Replication in fault-tolerant distributed CSP

Fadoua Chakchouk[1,2], Sylvain Piechowiak[1], René Mandiau[1], Julien Vion[1], Makram Soui[2], and Khaled Ghedira[2]

[1] LAMIH UMR CNRS 8201, University of Valenciennes, France
[2] ENSI, University of Manouba, Tunisia

Abstract. Distributed CSPs present distributed NP-complete problems which need to satisfy constraints to be solved. To solve such problems, many algorithms based on multi-agent systems have proposed in the literature, which may give wrong results if an agent dies. This paper proposes an approach which handles a failure of one agent to guarantee the accuracy of final results. To solve the DisCSPs, we use Multi-ABT as an algorithm of resolution. Our approach is based on a duplication principle; each local CSP is copied in another agent. Then, if one of them fails, its local CSP is supported by the one which owns the copy of its CSP. Experiments results show that our approach gives the same results obtained if a DisCSP is solved by Multi-ABT without failure.

1 Introduction

For real industrial applications, we must keep the robustness of our approach. The robustness is also a critical problem for multi-agent systems (MAS), and in particular to solve constraints satisfaction problems (DisCSP). To solve DisCSPs, many algorithms based on agents interaction have been proposed in the literature: each agent encapsulates a sub-problem from the global CSP, it solves it, then, agents interact with each other to find a global solution. We are convinced that the DisCSP approach facilitate the Knowledge representation of the application, and they already investigate varied problems such as timetabling/meeting problems, road traffic, multi-robot exploration [8]. For example, Solotorevsky and Gudes [10] present a nurses timetabling and transportation problem in a hospital. This problem aims to manage the rents transportation services of the hospital while respecting the timetabling of each department nurses.

In the literature, a failed system was defined by Tanenbaum et al [11] as a system which ”cannot meet its promises”. To detect an entity failure, authors propose to send checking activity message to agents of the system, to which the sender waits for response. Also, Dunagan et al [2] propose a failure notification service if an entity fails.

A failure in a DisCSP such as a death of an agent produces either a partial solution, or no solution, in the worst case. This paper presents an approach to obtain expected result if an agent dies. Our studies are based on a classical DisCSP, Multi-ABT [6] (and may be adapted to other models). To this end, the sub-problem of failed agent will be supported by another agent of the system chosen by a distribution algorithm. This agent has to solve its own sub-problem and the one of the failed agent. In the previous example, A failure of an agent in the nurses management problem presented by Solotorevsky and Gudes [10] provides a transportation planning incoherent with the

nurses timetabling. More generally, the failure agent does not allow a correct result: the property of robustness is not checked. The paper describes a proposal based on the replication (*i.e.*, a duplication of local CSP) to answer to this problem.

The remainder of this paper is structured as follows: Section 2 presents the definitions of DisCSP, Multi-ABT and discuss the related work. Section 3 explains our proposed approach. Section 4 presents our experiments and Section 5 concludes the paper and give some perspectives.

2 Background

A CSP is a set of variables which are related to each other through a set of constraints. Each variable has a number of values that can be assigned to it (called the variable domain).

A DisCSP is a CSP which is solved by a multi-agent system. Each agent encapsulates one or more variables. A DisCSP is solved if each variable has an assigned value from its domain, and the problem constraints are satisfied. Many algorithms are proposed in the literature. Generally, they concern the multi-variables DisCSP where each agent has more than one variable, such as Multi-Asynchronous Backtracking Algorithm (Multi-ABT) [6], Multi-Asynchronous Weak Commitment (Multi-AWC) [14], Asynchronous Forward Checking (AFC) [3], and Multi-Distributed Backtracking Algorithm (Multi-DBS) [8].

In this paper, we focus to Multi-ABT where agents are ranked according to a priority order. It works as follow:

- Each agent finds a solution for its local CSP and sends it to neighbors having a lower priority through an *OK? message*.
- Receiving *OK? message*, an agent records the message content and tries to find a coherent solution with it. If there is no coherent local solution, the agent sends *"BT message"* to the *OK? message* sender to modify its solution.
- Multi-ABT stops if the agent having higher priority does not find a coherent local solution (unsatisfiable DisCSP), or if the agent having lower priority find a coherent local solution (satisfiable DisCSP).

In the literature, different methods have proposed to improve robustness and fault tolerance of multi-agent system. Klein et al [7] propose an approach to handle agent death within a Contract Net protocol. This approach is based on domain-independent Exception Handling service. This service aims to find an agent which can support the died agent task. This agent is chosen after a negotiation between the agents. Fedoruck et al [4] proposed an approach based on a transparent agent duplication. The concept is to introduce proxies which acts as an interface between the replicates group and system components. The role of these proxies is to reveal replicates group as one entity. Another method based on the concept of duplication was proposed by Guessoum et al [1, 5] to identify critical agents in the system and duplicate them. A critical agent is the one which has a high probability to fail.

If an agent dies during the execution of Multi-ABT, the algorithm can provide wrong results. To handle agent failure within a multi-agent system, proposed approaches use

agents duplication, but no one is applied to solve Distributed CSPs. This paper proposes an approach which aims to guarantee the resolution of a DisCSP in presence of died agent.

3 Solving DisCSP process in presence of failed agent

This section presents the new messages (necessary for our approach) to detect a failed agent during Multi-ABT execution (3.1), the general process and different algorithms used to handle the failure (3.2). Then, it presents different properties for our approach (3.3).

3.1 Messages for Failure Detection

To detect an agent failure during solving a DisCSP using Multi-ABT, two additional message types are used:

Check(A_i, active): It is the same kind of message defined by Tanenbaum [11] – If an Agent A_i does not receive any message from one of its neighbors for a defined time interval, it sends it this message to check if it is still active. Receiving this message, an agent suspends its behavior to respond to it by a message having *"ACTIVE"* as a content, otherwise, it will be considered as failed agent.

Inform(A_i, Neighbors(A_i), A_j): After detecting an agent A_j failure, the agent A_i, which detects the failure, informs its neighbors of this failure. Receiving this message, *Neighbors(A_i)* transmit it to their neighbors, and so on. This diffusion aims to inform all the died agent neighbors of this failure.

3.2 Handling failure process

The general process of our approach operates as follow (Algorithm. 1):

- The process begins by executing a DistributeCSP algorithm (Algorithm 2) (Line 1). Receiving the additional CSPs, each agent creates a list CSP_{add} where it records received CSPs (Line 2-3).
- After distributing the local CSPs, the DisCSP solving starts by executing Multi-ABT (Line 4). If, after a time interval, no message is delivered from A_j to A_i (Line 5), the process of failure detection begins by sending $< Check >$ message to A_j (Line 6).
- If A_j does not respond to this message (Line 7), it is considered as a failed agent (Line 8), and the information message $< Info >$ is sent to A_i neighbors (Line 9).
- Receiving the information of A_j failure, each agent (and A_i) transmits this message, and checks if it owns the failed agent CSP in its CSP_{add} list (Line 10). Then, the agent merges its own local CSP with the one of the failed agent by executing *MergingCSP* algorithm (Algorithm. 3) (Line 11).
- After merging local CSPs, the agent informs A_j neighbors that it supports the failed agent CSP (Line 12) by sending a $< MergeCSP >$ message.

Algorithm 1: GeneralProcess

```
   Input: DisCSP : problem to solve
1  DistributeCSP();
2  foreach A_i, i ∈ {1, .., n} do
3      CSP_add(A_i) ← ∅ ;
4      Algorithm of resolution;
5      if TimeOut(A_j), A_j ∈ Neighbors(A_i) then
6          Send < Check > to (A_j);
7          if NotReply "ACTIVE" then
8              failed(A_j) ← True;
9              Send < Inform > to (Neighbors(A_i));
10     if (CSP(A_j) ∈ CSP_add_i) then
11         MergingCSP();
12         Send < MergeCSP > to (Neighbors(A_j));
13     Update Neighbors(A_i)
```

– In the end of general process, each agent updates its neighbors list. If it is the one which supports failed agent CSP, it adds failed agent neighbors to its neighbors list. Otherwise, it deletes failed agent from its neighbors list, and replaces it by the delegate one (Line 13).

DistributeCSP Algorithm This algorithm aims to have a copy of each local CSP among another agent. It guarantees that an agent can have one or more copies of CSPs, but a local CSP is copied once and once only, to ensure that the local CSP of failed agent will be supported by a single agent. Also, it avoids to obtain all local CSPs among a single agent. The algorithm is executed by a *Dispatcher Agent* which occurs only during the distribution, i.e. its absence does not affect the resolution of global CSP.

Algorithm 2: DistributeCSP

```
   Input: CSPs : all local CSPs
1  MaxCSP ← 0;
2  while CSPs ≠ ∅ do
3      foreach A_i, i ∈ {1, .., n} do
4          if size(CSP_add(A_i)) ≤ MaxCSP then
5              foreach Neig ∈ neighbors(A_i) do
6                  if ¬assigned(CSP_Neig) then
7                      CSP_add(A_i) ← CSP_add(A_i) ∪ {CSP_Neig} ;
8                      assigned(CSP_Neig) ← true;
9                      CSPs ← CSPs − {CSP_Neig} ;
10     MaxCSP ← MaxCSP + 1;
```

DistributeCSP Algorithm operates as follow: As input, the *Dispatcher agent* has the list of local CSPs of different agents. Firstly, the *Dispatcher Agent* defines MaxCSP as the maximum size of created CSP_{add} (Line 1). The distribution process is executed until obtained an empty CSPs list (Line 2). The *Dispatcher Agent* checks the CSP_{add} size of each agent (Lines 3). If it is lower than the *MaxCSP* value (Line 4), it checks whether the CSP of one of each agent A_i neighbors is assigned to another agent (Lines 5-6). Then, it passes to the next neighbor of A_i. Else, it assigns it to A_i (Lines 7-8), and deletes it from the no assigned local CSPs list (Line 8). The *MaxCSP* value is incremented after each iteration (Line9).

After distributing local CSPs, *Dispatcher Agent* behavior is suspended until receiving new failure information.

MergingCSP Algorithm MergingCSP algorithm is executed by the agent whose *CSP*$_{add}$ list contains a copy of failed agent CSP. This algorithm aims to merge two CSPs belonging to two different agents to obtain a single local CSP.

The CSPs merging is executed as follow: The merging is done between CSP of agent that execute this algorithm and failed agent A_j CSP. So, A_j CSP is declared as input. If A_i owns a copy of failed agent CSP in its CSP_{add} list, it executes merging instructions (Line 1). CSP variables of A_j are added to variables of A_i CSP (Lines 2-3), and all intra-agent constraints of A_j become intra-agent constraints of A_i (Line 5). Inter-agent constraints that interconnect A_i and A_j become intra-agent constraints of A_i (Lines 6-7). Also, inter-agent constraints that connect A_j and agents other than A_i become inter-constraints of A_i. Finally, and after updating its neighbors list, Agent A_i informs failed agent neighbors that it will support failed agent CSP by sending $< MergeCSP >$ message. $< MergeCSP(A_i, A_j) >$ message contains the agent which support failed agent (Agent A_i), and failed agent (Agent A_j). This message is sent in order to allow receiving message agents to update their neighbors list. Since A_i possess a new local CSP, it resumes CSP solving from the beginning.

The CSPs merging is all along consistent. In fact, during the global CSP construction, each variable and each intra-constraint belong to a single agent (an agent local CSP is unique).

Algorithm 3: MergingCSPs

```
   Input: CSP_j : CSP of failed agent
1  if CSP_j ∈ CSP_add(A_i) then
2      foreach Var_j ∈ CSPj do
3          Variables(A_i) ← Variables(A_i) ∪ {Var_j};
4      foreach C_ij ∈ CSP_j do
5          Constraints(A_i) ← Constraints(A_i) ∪ {C_ij};
6          if ¬intra(C_ij) then
7              intra(C_ij) ← True;
```

3.3 Properties

We show the main properties for our approach, which do not change the initial algorithm:

Soundness and completeness: Handling failure approach is sound and complete. If the approach stops, all DisCSP variables have assigned values, and all constraints are satisfied, if a solution exists [13].

Termination: This approach terminates after solving global DisCSP since Multi-ABT terminates [13]

Space complexity: Each agent needs $O(dn + e + 2d^n)$ of memory where e si the number of system constraints, d is size of each variable domain, n is the number of variables of each agent,

Time complexity: It is $O(m^2 + N + e_{intra}d^n + e_{inter}d^N)$ for each agent such as e_{intra} and e_{inter} are the number of constraints intra and inter-agent of each local CSP, respectively, d is the domain size of each agent, N is the total number of variables, and m is the number of agents in the system [9].

4 Experiments

This section describes different assumptions used in our experiments (4.1), and present obtained results (4.2).

4.1 Assumptions

To validate our approach, different DisCSPs are randomly generated according to parameters $< m, n, d, p >$ such as:

- m is the number of agents of the system
- N is the number of CSP variables with a domain d
- p is the hardness of each constraint in the system that presents the percentage of each constraint to be satisfied.

To guarantee the same degree of generated DisCSP hardness, proposed parameter values belong to transition phase defined by Xu et al. [12]. So that DisCSP belongs to transition phase, it is necessary that $d > \sqrt{N}$, such as N is the variables number of the system, and $p \leq 50\%$. Constraints number was defined by Xu et al. [12] as $e = -N.(\frac{ln(d)}{ln(1-p)})$.

To evaluate proposed failure handling, obtained results of handling failure are compared with those obtained if the failure is not detected, and if there is no failure. This comparison is done according to evaluation criteria proposed by Mandiau et al [8]: The number of exchanged messages, total CPU calculated from the beginning of Multi-ABT execution to the end of slower agent behavior, and the Number of Checked Constraints NCCC.

For each instance, an agent failure is simulated just after sending its first solution. The failed agent is chosen randomly. If the agent with lower priority does not receive

any message from any agent for 90 seconds, the algorithm stops. To do this, we used JADE multi-agent platform. Results of these simulations were obtained on a computer equipped with 2.4 GHz Intel Core i7 and 8Go of RAM. Results concern instances having as parameters $< m, n, 6, 0.4 >$.

4.2 Results

Figure 1 presents results of solving DisCSPs having failed agent with and without applying our approach. The y-axis presents the percentage of results that give a solution. This figure presents only instances that give solutions because, in absence of an agent, and if the result is wrong, the system displays automatically that there is no solution. In this figure, the number of expected results obtained by solving DisCSP in presence of failed agent decreases if the number of agents increases. But if our approach is applied, the same results as those obtained by applying Multi-ABT without failed agent are obtained (all instances giving solutions with Multi-ABT provide also solutions by applying our approach).

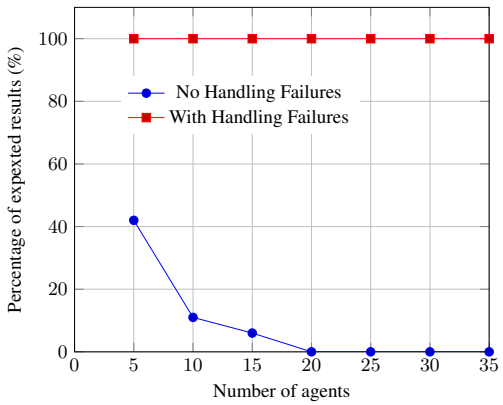

Fig. 1. Results obtained with and without handling failure

Figure 2 presents the variation of exchanged messages. The number of exchanged messages decreases if the system loses an agent. This decreasing is due to the loss of messages sent by the failure agent. Also, applying our approach, exchanged messages number increases. This increasing is due to the addition of exchanged messages after detecting the failed agent. In fact, since some agents resume the resolution from the beginning after detecting the failure, some messages are sent twice : before and after the failure detection.

Figure 3 presents the CPU spent by each resolution method. A decreasing of 40% is observed in DisCSPs with 30 agents which is due to the elimination of the failed agent behavior. On the other hand, an increasing of the CPU is observed, when applying our approach, which is due to the added information to handle. In fact, the time spent to

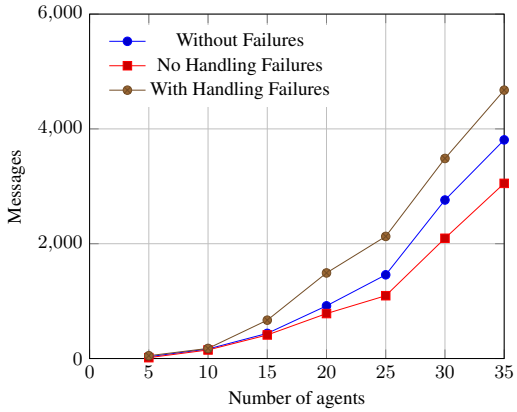

Fig. 2. DisCSP solving for the number of Messages

detect failed agent, merge the CSPs and transmit different information, is an extra time that added to the initial CPU time.

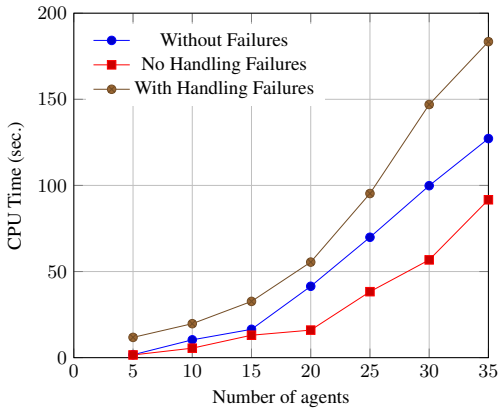

Fig. 3. DisCSP solving for the CPU Time

Figure 4 presents the number of NCCCs with and without agent failure. Its values decreases slightly if a failure is not handled, and this is due to the loss of some inter-agent constraints checking (which belongs to the failed agent). This decreasing of NCCCs number if an agent dies is due to the diminution of solutions number of the merging CSP. In fact, since the agent has more intra-agent constraints, the number of its solutions decreases. Then, during the resolution, the agent has less solutions to check.

The variation of these values does not affect the final result. In fact, this paper is interested to the final solution of the DisCSP, to obtain the same result as that obtained

without died agent. In spite of the variation of exchanged message number, of NCCCs number and of spent CPU time, the proposed method can solve a global DisCSP having a failed agent by giving the same result as the one given without died agent.

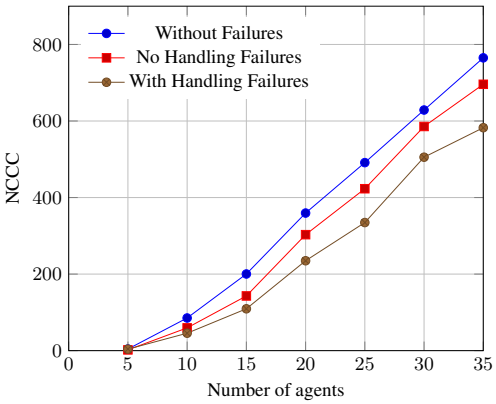

Fig. 4. DisCSP solving for NCCC

5 Conclusion

Distributed systems are used to solve some problems that can not be solved by centralized way. But, a failure in a distributed system can influence the final result. This paper proposes a method, which is applied on Multi-ABT, to solve a DisCSP if an agent fails. This method is based on the duplication of the local CSP of each agent. Each local CSP should be copied in another agent. If one agent fails, the agent which owns a copy of its CSP will supports it. The choice of this agent is made by an algorithm of distribution. During the resolution and after detecting the failed agent, the agent having a copy of its CSP merges its own CSP with the one of failed agent to obtain a new local CSP. The changes are done only in a local CSP, i.e. the global DisCSP is still the same one.

Most research on DisCSP considers that each agent encapsulates one variable, that's why, experiments consider single-variable case. The experiments results show that results obtained by applying the proposed method are the same as that given by Multi-ABT without failed agent. Also, they show that our method is expensive in term of CPU and exchanged messages. This method can be applied with multi-variable case, i.e. each agent encapsulates more than a single variable. Also, it can be adapted to be applied with other algorithms than Multi-ABT such as Multi-AWC and Multi-DBS. It can be adapted to handle too more than a single failure. The downside of this method is that not respect the privacy of each agent, since the local CSP of each agent should be copied in another one.

Bibliography

[1] S. Ductor, Z. Guessoum, and M. Ziane. Adaptive replication in fault-tolerant multi-agent systems. In *Proceedings of the 2011 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, pages 304–307, 2011.

[2] J. Dunagan, N.J.A. Harvey, M.B. Jones, D. Kostic, M. Theimer, and A. Wolman. FUSE: lightweight guaranteed distributed failure notification. In *6th Symposium on Operating System Design and Implementation*, pages 151–166, 2004.

[3] R. Ezzahir, C. Bessiere, M. Wahbi, I. Benelallam, and E.H. Bouyakhf. Asynchronous inter-level forward-checking for discsps. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 304–318. Springer-Verlag, 2009.

[4] A. Fedoruk and R. Deters. 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, 2002.

[5] Z. Guessoum, N. Faci, and J-P Briot. *Adaptive Replication of Large-Scale Multi-agent Systems – Towards a Fault-Tolerant Multi-agent Platform*, pages 238–253. Springer Berlin Heidelberg, 2006.

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

[7] M. Klein, J.A. Rodríguez-Aguilar, and C. Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Autonomous Agents and Multi-Agent Systems*, 7(1-2):179–189, 2003.

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

[9] P. Monier, S. Piechowiak, and R. Mandiau. A complete algorithm for discsp: Distributed backtracking with sessions (dbs). *Second International Workshop on: Optimisation in Multi-Agent Systems (OptMas), Eigth Joint Conference on Autonomous and Multi-Agent Systems*, 2009.

[10] G. Solotorevsky and E. Gudes. Solving a real-life nurses time tabling and transportation problem using distributed csp techniques. Technical report, In Constraints and Agents: Papers from the 1997 AAAI Workshop, 1997.

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

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

[13] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings of the 12th International Conference on Distributed Computing Systems, Yokohama, Japan, June 9-12, 1992*, pages 614–621, 1992.

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