Resilient Distributed Constraint Reasoning to Autonomously Configure and Adapt IoT Environments

PIERRE RUST, Orange Labs, France
GAUTHIER PICARD, ONERA/DTIS, Université de Toulouse, France
FANO RAMPARANY, Orange Labs, France

In this paper, we investigate multi-agent techniques to install autonomy and adaptation in IoT-based smart environment settings, like smart home scenarios. We particularly make use of the smart environment configuration problem (SECP) framework, and map it to a distributed optimization problem (DCOP). This consists in enabling smart objects to coordinate and self-configure as to meet both user-defined requirements and energy efficiency, by operating a distributed constraint reasoning process over a computation graph. As to cope with the dynamics of the environment and infrastructure (e.g. by adding or removing devices), we also specify the *k*-resilient distribution of graph-structured computations supporting agent decisions, over dynamic and physical multi-agent systems. We implement a self-organizing distributed repair method, based on a distributed constraint optimization algorithm to adapt the distribution as to ensure the system still performs collective decisions and remains resilient to upcoming changes. We provide a full stack of mechanisms to install resilience in operating stateless DCOP solution methods, which results in a robust approach using a fast DCOP algorithm to repair any stateless DCOP solution methods at runtime. We experimentally evaluate the performances of these techniques when operating stateless DCOP algorithms to solve SECP instances.

CCS Concepts: • **Computing methodologies → Multi-agent systems**; **Self-organization**; • **Theory of computation → Discrete optimization**; • **Human-centered computing → Ubiquitous and mobile computing**.

ACM Reference Format:
Pierre Rust, Gauthier Picard, and Fano Ramparany. 2022. Resilient Distributed Constraint Reasoning to Autonomously Configure and Adapt IoT Environments. *ACM Trans. Internet Technol.* 1, 1 (January 2022), 31 pages.

1 INTRODUCTION

Due to recent advances in technology and the decrease of the cost of embedded computing, both on the CPU and communication side, the ideas pioneered in academic works like *Pervasive Computing* and *Ubiquitous Computing* are currently materializing into real world solutions, which are generally referred to as the *Internet of Things (IoT)*. In these systems many connected devices, typically equipped with computation capabilities, are used together and allow building high-level services, which can be aware of, and act on, the real world. One specific use case for these systems is IoT-based *Ambient Intelligence (AmI)*, where this technology is used to facilitate the life of users by embedding computation, sensors and actuators in the environment. This area has been especially active in the past years, both in the industrial and the enthusiasts communities, with the emergence of many *Smart Home Environment (SHE)* solutions that seek to realize this idea in the home context. However, these systems are still in their infancy and their builders face many issues and challenges in order to reach the full potential of these new technical capabilities.

We envision an AmI setup as a distributed system where devices, which we consider as *agents*, have to cooperatively, autonomously and dynamically come up with a collective behavior that facilitates the life of the user. This solution will typically take the form of a *spontaneous configuration* of the environment, for example by setting, depending of various

Authors' addresses: Pierre Rust, Orange Labs, Rennes, France, pierre.rust@orange.com; Gauthier Picard, ONERA/DTIS, Université de Toulouse, France, Toulouse, gauthier.picard@onera.fr; Fano Ramparany, Orange Labs, Grenoble, France, fano.ramparany@orange.com.

2022. Manuscript submitted to ACM

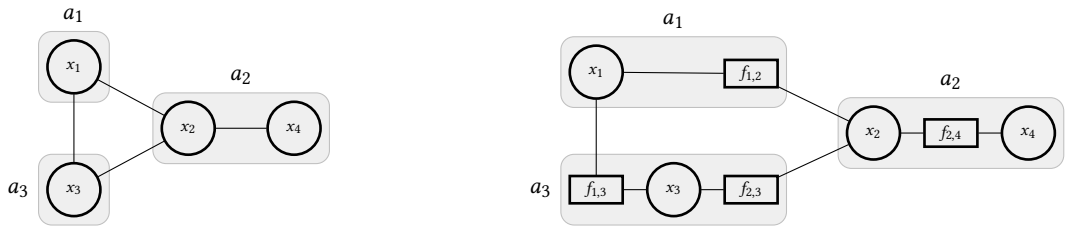
conditions, appropriate light, heat, humidity, etc. levels in the home. In any meaningful environment, many devices act on the same parameter (e.g. several light sources are used in the same area) and some coordination is required among them.

Modeling Goal-Oriented Smart Home Scenarios. The first question we wanted to answer in this research is how to model a SHE in a way that really matches the vision of AmI. One key element for that objective is that the user should not need to care about the inner workings of the system: detailed and manual configuration must be avoided in favor of the simple specification of goals. The devices should then autonomously decide how to reach these goals in the best possible way. To answer this challenge, we modeled coordination in an AmI environment as an optimization problem within the Distributed Constraint Optimization Problem (DCOP) framework. We coined the resulting model the Smart Environment Configuration Problem (SECP) [29].

Installing Decentralized Coordination in the Real World. Following the first challenge, an important question to answer is how to install such distributed model on the real devices that the SHE is made of. Indeed, most current research on the DCOP framework is based on assumptions that do not hold when applying the model to real world situations like IoT-based AmI –e.g. only one variable per agent, and non explicit deployment of constraints. When solving an optimization problem distributively, the decision making process is implemented through computations that perform the optimization process and must physically run on some hardware substrate. As a consequence, we argue that in order to apply the DCOP framework to real world problems, these decision making computations must be distributed on the agents/devices. This distribution must take into account the characteristics of the target environment, which encompass, for the SECP, the limited capabilities of the devices and the constrained communication. We thus proposed a definition of optimal distribution, for SECP and for more generic IoT systems, and developed optimal and heuristic approaches for computing this distribution, based on [30].

Providing Resilience in Decentralized Decision Making. The goal of IoT systems is to facilitate the life of its users. Therefore, an important challenge of such systems is ensuring reliability and resilience. However, a SHE is also a dynamic and open system: the characteristics of the problem may change at runtime and devices may join or leave the system at any moment. Thus, the question is how to ensure that the system keeps providing the services required by the users, with the equivalent quality of service and quality of experience, whilst the infrastructure is changing. For this purpose, we defined the concept of k-resilience and proposed several distributed solution methods to achieve this, by using replication-based repair techniques [32].

All these models and techniques compose a full stack of tools to install self-configuration, adaptation and resilience to IoT-based environments. The paper presents these contributions as follows. Section 2 first presents the core concepts related to distributed constraint optimization and some dedicated solution methods. Section 3 expounds the SECP model, which defines the problem to be cooperatively solved by a set of devices deployed in a smart environment. Section 4 specifically focuses on techniques to deploy the resulting decisions and computations over a set of smart devices, depending on their capabilities and connectivity. Section 5 next provides methods to adapt a distribution of computations to the dynamics of the infrastructure, e.g. when devices disappear due to failures and disconnection. We assess the different investigated models and techniques through experimental evaluation on simulated smart house environments, in Section 6, and briefly showcase a physical demonstrator made of a network of Raspberry Pis. We discuss related works in Section 7 before concluding with some perspectives in Section 8.



(a) Constraint graph

(b) Factor graph

Fig. 1. Graphical representations for DCOP

2 BACKGROUND ON DISTRIBUTED CONSTRAINT OPTIMIZATION

Let us introduce the overarching framework of our contributions: the Distributed Constraint Optimization Problem (DCOP) framework.

2.1 The DCOP Framework

While solution methods for standard Constraint Satisfaction Problem (CSP) and Constraint Optimization Problem (COP) [3] are usually centralized, an extension of these topics has emerged in the Multi-Agent Systems (MAS) research eco-system, where the constraint reasoning process is distributed among agents. Each agent has control over some of the variables and agents interact to find a solution to the problem. Distributed Constraint Satisfaction Problem (DisCSP) is the distributed counterpart of CSP and only considers hard constraints to formalize distributed problem solving [38]. It was later extended, and superseded in most works, by the DCOP framework.

Definition 1 (DCOP). *A discrete Distributed Constraint Optimization Problem (DCOP) is formally represented by a tuple* $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \alpha \rangle$*, where* $\mathcal{A} = \{a_1, \ldots, a_{|A|}\}$ *is a set of agents;* $\mathcal{X} = \{x_1, \ldots, x_n\}$ *are discrete variables, owned by the agents;* $\mathcal{D} = \{\mathcal{D}_1, \ldots, \mathcal{D}_n\}$ *is a set of finite domains, such that variable* x_i *takes values in* $\mathcal{D}_i = \{v_1^i, \ldots, v_k^i\}$*;* $\mathcal{F} = \{f_1, \ldots, f_m\}$ *is a set of soft constraints, where* f_i *is a function that defines a cost* $\in \mathbb{R} \cup \{\infty\}$ *for each combination of values to the variables in its scope;* $\alpha : \mathcal{X} \rightarrow \mathcal{A}$ *is a function that assigns the control of each variable to an agent.*

A solution to a DCOP is a complete assignment σ *that minimizes a global objective function* $F(\sigma)$ *that aggregates the individual costs* f_i*'s:* $\sigma^* = \text{argmin}_\sigma F(\sigma)$*. The sum is generally used as an aggregation function:* $\sigma^* = \text{argmin}_\sigma \sum_{f_i \in \mathcal{F}} f_i$*.*

Without loss of generality, the notion of cost can be replaced by the notion of utility $\in \mathbb{R} \cup \{-\infty\}$. In this case, solving a DCOP is a maximization problem of the overall sum of utilities. Moreover, DCOPs are also often represented using graphical models. Constraint graphs are commonly used to represent DCOPs where variables are represented as nodes and binary constraints as edges, with the addition of *enclosing nodes* (a.k.a. compound nodes) representing the agents and the variables they are responsible for –see Figure 1a. These nodes form a communication graph among agents. Factor graphs are also used for DCOPs. As constraints (a.k.a. factors) are explicitly represented in these graphs, they must also be attached to agents, as represented on Figure 1b.

2.2 DCOP Solution Methods

Like with classical constraint optimization, finding a solution for a DCOP is NP-hard in the general case. A large number of algorithms, complete and incomplete, many of which have several variants, have been proposed by the research community. In this section, we will not be able to give a detailed description of all algorithms, we redirect the reader

to [8] for a very comprehensive survey of DCOP and to the original papers for each algorithm. We thus focus on the algorithms that have been used the most during this study, namely DSA, MGM, and Max-Sum.

Distributed Stochastic Algorithm (DSA) is a family of incomplete, local search, very lightweight algorithms based on a rather simple idea: agents start with a random value from their domain and regularly evaluate if the quality of their own partial assignment, defined as the total values of those constraints in which they are involved, could be improved by selecting a new value [39]. This evaluation is based on the knowledge of the values currently selected by their neighbors (defined by the constraint graph). If this quality can be improved, the agent decides randomly, with an *activation probability p*, to select the corresponding value and send its updated state to its neighbors. Although not strictly *anytime*, DSA is an *iterative* algorithm and can be used to obtain a complete assignment at any time, in real time, with a solution quality improving, on average, over time. However, in the general case DSA provides no guarantee of monotonicity: as there is no coordination in the decision process and an agent's local knowledge may be outdated, two agents may simultaneously take contradictory decisions, resulting in a decrease in the overall result's quality. Five variations of this basic principle have been studied, depending on the strategy used for values change. DSA-B is considered to be the most efficient approach in the general case. The value used for activation probability p has also been shown in [39] to have a huge influence in DSA's efficiency and quality and exhibits phase transition property. When the right variant and activation probability have been selected for a given problem class, DSA provides very good quality results, with minimal network and computational load, which makes it highly scalable. Asynchronous Distributed Stochastic Algorithm (A-DSA) is the asynchronous version of DSA [9].

Maximum Gain Message (MGM) implements a gain message-passing protocol [18]. Like DSA, MGM algorithm is an incomplete local search algorithm that can handle n-ary constraints. MGM is a synchronous algorithm: at each round, agents compute the maximum change in quality, named *gain*, they could achieve by selecting a new value and send this gain to their neighbors. An agent is then allowed to change its value only if its gain is larger than the gain received from all its neighbors. This mechanism ensures that two variables involved in the same constraint will never change their value in the same round. This process repeats until a termination condition is met. MGM is able to guarantee monotonicity; eliminating the stochastic aspect of DSA ensures that the solution quality only improves over time. Monotonicity is a very interesting feature, at the expense of a higher tendency to become trapped in a local minima. To mitigate this issue, there exists a coordinated version of MGM, usually Maximum Gain Message with 2-coordination (MGM-2), but it can be extended to MGM-k, where k agents can coordinate a simultaneous (i.e. in the same round) change of values. Asynchronous Monotonic Distributed Local Search (AMDLS) [26] has recently been proposed and is similar to MGM but does not support coordination like MGM-k.

MaxSum is a inference-based incomplete algorithm [5]. It is a derivative of the *max-product* message passing algorithm in the logarithmic space. MaxSum is complete on acyclic constraint graphs, but approximate on cyclic graphs. It performs a marginalization process of the cost functions, and optimizing the costs for each given variable. The value assignments take into account their impact on the marginalized cost function. It operates on a factor graph and cost messages flow on the edges, from factors to variables, and vice versa. When a factor or a variable computes twice the same message for the same recipient, it stops propagation and the algorithm converges if all message propagation has stopped. Termination is usually implemented by both observing convergence and by force-stopping propagation after a predetermined number of rounds. By simply summing its incoming messages an agent can assess *at any time* an approximation of the marginal function of its variables and select the values that maximize the social welfare in the system by finding the argmax of this marginal function. This also means that MaxSum can be used to get a continuously updated solution, without waiting for termination, even in dynamic problems. Empirical evaluations show that it can compute very good quality

solutions with acceptable computation load compared to representative complete algorithms. It should be noted that it can also be implemented asynchronously, as stated in [5], with agents emitting updated messages whenever they receive an update from on of their neighbors. We denote this variant Asynchronous MaxSum Algorithm (A-MaxSum). MaxSum is also well suited to dynamic settings, as the agents can maintain an up-to-date estimate of the current state's utility by continuously emitting update messages.

3 MODELING SMART ENVIRONMENT CONFIGURATION PROBLEMS AS DISTRIBUTED CONSTRAINT OPTIMIZATION PROBLEMS

In this section, we start from a sample SHE scenario in order to illustrate the coordinated behaviors our model should implement in AmI systems and introduce notations required to map these behaviors to an optimization problem, as presented in [29]. We then show how this optimization problem can be solved in a distributed setting such as AmI scenarios. As SHEs can naturally be represented as MAS, we map our SECP optimization problem to a DCOP.

3.1 Sample Ambient Intelligence Scenario

We consider the following AmI scenario. Our system is a smart home, composed of many connected devices: light bulbs, roller shutters, TV sets, luminosity sensors and presence detectors, etc. The overall objective of our system is to maintain a luminosity level in the rooms of the house that satisfies the inhabitants. These users can express their wishes by configuring simple behaviors, aka *scenes*, using an application on a user interface device. These scenes can use the values of sensors or the states of some devices as triggers for setting a specific luminosity goal in a given area. Such a configuration can be expressed by a rule as represented in Example 1, although the users would of course not write it in this form, but more probably use some kind of graphical representation to express it.

Example 1 (Scene specification). Rule (1) defines a scene, triggered only when the light level of the living room is less than 60, where that light level should be set at 60 and the shutter of the living room should be closed:

```
IF   presence_living_room      =   1
AND  light_sensor_living_room  <   60
THEN light_level_living_room   ←   60
AND  shutter_living_room       ←   0
```
(1)

One important characteristic of such rules is that they do not contain the list of actions required, but only the objective requested by the user. This means that our AmI system will need to figure out the best actions that would lead to meet the requested objectives. It also means that the system uses whatever devices available when the scene is triggered: devices may become faulty and be added or removed, this will automatically be taken into account when searching for the solution. Finally, we want our system to select the most energy-saving configuration for a given scene.

Given this scenario, we want the *devices* of this system to self-organize and cooperate autonomously to reach the user-defined objectives, avoiding any dedicated device whose purpose would be to gather inputs from sensors and decide the sequence of actions required to fulfill these objectives.

Each of the connected devices in the AmI environment acts as a *sensor* or an *actuator*. Devices' capabilities and locations can be used to match candidate devices with a user-defined objective. We consider that a discovery mechanism is available and that the system knows at any given time the list of available devices with their characteristics. Such mechanisms are a common requirement in dynamic open environments and are generally based, in current systems, on mDNS [13] and DNS-SD [12]. As a consequence, we concentrate here on the coordination mechanism. We consider

each device in the system as an *agent* in a MAS and will devise ways of implementing coordination among these agents in a dynamic open system.

Finally, note that while we concentrate here, for simplicity's sake, on devices that can influence on the luminosity of the environment, this approach could be used for many other environmental settings and more generally for almost any behavior in an AmI environment.

3.2 Notations for SECP

Our AmI scenario can be seen as an optimization problem with values to assign to actuators, whilst maximizing the adequacy to user-defined scenes and minimizing the overall energy consumption.

3.2.1 Actuators. Let $\mathfrak{A}$ be the set of available actuators. We note $\mathcal{X}(\mathfrak{A})$ the set of variables x_i stating the values of actuators $i \in \mathfrak{A}$. We use $\mathbf{x}_i$ to refer to a possible state of $x_i \in \mathcal{X}(\mathfrak{A})$ (i.e. the value assigned to the variable x_i), that is $\mathbf{x}_i \in \mathcal{D}_{x_i}$ where $\mathcal{D}_{x_i}$ is the domain of x_i and contains values mapping to the actions available on the actuator i. For a light bulb for example, this is the list of light levels that can be emitted by the bulb

Each actuator i has a cost to be activated, noted $e_i : \mathcal{D}_{x_i} \rightarrow \mathbb{R}^+$. This cost can be directly derived from the consumption law of each device. We note $\mathcal{F}(\mathfrak{A}) = \{e_i | i \in \mathfrak{A}\}$ the set of costs for all actuators. Among the possible values, every actuator i has a possible "switched off" state value, noted $\mathbf{0} \in \mathcal{D}_{x_i}$, with an associated cost (most probably 0).

3.2.2 Sensors. Similarly, we note $\mathfrak{S}$ the set of available sensors, and $\mathcal{X}(\mathfrak{S})$ the set of variables s_l encapsulating their states. Each variable s_l take its value in a domain $\mathcal{D}_{s_l}$. We note $\mathbf{s}_l \in \mathcal{D}_{s_l}$ the current state of sensor $\ell \in \mathfrak{S}$. Sensor values reflect the state of the environment and are not directly controllable by the system: therefore they are *read-only* values and do not have a cost function.

3.2.3 Environment State. In order for the user to set their objectives (e.g. requesting a given light level in a given room), we also need to model the state of the environment. We note Φ the set of states of the environment, and $\mathcal{X}(\Phi)$ the state of variables y_j encapsulating these states. Each variable y_j takes its value in a domain $\mathcal{D}_{y_j}$ and we note $\mathbf{y}_j \in \mathcal{D}_{y_j}$ the current value of y_j. Like sensors, the environment's state is of course not directly controllable by the system.

3.2.4 Scenes. Let $\mathfrak{R}$ be the set of user-defined scene rules. Each scene k is specified as a condition-action rule expressed using the set of available devices $\mathfrak{A} \bigcup \mathfrak{S}$ (actuators and sensors).

The **action** part of scenes defines objectives by setting *target values* to *scene action variables*. These scene action variables can represent either some actuators or the state of the environment:

(1) When the rule requests some direct action on an actuator, the scene action variable is the variable representing the state $x_i \in \mathcal{X}(\mathfrak{A})$ of that actuator. This is the case in the rule of Example 1, which explicitly requires to close the shutter of the living room. We note $\mathbf{x}_i^k$ the target value defined by the user for the scene action variable x_i in the rule k, with $\mathbf{x}_i^k \in \mathcal{D}_{x_i}$ for all i and k.
(2) When the rule sets a target state for the environment the scene action variable is the variable $y_j \in \mathcal{X}(\Phi)$ representing that state of the environment. For example, the rule in Example 1 sets a target light level in the living room, which can be acted on by the light bulb actuators located in this room. We note $\mathbf{y}_j^k$ the target value defined by the user for the scene action variable y_j in the rule k, with $\mathbf{y}_j^k \in \mathcal{D}_{y_j}$ for all j and k.

The **condition** part of a scene is specified as a conjunction of boolean expressions using state of actuators, $x_i,\ i \in \mathfrak{A}$, or state of sensors, $s_l,\ l \in \mathfrak{S}$ and binary predicates (e.g. >, <, =). A scene rule can be either *active* or *inactive* depending on the state of devices appearing in the condition part of the rule.

3.2.5 Modeling Physical Constraints. In order for the system to be able to select the right actions to achieve the goals set by the rules, we must be able to reason upon the link between the actuators' actions and the state of the environment, which means we need a model of the physical interactions happening in the real world. For this purpose, we define functions that we call *physical models*, noted ϕ_k, where $S_{\phi_k} \subseteq \mathcal{X}(\mathfrak{A})$ is the scope of the model, i.e. the set of actuator variables influencing one particular aspect of the environment, and $\mathcal{D}_{\phi_k}$ is the domain of the variable representing the corresponding state of the environment.

$$\phi_k : \prod_{\varsigma \in S_{\phi_k}} \mathcal{D}_\varsigma \rightarrow \mathcal{D}_{\phi_k} \tag{2}$$

Example 2 (Physical model). We can consider that the level of light y_1 in a room depends on the total power of "light-emitting" devices located installed in the room, i.e. bulbs x_1 and x_2, and a roller shutter x_3: $y_1 = \phi_1(x_1, x_2, x_3) = 30x_1 + 30x_2 + 10x_3$. Weights assigned to each x_i are related to the luminous efficacy of each device [35].

Let $\overline{\phi_j} = |S_{\phi_j}|$ the arity of ϕ_j, and $\mathcal{F}(\Phi) = \{\phi_j\}$ be the set of all physical models between actuators and rule-defined values.

Note that a physical model function output value is considered here to be an *estimation* (or *prediction*) of the value of some y_j, based on the state of the actuators. The quality of the resulting system configuration depends tightly on the quality of this estimation, and thus on how we define these physical model functions. These functions could be assessed using physical laws, calibration and machine learning ; however in this work we consider that the functions of these physical models are known and can be used directly.

3.3 Formulating SECP as an Optimization Problem

Now that we have a definition of the impact of the actuators' action on the environment, we are able to assess if the objective of a given rule is met. For this purpose, we define for each scene a utility function, noted r_k, with $S_{r_k} \subseteq \mathcal{X}(\mathfrak{A}) \cup \mathcal{X}(\Phi) \cup \mathcal{X}(\mathfrak{S})$ being the scope of the rule, made of the actuators, sensors and scene action variables used in the rule:

$$r_k : \prod_{v \in S_{r_k}} \mathcal{D}_v \rightarrow \mathbb{R}$$

The more the states of the scene action variables (from $\mathcal{X}(\mathfrak{A})$ and $\mathcal{X}(\Phi)$) are close to the user's target values for this scene, the higher the utility. Moreover, if the conditions to activate the rule (from $\mathcal{X}(\mathfrak{A})$ and $\mathcal{X}(\mathfrak{S})$) are not met, the utility should be neutral, i.e. equals to 0. We can therefore consider r_ks to be functions of the distance between the states of the scene action variables x_is (resp. y_js) and the target values $\mathbf{x}_i^k$ (resp. $\mathbf{y}_j^k$). We note $\mathcal{F}(\mathfrak{R}) = \{r_k | k \in \mathfrak{R}\}$ the set of rule utility functions.

Example 3 (Scene rule utility). Let us consider the rule from Example 1, where s_1 is the value of the presence sensor. Here a possible utility function, which is the negated *distance* between the current value of y_1 and x_3 and their target

values $\mathbf{y}_1^1 = 60$ and $\mathbf{x}_3^1 = 0$:

$$r_1(y_1, s_1, x_3) = \begin{cases} -|y_1 - 60| + x_3 & \text{if } s_1 = 1 \\ 0 & \text{otherwise} \end{cases}$$

Using these notations, we can express the SECP as an optimization problem. Our goal is to maximize the utility of the user-defined rules, while at the same time minimizing the energy consumption of actuators. We must of course also honor the real world physical constraints, which are not something we can actually optimize and must be modeled as hard constraints. Based on this, we can straightforwardly map the SECP to a multi-criteria optimization problem.

$$\begin{aligned} &\underset{x_i \in \mathcal{X}(\mathfrak{A})}{\textbf{minimize}} \sum_{i \in \mathfrak{A}} e_i \quad \text{and} \quad \underset{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}}{\textbf{maximize}} \sum_{k \in \mathfrak{R}} r_k \\ &\textbf{subject to} \;\; \phi_j(v_j^1, \ldots, \overline{v_j^{\phi_j}}) = y_j \quad \forall y_j \in \mathcal{X}(\Phi) \end{aligned} \tag{3}$$

Given all the previous concepts and notations, we define the SECP as follows:

Definition 2 (Smart Environment Configuration Problem (SECP)). *Given a set of actuators $\mathfrak{A}$, and their related costs $e_i \in \mathcal{F}(\mathfrak{A})$, a set of sensors $\mathfrak{S}$, a set of scene rules $\mathfrak{R}$ and their related utility functions in $r_k \in \mathcal{F}(\mathfrak{R})$, a set of environment states Φ, and a set of physical dependency models $\mathcal{F}(\Phi)$, the* Smart Environment Configuration Problem *(or* SECP*) is represented by a tuple $\langle \mathfrak{A}, \mathcal{F}(\mathfrak{A}), \mathfrak{S}, \mathfrak{R}, \mathcal{F}(\mathfrak{R}), \Phi, \mathcal{F}(\Phi) \rangle$ and amounts to finding the configuration of actuators that maximizes the utility of the user-defined rules, whilst minimizing the global energy consumption and fulfilling the physical dependencies.*

3.4 Mapping the SECP to a DCOP

SECP is currently formulated as a multi-objective optimization problem, which is not convenient when mapping to a DCOP. We could use a Multi-objective DCOP (MO-DCOP) but solution methods for this DCOP extension are quite heavy and would not fit our target environment, composed of constrained devices. Instead, we choose to aggregate the two objectives to formulate the problem as a mono-objective optimization problem, using weights $\omega_u > 0$, $\omega_c > 0$:

$$\begin{aligned} &\underset{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}}{\textbf{maximize}} \; \omega_u \sum_{k \in \mathfrak{R}} r_k - \omega_c \sum_{i \in \mathfrak{A}} e_i \\ &\textbf{subject to} \;\; \phi_j(v_j^1, \ldots, \overline{v_j^{\phi_j}}) = y_j \quad \forall y_j \in \mathcal{X}(\Phi) \end{aligned} \tag{4}$$

As described in Section 2.1, a DCOP is represented by a tuple $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, \mathcal{F}, \mu \rangle$, therefore to map the SECP to a DCOP we need to define the sets of agents $\mathcal{A}$, variables $\mathcal{X}$, domains $\mathcal{D}$ and constraints $\mathcal{F}$ and the mapping function α.

3.4.1 Agents. In a DCOP, agents control decision variables and are responsible for selecting an appropriate value for each of the variable they own. This means that in a physical system agents are entities, embodied by real devices, that perform any computation required by the DCOP algorithm. Devices with only a sensing role are usually powered on battery and run as *sleepy nodes*, meaning that they switch off their communication interface most of the time to save energy. Therefore, we consider the actuator devices to be the agents, and $\mathcal{A} = \mathfrak{A}$. Agent with actuator i is denoted a_i.

3.4.2 Variables and Domains. The variables used in our DCOP are simply the set of variables used in the multi-objective optimization problem (3) representing the SECP. More precisely, the set $\mathcal{X}$ contains all the variables whose values are selected by an agent:

- Actuator variables x_i can clearly be controlled by agents and are part of $\mathcal{X}$
- Scene action variables y_i, which represents predicted states of the environment, are also part of x. These variables do not represent any action or decision in the physical environment, they are *modeling variables*. Yet, agents do try to affect their value, such that it will reduce the distance to the rule's goal (i.e. increase its utility) and $\mathcal{X}(\Phi) \subset \mathcal{X}$.
- On the other hand, the sensor variables s_l are not part of $\mathcal{X}$ as these variables represent sensor values that cannot be controlled by an agent.

This gives us the following definitions:

$$\mathcal{X} = \mathcal{X}(\mathfrak{A}) \cup \mathcal{X}(\Phi) \tag{5}$$

$$\mathcal{D} = \{\mathcal{D}_{x_i} | x_i \in \mathcal{X}(\mathfrak{A})\} \cup \{\mathcal{D}_{y_j} | y_i \in \mathcal{X}(\Phi)\} \tag{6}$$

3.4.3 Constraints. Constraints of the DCOP are obviously based on the constraints of the optimization problem (3). However, this problem has a mix of hard and soft constraints, which cannot be dealt with directly by DCOP algorithms, and DisCSP only support hard constraints. Therefore we need to encode the hard constraints as soft constraints with infinite costs, noted y_j for each $j \in \Phi$:

$$\varphi_j(x_j^1, \ldots, x_j^{\overline{\phi_j}}, y_j) = \begin{cases} 0 & \text{if } \phi_j(x_j^1, \ldots, x_j^{\overline{\phi_j}}) = y_j \\ -\infty & \text{otherwise} \end{cases} \tag{7}$$

We note the set of translated hard constraints $\mathcal{F}(\Phi)$ and the set of constraints of the DCOP can be defined as:

$$\mathcal{F} = \mathcal{F}(\mathfrak{A}) \cup \mathcal{F}(\mathfrak{R}) \cup \mathcal{F}(\Phi) \tag{8}$$

3.4.4 Objective. Using these definitions, SECP is then formulated as a DCOP with the following objective:

$$\underset{\substack{x_i \in \mathcal{X}(\mathfrak{A}) \\ y_j \in \mathcal{X}(\Phi)}}{\textbf{maximize}} \quad \omega_u \sum_{k \in \mathfrak{R}} r_k - \omega_c \sum_{i \in \mathfrak{A}} e_i + \sum_{j \in \Phi} \varphi_j \tag{9}$$

Example 4. Figure 2 represents a factor graph for the DCOP of the SHE from Example 1, with 3 actuators (2 light bulbs and a roller shutter), one physical model and environment state (the light level in the living room), two sensors (presence and luminosity) and one rule.

Example 5. Figure 3 depicts the factor graph for the SECP of a real house level. Note that, for clarity, rules have been omitted in this figure.

4 DISTRIBUTION OF THE DECISIONS AND COMPUTATIONS OVER A PHYSICAL MULTI-AGENT INFRASTRUCTURE

In this section, we present several techniques to distribute decision variables and computations over a set of agents. While very few works currently exist in this domain, we consider it to be an important part of using DCOP approaches in physical distributed environments. When implementing a DCOP in a distributed system, agents are embodied in physical objects (computers, connected objects, etc.) and the actions/decisions of these agents are modeled as variables.

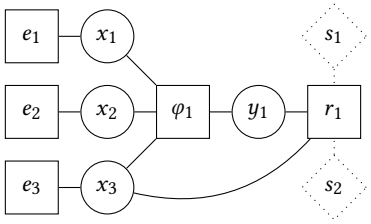


Fig. 2. Factor graph for the scenario of Example 1

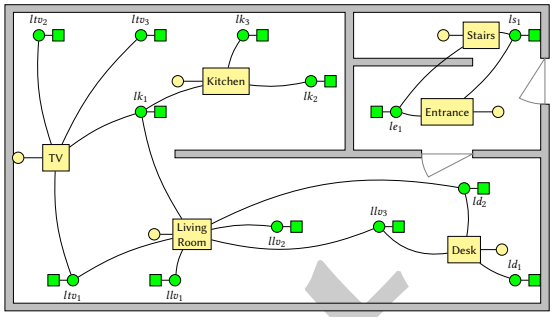


Fig. 3. Factor graph for a realistic full house level

In general, a variable that models the decision of an agent *naturally* belongs to the agent that is making, and potentially applying to the environment, that decision. However it is common, when modeling a problem with a DCOP, to define variables that can either model *shared decisions* or represent some abstract concepts needed for decision. One can think of such variables as *modeling artifacts* or *auxiliary variables*. In such cases, the definition of the α mapping is not straightforward.

4.1 Distributing Computations

As DCOP algorithms are distributed message passing algorithms, they are defined as a set of *building blocks*, where each block's behavior consists in sending and receiving messages. We call these building blocks *computations*, noted c_i:

Definition 3 (Computation). *In a DCOP algorithm, a computation c_i is a piece of code that runs on an agent and only interacts with other computations through message sending.*

We note C the set of computations required to solve a DCOP with a given DCOP algorithm. These computations communicate with each other through message sending and a computation can only send messages to computations that it knows and depends on, forming a graph whose edges can the defined as a set E_C. We note $\mathbf{N}(c_i) = \{c_j | (i, j) \in E_C\}$ the set of neighbors of c_i in this graph.

Definition 4 (Computation Graph). *A computation graph G_C is a tuple $\langle C, E_C \rangle$ where C is a set of computations and E_C is a set of edges (i, j) representing the dependencies between computations.*

Of course, the exact set of computations, and their dependencies, needed for a given system depends on the DCOP used to model the problem (number of variables, constraints, etc.) and the DCOP algorithm used to solve that problem. Some $c \in C$ are associated with a variable, whose value they are responsible for, while some other may serve other purposes. A DCOP algorithm defines what a computation does: what type of messages it reacts to, the structure and the content of the messages it sends, the conditions on which it selects a value for a variable (if applicable), etc. We can see a DCOP algorithm as a set of *computation definitions* and it's only when an algorithm is applied to a problem modeled as a DCOP that we have a set of computations, as depicted on Figure 4. As a matter of fact, the same DCOP can be instantiated as different sets of computations if solved with different algorithms.

Once these computations are defined, based on the DCOP and the algorithm, one must assign them to agents that will operate them. We call this assignment a *distribution*.

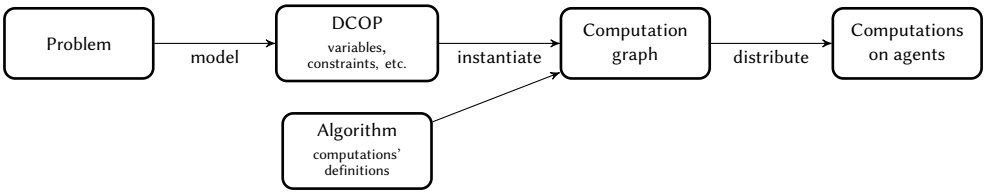


Fig. 4. Distribution of computations for an instantiated DCOP

Definition 5 (Distribution). *Given a set of agents $\mathcal{A}$ and a computation graph $G_C = \langle C, E_C \rangle$, a distribution is a mapping function $v : C \mapsto \mathcal{A}$ that assigns each computation to exactly one agent.*

We note $a_a = v(c_i)$ the agent hosting the computation c_i and $v^{-1}(a_a)$ the set of computations hosted on agent a_a. Notice that this distribution v is not necessarily the same function as the α mapping function; while α maps *variables* to agents, v maps *computations* to agents.

As we can see, the distribution is a more general concept than the mapping of variables to agents and better takes into account distributed implementation constraints. However, when such a mapping is given –even partially– by the problem we want to solve, it must obviously be respected. When defining a distribution, we should only distribute computations that are representing an element (variable or factor) that is not already mapped to an agent by μ.

Formally, we have seen that on some real life problems, α is often not defined for all $x \in \mathcal{X}$. We denote $\mathcal{X}_p$ the subset of $\mathcal{X}$ for which we have a mapping:

$$\mu : \mathcal{X}_p \mapsto \mathcal{A}, \quad \mathcal{X}_p \subseteq \mathcal{X} \tag{10}$$

As a computation represents one or several variables or constraints, the set of computations can be defined as follows, where $\mathbb{P}^+$ denotes the power set excluding the empty set:

$$C \subset \mathbb{P}^+(\mathcal{X}) \cup \mathbb{P}^+(\mathcal{F}) \tag{11}$$

And in order to honor the mapping given by μ, $v : C \mapsto \mathcal{A}$ must respect the following:

$$\forall c \in C, v(c) = \begin{cases} \alpha(c) & \text{if } c \in \mathcal{X}_p \\ a,\ a \in \mathcal{A} & \text{otherwise} \end{cases} \tag{12}$$

4.2 Computation Graph Distribution Problem

The model we introduce here targets generic computation graphs (and thus fits SECP), contrary to previous works which simply focused on factor graphs [30]. This model integrates several features that we consider to be necessary when deploying computations and services over IoT systems.

Let $G_C = \langle C, E_C \rangle$ be a computations graph, and let $\mathcal{A}$ be the set of agents which can host the computations $c_i \in C$. In IoT settings, communications are heterogeneous both in performance characteristics and costs; they can range from high speed fiber connection to cloud servers to low-power, short-range, slow wireless connection between constrained devices in the Local Area Network. As a consequence, for a distribution to be communication-efficient it should both generate as little communication 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 cost matrix: **route**: $\mathcal{A} \times \mathcal{A} \mapsto \mathbb{R}^+$ where **route**(a_m, a_n) is the communication cost between agents a_m and a_n. We note **msg**(c_i, c_j)

the size of the messages between the computations c_i and c_j. Using these functions, we can define the communication cost between the computation c_i hosted on agent a_m and c_j hosted on a_n as follows:

$$\forall c_i, c_j \in C, \quad \forall a_m, a_n \in \mathcal{A}, \quad \mathbf{com_a}(c_i, c_j, a_m, a_n) = \mathbf{msg}(c_i, c_j) \cdot \mathbf{route}(a_m, a_n) \tag{13}$$

This model for communication cost is quite flexible. We can, as previously, decide that there is no communication cost when the two computations are hosted on the same agent by simply setting:

$$\forall a_m \in \mathcal{A}, \ \mathbf{route}(a_m, a_m) = 0$$

We could also assign a (typically small) non-null cost for intra-agent communication and even specify different inter-agent communication costs depending on the type of agents. This approach also allows modeling systems where some agents cannot communicate with some other agents, by simply assigning infinite costs in this cost matrix.

When hosting a computation in an IoT environment, it is often desirable to favor some agents for other reasons than communication only. For example some computation might be very CPU intensive and we want to ensure it will be hosted on a server with a powerful CPU. When using cloud resources, there might also be different cost for hosting a computation on a given server, compared to another. Additionally some parts of the infrastructure might be less prone to disconnection and some computations may be less tolerant to sporadic connection. Finally, especially in IoT, some computation might be tightly linked to one specific physical element, as it is for example the case in our SECP model, and can only be reasonably hosted on that element. We model this affinity, or repulsiveness, between an agent and a computation with a function $\mathbf{cost}\colon \mathcal{A} \times C \mapsto \mathbb{R}^+$ that assign a cost for each pair (a_m, c_j). Notice that one can easily force a computation $c_\mathbf{j}$ to be hosted on a specific agent $a_\mathbf{n}$ by assigning an infinite hosting cost for all other agents:

$$\forall c_i \in C, \forall a_m \in \mathcal{A}, \quad \mathbf{cost}(c_i, a_m) = \begin{cases} 0 & \text{if } i = \mathbf{j} \text{ and } m = \mathbf{n} \\ \infty & \text{otherwise} \end{cases}$$

We also consider that an agent can only host a limited number of computations and model this with agents' capacity and computations' footprint noted respectively $\mathbf{w_{max}}(a_m)$ and $\mathbf{mem}(c_i)$

Using these definitions, we define an IoT optimal distribution of a computation graph as follows:

Definition 6 (IoT optimal distribution). *An **IoT optimal distribution** is a distribution v that minimizes the cost of communication between agents and minimize the cost of hosting computations while respecting the agents' capacity constraints.*

Definition 7 (CGDP). *Given a computation graph and a set of agents, the Computation Graph Distribution Problem (CGDP) amounts to assign each computation of the computation graph to an agent to obtain an IoT optimal distribution.*

4.3 Linear Program for Optimal Distribution

We can now encode our IoT optimal distribution problem as Integer Linear Program (ILP). Let c_i^m be a binary variable denoting whether the computation c_i is hosted on agent a_m. The binary variable α_{ij}^{mn} denotes if both computation c_i is hosted on agent a_m and c_j is hosted on a_n.

$$\forall c_i \in C, \quad c_i^m = \begin{cases} 1, & \text{if } v(c_i) = a_m \\ 0, & \text{otherwise} \end{cases} \quad \forall\ c_i, c_j \in\ C, \qquad \alpha_{ij}^{mn} = c_i^m \cdot c_j^n$$

Our communication efficiency objective amounts to minimize the communication cost for all edges of the computations graph. Additionally, we also aim at reducing the hosting cost. By aggregating these two objectives with penalizing

factors $\omega_{\mathbf{com}}$ and $\omega_{\mathbf{c_{host}}}$, we can define our distribution problem as a mono-objective optimization problem, modeled as follows:

$$\underset{c_i^m}{\textbf{minimize}} \quad \omega_{\mathbf{com}} \cdot \sum_{(i,j)\in E_C} \sum_{(m,n)\in \mathcal{A}^2} \mathbf{com_a}(c_i, c_j, a_m, a_n) \cdot \alpha_{ij}^{mn} + \omega_{\mathbf{c_{host}}} \cdot \sum_{(c_i,a_m)\in C\times\mathcal{A}} c_i^m \cdot \mathbf{cost}(a_m, c_i) \tag{14}$$

subject to

$$\forall a_m \in \mathcal{A}, \quad \sum_{c_i \in C} \mathbf{mem}(c_i) \cdot c_i^m \leq \mathbf{w_{max}}(a_m) \tag{15}$$

$$\forall c_i \in C, \quad \sum_{a_m \in \mathcal{A}} c_i^m = 1 \tag{16}$$

$$\forall c_i \in C, \quad \alpha_{ij}^{mn} \leq c_i^m \tag{17}$$

$$\forall c_j \in C, \quad \alpha_{ij}^{mn} \leq c_j^m \tag{18}$$

$$\forall c_i, c_j \in C, a_m \in \mathcal{A}, \quad \alpha_{ij}^{mn} \geq c_i^m + c_j^n - 1 \tag{19}$$

This ILP is flexible enough to accommodate a large panel of IoT scenarios; by using appropriate communication and hosting cost matrices one can for example easily use it to reproduce the ILPs designed specifically for a DCOP representing a SECP, both for constraint graph or a factor graph based algorithms.

Definition 8 (ILP-CGDP). *We term ILP-CGDP the 0/1 integer linear program consisting of objective (14) and constraints (15) to (19) which encodes the CGDP problem from Definition 7.*

This program gives us a definition of an optimal distribution, and can be solved by classical centralized solvers (GLPK in our experiments). However, the complexity is still very hard and it is only possible for relatively small systems. AmI systems, and more generally IoT systems can be very large, meaning that this approach would most probably not scale. Even though, the objective function can still be used to evaluate the quality of approximate distribution method.

4.4 Greedy Heuristic for Computation Graph Distribution

As ILP-CGDP might be too difficult to solve for large system, we also developed a heuristic that computes an approximate distribution. This heuristic is designed for generic computation graph and thus works for both constraint graph and factor graph based algorithms. We start by placing the computation with the highest footprint, and select the agent with enough remaining capacity that incurs the lowest aggregate communication and hosting costs. In case of ties, we chose the agent with the highest remaining capacity. Of course, this greedy heuristic is sub-optimal, but allows computing a distribution easily even for large systems. Notice however that if the system is severely constrained capacity-wise, it may fail to find a distribution even though one exists and would be found by ILP-CGDP (provided the ILP can be solved in a reasonable time).

Definition 9 (GH-CGDP). *We term GH-CGDP the method for distributing a computation graph using that greedy heuristic.*

5 INSTALLING RESILIENCE IN DYNAMIC SMART ENVIRONMENTS

When deploying such distributed systems as SECP in open environments like the one envisioned by AmI and IoT, it is difficult to consider that the problem, including the devices that run it, will never change during its active lifetime. SECP is designed to model an AmI system where devices cooperate autonomously, without any centralized decision

point, to reach user-defined objectives in a home environment. All elements of SECP might undergo changes during the nominal execution of the system. For instance, devices may be added or removed, or even their properties may change; users may add, remove or edit scenes; the configuration of a place can be changed, thus modifying its physical model; finally, the environment, providing sensed data will change by providing new temperature, humidity or presence values. Most of these changes are only parameter-level modifications, but some of them may impact the distribution itself, and thus requires system reconfiguration. Thus, we will now focus on situations where the infrastructure changes (e.g. objects disappear), and want to install some sort of resilience against these unpredictable events. This requires notably (i) preserving the problem definition (i.e. saving the set of computations defining the SECP) and (ii) maintaining the solving process state (i.e. enabling removing agents while the system is currently solving the SECP). We next meet these requirements through the notion of k-resilience, using a replica-placement model, namely Distributed Replica Placement Method (DRPM), and a dedicated repair model, namely DCOP Model for Computation Migration (DMCM). These models contributes to a solution method for installing resilience, we coined DRPM[DMCM].

5.1 k-Resilience

We define 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 computation graph G_C.

Definition 10. *Given a set of agents $\mathcal{A}$, a set of computations C, and a distribution μ, the system is k-**resilient** if for any $F \subset \mathcal{A}, |F| \leq k$, a new distribution $\mu' : X \rightarrow \mathcal{A} \backslash F$ 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 [20]. 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 (*almost stateless*).

We note $\rho(c_i)$ the set of agents that possess a replica for computation c_i. In a k-resilient system, each computation has k replicas, which are placed on agents that do not host the active version of the computation: $\forall c_i \in C, |\rho(c_i)| = k$ and $v(c_i) \notin \rho(c_i)$. On most systems, the placement of these replicas will be constrained by various other factors such as agents' capacities (memory and CPU), communication costs and problem-domain specific affinities or anti-affinities between some computations and some agents.

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 agents' capacities. Additionally, after departure of some agent(s), the k-resilience characteristic of the repaired system should be restored, as long as there are enough nodes available.

The problem of assigning replicas to hosts could be considered as an optimization problem, close to Definition 6. Ideally, we should optimize replica placement for communication and hosting costs. This would ensure that when agents fail, replicas are available on good candidate agents. However, the search space for this optimization is prohibitively large: (i) In a k-resilient system with n agents, there is $\sum_{0<i\leq k} \binom{n}{i}$ potential failure scenarios as we consider case where up to k agents out of n can fail simultaneously; (ii) With m computations, the number of possible *replica configurations* is $m \cdot \binom{n}{k}$, as we must select k agents to host the replicas for each of the m computations; (iii) Then, for each of these

replica configurations, there are m^k *activation configurations*, as exactly one of the k replicas must be activated for each orphaned computation. 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) (see [33]), which is NP-hard. Assuming we could compute the cost of all these activation configurations, it would still not be obvious which replica placement would be better: one could consider the one allowing the best activation-configuration, or the one allowing, on average, good quality activation configurations or even the one giving the best activation configurations over the set of possible failure scenarios. Obviously, defining the optimality for replica placement is very problem dependent. Thus, given that complexity, we opt for a distributed heuristic approach, described in the next section.

5.2 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 [28, p.90] (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 capacity constraints, DRPM places as much computations as possible and outputs the best resilience level it could achieve.

One hosting agent, called *initiator*, *iteratively* asks each of its 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.

This approach is based on the assumption that the initial distribution, computed using one of the methods introduced in Section 4, is optimal or at least of good quality. As a matter of fact, if the initiator agents fails, its orphaned computation will necessarily be migrated to one of the agents that possess its definition (i.e. that hosts one of its replicas). Therefore, by placing replicas on agents that have minimal communication and hosting cost compared to the initiator agent, we ensure that the computation will only be migrated to agents that favor a good quality distribution.

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

Definition 11. *Given a computation graph $\langle C, E_C\rangle$ and a set of agents $\mathcal{A}$, the* **route**–*graph is the edge-weighted graph $\langle \mathcal{A}, E, w\rangle$ where $\mathcal{A}$ is the set of vertices; $E = \{(a_m, a_n) |\ \exists (c_i, c_j) \in E_C,$ and $\nu(c_i) = a_m, \nu(c_j) = a_n\}$ is the set of edges; $w : E_C \rightarrow \mathbb{R}$ 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 a weighted edge representing the hosting cost of the agent, as in Figure 5.

Definition 12. *Given a* **route**–*graph $\langle \mathcal{A}, E, w\rangle$ and a computation c_i, the* **route+host**–*graph is the edge-weighted graph $\langle \mathcal{A}', E', \mathbf{cost}\rangle$ where $\mathcal{A}' = \mathcal{A} \cup \widetilde{\mathcal{A}}$ is the set of vertices where $\widetilde{\mathcal{A}} = \{\tilde{a}_m | a_m \in \mathcal{A}, a_m \neq \nu(c_i)\}$ is a set of extra vertices (one for each element in $\mathcal{A}$ except the host of c_i); $E' = E \cup \{(a_m, \tilde{a}_m) | \tilde{a}_m \in \mathcal{A}\}$ is the set of edges;* $\mathbf{cost} : E' \rightarrow \mathbb{R}$ *is the weight function s.t.* $\forall a_m, a_n \in \mathcal{A},\ \mathbf{cost}(a_m, a_n) = w(a_m, a_n), \forall \tilde{a}_m \in \widetilde{\mathcal{A}},\ \mathbf{cost}(a_m, \tilde{a}_m) = \mathbf{c_{host}}(a_m, c_i)$.

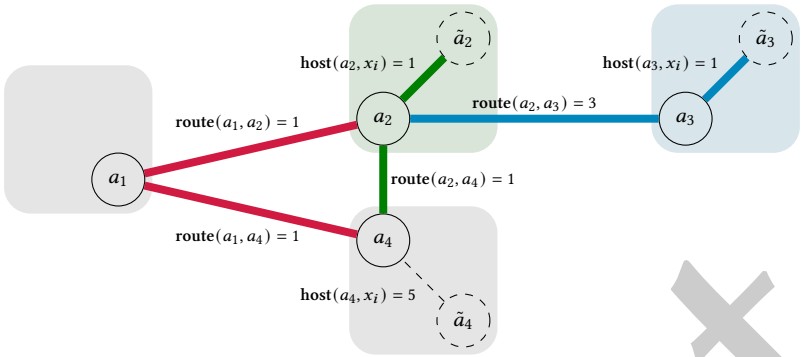


Fig. 5. A sample **route+host**–graph with 4 agents (in gray) and a sample execution of DRPM for placing two replicas for a computation x_i

Example 6. Figure 5 shows a **route+host**–graph for a computation graph distributed over 4 agents. Notice that agent a_1 has not extra vertex $\tilde{a}_1$, representing its hosting cost, as the **route+host**–graph on this figure is designed to place the replicas of the computations c_i hosted on a_1 (which must obviously be placed on other agents). As a matter of fact, when placing replicas, the **route+host**–graph is specific to an initiator agent and a computation. A different **route+host**–graph is expanded by each agent a_k and for each of the computation c_i hosted on a_k, to place the replica for computation c_i.

A **route+host**–graph is a search graph, expanded at runtime and explored for a particular computation c_i. Each agent operates as many instances of DRPM as computations to replicate over several **route+host**–graphs. For a given **route+host**–graph , each agent may encapsulate two vertices (one in $\mathcal{A}$ and its image in $\widetilde{\mathcal{A}}$) and may receive messages concerning their two vertices, and even self-send messages. Additionally, when assessing if an agent can host a replica for c_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 might reject hosting a computation whilst it may finally have enough memory to host it. Even communication-wise, the algorithm may result in 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 [28, p.90].

DRPM makes use of two message types, `REQUEST` and `ANSWER`, with the same fields :

- `current`: path of the request, as a list containing all vertices messages that have been passed through from the initiator vertices to the one receiving the current message.
- `budget`, `spent`: remaining budget for graph exploration and budget already spent on the current path,
- `known`: map assigning cost to already discovered paths to unvisited vertices which bookkeeps the cheapest paths so far,
- `visited`: list of already visited vertices,
- `k`: the remaining number of replicas to host,
- c_i: computation that must be replicated.

Algorithm 1: Handler for REQUEST

```
   Data: current, budget, spent, known, visited, k, c_i
1  known ← known \ current
2  if me ∉ visited then
3      visited ← visited ∪{me}
4      if can_host?(computation_i) then
5          k ← k − 1
6          add x_i to memory
7          if k = 0 then
8              a_p ← predecessor of me in current
9              send ANSWER(current,
                 budget+cost(me, a_p), spent-cost(me, a_p),
                 known, visited, k, c_i) to a_p
10             return
11 p ← argmin_{e∈{paths in known starting with current}} known[e]
12 if p ≠ ∅ then
13     a_n ← successor of me in p
14     if cost(me, a_n) ≤ budget then
15         current ← current + a_n
16         send REQUEST(current, budget-cost(me, a_n),
             spent+cost(me, a_n), known, visited, k, c_i)
             to a_n
17         return
18 foreach a_n ∈ {a_m | (a_m, me) ∈ E', a_m ∉ visited} do
19     if spent + cost(me, a_n) <
         min_{e∈{paths in known leading to a_n}} known[e] then
20         known[current + a_n] ← spent + cost(me, a_n)
21 a_p ← predecessor of me in current
22 send ANSWER(current, budget+cost(me, a_p),
     spent-cost(me, a_p), known, visited, k, c_i) to a_p
```

Algorithm 2: Handler for ANSWER

```
   Data: current, budget, spent, known, visited, k, c_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, c_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), spent+cost(me, a_n),
                 known, visited, k, c_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, c_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, c_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 fills 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 reaches the initiator (line 6). In the latter case, if the agent is the initiator it increases the budget and sends a request to the closest neighbor (line 14) if any; if there is no such neighbor left, 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).

Example 7. Figure 5 represents the execution of DRPM when agent a_1 places 2 replicas for computation c_i. The different colors in the edges depicts the path explored when increasing the budget.

At starts, a_1 initializes the known map with the paths to a_2 and a_4: known = $\{a_1 \rightarrow a_2 : 1, a_1 \rightarrow a_4 : 1\}$.

The first exploration of the graph (depicted in red) starts:

- a_1 starts by sending itself a REQUEST message with a budget of 1.
- When handling this message, a_1 forwards the request to a_2, with a budget of 0.
- Then a_2 fills out known and sends back an ANSWER to a_1, as the budget does not allow forwarding the request further.
- As it did with a_2, a_1 now sends a REQUEST to a_4, which fills out known and sends ANSWER back.
- At this point, a_1 as no other neighbor to forward the REQUEST and must increase the budget to 2, the cheapest path in known.

The same process (in green) is repeated with a budget of 2:

- This updated budget allows expanding the path up to $\tilde{a}_2$, where a first replica is placed.
- A new path to a_4 is discovered but not kept in known, as it already contains a cheaper path to that node.
- Once all paths that can be reached with this budget have been explored, a_1 increases the budget again.

Once again, a_1 explores the graph by REQUESTS messages, this time with a budget of 5 (in blue).

- This budget allows reaching $\tilde{a}_3$, and thus hosting the second replica.
- ANSWER messages are then sent back up to a_1, with k=0 (all required replicas have been placed) and DRPM terminates.

At the end of the process, replicas of c_i have been placed on a_2 and a_3, with path costs of respectively 2 and 5. No replica has been placed on a_4, as it would incur a higher path cost of 6.

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

Let's now show DRPM termination and communication load properties.

Theorem 8. *DRPM terminates.*

Proof. For $k = 1$, since DRPM costs are additive and monotonic, 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. □

Theorem 9. *DRPM requires $O(b^l)$ messages to terminate, with b the branching factor of the search tree, $l = d/e$ number of iterations, d the depth of the search tree, and $0 < e \leq 1$ the normalized step cost.*

Proof. DRPM's worst case is that the only possible hosts for replica are the last explored ones or there is no possible host. The number of explored nodes is $(l)b + (l-1)b^2 + \ldots + (1)b^l$ which is $O(b^l)$. It requires twice messages (request and answer), which is still $O(b^l)$. □

5.3 Migrating Computations

Once computations have been replicated using DRPM, these replicas can be used to repair a running system when an agent fails.

In a distributed systems, one cannot rely on the availability of a centralized decision-making entity to repair the system: such entity might not be available and selecting it from the set of available agents would introduce an extra distributed decision making process (such as leader election) and in the case of constrained devices, a single agent might not be powerful enough to make the decision.

Therefore, we model the repair problem itself as a DCOP, to be implemented by agents 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. Let's first introduce some notations.

We note C_c the set of candidate computations c_i that could or must be moved when the set of agents changes. For each of these computations, we note $\mathcal{A}_c^i$ the set of candidate agents that could host c_i. The set of all candidate agents, regardless of computations, is noted $\mathcal{A}_c = \cup_{c_i \in C_c} A_c^i$. C_c^m denotes the set of candidate computations that agent a_m could host. Deciding which agent $a_m \in \mathcal{A}_c$ hosts each computation $c_i \in C_c$ can be mapped to an optimization problem similar to ILP-CGDP presented in Section 4.3, restricted to $\mathcal{A}_c$ and C_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 (16) for each $c_i \in C_c$:

$$\sum_{a_m \in \mathcal{A}_c^i} c_i^m = 1 \tag{20}$$

Similarly, capacity constraints (15) can be reformulated as:

$$\sum_{c_i \in C_c^m} \mathbf{w}(c_i) \cdot c_i^m + \sum_{c_j \in \nu^{-1}(a_m) \setminus X_c} \mathbf{w}(c_j) \leq \mathbf{w_{max}}(a_m) \tag{21}$$

The hosting cost objective in (14) can be similarly formulated using one soft constraint for each candidate agent a_m:

$$\sum_{c_i \in C_c^m} \mathbf{c_{host}}(a_m, c_i) \cdot c_i^m \tag{22}$$

Finally, the communication costs in (14) are represented with a set of soft constraints. For an agent a_m, the communication cost incurred by hosting a computation c_i can be formulated as the sum of the cost of the cut edges (c_i, c_j) from the computation graph $\langle C, D \rangle$, (i.e. where $v^{-1}(c_j) \neq a_m$). Let's note N_i the neighbors of c_i in the computation graph. When a neighbor c_n is not a candidate computation (i.e. it might not be moved and $c_m \in N_i \backslash C_c$), the communication cost of the corresponding edge is simply given by $\mathbf{com_a}(c_i, c_j, a_m, v^{-1}(c_n))$. 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 $\sum_{a_n \in \mathcal{A}_c^j} c_j^n \cdot \mathbf{com}(i, j, m, n)$. With this we can write the communication cost soft constraint for agent a_m:

$$\sum_{(c_i, c_j) \in C_c^m \times N_i \backslash C_c} c_i^m \cdot \mathbf{com_a}(c_i, c_j, a_m, v^{-1}(c_j)) + \sum_{(c_i, c_j) \in C_c^m \times N_i \cap C_c} c_i^m \cdot \sum_{a_n \in \mathcal{A}_c^j} c_j^n \cdot \mathbf{com_a}(c_i, c_j, a_m, a_n) \tag{23}$$

We can now formulate the repair problem as a DCOP $\langle \mathcal{A}, \mathcal{X}, \mathcal{D}, C, \mu \rangle$ where $\mathcal{A}$ is the set of candidate agents A_c; $\mathcal{X}$ and $\mathcal{D}$ are respectively the set of decision variables c_i^m and their domain $\{0, 1\}$; C is composed of constraints (20), (21), (22), and (23) applied for each agent $a_m \in A_c$. (20) and (21) result in infinite costs when violated, while (22) and (23) directly define costs to be minimized; the mapping μ assigns each variable x_i^m to agent a_m.

Definition 13 (DMCM). *Given a set of candidate computations C_c and a set of candidate agents $\mathcal{A}_c$, we term DMCM the DCOP model for selecting a suitable agent for each of the computations.*

Notice that this model for computations migration is not specifically designed for fixing the system after agents failure; it simply implements the decision process for selecting a suitable agent to host some computation(s). As a consequence, it can be used both to implement repair, which we will present in the next section, or to re-distribute computations after some agent(s) arrival (a case that we won't describe in this paper).

5.4 Implementing Repair using DRPM[DMCM]

Using the DCOP-based model for selecting an agent when migrating a computation, the repair phase which supports k-resilience can be implemented. When up to k agents fail, repairing the system amounts to migrate each of the orphaned computations to one of the agents that possess its replica.

Definition 14 (DRPM[DMCM]). *We term DRPM[DMCM] the full solution method for k-resilience composed of DRPM for replication and the DMCM model for computation migration.*

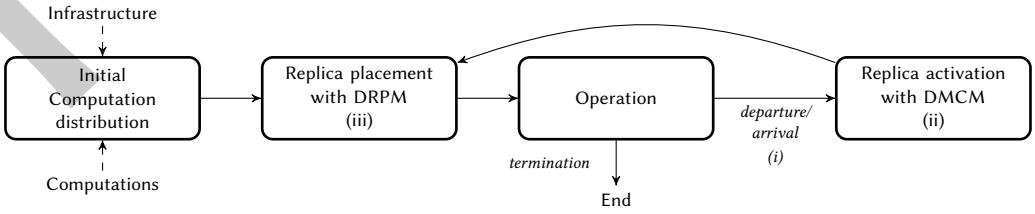


Fig. 6. DRPM[DMCM] life cycle in a glance.

Figure 6 shows the life cycle of this approach. Assuming initial deployment (using one of the methods discussed in Section 4) and replicas placement (using DRPM) have been performed at system bootstrap, the system will execute the following repair cycle all along its lifetime:

(i) *Detect departure/arrival*, which 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, all neighbor agents of a_m in the **route**–graph are aware of the departure.

(ii) *Replica activation* for missing computations, by solving the DCOP for computation migration, as to relocate computations that were hosted on the set of departed agents $\mathcal{A}_d$ to other agents. The candidate computations C_c are the orphaned computations hosted on these agents:

$$C_c = \cup_{a_m \in \mathcal{A}_d} \nu(a_m)$$

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 set of candidate agents $\mathcal{A}_c$ for an orphaned computation c_i maps the set of still available agents hosting a replica for this computation:

$$\mathcal{A}_c^i = \rho(c_i) \backslash \mathcal{A}_d$$

In a k-resilient system, as long as $|\mathcal{A}_d| \leq k$, we are sure that there will always be at least one agent in $\mathcal{A}_c^i$. Thus, step ii yields an assignment of each of the orphaned computations to one of the remaining agents hosting its replica.

(iii) *Replica placement* for missing computations using DRPM, and continue nominal operation, which 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.

5.5 Solving DMCM using a DCOP Algorithm

Now that the repair problem has been expressed as a DCOP, we discuss its resolution using a DCOP solution method. Many solution methods for DCOPs exist, several of which have been presented in Section 2.2. In brief, using these message passing protocols, agents coordinate to assign values to their variables. Each of these solution methods has specific characteristics (they might be complete or not, synchronous or asynchronous, etc.) and makes some assumptions on the environment and the problem (perfect message delivery, hard and/or soft constraints, etc.). Therefore, when solving a DCOP, it is very important to select a DCOP algorithm that matches the characteristics of the problem and its environment. In the case of DMCM, we can identify the following key characteristics to guide our choice of suitable solution methods:

(a) The problem must be solved by constrained devices.

(b) A solution must be found as quickly as possible, as the system will only get back to nominal operation once all orphaned computations have been successfully migrated.

(c) Our model contains hard constraints, to ensure all orphaned computation are migrated and that agent's capacity is honored, and soft constraints, for hosting and communication costs.

(d) A suboptimal solution is acceptable, as long as all hard constraints are satisfied. Indeed, we can reasonably sacrifice some optimality on hosting and communication cost, if orphaned computations are migrated to agents that have enough capacity to host them. Additionally, violating a hard constraint could also mean loosing an orphaned

computation, or activating several replicas for the same computation. In both cases, the system will be in an inconsistent state.

(e) As DMCM is designed to repair a distribution, the solution method used to solve it must not bring about a distribution problem itself, otherwise we would have a *chicken and egg* situation …

Characteristics (a), (b) and (c) compel us to select a lightweight suboptimal algorithm. Local search algorithms for instance, are fast and require very little computation on each agent. Characteristic (e) implies that we must use an algorithm for which the assignment of computations to agents is fully defined for the DMCM problem. As this model contains only binary variables c_i^m, where m maps to agent a_m, we can easily map each variable to an agent. As a consequence, by using constraint graph-based algorithms, which only define computations for variables, we avoid facing a distribution problem when deploying the DCOP used to solve DMCM. Characteristic (c) is more difficult to satisfy. As a matter of fact, few DCOP algorithms have been designed specifically to take into account a mix of hard and soft constraints and many iterative algorithms tend to break hard constraints when optimizing for soft constraints. A monotonic algorithm, like MGM [18] is particularly well suited for this situation; as the cost of the solution monotonically decreases, once the hard constraints (modeled with infinite costs) have been satisfied they will not be broken while optimizing the soft constraints. In our case, decisions require coordination between two agents: to move a computation from agent a_m to agent a_p, the binary variable c_i^m must take 0 as a value, while *simultaneously*, c_i^p must switch from 0 to 1. This need for simultaneous changes justifies the use of MGM-2 (Maximum Gain Message with 2-coordination) [18].

By applying MGM-2 to DRPM[DMCM] we obtain a repair method that we coin DRPM[MGM-2]. Of course, we could use any other DCOP algorithm that matches these characteristics.

For example algorithms such as BrC-DPOP [6] and CeC-DPOP [27], which should handle well our mix of soft and hard constraints, could also be good candidates for solving DMCM.

6 EXPERIMENTAL EVALUATION

Now, we experimentally evaluate the distribution methods, the replica placement method and the repair method we contributed in the previous section, on randomly generated SECP instances.

6.1 Experimental Setup

We consider a realistic smart home with actuators (light bulbs), physical models and user-defined rules. Notice that, for simplicity sake, we only consider light control in our experiments, even though our model could be applied to other parameters in a house. As presented previously, each actuator is represented by a variable x_i associated with an efficiency factor e_i, which defines a cost function as a linear function of the emitted luminosity. Each physical dependency model is represented by a pair (φ_j, y_j), where φ_j is defined as weighted sums (weights are randomly selected) of the luminosity levels emitted by the light bulbs in its scope and yield the theoretical resulting luminosity in a given place as an indirect scene action variable y_j. Finally, each rules r_k assigns target values to one or several scene action variables (actuators and models). Variables, models and rules are randomly connected and we only consider active rules, which have an actual influence on the problem. We use $\omega_c = 1$ and $\omega_u = 10$ as weights when aggregating the two objectives (respectively for energy cost and rules utility). We generate larger instances by increasing the number of lights, physical models and rules in the system, which represents progressively larger houses. Each physical model is randomly connected to 1 to 4 lights and each rule is randomly connected to 1 to 3 models or lights. The

smallest instances have 10 lights, 3 models and 2 rules, and the count of each of these elements is linearly increased, by increments of 10 lights, up to 90 lights, 27 models and 18 rules. Communication costs are uniform between agents (1) and hosting costs are identical for all computations (100), except light computations (0) to be hosted by smart light bulbs themselves. Agents' capacities are set to 1000 for instances smaller than 60, and to 1500 for larger instances.

We use the python library pyDCOP for generating the problem instances, the computation graph and the agents, and for computing the distributions [31]. For solving the linear program ILP-CGDP is based on, pyDCOP relies on the GLPK[1] solver.

All the generators and algorithms are provided in the pyDCOP[2] library [31].

6.2 Evaluating SECP Resolution in Static Settings

First, we solve 100 static SECP instances for each problem size, with a 120-seconds timeout, using A-DSA (variant B, $p = 0.7$) [9], MGM and MGM-2 [18], a customized Damped A-MaxSum, and DPOP [23], a complete inference algorithm as a benchmark. Our customized variant of A-MaxSum (based on Damped MaxSum, with 0.2 damping factor on variables and factors [2]) is designed to improve its behavior after a repair operation: once orphaned computations have been migrated and restarted on an active agent, the accumulated cost table of their neighbors is flushed. Additionally, the standard mechanism used to avoid sending duplicate messages (classically used to detect termination when messages converge [5]) is inhibited and belief propagation is restarted. Notice that this can be implemented in a distributed manner with a simple token passing approach.

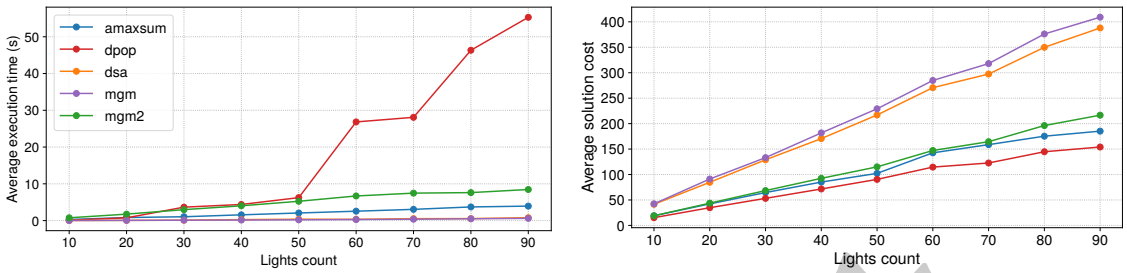
Figure 7b shows the cost of the solutions. DPOP always produces the lowest cost. MGM-2 and A-MaxSum provide very good quality results, while DSA and MGM display lower, similar quality. However, DPOP actually fails to solve numerous instances within the allocated time budget (up to 43% for problems with 90 light bulbs); for these cases we have used the time budget when computing the average execution time, in Figure 7a. DPOP is obviously much slower than the approximate algorithms and does not scale with increasing problem size. Overall, both A-DSA and A-MaxSum are good candidates for solving SECP, but at different expenses: speed for A-MaxSum and solution quality for A-DSA. Additionally, they both exhibit very interesting characteristics for our purpose: they are asynchronous, robuts to message loss and *almost-stateless*. Indeed, an A-DSA computation actually requires a single message round to rebuild its state, as it only keeps track of the values of its neighbors. A-MaxSum requires a few more rounds, but our customization makes this process faster. These features motivate our choice of using A-DSA and A-MaxSum in the next experiments.

6.3 Evaluating Generalized Distribution on SECP

In this experiment, we use our optimal and heuristic distribution methods on DCOPs representing SECP instances. 20 instances are generated for each problem size and distributed using GH-CGDP and ILP-CGDP (with a 30 seconds time limit to represent an acceptable installation time for an user), when using a constraint graph representation and a factor graph representation.

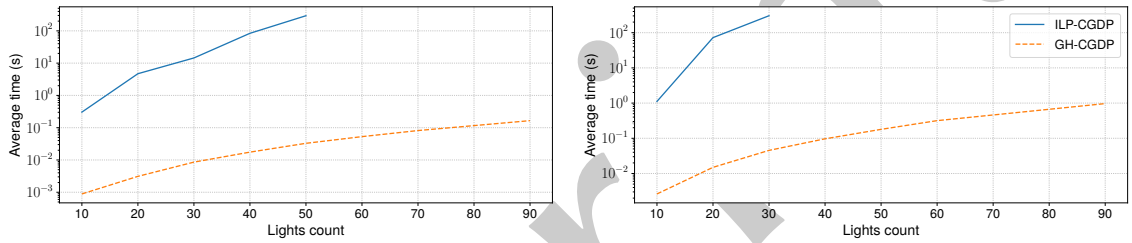
Figure 8 shows the time required to compute the distribution when using constraint graph (8a) and factor graph (8b) representations of the DCOP. We only plot optimal distribution times for problem sizes for which *all instances* could be distributed. Distribution is harder to compute when using a computation graph based on a factor graph representation, since more computations are required to operate the very same problem. The optimal distribution is restricted to

[1]https://www.gnu.org/software/glpk/
[2]https://github.com/Orange-OpenSource/pyDcop



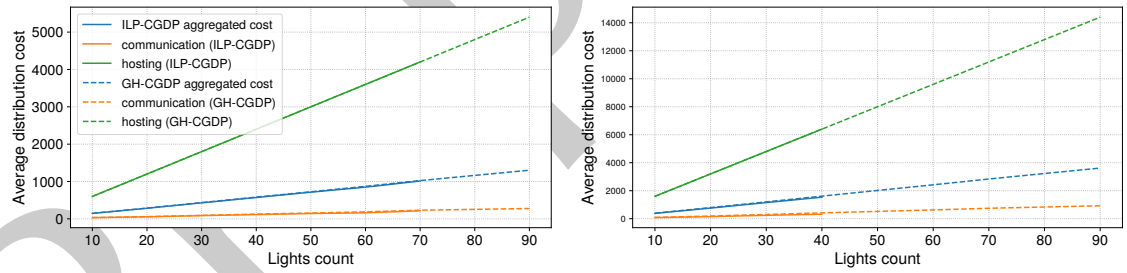
(a) Duration (b) Cost

Fig. 7. Execution time and solution cost when solving static SECP instances with DCOP algorithms



(a) Constraint graph representation (b) Factor graph representation

Fig. 8. Time for distributing SECP instances with optimal and heuristic methods

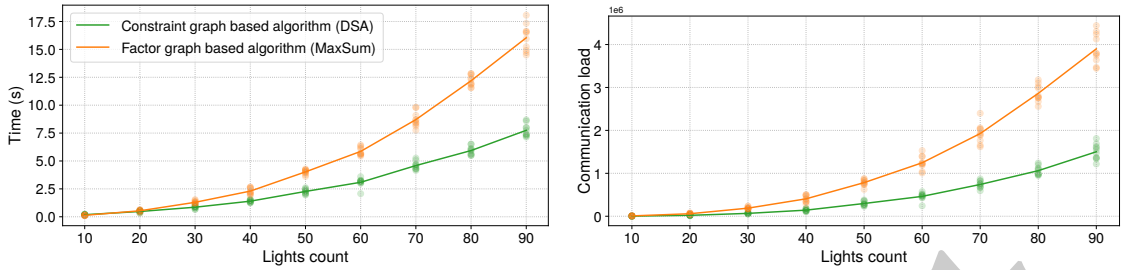


(a) Constraint graph representation (b) Factor graph representation

Fig. 9. Cost of distributions for SECP instances

relatively small instances while the heuristic distribution can be used on very large instances: it can easily deal with SECP with 90 lights and could distribute much larger instances.

Figure 9a and 9b depict the distribution costs for our SECP instances when using the heuristic and optimal methods, respectively with a constraint graph and factor graph representation. As the distribution cost is made of communication and hosting costs (see Definition 8), we also plot the communication and hosting components. Notice that we include in this figure the optimal cost of distribution for all problems size for which at least some instances could be distributed.



(a) Computation time

(b) Communication load (in bytes)

Fig. 10. Time and communication load to replicate computations for SECP instances

As previously, we can see that GH-CGDP produces very good quality distributions while requiring several orders of magnitude less time.

6.4 Evaluation of DRPM

We evaluate now DRPM, the replica placement method. We derive two computation graphs for each problem instance, one for DSA and one for A-MaxSum, which are respectively a constraint graph and a factor graph based algorithm. The multi-agent infrastructure used for these SECP problems is made of one agent for each light, and computations representing a light is assigned to the corresponding agent. The initial distribution is computed using GH-CGDP.

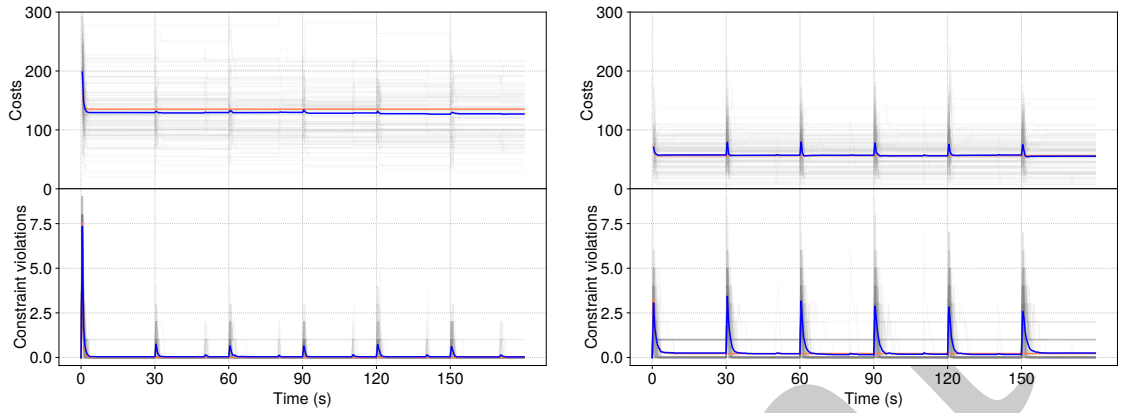
Figure 10a represents the time required to achieve 3-resilience on our SECP instances. As expected, replication is harder for A-MaxSum, as it requires more computations for the same problem. In any case, we argue that these replication times are perfectly reasonable for real-like system as this operation only needs to be run once when starting the system: during the nominal execution of the system, full replication is never needed and we can simply repair an existing replication if some agent fails or leave the system. Figure 10b shows the total communication loads induced by DRPM, which evolves in same way than computation time.

6.5 Evaluation of the DMCM Repair Method

In these experiments, we evaluate DMCM, our DCOP-based repair method, on running SECP. As DMCM models the repair process as a DCOP, we implement it using two different DCOP algorithms and compare their efficiency on this task. The algorithm used for repairing the SECP distribution is MGM-2 ($q = 0.5$).

One scenario of 5 events is generated for each instance; at each event 2 random agents (i.e. light device) are removed. The initial distribution of the computation graphs derived from these problems is computed with GH-CGDP.

The 100 SECP instances are solved with our customized A-MaxSum variant and A-DSA, as motivated in Section 6.2. During the solving process, we inject the scenario's events in the system every 30 seconds, each time removing 2 agents, and repair the system using DRPM[MGM-2]. After each repair we re-run DRPM, for migrated computations and computations whose replica were hosted on removed agents. This ensures that each computation in the system still has k replica. Our mechanism amounts to using a DCOP (solved with MGM-2) to repair and restore nominal operation on another DCOP (solved with A-DSA or A-MaxSum), which represents the initial dynamic problem we want to solve (in this case, SECP). Additionally, we also solve the same problems without any disturbance, in order to assess the impact of our repair method on the quality of the solution returned by A-DSA and A-MaxSum.



(a) A-DSA (b) A-MaxSum

Fig. 11. Cost and hard constraints violations of operating A-DSA and A-MaxSum to solve SECP, repaired with DRPM[MGM-2] (blue: with perturbations, red: without perturbation)

Figures 11a and 11b show the cost of the solutions found by A-DSA and A-MaxSum over time. As the SECP model contains both soft and hard constraints, we plot separately the number of violated hard constraints (bottom) and the sum of costs of the soft constraints (top). The results for each of the 100 instances are displayed in transparent grey and the average cost across all instances is plotted in blue. The average cost of the same instances solved without disturbance is plotted in red, but is barely visible as the average repaired cost is extremely similar. We can see that both A-DSA and A-MaxSum behave remarkably after a repair: during a short period after the repair the solution cost and the number of violated hard constraints increase, but quickly get back to the quality level achieved before the agents were removed. Overall, we can say that DRPM[MGM-2] is well suited for repairing running SECP and that the quality of the solution produced over time is barely affected by the repair operations. However, when comparing results produced by A-DSA and A-MaxSum, we can observe that A-DSA yields higher costs. After a repair, A-MaxSum generally breaks more hard constraints than A-DSA but we argue that it is not really problematic as it always manage to get back to the level it had before the disturbance. Notice however that the average number of hard constraints violations is not equal to zero. Indeed, there are some instances where A-MaxSum struggles with hard constraints.

6.6 Physical Demonstrator

pyDCOP and the techniques we developed in this paper have also been used to build a physical demonstrator. This demonstration illustrates the k-resilient distributed decision making process in an IoT system. Our scenario is based on a classical distributed weighted graph coloring problem (to be more illustrative), to which many real problems can be mapped. Each variable in the system maps to a vertex in the graph and can take one color as a value. Edges of the graph map to binary constraints, assigning a cost for each combination of colors taken by its associated variables/vertices. The goal is to find a assignment of colors that minimize the sum of these costs. Several graph structures can be used when generating instances of this problem. To model an IoT infrastructure, we use the Barabasi-Albert method [1], which produces graphs that follow a power-law, known to adequately model this kind of systems [36]. Each agent in

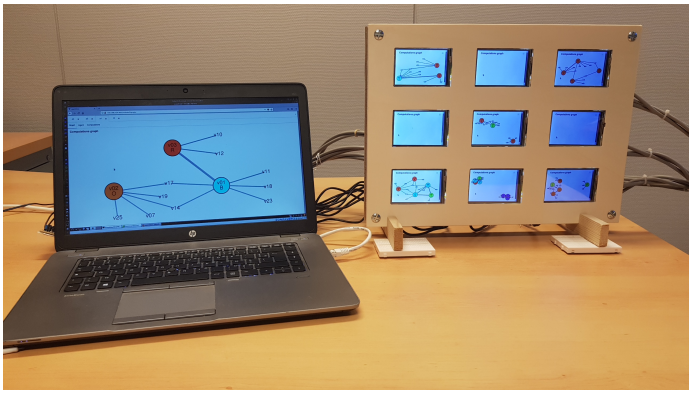


Fig. 12. A physical demonstrator for resilient system developed using pyDCOP on Raspberry Pis

the system is responsible for a subset of the variables and uses a DCOP algorithm to coordinate assignment of color to its variables.

The demonstrator (see Figure 12) is made of a 3×3 grid of Rapsberry Pis, each fitted with a small touch-screen. Each such device runs one pyDCOP agent and displays a GUI presenting the current state of this agent. Agents communicate via WebSocket. A central screen (an internet browser on a TV or computer screen) gives an overall view of the system and the current runtime metrics. During the demonstration, we dynamically remove random agents from the system. Remaining agents coordinate autonomously the repair process, which can be observed on their GUI. The self-repair, which implements DRPM[DMCM] is totally decentralized. A video presenting this demonstration is available online[3].

6.7 Summary and Discussion

This implementation of the full stack of resilience and adaptation mechanisms showed their real applicability. Indeed, while considering environments with several dozen of devices and models, the initial distribution of computations with the greedy heuristic has a good quality and obtained very quickly. Moreover, the replica placement with DRPM is performed in less than 20 seconds for the largest instances. This results in less than one minute setup time. In operation, repair and replication techniques are thus good candidates for in-house deployments, without too much time for users to wait for effective performance. Moreover, the execution of the DCOP algorithms we investigate (A-MaxSum and A-DSA) is not much impacted by our repair method. Repairing a distribution requires less than 10 seconds in the worst cases. The systems continue providing solutions, whilst missing agents, which demonstrates the resilience of these systems. Using problem encoding requiring less computations (choosing constraint graphs instead of factor graphs) is faster. The complexity of the repair process, encoded as a DCOP itself, strongly depends on the number of computations hosted on each agent: when using a factor graph many more computations are required while the number of agents does not change, meaning that on average substantially more computations must be migrated when an agent fails. Moreover, the operation of A-MaxSum is much more impacted by agent removals and repairing than A-DSA, which is very robust to such dynamics. Thus, A-DSA seems a very relevant candidate for implementing SECP in a real dynamic setting.

[3]https://www.dropbox.com/s/ozb0scwskkxqx6p/demoPyDCOP.mp4

7 RELATED WORKS

We now review of some related works focusing on distributed constraint processing for IoT-based AmI, distribution of decisions and computations, and resilience in dynamic settings.

7.1 Constraint Reasoning in Ambient Intelligence

In [4], Degeler and Lazovik use dynamic constraint reasoning for smart environment management. The desired behavior of the home is specified using logical rules. The problem is then encoded as a Dynamic Constraints Satisfaction Problem (DynCSP), where actuators are represented as *controllable variables*, in order to take into account the changes in the environment context. Kaldeli et al. argue in [14] that intelligent behavior in a SHE requires complex functionalities that involve several dynamically selected services provided by independent devices. They thus propose to combine a Service Oriented Architecture (SOA) design with automatic and dynamic service composition. The service composition is implemented using a domain-independent CSP-based planner. The planner reasons upon this model and generates a sequence of actions that must be performed. In [34], Song et al. design a self-adaptive system that takes user preferences into account and applies it to a SHE scenario. The set of adaptation policies is modeled as a CSP, which allows detecting conflicting goals and finding the best configuration that satisfies as many goals as possible. Parra et al. use a CSP to model a dynamic features selection in a software product line and optimize this configuration process [21]. This work is applied to a SHE scenario where an adaptive application must self-adapt to the type of devices and connectivity options currently available in the house.

However, the four aforementioned approaches rely on a central decision node. Some research has been done on applying the DCOP framework to settings that can be considered to be part of IoT and Ubiquitous Computing: sensor networks, radio frequency allocation, traffic light coordination, etc. However, to the best of our knowledge, few past works have focused on the use of the DCOP framework for AmI and SHE settings. In [22], the integration of complex services for AmI is mapped to a Multi-Agent Coordination (MAC) problem, then tackled using the DCOP framework. In their model, each application in the home offers one or several services and is represented by an agent. The resulting DCOP is continuously reasoned upon by the agents and yields a solution where output variables map to the desired behavior of the home. Their implementation is based on ADOPT-N. Fioretto et al. propose in [7] to use a DCOP approach for demand-side management in electric smart grid, based on the Smart Home Device Scheduling (SHDS) problem. This problem is modeled as a distributed scheduling problem, which is mapped to a DCOP that includes both soft constraints, for user preferences and energy consumption, and hard constraints, for temporal goals. Their implementation uses SH-MGM, a custom MGM-based algorithm. As a complement to the SHDS problem, Kluegel, Iqbal, Fioretto, Yeoh, and Pontelli propose in [16] a set of physical models for smart home devices and a data set of problem instances that can be used to benchmark solution methods.

7.2 Distributing Decisions

We argued that the commonly used assumption on DCOP distribution does not hold when working on real world problems and that it is necessary to consider the question of the distribution of decisions over a set of agents. While few works currently exists in this domain, we consider it to be an important part of using DCOP approaches in physical distributed environments. When devising this allocation, there are many elements that we can take into considerations. As a matter of fact, the placement of the computations on agents can have an important impact on the performance characteristics of the global system. Some distributions may improve response time, some others

may favor communication load between agents and some others may be better for other criteria like QoS or running cost. While distribution is seldom studied in the DCOP community, some recent works have started tackling it. We first proposed a distribution method dedicated to factor graphs [30], which aims at minimizing communication costs and hosting costs, as the methods presented in Section 4, which are more generic and can be applied to any graphical model and DCOP solver. Besides, in [15], Khan et al. analyze the placement of constraint graph nodes on agents from a performance point of view; their objective is to find a placement that minimizes the completion time of the DCOP. Like us, they argue that in many problems there are multiple possible mappings of nodes to agents. However, they only consider variable nodes and do not define a more general concept of distribution, which takes into account other types of nodes (factors, several variables, etc.).

7.3 Resilience of Decision Process in Dynamic Settings

One extension of the DCOP framework, namely Dynamic DCOP (Dyn-DCOP), deals with problems whose definition changes during execution, as is it the case in the SECP model. The classical approach, described as *reactive* [17, 24, 25, 37], is directly based on the model of a sequence of static DCOPs, where future DCOPs are entirely unknown. Each static DCOP is simply solved sequentially; any time the problem changes, the new DCOP is solved and the previous solution replaced. The advantage of this approach is that it can theoretically be used with any DCOP algorithm. One drawback of this approach is that this is only applicable if the rate of change is slow enough, compared to the time required to solve one of the DCOPs in the sequence, to terminate solving the problem before a new change occurs. Otherwise, the system would keep solving outdated problems and might even never produce a solution, as it restarts solving a new problem even though no solution as yet been found for the previous one. To avoid this issue, researchers have proposed algorithms that reuse information from previous DCOP to speed-up the search of the current one. A variant of this approach, also *reactive*, is to use DCOP algorithms that can dynamically adapt to the changes and keep working on the updated problem without restarting from scratch. Some works also consider the costs of switching from one solution to another and take this cost into account when selecting a solution for the next DCOP in the sequence. The target here is *solution stability*, which considers the change of solution in these dynamic systems and tries to minimize its effects.

Another approach, called *proactive* [10, 11, 19], is to consider that future DCOPs in the sequence are known in advance or that future potential changes may be at least partially anticipated. In that case, the objective is to look for solutions that are robust to these changes, that is to say solutions that require little or no changes despite the modification of the problem. Unfortunately, current solution methods for this model are either offline or too expensive to be used with anything but a very limited number of agents. In our case, we argue that we cannot predict future changes in the system and we consider that we do not really need the robustness of the solution that the self-stabilizing approach is aiming for. This characteristic is interesting when switching from one solution to another one induces a large cost to the system, as it can be the case for vehicle routing or meeting scheduling. In these cases, the solution stability is indeed primordial, and it is of paramount importance to take into account the cost of transitioning to a new solution versus the benefit provided by this new solution. Therefore, it is potentially more interesting to trade some optimality on the solution for a smaller amount of adaptation when changes occur.

Finally, the notion of k-resilience, where k replicas are placed on different agents, is inspired by techniques from distributed database systems [20], except that here we save computations definitions instead of data.

8 CONCLUSION

We proposed a full process to implement self-configuration in IoT-based smart environment settings. We modeled goal-oriented smart environment scenarios as DCOPs, and provided several methods to distribute and solve it in dynamic setup. We installed decentralized decision-making and coordination of smart things and services by operating DCOP algorithms. To equip this decision process with resilience to environment and infrastructure dynamics, two distributed algorithms have been devised: (i) a replica placement protocol (DRPM) and (ii) a repair protocol (DRPM[DMCM]) relying on replicas placed by DRPM and based itself on DCOP solution method MGM-2. We thus provide self-adaptation to DCOP solvers by using another DCOP-based repair process.

Our contributions have been evaluated experimentally by operating A-MaxSum and A-DSA on dynamic systems where agents disappear during the optimization process. On the different settings we investigated, on both synthetic scenarios and real IoT-based infrastructure build from a network of Raspberry Pis, our repair method DRPM[MGM-2] does not impact the SECP solving process, whilst missing some agents, which demonstrates the resilience of these systems. On our experiments, A-MaxSum operation is much more impacted by agent removals and repairing than A-DSA which is very robust to such dynamics.

This paper raised promising results with respect to resilience in operating distributed optimization processes. We only focused on the worst scenario with agent removals only. We will investigate less stressing scenarios, coming from a broader scope of graph-based computations, like high-performance computing or virtual network functions, where agents may be added to replace disappeared ones. Moreover, we proposed to use MGM-2 as the core repair algorithm, resulting in DRPM[MGM-2] method, but, other lightweight DCOP solution methods might be considered or even designed to the particular case of constraint graph or factor graph repair for instance DCOP algorithms better handling problems with both soft and hard constraints [6, 27]. Finally, approaches for preserving information disclosure while ensuring system resilience, and the resulting trade-off between resilience and privacy will be investigated in future research.

REFERENCES

[1] A.-L. Barabási and R. Albert. 1999. Emergence of Scaling in Random Networks. *Science* 286, 5439 (1999), 509–512.

[2] L. Cohen and R. Zivan. 2017. Max-sum Revisited: The Real Power of Damping. In *Autonomous Agents and Multiagent Systems*. Springer International Publishing, 111–124.

[3] R. Dechter. 2003. *Constraint Processing*. Morgan Kaufmann.

[4] V. Degeler and A. Lazovik. 2013-11. Dynamic Constraint Reasoning in Smart Environments. In *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 167–174.

[5] A. Farinelli, A. Rogers, A. Petcu, and N. R. Jennings. 2008. Decentralised Coordination of Low-Power Embedded Devices Using the Max-Sum Algorithm. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*. pp. 639–646.

[6] F. Fioretto, T. Le, W. Yeoh, E. Pontelli, and T. C. Son. 2014. Improving DPOP with Branch Consistency for Solving Distributed Constraint Optimization Problems. In *Principles and Practice of Constraint Programming* (Cham). 307–323.

[7] F. Fioretto, E. Pontelli, and W. Yeoh. 2017. A Multiagent System Approach to Scheduling Devices in Smart Homes. In *Proceedings of the International Workshop on Artificial Intelligence for Smart Grids and Smart Buildings*. 7 pages pages.

[8] F. Fioretto, E. Pontelli, and W. Yeoh. 2018-03-29. Distributed Constraint Optimization Problems and Applications: A Survey. *Journal of Artificial Intelligence Research* 61 (2018-03-29), 623–698. arXiv:1602.06347

[9] S. Fitzpatrick and L. Meertens. 2003. Distributed Coordination through Anarchic Optimization. In *Distributed Sensor Networks*. Springer, 257–295.

[10] K. D. Hoang, F. Fioretto, P. Hou, Ma. Yokoo, W. Yeoh, and R. Zivan. 2016. Proactive Dynamic Distributed Constraint Optimization. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*. 597–605.

[11] K. D. Hoang, P. Hou, F. Fioretto, W. Yeoh, R. Zivan, and M. Yokoo. 2017. Infinite-Horizon Proactive Dynamic DCOPs. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems* (São Paulo, Brazil) *(AAMAS '17)*. 212–220.

[12] IETF. 2013. DNS-SD.

[13] IETF. 2013. mDNS.

[14] E. Kaldeli, E. U. Warriach, A. Lazovik, and M. Aiello. 2013-05-01. Coordinating the Web of Services for a Smart Home. *ACM Transactions on the Web* 7, 2 (2013-05-01), 1–40.

[15] M. Khan, L. Tran-Thanh, W. Yeoh, and N. R. Jennings. 2018. A Near-Optimal Node-to-Agent Mapping Heuristic for GDL-Based DCOP Algorithms in Multi-Agent Systems. In *Roceedings of the International Conference on Autonomous Agents and Multiagent Systems*. 1613–1621.

[16] W. Kluegel, M. A. Iqbal, F. Fioretto, W. Yeoh, and E. Pontelli. 2017. A Realistic Dataset for the Smart Home Device Scheduling Problem for DCOPs. In *International Workshop on Optimisation in Multi-Agent Systems (OptMAS@AAMAS 2017)*, Vol. 10643. 125–142.

[17] R. N. Lass, E. Sultanik, and W. C. Regli. 2008. Dynamic Distributed Constraint Reasoning.. In *AAAI*. 1466–1469.

[18] R. T. Maheswaran, J. P. Pearce, and M. Tambe. 2004. Distributed Algorithms for DCOP: A Graphical-Game-Based Approach.. In *ISCA PDCS*. 432–439.

[19] Y. Naveh, R. Zivan, and W. Yeoh. 2017. Resilient Distributed Constraint Optimization Problems. In *International Workshop on Optimisation in Multi-Agent Systems (OptMAS@AAMAS 2017)*. 8 pages pages.

[20] M. T. Özsu and P. Valduriez. 2011. Data Replication. In *Principles of Distributed Database Systems, Third Edition*. 459–495.

[21] C. Parra, D. Romero, S. Mosser, R. Rouvoy, L. Duchien, and L. Seinturier. 2012. Using Constraint-Based Optimization and Variability to Support Continuous Self-Adaptation. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing - SAC '12*. 486.

[22] F. Pecora and A. Cesta. 2007-12-12. DCOP for Smart Homes: A Case Study. *Computational Intelligence* 23, 4 (2007-12-12), 395–419.

[23] A. Petcu and B. Faltings. 2004. A Distributed, Complete Method for Multi-Agent Constraint Optimization. In *CP 2004 - Fifth International Workshop on Distributed Constraint Reasoning (DCR2004)*. 15.

[24] A. Petcu and B. Faltings. 2005. Superstabilizing, Fault-Containing Distributed Combinatorial Optimization. In *Proceedings of the National Conference on Artificial Intelligence*, Vol. 20. 449.

[25] A. Petcu and B. Faltings. 2007-11. Optimal Solution Stability in Dynamic, Distributed Constraint Optimization. In *2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'07)*. 321–327.

[26] B. Rachmut, R. Zivan, and W. Yeoh. 2021. Latency-Aware Local Search for Distributed Constraint Optimization. In *Proceedings of the 20th International Conference on Autonomous Agents and MultiAgent Systems*. 1019–1027.

[27] M. Rashik, M. Rahman, M. Khan, M. Mamun-or Rashid, L. Tran-Thanh, and N. R. Jennings. 2021. Speeding up distributed pseudo-tree optimization procedures with cross edge consistency to solve DCOPs. 51 (2021), 1733–1746.

[28] S. J. Russell and P. Norvig. 2016. *Artificial Intelligence: A Modern Approach* (third edition, global edition ed.).

[29] P. Rust, G. Picard, and F. Ramparany. 2016. Using Message-passing DCOP Algorithms to Solve Energy-efficient Smart Environment Configuration Problems. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence (IJCAI-16)*, S. Kambhampati (Ed.). AAAI Press, 468–474.

[30] P. Rust, G. Picard, and F. Ramparany. 2017. On the Deployment of Factor Graph Elements to Operate Max-Sum in Dynamic Ambient Environments. In *Autonomous Agents and Multiagent Systems – AAMAS 2017 Workshops, Best Papers, Sao Paulo, Brazil, May 8-12, 2017, Revised Selected Papers (Lecture Notes in Artificial Intelligence (LNAI), Vol. 10642)*. Springer, 116–137.

[31] P. Rust, G. Picard, and F. Ramparany. 2019. pyDCOP, a DCOP library for IoT and dynamic systems. In *International Workshop on Optimisation in Multi-Agent Systems (OptMAS@AAMAS 2019)*.

[32] P. Rust, G. Picard, and F. Ramparany. 2020. Resilient Distributed Constraint Optimization in Physical Multi-Agent Systems. In *European Conference on Artificial Intelligence (ECAI)*. 195–202.

[33] T. Saraç and A. Sipahioglu. 2014-03. Generalized Quadratic Multiple Knapsack Problem and Two Solution Approaches. *Computers & Operations Research* 43 (2014-03), 78–89.

[34] H. Song, S. Barrett, A. Clarke, and S. Clarke. 2013. Self-Adaptation with End-User Preferences: Using Run-Time Models and Constraint Solving. In *Model-Driven Engineering Languages and Systems*, Vol. 8107. 555–571.

[35] A. Stimson. 1974. *Photometry and Radiometry for Engineers*. OCLC: 833226111.

[36] B. Yao, X. Liu, W. J. Zhang, X. E. Chen, X. M. Zhang, M. Yao, and Z. X. Zhao. 2014. Applying graph theory to the internet of things. In *2013 IEEE International Conference on High Performance Computing and Communications, HPCC 2013 and 2013 IEEE International Conference on Embedded and Ubiquitous Computing, EUC 2013*. 2354–2361.

[37] W. Yeoh, P. Varakantham, X. Sun, and S. Koenig. 2015-12. Incremental DCOP Search Algorithms for Solving Dynamic DCOP Problems. In *2015 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT)*. 257–264.

[38] M. Yokoo, T. Ishida, E.H. Durfee, and K. Kuwabara. 1992. Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In *Proceedings of the 12th International Conference on Distributed Computing Systems*. 614–621.

[39] W. Zhang, G. Wang, Z. Xing, and L. Wittenburg. 2005. Distributed Stochastic Search and Distributed Breakout: Properties, Comparison and Applications to Constraint Optimization Problems in Sensor Networks. *Artificial Intelligence* 161, 1–2 (2005), 55–87.