Self-Organized and Resilient Distribution of Decisions over Dynamic Multi-Agent Systems

P. Rust[1,2], G. Picard[2], and F. Ramparany[1]

[1] Orange Labs, France
{pierre.rust,fano.ramparany}@orange.com
[2] MINES Saint-Etienne, Laboratoire Hubert Curien UMR CNRS 5516, France
picard@emse.fr

Abstract. Here we define and model the notion of k-resilient distribution of decisions, implemented as computations, over dynamic and constrained multi-agent systems (like in IoT systems), and devise a self-organizing algorithm to repair the distribution as to ensure the system still provides the required services and remains resilient to upcoming changes. Resilience is based on the concept of replicates of computations, so that decisions hosted by disappearing nodes can activate on other nodes. The contributions are evaluated on scenarios coming from the distributed constraint optimization domain (DCOP), where computations are decision variables or constraints which are distributed over a set of agents. We experimentally evaluate the performances of our repair method on IoT-structured multi-agent systems which must keep solving a problem using the MaxSum algorithm while agents are disappearing.

1 Introduction

We consider the problem of distributing a set of decisions (implemented as computations) on a set of agents (or nodes). Computations are organized in a computation graph, where vertices represent computations and edges represent a dependency relation between computations. This kind of organization is used in many computation models, like for example factor graphs and constraints graphs used when solving distributed constraints optimization problems (DCOP) [3] or graph algorithms such as those addressed by Pregel or other BSP-based frameworks [5]. While these frameworks usually target high-performance cluster computing, we consider here Internet-of-Things (IoT) and edge computing scenarios, where computations run on distributed, highly heterogeneous, nodes and where a central coordination might not be desirable or even not possible [2].

In such settings, systems must be able to cope with node additions and failures: when a node stops responding, other nodes in the system must take responsibility and run the orphaned computations. Similarly, when a new node is added in the system, it might be useful to reconsider the distribution of computations in order to take advantage of the newcomer's computational capabilities, as proposed in [10]. As to cope with such dynamicity by keeping computations and decisions going whilst the infrastructure changes, one solution inspired by distributed databases is *replication* [14,13]. But here, in order to ensure resilience of decisions, we propose to replicate computation definitions instead

of data. More precisely, we define in this paper the notion of k-resilience, which characterizes systems able to provide the same functionalities (or make the same decisions) even when up to k nodes disappear. As far as we know, only [10] addressed the problem of adapting decision distribution at runtime and proposed a model for computing such distribution. However no distributed methods were implemented and evaluated while DCOP solution methods were running, and the disappearance of several nodes at the same time was not considered.

The contributions of this paper structure the paper as follows. We define the notion of optimal distribution of graph-based computations over a given infrastructure in Section 2. We define the notion of k-resilience, and propose a distributed iterative lengthening method with minimum path bookkeeping to deploy k replicas for each computation to ensure k-resilience, at runtime, in Section 3. We devise a distributed repair method, based on a DCOP model using replication to adapt the computation deployment following changes in the infrastructure (nodes disappearing), in Section 4. We evaluate experimentally our algorithmic contributions on multi-agent systems structured as IoT infrastructures whose functionality is to solve random DCOPs, in Section 5. Finally, we conclude the paper on some perspectives.

2 Assigning Computations to Agents

The placement of computations on the agents has an important impact on the performance characteristics of the global system: some distributions may improve response time, some other may favor communication load between nodes and some other may be better for other criteria like QoS or running cost.

Let $G = \langle \mathbf{X}, D \rangle$ be a computation graph, where $\mathbf{X}$ is the set of computations (or decisions) x_i, and D is the set of edges (i,j) representing the dependencies between computations (which implies that messages between neighbors computations are passed along these edges). Let $\mathbf{A}$ be the set of agents which can host the computation $x_i \in \mathbf{X}$. We note $\mu : \mathbf{X} \mapsto \mathbf{A}$ the function that maps computations to agents and $\mu^{-1}(a_m)$ the set of computations hosted on agent a_m. An agent can only host a limited quantity of computations, constrained by agent's *capacity* $\mathbf{w_{max}}(a_m)$, and computation's *weight*, $\mathbf{w}(x_i)$. We note x_i^m the boolean value in $\{0,1\}$ stating whether computation x_i is hosted on agent a_m.

Definition 1. *Given a set of agents* $\mathbf{A}$ *and a set of computations* $\mathbf{X}$*, a* ***distribution*** *is a mapping function* $\mu : \mathbf{X} \mapsto \mathbf{A}$ *that assigns each computation to exactly one agent and satisfies the agents' capacity constraints.*

Finding a distribution is a constraint satisfaction problem with:

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

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

When communication is constrained (like in IoT), distribution should generate as little load as possible and favor the cheapest communication links. We assume that all agents can potentially communicate with each other and model the communication cost with a matrix: $\mathbf{route}(m,n)$ is the communication cost between agents a_m and a_n.

Let $\mathbf{msg}(i,j)$ be the size of the messages between x_i and x_j, the communication cost between x_i on a_m and x_j on a_n is:

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

where there is no communication cost between computations hosted on the same agent. Costs to host some computation on an agent are modeled as a $\mathbf{host}$ function assigning a cost for each pair (a_m,x_j).

The quality of a distribution can then be evaluated using the following function, with $\omega \in [0,1]$:

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

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

Definition 2. *An **optimal distribution** is a distribution μ that minimize the cost of communication between agents and minimize the cost of hosting computations.*

Assigning computations to agents as stated in Definition 2 can be mapped to a linear program using linearizations of (4) and (5) as the minimization objectives, (2) as constraints, and adding constraints so that computation are only assigned to one agent. This model is more general than [10], dedicated to factor graphs without hosting costs. It is NP-hard and may drastically stress centralized modern solvers, even for instances with few agents and computations. It can be used when bootstrapping the system to centrally compute an initial distribution for small instances. In our experiments, we'll use this quality measure to evaluate the distributions provided by our techniques.

3 Resilient Distribution

Let's now consider the case of a dynamic infrastructure where agents may disappear. In a centralized setting, when some agents fail, one could use the linear program discussed in the previous section as to re-compute a new optimal distribution. However, accessing such a centralized computation may not be possible or desirable, depending on the requirements of the application scenario. Thus, we investigate decentralized techniques to cope with such dynamic in the infrastructure.

3.1 *k*-Resilience

We define the notion of k-resilience as the capacity for a system to repair itself and operate correctly even when up to k agents disappear. This means that after a recovery period, all computations must be active on exactly one agent and communicate one with another as specified by the graph $\mathbf{G}$.

Definition 3. *Given a set of agents $\mathbf{A}$, a set of computations $\mathbf{X}$, and a distribution μ, the system is k-**resilient** if for any $F \subset \mathbf{A}, |F| \leq k$, a new distribution $\mu' : X \to \mathbf{A}\backslash F$ exists.*

One pre-requisite to k-resilience is to still have access to the definition of every computation after a failure. One approach is to keep k *replicas* (copies of definitions) of each active computation on different agents. Provided that the k replicas are placed on different agents, no matter the subset of up to k agents that fails there will always be at least one replica left after the failure, as classically found in distributed database systems [5]. Here, we apply these ideas except we keep replicas of computation definitions instead of data records, which implies that computations must be *stateless* or that their state must be restorable.

Let's note that given the capacity constraints on the agents, keeping k replicas is not enough to warrant k-resilience and there might be no possible distribution. The maximum k value for which k-resilience can be achieved depends on the system and especially on agent's capacities. Additionally, the k-resilience characteristic of the repaired system should be restored, as long as there are enough nodes available.

3.2 Replica Placement

The problem of assigning replicas to hosts could be considered as an optimization problem, close to Definition 2. Ideally, we should optimize this distribution for communication and hosting costs. This would ensure that when agents fail, replicas are already placed on good candidate agents. However, the search space for this optimization is prohibitively large. In a k-resilient system with n agents, there is $\sum_{0<i\leq k}\binom{n}{i}$ potential failure scenarios (up to k agents out of n can fail simultaneously). With m computations, the number of possible *replica configurations* is $m \cdot \binom{n}{k}$ (for each of the m computations, select k agents to host the replicas). More practically, the problem of optimally distributing the k replicas of each computation on a given set of agents having different costs and capacities can be cast into a quadratic multiple knapsack problem (QMKP) [11], which is NP-hard.

Then, for each of these replica configurations, there are m^k *activation configurations* (for each orphaned computation, select one of the k replicas to be activated). Assuming we could compute, for each replica configuration, the cost of all these activation configurations, it would still not be obvious which replica configuration would be better: Is it the one that allows the best activation-configuration? Is it the one that, on average, allows good quality activation configurations? Or maybe the one that gives the best activation configurations over the set of possible failure scenarios? Obviously, defining the optimality for replica placement is a matter of choice and is very problem dependent. Thus, given that complexity, we opt for a distributed heuristic approach consisting in placing replicas on agents close to the agent hosting the current active computation.

3.3 Distributed Replica Placement Method

We propose here a distributed method, namely DRPM, to determine the hosts of the k replicas of a given computation x_i. DRPM is a distributed version of iterative lengthening (uniform cost search based on path costs) with minimum path bookkeeping to find the k best paths. The idea is to host replicas on closest neighbors with respect to communication and hosting costs and capacity constraints, by searching in a graph induced by computations dependencies. It outputs a distribution of k replicas (and the path costs to

their hosts) with minimum costs over a set of interconnected agents. If it is impossible to place the k replicas, due to memory constraints, DRPM places as much computations as possible and outputs the minimum resilience level it could achieve. One hosting agent, called *initiator*, *iteratively* asks each of his lowest-cost neighbors, in increasing cost order, until all replicas are placed. Candidate hosts are considered iteratively in increasing order of cost, which is composed of both communication cost (all along the path between the original computation and its replica) and the hosting cost of the agent hosting the replica.

Let's first define the graph specifying the communication costs which will be developed during the search process:

Definition 4 (route–graph). *Given a computation graph $\langle \mathbf{X}, D \rangle$, the* **route***–graph is the edge-weighted graph $\langle \mathbf{A}, E, w \rangle$ where $\mathbf{A}$ is the set of vertices, E is the set of edges with $E = \{(a_m, a_n) | \exists (x_i, x_j) \in D$, and $\mu(x_i) = a_m, \mu(x_j) = a_n\}$ and $w : E \to \mathbb{R}$ is the weight function $w(a_m, a_n) = \mathbf{route}(m, n)$.*

As to take into account both communication and hosting costs in the path costs, the **route**–graph is extended into a **route+host**–graph with extra leaf vertices attached to each agent in the neighboring graph, except the original host of the computation, with an edge weighted using the hosting cost of the agent, as illustrated in Figure 1.

Definition 5 (route+host–graph). *Given a* **route***–graph $\langle \mathbf{A}, E, w \rangle$ and a computation x_i, the* **route+host***–graph is the edge-weighted graph $\langle \mathbf{A}', E', \mathbf{cost} \rangle$ where $\mathbf{A}' = \mathbf{A} \cup \widetilde{\mathbf{A}}$ is the set of vertices where $\widetilde{\mathbf{A}} = \{\tilde{a}_m | a_m \in \mathbf{A}, a_m \neq \mu(x_i)\}$ is a set of extra vertices (one for each element in $\mathbf{A}$ except the host of x_i), $E' = E \cup \{(a_m, \tilde{a}_m) | \tilde{a}_m \in \mathbf{A}\}$ is the set of edges and $\mathbf{cost} : E' \to \mathbb{R}$ is the weight function s.t. $\forall a_m, a_n \in \mathbf{A}$, $\mathbf{cost}(a_m, a_n) = w(a_m, a_n)$, $\forall \tilde{a}_m \in \widetilde{\mathbf{A}}$, $\mathbf{cost}(a_m, \tilde{a}_m) = \mathbf{c_{host}}(a_m, x_i)$.*

A **route+host**–graph is a search graph, expanded at runtime and explored for a particular computation x_i. Each agent operates as many DRPM as computations to replicate over several **route+host**–graph's. For a given **route+host**–graph, each agent may

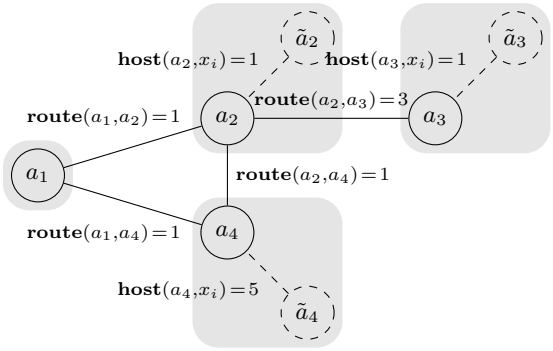

Fig. 1: A sample **route+host**–graph with 4 agents (in gray), where a_1 search for hosting computation x_i. For $k = 2$, DRMP places a replica on a_2 (cost of $1 + 1 = 2$) and another on a_3 (cost of $1 + 3 + 1 = 5$) if enough capacity on these two agents, since the minimum cost path to host on a_4 is higher ($1 + 5 = 6$).

Algorithm 1: Handler for REQUEST messages

```
Data: current, budget, spent, known, visited, k, x_i
known ← known \ current
if me ∉ visited then
    visited ← visited ∪ {me}
    if can_host?(x_i) then
        k ← k − 1
        add x_i to memory
        if k = 0 then
            a_p ← predecessor of me in current
            send ANSWER(current,
              budget+cost(me,a_p), spent-cost(me,a_p), known, visited, k, x_i) to a_p
            return
p ← argmin_{e ∈ {paths in known starting with current}} known[e]
if p ≠ ∅ then
    a_n ← successor of me in p
    if cost(me,a_n) ≤ budget then
        current ← current + a_n
        send REQUEST(current,
          budget-cost(me,a_n), spent+cost(me,a_n), known, visited, k, x_i) to a_n
        return
foreach a_n ∈ {a_m | (a_m,me) ∈ E', a_m ∉ visited} do
    if spent + cost(me,a_n) < min_{e ∈ {paths in known leading to a_n}} known[e] then
        known[current + a_n] ← spent + cost(me,a_n)
a_p ← predecessor of me in current
send ANSWER(current,
  budget+cost(me,a_p), spent-cost(me,a_p), known, visited, k, x_i) to a_p
```

encapsulate two vertices (one in $\mathbf{A}$ and its image in $\widetilde{\mathbf{A}}$) and may receive messages concerning their two vertices, and even self-send messages. Additionally, when assessing if an agent can host a replica for x_i, we ensure that it only accepts if it has enough capacity to activate any subset of size k of its replicas, using a predicate named `can_host?`. Of course this constraint is stronger than what might be actually needed, so, this distribution is not optimal with respect to hosting cost, since one agent reject hosting a computation whilst it may finally have enough memory to host it. Even communication-wise, the algorithm may results on a suboptimal distribution. However, if `can_host?` is provided by an oracle or if memory is not a real constraint, and replica placement only concerns one computation, the distribution would be optimal with respect to communication and hosting costs, since our algorithm implements an iterative lengthening search [9, p.90].

DRPM makes use of two message types (`REQUEST` and `ANSWER`) with the same fields: (i) `current`: path of the request, as a list containing all vertices messages have been passed through from the initiator vertices to the one receiving the current message, (ii) `budget`, `spent`: remaining budget for graph exploration and budget already spent on the current path, (iii) `known`: map assigning cost to already discovered paths to unvisited vertices which bookkeeps the cheapest paths so far, (iv) `visited`: list of already visited vertices, (v) `k`: the remaining number of replicas to host. (vi) x_i: computation that must be replicated,

Algorithm 2: Handler for `ANSWER` messages

```
   Data: current, budget, spent, known, visited, k, x_i
1  if k = 0 then
2      if me is root of current path then
3          terminate with target number of replicas placed
4      else
5          a_p ← predecessor of me in current
6          send ANSWER(current,
             budget+cost(me,a_p), spent-cost(me,a_p), known, visited, k, x_i) to a_p
7  else
8      p ← argmin_{e ∈ {paths in known starting with current}} known[e]
9      if me is root of current path then
10         if p ≠ ∅ then
11             budget ← budget + known[p]
12             a_n ← successor of me in p
13             current ← current + a_n
14             send REQUEST(current,
                 budget-cost(me,a_n), cost(me,a_n), known, visited, k, x_i) to a_n
15         else
16             terminate with fewer replicas than requested
17     else
18         if p ≠ ∅ then
19             a_n ← successor of me in p
20             if cost(me,a_n) ≤ budget then
21                 current ← current + a_n
22                 send REQUEST(current, budget-cost(me,a_n),
                     spent+cost(me,a_n), known, visited, k, x_i) to a_n
23         a_p ← predecessor of me in current
24         send ANSWER(current,
             budget+cost(me,a_p), spent-cost(me,a_p), known, visited, k, x_i) to a_p
```

At the beginning, the agent requiring a computation replication initializes `known` with the paths to its direct neighbors in the **route+host**–graph and sends itself a `REQUEST` message with a budget equals to the cheapest known path. Then, agents handle messages according to Algorithms 1 and 2. The protocol ends when all possible replicas have been placed (at most k).

When receiving a `REQUEST` message (Algorithm 1), either the agent can host a replica (lines 2-10), and thus decreases the number of replicas to place, or forwards the request to other agents (lines 11-22). In the first case, if all replicas have been placed, the agent answers back to its predecessor (line 9). When looking for other agents to host replicas, if there exists a minimum cost known path starting with the currently explored path which is reachable with the current budget, the agent forwards the request to its successor in this path (with an updated cost and budget, line 16). If there is no such path, the agent fill out the map of known paths with new paths leading to its neighbors in the **route+host**–graph, when they improve the existing known paths, and sends this back to its predecessor so that it will explore new possibilities (line 22).

When receiving an `ANSWER` message (Algorithm 2), the message can either notify that all replicas have been placed (lines 1-6) or that there exists at least one replica left to place. In the former case, if the agent is the initiator, it terminates the algorithm, whilst

having all the requested replicas placed (line 3), otherwise it forwards the answer back to its predecessor, until it reached the initiator (line 6). In the later case, if the agent is the initiator it increases the budget and send a request to the closest neighbor (line 14) if any; otherwise that means that there is no more path to explore and that all replicas cannot be placed, therefore the agent terminates (line 16). If the agent is not the initiator, but there exists some reachable path within current budget, it requests replication to its successor in the best known path, as when handling `REQUEST` messages (line 22). Finally, if there is no such path, it simply forwards the answer to its predecessor in the current path (line 24).

System-wise, each agent is responsible for the placement of the k replicas of all the active computations it currently hosts. Therefore, each agent executes DRPM once for each of its active computations. These multiple DRPM runs can be either sequentially or concurrently executed, but their result depends on message reception order. Note however that even when running multiple DRPM concurrently, an agent has only one message queue and handle incoming messages sequentially, which prevents him from accepting replicas that would exceed its capacity.

Theorem 1. *DRPM terminates.*

Proof. For $k = 1$, since DRPM costs are additive and monotonous, and it bookkeeps paths to unvisited vertices, it terminates like classical iterative lengthening, with the minimum cost path or empty path if not enough memory in agents to host the computation x_i. For $k > 1$, DRPM attempts to place each replica sequentially, it first searches for the best path (as for $k = 1$), then operates the same process for a second best path, and so on until either (i) the k replicas are placed (line 3 in Algorithm 2) or (ii) there is not enough memory to host the n^{th} replica (line 16 in Algorithm 2). Bookkeeping ensures the same path will not be considered twice, and thus consecutive search iterations output different paths with increasing path costs. So, in case (i), DRPM terminates when k replicas have been placed on the k best hosts; and in case (ii), it terminates when $k' < k$ replicas have been placed, where k' is the maximum number of replicas that can be placed.

4 Decentralized Repair Method

Given a mechanism to replicate computations, we now model the repair problem itself as a distributed constraint optimization problem (DCOP) [4], to be implemented by agents themselves, following a leave or an entry in the system, to move some computations to restore the correct function of the system or to increase the quality of the distribution of the computations over agents.

4.1 DCOP Formulation

Let's first introduce some notations. We note X_c the set of candidate computations x_i that could or must be moved when the set of agents changes. For each of these computations, we note A_c^i the set of candidate agents that could host x_i. The set of all candidate agents, regardless of computations, is noted $A_c = \cup_{x_i \in X_c} A_c^i$ and X_c^m denotes the set of computations that agent a_m could host. Deciding which agent $a_m \in A_c$ hosts each computation $x_i \in X_c$ can be mapped to an optimization problem similar to the

one presented in Section 2, restricted to A_c and X_c: communication and hosting costs should be minimized while honoring the capacity constraints of agents.

To ensure that each candidate computation is hosted on exactly one agent, we rewrite constraints (1) for each $x_i \in X_c$:

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

Similarly, capacity constraints (2) can be reformulated as:

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

The hosting cost objective in (5) can be similarly formulated using one soft constraint for each candidate agent a_m:

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

Finally, the communication costs in (4) is represented with a set of soft constraints. For an agent a_m, the communication cost incurred by hosting a computation x_i can be formulated as the sum of the cost of the cut edges (x_i, x_j) from the computation graph $\langle \mathbf{X}, D \rangle$, (i.e. where $\mu^{-1}(x_j) \neq a_m$). Let's note N_i the neighbors of x_i in the computation graph. When a neighbor x_n is not a candidate computation (i.e. it might not be moved and $x_m \in N_i \setminus X_c$), the communication cost of the corresponding edge is simply given by $\mathbf{c_{com}}(i, j, m, \mu^{-1}(x_m))$. For neighbors that might be moved, the communication cost depends on the candidate agent that is chosen to host it and can be written as $\sum_{a_n \in A_c^j} x_j^n \cdot \mathbf{c_{com}}(i, j, m, n)$. With this we can write the communication cost soft constraint for agent a_m:

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

We can now formulate the repair problem as a DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{C}, \mu \rangle$ where $\mathcal{A}$ is the set of candidate agents A_c, $\mathcal{X}$ and $\mathcal{D}$ are respectively the set of decision variables x_i^m and their domain $\{0, 1\}$ and $\mathcal{C}$ is composed of constraints (6), (7), (8), and (9) applied for each agent $a_m \in A_c$. (6) and (7) result in infinite costs when violated, while (8) and (9) directly define costs to be minimized. The mapping function μ assigns each variable x_i^m to agent a_m.

4.2 Implementing Repair using a DCOP Solver

Now that our repair problem has been expressed as a DCOP, we discuss its resolution using a DCOP solution method. Several solution methods for DCOPs exist, like search

algorithms [4,6] and inference algorithms [8,12,3], to cite a few. In brief, using these message passing protocols (synchronous or not), agents coordinate to assign values to their variables. In our case, we opt for a lightweight, fast and iterative method, namely MGM-2. We select MGM-2 because its monotonic property, which implies that once the hard constraints (6) and (7) have been satisfied they will not be broken while MGM-2 optimize the soft constraints (8) and (9). By comparison, a stochastic algorithm like DSA for example, would not provide this guarantee. Additionally, most of the decisions to know who will host a computation are coordinated between two agents: to move a computation from one agent a_m to an agent a_p, the binary variable x_i^m must take 0 as a value, while *at the same time*, x_i^p must switch from 0 to 1. This need for simultaneous changes eliminates non-coordinated algorithms like MGM-1. Finally, MGM-2 is lightweight, which fits our target use-cases.

Combined with our replica placement method, we obtain a repair method, called DRPM+MGM-2 that can be used to adapt both to agent departure and arrival, by using the appropriate definitions of the sets A_c and X_c. Let's discuss agent departure case, since it is the most stressing situation. Assuming initial deployment and replica placement have been performed at system boot strap, the system will execute the following repair cycle all along its lifetime: (a) Detect departure/arrival; (b) Activate replicas of missing computations (using MGM-2); (c) Place new replicas for missing computations (using DRPM), and continue nominal operation.

Step (a) assumes some discovery and keep alive mechanisms that automatically inform some agents of any events in the infrastructure. So when an agent a_m fails or is removed, we consider all neighbor agents of a_m in the **route**–graph are aware of the departure. Step (b) relocates computations that were hosted on the set of departed agents A_d to other agents. The candidate computations are the orphaned computations hosted on these agents: $X_c = \cup_{a_m \in A_d} \mu(a_m)$. To avoid extra delay and communication during the repair phase these orphaned computations should be assigned to agents that already have the necessary information to run an active version of the computation. This means that the candidate agent for an orphaned computation x_i maps the set of still available agents hosting a replica for this computation: $A_c^i = \rho(x_i) \setminus A_d$. In a k-resilient system, as long as $|A_d| \leq k$, we are sure that there will always be at least one agent in A_c^i. Thus, step (b) yields an assignment of each of the orphaned computations to one of the remaining agents hosting its replica. Step (c) maintains a good resilience level in the system by repairing the replica distribution using DRPM on a smaller problem, since many replicas are already placed.

5 Experimental Evaluations

In this section we analyze both the quality of distributions after repair using DRPM+MGM-2 method, and the impact of the repair process on the performance of computations operated of the set of agents. Here, we choose to illustrate our repair framework on a DCOP solution method, namely MaxSum [3], as the process to be executed by our system, and factor graphs (FGs) as computation graphs to be deployed and repaired at runtime.

5.1 Experimental Setup

We run the experiments using our own multi-threaded python-based DCOP library, namely pyDCOP[3]. To evaluate our repair framework, each instance is composed of three components to generate: a multi-agent topology (the infrastructure), a problem topology (a FG), a disturbance scenario (a set of disappearance events).

We evaluate our algorithms on a *scale-free infrastructure* emulating a multi-agent system structured like a typical IoT environment with very few powerful and highly connected agents (cloud servers and calculators), some moderately powerful and connected agents (gateways and personal computers), and many slow and loosely connected nodes (the devices). For this reason, we use a scale-free graph model, as networks with power-law degree distribution are known to represent well this kind of setups [15].

Our test problem is made of $|\mathbf{V}| = 100$ decision variables v_k , with domain size $|D_k| = 10$, and binary constraints. These variables are the nodes of a scale-free graph generated using the Barabási–Albert model [1] (with [7], starting with two nodes and iteratively adding nodes attached to two existing nodes). Edges of this graph represent the binary constraints of our problem. We have $2\cdot(|\mathbf{V}|-2)+1=197$ constraints in our case. Their cost function is generated using uniform random integer in $U[0,100]$ for each possible assignment. As we are using MaxSum, we have $|\mathbf{X}|=|\mathbf{V}|+2\cdot(|\mathbf{V}|-2)+1=297$ computations x_i in our factor graph, that must distributed on the infrastructure's agents. Notice that while we are using MaxSum to solve this IoT like problem, the repair of the distribution of this problem is performed using MGM-2, as discussed in section 4.2.

The infrastructure is made of $|\mathbf{A}| = 100$ agents, each holding one of the decision variables, and is defined by $\mathbf{c_{host}}$, $\mathbf{route}$, $\mathbf{w_{max}}$, $\mathbf{w}$ and $\mathbf{msg}$ as follows:

- Hosting costs $\mathbf{c_{host}}(a_m,x_j)$ are random in $U[0,10]$ except for the computation x_i, responsible for the variable v_i, initially hosted on a_i : $\mathbf{c_{host}}(a_i,x_i)=0$.
- Route costs $\mathbf{route}(a_m,a_n)$ are defined in a way that respects the scale-free distribution of the graph, and the analogy with IoT systems: agents with many neighbors have a low communication while agents few neighbors have an higher communication cost. More precisely, $\mathbf{route}(a_m,a_n) = \frac{1+||N(a_m)|-|N(a_n)||}{|N(a_m)|+|N(a_n)|}$ where $|N(a_i)|$ is the number of neighbors of a_i in the scale free graph.
- The capacity of each agent depends on the weight of its decision variable and is set to a large value, to ensure that all replicas can be hosted and k-resiliency is possible, even after several repairs: $\mathbf{w_{max}}(a_i)=100*\mathbf{w}(x_i)$.
- Finally, $\mathbf{w}$ and $\mathbf{msg}$ depends on the solution method used. As we are using Max-Sum, the size of the messages is a direct function of the size of domain of the variable: $\mathbf{msg}(i,j)=|D_j|=10$ and the weight of variable and factors computations is respectively proportional to the size of the variable's domain and the sum of the size of the linked variable's domains.

Initially, each variable (decision) is assigned to an agent, and factors are optimally placed using the LP discussed in Section 2. We use $\omega=0.5$.

We generate disturbance scenarios as sequences of perturbation events happening every 30 seconds. At each such event, 3 randomly chosen agents disappear as to analyze

[3] https://github.com/Orange-OpenSource/pyDcop

Event #	1	2	3	4	5	end
Pertubation	6152	6234	6257	6408	6469	6128
No perturbation	5784	5582	5672	5644	5638	5623
Variation (%)	6.4	11.7	10.3	13.5	14.7	8.9

Table 1: Average maximum solution costs after repair and comparison with the cost without perturbation

the impact on DCOP solution methods, and observe the 3-resilience of our system. We generate 20 instances (infrastructure, problem and scenario), and run the scenario 5 times for each instance. Additionally we also solve the same problems (5 runs for each of the 20 instances) without any disturbance, in order to assess the impact of our repair methods on the quality of the solution returned by MaxSum. In both cases results are averaged over all instances. Experiments are performed on an Intel core i7 CPU with 16GB RAM.

5.2 Impact of repairing on DCOP Operations

Here, we look at the runtime state of process (cost of the current MaxSum solution) with and without disturbance, to analyze how DRMP+MGM-2 changes the operation of the running process, on our *scale-free* problems.

Figure 2 shows the cost of the solution found by MaxSum over time. The cost of each of the 100 runs is displayed in blue with low opacity, the overall shapes illustrates the fact that the system's behavior is consistent across the various instances. The blue and red lines represent respectively the average cost of the solution when running with perturbations and without perturbation. The five yellow areas denotes the repair processes due to the removal of agents ; during these periods some variables are temporarily not hosted and we cannot compute the correct cost of the solution, which explain the flat lines at these points. We evaluated the average time to repair a distribution at less than 10 seconds.

We can see that the solutions on the disturbed system degrade when agents are removed, but improve again when the system recovers. This can be explained by the fact that the replicas that are activated by the repair process, as opposed to the computation that were hosted on removed agents, do not have any accumulated knowledge. In belief propagation algorithms like MaxSum, computations are not really stateless, as required by our approach of k-resilience (see section 3): they accumulate information about constraints and preferences from their neighbors. When activating a replica, the new active computation start afresh and an indeterminate number of message rounds are needed to restore that information. When given more time after a disturbance, 90 seconds after the last repair on our experiment, the systems recovers and the quality loss decrease to 8%.

Table 1 shows the maximum cost of the solution after each event and repair process, and the average cost at the same time when solving the problem without any perturbation. As we can see, these results show that our repair scheme has a relatively low impact on MaxSum operation since costs remain in average very close (6 to 14% compared to solutions without disturbance).

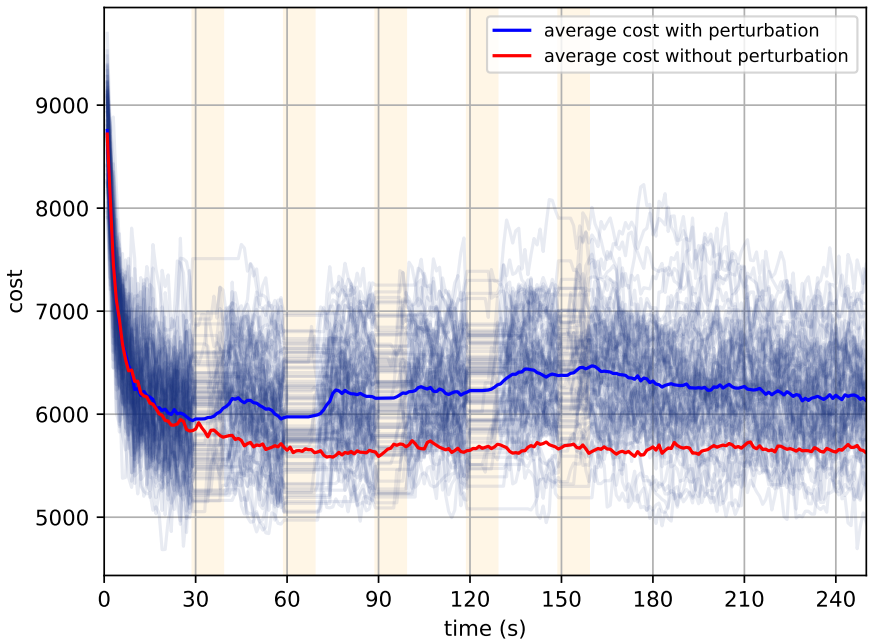

Fig. 2: Average cost of MaxSum solution at runtime, on scale-free DCOPs, with (blue) and without perturbation (red).

5.3 Quality of Repaired Distributions

To evaluate the quality of our repaired distributions, we measure the degradation of the distribution all along the system lifetime. At each event, we assess the cost of the current distribution using equations 4 and 5, against the initial distribution cost (which is optimal, but cannot be computed at runtime). Figure 3 shows the distribution costs for the 100 runs. As the global distribution cost is made of communication and hostings costs, we also plot these two costs independently. We can see the global cost increases when repairing the system, which was expected as the initial distribution is optimal while repair distribution are approximate optimization, where for each orphaned computation, replacement agents can only be selected among those hosting a replica, placed using another approximate algorithm: DRPM. More surprisingly, the communication cost increases when repairing, while the hosting cost decreases. Our explanation for this phenomenon is that the hosting constraints 8 are relatively simple and always have a limited number of binary variables in their scope. On the other hand, communication constraints 9 are much more complex and might have high arity; we experimentally counted more than 20 variables in some cases. When solving the repair DCOP with MGM-2, the algorithm struggles with these high arity constraints, and hence favor hosting at the expense of communication costs.

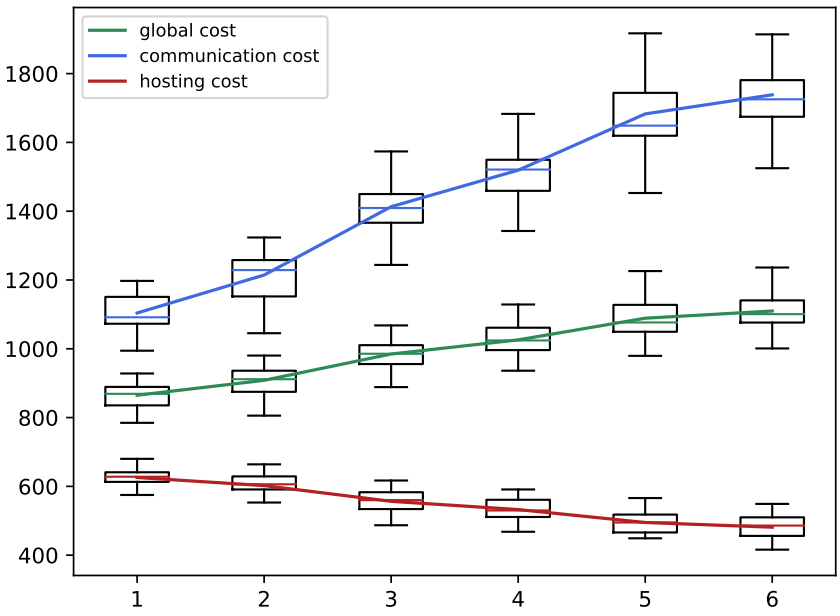

Fig. 3: Cost of computation distribution after each event

Overall, we can observe that our repair method results in quite good quality distribution, with 25% increased cost in average compared to the initial optimal distribution, even after removing 15 agents over 100, which represents an important portion of the initial system.

6 Conclusions

We investigated the resilient distribution of computations over a dynamic set of agents, and two distributed algorithms have been proposed: (i) a replica placement protocol (DRPM) and (ii) a repair protocol (DRPM+MGM-2) relying on replicas placed by DRMP and based on classical DCOP solution method MGM-2. Our contributions have been evaluated experimentally through operation of MaxSum on a dynamic multi-agent systems where nodes disappears during the optimisation process. On IoT-based infrastructures, operating MaxSum is not much impacted by our repair methods, and the systems continue providing solutions, whilst agents are disappearing, which demonstrates the resilience of these systems.

In this paper, whilst providing promising preliminary results, we only explored a reduced portion of all the possible parameter settings (k, $|\mathbf{A}|$, $|\mathbf{X}|$, costs functions, etc.). We will conduct a more detailed experimental evaluation, and implement a parameter space procedure to evaluate deeper our contributed repair framework. Moreover, since we only focused on the worst scenario with agent removals only, we will investigate more

realistic scenarios where agents may be added to replace disappeared ones. Finally, this work originated from a concrete need to deploy agents operating DCOPs on real devices. So our future step is to evaluate DRPM+MGM2 on real IoT environments structured like the synthetic ones we analyzed here.

References

1. Albert, R., Barabási, A.L.: Statistical mechanics of complex networks. Rev. Mod. Phys. 74, 47–97 (Jan 2002), https://link.aps.org/doi/10.1103/RevModPhys.74.47
2. Bonomi, F., Milito, R., Zhu, J., Addepalli, S.: Fog computing and its role in the internet of things. In: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing. pp. 13–16. MCC '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2342509.2342513
3. Farinelli, A., Rogers, A., Petcu, A., Jennings, N.R.: Decentralised coordination of low-power embedded devices using the max-sum algorithm. In: International Conference on Autonomous Agents and Multiagent Systems (AAMAS'08). pp. 639–646 (2008), http://dl.acm.org/citation.cfm?id=1402298.1402313
4. Maheswaran, R., Pearce, J., Tambe, M.: Distributed algorithms for dcop: A graphical-game-based approach. In: Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems (PDCS), San Francisco, CA. pp. 432–439 (2004)
5. Malewicz, G., Austern, M.H., Bik, A.J., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. pp. 135–146. SIGMOD '10, ACM (2010), http://doi.acm.org/10.1145/1807167.1807184
6. Modi, P., Shen, W., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence Journal (2005)
7. NetworkX: Software for complex networks. https://networkx.github.io [Online (accessed Jan 22, 2018)] (2018)
8. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: International Joint Conference on Artificial Intelligence (IJCAI'05). pp. 266–271 (2005)
9. Russel, S., Norvig, P.: Artificial Intelligence: a Modern Approach. Prentice-Hall, 3rd edn. (2009)
10. Rust, P., Picard, G., Ramparany, F.: On the deployment of factor graph elements to operate max-sum in dynamic ambient environments. In: 8th International Workshop on Optimisation in Multi-Agent Systems (OPTMAS 2017, in conjunction with AAMAS 2017) (2017), https://www.cs.nmsu.edu/~wyeoh/OPTMAS2017/
11. Saraç, T., Sipahioglu, A.: Generalized quadratic multiple knapsack problem and two solution approaches. Computers & Operations Research 43(Supplement C), 78 – 89 (2014), http://www.sciencedirect.com/science/article/pii/S0305054813002244
12. Vinyals, M., Rodriguez-Aguilar, J.A., Cerquides, J.: Constructing a unifying theory of dynamic programming dcop algorithms via the generalized distributive law. Autonomous Agents and Multi-Agent Systems 22(3), 439–464 (2010), http://dx.doi.org/10.1007/s10458-010-9132-7
13. Wattenhofer, R.: Principles of Distributed Computing (2015)
14. Wolfson, O., Milo, A.: The multicast policy and its relationship to replicated data placement. ACM Trans. Database Syst. 16(1), 181–205 (Mar 1991), http://doi.acm.org/10.1145/103140.103146
15. Yao, B., Liu, X., Zhang, W.J., Chen, X.E., Zhang, X.M., Yao, M., Zhao, Z.X.: Applying graph theory to the internet of things. Proceedings - 2013 IEEE International Conference on High

Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013 pp. 2354–2361 (2014)