Fault Tolerance in DisCSPs: Several failures case

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
fadoua.chakchouk@univ-valenciennes.fr,
[2] ENSI, University of Manouba, Tunisia

Abstract. To solve a distributed problem in presence of a failed entity, we have to find a way to accomplish the failed entity tasks. In this paper, we present an approach which guarantees the resolution of DisCSPs in presence of failed agents. This approach is based on local CSPs replication principle: each failed agent local CSP is replicated in another agent which will support it. Obtained results confirm that our approach can solve a DisCSP in presence of failed agents by giving a solution when it exists.

Keywords: DisCSP, Robustness, Fault tolerance, Replication, Agent failure

1 Introduction

Distributed systems as Multi-Agent Systems (MAS) are defined as a set of entities that appear to users as a single system. This kind of system is characterized by a partial failure notion : if a component fails, it can affect *the proper operation of other components, while at the same time leaving yet other components totally unaffected* [8]. A failed system is defined in the literature as a system which cannot achieve its goals. In this paper, we are interested to take account agents' failures in a Multi-Agent System to solve Distributed Constraint Satisfaction Problems (DisCSP). In centralized systems, the failure of an agent causes the abandon of the solution search. However, in distributed systems, we defend the idea that the failure of an agent can be covered by replicating the failed agent tasks to another agent.

This paper presents an approach to solve DisCSP that can support the failure of more than one agent. It is based on replication principle: each failed agent local CSP is replicated in another active agent. The organization of this paper is as follows: Section 2 presents DisCSP definition and the fault tolerance in multi-agent systems. Section 3 describes different steps of our proposed approach. Section 4 presents results obtained with the proposed method to solve DisCSPs. Finally, Section 5 concludes and gives our perspectives.

2 DisCSP and fault tolerance

To solve a DisCSP, each agent has a set of variables to which it assigns values from a predefined domains. These agents communicate together in order to satisfy a set of constraints that connect variables. This paper is intended to solve DisCSPs where each agent has more than one variable. We focus on Multi-ABT algorithm [6]. In fact, this algorithm is well-known in this field.

Results provided by this algorithm in presence of failures can be wrong. According to Tanenbaum and Steen [8], the distribution concept of a system aims to *recover from partial failures without seriously affecting the overall performance.* A failure can affect agents, such as crash failures (sudden stop of an agent), or byzantine failures (if an agent provides wrong results which are considered correct). It can also affect communication between agents, such as omission failures (if an agent does not respond to requests), or Timing failures (if an agent exceeds a response time interval).

To cover these failures, several methods are proposed in the literature, based on agents replication, or the utilization of sentinels. An approach based on replication agent is proposed by Fedoruck et al [3]. This approach principle introduces proxies which reveal replicates group as one entity. Some proxies choose and manage replication modes, while other proxies manage all internal and external group communications. Other approaches based on critical agents replication are proposed by Ductor et al. [2] or Guessoum et al [4]. A critical agent is the agent having the higher probability to fail. Sentinels were proposed for the first time by [5] as entities that monitor agents features, and protect them from undesirable events.

3 Proposed Approach

In this paper, we propose an approach that handles the crash failure in a DisCSP. To solve DisCSP in presence of failed agent, its local CSP should be solved. Our approach aims to find a global solution by solving failed agent CSP. The approach detects the failed agent, and assigns its CSP to another agent by replicating the CSPs of failed agents in other ones. This assignment is done by an agent called Dispatcher Agent (A_{Dis}). In this section, we present the solving process details : the failed agent detection, and the handling failure details.

3.1 Failure Detection

To detect a failed agent, additional messages are exchanged between agents.

Check (state) message This message is sent by an agent A_i to its neighbor A_j. Tanenbaum [8] introduced this kind of message within distributed systems: if A_i does not receive any solving message from one neighbor A_j for a time interval, it sends to it this message to check its state. If A_i does not receive a response for this message from A_j, it considers A_j as a failed agent. The Check Message has the highest priority; i.e. if an agent receives this message, it suspends its behavior to reply to it.

Active ( ) message This message is sent by A_j to A_i as a response to a Check message received from A_i.

isFailed (A_j) message It is sent by an agent A_i to the agent A_{Dis} to inform it that an agent A_j is failed. Receiving this message, Agent A_{Dis} replicates each failed agent CSP to another agent.

A failed agent is an agent which can neither receive nor send messages. The failure is detected during the global DisCSP solving.

3.2 Failure Handling Algorithm

The proposed approach is composed from the solver agents A_i of the DisCSP, and the Dispatcher Agent A_{Dis}. Each agent A_i has a list called CSP_{add}, where it will store the replicas sent by Agent A_{Dis} later. A replicas is a copy of a failed agent local CSP.

The agents start the CSP solving as follows (Algorithm 1): Each solver agent $self$ solves its local CSP and interacts with its neighbors according to Multi-ABT algorithm (Line 1). During the CSP solving, if the agent $self$ does not receive solving messages from one neighbor A_j after a time interval ($TimeOut$) (Line 2), $self$ sends to A_j a $Check(state)$ message (Line 3). If $self$ does not receive an $Active$ message from A_j for an interval time ($TimeOutState$) (Line 4), A_i considers it as a failed agent (Line 5), and informs its Agent A_{Dis} by sending to it $isFailed(A_j)$ message (Line 6).

After identifying failed agents (Algorithm 2) and informing Agent A_{Dis}, Agent A_{Dis} sends the failed agents' CSPs replicas to the solver agents. Each agent A_i records the received replicas in its CSP_{add} list. Then, it merges the CSP_{add} with its local CSP by executing *MergingCSP()* algorithm (Line 2). Each agent informs its neighbors that it supports A_j local CSP by sending $MergeCSP(A_i, A_j)$ message (Line 3), which contains the ID of a failed agent (A_j), and its delegate (A_k). Receiving this message, each agent deletes failed agents from its neighbors list, replace them by the delegate ones, and transmits the $MergeCSP(A_i, A_j)$ message to its neighbors.

Algorithm 1: Failure detection

```
   Input: CSP : local CSP to solve
1  Multi-ABT:local CSP solving;
2  if TimeOut(A_j), A_j ∈ Neighbors(self) then
3      self sends Check(state)to(A_j);
4      if self does not receive Active() from (A_j) after TimeOutState(A_j)
         then
5          A_j is considered failed;
6          self sends isFailed(A_j) to A_Dis;
```

Algorithm 2: After failure detection

```
   Input: A_j : the failed agent
1  foreach (CSP(A_j) ∈ CSP_add(self)) do
2      MergingCSP();
3      self sends MergeCSP(A_i, A_j) to its neighbors;
```

3.3 Replication Process

This process aims to replicate the local CSP of failed agents in other ones. It ensures that an agent can have more than one copy of different CSPs, but a CSP is replicated within only one agent. According to this process, a CSP of a failed agent is supported by only one agent. This process is executed only and only if the Dispatcher Agent A_{Dis} receives information of failed agents. We suppose that Agent A_{Dis} knows already the failed agents number (before starting DisCSP solving).

Algorithm 3: ReplicateCSP: Dispatcher Agent

```
   Input: CSPs : list of failed agents local CSPs
1  foreach A_i, i ∈ {1, .., m} do
2      if A_i is not failed then
3          List(A_i) ← { } ;
4          repeat
5              foreach A_j ∈ neighbors(A_i) do
6                  if ¬replicated(CSP_j) then
7                      List(A_i) ← List(A_i) ∪ {CSP_j} ;
8                      CSPs ← CSPs − {CSP_j} ;
           until CSPs = { };
```

To replicate local CSPs, the Agent A_{Dis} has the list of failed agents' CSPs as an input (Algorithm 3). Firstly, it proceeds by creating an empty list $List(A_i)$ for each active agent A_i (Lines 1-3) where it records the replicas of CSPs assigned to each agent. Then, it sends the lists to agents in the end of replication process. After that, for each agent (A_i)'s neighbor (Line 5), *Dispatcher Agent* checks if its CSP (CSP_j) is replicated into another agent or not. If not, it assigns CSP_j to Agent A_i by adding it to $List(A_i)$ (Line 6-7). If all neighbors CSPs are replicated into agents other than A_i, Agent A_i will not have any additional CSP (its CSP_{add} remains empty during the DisCSP resolution). After replicating CSP_j into Agent A_i, *Dispatcher Agent* deletes CSP_j from the list of unassigned local CSPs (Line 8). This process is repeated until replicating all the local CSPs of failed agents (obtaining an empty list of unassigned CSPs).

3.4 Merging Process

The Merging algorithm (Algorithm 4) is executed by Agent A_i having at least one replicas of a failed agent CSP. The goal is to merge CSPs into a single one.

Algorithm 4: MergingCSPs

Input: $CSP(A_j)$: CSP of failed agent

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 $InterC(A_i, A_k) \leftarrow InterC(A_i, A_k) \cup \{InterC(A_j, A_{k \neq i})\}$;

Agent A_i starts this process by merging its variables with those of A_j by adding them to its variables list (Line 1), as well as its list of constraints: it updates its intra-agent constraints ($IntraC(A_i)$) by adding to it $intraC(A_j)$ list and $interC(A_j, A_i)$ list that connect A_i to A_j (Line 2). The rest of inter-agent constraints of A_j ($interC(A_j, A_k)$) are added to the inter-agent constraints list of A_i (Line 3).

After merging CSPs, Agent A_i updates its neighbors list by adding Agent A_j neighbors to its neighbors list, and informs them that it supports Agent A_j CSP. To transmit this information, Agent A_i sends $MergeCSP(A_i, A_j)$ message to Agent A_j neighbors. Receiving this message, agents update their neighbors list by removing Agent A_j and replacing it by Agent A_i.

4 Hypothesis and experiments

This section presents several hypothesis used to realize experiments (4.1) and obtained results (4.2).

4.1 Hypothesis

During the experiments, DisCSPs are randomly generated having as parameters $\langle m, n, d, p \rangle$ such as: (i) m is the number of agents, (ii) n is the number of variables for each agent with d as a domain size, (iii) p is the hardness of each DisCSP constraint. The generated DisCSPs are presented as connected graph. During Multi-ABT execution, failures are simulated either before receiving a first message, or just after sending a first solution. An agent has 10 seconds to reply to a *Check* message, otherwise, it will be considered as a failed one. This interval is sufficient to give an agent time to interrupt its behavior and respond to *Check* message.

To evaluate our proposed approach, results are compared by increasing the number of failed agents, and the number of agents. The comparison is done according to: the number of exchanged messages to solve the DisCSP and to detect and handle the failures, and the CPU time calculated, from the beginning of the

DisCSP solving to the end of the slowest agent behavior, and from the detection of a failure until the resumption of the DisCSP resolution. Results presented in the next section concern DisCSP generated with parameters $\langle m, 4, 4, 0.5 \rangle$.

4.2 Experiments and results

This section presents results obtained by varying the number of failures. Experiments are done with JADE multi-agent platform. Results of simulations were obtained on a computer equipped with 2.4 GHz Intel Core i7 and 8GB of RAM. The number of instances is 50 for each experiment. In theory, by increasing the number of failed agents, the number of messages exchanged decreases, since the number of communicating agents decreases. Also, the total CPU time increases. In fact, the time spent to detect and handle the failures is an extra time added to the initial one. Through these experiments, we aim to validate these hypothesis, and to improve results obtained by Chakchouk et al. [1].

Table 1. DisCSP solving number of Messages

Failed agents number \ Agents number		4	6	8	10	12
1 failure	Multi-ABT	64.35	428.82	1394.75	2506.2	4237.6
	Additional	40.7	26.7	41.2	68.4	118.9
2 failures	Multi-ABT	52.3	414.15	1262.15	2770.71	5358.4
	Additional	27.5	21.15	38.45	64.75	105.42
3 failures	Multi-ABT	50.07	366.25	1163.2	2305.7	4140.17
	Additional	16	18.25	34.05	51.7	88.64
4 failures	Multi-ABT	-	203.47	868.95	2243.3	5200.55
	Additional	-	14.73	28.35	51.45	89.1
6 failures	Multi-ABT	-	-	523.05	1661.75	4348.9
	Additional	-	-	23	35.2	53.4
8 failures	Multi-ABT	-	-	-	-	3252.44
	Additional	-	-	-	-	39.45

Table 1 presents the impact of the failed agent's number variation on the exchanged messages number. It contains the number of solving messages (Multi-ABT messages) in presence of failures, and the number of exchanged messages to detect and handle failures (additional messages). We can observe that, in terms of the number of exchanged solving messages, the number decreases by increasing the failed agents number. In fact, merged CSPs number increases by increasing the number of failed agents. Then, the number of inter-agent constraints decreases, and each delegated agent has fewer neighbors with whom it communicates.

Note also that the number of additional messages decreases by increasing the number of failed agents. In fact, the most exchanged message is the checking message *Check*. Since the number of failures increases, the number of replies

to it, by sending *Active* message, decreases. In addition, if an agent is active, it can receive a *Check* message several times from the same agent. However, once declared failed, no more *Check* messages are sent to it. As after merging of CSPs, the diffusion of the $NewCSP$ message decreases with the increase of failed agents number.

Table 2. DisCSP solving CPU time

Failed agents number \ Agents number		4	6	8	10	12
1 failure	Total CPU (*sec*)	21.44	23.53	27.24	32.65	41.2
	Additional CPU (*sec*)	1.2	1.32	2.21	4.14	5.35
	Dispatching CPU ($10^{-3}sec$)	2.75	2.64	3	3.05	3.55
2 failures	Total CPU (*sec*)	29.35	27.75	36.37	30.01	34.69
	Additional CPU (*sec*)	1.32	1.87	2.78	2.91	5.81
	Dispatching CPU ($10^{-3}sec$)	4	5.26	7.06	6.88	9.44
3 failures	Total CPU (*sec*)	31.53	35.14	37.94	39.07	47.85
	Additional CPU (*sec*)	2.72	2.82	2.93	3.51	4.71
	Dispatching CPU ($10^{-3}sec$)	4.56	6.42	7.95	10.2	11.5
4 failures	Total CPU (*sec*)	-	42.24	38.97	43.29	53.45
	Additional CPU (*sec*)	-	3.12	3.4	3.56	6.29
	Dispatching CPU ($10^{-3}sec$)	-	8.89	9.9	11.84	13.6
6 failures	Total CPU (*sec*)	-	-	59.97	56.88	73.59
	Additional CPU (*sec*)	-	-	4.34	3.93	8.05
	Dispatching CPU ($10^{-3}sec$)	-	-	12.9	16.21	25.82
8 failures	Total CPU (*sec*)	-	-	-	-	91.18
	Additional CPU (*sec*)	-	-	-	-	11.49
	Dispatching CPU ($10^{-3}sec$)	-	-	-	-	32.8

Table 2 shows the variation of the CPU time calculated from the beginning of the DisCSP solving to the end of the slowest agent behavior (total CPU), by increasing the number of failed agents. Also, it contains the CPU time spent to detect and handle failures (additional CPU), and that spent by Dispatcher Agent to replicate failed agents' CSPs (dispatching CPU). We observe that by increasing the number of failed agents, the additional CPU time increases. In fact, the detection of one more failure requires an additional exchange and waiting for messages. During the replication process, the Dispatcher Agent browses all the CSPs of failed agents and all their neighbors, which explains the increase of the dispatching CPU by increasing the number of failures.

The increase of these CPU times, as well as the resumption of the resolution after failures detection, generate the increase of the total CPU time. In fact, after merging CSPs, the delegated agents reproduce all their local solutions, and all the agents resume the DisCSP resolution from the beginning.

The obtained results show that additional processes produce an increase in terms of CPU time, but the additional CPU time is almost negligible compared

to the total CPU time spent to solve the DisCSP. These processes, also, produce a decreasing of the exchanged messages number. The most important is that at the end of DisCSP solving, we obtain a solution if it exists.

5 Conclusion

This paper describes an approach, which is applied on Multi-ABT, to solve a DisCSP if more than one agent fail. This method is based on the replication of local CSP of each failed agent : each local CSP of failed agents has a replicas in another active agent. If an agent fails, its neighbor, which owns its CSP replicas, supports it by merging the replicas with its own CSP. This leads 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.

Experiments results show that this approach give expected results (a solution if it exists, otherwise no solution). Also, they show that our method increases in term of CPU time, but decreases in term of exchanged messages. This approach can be adapted to be applied with other algorithms than Multi-ABT. The next step of this work will be try other kind of failures, such as the presence of malicious or liar agent. These failures introduce the trust and reputation notion between agents [7].

References

1. F. Chakchouk, J. Vion, S. Piechowiak, R. Mandiau, M. Soui, and K. Ghedira. Replication in fault-tolerant distributed CSP. IEA/AIE 2017, Springer, 2017.
2. S. Ductor, Z. Guessoum, and M. Ziane. Adaptive replication in fault-tolerant multi-agent systems. In *Proceedings of International Conference on Intelligent Agent Technology*, pages 304–307, 2011.
3. 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, 2002.
4. Z Guessoum, JP Briot, and N Faci. Towards fault-tolerant massively multiagent systems. In *International Workshop on Massively Multiagent Systems*, pages 55–69. Springer, 2004.
5. S Hägg. *A sentinel approach to fault handling in multi-agent systems*, pages 181–195. Springer Berlin Heidelberg, 1997.
6. K. Hirayama, M. Yokoo, and K. Sycara. An easy-hard-easy cost profile in distributed constraint satisfaction. *Information Processing Society of Japan journal*, pages 2217–2225, 2004.
7. T.D Huynh, N.R Jennings, and N.R Shadbolt. An integrated trust and reputation model for open multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, pages 119–154, 2006.
8. A.S. Tanenbaum and M.V Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., 2006.