 Processus Général

Dans cette section, nous présentons le fonctionnement détaillé des agents afin de traiter une défaillance.

Au début de la résolution, chaque agent envoie son propre CSP local et sa liste des voisins à l'*Agent Dispatcher* pour les répliquer. Le processus général commence par exécuter l'algorithme $Duplication(CSPs)$ (Algorithme 1) par l'*Agent Dispatcher*. Ensuite, il envoie à chaque agent A_i la copie du CSP correspondante, grâce au message $\langle Affectation \rangle$. En recevant ce message, chaque agent A_i enregistre la liste des copies reçue dans sa liste $Dupp(A_i)$.

La figure 3.3 présente le comportement d'un agent A_i durant la résolution d'un DisCSP en présence d'un agent défaillant. Le processus possède le CSP global à résoudre comme entrée. Les agents commencent à résoudre le CSP en proposant chacun une solution locale et l'envoyant à ses accointances inférieures. En recevant la proposition, chaque agent cherche une solution locale cohérente à la proposition reçue. L'échange des solutions locales est effectué grâce aux messages définis par Multi-ABT. Durant la résolution, si un agent A_i ne reçoit aucun message de la part d'un de ses voisins A_j pendant un intervalle de temps $Delai(A_j)$, il lui envoie un message $\langle \mathrm{CHECK} \rangle$. Si l'agent A_j est actif, il répond en envoyant un message $\langle \mathrm{ACTIVE} \rangle$ à l'agent A_i. En recevant le message de confirmation d'activité $\langle \mathrm{ACTIVE} \rangle$, l'agent A_i considère l'agent A_j comme agent actif, et met à jour la valeur de $Delai(A_j)$. Si aucun message n'est reçu (ni $\langle \mathrm{ACTIVE} \rangle$, ni aucun autre message), l'agent A_i considère l'agent A_j comme défaillant et en informe ses voisins grâce au message $\langle \mathrm{isFailed}(A_j, voisins) \rangle$. Les agents qui ont reçu le message $\langle \mathrm{isFailed} \rangle$ appellent l'écouteur **recevoirInfo** (Écouteur 1). Si les agents reçoivent l'information de la défaillance d'un même agent plusieurs fois (cas où la défaillance d'un agent a été détectée par plus qu'un agent), ils la considèrent comme information obsolète, et ignorent le message. Sinon, ils diffusent l'information en envoyant le message $\langle \mathrm{isFailed}(A_j, \Gamma(self)) \rangle$ à

Ecouteur 1 : recevoirInfo $\langle \text{isFailed}(A_j, \Gamma) \rangle$

1 Envoyer $\langle \text{isFailed}(A_j, \Gamma(self)) \rangle, \Gamma(self) \not\subset \Gamma$;
2 **si** $CSP_{A_j} \in Dupp(self)$ **alors**
3 Fusion des CSP (CSP_{A_j});
4 Mettre à jour $\Gamma(self)$;
5 Envoyer $\langle \text{NewCSP}(A_j, \Gamma(A_j)) \rangle$;

leurs voisins (Ligne 1). La liste des voisins Γ contient l'ensemble des voisins qui ont déjà reçu l'information de défaillance. Donc l'agent $self$ envoie le message $\langle \text{isFailed} \rangle$ à ses voisins qui n'appartiennent pas à la liste Γ. Ensuite, chaque agent, ainsi que l'agent qui a détecté la défaillance, vérifie si sa liste des copies des CSPs locaux $Dupp(A)$ contient une réplication du CSP de l'agent défaillant (Ligne 2). L'agent qui trouve la copie de CSP_j exécute l'algorithme de fusion (Ligne 3) et met à jour l'ensemble de ses voisins (Ligne 4). Après la mise à jour des voisins, l'agent délégué informe les voisins de l'agent défaillant qu'il prend en charge la résolution de son CSP local. Cette information est transmise grâce au message $\langle \text{NewCSP} \rangle$ (Ligne 5). En recevant ce message, les agents remplacent l'agent défaillant A_j par l'agent émetteur du message $self$, et mettent à jour l'ordre de priorité des variables.

Après la mise à jour des voisins, les agents reprennent leur activité en cherchant une solution au CSP global. Si cette solution est trouvée, le processus fournie cette solution comme résultat, sinon l'algorithme s'arrête.

Mise à jour des voisins

La mise à jour des voisins inclut la mise à jour de l'ordre de priorité des agents et des variables. Après la fusion des CSPs, l'agent ajoute les voisins de l'agent défaillant à sa liste de ses voisins, ainsi que leurs priorités. Les priorités des agents sont mis à jour selon le nouveau nombre de contraintes inter-agents de l'agent délégué : selon Multi-ABT, l'agent ayant le plus grand nombre de contraintes inter-agents possède la plus haute priorité.

Au niveau de la mise à jour des priorités des variables, pour chaque nouvelle variable

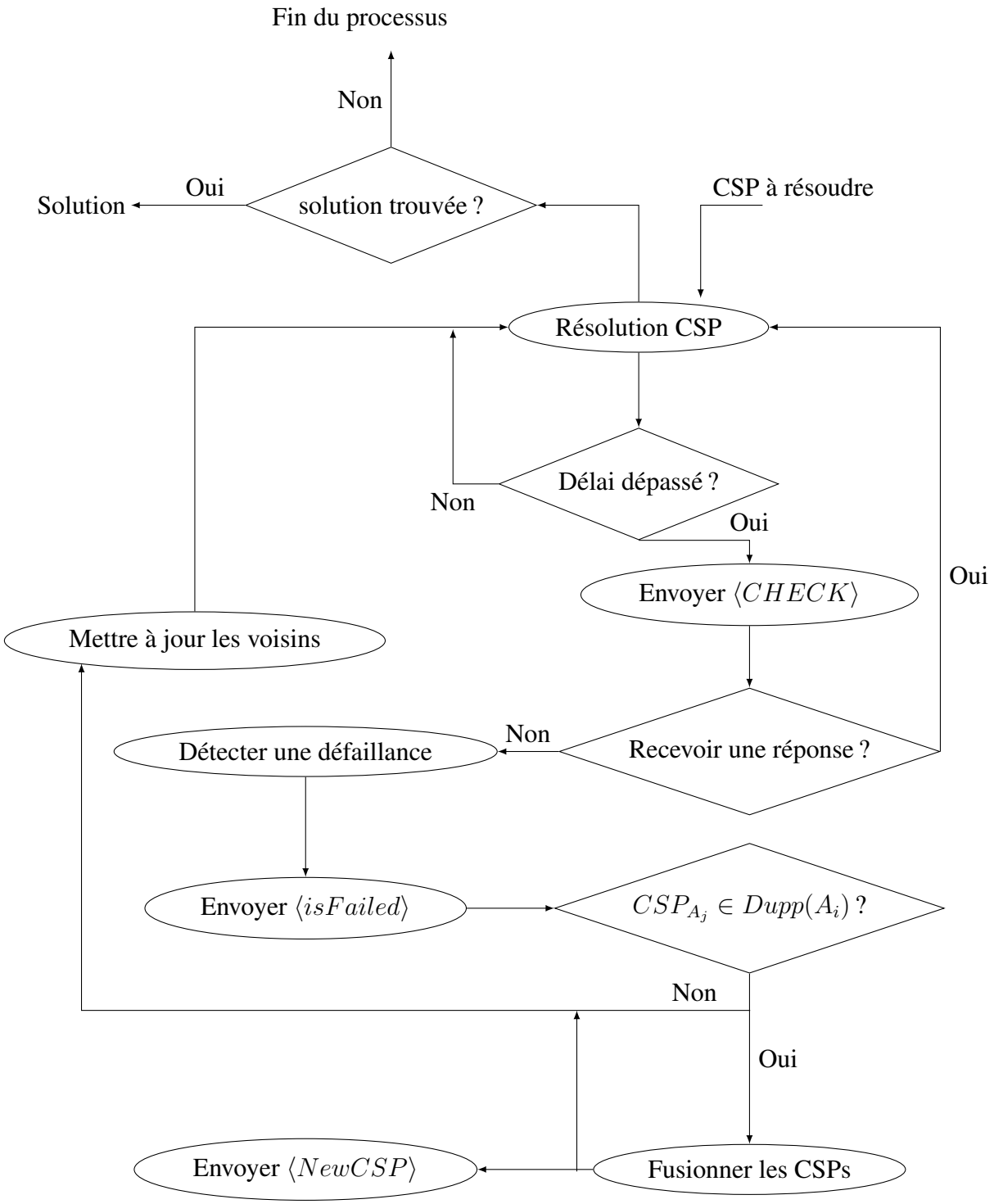


FIGURE 3.3 – Comportement d'un agent A_i

X_j appartenant au CSP local obtenu après la fusion, la modification de la priorité de X_j par rapport à une variable X_i se fait comme suit :

- Si X_i et X_j appartiennent au même agent, la priorité est affectée selon l'ordre lexicographique des variables.
- Si les variables appartiennent à deux agents différents A_i et A_k (A_i est l'agent qui a fusionné deux CSPs), la priorité des variables change selon leurs ordres de priorité : si $A_k \in \Gamma^-(A_i)$, alors $X_i \succ X_j$ tel que $X_j \in vars(A_k)$. Aussi, si $A_k \in \Gamma^+(A_i)$, alors $X_j \succ X_i$.

3.4.2 Illustration du processus de traitement de défaillance

Soit le DisCSP multi-variables présenté par la figure 3.4. Rappelons qu'il s'agit d'un problème de coloration de graphe composé de 3 agents avec un ordre de priorité $A_0 \succ A_1 \succ A_2$, et ayant chacun 3 variables. Chaque variable peut avoir les couleurs ○ ou ●.

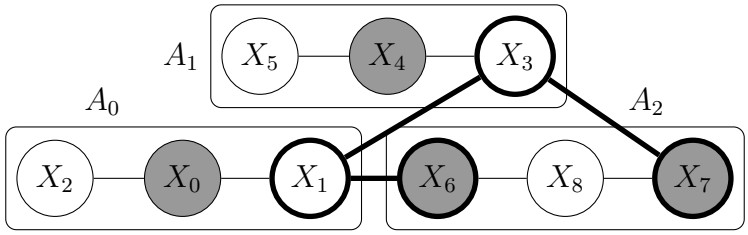


FIGURE 3.4 – Exemple de CSP Distribué [Mandiau et al., 2014]

Notre approche commence par exécuter l'algorithme $Duplication(CSPs)$ (Algorithme 1). La liste des CSPs locaux de L'*Agent Dispatcher* est composée de l'ensemble des triplets suivant :

- $(A_0, CSP_0, \{CSP_2, CSP_1\})$
- $(A_1, CSP_1, \{CSP_2, CSP_0\})$
- $(A_2, CSP_2, \{CSP_1, CSP_0\})$

Dans cet exemple, nous utilisons la notation CSP'_i pour la copie d'un CSP_i afin de différencier les CSPs originaux de leurs copies. L'*Agent Dispatcher* choisit le CSP du premier voisin de A_0 (CSP_2), et affecte sa copie à l'agent A_0. Ensuite, il passe à

l'agent suivant A_1 et vérifie si le CSP de son premier voisin (CSP_2) est déjà copié chez un autre agent. Puisque CSP_2 est déjà copié chez l'agent A_0, l'*Agent Dispatcher* passe au CSP du voisin d'après (CSP_0), et le copie chez l'agent A_1. Finalement, il passe à l'agent A_2, et lui affecte une copie du CSP_1 puisqu'il n'est pas encore copié. A la fin de ce processus, l'*Agent Dispatcher* envoie chaque affectation à l'agent qui lui correspond. Après la réception des affectations, les agents auront les listes des CSP_{add} suivants : $Dupp(A_0) = \{CSP'_2\}$, $Dupp(A_1) = \{CSP'_0\}$ et $Dupp(A_2) = \{CSP'_1\}$ (Figure 3.5). Les CSPs locaux CSP'_0, CSP'_1 et CSP'_2 correspondent aux copies des CSPs CSP_0, CSP_1 et CSP_2 respectivement.

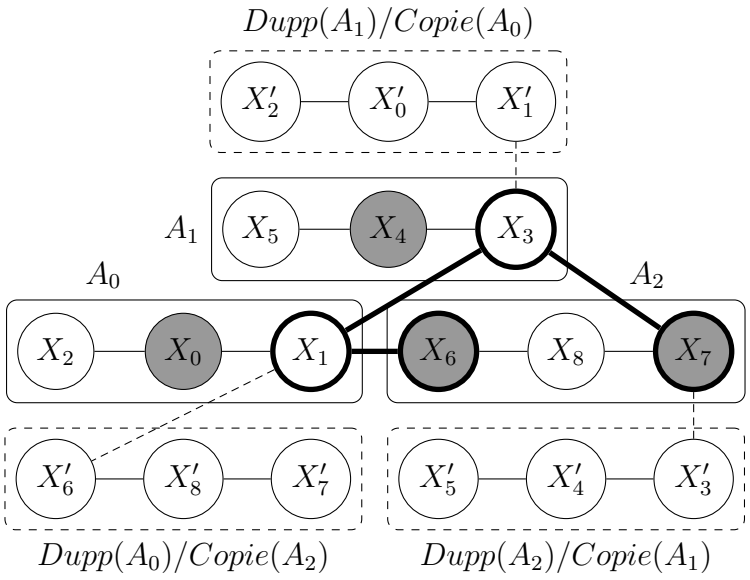


FIGURE 3.5 – Résultat du processus de réplication

Après la création des listes $Dupp(A_i)$, l'algorithme de résolution Multi-ABT est exécuté. Tout d'abord, chaque agent propose une solution locale à son propre CSP : A_0, A_1 et A_2 proposent les solutions ($X_0 = \bullet, X_1 = \circ, X_2 = \circ$), ($X_3 = \circ, X_4 = \bullet, X_5 = \circ$) et ($X_6 = \bullet, X_7 = \bullet, X_8 = \circ$) respectivement. Ensuite, chacun envoie sa solution à ses accointances inférieures (Figure 3.6) : L'agent A_0 envoie sa solution aux agents $\{A_1, A_2\}$ (Messages M_1 et M_2), et l'agent A_1 envoie sa solution à l'agent A_2 (Message M_3). En recevant le message M_1, l'agent A_1 modifie sa solution pour en proposer une autre consistante avec le contenu du message M_1, et l'envoie à l'agent A_2 (Message M_4). En recevant les messages M_2 et M_3, l'agent A_2 ne change pas de solution puisqu'elle est consistante

avec le contenu des messages reçus. En recevant le message M_4, l'agent A_2 ne trouve pas de solution consistante à son contenu, alors il envoie un message de backtracking $\langle \mathrm{BT} \rangle$ à l'agent A_1 (Message M_5), qui va le transmettre à l'agent A_0 (Message M_6). En recevant ce dernier, l'agent A_0 modifie sa solution à ($X_0 = \circ, X_1 = \bullet, X_2 = \bullet$), et l'envoie aux agents A_1 et A_2. Les agents continuent à échanger leurs solutions locales jusqu'à ce que l'agent A_0 se trouve dans une situation où il n'a plus de solutions locales à proposer.

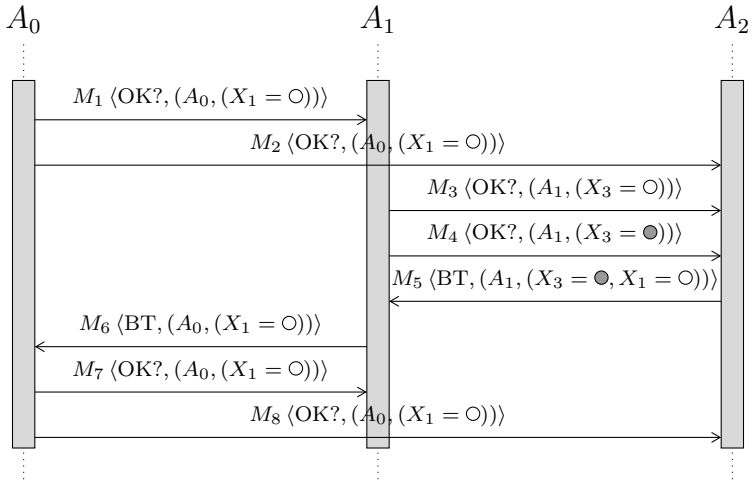


FIGURE 3.6 – Messages échangés entre les agents pour résoudre le DisCSP (Figure 3.4)

Maintenant, afin d'appliquer notre approche, supposons que l'agent A_0 est tombé en panne juste après avoir envoyé le message M_2. Et supposons que l'agent A_1 détecte cette panne, puisque l'agent A_2 est occupé par le traitement des différents messages reçus (Figure 3.7). En envoyant le message M_6, l'agent A_1 attend une réponse de la part de l'agent A_0 (Message M_7 de la figure 3.6). Après un intervalle de temps $Delai(A_0)$, l'agent A_1 ne reçoit aucun message de la part de l'agent A_0, alors il lui envoie un message $\langle \mathrm{CHECK}(A_1) \rangle$ (Message M_7). Après un intervalle de temps $DelaiActif(A_0)$, l'agent A_1 ne reçoit toujours pas de réponse à son message, alors il considère l'agent A_0 comme défaillant et en informe l'agent A_2 grâce au message $\langle \mathrm{isFailed}(A_0, A_2) \rangle$ (Message M_8). Ensuite, les agents A_1 et A_2 vérifient le contenu de leurs listes $Dupp(A_i)$.

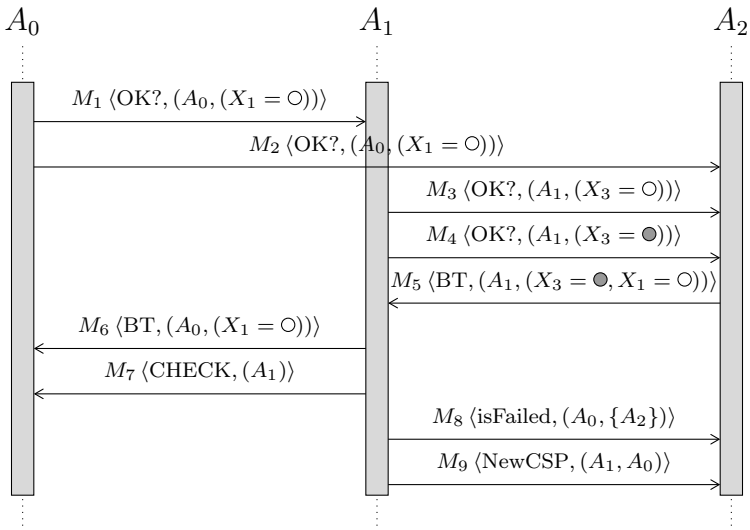


FIGURE 3.7 – Détection de la défaillance de A_0

Durant le processus de réplication, la copie de CSP_0 a été affecté à l'agent A_1 ; Il prend donc en charge la résolution de CSP_0. Il fusionne son propre CSP avec CSP'_0 en exécutant l'algorithme de fusion des CSPs (Algorithme 3). A la fin du processus de fusion (Figure 3.8), l'agent A_1 aura un nouveau CSP composé de :

- $\mathscr{X}$ = $\{X'_0, X'_1, X'_2, X_3, X_4, X_5\}$ tel que $\{X'_0, X'_1, X'_2\}$ correspondent aux variables $\{X_0, X_1, X_2\}$
- $IntraC(A_1)$ = $\{C'_{01}, C'_{02}, C'_{13}, C_{34}, C_{45}\}$ tel que $\{C'_{01}, C'_{02}, C'_{13}\}$ correspondent à $\{C_{01}, C_{02}, C_{13}\}$
- $InterC(A_1, A_2)$ = $\{C'_{16}, C_{37}\}$ tel que C'_{16} correspond à C_{16}

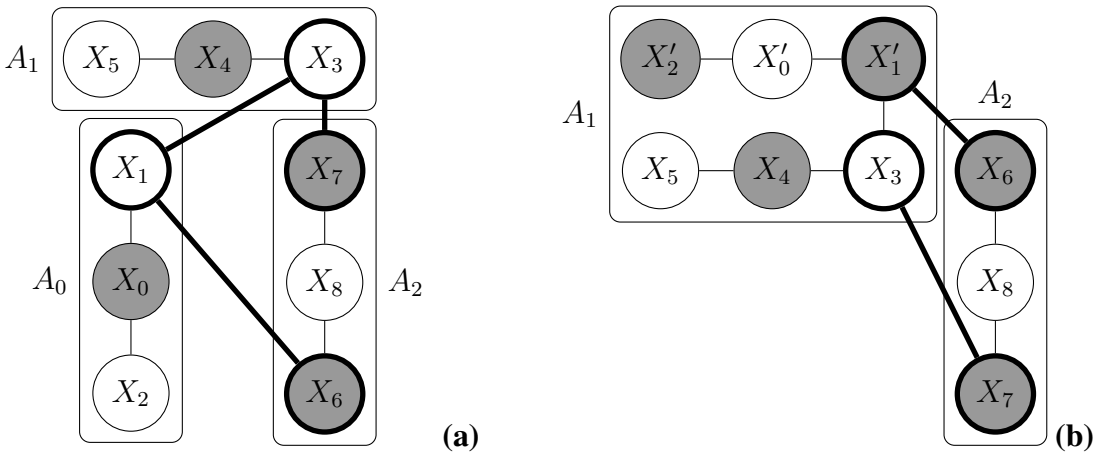


FIGURE 3.8 – Répartition des CSPs locaux avant (a) et après (b) la fusion des CSPs

Après la fusion, nous pouvons observer que la contrainte qui relie les variables X_1 et X_3 (C_{13}), qui était une contrainte inter-agent, devient une contrainte intra-agent de l'agent A_1 (C'_{13}). Après la fusion des CSPs, l'agent A_1 informe l'agent A_2 (le seul voisin que possède l'agent A_0) qu'il prend en charge CSP_0 grâce au message $\langle \mathrm{NewCSP}(A_1, A_0)\rangle)$ (Message 9). En recevant ce message, l'agent A_2 remplace l'agent A_0 par A_1 dans sa liste des voisins (qui fait partie déjà de l'ensemble de ses voisins). A la fin, l'agent A_1 met à jour sa liste des voisins en supprimant l'agent A_0, et les agents A_1 et A_2 reprennent la résolution du DisCSP depuis le début.

Dans la section qui suit, nous détaillons les différents propriétés de notre approche.

3.4.3 Propriétés de l'approche proposée

Dans cette section, nous montrons que notre approche est correcte, complète, et qu'elle se termine. Ensuite, nous présentons la complexité spatiale et temporelle de cette approche.

1. Preuve de correction

Théorème 1. Le résultat final de la résolution donne une solution seulement si toutes les contraintes du DisCSP sont satisfaites en présence d'un seul agent défaillant.

Démonstration. L'approche est basée sur l'algorithme Multi-ABT qui est correct [Yokoo et al., 1992]. Si une solution existe, la résolution du CSP de l'agent défaillant est garantie puisqu'il est copié au sein d'un autre agent qui prend en charge sa résolution. □

2. Preuve de complétude

Théorème 2. L'approche proposée est complète puisqu'elle utilise l'algorithme Multi-ABT comme algorithme de résolution dont la complétude a été prouvée par [Yokoo et al., 1992].

Démonstration. Afin de prouver la complétude de notre approche, nous devons prouver la complétude du processus de la réplication exécutée par l'*Agent Dispatcher*, et la complétude de la fusion des CSPs.

L'affectation des copies des CSPs aux agents dépend de l'ordre du parcours des voisins. En modifiant l'ordre des voisins, l'algorithme de réplication est capable de générer toutes les combinaisons d'affectation possibles. Reprenons par exemple la figure 3.4. Si l'*Agent Dispatcher* parcourt les voisins des agents dans cet ordre : $\Gamma(A_0) = \{A_1, A_2\}, \Gamma(A_1) = \{A_0, A_2\}, \Gamma(A_2) = \{A_0, A_1\}$, la réplication est effectuée comme suit : $Affect = \{(A_0, \{CSP_1, CSP_2\}), (A$ Mais si les voisins sont parcourus dans cet ordre $\Gamma(A_0) = \{A_2, A_1\}, \Gamma(A_1) = \{A_2, A_0\}, \Gamma(A_2) = \{A_1, A_0\}$, le résultat de la réplication sera $Affect = \{(A_0, CSP_2), (A_1, CSP_0), (A_2, CSP_1)\}$. D'où la complétude du processus de réplication.

A partir de la complétude de Multi-ABT, chaque agent est capable de générer toutes ses solutions locales. Après la fusion des CSPs, l'agent délégué garde cette propriété et peut générer toutes les solutions locales de son nouveau CSP. D'où la complétude du processus de la fusion des CSPs. □

3. Preuve de terminaison

Théorème 3. L'approche se termine dès qu'une solution est trouvée ou s'il n'existe pas de solution.

Démonstration. Afin de prouver la terminaison de notre approche, il faut d'abord prouver que les processus de distribution et de fusion se terminent.

L'algorithme de réplication (Algorithme 1) possède une liste de CSPs locaux comme entrée. Après chaque itération, l'*Agent Dispatcher* supprime un CSP local de cette liste jusqu'à avoir une liste vide. C'est à ce moment là que l'algorithme de réplication se termine. Concernant l'algorithme de fusion des CSPs (Algorithme 3), il fusionne deux listes en une seule liste : les deux ensembles des variables sont fusionnés en un seul ensemble, ainsi que les ensembles de contraintes. Une fois que ces listes sont fusionnées, l'algorithme de fusion se termine. Le processus général se termine quand l'algorithme Multi-ABT, dont la terminaison est prouvée par [Yokoo et al., 1992], s'arrête. □

4. Complexité spatiale

Théorème 4. La complexité spatiale de notre approche est d'ordre $O(m.d^n)$.

Démonstration. Dans Multi-ABT, afin de trouver toutes ses solutions locales, un agent a

besoin d'un espace mémoire d'ordre $O(d^n)$ tel que d est la taille du domaine de chaque variable, et n est le nombre de variables par agent. Pour répliquer les CSPs, l'*Agent Dispatcher* possède la liste des CSPs locaux, donc il a besoin d'un espace mémoire pour les variables de taille $O(N)$ tel que N est le nombre total de variables du problème. Puisque le problème possède m agents, notre approche nécessite un espace mémoire de taille $O(m.d^n)$. Cet espace est négligeable par rapport à $O(d^n)$. □

5. Complexité temporelle

Théorème 5. La compléxité temporelle de notre approche est d'ordre $O(d^N)$.

Démonstration. Pour trouver les solutions locales, la complexité temporelle est d'ordre $O(e_{intra_i}.{d_i}^n)$ pour chaque agent i, tel que e_{intra_i} est le nombre de contraintes intra-agents d'un agent i, et d est la taille de son domaine. Dans le pire des cas, l'algorithme de résolution vérifie toutes les combinaisons possibles des solutions locales pour chaque contrainte inter-agents. Cette vérification est d'ordre $O(e_{inter}.d^N)$ avec N est le nombre total de variables, et e_{inter} le nombre de contraintes inter-agents. Concernant l'algorithme de réplication, au pire des cas, chaque agent possède $(m-1)$ voisins, avec m est le nombre d'agents. Donc l'algorithme parcourt (m^2-1), d'où la complexité du processus de réplication d'ordre $O(m^2)$. Cette valeur est négligeable par rapport à la résolution du DisCSP. Les messages échangés pour détecter une défaillance et pour résoudre un DisCSP sont échangés dans un temps polynomial. Alors, la complexité de l'approche est d'ordre $O(\sum_{i=0}^{m-1} e_{intra_i}.d^n + e_{inter}.d^N)$.

$\sum_{i=0}^{m-1} e_{intra_i}.d^n$ est équivalente à $e_{intra} \sum_{i=0}^{m-1} .d^n$ avec e_{intra} est le nombre de toutes les contraintes intra-agents. Puisque $e_{intra} < e$ et $e_{inter} < e$ avec e le nombre total de contraintes, la complexité devient d'ordre $O(e. \sum_{i=0}^{m-1} .d^n + e.d^N)$. Nous avons $n << N$, alors $d^n << d^N$. La complexité devient d'ordre $O(e.d^N)$. Cette dernière reste négligeable par rapport à d^N. Donc, l'approche possède une complexité d'ordre $O(d^N)$. □

3.5 Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche permettant la résolution d'un DisCSP en présence d'agents défaillants. Elle est basée sur le principe de la répli-

cation : chaque CSP local est copié au sein d'un autre agent. Le CSP local de l'agent défaillant est pris en charge par l'agent ayant la copie de son CSP appelé *agent délégué*. Cette réplication est effectuée par un *Agent Dispatcher*. La détection d'un agent défaillant est assurée par l'échange de nouveaux messages appartenant au modèle *pull*, et la prise en charge est effectuée par la fusion des CSPs locaux. La fusion fait apparaitre un nouveau CSP local appartenant à l'*agent délégué*.

La présence d'un agent défaillant engendre la perte d'une partie du CSP global. Pour remédier à cette perte, la fusion du CSP local de l'agent délégué et le réplicat du CSP de l'agent défaillant assure la conservation du CSP initial. Dans le chapitre qui suit, nous présentons les résultats obtenus en appliquant notre approche sur un ensemble d'instances ayant des paramètres variés. Nous présenterons aussi les résultats obtenus en cas d'élimination de certains éléments (tel que le message $\langle \mathrm{CHECK} \rangle$).

Chapitre 4

Résolution de DisCSP en présence d'un agent défaillant

Sommaire

4.1 Introduction

Afin de valider notre approche proposée de tolérance aux fautes au niveau des DisCSP en présence d'agents défaillants, nous avons effectué un ensemble d'expérimentations. À partir de ces expérimentations, nous visons à résoudre des DisCSPs en présence d'un ensemble d'agents défaillants.

Dans la 1^{ere} section (4.2), nous présentons les protocoles des expérimentations, en citant les paramètres à partir desquels les DisCSPs sont créés, et les critères d'évaluation des résultats obtenus.

La 2^{eme} section (4.3) concerne les résultats obtenus en exécutant l'algorithme de réplication. Nous visons à montrer que l'application de cet algorithme ne risque pas de centraliser le problème initial.

Dans la 3^{eme}section (4.4), nous présentons l'impact de la variation des différents paramètres de création des DisCSPs, en présence d'un agent défaillant. Et afin d'améliorer les résultats obtenus, nous proposons une évaluation des résultats en éliminant la diffusion du message d'information $\langle \mathrm{isFailed} \rangle$. Cela afin de vérifier si la diffusion de l'information de défaillance influence le temps de traitement de la défaillance.

Dans la dernière section (4.5), nous présentons les résultats obtenus lors de la résolution des DisCSPs en présence de plusieurs agents. Ces résultats contiennent le pourcentage des instances qui ont donné les résultats attendus en présence de plusieurs défaillances, et la variation des critères d'évaluation.

4.2 Protocoles des expérimentations

Dans cette section, nous présentons les conditions d'exécution de nos expérimentations. Les instances des DisCSPs sont générées aléatoirement selon les paramètres $\langle m, n, d, p \rangle$ tel que :

- m est le nombre d'agents du DisCSP.
- n est le nombre de variables par agent, ayant chacune un domaine de taille d.
- p est la dureté des contraintes inter et intra-agents des DisCSPs.

Afin de générer les CSPs aléatoirement, nous proposons la valeur du nombre de contraintes $e = -N.(\frac{ln(d)}{ln(1-p)})$ avec N est le nombre de variables ayant chacune un domaine de taille d. Cette valeur de e est obtenue à partir des travaux proposés par [Xu et al., 2007] : pour définir les valeurs de d et e, [Xu et al., 2007] ont proposé 2 paramètres $\alpha > 0$ et $r > 0$ pour générer un DisCSP tel que :

$$d = N^{\alpha} \tag{4.1}$$

$$e = r.N.\ln(N) \tag{4.2}$$

L'équation (4.1), α nous donne :

$$\alpha = \frac{\ln(d)}{\ln(N)} \tag{4.3}$$

Un point critique [Xu et al., 2007] se définit suivant la valeur de p tel que :

$$p = 1 - exp^{\frac{-\alpha}{r}} \tag{4.4}$$

sous condition que $\alpha > 0.5$ et $p \leqslant 50\,\%$, d'où la définition de la valeur de r

$$r = \frac{-\alpha}{\ln(1-p)} \tag{4.5}$$

A partir des équations (4.2), (4.3) et (4.4), le nombre de contraintes est alors défini par :

$$e = -N.(\frac{ln(d)}{ln(1-p)})$$

Pour générer des instances de DisCSPs, nous varions les valeurs des paramètres m et n en fixant le nombre de contraintes e à la valeur prédéfinie précédemment. Durant les expérimentations, chaque agent possède un $DelaiActif$ de 10 secondes (Définition 20). Si la valeur de $DelaiActif$ est inférieure à 10 secondes, un agent risque d'être considéré comme défaillant alors qu'il est actif. C'est le cas si nous fixons la valeur de $DelaiActif$ à 7 secondes.

Selon [Stanković et al., 2017], pour évaluer une approche de tolérance aux fautes, une comparaison est nécessaire soit entre des approches existantes, soit entre les résultats obtenus avec et sans traitement d'une défaillance. Afin d'évaluer notre approche, nous avons utilisé la deuxième méthode d'évaluation en comparant : (i) les résultats obtenus en traitant la défaillance, (ii) ceux obtenus en exécutant Multi-ABT sans défaillance, et (iii) ceux obtenus avec une défaillance non détectée. Ces résultats sont comparés selon :

- Le nombre de messages échangés : les messages de résolution échangés par Multi-ABT, et ceux échangés lors de la détection et de traitement de la défaillance.
- Le temps d'exécution CPU de l'approche avec et sans défaillance, en précisant le temps d'exécution de chaque étape de l'approche. Ce temps CPU est mesuré dès le début de la résolution du DisCSP jusqu'à la fin du comportement de l'agent le plus lent.

- Le nombre de vérifications des contraintes durant la résolution des DisCSPs (NCCCs).

La comparaison de ces critères est effectuée par rapport à la moyenne des valeurs obtenues pour 50 instances de DisCSPs. La défaillance d'un agent est simulée soit après l'envoie de sa première solution, soit avant de recevoir un premier message. Les expérimentations sont effectuées avec la plateforme multi-agent JADE [Bellifemine et al., 2002]. Les résultats des simulations sont obtenus avec une machine équipée de 2.4 GHz Intel Core i7, et une mémoire RAM de taille $8GB$.

Dans la prochaine section, nous présentons les résultats obtenus en exécutant l'algorithme de réplication (exécuté par l'*Agent Dispatcher*).

4.3 Implémentation de l'algorithme de réplication

Dans cette section, nous évaluons les résultats de la réplication des CSPs fournis lors de l'exécution de l'algorithme de réplication (Algorithme 1).

Comme nous l'avons déjà précisé dans le chapitre 3, la conception de l'algorithme de réplication assure que chaque CSP local soit répliqué au sein d'un autre agent une et une seule fois, et que chaque agent peut avoir des copies de plusieurs CSPs locaux. L'affectation de plusieurs copies de CSPs à un seul agent peut entrainer la centralisation du problème : tous les CSPs locaux risquent d'être copiés au sein du même agent.

Afin de vérifier si nous nous retrouvons dans une telle situation, nous avons comparé le nombre d'agents du système à celui des agents ayant une liste $Dupp(A)$ non vide. Nous avons comparé aussi le nombre minimal et maximal de contraintes intra-agents en cas de fusion des CSPs afin de vérifier si après la fusion, les agents délégués auront tous des CSPs locaux de taille proportionnelle.

Le tableau 4.1 contient les valeurs d'évaluation de l'algorithme de réplication tel que :

Nb_agents : Le nombre d'agents du système.

Nb_copies : Le nombre minimal et maximal d'agents ayant une liste $Dupp(A)$ non vide.

Ctes_intra : Pour chaque instance, nous sélectionnons le nombre maximal de contraintes intra-agents en cas de fusion. **Ctes_intra** présente les valeurs minimales et maxi-

males fournies par les instances et la moyenne du nombre de contraintes que possède un agent, en cas de fusion.

TABLE 4.1 – Résultats de l'algorithme de réplication

Nb_agents	Nb_copies		Ctes_intra		
	Min	Max	Min	Max	Moy
3	2	3	27	54	39
4	3	4	27	55	38
6	5	6	26	55	31
8	7	8	28	54	29
10	10	10	28	28	28
12	12	12	28	28	28

Les valeurs présentées dans le tableau 4.1 sont obtenues en éxecutant l'algorithme de réplication pour 50 instances de DisCSPs ayant comme paramètres $\langle m, 4, 4, 0.5\rangle$. Ces paramètres permettent d'augmenter le nombre d'agents jusqu'à 12 agents sans avoir un problème d'espace mémoire. En plus, la valeur de $p = 0.5$ permet d'avoir des instances de CSPs qui peuvent soit avoir une solution, soit ne pas avoir de solution d'une manière égale.

Au niveau du nombre d'agents ayant des copies de CSPs locaux, nous remarquons qu'un seul agent ne possède pas de copie si le nombre d'agents est minimal : au pire des cas, pour m agents, $(m - 1)$ agents possèdent des copies d'autres agents. Donc, si le nombre d'agents est inférieur à 10, un des agents aura des copies de deux CSPs différents, ce qui explique la différence entre les valeurs minimales et maximales de $Ctes_{intra}$ (un des agents délégués prendra en charge la résolution des CSPs locaux de deux agents défaillants). A partir de 10 agents, tous les agents ont des ensembles $Dupp$ non vides. Dans ce cas, chaque agent prendra en charge la résolution du CSP local d'un seul agent défaillant. Nous remarquons qu'à partir de 10 agents, les agents délégués possèdent des CSPs locaux de même taille (même nombre de variables et même nombre de contraintes intra-agents). Donc, plus le nombre d'agents du DisCSP augmente, plus la réplication des CSPs est équilibrée. A partir de ces résultats, nous pouvons admettre qu'en exécutant l'algorithme de réplication, il n'y a pas de risque que tous les CSPs locaux soient répliqués

au sein du même agent. De cette façon, notons que le problème initial conserve son aspect distribué même en copiant les CSPs locaux des agents défaillants au sein d'autres agents, et quelque soit le nombre d'agents du DisCSP. Dans la prochaine section, nous présentons les résultats obtenus en traitant un agent défaillant lors de la résolution de DisCSPs.

4.4 Evaluation expérimentale en présence d'un seul agent défaillant

Après la détection d'un agent défaillant, l'agent délégué fusionne son propre CSP avec la copie du CSP de l'agent défaillant. Dans le cas d'un CSP mono-variable, après la fusion, l'agent délégué aura 2 variables dans son CSP local. Dans ce cas, nous ne parlons plus de CSP mono-variable, mais de CSP multi-variables. C'est pour cette raison que nous présentons les résultats de la résolution dans le cadre des CSPs multi-variables. Dans cette section, nous nous intéressons aux résultats obtenus en appliquant notre approche en présence d'un seul agent défaillant, et en faisant varier le nombre d'agents dans les cas suivants :

- Sans défaillance : les valeurs obtenues en appliquant Multi-ABT.
- Défaillance non détectée : les valeurs obtenues en exécutant Multi-ABT en présence d'une défaillance non détectée (sans appliquer notre approche).
- Défaillance traitée : les valeurs obtenues en appliquant notre approche.

4.4.1 Impact du nombre d'agents

Les DisCSPs sont générés avec les paramètres $\langle m, 4, 4, 0.5\rangle$. La figure 4.1 présente la variation des critères d'évaluation en faisant varier le nombre d'agents entre 3 et 12. Au delà de 12 agents, nous nous retrouvons avec un problème d'espace mémoire. Dans le cas mono-variables, le nombre d'agents peut atteindre 35 agents [Chakchouk et al., 2017].

Les messages échangés en appliquant notre approche sont divisés en 2 catégories : (i) les messages de résolution liés à Multi-ABT ($\langle \mathrm{OK?}\rangle$ et $\langle \mathrm{BT}\rangle$), et (ii) les messages

supplémentaires échangés pour détecter et traiter la défaillance. Le tableau 4.2 contient le nombre de messages de résolution et supplémentaires échangés.

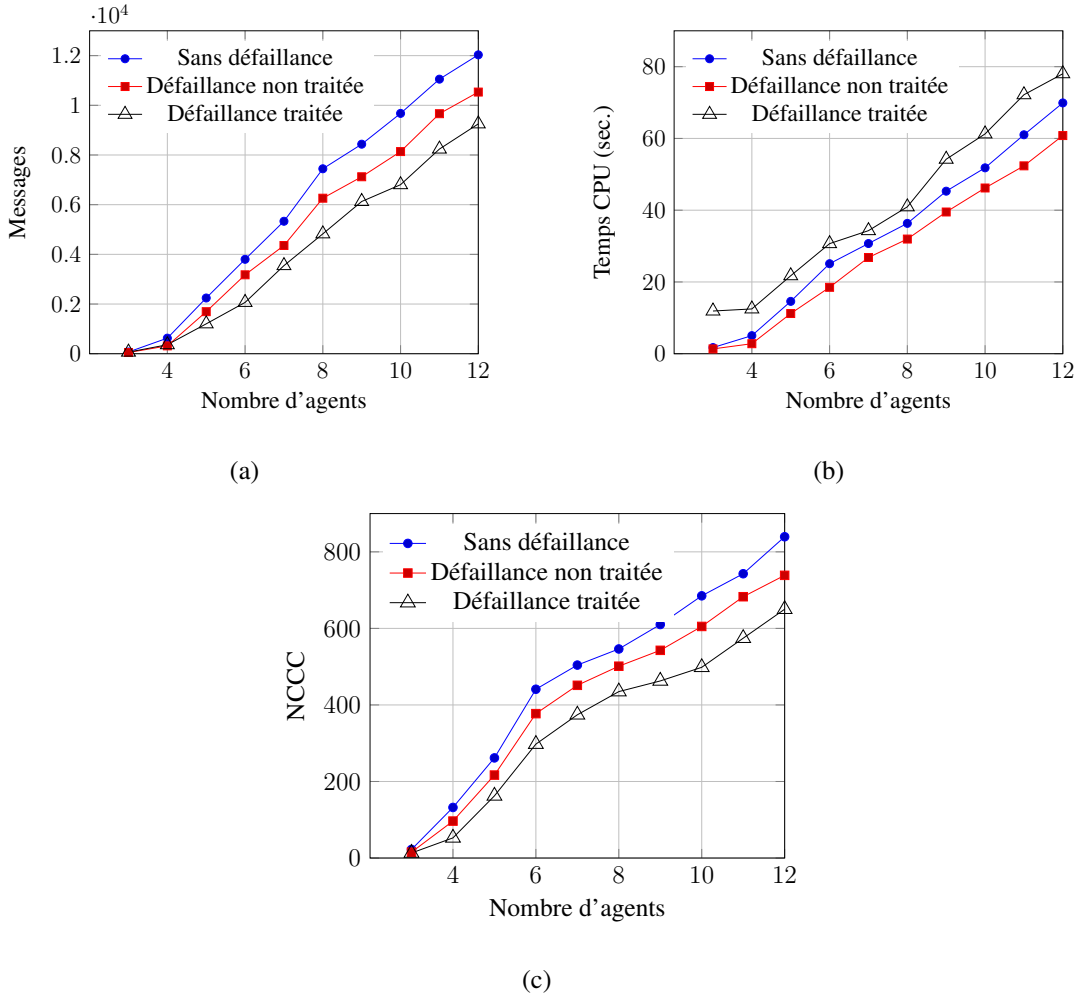


FIGURE 4.1 – Impact du nombre d'agents pour un problème $\langle m, 4, 4, 0.5 \rangle$

D'après les courbes présentées dans la figure 4.1(a), nous remarquons que si la défaillance n'est pas détectée, le nombre de messages diminue. Cela est dûe à la perte des messages supposés être envoyés par l'agent défaillant. Cette diminution est observée aussi si la défaillance est traitée : puisque deux CSPs locaux ont été fusionnés, le nombre de contraintes inter-agents diminue, pouvant engendrer la diminution de l'échange des messages entre deux agents.

Après la fusion des CSPs, les agents reprennent la résolution du CSP. Lors de la reprise, les agents modifient le contenu de leurs listes AgentView en éliminant les solutions proposées auparavant par l'agent défaillant et l'agent délégué. Ensuite, ils proposent des solutions cohérentes avec le nouveau contenu de leurs AgentView. Donc les solutions

TABLE 4.2 – Nombre de messages pour un problème $\langle m, 4, 4, 0.5\rangle$ en traitant un agent défaillant

Nombre d'agents	3	4	6	8	10	12
Messages de résolution	64.2	361.5	2065	4825	6804.5	9252.4
Messages supplémentaires	28.5	44.6	232.25	694.54	997.83	2012

proposées par un agent avant la détection de la défaillance, et qui ne sont pas cohérentes avec le nouveau contenu de l'$\mathrm{AgentView}$, ne sont pas proposées une nouvelle fois. En remplaçant l'agent défaillant par l'agent délégué dans leurs listes de voisins, les agents échangent avec l'agent délégué les mêmes messages échangés avec l'agent défaillant en exécutant Multi-ABT sans défaillance. Mais, ce n'est pas le cas pour l'agent délégué : la contrainte qui le lie avec l'agent défaillant est devenu une contrainte intra-agent. Donc, les messages supposés être échangés entre l'agent délégué et l'agent défaillant disparaissent, expliquant la diminution du nombre de messages de résolution échangés si la défaillance est traitée. D'après les valeurs présentées dans la tableau 4.2, nous remarquons que le nombre de messages échangés pour détecter et traiter la défaillance (les messages supplémentaires) varie de $40\,\%$ à $2\,\%$ du nombre global de messages échangés : il dépend du nombre d'agents du CSP mais pas du nombre de messages de résolution échangés.

TABLE 4.3 – Variation du temps CPU (en sec) pour un problème $\langle m, 4, 4, 0.5\rangle$ en traitant un agent défaillant

Agents	3	4	6	8	10	12
CPU total	11.9	12.48	30.68	40.94	61.26	78.03
CPU supplémentaire	0.31	0.38	0.74	1.33	1.94	3.35
CPU de la réplication	0.01	0.01	0.03	0.03	0.06	0.06

La figure 4.1(b) présente le temps CPU de résolution des DisCSPs avec et sans présence d'agent défaillant. Le tableau 4.2 contient les valeurs du temps CPU total mesuré dès le début de la résolution jusqu'à la fin du comportement de l'agent le plus lent. Il contient aussi le temps CPU supplémentaire mesuré dès la détection de l'agent défaillant jusqu'à la reprise de la résolution après la fusion des CSPs locaux, et le temps CPU de la réplication des CSPs. D'après la figure 4.1(b), une diminution est observée en présence

d'une défaillance non détectée. Cette diminution est dûe à la disparition du comportement de l'agent défaillant (échange de messages, recherche d'une solution locale, etc). D'un autre côté, une augmentation du temps CPU est observée en appliquant notre approche. Cette augmentation est dûe au traitement des informations supplémentaires : le temps de détection de l'agent défaillant, la fusion des CSPs locaux, et la transmission des informations de défaillance (les messages $\langle \mathrm{isFailed} \rangle$ et $\langle \mathrm{NewCSP} \rangle$), est un temps supplémentaire ajouté au CPU initial. Mais, d'après les valeurs du tableau 4.3, ce temps CPU supplémentaire reste négligeable puisqu'il présente entre $1{,}4\,\%$ et $5\,\%$ du temps CPU total. Le temps CPU utilisé par l'*Agent Dispatcher* est aussi négligeable par rapport au temps CPU total, puisqu'il ne dépasse pas $0{,}1\,\%$.

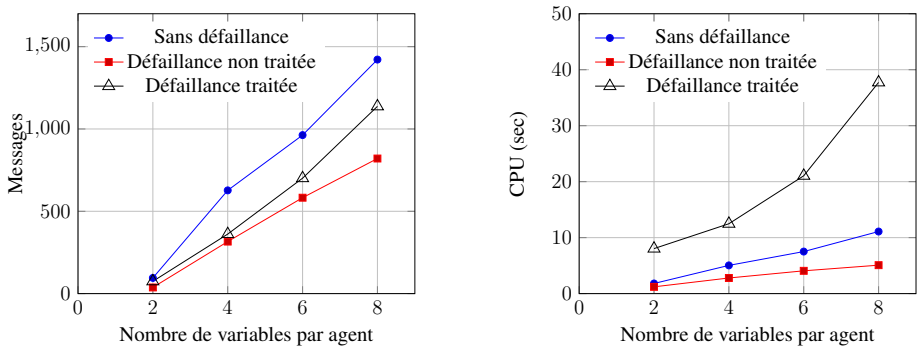
La figure 4.1(c) présente le nombre de vérifications des contraintes NCCCs. Nous observons une diminution des NCCCs si la défaillance n'est pas détectée, dûe à la perte de vérification de certaines contraintes inter-agents (concernant l'agent défaillant). En appliquant le processus de traitement de la défaillance, les valeurs de NCCCs diminuent. En effet, le nombre de solutions à vérifier de l'agent délégué est réduit : plus le nombre de ses contraintes intra-agents augmente, plus le nombre de ses solutions locales diminue.

Les résultats obtenus ont montré que l'approche proposée est capable de résoudre un DisCSP en présence d'un seul agent défaillant. Nous remarquons qu'en augmentant le nombre d'agents, les valeurs de nos critères d'évaluation augmentent. Nous observons aussi que notre approche n'est coûteuse qu'au niveau du temps CPU. Dans la prochaine section, nous présentons l'impact de la variation du nombre de variables par agent.

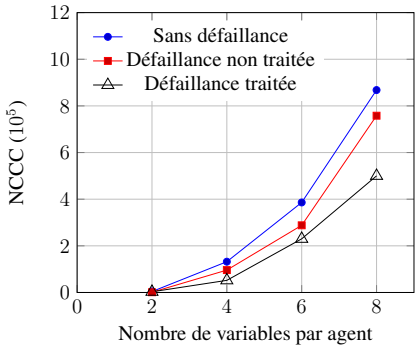
4.4.2 Impact du nombre de variables

Nous étudions l'impact du nombre de variables par agent en faisant varier le nombre de variables entre 2 et 8. Avec 10 variables par agent, nous nous trouvons avec un problème d'espace mémoire dû à la génération des solutions locales après la fusion. En effet, durant les expérimentations, un agent est incapable de résoudre un CSP local avec 20 variables. Les résultats concernent des instances de paramètres $\langle 4, n, 4, 0.5 \rangle$.

La figure 4.2 présente la variation des critères d'évaluation en modifiant le nombre de variables par agent, et le tableau 4.4 contient le nombre de messages de résolution et supplémentaires. D'après les courbes de la figure 4.2(a), nous remarquons que le nombre de



(a) (b)



(c)

FIGURE 4.2 – Impact du nombre de variables par agent pour un problème $\langle 4, n, 4, 0.5\rangle$

messages, selon notre approche, diminue par rapport à celui obtenu durant la résolution du DisCSP sans présence d'agent défaillant. Cela est dû à la diminution du nombre de contraintes inter-agents, et donc de la diminution du nombre de messages échangés entre les agents. D'un autre côté, nous remarquons que le nombre de messages lors du traitement de la défaillance est supérieur à celui obtenu si la défaillance n'est pas traitée. Dans ce cas, la défaillance est détectée après que les agents aient échangé tous leurs messages de résolution. D'après les valeurs du tableau 4.4, les messages dédiés à la détection et au traitement de la défaillance varient de $50\,\%$ à $7\,\%$.

TABLE 4.4 – Nombre de messages pour un problème $\langle 4, n, 4, 0.5\rangle$ en traitant un agent défaillant

Nombre de variables	2	4	6	8
Messages de résolution	74.55	361.5	701.04	1137.37
Messages supplémentaires	39.05	44.6	63.5	82.25

TABLE 4.5 – Variation du temps CPU (en sec) pour un problème $\langle 4, n, 4, 0.5\rangle$

Nombre de variables	2	4	6	8
CPU total	8.04	12.48	21.05	37.73
CPU supplémentaire	0.21	0.38	1.12	2.74
CPU de la réplication	0.01	0.01	0.01	0.02

Au niveau de la variation du temps CPU présentée par la figure 4.2(b), nous remarquons qu'en appliquant notre approche, le temps CPU augmente par rapport à celui des autres cas. La résolution des DisCSPs en présence d'une défaillance non traitée, ou s'il n'existe pas de défaillance, ne dépasse pas 10 secondes. Mais puisque nous avons fixé auparavant les paramètres de détection de défaillance (les valeurs $Delai$ et $DelaiActif$) à 10 secondes, le temps CPU est doublé avant même de commencer le traitement. D'après le contenu du tableau 4.5, le temps d'exécution de détection et de traitement de la défaillance ne dépasse pas $16\,\%$ du temps d'exécution global. En effet, en augmentant le nombre de variables, le temps CPU de la fusion des CSPs augmente. Malgré l'augmentation du nombre de variables, le temps CPU de la réplication des CSPs reste négligeable par rapport au temps CPU total.

Par contre, nous remarquons que la variation du nombre de variables n'influence pas la croissance du nombre de vérifications des contraintes (NCCCs) (Figure 4.2(c)) : la croissance des valeurs de NCCCs reste monotone. La diminution du NCCCs est dûe à la diminution du nombre de contraintes à vérifier : après la fusion, le nombre de contraintes inter-agents diminue, et l'agent délégué possède plus de contraintes intra-agents à satisfaire, donc moins de solutions locales à vérifier.

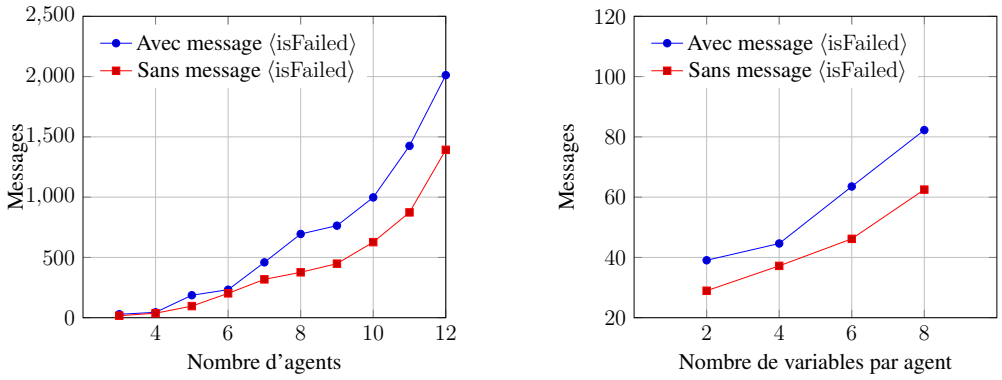
Selon les valeurs obtenues dans cette section, nous avons remarqué que la variation du nombre de variables par agent a le même impact que celui de la variation du nombre d'agents du DisCSP : la variation du nombre de variables n'est coûteux qu'au niveau du temps CPU. Mais ce dernier peut être moins coûteux si nous diminuons les valeurs des délais de détection de la défaillance. Il peut être aussi moins coûteux si nous éliminons les messages $\langle \mathrm{isFailed}\rangle$. Afin de valider cette hypothèse, nous présentons, dans la prochaine section, les résultats obtenus en cas d'élimination du message $\langle \mathrm{isFailed}\rangle$.

4.4.3 Evaluation sans messages d'information ⟨isFailed⟩

Afin d'améliorer notre approche, nous proposons d'éliminer quelques messages lors de traitement de la défaillance. Dans cette section, nous vérifions si cette amélioration affecte les résultats finaux de résolution des DisCSPs.

Durant la résolution des DisCSPs en présence d'un agent défaillant, nous avons introduit des messages d'informations ⟨isFailed⟩ qui circulent entre les agents. Le but de ces messages est d'informer les agents du système de la présence d'un agent défaillant. Mais l'inconvénient de ces messages est qu'ils sont transmis fréquemment à des agents qui ne sont pas concernés, c'est-à-dire, à des agents qui ne sont pas voisins de l'agent défaillant.

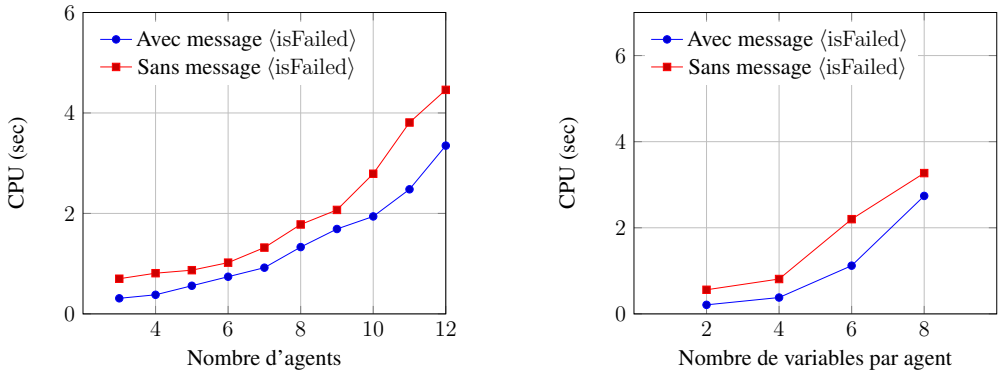
En éliminant ce type de messages, les agents procèdent comme suit : chaque agent vérifie l'activité de ses voisins en transmettant les messages ⟨CHECK⟩. Si l'agent qui détecte la défaillance possède la copie de l'agent défaillant, il effectue la fusion des CSPs locaux ; sinon, il continue la résolution du DisCSP. Dans cette section, nous présentons l'impact de l'élimination du message ⟨isFailed⟩.



(a) (b)

FIGURE 4.3 – Elimination du message ⟨isFailed⟩

La figure 4.3 présente la variation du nombre de messages supplémentaires échangés (⟨CHECK⟩ et ⟨ACTIVE⟩) en éliminant la transmission des messages ⟨isFailed⟩. A partir des résultats obtenus, nous pouvons remarquer que, quelque soit le paramètre que nous faisons varier (nombre d'agents ou nombre de variables), le nombre de messages supplémentaires échangés en l'absence du message ⟨isFailed⟩ est inférieur à celui échangé si les agents sont informés de la défaillance. Cette diminution montre que le nombre de messages ⟨CHECK⟩ reste négligeable.



(a) (b)

FIGURE 4.4 – Elimination du message ⟨isFailed⟩

La figure 4.4 présente la variation du temps CPU de détection et du traitement de la défaillance. Elle présente les résultats obtenus en présence des messages d'informations ⟨isFailed⟩ et ceux obtenus en les supprimant. Selon les résultats obtenus, nous remarquons une augmentation du temps CPU. En diffusant le message ⟨isFailed⟩, l'information de la défaillance circule entre les agents, ce qui accélère le temps de traitement (l'agent délégué est informé plus rapidement). Par contre, dans ce cas, la défaillance n'est traitée que si l'agent délégué la détecte, expliquant le retard de la fusion.

Dans cette section, nous avons essayé d'améliorer notre approche en éliminant le message d'information ⟨isFailed⟩. Les résultats obtenus montrent que l'échange de ce message est important si nous voulons que l'agent délégué traite la défaillance le plus rapidement possible. Par contre, il est coûteux au niveau du nombre de messages échangés.

4.5 Résolution avec plusieurs agents défaillants

Les résultats obtenus dans la section précédente montrent que l'approche proposée est capable de résoudre un DisCSP en présence d'un seul agent défaillant. Dans cette section, nous présentons les résultats de la résolution d'un DisCSP en présence de plusieurs agents défaillants. Nous avons généré aléatoirement 50 instances de DisCSP mono-variable de paramètres $m = 10$, $d = 5$ et $d = 0.4$ ($\langle 10, 1, 5, 0.4 \rangle$). Les résultats sont obtenus en faisant varier le nombre d'agents défaillants. Les agents défaillants sont choisis aléatoirement. Un agent tombe en panne soit après avoir envoyé sa première solution (s'il possède des

accointances inférieures), soit dès le début de la résolution (s'il possède la plus petite priorité).

TABLE 4.6 – Impact du nombre des agents défaillants : $\langle m, n, d, p\rangle = \langle 10, 1, 5, 0.4\rangle$

Agents défaillants		Déf. non détéctées	Déf. détéctées
	Déf. traitées	–	100 %
1	Messages	100,76	163,27
	CPU (sec)	22,43	24,25
	NCCC ($\cdot 10^3$)	157,8	117,65
	Déf. traitées	–	95 %
2	Messages	74,6	93,25
	CPU (sec)	21,32	25,63
	NCCC ($\cdot 10^3$)	88,79	70,52
	Déf. traitées	–	30 %
3	Messages	64,77	82
	CPU (sec)	20,43	26,8
	NCCC ($\cdot 10^3$)	69,57	55,7
	Déf. traitées	–	20 %
4	Messages	59,9	71,2
	CPU (sec)	19,32	28,49
	NCCC ($\cdot 10^3$)	49,38	30,62
	Déf. traitées	–	0 %
5	Messages	42,58	58,79
	CPU (sec)	19,03	32,54
	NCCC ($\cdot 10^3$)	31,89	20,09

Le tableau 4.6 présente les résultats obtenus en exécutant multi-ABT en présence de plusieurs agents défaillants. Il présente le nombre de messages de résolution échangés, le temps CPU total de la résolution du DisCSP, et les valeurs du NCCC en présence d'agents défaillants traités et non traités. Il présente aussi le pourcentage des instances qui ont traité toutes les défaillances présentes dans le DisCSP. Les résultats obtenus montrent qu'en présence d'un seul agent défaillant, toutes les instances générées fournissent des résultats corrects (elles fournissent une solution s'il en existe). Par contre, en présence de 2 agents défaillants, seulement $95\,\%$ des instances générées détectent les deux défaillances, mais ne fournissent pas forcement les résultats attendus. À partir de 3 agents défaillants, moins de $30\,\%$ des instances détectent toutes les défaillances mais donnent des résultats erronés.

En présence de 5 agents défaillants, notre approche n'est pas capable de détecter tous les agents défaillants.

Nous remarquons aussi une augmentation du nombre de messages échangés si quelques défaillances sont traitées, par rapport aux messages échangés si les défaillances ne sont pas traitées. En traitant les agents défaillants, les messages supposés être échangés entre les agents défaillants et les autres agents sont échangés entre l'agent délégué et les autres agents. Par contre, si les défaillances ne sont pas traitées, les messages supposés être échangés entre l'agent défaillant et les autres agents sont éliminés. D'un autre côté, le nombre de messages échangés diminue en augmentant le nombre d'agents défaillants. L'augmentation du nombre d'agents défaillants non traités engendre la diminution de la taille du DisCSP, c'est-à-dire, certains CSP locaux disparaissent du DisCSP global. Donc, le nombre de contraintes inter-agents diminue, d'où la diminution de l'interaction entre les agents.

Au niveau du temps CPU, si les défaillances ne sont pas traitées, plus le nombre d'agents défaillants augmente, plus le temps CPU diminue. Cela est dû à la diminution de la taille du CSP global, puisque les CSPs locaux des agents défaillants ne font plus partie du CSP. Mais, si certaines défaillances sont traitées, le temps d'exécution augmente si le nombre d'agents défaillants augmente. Cette augmentation est liée au temps supplémentaire dédié au traitement de la défaillance et la reprise de la résolution. Finalement, nous pouvons observer une diminution du nombre de vérifications de contraintes NCCCs en augmentant le nombre d'agents défaillants. Cette diminution est observée si les défaillances ont été traitées ou pas. Dans ces deux cas, les agents ont moins de contraintes à vérifier.

Selon les résultats présentés dans cette section, nous pouvons déduire que notre approche est capable de résoudre un DisCSP en présence d'un seul agent défaillant. A partir de deux agents défaillants, l'approche peut fournir des résultats erronés.

4.6 Conclusion

Dans ce chapitre, nous avons présenté l'impact de la présence d'un agent défaillant lors de la résolution d'un DisCSP. D'après les résultats obtenus, notre approche est ca-

pable de résoudre un DisCSP en présence d'un seul agent défaillant. Les résultats sont obtenus en résolvant des instances de DisCSPs générées aléatoirement, et en les comparant avec et sans présence d'agents défaillants. Ces résultats ont confirmé que notre approche est capable de résoudre un DisCSP en présence d'un seul agent défaillant. Nous avons montré aussi que la réplication des CSPs locaux conserve l'aspect distribué du problème.

Malgré son coût élevé du temps CPU, notre approche est capable de résoudre un DisCSP en présence d'un seul agent défaillant. Mais en présence de plusieurs agents défaillants, elle fournit des résultats erronés : soit elle détecte tous les agents défaillants et fournit des résultats erronés, soit elle ne détecte pas tous les agents défaillants. Dans ce cas, une partie du DisCSP n'est pas résolue. Dans le prochain chapitre, nous adaptons notre approche, en identifiant ses failles, afin de résoudre un DisCSP en présence de plusieurs agents défaillants.

Chapitre 5

Traitement de plusieurs défaillances

Sommaire

5.1 Introduction

Dans ce chapitre, nous présentons une amélioration de notre approche proposée dans les chapitres précédents. Cette nouvelle proposition vise à adapter notre approche à résoudre des DisCSPs en présence de plusieurs agents défaillants.

Dans la 1^{ere} section (5.2), nous présentons les raisons pour lesquelles notre approche ne traite pas plusieurs pannes. Cette section présente un contre-exemple de l'application de notre approche en présence de plusieurs agents défaillants. Cet exemple met en évidence les limites de l'approche proposée précédemment.

Dans la 2^{eme} section (5.3), nous décrivons l'adaptation de notre approche pour traiter plusieurs agents défaillants, en détaillant les modifications effectuées. À la fin de la section, nous présentons une illustration de cette adaptation.

Dans la 3^{eme} section (5.4), nous présentons les résultats obtenus en exécutant notre nouvelle approche. Ces résultats présentent le nombre de messages échangés, et le temps CPU de résolution. Ils sont obtenus en faisant varier les différents paramètres de création des DisCSPs, et le nombre d'agents défaillants.

5.2 Limites de notre approche

Dans cette section, nous nous intéressons aux raisons pour lesquelles notre approche n'est pas capable de résoudre un DisCSP en présence de plusieurs agents défaillants. La figure 5.1 présente un problème de coloration de graphe (chaque variable doit avoir une couleur différente de celle de ses variables voisines). Ce dernier est présenté sous forme d'un DisCSP composé de 5 agents d'ordre de priorité $A_1 \succ A_2 \succ A_0 \succ A_3 \succ A_4$. Chaque agent encapsule une seule variable de domaine $\{\circ, \bullet\}$. Notons que selon ces données, ce DisCSP ne possède pas de solution.

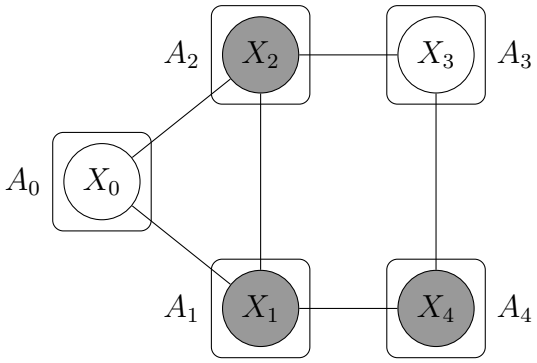


FIGURE 5.1 – Exemple de problème de coloration de graphe

Supposons que les agents A_2 et A_3 présentent les agents défaillants. L'approche est uti-

lisée sans diffusion du message $\langle isFailed \rangle$). Selon notre approche, les agents procèdent comme suit :

1. L'application de l'algorithme de réplication (Algorithme 1) donne la distribution suivante :
   - $Dupp(A_0) = \{CSP_2\}$
   - $Dupp(A_1) = \{CSP_0\}$
   - $Dupp(A_2) = \{CSP_1\}$
   - $Dupp(A_3) = \{CSP_4\}$
   - $Dupp(A_4) = \{CSP_3\}$
2. Après la détection des agents défaillants, l'agent A_0 fusionne son CSP avec la copie du CSP_2, et l'agent A_4 fusionne son CSP avec la copie du CSP_3 (figure 5.2).
3. L'agent A_0 envoie le message $\langle NewCSP \rangle$ aux agents A_1 et A_3 puisqu'il n'est pas au courant que l'agent A_3 est défaillant, et l'agent A_4 envoie le message $\langle NewCSP \rangle$ à l'agent A_2.
4. Après la mise à jour des agents, l'ordre de priorité des agents devient comme suit : $A_0 \succ A_1 \succ A_4$.

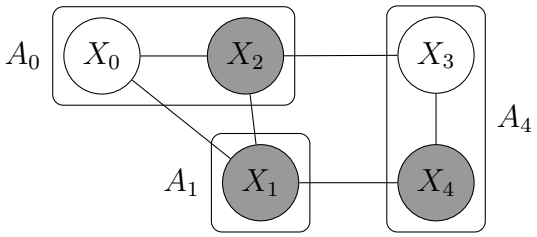


FIGURE 5.2 – Fusion des CSPs après la défaillance des agents A_2 et A_3

Lors de la reprise de la résolution du DisCSP, Multi-ABT impose que l'agent A_0 envoie sa solution locale aux agents A_1 et A_4. Mais ce n'est pas le cas pendant les expérimentations. En effet, A_0 n'enverra jamais de message à l'agent A_4. Même si l'agent A_0 a l'information de la défaillance de l'agent A_3 (si l'information de la défaillance est diffusée), il ne saura jamais que c'est l'agent A_4 qui le remplace. Donc, l'agent A_4 ne fait

pas partie de l'ensemble des voisins de A_0. Pour remédier à ce problème, il faut informer l'agent A_0 que c'est l'agent A_4 qui remplace A_3. De cette façon, la mise à jour des voisins restera cohérente.

Dans un 2^{eme} cas de l'exemple utilisé, et en utilisant le même ordre de distribution présenté précédemment, supposons que les agents A_3 et A_4 sont défaillants. Après la détection de ces défaillances, leurs CSPs locaux ne seront pris en charge par aucun autre agent. Leurs CSPs sont répliqués l'un au sein de l'autre : lorsque l'agent A_3 tombe en panne, et selon l'ordre de distribution effectué, c'est l'agent A_4 qui prend CSP_3 en charge. Mais, ce n'est pas le cas si l'agent A_4 est aussi défaillant. Dans ce cas, et puisque $CSP_4 \in Dupp(A_3)$, les CSPs locaux des agents A_3 et A_4 seront exclus du CSP global. De cette façon, la structure du CSP initial n'est plus conservée. Pour éviter cette situation, il faut garantir que les CSPs sont répliqués aux sein des agents non défaillants.

Ces contre exemples montrent que notre approche est limitée à la résolution d'un DisCSP en présence d'un seul agent défaillant. Pour l'améliorer, nous effectuons des modifications au niveau de la diffusion du message $\langle NewCSP \rangle$, et au niveau de l'exécution de l'algorithme de réplication. Ces modifications sont détaillées dans la prochaine section.

5.3 Adaptation de l'approche proposée

Dans cette section, nous présentons l'ensemble des modifications pour résoudre un DisCSP en présence de plusieurs agents défaillants [Chakchouk et al., 2018]. Ces modifications concernent la diffusion de l'information de délégation du CSP local d'un agent défaillant (5.3.1), et le moment d'exécution de l'algorithme de réplication (5.3.2).

5.3.1 Adaptation de la diffusion du message $\langle \mathrm{NewCSP} \rangle$

Dans le chapitre 3, nous avons indiqué que le message $\langle \mathrm{NewCSP} \rangle$ est envoyé par l'agent délégué aux voisins de l'agent défaillant, afin qu'ils mettent à jour leurs listes de voisins. Dans la section précédente, nous avons montré que cette méthode n'est pas capable de résoudre un DisCSP en présence de plusieurs agents défaillants.

Pour éviter la première situation présentée dans la section précédente, il faut que tous les agents connaissent le délégué de chaque agent défaillant. Pour ce faire, nous proposons

de diffuser le message $\langle \mathrm{NewCSP} \rangle$ à tous les agents non défaillants du système. De cette façon, le message prend la forme $\langle \mathrm{NewCSP}(A_i, A_j, \Gamma) \rangle$ tel que : (i) l'agent A_i est l'agent délégué, (ii) l'agent A_j est l'agent défaillant remplacé par A_i, et (iii) Γ est l'ensemble des voisins de l'agent émetteur du message. Donc, en considérant cette modification, les agents procèdent comme suit :

- Après avoir fusionné son CSP avec la copie du CSP de l'agent défaillant, l'agent délégué envoie $\langle \mathrm{NewCSP} \rangle$ à tous ses voisins, y compris les voisins de l'agent défaillant.
- En recevant ce message, un agent met à jour sa liste de voisins, et transmet le message à ses voisins.

Supposons qu'un agent A_i délègue un agent A_j, et qu'il reçoive un message $\langle \mathrm{NewCSP}(A_k, A_l, \Gamma(A_k)) \rangle$ indiquant qu'un agent A_k délègue un agent défaillant A_l. L'agent A_i met à jour sa liste de voisins comme suit :

- L'agent A_i ajoute à sa liste de voisins les voisins de l'agent A_j.
- Si l'agent A_l fait déjà partie des voisins de l'agent A_i, A_i remplace A_l par A_k.
- Si l'agent A_l fait partie des voisins de l'agent A_j, A_i ajoute A_k à sa liste de voisins.
- Sinon A_i transmet tout simplement le message à ses voisins et il ignore son contenu.

5.3.2 Adaptation de la réplication des CSPs locaux

La modification de l'algorithme de réplication vise à éviter la 2^{eme} situation décrite dans la section 5.2. Pour garantir que les agents délégués ne font pas partie des agents défaillants, nous proposons de répliquer les CSPs locaux après la détection des défaillances.

Pour ce faire, à chaque fois qu'un agent détecte une défaillance, il en informe l'Agent Dispatcher. Ce dernier exécute l'algorithme de réplication après avoir reçu les identifiants de tous les agents défaillants. Les instructions de l'algorithme de réplication ne changent pas, le seul changement est au niveau de la liste d'entrée (algorithme 1). En recevant les informations de défaillance, l'Agent Dispatcher (i) élimine les agents défaillants de la liste des agents auxquels il affectera des copies des CSPs locaux, et (ii) affecte les CSPs locaux

des agents défaillants aux agents actifs selon l'algorithme de réplication. Dans ce cas, la liste de tous les CSPs locaux n'est plus définie comme une entrée, et elle est remplacée par la liste de CSPs locaux d'agents défaillants.

5.3.3 Illustration

Soit l'exemple de coloration de graphe présenté à la figure 5.1. Les figures 5.3 et 5.4 présentent l'échange de messages entre les agents de résolution, et l'agent délégué. La figure 5.3 présente l'échange de messages avant la fusion des CSPs. Supposons que nous n'utilisons pas la diffusion de l'information de défaillance entre les agents (réf. section 4.4.3). Les agents commencent la résolution en échangeant les différents messages

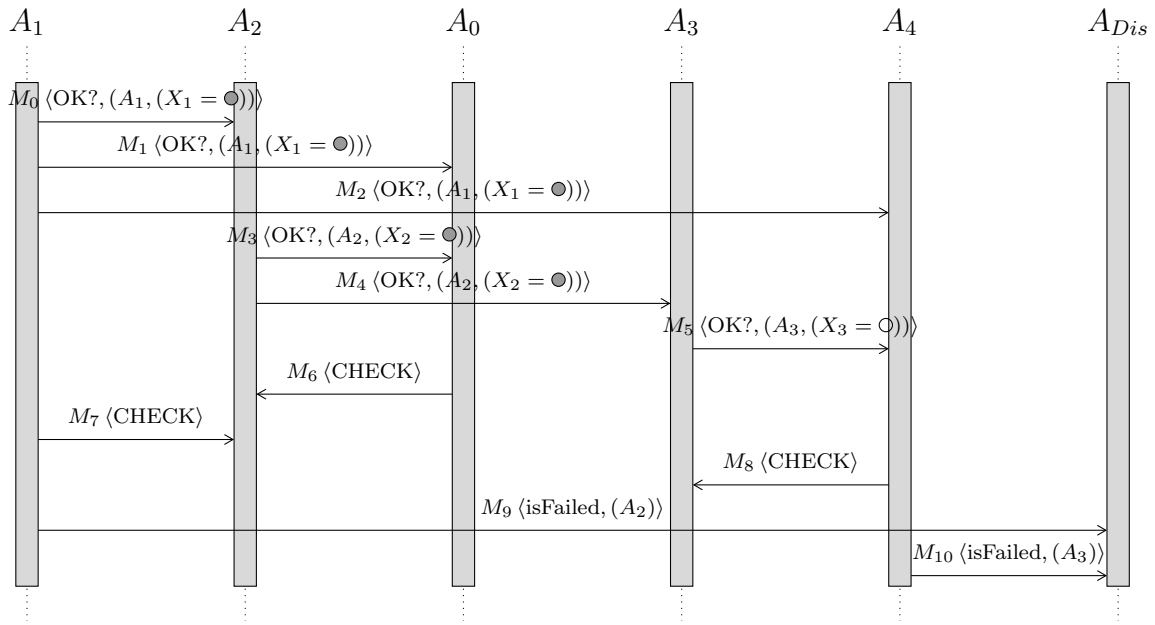


FIGURE 5.3 – Détection de la défaillance des agents A_2 et A_3

de résolution de Multi-ABT (Messages M_0 à M_5). Reprenons le même scénario de défaillance présenté dans la section 5.2 : supposons que les agents A_2 et A_3 tombent en panne juste après avoir envoyé leurs premières solutions via les messages M_4 et M_5 respectivement. Puisque nous n'utilisons pas la diffusion de l'information de défaillance, un agent défaillant peut être détecté par tous ses voisins. Après avoir dépassé $Delai(A_2)$, et $Delai(A_3)$ (réf. définition 19), les agents A_0 et A_1 envoient des messages ⟨CHECK⟩ à l'agent A_2 (Messages M_6 et M_7). L'activité de l'agent A_3 est vérifiée par l'agent A_4 via

le message M_8. Les agents A_2 et A_3 ne répondent pas dans un délai $DelaiActif(A)$ (réf. définition 20), donc l'agent A_4 concerne l'agent A_3 comme défaillant. Supposons que l'agent A_1 est le premier qui a détecté la défaillance de l'agent A_2. Après avoir détecté les agents défaillants, les agents A_1 et A_4 en informent l'*Agent Dispatcher*, en lui transmettant le message $\langle \text{isFailed} \rangle$ (messages M_9 et M_{10}). Pour ce faire, le contenu du message $\langle \text{isFailed} \rangle$ prend la forme $\langle \text{isFailed}(A_i) \rangle$ contenant l'identifiant de l'agent défaillant A_i. Si l'*Agent Dispatcher* reçoit l'information de défaillance d'un même agent plus d'une fois, il ignore le message. Par exemple, si l'Agent A_0 envoie le message $\langle \text{isFailed}(A_2) \rangle$, l'*Agent Dispatcher* ignore ce message puisqu'il a déjà reçu cette information de l'agent A_1.

En recevant ces informations, l'*Agent Dispatcher* réplique les CSPs locaux des agents défaillants chez les autres agents actifs, et envoie à chaque agent la copie du CSP qu'il doit prendre en charge. L'exécution de l'algorithme de réplication donne le résultat suivant :

- $Dupp(A_0) = \{CSP_2\}$
- $Dupp(A_1) = \{\}$
- $Dupp(A_4) = \{CSP_3\}$

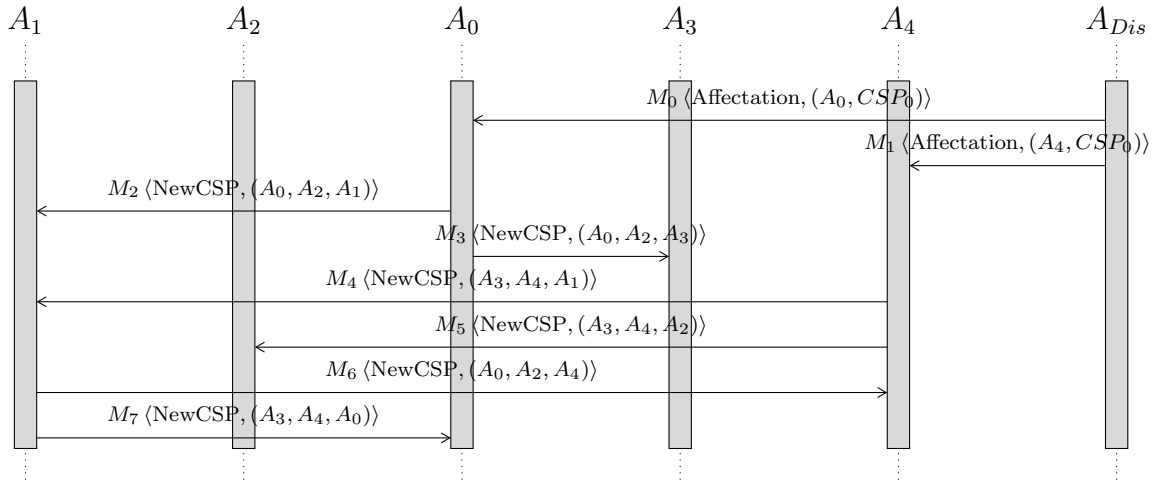


FIGURE 5.4 – Diffusion du message $\langle \text{NewCSP} \rangle$

La figure 5.4 présente l'échange de messages effectué entre les agents après la détection de défaillances. L'*Agent Dispatcher* envoie le résultat de la distribution des copies

des CSPs locaux aux agents A_0 et A_4 via les messages M_0 et M_1. En recevant ces messages, les agents A_0 et A_4 fusionnent leurs CSPs locaux avec les réplicats reçus pour obtenir le DisCSP de la figure 5.2. Après la fusion, et avant la diffusion des messages $\langle \mathrm{NewCSP} \rangle$, les agents A_0 et A_4 possèdent comme liste de voisins $\{A_1, A_3\}$ et $\{A_1, A_2\}$, respectivement. Ensuite, ils diffusent le message $\langle \mathrm{NewCSP} \rangle$:

- L'agent A_0 envoie $\langle \mathrm{NewCSP}(A_0, A_2, \{A_1, A_3\}) \rangle$ à ses voisins A_1 et A_3 (Messages M_2 et M_3) . Puisqu'il n'a pas l'information que l'agent A_3 est défaillant, il lui envoie aussi le message (qui ne sera pas traité).

- En recevant le message M_2, l'agent A_1 transmet ce message à l'agent A_4, en envoyant $\langle \mathrm{NewCSP}(A_0, A_2, \{A_4\}) \rangle$ (Message M_6), et met à jour sa liste de voisins en remplaçant l'agent A_2 par l'agent A_0.

- En recevant le message M_6, l'agent A_4 vérifie si l'agent A_2 fait partie de sa nouvelle liste de voisins après la fusion (appartient aux voisins de l'agent A_3) : il remplace l'agent A_2 par A_0. Par conséquent, les agents A_1 et A_4 auront comme liste de voisins $\{A_0, A_4\}$ et $\{A_1, A_0\}$, respectivement.

- L'agent A_4 envoie $\langle \mathrm{NewCSP}(A_3, A_4, \{A_1, A_2\}) \rangle$ à ses voisins $\{A_1, A_2\}$ (Messages M_5 et M_6). L'agent A_4 n'a pas l'information de défaillance de l'agent A_2.

- En recevant le message M_4, l'agent A_1 transmet $\langle \mathrm{NewCSP}(A_3, A_4, \{A_0\}) \rangle$ à l'agent A_0 (Message M_7). Puisque A_3 n'appartient pas à la liste de voisins de A_0, A_0 n'a pas de mise à jour à effectuer.

- En recevant le message M_7, l'agent A_0 remplace son voisin A_3 par l'agent A_4. De cette façon, l'agent A_0 aura comme voisins, les agents $\{A_1, A_4\}$.

Avec cette méthode de diffusion de l'information de délégation, notre approche est capable de traiter, plusieurs défaillances. Dans la section qui suit, nous présentons les résultats obtenus en implémentant cette méthode et faisant varier le nombre d'agents défaillants.

5.4 Expérimentations

Dans cette section, nous nous intéressons aux résultats obtenus lors de la résolution d'un DisCSP en présence de plusieurs agents défaillants. Nous présentons l'effet de la variation du nombre d'agents défaillants sur la variation du nombre de messages échangés, ainsi que sur le temps CPU de la résolution du DisCSP.

5.4.1 Algorithme de réplication

Les résultats obtenus auparavant concernent l'exécution de l'algorithme de réplication tout au début de la résolution du DisCSP. Par contre, pour évaluer cette méthode, et pour préciser le moment de l'exécution de l'algorithme de réplication, nous supposons que l'*Agent Dispatcher* connait déjà le nombre d'agents défaillants. L'algorithme n'est exécuté que si l'*Agent Dispatcher* reçoit les messages $\langle \mathrm{isFailed} \rangle$ concernant tous les agents défaillants.

TABLE 5.1 – Temps CPU $(10^{-3}sec)$ de distribution des CSPs locaux

Nombre d'agents / Agents défaillants	4	6	8	10	12
Sans Défaillance (anciennes valeurs)	19.33	27.7	57.06	63.31	72.16
1 Défaillance	2.75	2.64	3	3.05	3.55
2 Défaillances	4	5.26	7.06	6.88	9.44
3 Défaillances	4.56	6.42	7.95	10.2	11.5
4 Défaillances	-	8.89	9.9	11.84	13.6
6 Défaillances	-	-	12.9	16.21	25.82
8 Défaillances	-	-	-	-	32.8

Le tableau 5.1 présente le temps CPU obtenu en exécutant l'algorithme de réplication dans un DisCSP de paramètres $\langle m, 4, 4, 0.5 \rangle$ et en faisant varier le nombre d'agents défaillants. Il contient les valeurs du temps CPU obtenues lors de l'exécution de l'algorithme au début de la résolution comme présenté dans le chapitre 4 (sans défaillance), et celles obtenues en exécutant l'algorithme après la détection des défaillances. Selon les

valeurs obtenues, nous remarquons que la 2^{eme} méthode prend moins de temps à répliquer les CSPs locaux. En effet, pour distribuer les CSPs locaux, l'*Agent Dispatcher* n'est pas obligé de parcourir tous les agents, et tous les CSPs. Il parcourt seulement les agents non défaillants, et les CSPs des agents défaillants. De cette façon, nous obtenons des valeurs qui sont largement inférieures à celles obtenues auparavant. Nous remarquons aussi que le temps CPU augmente en augmentant le nombre d'agents défaillants. En effet, plus il y a d'agents défaillants, plus il y a de CSPs à affecter. Donc l'algorithme parcourt les CSPs autant de fois qu'il existe des CSPs non affectés.

5.4.2 Variation du nombre d'agents

Les tableaux 5.2 et 5.3 présentent le nombre de messages échangés et du temps CPU de résolution des DisCSPs, de paramètres $\langle m, 4, 4, 0.5\rangle$, en faisant varier le nombre d'agents. Le tableau 5.2 présente le nombre de messages échangés. Il contient le nombre de messages de résolution définis par Multi-ABT, et le nombre de messages supplémentaires. Ces messages présentent le nombre de messages $\langle \mathrm{CHECK}\rangle$ et $\langle \mathrm{NewCSP}\rangle$.

TABLE 5.2 – Nombre de messages échangés

Agents défaillants \ Nombre d'agents		4	6	8	10	12
1 agent	Multi-ABT	64.35	428.82	1394.75	2506.2	4237.6
	Supplémentaire	40.7	26.7	41.2	68.4	118.9
2 agents	Multi-ABT	52.3	414.15	1262.15	2770.71	5358.4
	Supplémentaire	27.5	21.15	38.45	64.75	105.42
3 agents	Multi-ABT	–	366.25	1163.2	2305.7	4140.17
	Supplémentaire	–	18.25	34.05	51.7	88.64
4 agents	Multi-ABT	-	203.47	868.95	2243.3	5200.55
	Supplémentaire	-	14.73	28.35	51.45	89.1
6 agents	Multi-ABT	-	-	523.05	1661.75	4348.9
	Supplémentaire	-	-	23	35.2	53.4
8 agents	Multi-ABT	-	-	-	1359	3252.44
	Supplémentaire	-	-	-	23.6	39.45

En augmentant le nombre d'agents défaillants, nous remarquons que le nombre de messages de résolution échangés diminue. En effet, le nombre de CSPs fusionnés aug-

mente en augmentant le nombre d'agents défaillants. Cette fusion engendre la diminution du nombre de contraintes inter-agents, d'où la diminution du nombre de voisins de chaque agent délégué avec lesquels il communique.

Nous pouvons observer aussi une diminution au niveau du nombre de messages supplémentaires en augmentant le nombre d'agents défaillants. Puisque le nombre d'agents défaillants augmente, la fréquence de l'envoi des messages $\langle \mathrm{ACTIVE} \rangle$ diminue. Un agent non défaillant peut recevoir un message $\langle \mathrm{CHECK} \rangle$ plusieurs fois de la part d'un même agent. Par contre, un agent défaillant reçoit un message $\langle \mathrm{CHECK} \rangle$ une seule fois de la part du même agent, ce qui explique la diminution des messages échangés $\langle \mathrm{CHECK} \rangle$.

TABLE 5.3 – Temps CPU (sec) de résolution de DisCSP

Agents défaillants \ Nombre d'agents		4	6	8	10	12
1 agent	CPU Total	21.44	23.53	27.24	32.65	41.2
	CPU supplémentaire	1.2	1.32	2.21	4.14	5.35
2 agents	CPU Total	29.35	27.75	36.37	30.01	34.69
	CPU supplémentaire	1.32	1.87	2.78	2.91	5.81
3 agents	CPU Total	–	35.14	37.94	39.07	47.85
	CPU supplémentaire	–	2.82	2.93	3.51	4.71
4 agents	CPU Total	-	42.24	38.97	43.29	53.45
	CPU supplémentaire	-	3.12	3.4	3.56	6.29
6 agents	CPU Total	-	-	59.97	56.88	73.59
	CPU supplémentaire	-	-	4.34	3.93	8.05
8 agents	CPU Total	-	-	-	36.12	91.18
	CPU supplémentaire	-	-	-	7.86	11.49

Le tableau 5.3 présente la variation du temps CPU en faisant varier le nombre d'agents défaillants. Il contient le temps CPU total de la résolution du DisCSP, et le temps CPU supplémentaire de détection et de traitement des défaillances. Nous observons qu'en augmentant le nombre d'agents défaillants, le temps CPU supplémentaire augmente. La détection de plusieurs défaillances nécessite un échange et un temps d'attente supplémentaires. L'augmentation de ce temps CPU, et la reprise de la résolution après la détection de défaillance, engendre l'augmentation du temps CPU total.

Les résultats obtenus montrent que les processus de détection et de traitement de plu-

sieurs défaillances produisent une augmentation au niveau du temps CPU, mais le temps CPU supplémentaire reste négligeable par rapport au temps CPU de résolution du DisCSP.

5.4.3 Variation du nombre de variables

Les tableaux 5.4 et 5.5 présentent le nombre de messages et le temps CPU de résolution des DisCSPs de paramètres $\langle 4, n, 4, 0.5\rangle$. Le nombre d'agents défaillants est limité à 2 agents pour éviter la centralisation du problème. En effet, si nous considérons 3 agents défaillants, le DisCSP sera résolu par un seul agent, et ne sera plus distribué. Avec un nombre de 10 variables par agent, un problème d'espace mémoire survient.

TABLE 5.4 – Nombre de messages échangés

Agents défaillants \ Nombre de variables		2	4	6	8
1 agent	Multi-ABT	73.55	64.35	50	61.3
	Supplémentaire	65.65	40.7	9.6	10.55
2 agents	Multi-ABT	41.1	52.3	26.25	37.1
	Supplémentaire	24.15	27.5	7.85	7.95

Le tableau 5.4 présente le nombre de messages échangés dans le cas d'une et de 2 défaillances. Nous remarquons qu'en augmentant le nombre de variables et le nombre d'agents défaillants, le nombre de messages échangés diminue. Plus le nombre d'agents défaillants augmente, plus le nombre de contraintes inter-agents diminue, d'où la diminution du nombre de messages échangés entre les agents.

TABLE 5.5 – Temps CPU (sec) de résolution de DisCSP

Agents défaillants \ Nombre de variables		2	4	6	8
1 agent	CPU Total	21.65	21.44	19.28	13.47
	CPU supplémentaire	1.31	1.2	0.97	1.14
2 agents	CPU Total	46.6	29.35	24.72	23.9
	CPU supplémentaire	1.75	1.32	1.09	1.38

Au niveau du temps CPU présenté dans le tableau 5.5, nous remarquons une augmentation au niveau du temps CPU en augmentant le nombre d'agents défaillants. Plus

le nombre d'agents défaillants augmente, plus le temps de détection et de traitement des défaillances augmente. Nous remarquons aussi qu'en augmentant le nombre de variables, le temps CPU diminue : lors de la reprise de la résolution, un agent délégué peut ne pas trouver de solution locale à proposer comme 1^{ere} solution à son nouveau CSP local. Dans ce cas, il informe les autres agents que le DisCSP ne possède pas de solution. Ainsi, les autres agents s'arrêtent et ne continuent pas la résolution.

5.5 Conclusion

Dans ce chapitre, nous avons présenté une adaptation de notre approche pour résoudre un DisCSP en présence de plusieurs agents défaillants. L'approche proposée précédemment ne traitait qu'une seule défaillance. Cela est dû au manque de transmission de certaines informations, et de traitement de certains cas particuliers.

Pour traiter plusieurs défaillances, nous avons gardé le même principe de l'approche de base, mais en transmettant le message $\langle \mathrm{NewCSP} \rangle$ à tous les agents du DisCSP, et en répliquant seulement les CSPs locaux des agents défaillants. Les résultats obtenus ont montré que grâce à cette adaptation, notre approche est capable de résoudre un DisCSP en présence de plusieurs agents défaillants. La variation du temps CPU et du nombre de messages échangés dûe à cette adaptation reste négligeable par rapport aux mesures totales de la résolution.

L'absence d'un ensemble d'agents peut provoquer la modification de la modélisation du DisCSP initial. Mais grâce à la fusion des CSPs locaux, notre approche est capable de conserver l'aspect distribué de la modélisation initiale. Malgré son efficacité, notre approche reste incapable de résoudre certains cas en présence de plusieurs agents défaillants, tel que le cas où un agent et tous ses voisins sont défaillants. Dans ce cas, la défaillance ne sera pas détectée.

Conclusion et Perspectives

Dans cette thèse, nous nous sommes intéressés à la résolution des DisCSPs en présence de plusieurs agents défaillants. L'approche proposée est basée sur une technique de réplication. L'objectif initial était de trouver une solution à un DisCSP s'il en existe même en cas de défaillance de plusieurs agents.

Le premier chapitre a présenté le contexte de l'approche proposée. Cette dernière appartient aux domaines des SMAs et du raisonnement par contraintes. En effet, un DisCSP est résolu par un ensemble d'agents qui communiquent ensemble selon un algorithme de résolution afin de satisfaire les contraintes du problème. Ce chapitre a défini aussi l'algorithme de résolution Multi-ABT en détaillant son comportement. L'utilisation de cet algorithme a montré qu'il n'est pas capable de résoudre un DisCSP en présence d'un agent défaillant.

L'état de l'art présenté dans le deuxième chapitre a montré qu'il existe plusieurs méthodes de tolérance aux fautes. Il a défini des méthodes de détection et de traitement des défaillances au sein d'un système distribué en général, et d'un SMA en particulier. Les méthodes proposées ont engendré l'apparition de deux modèles d'interaction entre les détecteurs de défaillance et les composants du système. Nous avons utilisé le modèle *push* pour détecter la défaillance d'un agent. Ces méthodes ont montré aussi que la réplication est la base de la plupart des travaux existants, mais aucune d'elles n'a été appliquée pour résoudre un DisCSP en présence d'agents défaillants.

Dans le troisième chapitre, nous avons proposé notre approche destinée à résoudre un DisCSP en présence d'agents défaillants. Elle est basée sur la réplication des CSPs locaux : si un agent est défaillant, un autre agent prend en charge la résolution de son CSP

local. Cette prise en charge est effectuée en fusionnant les réplicats des CSPs des agents défaillants avec des CSPs d'autres agents. La fusion permet de conserver la formalisation initiale du problème, et garantit la résolution des CSPs des agents défaillants.

Le quatrième chapitre a contenu la validation de notre approche. Pour ce faire, nous avons simulé les défaillances des agents pour résoudre des DisCSPs en variant le nombre d'agents et le nombre de variables. Les résultats obtenus ont montré qu'en présence d'un agent défaillant, toutes les instances fournissent les résultats attendus. Les expérimentations ont montré que notre approche est coûteuse au niveau du temps CPU, mais le temps supplémentaire dédié à la détection et au traitement de la défaillance reste négligeable par rapport au temps CPU total.

Dans le cinquième chapitre, nous avons amélioré notre approche afin de traiter plusieurs défaillances. A partir des résultats obtenus du chapitre 4, nous avons extrait les limites de l'approche. L'amélioration consiste à transmettre à tous les agents les identifiants des agents défaillants ainsi que les identifiants de leurs délégués. De cette façon, tous les agents connaissent l'agent délégué de chaque agent défaillant. L'amélioration consiste aussi à répliquer seulement les agents défaillants. Ainsi, les agents peuvent remplacer tous leurs voisins défaillants par leurs agents délégués. Cette adaptation nous a permis de résoudre un DisCSP quelque soit le nombre d'agents défaillants.

Perspectives

Dans ce travail, malgré le coût élevé de l'approche proposée, nous avons réussi à résoudre des DisCSPs en présence d'agents défaillants : tous les agents défaillants ont été détectés. Mais si un agent et tous ses voisins sont défaillants, l'agent défaillant en question ne sera pas détecté. Dans ce cas, notre méthode de détection de défaillance n'est plus efficace. Alors, il faut envisager plusieurs améliorations pour généraliser notre approche. Pour garantir la détection de tous les agents défaillants quelque soit l'état de leurs voisins, un agent particulier dédié à la détection des défaillances peut être conçu. Cet agent peut communiquer avec tous les agents en échangeant des messages de vérification d'activité, et il sera chargé d'informer les agents non défaillants de la présence d'agents défaillants. En cas d'interruption temporaire de la communication entre un agent et ses voisins alors qu'il est actif, notre méthode de détection considère cet agent comme dé-

faillant. Par contre, en utilisant un agent particulier de détection, un agent actif ne risque pas d'être considéré comme défaillant.

Dans ce travail, nous avons adapté la technique de la réplication des CSPs locaux à l'algorithme de résolution Multi-ABT. Cette technique peut être adaptée à d'autres algorithmes tel que Multi-AWC. Pour cet algorithme, l'ordre de priorité des variables est dynamique. Alors, lors de la réplication, le dernier ordre de priorité affecté aux variables n'est pas conservé. Pour répliquer le dernier ordre de priorité utilisé, chaque agent peut communiquer à l'*Agent Dispatcher* l'ordre de ses variables à chaque modification. Dans ce cas, un grand nombre de messages entre les agents et l'*Agent Dispatcher* sera échangé. Afin d'éviter cet échange de messages, une autre méthode peut être proposée, mais qui ne conserve pas le dernier ordre de priorité. Après la fusion, les variables initiales de l'agent délégué peuvent conserver leur dernier ordre de priorité. Par contre, les variables répliquées reprennent les valeurs initiales de leur ordre.

Malgré son efficacité, l'application de la réplication ne respecte pas la privacité des informations des agents (la volonté d'un agent de préserver ses informations confidentielles) [Savaux, 2017]. En effet, pour chaque CSP local, il y a un autre agent différent du sien qui connait ses caractéristiques. Les travaux de [Savaux et al., 2016] proposent d'introduire des coûts relatifs à la privacité des contraintes des CSPs des agents. Ils utilisent ces coûts en modifiant la recherche de solution afin de diminuer la perte de privacité, et supposent qu'une solution ne satisfait pas forcement toutes les contraintes. En se basant sur ce principe lors de la réplication, l'*Agent Dispatcher* peut répliquer les contraintes ayant un coût de privacité minimal. De cette façon, les agents défaillants conservent la privacité des contraintes de leurs CSPs même après la fusion. Dans ce cas, il faut aussi supposer qu'une solution peut ne pas satisfaire toutes les contraintes.

La confiance et la réputation entre les agents peut aussi intervenir lors de la réplication des CSPs locaux [Huynh et al., 2006, Vu et al., 2009]. Un agent peut refuser de prendre en charge le CSP d'un autre agent s'il n'a pas confiance en lui, ou un agent peut déléguer son CSP dès le début de la résolution à l'agent en qui il a le plus confiance. Ce degré de confiance peut être établi à partir du type des messages reçus de la part d'un voisin, par exemple, si un agent reçoit plusieurs fois un message de backtracking de la part du même voisin, le degré de confiance entre ces deux agents diminue.

Annexe A

Algorithmes de résolution

A.1 AWC/ Multi-AWC

L'ordre de priorité des agents dans l'algorithme Asynchronous Weak-Commitment (AWC) [Yokoo, 1995] est dynamique. Tous les agents possèdent un ordre de priorité qui est égale à zéro. AWC utilise les mêmes messages de résolution que ABT : $\langle \mathrm{OK?} \rangle$ pour la transmission des solutions locales, et $\langle \mathrm{BT} \rangle$ pour les backtracking. Pour prendre en considération l'évolution de l'ordre de priorité, le message $\langle \mathrm{OK?} \rangle$ prend la forme $\langle$ *variable, valeur, priorité de l'émetteur* $\rangle$. Ce triplet est enregistré dans l'$\mathrm{AgentView}$ de l'agent receveur du message. Une instanciation est cohérente si la valeur courante de l'agent vérifie les contraintes qui le lient à ses accointances supérieures. Mais si un agent n'arrive pas à trouver une valeur consistante, sa priorité change et devient la plus élevée de ses voisins ($priorite \leftarrow prioriteMax + 1$). AWC a été étendu à Multi-AWC [Yokoo and Hirayama, 1998] pour résoudre des DisCSPs multi-variables. Dans cet algorithme, nous parlons de priorité au niveau des variables de chaque agent et non pas au niveau des agents. Un agent peut avoir des variables de priorités supérieures et inférieures à celles d'un autre agent. Au cours de la résolution, un agent change au fur et à mesure les valeurs de ses variables. Il commence par la variable de plus haute priorité et l'instancie afin de trouver une valeur qui satisfait les contraintes liées aux variables de priorité supérieure. Si aucune valeur n'est cohérente, l'agent augmente la priorité de cette variable pour qu'elle ait une priorité plus élevée que celle de ses variables voisines. Ensuite, l'agent passe à la variable d'après (selon l'ordre de priorité). Le message $\langle \mathrm{OK?} \rangle$ est présenté sous

la forme ⟨ *émetteur, variable, valeur, priorité la plus élevée des variables* ⟩. Ces étapes sont répétées par l'agent jusqu'à ne plus avoir de variables violant de contraintes avec les variables voisines supérieures. Les messages sont échangés au fur et à mesure que l'agent modifie les valeurs de ses variables.

A.2 DBS/ Multi-DBS

Distributed Backtracking with Session (DBS) [Doniec et al., 2005] a été conçu pour résoudre les DisCSPs mono-variables et a été étendu à Multi-DBS [Mandiau et al., 2014] pour résoudre les DisCSPs multi-variables. Le but de cet algorithme est de garder une trace des recherches effectuées afin de déterminer la terminaison de la résolution. DBS utilise des sessions, attribuées aux agents et initialisées à 0, pour ne pas traiter les messages devenus obsolètes. La valeur de la session est incrémentée à chaque changement d'une instanciation lors de le réception d'un message $\langle \mathrm{OK?} \rangle$ qui prend la forme ⟨ *émetteur, variable, valeur, session* ⟩. Le message $\langle \mathrm{BT} \rangle$ prend la forme ⟨ émetteur, variable, valeur, session, liste des backtracking⟩. Quand un agent reçoit un $\langle \mathrm{BT} \rangle$, il ne le traite que si la valeur de la session du message est la même que la sienne.

A.3 AFC

L'algorithme Asynchronous Forward-Checking (AFC) [Meisels and Zivan, 2007] est conçu pour résoudre des DisCSPs multi-variables. Les solutions au niveau de cet algorithme sont cherchées d'une manière synchrone tout en gardant l'aspect asynchrone lors de la propagation des instanciations (Forward Checking).

Dans cet algorithme, les agents utilisent un message $\langle \mathrm{CPA} \rangle$ (Current Partial Assignment) pour envoyer leurs solutions locales à leurs voisins. Les agents recevant ce message cherchent une solution cohérente avec le contenu du message $\langle \mathrm{CPA} \rangle$, et l'envoient à leurs accointances inférieures. Contrairement aux autres algorithmes, les agents ne transmettent leurs solutions que s'ils reçoivent au moins un message $\langle \mathrm{CPA} \rangle$ de la part de l'un de ses accointances supérieures. Le but est d'étendre le contenu de ce message en une solution globale. C'est la partie synchrone de l'algorithme.

En envoyant un message $\langle \mathrm{CPA} \rangle$, chaque agent envoie une copie de ce message dans un autre message $\langle \mathrm{FC\_CPA} \rangle$ aux agents n'appartenant pas au $\langle \mathrm{CPA} \rangle$. En recevant cette copie, chaque agent met à jour les domaines de ses variables en éliminant, du domaine, les valeurs non consistantes avec le contenu du message $\langle \mathrm{FC\_CPA} \rangle$ reçu. C'est la partie asynchrone de l'algorithme. Si l'un des agents ne trouve pas de solution cohérente avec la proposition reçue, ou s'il se trouve avec un domaine vide (après la mise à jour des valeurs du domaine), il envoie un message $\langle \mathrm{Not\_OK} \rangle$ à l'agent appartenant au $\langle \mathrm{CPA} \rangle$ ou $\langle \mathrm{FC\_CPA} \rangle$ et ayant la plus petite priorité. A la fin de l'algorithme, si un $\langle \mathrm{CPA} \rangle$ contient toutes les variables du DisCSP, alors il existe une solution. Sinon, l'un des agents détecte qu'il n'existe pas de solution et arrête la résolution du problème.

A.4 Comparaison des algorithmes

Le tableau A.1 présente une comparaison entre les algorithmes de résolution. Pour cette comparaison, nous avons spécifié pour chaque algorithme :

- L'ordre de priorité adapté lors de son exécution
- La présence d'une création de liens entre des agents qui ne sont pas voisins
- Le type des messages échangés pour l'échange des solutions entre les agents
- Les structures de données utilisées durant la résolution

Pour résoudre un DisCSP, c'est l'ordre de priorité entre les agents qui décide le sens d'envoi des messages. Dans le cas mono-variable, l'ordre de priorité concerne les agents, par contre, dans le cas multi-variables, l'ordre de priorité concerne les agents et les variables. Dans ABT/Multi-ABT, DBS/Multi-DBS et AFC, cet ordre est statique. Contrairement à AWC, où l'ordre des agents est dynamique, et Multi-AWC où l'ordre de priorité des variables est dynamique afin de détecter rapidement la cause de l'inconsistance des valeurs proposées en évitant des vérifications inutiles de contraintes.

Durant la résolution d'un DisCSP, certains agents reçoivent des messages de backtrack contenant des agents qui n'appartiennent pas à leurs listes de voisins. Dans les algorithmes ABT, AWC et DBS, ces agents envoient aux autres agents une demande de création de lien

TABLE A.1 – Comparaison des algorithmes

Algorithmes	Ordre de priorité	Création de liens	Nogood	Type de messages	
				$\langle OK? \rangle$	$\langle BT \rangle$
ABT/Multi-ABT	Statique	oui	oui	oui	oui
AWC/Multi-AWC	Dynamique	oui	oui	oui	oui
DBS/Multi-DBS	Statique	oui	non	oui	oui
AFC	Statique	non	oui	$\langle \mathrm{CPA} \rangle$	$\langle \mathrm{Not\_OK} \rangle$

afin qu'ils reçoivent par la suite les valeurs de leurs instanciations. Par contre, l'algorithme AFC considère que chaque agent connait les autres agents et ne nécessite pas une demande de création de lien.

Chaque agent enregistre les solutions locales proposées par ses voisins dans une structure de données $\mathrm{AgentView}$ quelque soit l'algorithme de résolution utilisé. Mais au niveau des messages de backtrack, l'algorithme DBS/Multi-DBS est le seul qui n'utilise pas une structure de données Nogood. Il remplace cette structure par l'utilisation des sessions afin de distinguer les messages de backtrack à traiter de ceux qu'il ne faut pas traiter.

Les algorithmes de résolution, sauf AFC, utilisent le message $\langle \mathrm{OK?} \rangle$ pour échanger des solutions locales, et $\langle \mathrm{BT} \rangle$ pour demander de changer une solution proposée. Les messages $\langle \mathrm{OK?} \rangle$ sont envoyés d'une manière asynchrone. L'algorithme AFC utilise un message $\langle \mathrm{CPA} \rangle$ pour propager une solution locale à une solution globale, et un message $\langle \mathrm{Not\_OK} \rangle$ pour demander la modification d'une solution locale reçue. Les messages $\langle \mathrm{CPA} \rangle$ sont échangés d'une manière synchrone : un agent n'envoie de message $\langle \mathrm{CPA} \rangle$ qu'en recevant un message $\langle \mathrm{CPA} \rangle$ de la part d'un autre agent.

Bibliographie

[Almeida et al., 2007] Almeida, A., Briot, J.-P., Aknine, S., Guessoum, Z., and Marin, O. (2007). Towards autonomic fault-tolerant multi-agent systems. In *The 2nd Latin American Autonomic Computing Symposium (LAACS'2007), Petropolis, RJ, Brésil*.

[Almeida et al., 2006] Almeida, A. L., Aknine, S., Briot, J.-P., and Malenfant, J. (2006). Plan-based replication for fault-tolerant multi-agent systems. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pages 7–pp. IEEE.

[Amalberti and Deblon, 1992] Amalberti, R. and Deblon, F. (1992). Cognitive modelling of fighter aircraft process control : a step towards an intelligent on-board assistance system. *International Journal of Man-Machine Studies*, 36 :639–671.

[Arlat et al., 2006] Arlat, J., Crouzet, Y., Deswarte, Y., Fabre, J.-C., Laprie, J.-C., and Powell, D. (2006). Tolérance aux fautes. *Encyclopédie de l'informatique et des systèmes d'information. Vuibert, Paris, France*, 92.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1) :11–33.

[Bellifemine et al., 2002] Bellifemine, F., Caire, G., Trucco, T., and Rimassa, G. (2002). Jade programmer's guide. *Jade version*, 3 :13–39.

[Bessière et al., 2001] Bessière, C., Maestre, A., and Meseguer, P. (2001). Distributed dynamic backtracking. *CP*, 2239 :772.

[Briot and Demazeau, 2001] Briot, J.-P. and Demazeau, Y. (2001). *Principes et architectures des systèmes multi-Agents*. Hermès-Lavoisier.

[Caglayan et al., 1998] Caglayan, A., Harrison, C., Gaïti, D., and Joly, R. (1998). *Les agents : applications bureautiques, Internet et intranet*. InterEditions.

[Chaib-Draa et al., 1992] Chaib-Draa, B., Moulin, B., Mandiau, R., and Millot, P. (1992). Trends in distributed artificial intelligence. *Artificial Intelligence Review*, 6(1) :35–66.

[Chakchouk et al., 2018] Chakchouk, F., Piechowiak, S., Mandiau, R., Vion, J., Soui, M., and Ghedira, K. (2018). Fault tolerance in discsps : Several failures case. In *Distributed Computing and Artificial Intelligence*, pages 204–212. Springer.

[Chakchouk et al., 2017] Chakchouk, F., Vion, J., Piechowiak, S., Mandiau, R., Soui, M., and Ghedira, K. (2017). Replication in fault-tolerant distributed CSP. IEA/AIE 2017, Springer.

[Climent et al., 2010] Climent, L., Salido, M., and Barber, F. (2010). Robust solutions in changing constraint satisfaction problems. In *Trends in Applied Intelligent Systems*, pages 752–761.

[Climent et al., 2013] Climent, L., Wallace, R., Salido, M., and F.Barber (2013). Finding robust solutions for constraint satisfaction problems with discrete and ordered domains by coverings. *Artificial Intelligence Review*.

[Coulouris et al., 2011] Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems : Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition.

[Dechter and Dechter, 1988] Dechter, R. and Dechter, A. (1988). *Belief maintenance in dynamic constraint networks*. University of California, Computer Science Department.

[Dellarocas and Klein, 2000] Dellarocas, C. and Klein, M. (2000). An experimental evaluation of domain-independent fault handling services in open multi-agent systems. In *MultiAgent Systems, 2000. Proceedings. Fourth International Conference on*, pages 95–102. IEEE.

[Demazeau and Müller, 1991] Demazeau, Y. and Müller, J.-P. (1991). *Decentralized AI, 2*. Elsevier.

[Doniec et al., 2005] Doniec, A., Piechowiak, S., and Mandiau, R. (2005). A discsp solving algorithm based on sessions. In *Proceedings of the Eighteenth International Florida Artificial Intelligence Research Society Conference*, pages 666–670.

[Fedoruk and Deters, 2002] Fedoruk, A. and Deters, R. (2002). Improving fault-tolerance by replicating agents. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems : Part 2*, pages 737–744. ACM.

[Felber et al., 1999] Felber, P., Défago, X., Guerraoui, R., and Oser, P. (1999). Failure detectors as first class objects. In *Distributed Objects and Applications, 1999. Proceedings of the International Symposium on*, pages 132–141. IEEE.

[Ferber, 1995] Ferber, J. (1995). *Les Systèmes multi-agents : vers une intelligence collective*. I.I.A. Informatique intelligence artificielle. InterEditions.

[Ferber and Drogoul, 1992] Ferber, J. and Drogoul, A. (1992). Using reactive multi-agent systems in simulation and problem solving. *Distributed artificial intelligence : Theory and praxis*, 5 :53–80.

[Fetzer et al., 2001] Fetzer, C., Raynal, M., and Tronel, F. (2001). An adaptive failure detection protocol. In *Dependable Computing, 2001. Proceedings. 2001 Pacific Rim International Symposium on*, pages 146–153. IEEE.

[Frécon and Kazar, 2009] Frécon, L. and Kazar, O. (2009). *Manuel d'intelligence artificielle*. PPUR Presses polytechniques.

[Frey et al., 2003] Frey, D., Nimis, J., Wörn, H., and Lockemann, P. (2003). Benchmarking and robust multi-agent-based production planning and control. *Engineering Applications of Artificial Intelligence*, 16(4) :307–320.

[Gärtner, 1999] Gärtner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1) :1–26.

[Goyal and Yadav, 2011] Goyal, H. and Yadav, S. (2011). Multi-agent distributed artificial intelligence. *International Journal of Soft Computing and Engineering*, 1 :15–18.

[Guessoum et al., 2004] Guessoum, Z., Briot, J.-P., and Faci, N. (2004). Towards fault-tolerant massively multiagent systems. In *International Workshop on Massively Multiagent Systems*, pages 55–69. Springer.

[Hägg, 1997] Hägg, S. (1997). *A sentinel approach to fault handling in multi-agent systems*, pages 181–195. Springer Berlin Heidelberg.

[Hayashibara et al., 2002] Hayashibara, N., Cherif, A., and Katayama, T. (2002). Failure detectors for large-scale distributed systems. In *Reliable Distributed Systems, 2002. Proceedings. 21st IEEE Symposium on*, pages 404–409. IEEE.

[Hebrard, 2004] Hebrard, E. (2004). Super solutions in constraint programming. In *Contributions to the Doctoral Programme of the 2nd International Joint Conference on Automated Reasoning*.

[Hebrard et al., 2004] Hebrard, E., Hnich, B., and Walsh, T. (2004). Robust solutions for constraint satisfaction and optimization. In *Proceedings of the 9th National Conference on AI, 16th Conference on IAAI*, pages 952–953.

[Hirayama et al., 2000] Hirayama, K., Yokoo, M., and Sycara, K. (2000). The phase transition in distributed constraint satisfaction problems : First results. In *Principles and Practice of Constraint Programming*, pages 515–519.

[Hirayama et al., 2004] Hirayama, K., Yokoo, M., and Sycara, K. (2004). An easy-hard-easy cost profile in distributed constraint satisfaction. *Information Processing Society of Japan journal*, 45(9) :2217–2225.

[Huhns, 2012] Huhns, M. N. (2012). *Distributed artificial intelligence*, volume 1. Elsevier.

[Huynh et al., 2006] Huynh, T., Jennings, N., and Shadbolt, N. (2006). An integrated trust and reputation model for open multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 13(2) :119–154.

[Khan et al., 2005] Khan, Z. A., Shahid, S., Ahmad, H. F., Ali, A., and Suguri, H. (2005). Decentralized architecture for fault tolerant multi agent system. In *Autonomous Decentralized Systems, 2005. ISADS 2005. Proceedings*, pages 167–174. IEEE.

[Klein and Dellarocas, 1999] Klein, M. and Dellarocas, C. (1999). Exception handling in agent systems. In *Proceedings of the third annual conference on Autonomous Agents*, pages 62–68. ACM.

[Kshemkalyani and Singhal, 2011] Kshemkalyani, A. D. and Singhal, M. (2011). *Distributed computing : principles, algorithms, and systems*. Cambridge University Press.

[Kumar and Cohen, 2000] Kumar, S. and Cohen, P. R. (2000). Towards a fault-tolerant multi-agent system architecture. In *Proceedings of the fourth international conference on Autonomous agents*, pages 459–466. ACM.

[Kumar et al., 2000] Kumar, S., Cohen, P. R., and Levesque, H. J. (2000). The adaptive agent architecture : Achieving fault-tolerance using persistent broker teams. In *MultiAgent Systems*, pages 159–166. IEEE.

[Labidi and Lejouad, 1993] Labidi, S. and Lejouad, W. (1993). *De l'intelligence artificielle distribuée aux systèmes multi-agents*. PhD thesis, INRIA.

[Luger, 2005] Luger, G. F. (2005). *Artificial intelligence : structures and strategies for complex problem solving*. Pearson education.

[Lussier et al., 2004] Lussier, B., Chatila, R., Ingrand, F., Killijian, M.-O., and Powell, D. (2004). On fault tolerance and robustness in autonomous systems. In *Proceedings of the 3rd IARP-IEEE/RAS-EURON*, pages 351–358.

[Mandiau et al., 2014] Mandiau, R., Vion, J., Piechowiak, S., and Monier, P. (2014). Multi-variable distributed backtracking with sessions. *Appl. Intell.*, 41(3) :736–758.

[McCorduck et al., 1977] McCorduck, P., Minsky, M., Selfridge, O. G., and Simon, H. A. (1977). History of artificial intelligence. In *IJCAI*, pages 951–954.

[Meisels and Zivan, 2007] Meisels, A. and Zivan, R. (2007). Asynchronous forward-checking for discsps. *Constraints*, 12(1) :131–150.

[Monier, 2012] Monier, P. (2012). *DBS multi-variables pour des problèmes de coordination multi-agents*. PhD thesis, Université de Valenciennes.

[Nagi, 2001] Nagi, K. (2001). *Transactional agents : towards a robust multi-agent system*, volume 2249. Springer Science & Business Media.

[Nasreen et al., 2016] Nasreen, M., Ganesh, A., and Sunitha, C. (2016). A study on byzantine fault tolerance methods in distributed networks. *Procedia Computer Science*, 87 :50–54.

[Paraschiv, 2004] Paraschiv, C. (2004). *Les agents intelligents pour un nouveau commerce électronique*. Hermès science publ.

[Perumalla and Park, 2014] Perumalla, K. S. and Park, A. J. (2014). Reverse computation for rollback-based fault tolerance in large parallel systems. *Cluster Computing*, 17(2) :303–313.

[Poole and Mackworth, 2010] Poole, D. L. and Mackworth, A. K. (2010). *Artificial Intelligence : foundations of computational agents*. Cambridge University Press.

[Roosta, 2000] Roosta, S. H. (2000). Artificial intelligence and parallel processing. In *Parallel Processing and Parallel Algorithms*, pages 501–534. Springer.

[Russell and Norvig, 2010] Russell, S. and Norvig, P. (2010). *Intelligence artificielle : Avec plus de 500 exercices*. Pearson Education France.

[Savaux, 2017] Savaux, J. (2017). *Privacité dans les problèmes distribués contraints pour agents basés utilité*. PhD thesis, Valenciennes Univ, France.

[Savaux et al., 2016] Savaux, J., Vion, J., Piechowiak, S., Mandiau, R., Matsui, T., Hirayama, K., Yokoo, M., and Silaghi, M. (2016). Privacité dans les discsp pour agents utilitaires. In *Systèmes Multi-Agents et simulation - Vingt-quatrièmes journées francophones sur les systèmes multi-agents, JFSMA 16, Saint-Martin-du-Vivier (Rouen), France, Octobre 5-7, 2016.*, pages 129–138.

[Shoham and Leyton-Brown, 2008] Shoham, Y. and Leyton-Brown, K. (2008). *Multiagent systems : Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press.

[Stanković et al., 2017] Stanković, R., Štula, M., and Maras, J. (2017). Evaluating fault tolerance approaches in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 31(1) :151–177.

[Stelling et al., 1999] Stelling, P., DeMatteis, C., Foster, I., Kesselman, C., Lee, C., and von Laszewski, G. (1999). A fault detection service for wide area distributed computations. *Cluster Computing*, 2(2) :117–128.

[Tanenbaum and Steen, 2006] Tanenbaum, A. and Steen, M. (2006). *Distributed Systems : Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc.

[Vu et al., 2009] Vu, Q. N., Gaudou, B., Canal, R., and Hassas, S. (2009). Coherence and Robustness in a Disturbed MAS. In *2009 IEEE-RIVF International Conference on Computing and Communication Technologies*, pages 1–4. Ieee.

[Wallace and Freuder, 1998] Wallace, R. and Freuder, E. (1998). Stable solutions for dynamic constraint satisfaction problems. In *Principles and Practice of Constraint Programming ,Italy*, pages 447–461.

[Weiss, 1999] Weiss, G. (1999). *Multiagent systems : a modern approach to distributed artificial intelligence*. MIT press.

[Wiesmann et al., 1999] Wiesmann, M., Pedone, F., and Schiper, A. (1999). A systematic classification of replicated database protocols based on atomic broadcast. In *Proceedings of the 3rd ERSADS" 99*, number LSR-CONF-1999-009.

[Xu et al., 2007] Xu, K., Boussemart, F., Hemery, F., and Lecoutre, C. (2007). Random constraint satisfaction : Easy generation of hard (satisfiable) instances. *Artif. Intell.*, 171(8-9) :514–534.

[Yeoh and Yokoo, 2012] Yeoh, W. and Yokoo, M. (2012). Distributed problem solving. *AI Magazine*, 33(3) :53.

[Yokoo, 1995] Yokoo, M. (1995). Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Principles and Practice of Constraint Programming - CP'95s*, pages 88–102.

[Yokoo, 2001] Yokoo, M. (2001). Distributed constraint satisfaction : foundations of cooperation in multi-agent systems.

[Yokoo et al., 1992] Yokoo, M., Durfee, E., Ishida, T., and Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the 12th International Conference on DCS, Japan*, pages 614–621.

[Yokoo and Hirayama, 1998] Yokoo, M. and Hirayama, K. (1998). Distributed constraint satisfaction algorithm for complex local problems. In *Proceedings of the Third International Conference on Multiagent Systems*, pages 372–381.

Résumé

Nous visons à garantir la résolution d'un DisCSP en présence d'un ou plusieurs agents défaillants. Les méthodes traitant la tolérance aux fautes au sein des SMAs visent la continuité du fonctionnement du système. Mais, aucune de ces méthodes n'est appliquée pour résoudre un DisCSP. La défaillance d'un agent au cours de la résolution d'un DisCSP engendre la perte d'une partie du DisCSP global, d'où l'obtention d'un résultat erroné. Donc pour obtenir les résultats attendus, il faut garantir la résolution du CSP local de l'agent défaillant. Nous proposons de répliquer les CSPs locaux des agents défaillants au sein des agents non défaillants. Cette réplication permet la résolution du CSP local de l'agent défaillant par un autre agent. Cette résolution est effectuée en fusionnant les réplicats de CSPs des agents défaillants avec les CSPs des autres agents. Cette fusion permet la conservation de la modélisation initiale du DisCSP. L'algorithme de distribution des réplicats proposé garantit que les CSPs des agents défaillants ne soient pas répliqués au sein du même agent. De cette façon, le problème conserve son aspect distribué.

Mots-clés: Robustesse, Tolérance aux fautes, Systèmes Multi-agents, Agent défaillant

Abstract

We aim to ensure a DisCSP resolution in presence of failed agents. Methods handling fault tolerance in MASs aim to ensure the continuity of the system operation. But, none of these methods are applied to solve a DisCSP. The failure of an agent generates the loss of a part of the DisCSP providing wrong results. Therefore, to obtain expected results, it is necessary to ensure the resolution of the failed agent local CSP. We propose to replicate the local CSPs of the failed agents within active agents. This replication allows local CSP resolution of the failed agent by another agent. The resolution is done by merging the replicates of failed agents CSPs with the CSPs of other agents. This technique conserve the initial DisCSP modeling. The proposed replicates distribution algorithm ensures that the CSPs of failed agents are not replicated within the same agent. In this way, the problem keeps its distributed aspect.

Keywords: Robustness, Fault tolerance, Multi-agent Systems, Failed agent