From Task Graphs to Concrete Actions: A New Task Mapping Algorithm for the Future Internet of Things

Benjamin Billet and Valérie Issarny
ARLES Project-Team, Inria Paris-Rocquencourt,
{benjamin.billet, valerie.issarny}@inria.fr

***Abstract*—Task mapping, which basically consists of mapping a set of tasks onto a set of nodes, is a well-known problem in distributed computing research. As a particular case of distributed systems, the Internet of Things (IoT) poses a set of renewed challenges, because of its scale, heterogeneity and properties traditionally associated with wireless sensor networks (WSN), shared sensing, continous processing and real time computing. To handle IoT features, we present a formalization of the task mapping problem that captures the varying consumption of resources and various constraints (location, capabilities, QoS) in order to compute a mapping that guarantees the lifetime of the concurrent tasks inside the network and the fair allocation of tasks among the nodes. It results in a binary programming problem for which we provide an efficient heuristic that allows its resolution in polynomial time. Our experiments show that our heuristic: (i) gives solutions that are close to optimal and (ii) can be implemented on reasonably powerful Things and performed directly within the network, without requiring any centralized infrastructure.**

Keywords*—*Internet of Things, Task Mapping, Sensor Networks.

I. Introduction

The Internet of Things (IoT) is an extension of the Internet network to objects, or "Things" of the physical world, including the devices and appliances that we are familiar with and inert objects identified by radio-frequency. As a step toward pervasive computing, the IoT aims to enable the Things to cooperate autonomously, and allow humans to interact with the physical world as simply as they do with the virtual world (web pages, web services, streams, etc.) [1].

By involving a large number of embedded and autonomous systems, the IoT shares many characteristics with Wireless Sensor Networks (WSNs) [1]. It is indeed a network where many battery-powered and hardware-limited devices collect data from sensors and continuously process them, possibly in order to control actuators. Similarly to WSNs, wireless communication consumes a lot of energy [2] and in-network processing [3] can significantly increase the network lifetime by avoiding unnecessary data exchanges between devices or with centralized collection points.

Allocating tasks within the network in a way that maximizes some benefits (e.g., network lifetime, througput, accuracy, etc.) is known as the *task mapping problem* and consists in mapping a set of operations (a task) onto a set of nodes, given the properties of each node and the data flow among the operations.

In practice, different forms of the task mapping problem has been studied in various contexts (e.g., distributed and parallel computing, and specifically WSNs) and their complexity is strongly tied to the features of the environment considered. As the IoT is a new environment that derives from WSNs but also introduces its own characteristics, we thus need to formalize a task mapping problem consistent with the IoT use cases and associated technologies, as illustrated by typical IoT scenarios like those presented in [1], [4]–[6].

First, the IoT emphasizes scenarios where Things are shared between users (shared sensing). In these scenarios, the concurrency among tasks must be considered, by analyzing the resources consumed by the previously deployed tasks. In addition, as users open their devices for shared processing, new constraints (e.g., the quantity of resources allocated to the tasks of other users) must be considered, as well as the fair distribution of tasks among the network in order to avoid exhausting some specific nodes (load balancing).

Much of the data that flow within the IoT are acquired from phenomena that constantly evolve, producing endless streams that need to be continuously processed. Because the IoT is highly heterogeneous, various complex operations can be achieved within the network, sometimes requiring specific hardware (e.g., FPGA-based features) or software (e.g., domain-specific libraries) functionalities from the Things that execute them. The amount of resources consumed by these operations can vary quickly over time, depending on their inputs (e.g., audio frames encoded with variable bit rates) and their internal states (e.g., machine learning algorithms). This characteristic must be accurately taken into account to ensure that the network can perform the tasks for as long as required.

As far as we know, there is no task mapping approach that simultaneously handles constraints associated with embedded systems (minimize energy consumption and wireless network communication in a resource-constrained environment), shared sensing (concurrent tasks and load balancing) and continuous processing (varying resource consumption, unbounded streams) in a heterogeneous mobile network (various hardware and software functionalities available, location-aware devices and QoS constraints) assumed to be large-scale.

Toward this goal, Section II first shows that the background of task mapping for distributed systems partially adresses some IoT features in other contexts, motivating the need of a specific IoT formalization. Then, we introduce a formal description of the IoT environment in Section III, describing how data streams and continuous processing tasks are modeled, regarding their hetereogeneity, their varying consumption of resources over time and their constraints. In Section IV, we present a binary programming formulation of the task mapping problem, which gives optimal results. However, as this type of problem can be very costly and time consuming, especially for large real cases, we propose a heuristic algorithm to solve the problem in polynomial time. In Section V, we evaluate

(i) the accuracy of our system model, by analyzing an actual audio-processing task characterized by a highly-varying CPU consumption, and (ii) the efficiency of our algorithm for randomly-generated and real world problems, by comparing the computation times and the quantitative variances of the optimal and heuristic solutions. In addition, we show that our algorithm is simple and fast enough to be implemented directly onto fairly powerful Things (e.g., smartphones or small computers like Raspberry Pi[1]), with a low suboptimality overhead induced by the heuristic. Finally, we conclude in Section VI with a summary of our contribution and perspectives for future work.

II. Background

The task mapping problem has been extensively studied in the literature for various application contexts. For example, in *parallel computing*, where a set of processes has to be mapped on a set of processors/cores [7], in *hardware/sotfware codesign*, where functional specifications must be partitionned into hardware and software [8], and in *distributed computing*, where many operations have to be mapped in a network, a grid, a cluster or the cloud [9]. As a particular case of distributed computing, *WSN* emphasizes the in-network processing in order to increase the network's lifetime [2], [3].

The task mapping problem is known in the literature to be a hard problem [7], often formalized as a mono- or multi-objective optimization problem such as linear or constraint programming [10]–[14]. The problem is usually solved by a heuristic algorithm [11]–[16] or a metaheuristic, e.g., genetic algorithms [10], [17] or swarm intelligence [18]. Other techniques are encountered in the literature to model the problem, such as game theory [19], the single facility location problem [20], [21] or dynamic programming [20], [22]. These solutions, similar to heuristic ones, do not give an optimal solution regarding the entire task and the global network.

In practice, each approach takes into account the specifics of the environment considered. The main assumptions, the evaluation parameters and the constraints vary from one domain to another, according to the goals and the use cases. While the task mapping problem is well-studied, its resolution indeed depends on the context and how the environment is described, leading to a large number of task mapping problem instances.

Specifically, the IoT features a number of particular characteristics and challenges [1], [4] that require revisiting the formulation of task mapping for the given context. Key features to be considered include:

- The IoT is composed of an ecosystem of devices that are shared among users (shared sensing), thus a Thing can run various concurrent tasks for different users. As an incentive for users to share their devices, the tasks must be fairly distributed among them to avoid exhausting some specific devices.
- The IoT is highly heterogeneous, as it connects a huge number of highly heterogeneous devices (e.g., embedded systems, sensors, smartphones, tablets) that exhibit diverse properties in terms of power, autonomy and connectivity. IoT devices can be static or mobile, battery-powered or connected to a power supply, and have a variety of specific functionalities (e.g., a hardware video transcoder, a pattern and image recognition API or a programmable logic device for cryptography).
- The IoT is expected to be an extension of the Internet network and thus leverages associated technologies. Standards like 6LoWPAN [23] are intended to enable resource-constrained devices to communicate over IP. Consequently, each node can rely on the Internet routing infrastructure that is continuously powered, thereby making the routing cost constant, regardless of the number of powered routers involved. Nevertheless, even though the routing problem is less significant, sending each raw sensed value to a processing system (e.g., the cloud) is not recommended, as wireless communication is highly energy-consuming.
- The IoT is strongly tied to the physical world, where every phenomenon is constantly evolving and where the Things' location must be considered for various tasks. These data streams have a strong impact on how the data are produced, processed and consumed. Many sensing scenarios imply that the devices execute sensing and processing tasks until battery depletion or break down [1], [2].

Regarding the prolific literature, the above characteristics have been investigated independently, in particular in WSNs and cloud computing. Practically, the IoT shares various properties with WSNs, as they both involve resource-constrained devices (especially in terms of energy), costly wireless communication, hardware and software heterogeneity, sensors and actuators, and sometimes mobility. The IoT shares as well some characteristics with cloud computing, regarding the network infrastructure and the parallel execution of complex tasks within an elastic pool of computing resources [24].

Basically, task mapping approaches for WSNs and cloud computing relate to dealing with the deployment of dependant heterogeneous tasks onto heterogeneous nodes organized into a single-hop [12] or multi-hop [13], [14], [18], [20]–[22] network. While energy consumption is a fundamental concern in the context of WSNs, cloud computing is oriented toward minimizing the execution time, the energy being further assessed in the context of green cloud computing [25]. Consequently, cloud computing approaches never consider the energy as a crucial limitation of the execution environment, while the IoT tasks, in constrast, require strong guarantees regarding the energy consumption.

Most WSNs approaches do not consider concurrent tasks, and so rarely take into account the prior mapping computations [18], [19]. In addition, the ability of the network to deploy the entire task is usually assumed, without considering the hardware limits of each node [12], [15], [21], [22]. On the contrary, task mapping for cloud computing emphasizes the sharing of computing resources among the tasks [9], [16], [24], and the elasticity of the network makes the cloud virtually able to host any set of tasks. However, in the IoT, tasks and devices are highly heterogeneous and there is no guarantee that (i) a task can be deployed as is on a device and (ii) that the set of devices is able to perform all the tasks. In contrast to cloud computing, these properties must be met before the actual deployment, as a future failure does not only cost time but also energy.

Finally, the specific features of continuous processing, such as the impact of data streams with varying throughput, the evolution of stateful tasks over time, the varying consumption of resources and the long-term computation, are hardly ever

[1] www.raspberrypi.org

considered in task mapping for WSNs as well as for cloud computing. Nevertheless, some studies divide the continuous execution into cycles, that sometimes return to an original state (i.e., a round) [12]–[14]. Consequently, the mapping is computed relatively to the current cycle, without considering the influence of the previous states. Generally, the variations of the streams' throughputs, the resources consumptions and the internal state sizes of tasks are neglected or simplified to simple constants (average or maximum values), which lead to false positive or false negative results.

In light of the above, a new task mapping problem must be formulated according to the specifics of the IoT, by generalizing the existing system models (i.e., the way to describe tasks and devices) for WSNs and cloud computing under a comprehensive and unified IoT system model. Specifically, an accurate way to represent continuous processing must be provided, without simplifying the continuous computation to a sequence of discrete cycles, in order to make the task mapping process more reliable and precise, even when very long time periods have to be considered. In addition to minimizing the energy consumption, this new system model must also ensure some strong properties, such as the fair distribution of tasks inside the network (load balancing) and the guarantee that, even if new tasks are deployed, each running task will have enough resource to be entirely executed.

III. Problem Formulation in the IoT Context

Globally, the task mapping problem consists in finding the best way to allocate a *task* on a set of *nodes* (the Things). As the IoT is an extension of the Internet, and can involve any device able to perform sensing or processing tasks (including, e.g., computer grids and the cloud), the set of nodes is a subset of eligible nodes with respect to the task. We assume that this subset is given by an external registry, sometimes referred to as the ONS (Object Name Service) [1] in the literature. The ONS references the nodes and can be requested to get those that match specific properties.

An IoT-centric task is a composition of *operations* subdividing into sensing, processing and actuation. Two operations a and b can be linked together by a *stream*, which establishes a precedence relationship between them: a datum produced by a causes a reaction from b. The mapping must then consider all the characteristics and *constraints* expressed by each operation, so that they are assigned to nodes that meet the required properties (resources, location, etc.). In addition, as Things are shared, many operations can coexist on a Thing when a new one is deployed. Consequently, every mapping computation depends on previous ones, i.e., a mapping computed at time t must take account of all the tasks that have been deployed on each Thing since the time they were turned on (denoted as t_0).

In the light of the above, we introduce the following notations and vocabulary to formalize the task mapping problem in the IoT context.

a) Streams: The IoT primarily manages data acquired from the physical world. These data evolve continuously and are presented as theoretically infinite streams. Each item produced by a stream is associated with a timestamp (the production time). Usually, the usefulness of an item decreases over time (i.e., old data are potentially less useful than fresh data).

We define a *stream* s as an infinite sequence (I_t) indexed by t (timestamps), and where each term I_t is a *stream item*, i.e., a measurement sensed from the real world. We denote the set of all streams as $\mathbb{S}$, and the set of all items producible by a stream s as $\mathbb{I}_s$. Similarly, we use $s(x)$ to denote the finite sub-sequence of (I_t) where $t_0 \leq t \leq x$ (i.e., the set of items produced by a stream s between the time t_0 and the time x), and $|s(x)|$ its size. In this new sequence, the terms' order is preserved by the time relationship that exists in $s = (I_t)$.

The number of items produced by a stream can vary over time, and we denote its average throughput as $\rho = \lim_{t\to\infty} |s(t)| \div t$. The function $\rho(t)$ describes the throughput variations, by associating at each time t, the corresponding number of items produced per second:

$$\rho(t) = \lim_{\Delta t \to 0} \frac{|s(t)| - |s(t-\Delta t)|}{t - \Delta t} \text{ with } |s(t)| = \int_0^t \rho(t)dt$$

b) Task Graph: Formally, a *task* is modeled by a *task graph* that expresses the operations that must be distributed over the network. It consists in a directed acyclic graph, called $GL = (VL, EL)$, where $VL = \{vl_i\}_{0<i\leq|VL|}$ is the set of *operations* and $EL \subseteq VL^2$ is the set of edges that link two operations. Each edge represents a stream, called $s_{ij} \in \mathbb{S}$, that flows from the operation vl_i to vl_j, i.e., $(vl_i, vl_j) \in EL$.

An operation $vl_i \in VL$ is expressed as a pair (f^i, d^i) where $f^i : \mathbb{S}^p \to \mathbb{S}^q$ is a function that consumes p streams and produces q streams, for d^i seconds with, usually, $d^i \to +\infty$. We denote the sets of input and output streams of vl_i as $IS^i = \{s_{ji} \in \mathbb{S}, (vl_j, vl_i) \in EL\}$ and $OS^i = \{s_{ij} \in \mathbb{S}, (vl_i, vl_j) \in EL\}$.

The throughput of the output streams depends on the throughput of the input streams and the function f^i. However, in practice, we can not get the actual throughput of output streams, as it would actually require processing the input streams. As a solution, we use an approximation technique similar to [20], which consists in defining a reduction ratio, called η, associated to f^i. This ratio represents the reduction factor between the throughput of the input streams and the throughput of the output streams, as:

$$\forall s_{ij} \in OS^i, \rho_{ij}(t) = \eta \sum_{s_{ki} \in IS^i} \rho_{ki}(t)$$

Concretely, this value can be determined analytically for simple operations (counting, computing an average, etc.) or can be obtained empirically by profiling the operation and the physical phenomenon, depending on the accuracy required.

Task modeling implies describing the resource needs of an operation vl_i over time. Unlike common approaches that split the continuous processing into discrete cycles, we believe that continuous consumption of resources should be modeled as continuous functions rather than constants in order to increase the accuracy of the mapping over time. The throughput of vl_i can indeed vary quickly, constantly affecting the need for memory or computation power.

The functions $omem^i(t)$ and $odsk^i(t)$ (which are shaped as constant or increasing functions) express the memory and disk space (bytes) needed by vl_i between the deployment of the task graph, denoted as t_s, and the time t. The function $ocpu^i(t)$ models the number of instructions per second (ips) required to compute the operation at each time t.

The functions $opwr^i_{run}(t)$ and $opwr^i_{com}(t)$ model how much energy has been consumed since t_s. $opwr^i_{run}(t)$ de-

scribes the energy used for computation, and $opwr^i_{com}(t)$ describes the energy used for communication, according to the throughput of the output streams. These two functions are directly linked to the characteristics of the Thing where the operation is executed:

$$opwr^i_{run}(t) = c_{run} \int_{t_s}^{t} ocpu^i(t)dt$$

$$opwr^i_{com}(t) = c_{com} \sum_{s_{ij} \in OS^i} |s_{ij}(t)|$$

with c_{run} the average cost of a cpu instruction and c_{com} the average for sending one byte.

In addition, vl_i can express a set of *constraints* C^i that specify the requirements of the operation regarding the execution environment. Each constraint $c \in C^i$ is described generically as a function, which given a node n, produces a binary value:

$$c(n) = \begin{cases} 1 & \text{if } n \text{ satisfies the constraint,} \\ 0 & \text{otherwise.} \end{cases}$$

Various constraints may be expressed in a similar way:

- Location constraint: Specifies that vl_i must be deployed only in a given area. An area can be described in several ways, such as a set of dots, a circle, or a set of tags (e.g., "building-1"). In the case of mobile Things, these constraints are based on previous locations of the Thing (e.g., the Thing spent $x\%$ of its time in the area).
- Sensing constraint: Specifies that vl_i requires a specific sensor that matches some properties (e.g., sensor type, sample rate, accuracy, response time).
- Feature constraint: Specifies that vl_i requires specific software or hardware functionalities (e.g., video decoding, voice recognition, encryption, mining, etc.).
- QoS constraint: Specifies that vl_i must be deployed in a node that meets some QoS constraints, like those presented in [10] which are related to communication reliability, computation time and security.

c) Nodes: We call N the set of nodes where each node $n_i \in N$ is defined by its overall memory, cpu, disk and energy capacities, respectively denoted $nmem^i$ (byte), $ncpu^i$ (ips), $ndsk^i$ (byte) and $npwr^i$ (coulomb or mAh). Four functions model how these resources are currently being consumed, depending on the operations already deployed on the given node n_i: $nmem^i_0(t)$ (byte), $ncpu^i_0(t)$ (ips), $ndsk^i_0(t)$ (byte) and $npwr^i_0(t)$ (mA). These functions $n*^i_0$ are used to anticipate the future resource states, and can be approximated as the sum of consumed resources by each operation deployed on n_i:

$$npwr^i_0(t) = \alpha_{pwr} + t \cdot \beta_{pwr} + \sum_{vl_j \in D_o} opwr^j(t)$$

$$ncpu^i_0(t) = \alpha_{cpu} + \sum_{vl_j \in D_o} ocpu^j(t)$$

$$nmem^i_0(t) = \alpha_{mem} + \sum_{vl_j \in D_o} omem^j(t)$$

$$ndsk^i_0(t) = \alpha_{dsk} + \sum_{vl_j \in D_o} odsk^j(t)$$

with α_* being the initial consumption of each resource when the Thing was turned on, β_{pwr} the average energy consumed by the idle Thing per unit of time, and D_o the set of operations that are running on n_i. Note that more accurate aggregation models can be used if required, for example to assess the scheduling and context switching cost.

Many metadata are involved in the node description and are used to solve the constraints of the operation: (i) location, (ii) available sensors and actuators, (iii) specific hardware and software functionalities[2], and (iv) the QoS values (e.g., response time, packet loss).

d) Task Mapping: The task mapping problem consists in finding a mapping that maximizes/minimizes some target properties, while satisfying all the constraints of the operations, without exceeding the available amount of resources. In WSNs, the decisive property is the overall energy consumed by the network, which must be as low as possible. For each node, the energy consumption can be computed from the number of instructions that the cpu has to execute and the amount of data that must be exchanged. The energy cost of the latter depends on the transmission cost as two linked operations typically consume less energy when co-located on a node.

However, as the IoT network is shared among independent tasks, it is not sufficient to consider the overall energy consumption when computing task mapping. Indeed, this may lead to systematically selecting similar nodes, thereby exhausting them. This behavior is counterproductive in the context of participatory sensing, as some resource providers would be penalized over others. Similarly, any surrogate proxy or content delivery network would be systematically asked to perform every operation, even the simplest ones, with the consequence of losing the in-network processing benefits. To mitigate this problem, our task mapping approach favors the fair distribution of operations inside the network, by reducing the probability that an operation is mapped on a node when this node is already loaded.

Formally, we call *task mapping* the matrix M of size $|N| \times |VL|$ where $m_{ij} = 1$ when the operation vl_i is deployed on the node $n_j \in N$, and $m_{ij} = 0$ otherwise.

Assessing the quality of a task mapping M can not be done when t approaches infinity. Operations are indeed intended to be executed indefinitely, even if the input streams are bound by windows. If we assume that a node can not be powered up again, the energy consumption is a divergent function. Consequently, it would be meaningless to evaluate the energy consumed by an operation when t approaches infinity. As a solution, we introduce a duration δ_l that expresses the minimal *lifetime of the task*. This lifetime, which is expressed as a user constraint, indicates that a task must be executed at least δ_l units of time. Subsequently, solving the task mapping problem consists in finding the best mapping, given the consumption of resources when t approaches δ_l.

In the case of concurrent tasks, we have to guarantee that a future mapping will not invalidate the previous mapping (i.e., a new deployed task must not lead to battery depletion before the completion of every existing task). To solve this problem, the mapping has to be computed when t approaches a maximum lifetime, denoted as Δ_l. Let D be the set of deployed tasks, expressed as triples (GL, t_s, δ_l) with t_s the task deployment time, δ_l the task lifetime, and $(GL^0, t^0_s, \delta^0_l)$ the task considered for the current mapping computation. The maximum lifetime

[2]Nevertheless, we consider that every node is able to perform a fixed set of trivial operations (counting, filtering, etc.).

is then defined as $\Delta_l = max(\lambda, \delta_l^0)$ with $\lambda = max(t_s^i + \delta_l^i) - t_s^0, \forall (GL^i, t_s^i, \delta_l^i) \in D$.

IV. Task Mapping Optimization Problem

To clarify the presentation, we introduce four functions, called $pwr^i(t), mem^i(t), cpu^i(t)$ and $dsk^i(t)$. These functions represent the overall consumption of each resource on a node n_i. The functions $mem^i(t), cpu^i(t)$ and $dsk^i(t)$ aggregate memory, cpu and disk usage of all the operations deployed onto n_i. The function pwr^i combines the energy consumed by all the operations for computation, $opwr^j_{run}$, and communication, $opwr^j_{com}$. The latter is ignored when two linked operations are mapped on the same node.

$$pwr^i(t) = \sum_{vl_j \in VL} m_{ji} \cdot \left(opwr^j_{run}(t) + \sum_{s_{jk} \in OS^j} (1 - m_{ki}) \cdot opwr^j_{com}(t) \right)$$

$$mem^i(t) = \sum_{vl_j \in VL} m_{ji} \cdot omem^j(t)$$

$$cpu^i(t) = \sum_{vl_j \in VL} m_{ji} \cdot ocpu^j(t)$$

$$dsk^i(t) = \sum_{vl_j \in VL} m_{ji} \cdot odsk^j(t)$$

Solving the task mapping problem in the IoT context consists in (i) minimizing the consumption of resources by each Thing in the network and (ii) load balancing the operations among the nodes. For this purpose, we introduce four objective functions that must be minimized, called $g_{pwr}, g_{mem}, g_{cpu}$ and g_{dsk}. These functions evaluate the consumption of each resource (energy, memory, cpu and disk) using the functions $pwr^i(t), mem^i(t), cpu^i(t)$ and $dsk^i(t)$. In addition, the fair distribution of the operations is evaluated using the amount of resources consumed by an operation vl_i on a node n. For example, if vl_i consumes 20% of n_1's memory, and 15% of n_2's memory, g_{mem} will give a better score to n_2, regardless of the amount of memory available on n_1 and n_2. This mechanism mitigates the penalization effect described in the previous section, by guaranteeing that the percentage of resources consumed will be similar for each Thing.

$$g_{pwr}(t) = \sum_{n_i \in N} \frac{pwr^i(t)}{npwr^i - npwr^i_0(t_s + t)}$$

$$g_{mem}(t) = \sum_{n_i \in N} \frac{mem^i(t)}{nmem^i - nmem^i_0(t_s + t)}$$

$$g_{cpu}(t) = \sum_{n_i \in N} \frac{\int_0^t cpu^i(t)dt}{(t_s + t) \cdot ncpu^i - \int_0^{t_s+t} ocpu^i_0(t)dt}$$

$$g_{dsk}(t) = \sum_{n_i \in N} \frac{dsk^i(t)}{ndsk^i - ndsk^i_0(t_s + t)}$$

In order to reduce the complexity of multi-objective programming, an aggregation function G is used to combine the results of the four objective functions. We will use a sum in the remainder of this paper, but G can be any polynomial function (e.g., weighted sum or quadratic polynomial) provided as a parameter of the task mapping process. Using G, finding a task mapping can be formulated as a binary programming problem [26], since the decision variables m_{ij} are only binary:

minimize $G\Big(g_{pwr}(\delta_l), g_{mem}(\delta_l), g_{cpu}(\delta_l), g_{dsk}(\delta_l)\Big)$

subject to $\forall vl_i \in VL, c_k \in C^i, n_j \in N$ and $T = t_s + \Delta_l$

$$\sum_{n_j \in N} m_{ij} = 1 \quad (1)$$

$$npwr^j_0(T) + pwr^j(\delta_l) \leq npwr^j, \quad (2)$$

$$nmem^j_0(T) + mem^j(\delta_l) \leq nmem^j, \quad (3)$$

$$\int_0^T ncpu^j_0(t)dt + \int_0^{\delta_l} cpu^j(t)dt \leq T \cdot ncpu^j, \quad (4)$$

$$ndsk^j_0(T) + dsk^j(\delta_l) \leq ndsk^j \quad (5)$$

$$m_{ij} \leq c_k(n_j) \quad (6)$$

Constraint (1) ensures that each operation is mapped strictly once. Constraints (2) to (5) guarantee that, for each node, the computed mapping will not consume more resources than are available. It has to be noted that, in the context of shared sensing, the constants $npwr^i, nmem^i, ncpu^i$ and $ndsk^i$ can be replaced by user-provided values that define the amount of resources dedicated to executing the task. Finally, Constraint (6) ensures that all the constraints expressed by each operation are satisfied by the chosen nodes.

Actually, the problem as expressed above is not linear because Constraint (2) multiplies the decision variables m_{ji} and m_{ki}. However, the problem can be linearized by expanding the energy constraint and using a technique similar to [13]. This consists in introducing $|VL|^2 \times |N|^2$ new decision variables m_{ijkl}, with $m_{ijkl} = 1$ when the operation i is deployed on node j and the operation k is deployed on node l, and 0 otherwise. $pwr^i(t)$ is then updated as:

$$\sum_{vl_j \in VL} \left(m_{ji} \cdot opwr^j_{run}(t) + \sum_{s_{jk} \in OS^j} (m_{ji} - m_{jiki}) opwr^j_{com}(t) \right)$$

In addition, new constraints must be added to maintain the consistency of the decision variables. Constraints (7) and (8) ensure that m_{ij} or m_{kl} will never be 0 if m_{ijkl} equals 1. Constraint (9) guarantees that m_{ijkl} will never be equal to 0 if m_{ij} and m_{kl} equal 1.

$$\forall vl_i, vl_k \in VL, n_j, n_l \in N, m_{ij} - m_{ijkl} \geq 0 \quad (7)$$

$$\forall vl_i, vl_k \in VL, n_j, n_l \in N, m_{kl} - m_{ijkl} \geq 0 \quad (8)$$

$$\forall vl_i, vl_k \in VL, n_j, n_l \in N, 1 + m_{ijkl} \geq m_{ij} + m_{kl} \quad (9)$$

Binary linear programming problems can be solved optimally, however, this can be highly time-consuming in practice [26]. As a solution, we introduce a heuristic and a greedy algorithm that (i) solve the problem in a reasonably short time and (ii) is simple enough to be executed opportunistically in the network by a fairly powerful Things elected dynamically.

Algorithm 1 scans the set of operations and the set of nodes and tries, firstly, to ascertain if the mapping of an operation vl_i on a node n_j is *feasible*, i.e., that each constraint of vl_i is satisfied by n_j (resources, location, functionalities, QoS, etc.). The routine *is-feasible*, presented in Algorithm 2, is called for each pair (vl_i, n_j) and checks if the pair is a feasible solution and should be evaluated further.

If the pair (vl_i, n_j) is feasible, a score is computed based on the resources' usage (e.g., n_j would consume $x\%$ of its

Algorithm 1 Construction of M

Require: N, GL
  for all $vl_i \in VL$ **do**
    $val \leftarrow \infty$
    for all $n_j \in N$ **do**
      if is-feasible(vl_i, n_j) **then**
        $pwrs \leftarrow pwr^i(\delta_l)/(npwr^j - npwr_0^j(T))$
        $mems \leftarrow mem^i(\delta_l)/(nmem^j - nmem_0^j(T))$
        $cpus \leftarrow \int_0^{\delta_l} cpu^i(t)dt/(T \cdot ncpu^i - \int_0^T ocpu_0^i(t)dt)$
        $dsks \leftarrow dsk^i(\delta_l)/(ndsk^j - ndsk_0^j(T))$
        $x \leftarrow G(pwrs, mems, cpus, dsks, qos)$
        if $x < val$ **then**
          $val \leftarrow x$
          best-node $\leftarrow n_j$
        end if
      end if
    end for
    if *best-node* $\neq \emptyset$ **then**
      $M[vl_i][best\text{-}node] \leftarrow 1$
    else
      no satisfying solution
    end if
  end for

Algorithm 2 is-feasible

Require: vl_i, n_j
  if $npwr_0^j(T) + pwr^i(\delta_l) > npwr^j$
  or $nmem_0^j(T) + mem^i(\delta_l) > nmem^j$
  or $\int_0^T ncpu_0^j(t)dt + \int_0^{\delta_l} cpu^j(t)dt > T \cdot ncpu^j$
  or $ndsk_0^j(T) + dsk^i(\delta_l) > ndsk^j$ **then**
    return false
  end if
  for all $c_k \in C^i$ **do**
    if not $c_k(n_j)$ **then**
      return false
    end if
  end for
  return true

remaining resources to compute vl_i) using an aggregation function G. In constrast to the binary programming problem which is limited to polynomial functions, any function G is supported by the algorithm. In addition, the QoS parameters are evaluated at the same time as an additional input of G.

The time complexity of the algorithm is $O(|N| \times |VL| \times \sum_{vl_i \in VL} |C^i|)$ as, for each pair (vl_i, n_j), *is-feasible* performs an unchanging number of steps, except for the constraint space C^i. The space complexity is $O(|N| \times |VL|)$, as only the size of the matrix M varies depending on N and VL.

V. Assessment

The evaluation of our work includes three steps. First, we show the benefit of using continuous functions to model the resource consumptions, compared to approximated constants (e.g., the averages or the maximum), for a profiled audio decoding task. Second, we assess the efficiency of our algorithm for various randomly-generated problems where we compare quantitatively the optimal solutions to the heuristic solutions. In addition, we collect the execution time of the algorithm, both on a desktop computer and a smartphone, in order to show that the mappings can be computed by simply using a reasonably powerful node of the network. Finally, we analyze the results of our algorithm for an actual concrete task that is often considered in task mapping and WSNs literature [13], [14], [27], i.e., air conditioning management in a smart building. We perform the experiments for varying complexity, i.e., several building sizes and lifetimes.

	Average	Maximum	Continuous	Reality
Parallel tasks	16	6	8	8.2 $\sigma = 0.48$
Consecutive tasks	17	7	17	16.7 $\sigma = 1.3$

Table I: Estimation of the device capacity.

A. Accuracy in the continuous domain

Our first experiment evaluates the benefit of using continuous functions to model the resource consumptions, compared to approximated constants, such as average or maximum values. The modeled task consists into (i) decoding two audio streams, (ii) synchronizing them, and (iii) applying a simple echo effect by shifting the samples. The audio data are encoded using the lossless FLAC format, which implies that the bitrate is naturally varying as well as the time needed to decode an audio frame. Consequently, while (ii) and (iii) are featured by a constant consumption, (i) needs to be profiled. This is done by running it several times on a smartphone (Galaxy Nexus) and collecting actual CPU measurements (memory is not analyzed here, as the CPU consumption is far more higher than the memory consumption). The CPU continuous consumption function is then built by performing a Discrete Fourier Transform (DFT) with the acquired measurements.

We used our system model to determine (i) how many tasks can be executed in parallel without reaching the full CPU load (such a case would introduce artefacts while playing the audio stream), and (ii) how many consecutive tasks can be executed until the device runs out of battery. The experiment was performed three times, with $ocpu(t)$ equals to: the continuous CPU consumption function based on the DFT, its average and its maximum value. In addition, we measured the values of (i) and (ii) empirically, by deploying tasks on an actual device. The results are presented in Table I and shows that the estimation based on the continuous CPU consumption function is more accurate than the estimations based on average and maximum values. By hiding details, the average leads to false positives (the deployment would have failed with 16 tasks) while the maximum, by being too pessimistic, leads to false negatives (2 more tasks could have been deployed).

B. Heuristic efficiency

The efficiency of our algorithm mainly depends on how the task graph is browsed. When the algorithm evaluates the mapping of an operation vl_i on a node n_j, there is an uncertainty related to the unmapped successors, as the energy cost is reduced when two operations are mapped onto the same node. Consequently, if a successor of vl_i is unmapped when the pair (vl_i, n_j) is evaluated, the resource consumption can be overvalued. If we call VL' the set of unmapped nodes when (vl_i, n_j) is evaluated, then the potential error (the extra cost) is:

$$\epsilon^i = \sum_{s_{ik} \in OS^i} |VL' \cap \{vl_k\}| \cdot opwr_{com}^i(\delta_l)$$

When ϵ^i is high, wrong suboptimal solutions can be favored. In the worst cases, the algorithm may fail to find a solution (dead end). Consequently, a graph search algorithm produces better results if it minimizes ϵ^i for each (vl_i, n_j).

In our experiments, we evaluate problems of different sizes, for four search methods: (i) source to sink, (ii) sink to source, (iii) random node and (iv) lowest ϵ^i first (or "epsilon search"). Each problem, denoted $A \times B$, consists in mapping a task graph of A operations on B nodes, which are both generated

	Source to Sink			Sink to Source		
	Overhead	Min/Max	Failures	Overhead	Min/Max	Failures
5×5	18.26	[0, 44.52]	0	8.38	[0, 33.34]	0
5×10	12.66	[0, 39.17]	0	5.51	[0, 21.7]	0
10×5	17.61	[5.55, 56.12]	7	14.05	[0, 42.39]	0
10×10	16.93	[3.38, 37.3]	13	9.82	[0, 25.29]	0
15×5	17.86	[4.94, 51.83]	23	13.47	[0, 39.06]	0
	Epsilon			**Random**		
	Overhead	Min/Max	Failures	Overhead	Min/Max	Failures
5×5	8.19	[0, 28.06]	0	25.96	[0, 34.52]	0
5×10	5.42	[0, 21.7]	0	21.87	[0, 24.5]	0
10×5	12.87	[0, 42.39]	0	25.57	[5.01, 45.63]	2
10×10	9.74	[0, 24.77]	0	22.18	[3.61, 23.98]	6
15×5	13.47	[0, 35.27]	0	24.77	[5.27, 36.34]	14

Table II: Overhead and failure percentages.

	5×5	5×10	10×5	10×10	15×5	20×20	50×50	100×100	1000×1000
Optimal	63ms	1s	8s	60s	483s	>20mn	>20mn	>20mn	>20mn
Heuristic (PC)	<1ms	<1ms	<1ms	1ms	2ms	3ms	6ms	12ms	680ms
Heuristic (Android)	<1ms	1ms	2ms	3ms	28ms	101ms	630ms	1s	10s

Table III: Computation time.

randomly. The random process picks A operations, numbered from 1 to A, from a fixed set of variously-complex operations (constant, polynomial, etc.). Constraints are then randomly generated for each operation using predefined sets of locations and operations. Finally, random links are created between operations, cycles being avoided by following a simple rule: $\forall(i,j) \in EL, i < j$. The B nodes are generated randomly as well, using a set of devices' profiles (motes, embedded devices, smartphones). The functions $npwr_0^i$, $ncpu_0^i$, $nmem_0^i$ and $ndsk_0^i$, that model the current load, are also selected from a set of predefined functions.

For each problem $A \times B$, one hundred *non-trivial* problems are generated. Non-trivial means that the task graph can not be deployed on a single node. In other words, the overall resources consumption of each task graph, excluding the communication cost, is at least x times greater than the energy, cpu, memory and disk resources available on the most powerful node.

The generated problems are then processed by the linear programming solver *lp_solve*[3] and the proposed greedy algorithm. The latter is executed both on a desktop computer (dual-core 3.2GHz, 4GB memory) and a smartphone (dual-core 1.2GHz ARM Cortex-A9, 1GB memory). In addition, we compute the worst possible solution and compare it to the heuristic solution, in order to show that the results of the algorithm are always closer to the optimal solution than the worst. Finally, we count the number of *mapping failures*, where the algorithm does not find a feasible solution while the optimal solver does find one. Table II presents the average gap between the optimal and heuristic solutions for each class of problems, and the percentage of failures. Table III shows the time taken to compute the solutions.

We notice that in most cases, the sink to source and the epsilon searches give the best results (in terms of average values and maxima) and should be preferred. This is because the objective function only considers emission costs, as they are more significant than reception costs, especially in a wireless context. As the sink to source walk maps the terminal nodes first, the uncertainty is then reduced when a pair (vl_i, n_j) is evaluated, as most predecessors of vl_i are already mapped.

In addition, these two searches drastically reduce the number of failures, because the algorithm is rarely blocked

[3]lpsolve.sourceforge.net/5.5

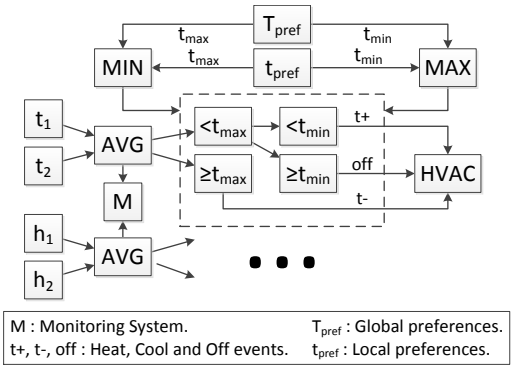

Figure 1: Example of task graph for a room with two temperature sensors.

in uncertain suboptimal solutions that could be dead ends. However, if the results are close to the optimal, there is a small overhead, especially for the hardest problems. This phenomenon can be observed in Figure 2, which presents the results of 100 runs for the 15×5 experiment and the measured values of the objective function for the optimal, the heuristic (random and epsilon search) solutions and the worst solutions. This overhead is due to the behaviour of the algorithm, that gradually maps the operations and quickly moves toward a solution; when an operation vl_i is mapped on a node n_j, the probability of mapping a predecessor vl_k of vl_i (i.e., $s_{ki} \in IS^i$) on the same node n_j strongly increases.

However, Figure 2 shows that the overhead of the algorithm is quite low (on average between 10 and 20 %). In addition, the algorithm is rarely blocked in a suboptimal solution, and always far from the worst one. In addition, this overhead is balanced by the significant improvment in computation time, even for large problems (1000×1000). Further, due to its reasonable complexity, the algorithm can be deployed directly on a simple smartphone, as shown in Table III. As a benefit, the mapping computation can be performed inside the network, avoiding the need for a centralized powerful computation point.

C. Reference Problem: HVAC Mapping

We have shown the efficiency of our algorithm for a set of non-trivial problems. We now evaluate it for a real-world problem: the management of a heating, ventilation and air conditioning (HVAC) system. It is a common problem, frequently cited in the WSN literature [13], [14], [27].

According to this case study, a building is divided into several rooms equipped with temperature and humidity sensors that are used to control the rooms' air-conditioner. Each air-conditioner can be managed locally by the users, by manually defining their preferences for the room. In addition, the central building management system is used by the administrator to define global policies about the permitted temperature and humidity (e.g., never exceed 30°C). These global policies have priority over the local preferences, and we want the system to collect data from the sensors and control the air-conditioner accordingly.

In practice, each room contains x temperature sensors and y humidy sensors, that can communicate through a wireless network with the air-conditioner and the control center that stores the global policies and the HVAC monitoring software. A task graph example, corresponding to a room with two temperature sensors t_1, t_2 and two humidity sensors h_1, h_2

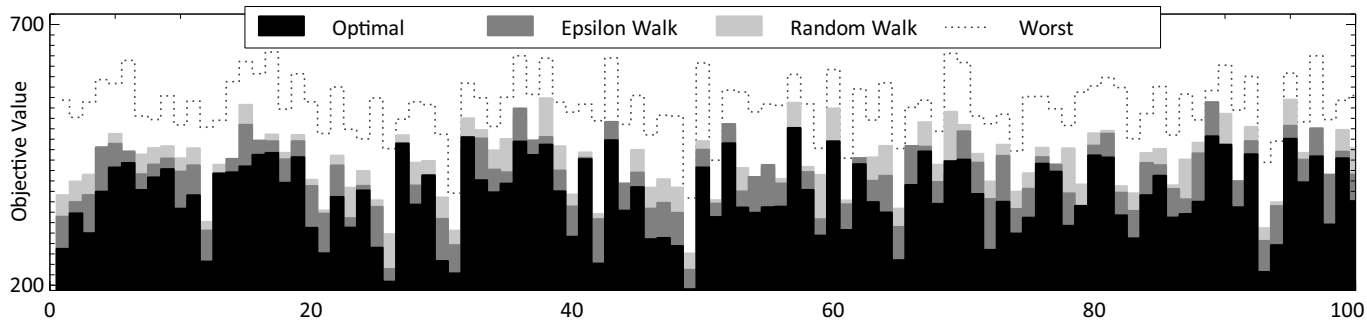

Figure 2: 100 runs of the non-trivial 15×5 experiment (i.e., mapping 15 operators on 5 nodes).

Problem Class	Device Types	Lifetime (δ_l)	Load by nodes (%)
easy	small computers	24 hours	< 10
average	sunspot	90 days	$[10, 50]$
hard	motes	1 year	> 50

Table IV: Description of problem classes.

	(2, 2)	(2, 4)	(2, 6)	(4, 2)	(4, 4)	(4, 6)	(6, 2)	(6, 4)	(6, 6)
Trivial	50ms	120ms	400ms	580ms	4s	18s	19s	83s	727s
Average	60ms	150ms	500ms	650ms	5s	23s	26s	92s	932s
Hard	1822s	2971s	3h	7h	16h	28h	-	-	-

Table V: Computation times for optimal solution.

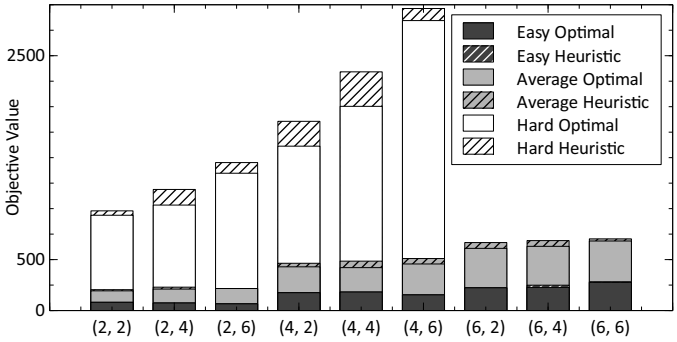

Figure 3: Objective values.

is presented in Figure 1. In this example, the sensed data are aggregated by the operations *AVG* (average) and then compared to some thresholds. These thresholds are provided by the operations *MIN* and *MAX* that decide between the global policies (provided by the control center) and the local preferences (provided by the air-conditioner). If the sensed values are outside the desired range $[t_{min}, t_{max}]$ and $[h_{min}, h_{max}]$, the air-conditioner is instructed to adjust the temperature and the humidity accordingly. The task graph is highly detailed on purpose, as a higher granularity leads to a better distribution when necessary.

Similarly to the previous experiments, we study problems of different sizes, denoted as (A, B), for buildings of A rooms with B humidity and temperature sensors, with a sampling rate of one measurement per minute. Each problem consists in mapping $17A + 2B + 3$ operations in $2B$ nodes.

For each instance (A, B), we simulate many variants with different complexities, by changing the lifetime δ_l and the type of device (motes, small computers) used to execute the operation. We classify these problems into three classes of difficulty, presented in Table IV. In practice, we stated that the HVAC problem is: (i) easy for a lifetime of 24 hours with powerful nodes (e.g., Raspberry Pi), (ii) average for a lifetime of 90 days (3 months) with averagely powerful nodes, like Sun SPOT[4], and (iii) hard for a lifetime of one year with resource-constrained motes (e.g., Waspmote[4] or Arduino-based devices[4]).

The results of all the experiments, size and class of problems, are presented in Table 3. Here, the solver finds optimal solutions for bigger task graphs, as the complexity of the problem is lower than in the first experiments. There are indeed enough devices to compute the operations and the solution space exploration is then performed more quickly. However, some high-complexity problems are not solved even after many hours, as shown by the computation times in Table V.

We notice that the gap between optimal and heuristic solutions is non-existant for the simplest problems, i.e., the problems with the least potential uncertainty ϵ^i. For the other problem classes, the overhead induced by the heuristic solutions is limited (less than 20%) even for large or hard problems, such as the (6, 6) problem which consists in mapping 117 operations on 12 devices.

All these results, both for the randomly-generated problems and the real HVAC problem, demonstrate that our heuristic algorithm finds good mappings with a low overhead that is compensated by the decrease in computation time. In practice, the execution time is short enough to enable the algorithm to be implemented directly on some Things without the need for any dedicated infrastructure.

VI. Conclusion

In this paper, we presented a formalization of the task mapping problem in the context of the IoT that captures more accurately the specifics of the domain, and a heuristic algorithm that finds the mapping in polynomial time. We implemented this algorithm and showed experimentally that the computed mappings are close to optimal solutions, with a small computation time that enables it to be used directly in the network, simplifying greatly the deployment (specifically for autonomous networks).

As far as we know, the features of our approach are unique, gathering together the characteristics of continuous processing, real-time shared sensing and the IoT. Regarding continuous processing, our model considers the operations as an infinite process having a varying impact on the overall resource consumption, depending on the operations' complexity, the sampling rate of sensors, the throughput of the streams, and the propagation of the events in the network. In addition, throughput and load variations are modeled as continuous time functions that express accurately the evolutions of each operation at each time, as opposed to an approximate sequence of discrete cycles. Concerning shared sensing, our approach takes into account the previously deployed tasks and their

[4]sunspotworld.com, libelium.com, arduino.cc

evolution when a new mapping is computed. In addition, the fair distribution of the nodes in the network (load balancing) is considered as a criterion at least as important as energy consumption in order to not penalize the most powerful nodes. Regarding the IoT itself, our approach considers heterogeneous Things, in terms of storage and computation capacities, and in terms of specific hardware and software functionalities that they offer. The common IoT constraints (location, sensor type, functionalities, etc.) are evaluated to compute the mapping, as well as the QoS parameters. Finally, our algorithm produces reliable results by guaranteeing, as soon as the mapping is computed, that the network will be able to execute the task. In addition, the algorithm ensures that the task will be executed for a specified lifetime, as a hard constraint during the mapping computation.

Nonetheless, our work can be improved in several ways. First, we would like to estimate what is the maximum gap between optimal and heuristic solutions, given various task graph topologies.Second, our algorithm is a centralized one. Distributed task mapping algorithms have some disadvantages (significant message overhead, continuous migration that drains batteries, etc.), but are more scalable in practice [22]. However, our approach is light enough to be used in large-scale networks, as it can be deployed in reasonably powerful Things, like smartphones. In addition, like all state-of-the-art task mapping algorithms, the amount of user input needed to compute the mapping may be high, depending on the desired accuracy. Some of these data require profiling the task and the operations and require predictive models about the physical phenomenon measured.

We plan to create and evaluate a distributed version of our algorithm, that would take advantage of (i) the data acquired locally in the Things (e.g., extract the time functions from real measurements) and (ii) a delegation mechanism in order to self-organize the network in the best possible way to accomplish the set of given tasks. At the same time, we plan to use these locally acquired data to perform the dynamic reconfiguration (remapping and migration) when the network evolves significantly (mobility, new nodes, broken down or depleted nodes, failures, etc.).

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Computer Networks*, vol. 54, no. 15, 2010.

[2] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Computing Surveys*, vol. 43, no. 3, 2011.

[3] E. Fasolo, M. Rossi, J. Widmer, and M. Zorzi, "In-network aggregation techniques for wireless sensor networks: a survey," *IEEE Wireless Communications*, vol. 14, no. 2, 2007.

[4] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of Things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, vol. 29, no. 7, 2013.

[5] M. Whitney and H. Richter Lipford, "Participatory sensing for community building," in *Proc. of the 29th International Conference on Human Factors in Computing Systems*, 2011.

[6] C. de Farias, L. Pirmez, F. Delicato, W. Li, A. Zomaya, and J. de Souza, "A scheduling algorithm for shared sensor and actuator networks," in *Proc. of the 27th International Conference Information Networking*, 2013.

[7] S. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. 30, no. 3, 1981.

[8] J. Teich, "Hardware/software codesign: The past, the present, and predicting the future," *Proceedings of the IEEE*, vol. 100, no. Special Centennial Issue, 2012.

[9] X. Grehant, I. Demeure, and S. Jarp, "A survey of task mapping on production grids," *ACM Computing Surveys*, vol. 45, no. 3, 2013.

[10] S. Malek, N. Medvidovic, and M. Mikic-Rakic, "An extensible framework for improving a distributed software system's deployment architecture," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, 2012.

[11] J. Li, M. Qiu, J.-W. Niu, and T. Chen, "Battery-aware task scheduling in distributed mobile systems with lifetime constraint," in *Proc. of the 16th Asia and South Pacific Design Automation Conference*, 2011.

[12] Y. Yu and V. K. Prasanna, "Energy-balanced task allocation for collaborative processing in wireless sensor networks," *Mobile Networks and Applications*, vol. 10, no. 1-2, 2005.

[13] A. Pathak and V. Prasanna, "Energy-efficient task mapping for data-driven sensor network macroprogramming," *IEEE Transactions on Computers*, vol. 59, no. 7, 2010.

[14] F. H. Bijarbooneh, P. Flener, E. Ngai, and J. Pearson, "Energy-efficient task-mapping for data-driven sensor network macroprogramming using constraint programming," in *Proc. of the 9th International Workshop on Constraint Modelling and Reformulation*, 2010.

[15] W. Li, F. C. Delicato, and A. Y. Zomaya, "Adaptive energy-efficient scheduling for hierarchical wireless sensor networks," *ACM Transactions on Sensor Networks*, vol. 9, no. 3, 2013.

[16] J. Octavio Gutierrez-Garcia and K. M. Sim, "A family of heuristics for agent-based elastic cloud bag-of-tasks concurrent scheduling," *Future Generation Computer Systems*, vol. 29, no. 7, 2013.

[17] M. Gen and L. Lin, "Multiobjective evolutionary algorithm for manufacturing scheduling problems: state-of-the-art survey," *Journal of Intelligent Manufacturing*, vol. 24, no. 3, 2013.

[18] I. Caliskanelli, J. Harbin, L. Indrusiak, P. Mitchell, D. Chesmore, and F. Polack, "Runtime optimisation in wsns for load balancing using pheromone signalling," in *Proc. of the 3rd IEEE International Conference Networked Embedded Systems for Every Application (NESEA)*, 2012.

[19] N. Edalat, C.-K. Tham, and W. Xiao, "An auction-based strategy for distributed task allocation in wireless sensor networks," *Computer Communications*, vol. 35, no. 8, 2012.

[20] Z. Lu, Y. Wen, R. Fan, S.-L. Tan, and J. Biswas, "Toward efficient distributed algorithms for in-network binary operator tree placement in wireless sensor networks," *IEEE Selected Areas in Communications*, vol. 31, no. 4, 2013.

[21] G. Chatzimilioudis, A. Cuzzocrea, D. Gunopulos, and N. Mamoulis, "A novel distributed framework for optimizing query routing trees in wireless sensor networks via optimal operator placement," *Computer and System Sciences*, vol. 79, no. 3, 2013.

[22] Z. Abrams and J. Liu, "Greedy is good: On service tree placement for in-network stream processing," in *Proc. of the 26th IEEE International Conference on Distributed Computing Systems*, 2006.

[23] G. Mulligan, "The 6lowpan architecture," in *Proc. of the 4th workshop on Embedded networked sensors*, 2007.

[24] W. Zhang, L. Wang, Y. Ma, and D. Liu, "Design and implementation of task scheduling strategies for massive remote sensing data processing across multiple data centers," *Software: Practice and Experience*, vol. 43, no. 10, 2013.

[25] F. Cao and M. Zhu, "Energy-aware workflow job scheduling for green clouds," in *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*, 2013.

[26] M. Junger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleybank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, *50 Years of Integer Programming 1958-2008*. Springer, 2010.

[27] M. Demirbas, "Wireless Sensor Networks for Monitoring of Large Public Buildings," Univ. at Buffalo, Tech. Rep., 2005.