# Introducing Agent Evictions to Improve Application Placement in Wireless Distributed Systems

Nikos Tziritas[1,6], Petros Lampsas[2,6], Spyros Lalis[3,6], Thanasis Loukopoulos[2], Samee Ullah Khan[1,4], Cheng-Zhong Xu[1,5]

[1]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen, 518067, China
{nikolaos, cz.xu}@siat.ac.cn

[2]Dept. of Informatics and Computer Technology, Technological Educational Institute of Lamia, Lamia, Greece
{plam, luke}@teilam.gr

[3]Computer and Communication Engineering Dept., University of Thessaly, Volos, Greece
lalis@inf.uth.gr

[4]Department of Electrical and Computer Engineering, North Dakota State University, Fargo, ND 58108-6050, USA
samee.khan@ndsu.edu

[5]Department of Electrical and Computer Engineering, Wayne State University, Detroit, MI, 48202, USA
czxu@wayne.edu

[6]Center for Research and Technology, Institute of Thessaly, Volos, Greece

***Abstract*—With the development of mobile code frameworks for embedded systems, an application can be structured as a set of cooperating components (agents) that are placed on the nodes of the system in a flexible way. Reducing the network traffic caused by the application components is a crucial issue for the increase in the lifetime of wireless embedded systems, since it is widely known that the communication cost plays the most significant role in the energy consumption of embedded devices. To this end, most placement algorithms place or move an agent towards the center of gravity of the communication workload. However, if the target node does not have enough capacity, the attempt is usually aborted. In this paper, we introduce eviction-enabled algorithms that allow nodes to free capacity by forcing a locally hosted agent to move to another node, even at a loss, to accept a new and potentially more beneficial agent. To the best of our knowledge, this is the first time that agents are evicted by local hosts to enable beneficial agent migrations and eventually improve the total network cost. In this paper, we provide algorithms tackling the aforementioned problem in a fully distributed manner. We also present and discuss the results of extensive simulations, showing that eviction-enabled algorithms can outperform their counterparts by up to 300%.**

***Keywords - distributed systems; placement algorithms; mobile agents; wireless sensor networks***

## I. INTRODUCTION

Mobile code technologies for embedded systems, like Agilla [1], SmartMessages [3], MagnetOS [8], DFuse [7] and POBICOS [12], allow the programmer to structure an application as a set of mobile components (also referred to as agents) that can be placed on different nodes of a wireless sensor and actuator network. From a system perspective, one of the key challenges is to optimize the placement of the application taking into account the communication between the components to minimize the load of the wireless network. Given that wireless communication typically consumes (far) more energy than local processing, reducing the network load is especially important on battery-powered nodes because of the increase in the lifetime of nodes.

To reduce the amount of traffic between nodes, placement algorithms usually try to move a component (agent) towards the center of gravity of the communication load ([7], [8], [15]). In most cases, it is assumed that nodes have abundant hosting capacity, thus algorithms either do not take into account aforementioned constraints or simply abort the placement if the target node does not have enough capacity. However, this may lead to a bad placement of the application components (in terms of the traffic generated over the wireless network), even though a better placement is feasible.

In this paper, we address the problem by introducing so-called eviction-enabled algorithms, that consider migrations that are not beneficial on their own, but free space is used to enable additional (beneficial) migrations. Of course, the ultimate goal is to reduce the network load, so the total benefit of the migrations must be greater than the cost of the non-beneficial ones, leading in that way to an overall better placement.

Our contributions boil down to the following: **(a)** we prove that the agent migration problem is NP-hard in the case of

resource-constrained nodes; **(b)** the proposed algorithms are fully distributed and simple enough to be implemented on embedded systems with limited memory and computing power; **(c)** we provide a mechanism to stop/start, in a dynamic way, the dispatch of control messages exchanged for discovering potential destination nodes with enough free capacity to host a migrating agent. This mechanism is referred to as the *radio silence mechanism*.

The rest of the paper is structured as follows. Section II describes the problem formulation, application and system model; Section III presents the eviction-based algorithms; Section IV provides an experimental evaluation of the proposed algorithms through simulation; finally, Section V contains related work and Section VI concludes the paper.

## II. APPLICATION, SYSTEM MODEL AND PROBLEM FORMULATION

In this section we present the application and system model and proceed by giving the problem formulation.

### A. Application model

We consider applications structured as hierarchies (trees) of mobile software components, called agents, that cooperate with each other by exchanging messages. Agents at the bottom of the hierarchy interact with the physical environment by acquiring information or effecting change through the sensors and the actuators of a node, respectively. Since these agents rely on special resources to perform the task, these agents are called *node-specific*. The rest of the agents in the hierarchy implement higher-level data aggregation or processing functionality. This is done using general-purpose computing resources because these agents are called *node-neutral*.

Fig. 1 shows an indicative agent tree for a temperature control application. In this case, the application employs three node-neutral agents (the root and two aggregator agents) and five node-specific agents (two motion detector agents, two temperature sensor agents, and one agent for controlling the air-conditioner). At deployment time, system-level mechanisms are responsible for creating agents on the nodes of the network with sufficient hosting capacity and the appropriate special resources (sensors and actuators).

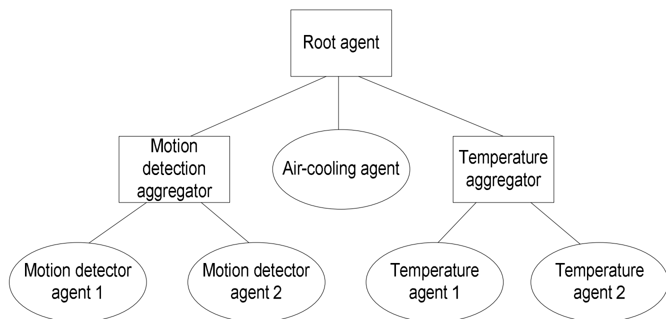


Figure 1. Tree structure of the temperature control application.

Once a node-neutral agent is created, the system may decide at runtime to move it to another node. We assume that migrations are fully transparent for the application, with the system taking care of all suspend-resume, state transfer, and message redirection issues. Also the migration step is supported only for node-neutral agents because they are node-independent by design that can be moved between nodes without violating any assumptions of the programmer.

### B. System model

The system consists of nodes with special sensing/actuating capabilities and limited storage capacity. Let $n_i$ and $c(n_i)$ denote the $i^{th}$ node and its hosting capacity, respectively. Note that the capacity of a node imposes a generic constraint to the number of both node-neutral and node-specific agents that can be hosted.

The nodes communicate with each other via short-range radio. We assume a tree-based routing structure, whereby any two nodes are connected via a single, possibly multi-hop, path. Let $r_{ij}$ denote the number of hops between $n_i$ and $n_j$. We assume that the links of the routing structure are bidirectional, thus $r_{ij}=r_{ji}$. Also, $r_{ii}=0$.

The system can host several applications, each one having its own node-neutral and node-specific agents. Let $a_k$, $s(\alpha_k)$, $h(\alpha_k)$ be the $k^{th}$ agent in the system, the size and the node hosting it, where $1 \leq k \leq NA$ and $NA+1 \leq k \leq NA+SA$ enumerates all node-neutral and all node-specific agents, respectively. An agent $\alpha_k$ may exchange messages with its relatives (parent or children) in the application tree, let these relatives be denoted by $RS_k$. Also, let $T$ be a $(NA+SA)x(NA+SA)$ matrix that encodes the communication between agents. Specifically, $T_{km}$ denotes the unidirectional traffic from $\alpha_k$ to $\alpha_m$, i.e., the number of data units $\alpha_k$ sends to $\alpha_m$ over a specific period (note that, in the general case, $T_{km} \neq T_{mk}$).

### C. Problem Formulation

The objective of this paper is to reduce the amount of wireless traffic between nodes due to the application-level communication, i.e., the messages exchanged between agents. Notably, the communication between two agents introduces traffic not only for the two nodes that host them but also for the nodes that route traffic between the hosts.

Without loss of generality, we assume the agents of an application are placed on the nodes of the system in a non-optimal way. Then, our goal is to perform a series of agent migrations to achieve a better agent placement that reduces (ideally, minimizes) the wireless network traffic.

In the sequel, we provide a proof sketch that the agent placement problem is NP-hard, by reduction to the knapsack problem. Assume a knapsack instance with $K$ objects $o_k$ where $s_k$ and $v_k$ denote the size and value of $o_k$, respectively. The knapsack problem consists of finding the collection of objects of maximum total value that fits in the knapsack of size $S$.

We can transform any such statement to an equivalent statement of the agent placement problem studied in this paper as follows. The application tree consists of the root and two more levels. In the first level, $K$ generic agents $a_k$ exist, corresponding one to one to the knapsack objects. In the second level, $K$ non-generic agents $a_{k'}$ exist, so that each generic agent

$a_k$ communicates with exactly one non-generic agent $a_{k'}$ and vice versa.

The communication cost between the root and the generic agents is set to $e$, where $e \leq \min(v_k)$, and between the generic agent $a_k$ and the non generic $a_{k'}$ is set to $v_k - e$. Two nodes exist in the network $n_1$ and $n_2$. All generic agents are initially hosted at $n_1$, while $n_2$ holds all non-generic agents together with the root agent. The size of $a_k$ is set to the corresponding knapsack's object size $s_k$, the size of the root agent is set to $1+\sum_{\forall k} s_k$, while the size of the non-generic agents can be any positive value. Finally, the capacity of $n_1$ is set to $\sum_{\forall k} s_k$, i.e., just enough to hold the generic agents hosted there, while the capacity of $n_2$ is set so that $S$ free capacity remains. In the constructed agent placement problem instance, the network load is due to the communication of agents hosted at $n_1$ with the root agent and the agents of hosted at $n_2$. The total network load of this placement is $\sum_{\forall k}(v_k - e) + \sum_{\forall k} e = \sum_{\forall k} v_k$. The only migrations that can be considered to minimize the load involve generic agents moving from $n_1$ to $n_2$ (non-generic agents cannot move and the size of the root agent is greater than the capacity of $n_1$). It is easy to see that each migration of $a_k$ from $n_1$ to $n_2$, decreases the network cost by $v_k$ and can only be done provided that the free space $S$ at $n_2$ is not covered. Thus, a solution to the aforementioned agent placement problem instance provides a solution to the initial knapsack instance.

### D. *Migration benefit/penalty and eligibility*

We focus on a distributed solution whereby each node decides locally which agents to migrate on which nodes, based on the agents' incoming and outgoing load with other agents.

Using the previous notations, the load incurred by $a_k$ when placed at $n_i$ can be expressed as follows:

$$l_i^k = \sum_{a_m \in RS_k} (T_{km} + T_{mk}) r_{ih(a_m)} \qquad (1)$$

Let $M_{ij}^k$ refer to the migration of $\alpha_k$ from $n_i$ to $n_j$. The benefit/penalty of $M_{ij}^k$, in terms of the load difference (positive or negative) of the placement obtained after $M_{ij}^k$ takes place compared to the current placement, is given by:

$$B_{ij}^k = l_j^k - l_i^k \qquad (2)$$

In order for the migration of $a_k$ from $n_i$ to $n_j$ ($M_{ij}^k$) to be eligible, $\alpha_k$ should be node-neutral and the destination node $n_j$ should have enough free capacity:

$$a_k, 1 \leq k \leq NA \qquad (3)$$

$$c(n_j) \geq s(a_k) + \sum_{m}^{NA+SA} s(a_m) \mid h(a_m) = n_j \qquad (4)$$

Each migration $M_{ij}^k$ leads to a new placement, that may incur a lower or perhaps a higher agent-level communication over the network, depending on whether $B_{ij}^k$ is positive or negative. In the former case, we refer to the migration as *beneficial* else *non-beneficial*. Not all beneficial migrations are eligible, due to the capacity constraint equ. (4).

### E. *Evictions*

To alleviate the aforementioned problem, we consider performing (possibly non-beneficial) migrations that free node capacity. We refer to such migrations as *evictions*. The idea is to exploit the capacity being released this way to perform beneficial migrations. Obviously, per definition, evictions cannot (by themselves) reduce the amount of application-level traffic over the network. To achieve such a load reduction evictions must be followed by at least one migration with a benefit that outweighs their penalty.

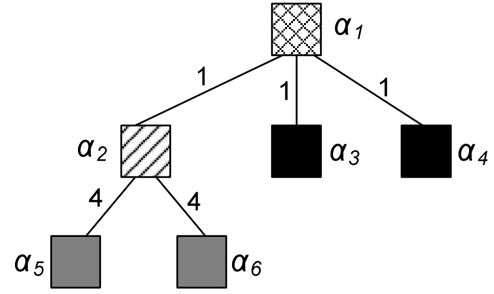


Figure 2. Application tree structure and inter-agent message traffic.

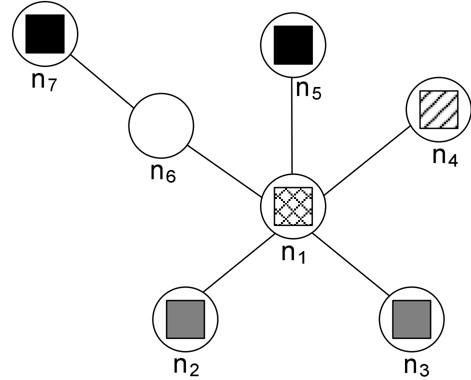


Figure 3. Placement of the application agents on the nodes of a network.

In the sequel, we give an example to illustrate this scenario. Assume the application depicted in Fig. 2, that comprises four node-specific ($a_3$, $a_4$, $a_5$, $a_6$) and two node-neutral ($a_1$, $a_2$) agents. The link weights represent the message traffic between agents (as the number of data units exchanged per time unit, e.g., bytes per second). Also assume that the application is deployed in a network of seven nodes as shown in Fig. 3, where each node has enough capacity to host only one agent.

Let us first consider node-neutral agent $a_1$. There is no better placement for it, because every migration of $a_1$ away from $n_1$ is non-beneficial as per equ. (2). Let us now consider agent $a_2$. In this case, a migration from $n_4$ to $n_1$ would yield a benefit of 9 as per equ. (2). But note that $M_{41}^{2}$ is not feasible due to the capacity constraint equ. (4) for $n_1$. However, this can be made feasible by evicting $a_1$ to $n_6$ at a penalty of 1. If both migrations are performed ( $M_{16}^{1}$ followed by $M_{41}^{2}$ ) a better placement will be obtained for the application, with a benefit of 8 vs. the current placement.

## III. HEURISTICS

In this section, we propose heuristics that consider evictions, that in turn enable a beneficial migration so that the cumulative benefit/penalty is positive.

### *A. Single Path Algorithm (SP)*

In this algorithm, each node iterates through the list of locally hosted node-neutral agents to find the one (if any) that is most beneficial to migrate to a neighboring node. Then, it sends to the respective destination a hosting request with the identifier of the agent to be migrated, the size and the benefit of the migration as per equ. (2).

When a node receives a hosting request it checks if it has enough free capacity to host the agent in question, in which case it sends a positive reply. Else, the host considers one or more evictions (in increasing order of their penalty) until enough free capacity is secured (or the cumulative penalty outweighs the benefit of the request). Then, for each such eviction, a hosting request is issued carrying the remaining benefit (used to decide for more evictions downstream). If all replies are positive and the total penalty does not exceed the benefit, a positive reply is sent back to the node that issued the hosting request.

When a node responds positively to a hosting request, it *reserves* the capacity required to host the agent in question, including the capacity (still) being used for the agents that are to be evicted. The above ensures that it will be possible to perform the respective migration, if the node that issued the hosting request decides to proceed. The aforementioned reservations are cancelled when a node receives a negative reply. Also, in the case of eviction groups, if a single reply is negative, then a cancellation message is sent to the nodes that replied positively.

Finally, to avoid races, an agent is not considered for several migration or eviction processes simultaneously. Moreover, we limit the degree of "recursive" forwarding of hosting requests via a hop limit, specified by the nodes that initiates the migration process.

The protocol execution on source node $n_s$

```
for each local node-neutral agent a_k {
  for each neighbor node n_d {
    calculate potential benefit B^k_sd
    update most beneficial migration m
  }
}

if (m.benefit > 0) {
  send (m.dst, [HostReq, m.aid, m.asize, m.benefit]);
  recv(m.dst, [HostReply, res, penalty]);
  if (res=OK) { start migration m }
}
```

The destination $n_d$ receives from $n_s$ [*HostReq*, *aid*, *asize*, *benefit*]

```
if (freeSpace > asize) {
    reserveSpace(asize);
    send (n_s, [HostReply, OK, 0]);
  }
  else {
    evict := {}; espace := 0; penalty := 0;
    do {
      for each local node-neutral agent a_k not in evict {
        for each neighbor node n_d' ≠n_s {
          calculate potential benefit B^k_sd'
          update most beneficial migration m
        }
      }
      penalty := penalty – m.benefit; // >0 for evictions
      if (penalty >= benefit) { break; }
      evict := evict + {m};
      espace := espace + m.asize;
    } while (espace + freeSpace <= asize);

    if (penalty >= benefit) {
      send (n_s, [HostReply, NOK, 0]);
    }
    else {
      reserveSpace(freeSpace + espace);
      rembenefit := benefit – penalty;
      for each m in evict {
        send (m.dst, [HostReq, m.aid, m.asize, rembenefit]);
      }
      replies := {};
      for each eviction m in evict {
        recv(m.dst, [HostReply, res, penalty2]);
        penalty := penalty + penalty2;
        replies  := replies + {res};
      }
      if (all replies are OK) and (benefit > penalty) {
        send(n_s, [HostReply, OK, penalty]);
        for each m in evict { start migration m }
      }
      else {
        send(n_s, [HostReply, NOK, 0]);
        for each m in evict { cancel reserved space }
      }
    }
  }
```

Figure 4. Pseudocode of the SP algorithm.

## B. Network Flooding Algorithm (FL)

In single path algorithm (SP) a node chooses to evict agents in increasing order of the respective penalty. However, the latter is calculated locally, without knowing what the actual penalty of such migrations will be (an eviction may lead to further evictions downstream). To address the problem, we propose an algorithm where the agent to be evicted is chosen based on the smallest "total" penalty of the action.

The main difference of FL compared to SP is that the algorithm *determines* the cost of an agent eviction by investigating all possible destinations; not just the most promising one according to local knowledge. More specifically, a so-called probe request is sent to each destination that is a candidate for hosting the agent to be evicted. When all replies arrive, the one with the greatest benefit (smallest penalty) is selected and the corresponding node is appointed as the destination for the migration/eviction in question.

The probe replies travel back the same way hosting replies do, with the difference that a reply also includes, besides the cumulative penalty, the respective eviction list. Eventually, the node that started the process (issued the probe request for the beneficial migration) receives such a reply. If this is positive, a hosting request is sent downstream, else the migration is (silently) cancelled. Unlike in SP, a hosting request specifies the evictions to be performed, therefore a node knows what agent(s) has to evict to what nodes.

Consider an application that is deployed in a network of nodes as shown in Fig. 5. Assume that each node is able to host just one agent, and that all agents are node-neutral and of the same size. Also, without going into the details of the agent-level message traffic, let the benefit/penalty of agent migrations be as listed in Table I.

TABLE I. BENEFIT/ PENALTY PER AGENT MIGRATION IN FIG. 5

| $M_{12}^{1}$ | $M_{23}^{2}$ | $M_{24}^{2}$ | $M_{35}^{3}$ | $M_{46}^{4}$ | $M_{67}^{5}$ |
|---|---|---|---|---|---|
| 20 | -7 | -2 | -1 | -5 | -5 |

Given that the only beneficial migration is that of $a_1$ from $n_1$ node to $n_2$ *node*. The node $n_1$ will send a probe request to $n_2$ with a benefit value of 20. Since the node $n_2$ does not have enough free capacity to host $a_1$, it will consider evicting $a_2$ to $n_3$ with a penalty of 7, or to $n_4$ with a penalty of 2. Since both penalties are smaller than the benefit of the probe request, in turn, $n_2$ sends a probe to both destinations, with a remaining benefit of 13 and 18, respectively. In the same spirit, when $n_3$ receives the request from $n_2$, it considers evicting $a_3$ to $n_5$ with a penalty of 1, and sends a corresponding probe request with a remaining benefit of 12. Given that the node $n_5$ has sufficient free capacity to host $a_3$, the destination node sends back a positive reply with a penalty of 0 and an empty eviction list. When the node $n_3$ receives this reply, it sends a positive reply to $n_2$ with the cumulative penalty of 1 and an updated eviction list that includes $M_{35}^{3}$. Similarly, the node $n_2$ will receive from the node $n_4$ a positive reply with a cumulative penalty of 10 and the respective eviction list $\{M_{46}^{4}, M_{67}^{5}\}$. The node $n_2$ will chose the reply with the smallest penalty, i.e., that of node $n_3$, and will reply positively to node $n_1$ with a cumulative penalty of 8 and the eviction list $\{M_{23}^{2}, M_{35}^{3}\}$. Finally, upon receipt of a positive reply, $n_1$ will issue a respective hosting request that will be propagated down the chosen path (not shown in Fig. 5). Note that in this example SP would choose to evict $a_2$ towards $n_4$ leading to an inferior placement.

Unlike in SP, an agent may be considered for eviction in the context of different requests at the same time. This process is to reduce excessive "locking conflicts" that would occur due to the flooding nature of the algorithm. More specifically, a host request can be issued for an agent that is already involved in a probe request for which no reply has been received yet. In other words, hosting requests have precedence over probe requests. However, to avoid having numerous races, that in turn may result in many failed hosting requests, a hosting request cannot concern an agent involved in another pending hosting request and a probe request cannot concern an agent involved in a pending probe or hosting request. We also note that probe replies not do guarantee capacity reservation. As a consequence a node may receive a hosting request for an agent that is no longer hosted locally (in which case it sends a negative reply).

## C. Convergence

Migrations and evictions are performed to reduce the application-level message traffic over the network. The algorithms decide for one or more evictions in the context of a beneficial migration, only if the series of migrations and evictions will reduce the total network load by at least 1. Assuming a stable communication pattern between the agents, totaling $x$ data units per time unit, at most $x$ beneficial migrations can take place. While each beneficial migration may trigger a number of evictions, this number of evictions is also bounded by the network diameter (there are no cycles). The total number of migrations is bounded and, eventually, there will be no more migrations or evictions to perform.

It is important to note that a beneficial migration as per equ. (2) is guaranteed to lead to a better placement only if agents that communicate with each other directly (in the application tree) are not allowed to change hosts concurrently. Otherwise, it would be possible to have a never ending loop of "swaps". The algorithms can be easily extended to satisfy the aforementioned constraint, e.g., by notifying the relatives of an agent before commencing with the actual migration process.

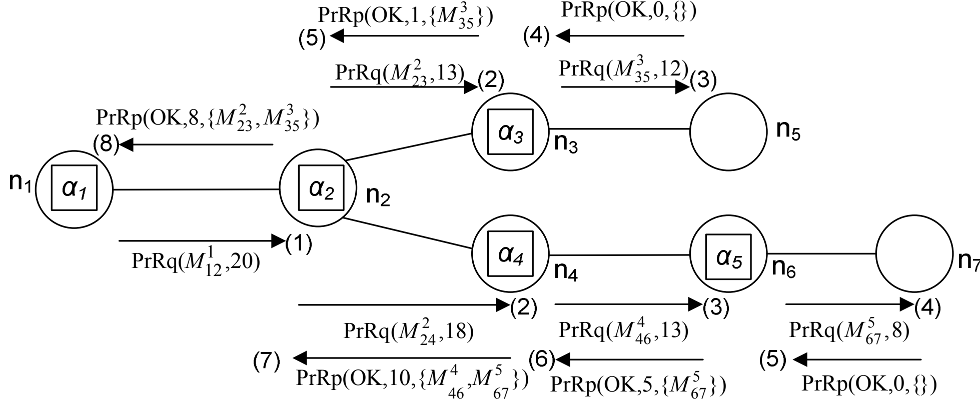


Figure 5. Example of probe request/reply generation and propagation in the FL algorithm.

### *D. Radio silence mechanism*

Both algorithms are extended with a mechanism that stops the respective protocols from generating messages ad infinitum once convergence is reached. The process works as follows: (a) each time a negative reply is sent to a node, the node is added to an *update list*; (b) when a node receives a negative reply, it adds the sender to a *block list* (blocked nodes are not considered as candidates for probe and hosting requests); (c) when a node frees capacity (due to the migration of a local agent to a remote node), it sends an *update message* to each node in the update list, and clears the list; and (d) when a node receives an update message, it removes the sender from the block list, and forwards the update to the neighbors.

Due to convergence, eventually, agent migrations will stop. The source(s) of the last migration(s) will issue update messages due to the hosting capacity freed locally, triggering the generation of host/probe requests at other nodes. But given that convergence has been reached, no more migrations can be decided. Therefore, each node from which a hosting/probe request originated will receive a negative reply, and will suppress the generation of new requests due to the blocking policy. When the final communication phase is over, there are no nodes that can generate any new update messages or hosting/probe requests, the radio silence is achieved.

## IV. EVALUATION

To study the performance of the proposed algorithms, we conducted experiments using NS-2 [17]. The details of the simulation setup and the results of the experiments are discussed in the next subsections.

### *A. Setup*

*Network generation.* Two classes of networks are used, with 50 and 20 nodes placed randomly in a plane of 120×120 and 80×80 distance units, respectively. In both cases, two nodes are assumed to be in range of each other if the Euclidean distance is less than 30 units. The corresponding tree-based routing topology is obtained by constructing a spanning tree. Five different topologies are generated for each network class.

*Application generation.* The application tree structure is also generated randomly, based on an initial (given) number of node-specific agents. The agents are divided into disjoint groups of 5, and for each group 2-5 agents are randomly picked as children of a new node-neutral agent. In subsequent iterations, orphan (node-specific and node-neutral) agents are (again) randomly split in groups of 5, and the process of parent creation is repeated, until a single agent becomes the root of the application. Three different application structures are generated this way, with (50, 22), (25, 12) and (10, 5) (node-specific, node-neutral) agents, referred to as app-50, app-25 and app-10, respectively. The agents are assigned sizes randomly, from 100 to 1000 units. The initial placement of agents on the network is random, and nodes are initialized to have *exactly* the storage capacity required to host the agents that have been assigned.

*Application traffic.* Each node-specific agent is randomly assigned a communication profile, sending 1 to 5 data values per time unit towards the parent. For node-neutral agents we consider two cases: (i) the agent sends to the parent the average of the values received from the children (digester), and (ii) the agent sends to the parent the sum of the data values received from the children (forwarder). In the experiments, we employ a 50-50 mix, each node-neutral agent being randomly assigned one of the two modes. The application-level message traffic pattern remains stable to allow the algorithms to converge.

### *B. Reference algorithms*

As a reference for the results achieved by SP and FL, we run the ILA algorithm [13]. ILA chooses to perform *only* beneficial migrations, in the same way a beneficial migration is decided in the SP and FL algorithms. Information about the free capacity of neighboring nodes is acquired in a lazy fashion, through the replies received in response to migration requests (initially, all neighbors are assumed to have full nominal capacity free). ILA does not have a mechanism for notifying nodes when capacity is freed. Instead, with a certain probability (0.2 in our experiments) each neighboring node is optimistically assumed to have enough free capacity. The best candidate, as per equ. (2), is contacted to check whether it can actually host the agent in question. As a consequence ILA

never achieves radio silence; even though it is guaranteed to converge, i.e., stop performing migrations. In our simulations, we stop running ILA when no migration is accomplished by any node in four consequent iterations.

We also employ an exhaustive algorithm that computes the best placement, by starting from an unoccupied network and trying out all possible combinations of agents on nodes. However, the placement obtained this way may not be actually feasible, because it may be impossible to reach from the initial placement by performing a series of eligible agent migrations and evictions, due to the capacity constraint as present in equ. (4). Thus the corresponding network cost represents a lower bound on what could be achieved even by an optimal algorithm.

### C. *Experiments*

In a first set of experiments, we compare the placements obtained for the 20-node networks and one app-10 application, the initial hosting capacity of the nodes increases to 1-4 times the average agent size in the system. We report the average results for the five different network topologies and five different initial placements for each topology (i.e., 25 runs). No large variances were recorded.

Fig. 6 illustrates the load reduction vs. the initial placement achieved by the algorithms. As it can be inferred by the trends, both SP and FL achieve a significant reduction of the network load. The improvement over ILA is roughly 30-20% when nodes have a rather modest amount of free capacity. Moreover, when the extra free capacity is (just) 2 times the average agent size, SP and FL algorithms perform close to the exhaustive algorithm, which is merely 10% better; a very positive sign as to their effectiveness. When nodes have considerable free capacity, SP, FL and ILA achieve practically equally good placements, a trend observed throughout all our experiments. This is natural since the probability of a node becoming the bottleneck for beneficial migrations drops with increasing free capacity therefore, good placements can be reached without (any) agent evictions.

For the next experiments, we run the algorithms in the 50-node networks where a mix of fifteen applications (five app-50, five app-25 and five app-10 applications) were deployed. This time we increase the free space of each node by 2, 5, 10, 20, 40 and 80 times the average agent size. We do not run the exhaustive algorithm due to its prohibitive running time complexity.

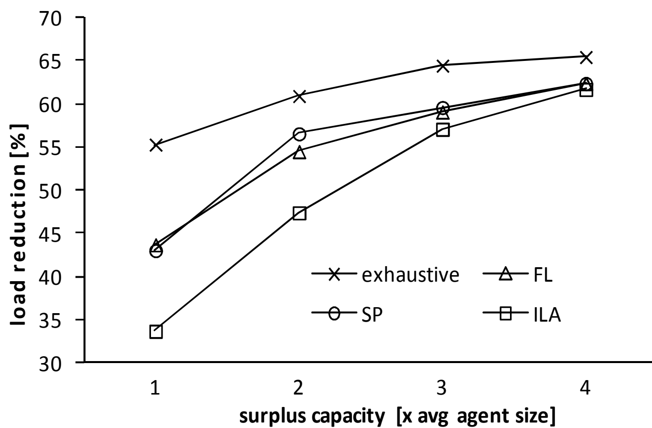


Figure 6. Load reduction vs. additional capacity (20 nodes, app-10).

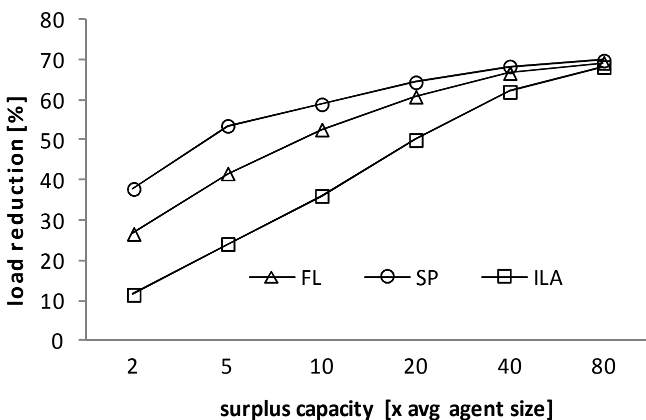


Figure 7. Load reduction vs. additional capacity (50 nodes, app-mix).

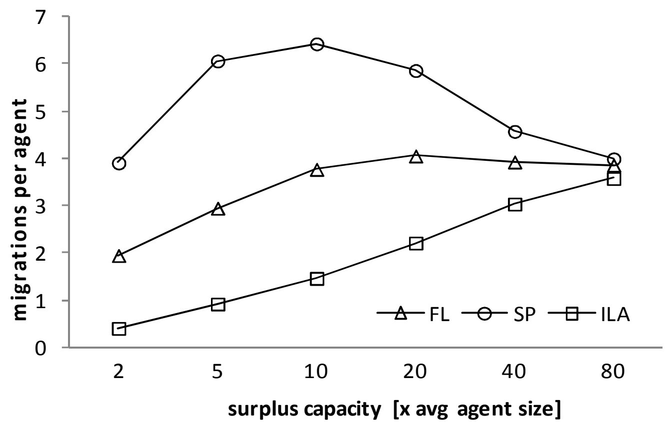


Figure 8. Migrations vs. additional capacity (50 nodes, app-mix).

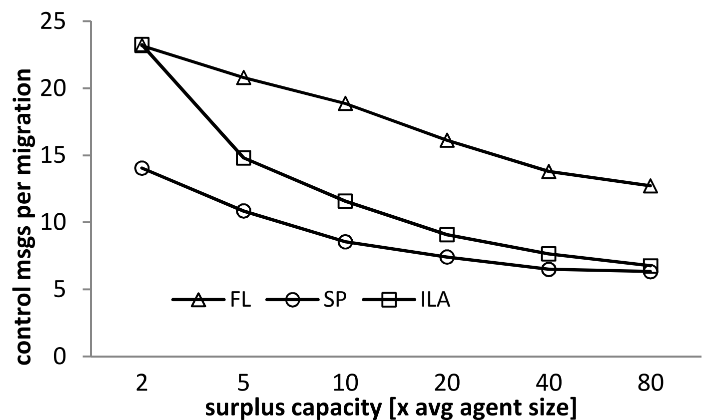


Figure 9. Control msgs vs. additional capacity (50 nodes, app-mix).

The trend in Fig. 7, is similar to the one that was observed in the small-scale experiment. However, the improvement of SP and FL vs. ILA becomes impressive, ranges between 320% and 220%, when the hosting capacity of nodes is limited. SP performs better placements than FL. In fact, when capacity is tight SP produces placements that are almost 1.5 times better compared to FL, that in turn produces placements close to 2.5 times as efficient compared to the ones produced by ILA.

The inferiority of FL vs. SP is attributed to the contention introduced by the flooding mechanism. In a large-scale system, it is very likely that several migrations and evictions will be attempted concurrently, that in turn leads to a large number of conflicts, where beneficial migrations are hindered by less beneficial ones (including evictions). Moreover, given that each such conflict leads to the generation of negative replies, the radio silence mechanism may be activated prematurely, missing opportunities for migrations/evictions.

The ability of SP to perform a larger number of migrations (and evictions) than FL is clearly shown in Fig. 8, which plots the number of migrations/evictions performed per agent in the system. The difference between SP and FL is more pronounced when capacity is tight, which is also the case when SP performs notably better than FL. As free node capacity increases, the number of beneficial migrations that can be performed without having to do any evictions grows, thus all algorithms perform a comparable number of migrations (and SP starts performing fewer migrations in total as the number of evictions drop). ILA performs the smallest number of migrations, by far when free capacity is scarce, because it does not perform any evictions.

We also measure the number of so-called control messages generated by FL, SP and ILA to decide about migrations (and evictions). Fig. 9 shows the ratio of control messages to the number of migrations performed. Clearly, SP is more efficient than both FL and ILA, especially when nodes have little free capacity. The greater per-migration protocol overhead of FL is partly due to the fact that it performs fewer migrations than SP. Furthermore, for each beneficial migration, FL algorithm floods the network with probe requests and replies to find the best possible series of evictions, whereas SP algorithm picks a single path.

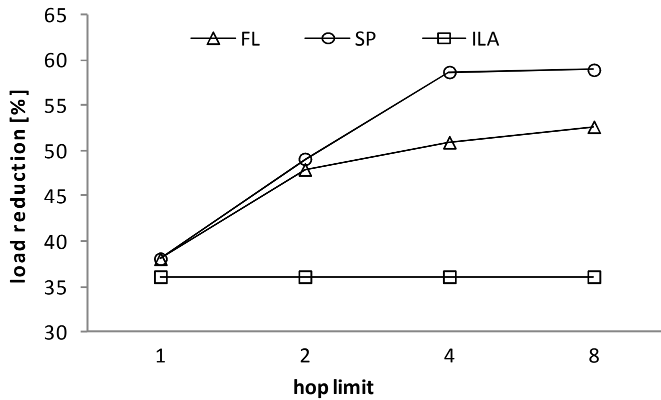


Figure 10. Load reduction vs. hop limit (50 nodes, cap +10, app-mix).

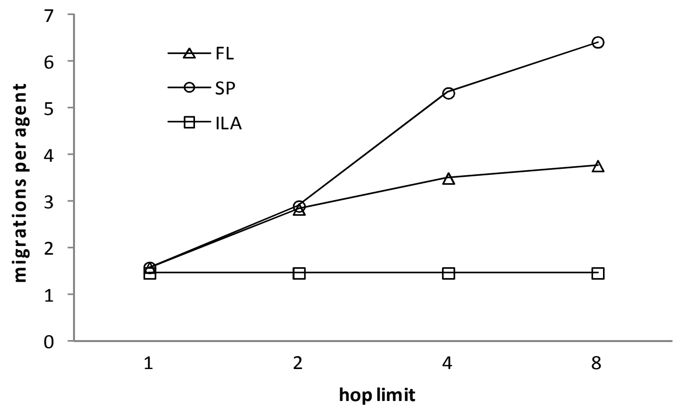


Figure 11. Migrations vs. hop limit (50 nodes, cap +10, app-mix).

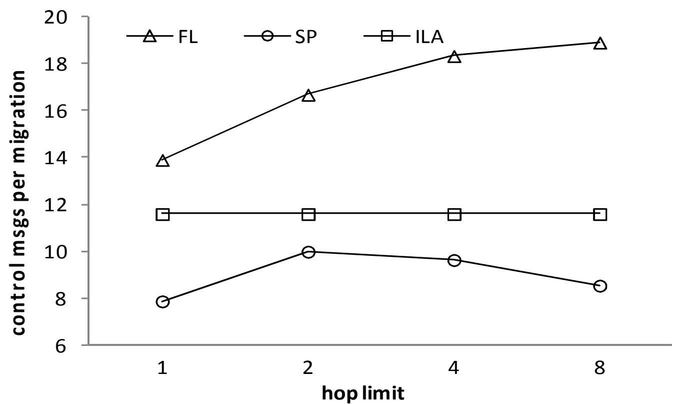


Figure 12. Control msgs vs. hop limit (50 nodes, cap +10, app-mix).

The high per-migration overhead of ILA is also due to the fewer migrations accomplished compared to SP and FL algorithms. This overhead is clearly visible when free capacity is tight. However, ILA continues to exhibit a notable overhead even when nodes have abundant free capacity and the number of migrations performed is close to that of SP and FL algorithms. The reason is that even if a node is found occupied, ILA will still consider it (with 0.2 probability) as a destination for a beneficial agent migration. As a result of contacting nodes in this optimistic way, the number of unsuccessful migration attempts remains high.

In a final set of experiments, we measure the impact of limiting the hops of hosting and probe requests in SP and FL algorithms. We use the 50-node networks and application mix of the previous experiments, while fixing the extra free node capacity at 10 times the average agent size in the system. The load reduction achieved, the number of migrations per agent and the number of control messages per migration are depicted in Figs. 10, 11 and 12, respectively, with the hop limit varying from 1 to 8 units. The behavior of ILA is not affected by this parameter (the algorithm only issues 1-hop requests for beneficial migrations).

Both SP and FL algorithms exhibit a similar performance for small hop limits. As the hop limit increases, SP algorithm

clearly outperforms FL algorithm, due to the growing negative effects of flooding (and contention). It is interesting to observe that the load reduction achieved by SP flattens at 4 hops being practically identical to the reduction achieved at 8 hops, despite the larger number of migrations (and evictions) performed in the latter case. This is attributed to the fact that, from a certain point onwards, additional evictions do not lead to a significantly better application placement. More specifically, the average diameter of the 50-node networks used in our simulations is 10. Moreover, a hop limit of 4 is already sufficient for a node that is not located at the periphery of the network to reach almost all other nodes (requests issued by that node can cover an area with a diameter of 8). The protocol overhead of SP starts dropping at 4 hops and this trend continues at 8 hops. The reason is that there are fewer opportunities to perform migrations (and evictions) when the hop limit is small, while the protocol overhead is amortized as the number of migrations grows at larger hop limits. On the other hand, the per-migration overhead of FL algorithm increases steadily due to the scalability problems of the flooding approach.

### *D. Result summary*

Both SP and FL algorithms produce significantly better placements than ILA algorithm when nodes have limited hosting capacity. Also, SP consistently outperforms FL, not only in the placement achieved but also in the per-migration protocol overhead.

## V. RELATED WORK

Placement problems have been tackled in the past under various contexts, e.g., file allocation and task allocation (see [4], [6] and [9], to name a few). The above works differ from ours either in the network and application structure assumed as well as in scope. In the context of WSNs, Tian et al. [10] discusses task allocation with the aim to minimize energy consumption. The application model assumed is different from our approach. Moreover, the authors assume that an application is executed within a sphere of a single-hop cluster. In [16], the authors also consider agent migrations but with the aim of defining optimal paths to collect data.

Mobile code based systems are subsumed in the general category of systems that afford programming abstractions for WSNs (an overview is given in [5]). Such systems provide a programming environment and/or middleware for pervasive applications. One distinction between such systems could be made based on whether migrations are performed explicitly by the programmer, or as a result of a decision of the middleware to optimize some predefined cost function (e.g. network load). Agilla [1], Smart Messages [3] and SensorWare [2] are indicative systems that belong in the former category, whereas MagnetOS [8], and DFuse [7] in the latter.

In the context of POBICOS [12] we have designed and implemented mechanisms for application placement on appropriate resource-constrained networked nodes [13], [11]. We have also implemented and evaluated distributed algorithms for the dynamic migration of agents (either a single agent or an application subtree), in a system of nodes with the objective of reducing the network load due to agent-level communication [14] [15]. To the best of our knowledge, the approach described in this paper is the first attempt to introduce agent evictions to improve application placement for wireless distributed systems.

## VI. CONCLUSIONS

We have described distributed algorithms for migrating agents between the nodes of a wireless embedded system to reduce application-level network traffic. Our approach introduces migrations that may be non-beneficial on their own, but free space to enable beneficial migrations, that eventually leads to an overall better application placement on nodes. We also presented and discussed the results of extensive simulations, showing that the proposed approach outperforms solutions based solely on beneficial migrations, resulting in placements that reduce network traffic significantly.

## ACKNOWLEDGMENTS

This work has been funded in part by the FP7/ICT Program of the European Union, under project POBICOS – Platform for Opportunistic Behaviour in Incompletely Specified, Heterogeneous Object Communities, contract Nr. FP7-ICT-223984.

Samee U. Khan’s work was partly supported by the Young International Scientist Fellowship of the Chinese Academy of Sciences, (Grant No. 2011Y2GA01).

Nikos Tziritas’s work was partly supported by the Postdoctoral Fellowship of the Chinese Academy of Sciences.

## REFERENCES

[1] L. Fok, G. Roman, and C. Lu, “Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications,” in Proc. ICDCS 2005.

[2] A. Boulis, C.-C. Han, R. Shea, and M.B. Srivastava, “SensorWare: Programming sensor networks beyond code update and querying,” Pervasive and Mobile Computing Journal, 3(4), 2007, Elsevier.

[3] P. Kang, C. Borcea, G. Xu, A. Saxena, U. Kremer, and L. Iftode, “Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems,” The Computer Journal, 47(4), 2004, Oxford.

[4] V.M. Lo, “Heuristic Algorithms for Task Assignment in Distributed Systems,” IEEE Transactions on Computers, 31 (11), pp. 1384-1397, 1988.

[5] L. Mottola, and G.P. Picco, “Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art,” ACM Computing Surveys 43(3), 2011.

[6] D. Gu, F. Drews, and L.R. Welch, “Robust task allocation for dynamic distributed real-time systems subject to multiple environmental parameters,” in Proc. ICDCS 2005.

[7] U. Ramachandran, R. Kumar, M. Wolenetz, B. Cooper, B. Agarwalla, J. Shin, P. Hutto, and A. Paul, “Dynamic Data Fusion for Future Sensor Networks,” ACM Transactions on Sensor Networks, 2(3), 2006.

[8] H. Liu, T. Roeder, K. Walsh, R. Barr, and E.G. Sirer, “Design and Implementation of a Single System Image Operating System for Ad Hoc Networks,” in Proc. MobiSys 2005.

[9] U. Srivastava, K. Munagala, and J. Widom, “Operator Placement for In-Network Stream Query Processing,” in Proc. PODS 2005.

[10] Y. Tian, J. Boangoat, E. Ekici, and F. Özgüner, “Real-time Task Mapping and Scheduling for Collaborative in-Network Processing in DVS-enabled Wireless Sensor Networks,” in Proc. IPDPS 2006.

[11] N. Tziritas, "Algorithms and System-level Support for Agent Placement and Migration in Wireless Sensor Networks", Ph.D. Dissertation, Univ. of Thessally, Volos, Greece, 2011.

[12] N. Tziritas, G. Georgakoudis, S. Lalis, T. Paczesny, J. Domaszewicz, P. Lampsas, T. Loukopoulos, "Middleware Mechanisms for Agent Mobility in Wireless Sensor Networks", In Proc. S-CUBE 2012.

[13] N. Tziritas, T. Loukopoulos, S. Lalis, and P. Lampsas, "On Deploying Tree Structured Agent Applications in Networked Embedded Systems," in Proc. EUROPAR 2010.

[14] N. Tziritas, T. Loukopoulos, S. Lalis, and P. Lampsas, "Agent Placement in Wireless Embedded Systems: Memory Space and Energy Optimizations," in Journal of Simulation Modeling Practice and Theory 19(6):1445-1464, Elsevier, 2011.

[15] N. Tziritas, T. Loukopoulos, S. Lalis, and P. Lampsas, "GRAL: A Grouping Algorithm to Optimize Application Placement in Wireless Embedded Systems," in Proc. IPDPS 2011.

[16] Q. Wu, N. Rao, J. Barhen, S. Iyenger, V. Vaishnavi, H. Qi, and K. Chakrabarty, "On Computing Mobile Agent Routes for Data Fusion in Distributed Sensor Networks," IEEE Transactions on Knowledge and Data Engineering, 16(6), 2004.

[17] Network Simulator 2 (ns2), http://www.isi.edu/nsnam/ns/