

On Reconfiguring Embedded Application Placement on Smart Sensing and Actuating Environments

Nikos Tziritas, Samee Ullah Khan and Thanasis Loukopoulos

Abstract In a smart home environment appliances and objects have sensing, actuating and wireless communication capabilities. Recent embedded middleware initiatives aim at providing an easy to use programming framework for developing applications that require the cooperation of sensing and actuating nodes. To achieve this, the mobile agent paradigm is adopted under which an application consists of a set of communication agents residing at nodes of the system with adequate resources. Of particular importance in such system is to decide which agent to place where so that network traffic is optimized. In this chapter we discuss the problem of reconfiguring an initial agent placement so that the resulting scheme is more energy efficient. We formulate the aforementioned *agent placement problem (APR)* which turns out to be particularly challenging due to the various constraints involved. A heuristic algorithm based on graph coloring is proposed and evaluated against a greedy approach under various scenarios.

Nikos Tziritas
Dept. of Computer and Communication Engineering, Univ. of Thessaly, Volos, Greece, e-mail: nitzirit@inf.uth.gr

Center for Research and Technology Thessaly (CERETETH), Volos, Greece

Samee Ullah Khan
Department of Electrical and Computer Engineering, North Dakota State University, Fargo, USA, e-mail: samee.khan@ndsu.edu

Thanasis Loukopoulos
Dept. of Informatics and Computer Technology, TEI of Lamia, Lamia, Greece, e-mail: luke@tcilam.gr

Center for Research and Technology Thessaly (CERETETH), Volos, Greece

1 Introduction

In a smart environment, appliances, devices and apparatus are enhanced with sensing and/or actuating capabilities. Applications require the communication and co-operation of smart objects in order to achieve the required functionality. A number of middleware initiatives, e.g., Pobicos [13], Rovers [2], MagnetOS [9], aim at providing an easy way to code applications for smart environments by splitting the initial application into a number of cooperating mobile agents that can be freely installed and moved among the system nodes, given adequate resources. Such agent placement must be reconfigured from time to time due to changes in the environment. Here, we investigate the agent placement problem (APR) problem from the standpoint of optimizing the network traffic.

The rest of the chapter is organized as follows. Subsections 1 illustrate the application model, states our motivation and describes related work; Section 2 gives the problem definition; Section 3 illustrates the proposed algorithm which is experimentally evaluated in Section 4; finally, Section 5 concludes the paper.

1.1 Application model

Consider the following application meant to identify a potential fire hazard:

```
while (true)
  for each room R
    if avg(‘temp’,R)>X & \& avg(‘hum’,R)<Y
      notify\_user();
```

One possible structure of the application into cooperating agents is shown in Fig. 1. Separate temperature and humidity gathering agents are installed at each room. They report to aggregation agents which compute the average of these parameters and communicate with alarm triggering agents which in turn send results to the hazard notification agent at the root.

Notice that this is not the only application structure available. For instance, one could have a two level hierarchy whereby the measuring agents report directly to a single agent responsible for carrying the whole application logic (i.e., averaging, triggering and notifying). Also, notice that an application structure should not necessarily form a tree hierarchy. For instance one measuring agent might report to two different aggregators. Regardless, of the particulars of an application structure, we can distinguish the resources required by the agents into two categories: (i) *functionality resources*, e.g., temperature or other sensing/actuating capabilities and (ii) *computational resources*, e.g., processing power, memory, bandwidth etc.

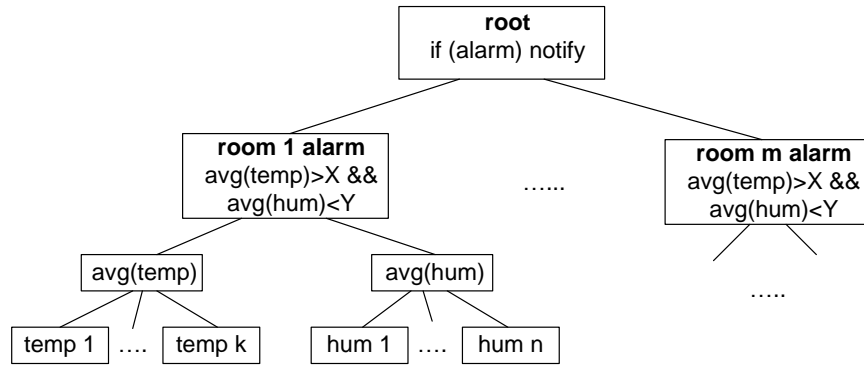


Fig. 1 Application example

1.2 Motivation

All agents require computational resources, but only some of them require functionality resources, we call the latter *non-generic* agents and distinguish them from generic ones which require only computational resources. In the example of Fig. 1 the temperature and humidity gathering agents in the lowest level of the hierarchy are non-generic agents. Non-generic agents have limitations concerning their mobility, e.g., a temperature gathering agent must rest on a node with similar sensing capabilities. Furthermore, multiple copies of them might exist in the network and placing two copies at the same node should be forbidden (for instance it makes no sense to have two humidity measuring agents of the same application installed at the same node).

In this work we assume a smart home or smart office environment with a central monitoring entity, e.g., a desktop computer or a set-top box. This entity is responsible for deciding about the agents' placement, having full knowledge of the present placement scheme, the network and the respective smart node capabilities. The goal is to place agents in nodes having the required resources, so that communication traffic is minimized, thus reducing battery consumption and saving bandwidth. Our work is inspired by the Pobicos system [13] (among others), whereby when the agents of an application are first created they are placed in random nodes having the adequate resources.

1.3 Related work and contributions

Mobile code/agent based systems are subsumed in the general category of systems that afford programming abstractions for WSNs. Such systems include Pobicos [13], Rovers [2], Mate [10], Agilla [3], SensorWare [1], Smart Messages [4], Magne-

tOS [9], Pleiades [6] and DFuse [11] to name a few, while a survey can be found in [7].

The above systems are distinguished depending on whether they provide transparent agent placement [6, 9, 11, 13] or require user intervention [1, 2, 3, 4, 10]. Pleiades and MagnetOS follow a similar logic concerning agent placement whereby an agent is moved towards the center of gravity of its respective load in an attempt to minimize the total communication traffic.

DFuse is an architectural framework for performing data fusion in sensor networks. It performs fusion function placement and dynamic relocation in an attempt to minimize communication. Similar placement problems originate from the other fields as well. For instance, in the database field, [16] tackles the problem of placing filter and join query operators in the nodes of a WSN. A similar query optimization problem is discussed in [12] for a network resembling the Internet, while in [8] the authors consider the placement of services in the Internet. Finally, in [17] the authors consider both operator and caching placement over a WSN. We differ from these works either in system's scope [8, 12], or with regards to the applications they consider [11, 16, 17], as we target general applications, which might not solely involve fusion and/or querying.

A similar to MagnetOS placement algorithm is also used in the Pobicos system which is under development. In [15] the authors compared this algorithm which is based on migrating agents one by one, against an algorithm that considers agent groups for migrating. Simulation results demonstrated that group migrations have a higher potential, while both algorithms deviate from optimality as system nodes become more resource constrained. The same authors in [14] discussed centralized algorithms for opening space in order to accommodate a newcomer agent, as well as for reconfiguration once the agent is already placed. The optimization target was to maximize the lifetime of the first node that depletes its battery.

Our work although inspired by the aforementioned systems (particularly Pobicos) is more general in scope in two major ways. The first is that we consider a graph application structure, whereas in Pobicos the application structure is assumed to be a tree. The second one is that we enable non-generic agents to migrate provided the target node meets their functionality requirements. This introduces additional constraints to be tackled making the problem harder to solve.

In this chapter we focus on the reconfiguration problem proposed at [14] but with more general system and application assumptions so as not to restrict our results to Pobicos and a different target function, namely, network traffic cost instead of node lifetime. The contributions include a novel reconfiguration algorithm which follows a different principle compared to the one in [14]. Although the algorithm can be run in a distributed fashion we restrict our discussion to its centralized execution. Experimental comparisons against a greedy reconfiguration algorithm demonstrate the viability of our approach.

2 Problem Definition

This section first introduces the system model, then proceeds with formulating the APR problem.

2.1 System model

Let the system comprise of N nodes with sensing/actuating capabilities denoted by n_i , $1 \leq i \leq N$ and A agents denoted by a_k , $1 \leq k \leq A$. Nodes have a certain amount of computational resources and agents a certain demand requirement for them. Let $r(n_i)$ depict the level of computational resources available at n_i (can represent e.g., memory or bandwidth). Similarly we denote by $r(a_k)$ the amount of these resources that must be available at a node in order for agent a_k to execute correctly. It is straightforward to include more than one computational constraints in the model if necessary.

As explained the mobility of non-generic agents is not only dependent on the computational resources at the destination. A binary $N \times A$ eligibility matrix L is used to encode whether a node has the required *functionality resources* (thus is eligible to hold the agent) as follows: $L_{ik} = 1$ if n_i provides the required by a_k functionality, 0 otherwise. Recall also that non-generic agents belonging to the same application and providing the same functionality must not reside at the same node. We model it through an $A \times A$ binary mutual exclusion matrix F , whereby $F_{kw} = 1$ if a_k must not reside at the same node with a_w , 0 if no such requirement is necessary.

Nodes communicate with each other via some wireless technology (which is treated as a black box). The underlying routing paths are abstracted as a graph, its vertices representing nodes and its edges representing bidirectional routing-level links between them. In this work we consider *tree-based routing*, i.e., there is exactly one path for connecting any two nodes. Let h_{ij} be the length of the path between n_i and n_j ; equal to 0 for $i = j$. Communication between agents is captured via an $A \times A$ matrix C , where C_{kw} denotes the data units sent on average from agent a_k to a_w per time unit.

2.2 Problem formulation

A binary $N \times A$ matrix P is used to encode agent placement at nodes as follows: $P_{ik} = 1$ if a_k is in n_i , 0 otherwise. The APR problem can then be stated as follows: given an initial placement P^{old} of application agents on nodes, define a new placement P^{new} so that the overall network load due to agent communication is minimized. As a secondary optimization target we also require that the network cost due to the migrations performed in order to switch from the initial placement P^{old} to the

new one P^{new} is also minimal. The network load T due to agent communication is given by:

$$T = \sum_{k=1}^A \sum_{n=1}^A \left(C_{kn} \cdot \sum_{i=1}^N \sum_{j=1}^N h_{ij} P_{ik} P_{jn} \right) \quad (1)$$

Thus, the benefit in agent communication terms by switching from P^{old} to P^{new} is:

$$B = T^{old} - T^{new} \quad (2)$$

A single migration incurs a cost proportional to the agent size and the hop distance between the start and destination node. We assume that there exists a single monitoring node (let n_m) which also acts as an entry point for the arriving agents in the system (e.g., for security reasons) and keeps an immutable copy of all agents' code. Migrations are performed by sending a copy of the agent's code from n_m and the agent's status from the node where the agent currently resides. For simplicity, we assume that the size of the status is negligible, compared to the code size, which is denoted by s_k . Therefore, given an initial placement P^{old} and the one that must be implemented P^{new} , the total migration cost M can be computed as follows:

$$M = \sum_{k=1}^A \sum_{i=1}^N P_{ik}^{new} (1 - P_{ik}^{old}) h_{mi} \cdot s_k \quad (3)$$

Minimizing agent communication cost 1 and migration cost 3 are conflicting, since 3 is minimized if P^{new} is the same as P^{old} . Intuitively, 3 acts as an overhead which can be fully or partially offset by the reduction in agent communication cost 2, depending on whether P^{new} will remain unchanged for a sufficient large time. Let α be a constant depicting the importance of migration cost over agent communication. Then the APR problem can be stated as: *given an initial agent placement P^{old} find a new placement P^{new} such as the following function is optimized:*

$$\max D = B - \alpha M \quad (4)$$

subject to the following constraints:

$$\sum_{k=1}^A P_{ik}^{new} r(a_k) \leq r(n_i), \forall i \quad (5)$$

$$\sum_{i=1}^N P_{ik}^{new} = 1, \forall k \quad (6)$$

$$P_{ik}^{new} (1 - L_{ik}) = 0, \forall i, k \quad (7)$$

$$F_{kw} P_{ik}^{new} P_{iw}^{new} = 0, \forall i, k, w \quad (8)$$

Constraint (5) states that node capacity constraints should not be violated. Constraint (6) enforces that each agent should be placed at exactly one node. In addition, this placement must be eligible functionality wise (7) and there should not be conflicts with other agents residing at the same node (8). By constraint (5) it is easy to see that the relevant APR decision problem is NP-complete having (among others) a knapsack component [5]. In the following section we present heuristics to tackle it.

3 Algorithms

The proposed algorithms are based on the concept of pair-wise agent exchanges between system nodes. We begin our discussion by presenting the core exchange method in a system consisting of two nodes, then generalize for a system of $N > 2$ nodes. We also present a greedy method used for comparison reasons in the experiments.

3.1 The APR problem with 2 nodes

Consider the APR problem for the case where the system consists of two nodes n_1 and n_2 and a monitoring node n_m . All nodes are assumed to have 1-hop distance between each other. We illustrate the functionality of the algorithm through an example. Assume a total of 5 agents are already placed at the system's nodes as follows: a_1, a_2 and a_3 are placed at n_1 and a_4, a_5 at n_2 . Table 1 depicts the load generated due to agent communication, as well as the agents' resource requirements.

Table 1 Agent communication load and resource requirements

$r(a_k)$	a_1	a_2	a_3	a_4	a_5	
2	a_1	0	4	0	1	0
1	a_2	1	0	0	1	0
2	a_3	0	2	0	2	3
3	a_4	2	0	4	0	0
2	a_5	0	0	5	5	0

Let the capacity of the two nodes (resource wise) be: $r(n_1) = 7$ and $r(n_2) = 5$. Assuming migrations incur no cost and that no functionality or mutual exclusion constraints exist, APR can be transformed to a graph coloring problem as follows. Let $G(V, E)$ be the agent communication graph, whereby the vertices of the graph correspond one to one with the agents and an edge (a_k, a_w) exists if a_k and a_w communicate with each other. Each edge has a weight $w(a_k, a_w)$ which equals the communication cost between a_k and a_w across both directions, i.e., $w(a_k, a_w) =$

$C_{kw} + C_{wk}$. Furthermore, each vertex has a weight $w(a_k)$ equaling the computational resource demands by a_k . Fig. 2 shows the graph G of the example.

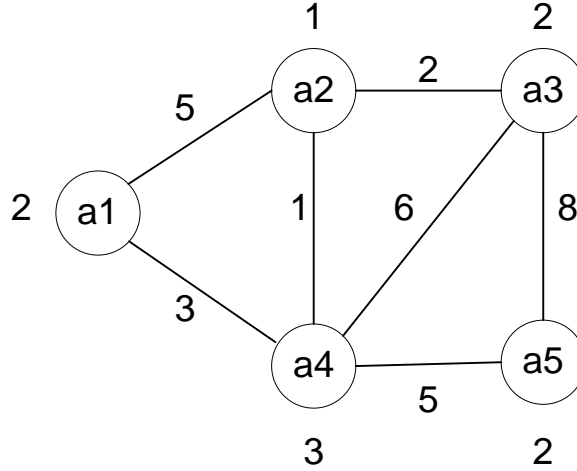


Fig. 2 Agent communication graph

APR can then be restated as: having constructed the graph G , find a 2-color scheme (e.g., red, black) so that the total weight of the red vertices does not exceed $r(n_1)$, the total weight of the black vertices does not exceed $r(n_2)$ and the total weight of edges connecting one red with one black vertex is minimized. Notice, that the two problems are equivalent since we can rewrite (4) as (T^{old} is a constant): $minD' = T^{new} + \alpha M$. Thus, with the migration cost being zero only T^{new} must be minimized, which is what the coloring problem does.

Migration cost is included as follows. The communication graph is extended by adding two more vertices, representing the two system nodes (n_1 and n_2 in the example). These vertices have 0 weight and are colored, e.g., n_1 is red and n_2 black. Links of the form (n_1, a_k) and (n_2, a_k) connecting n_1 respectively n_2 , with all agent vertices are created. The respective weights of these links are defined as follows. If a_k currently resides at n_1 , then $w(n_1, a_k)$ is set to equal the migration cost for transferring a_k from n_1 to n_2 and $w(n_2, a_k)$ is set to 0. The case where a_k resides at n_2 is symmetric. The reason for setting link weights as above, is that if a_k is colored with the same color as the node it currently resides, then no migration cost should be charged while in the opposite case the migration cost must be charged. Fig. 3 illustrates the above extension for the agents a_3 and a_5 assuming that all agent sizes is 8 and that the constant $\alpha = 0.5$. Red vertex (n_1) is shown striped, while black vertex (n_2) is shown grayed. Since all migrations are assumed to be performed via the monitoring node (hop distance of 1 against n_1 and n_2), all edges whereby the migration cost must be charged have a weight of 4 (equals α *agent size*hop distance).

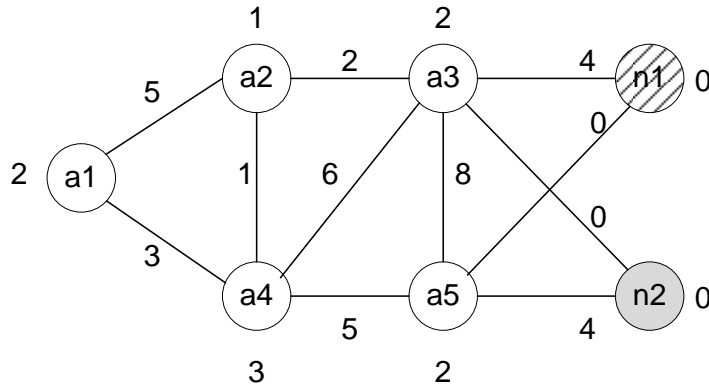


Fig. 3 Extending the communication graph

Functionality constraints are included in the model by placing a fixed color at a vertex. For instance, if in the example $L_{21} = 0$, then a_1 vertex will be painted in red, i.e., a_1 will be forced to stay at n_1 . Finally, mutual exclusion constraints are included by adding constraints on the allowable coloring. For example in modeling that $F_{kw} = 1$, it is equivalent to say that a_k and a_w vertices must have different colors.

3.2 The agent exchange algorithm

Here we present an algorithm namely, the agent exchange algorithm (AXA), to come with a solution for the 2-node version of APR. AXA uses the transformation of APR into the equivalent coloring problem presented in Sec. 3.1.

The algorithm works in iterations. In each iteration, the edge with the highest weight is selected and the vertices it connects are merged. In case these vertices have a mutual exclusion constraint, the merging is not performed and the edge connecting them is colored grey (i.e., not to be considered further). Otherwise, the new vertex has the cumulative weight of the previous ones and their remaining edges. If any of the merged vertices is colored then the new vertex will also be colored (with the same color). In case the two vertices to be merged are colored with different color each, merging is not performed and the respective edge becomes grey. Fig. 4 shows the resulting graph by merging a_3 with a_5 .

After, the merge is performed, AXA attempts to find a feasible coloring in the new graph. To this end it solves knapsack two times, once for n_1 and once for n_2 , with the candidate objects being the a_k vertices (the benefit and size of each object being the vertex' weight). In the previous example (Fig. 4) by solving knapsack on n_1 (the red node) we get the following objects to be placed: a_1, a_2, a_3, a_5 , filling the resource capacity of n_1 which is 7. Having obtained a knapsack solution for n_1 , the algorithm checks if the remaining objects fit in n_2 . In the example only a_4

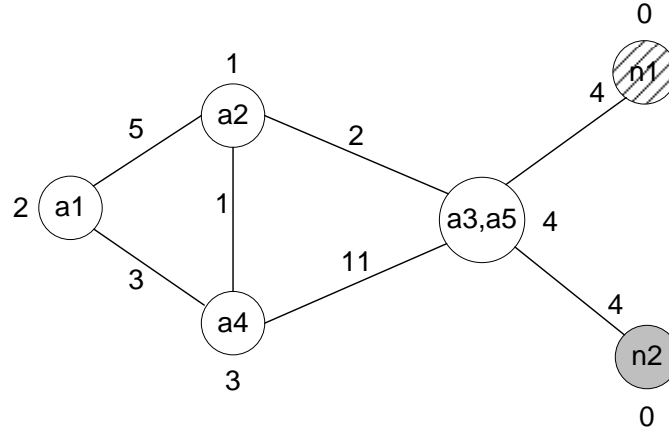


Fig. 4 Resulting graph after merging

remains which fits in n_2 since $r(n_2)$ was assumed 5. If so, the algorithm keeps the merged vertex without coloring it and proceeds with the next iteration. Otherwise, the algorithm attempts to find a valid placement by solving knapsack for n_2 (the black node) and checking whether the remaining objects fit at n_1 . If after trying both knapsack solutions AXA is unable to find a valid placement involving all the objects it backtracks to the graph state before merging, marking the edge under consideration as grey.

The algorithm continues with the merging process by selecting the next uncolored edge of highest weight and so on so for until all remaining nodes and/or edges are colored. The final agent placement is derived by solving knapsack as above for the remaining (if any) uncolored nodes.

3.3 Extending to N nodes

Tackling the case of $N > 2$ nodes is done with the pair-wise reconfiguration algorithm (PRA), the pseudocode of which is shown in Alg. 1.

PRA iterates through all node pairs applying AXA. If during an iteration AXA manages to define a better placement according to (4), the process reiterates, otherwise it ends producing the final agent placement. In order for AXA to successfully optimize locally, i.e., within a node pair, the agent placement, adaptations are required to the way agent communication load and migration costs are modeled. We illustrate them through an example.

Assume the network of Fig. 5, with 7 nodes plus the monitoring node n_m . Let the agents of Table 1 be already placed on n_2 and n_5 as follows: n_2 has a_1, a_2, a_3 and n_5 has a_4 and a_5 . In other terms n_2 and n_5 in this example have the same role as n_1 and

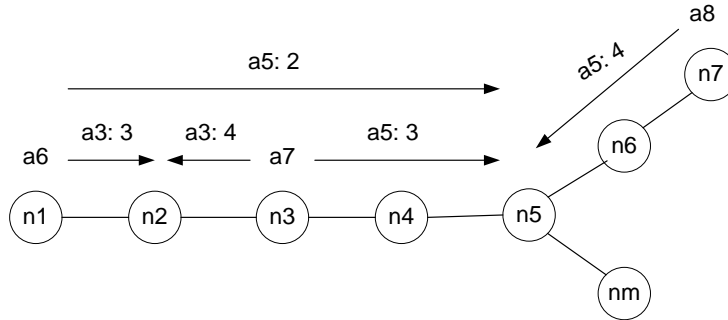
Algorithm 1 Pseudocode of PRA

```

1: found:=true;
2: while found do
3:   found:=false;
4:   for i=1 to N do
5:     for j=1 to N do
6:       apply AXA over  $(n_i, n_j)$  pair;
7:       if  $D > 0$  then
8:         found:=true;
9:         keep AXA changes;
10:      else
11:        discard AXA changes;
12:      end if
13:    end for
14:  end for
15: end while

```

n_2 in the example of Sec. 3.1. Assuming only these agents exist in the network, the equivalent graph coloring problem is similar to the one in Fig. 3, with the exception being that the hop count must be taken into account both on edges representing agent communication (a_k, a_w) and on edges representing migration cost (n_i, a_k) . So all $w(a_k, a_w)$ edge weights will be multiplied by a factor of 3 (the hop distance between n_2 and n_5), while all edge weights $w(n_i, a_k)$ will be multiplied by the hop distance between n_m and the node of the opposite color with which n_i was painted. For instance, $w(n_2, a_2)$ will remain 4 since the distance between n_m and n_5 (the black node) is 1, while $w(n_5, a_4)$ will now be 16 since $h_{m2} = 4$. Fig. 6 depicts the resulting problem graph. For clarity, the only edges connecting n_2 and n_5 with agent vertices are the ones related to a_3 and a_5 .

**Fig. 5** Example network

In the general case, agents placed on nodes other than the pair in question (n_2, n_5) might generate load towards some of the agents placed on the pair. Fig. 5 gives an example, whereby 3 more agents exist, namely: a_6 which is placed at n_1 , a_7 at n_3

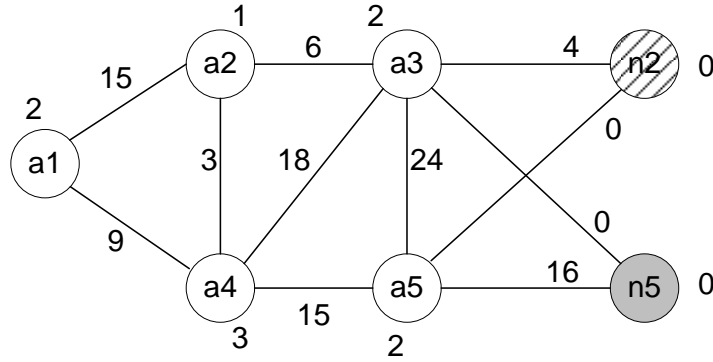


Fig. 6 Including hop distance in the problem graph

and a_8 at n_7 . The figure also shows the load these agents generate towards the ones placed at n_2 and n_5 , specifically: $C_{63} = 3$, $C_{65} = 2$, $C_{73} = 4$, $C_{75} = 3$ and $C_{85} = 4$.

Such external (to the node pair) load must be incorporated to the graph coloring model in order for it to map to APR correctly. This external load can be viewed as another form of node related cost in the problem graph, as was the case with migration. Consider for instance the migration of a_5 from n_5 to n_2 . Aside from the migration cost of 16 to transfer a_5 from n_5 to n_2 there will also be a change on the cost of the external load directed to/from a_5 . For instance, the load generated by (a_5, a_8) communication will not incur a cost of 8, but rather a cost of 20 since the hop distance between the two agents will increase from 2 to 5. In order to incorporate the above case in the problem graph it suffices to augment $w(n_5, a_5)$ by the load incurred if a_5 moved to n_2 , i.e., 20 and $w(n_2, a_5)$ by the incurred load if a_5 stayed in n_5 , i.e., 8. Repeating the process for all the external loads of a_5 results in $w(n_5, a_5)$ being augmented by a factor of: 20 (a_8 's load) + 3 (a_7 's load) + 2 (a_6 's load) for a total of 25 and $w(n_2, a_5)$ being augmented by: 8 (a_8 's load) + 6 (a_7 's load) + 8 (a_6 's load) for a total of 22. Fig. 7 illustrates the final graph coloring transformation for the example of Fig. 5. Again, to avoid cluttering, only edges between n_2 , n_5 and a_3 , a_5 are shown.

The adaptations done to include the external (to the node pair) communication load, correctly charge the relevant costs when the agent vertices will be colored. For instance, if a_5 is colored with the same color as n_5 (black), meaning a_5 is placed at n_5 , then the external cost of a_5 directed towards n_5 must be charged without any migration cost, which is what the (n_2, a_5) link will do. Similarly, if a_5 is placed at n_2 , i.e., colored red, then both the migration cost and the external cost of the communication load towards a_5 located at n_2 must be charged, which is achieved through the (n_5, a_5) link.

However, a subtle change must be made to AXA in order for it to function properly. Recall, that AXA selects the link of highest weight and attempts to merge the nodes it involves. The rationale for the decision is to attempt to place together agents

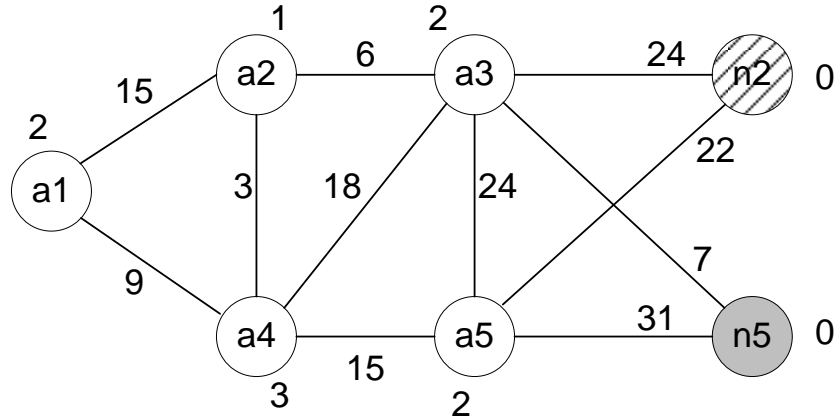


Fig. 7 Resulting problem graph

communicating heavily with each other. So, if a_3 and a_5 are placed together, then the communication load among them will be alleviated and a benefit of $w(a_3, a_5) = 24$, will occur. However, the same is not true when considering edges involving a vertex representing a system node. For instance, if a_5, n_5 are merged the actual benefit will not be 31, but rather the cost difference between placing a_5 at n_5 and at n_2 . As discussed the actual cost will be charged correctly, through the link (n_2, a_5) , nevertheless, AXA would have begun the coloring/merging process from a less beneficial edge. For this reason, at the sorting step of AXA all edges of the form (n_i, a_k) do not participate with their weights, but rather with the weight difference: $w(n_i, a_k) - w(n_j, a_k)$, assuming n_i and n_j are the system nodes for which AXA runs.

3.4 Greedy algorithmic approach

Thus far, we have shown how the APR problem when viewed locally from the standpoint of a node pair can be transformed into a graph coloring problem. We also discussed both an algorithm to derive a solution to the coloring problem (AXA) and how it can be invoked in order to tackle the global APR problem (PRA). For comparison reasons here, we discuss another algorithm to solve APR based on the greedy approach.

Starting from the initial placement, *Greedy* iteratively selects an agent to migrate and performs the migration. Specifically, at each iteration all $A * N$ possible migrations are considered and the one that optimizes (4) the most, subject to the constraints (5)-(8) is selected. The process is repeated until no further beneficial migration can be defined.

4 Experiments

This section describes the experimental evaluation of PRA. Section 4.1 presents the experimental setup. Section 4.2 gives a comparison of PRA and Greedy against exhaustive search for a small experiment, while in Section 4.3 we compare PRA against Greedy for a larger experimental setup. Finally, Sec. 4.4 summarizes the experimental findings.

4.1 Experimental setup

Due to the fact that the POBICOS middleware is currently under development we conducted the experimental evaluation using simulation experiments. The details of the simulation setup which roughly followed the ones in [14, 15] are briefly given below.

Network generation. Two types of networks were constructed, one with 7 and one with 30 nodes. In both networks an extra node played the role of the monitoring node. Nodes were placed randomly in a 100×100 2D plain and assumed to be in range of each other if their Euclidean distance was less than 30 distance units. In the resulting network topology graph, a spanning tree was calculated and acted as the corresponding tree-based routing topology.

Application generation. The application tree structure is generated randomly, based on the (given) number of non-generic agents. The initial non-generic agents are split in disjoint groups of 5, and for each group 2-5 agents are randomly chosen as children of a new generic agent. In next iterations, orphan (generic and non-generic) agents are (again) randomly split in groups of 5 and the process of parent creation is repeated, until a single agent remains which becomes the root of the application. With the above method which was used in [15] the resulting application is a tree, its leaves consisting of non-generic agents. Since the scope of this work is broader tackling general application graphs as opposed to trees, we alter the resulting application tree as follows. For each generic agent two more non-generic agents were assumed to be its children, thus, these non-generic agents had two (or more) parents. Two different application structures were generated with this way *app-10* and *app-40*, each with 10 and 40 non-generic agents respectively.

Application traffic. We assumed that the communication load between a non-generic and a generic agent was between 10 to 50 data units per time unit. For the load between generic agents we considered three cases: (i) *avg*: a generic agent sends the average of the load received from its children, corresponding to a data aggregation scenario; (ii) *lsum*: a generic agent sends to its parent the sum of the loads received from its children, corresponding to a forwarding scenario; and (iii) *lmix*: half of the generic agents (randomly chosen) generate load according to *avg* and the other half according to *lsum*. Unless otherwise stated, the constant α (see (4)) governing the importance of migration cost versus communication load was set to 0.01.

Other parameters. The size of agents varied uniformly between 100 and 1,000 data units. All the non-generic agents that have the same parent were assumed to share one common functionality resource requirement and had a mutual exclusion constraint among them. Non-generic agents with different parents were assumed to differ in at least one functionality requirement. In the experiments we begin with an initial placement and run the algorithms to define a better one. This initial placement is derived by placing the non generic agents first. Specifically for every group of non-generic agents with the same parent (let ng in cardinality), $(1 + \beta)ng$ nodes (randomly selected) were assumed to have adequate functionality resources to hold the agents, i.e., for a node n_i and an agent ak such as above, $L_{ik} = 1$. Unless otherwise stated, constant β takes a value of 0.5. In the initial placement the non-generic agents were placed randomly to nodes having the required functionality in such a way so as to respect mutual exclusion constraints as well. Having placed the non-generic agents, generic agents were placed afterwards, again in a random fashion. Last, in the experiments we assume that the computational resource of interest is memory and that all nodes start with an initial capacity equaling the size of the agents assigned to them by the initial placement.

4.2 Comparison against the optimal

In this set of experiments we compare both PRA and Greedy against the optimal solution derived through exhaustive search. For this reason we used the smaller 7-node network type and *app-10* application. Five different network topologies were generated and five different *app-10* applications. Results depict the average of the combined runs (25 in total).

First we recorded the performance of the algorithms regarding the quality of the placement scheme they reach, as a percentage of the optimal performance. Assuming that in the initial placement *init* communication load is incurred per time unit, that in the optimal scheme *opt* communication load is incurred and that in the placement calculated by the algorithms *alg* communication load is incurred, the percentage of the optimal performance achieved by an algorithm is characterized by the ratio: $(init - alg) / (init - opt)$, i.e., how much load reduction an algorithm achieves compared to the optimal. Table 2 presents the results for PRA and Greedy for two different load types: *lavg* and *lsum*. We also varied the amount of extra free capacity available at the system nodes. So for instance *lavg(2)*, means that each node had just enough capacity to hold the agents allocated there in the initial placement plus extra space equaling 2 times the average agent size.

Table 2 Solution quality compared to the optimal

	<i>lavg(1)</i>	<i>lavg(2)</i>	<i>lavg(3)</i>	<i>lsum(1)</i>	<i>lsum(2)</i>	<i>lsum(3)</i>
Greedy	81.7%	88.4%	95.4%	86.8%	86.9%	86.9%
PRA	85.5%	100%	100%	89.8%	100%	100%

We can observe from Table 2 that PRA constantly outperforms the simpler Greedy algorithm. In fact, the difference between PRA and the optimal scheme is not large when capacity is tight (plus one extra space for an agent), while with a less tight constraint, PRA achieves the optimal performance. It is also worth noting that the Greedy algorithm never achieves an optimal performance.

Table 3 Migrations performed

	$lavg(1)$	$lavg(2)$	$lavg(3)$	$lsum(1)$	$lsum(2)$	$lsum(3)$
Greedy	9.2	9.4	10.4	8.7	10.1	10.1
PRA	8.8	10.0	10.0	9.2	9.5	9.5

We also recorded in Table 3 the number of migrations performed by each of the algorithms. Recall, that the application type used was *app-10*, involving 10 non-generic agents and roughly 6 generic, for a total of 16 agents. Results here were mixed with PRA doing more or less migrations compared to Greedy depending on the scenario. However, the fact that in certain cases where PRA achieves the optimal, e.g., $lavg(3)$, $lsum(3)$, PRA also performs less migrations compared to Greedy, illustrates even more the merits of our approach.

4.3 Experiments with a larger network

Here we conducted experiments using the larger network case (30 nodes + the monitor). Five different network topologies were generated and each experiment depicts the average. Eight applications of type *app-40* were assumed to be initially placed, while the load model was *lmix*. We plot the percentage of load reduction achieved compared to the initial placement, i.e., $(init-alg)/init$. Since the exhaustive algorithm could not produce results within acceptable time, we only compared PRA against Greedy.

Fig. 8 demonstrates the performance of the algorithms as more capacity is added at each node e.g., the value of 4 in the x-axis means that each node has capacity equaling the necessary one to hold the agents initially placed there, plus 4 times the average agent size. The first thing to notice, is that the achievable saves by both algorithms increase to the surplus capacity at the nodes which is expected since with tighter capacity agents that should have been placed together might not be able to do so. Notice that PRA manages to reduce the initial load by more than 60% in all cases and by roughly 10% more compared to Greedy, a fact that further reinforces the viability of our approach.

Last, in Fig. 9 we measure the performance of the algorithms as the functionality constraints become less tight. Recall from Sec. 4.1 that each non-generic agent group having the same parent is assumed to require the same functionality resource. Assuming ng is the group size (5 in our case) then $(1 + \beta)ng$ nodes are assumed to provide such functionality. In the x-axis of Fig. 9 we vary the constant β by 50%,

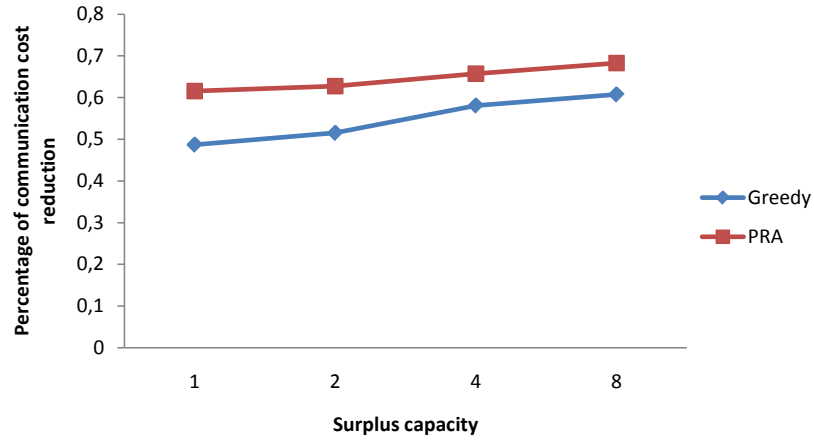


Fig. 8 Performance of the algorithms against increased node capacity

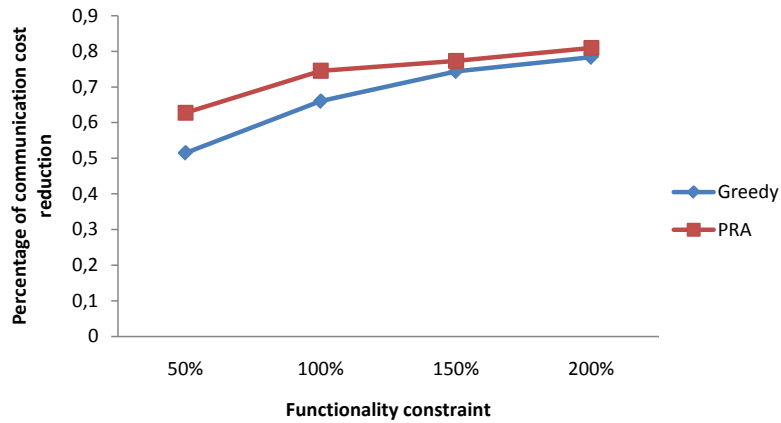


Fig. 9 Performance of the algorithms against looser functionality constraints

100%, 150% and 200% essentially increasing the number of possible hosts (functionality wise) from 5 to 7.5, 10, 12.5 and 15. As expected with more candidate locations available for each agent, there is an increased optimization potential compared to the random initial placement. Both PRA and Greedy exploit this potential resulting in a performance increase (PRA achieves roughly 80% savings by the end of the plot). Again PRA outperforms Greedy with their difference becoming small

in the 150% and 200% case. In a sense, this result means that as the nodes of the system become more homogeneous, Greedy might be a viable alternative, whereas for heterogeneous networks PRA is a clear winner.

4.4 Discussion

Summarizing the experiments we can state the following: (i) judging from the optimization margin left by the initial placement, any random solution to APR will probably be particularly inefficient; (ii) PRA achieves performance close to optimal particularly if the computational capacity constraint is not very tight; and (iii) simpler algorithms based on a pure greedy paradigm cannot achieve equivalent performance compared to PRA particularly, in networks with a heterogeneity degree as is usually the case in a smart home environment. Before proceeding with the conclusions we would like to mention that the increased performance offered by PRA does not involve a prohibitive runtime cost. All the experiments were run in an ordinary laptop carrying an Intel Pentium Dual CPU T3200 processor at 2GHz with 3GB of memory. Even in the larger setup of Sec. 4.3 the running time of PRA never exceeded a couple of seconds.

5 Conclusions

In this work we tackled the APR problem by iteratively solving it for node pairs. To do so we illustrated a graph coloring problem transformation and proposed an algorithm (AXA) to derive a solution for the equivalent problem. Through simulation experiments the final algorithmic scheme (PRA) was found to outperform a simpler greedy approach, while achieving the optimal solution in many cases. Due to the targeted environment, i.e., smart homes, we considered the case of centralized execution.

Nevertheless, our core contribution (AXA) is distributed in nature involving only a node pair. As part of our future work we plan to investigate adaptations to the centralized pairing mechanism (PRA) that will allow the algorithm to execute in a fully distributed manner.

Acknowledgment

This work is funded by the 7th Framework Program of the European Community, project POBICOS - Platform for Opportunistic Behavior in Incompletely Specified, Heterogeneous Object Communities, contract nr. FP7-ICT-223984.

Nikos Tziritas is partially supported by the Alexander S. Onassis Public Benefit Foundation in Greece.

References

1. Shea R. Boulis A., Han C.-C. and Srivastava M.B. Sensorware: Programming sensor networks beyond code update and querying. *Pervasive and Mobile Computing Journal*, 3(4), 2007.
2. Pruszkowski A. Golanski M. Domaszewicz J., Roj M. and Kacperski K. Rovers: Pervasive computing platform for heterogeneous sensor-actuator networks. In *In Proceedings of WoW-MoM*, 2006.
3. Roman G.C. Fok C.L. and Lu C. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *In Proceedings of ICDCS 2005*, 2005.
4. Xu G. Saxena A. Kremer U. Kang P., Borcea C. and Iftode L. Smart messages: A distributed computing platform for networks of embedded systems. *The Computer Journal, British Computer Society, Oxford University Press*, 47(4), 2004.
5. Pferschy U. Kellerer H. and Pisinger D. *Knapsack Problems*. Springer, 2004.
6. Millstein T. Kothari N., Gummadi R. and Govindan R. Reliable and efficient programming abstractions for wireless sensor networks. In *In Proceedings of PLDI 2007*, 2007.
7. Mottola L. and Picco G.P. Programming wireless sensor networks: Fundamental concepts and state of the art. In *to appear in ACM Computing Surveys*.
8. Oikonomou K. Stavrakakis I. Laoutaris N., Smaragdakis G. and Bestavros A. Distributed placement of service facilities in large-scale networks. In *In Proceedings of INFOCOM 2007*, 2007.
9. Walsh K. Barr R. Liu H., Roeder T. and Sireer E.G. Design and implementation of a single system image operating system for ad hoc networks. In *In Proceedings of MOBISYS 2005*, 2005.
10. Levis P. and Culler D. Mate: A tiny virtual machine for sensor networks. In *In Proceedings of ASPLOS 2002*, 2002.
11. Wolenetz M. Cooper B. Agarwalla B. Shin J. Hutto P. Ramachandran U., Kumar R. and Paul A. Dynamic data fusion for future sensor networks. *ACM Transactions on Sensor Networks*, 2(3), 2006.
12. Munagala K. Srivastava U. and Widom J. Operator placement for in-network stream query processing. In *In Proceedings of PODS 2005*, 2005.
13. STREP/FP7-ICT: Platform for opportunistic behaviour in incompletely specified, heterogeneous object communities (pobicos). In <http://www.ict-pobicos.eu/index.htm>.
14. Lalis S. Tziritas N., Loukopoulos T. and Lampsas P. Agent placement in wireless embedded systems: Memory space and energy optimizations. In *In Proceedings of IPDPS 2010, PME0-UCNS Workshop*, 2010.
15. Lalis S. Tziritas N., Loukopoulos T. and Lampsas P. Gral: A grouping algorithm to optimize application placement in wireless embedded systems. In *In Proceedings of IPDPS 2011*, 2011.
16. Özsü M. T. Yang X., Lim H. B. and Tan K. L. In-network execution of monitoring queries in sensor networks. In *In Proceedings of SIGMOD 2007*, 2007.
17. Towsley D. Ying L., Liu Z. and Xia C.H. Distributed operator placement and data caching in large scale sensor networks. In *In Proceedings of INFOCOM 2008*, 2008.