Coordination Strategies for Agent Migrations in Wireless Sensor Networks

Nikos Tziritas[1], Thanasis Loukopoulos[2], Spyros Lalis[2], Samee Ullah Khan[3], Cheng-Zhong Xu[1,4]
[1]Chinese Academy of Sciences, China, {nikolaos, cz.xu}@siat.ac.cn
[2]University of Thessaly, Greece, luke@dib.uth.gr, lalis@inf.uth.gr
[3]North Dakota State University, USA, samee.khan@ndsu.edu
[4]Wayne State University, USA, czxu@wayne.edu

Abstract— **Agent-based middleware platforms for wireless sensor networks (WSNs) have received a lot of attention during the last years, due to their great flexibility in re-programming, monitoring, handling, and optimizing the application as well as the whole system. Even though many algorithms have been proposed for the dynamic placement of agents within the WSN, they do not take into account the coordination aspects of such migrations. This not only may result in slow convergence but also in perpetual oscillations of agent migrations, degrading application and system performance. In this paper, we propose full-coordination, semi-coordination, and non-coordination agent migration strategies, and evaluate their convergence and network overhead. We also provide proofs that convergence is guaranteed when dynamic agent placement algorithms adopt our proposed strategies. Our results show that the semi-coordination strategy is superior in terms of both network overhead and convergence rate.**

Keywords- ***agent placement, agent migration, network overhead minimization, application deployment***

I. INTRODUCTION

Wireless sensor and actuator networks (WSNs) have become a significant area for researchers and industry. On the one hand, researchers focus on adjusting existing protocols or developing new ones, taking into account the limited computing, storage, communication and energy resources of sensor/actuator nodes. On the other hand, industry has many reasons to be involved in such a field since domains as military, home automation, forest protection need tiny sensors/actuators, rendering sensor networks a lucrative business. Also, companies and research centers are often funded under the context of a project to advance such technologies and to unify the research sector with that of industrial.

POBICOS [19] is such a project (among others [10], [11], [16]) with the purpose of constructing an agent-based platform for opportunistic applications on top of wireless sensor and actuator networks. The concept of POBICOS is to provide a service for hosting third party applications. An application is built as a hierarchy of communicating agents. However, this is done without making any hard assumptions about the network (number/topology of nodes and available sensors/actuators) where the application will be deployed. Thus, the application agent tree can vary in different deployments, depending on the WSN configuration.

Such a work is not far from being established in various domains in the near future. However, the owner of such a system should not be concerned in replacing or replenishing the battery of any object in the system; so we focus on making such a service as long-living as possible by reducing the overall communication cost and consequently to increase the service's lifetime. As we will discuss in the following section, we have two kinds of agents the mobile and the immobile ones. Lifetime optimization involves migrating the movable agents so as to reduce the communication overhead. Network overhead reduction is of paramount importance in WSNs, since it plays a pivotal role in energy consumption [15]. However, a question that arises is how frequently one or more agents will need to migrate; this depends completely on the environment the application is running on. In case of an environment where the sensing/actuating objects change location in a rapid fashion, the migration of the agents will become a frequent process. Also, if the sending rate of the agents changes at a quick pace, then more migrations will be required. In our work we evaluate a number of strategies as to how fast they converge, how quickly they react to a change, and to what extent they reduce the communication cost when such a change occurs.

In this paper, we discuss three distributed strategies to evaluate the network overhead, agent migration oscillations, as well as convergence rate. The proposed strategies are also extended to consider computing capacity constraints on nodes. We also guarantee the convergence of the proposed strategies through proofs. Last we perform an experimental evaluation to compare the differences among the proposed strategies.

The paper is organized as follows: Section II discusses works relevant to ours. In Section III, we give a description of the system model as well as the problem definition. The three strategies for single agent migrations are detailed on Section IV, with their extensions regarding group agent migrations and computing capacity constraints being discussed in Section V and Section VI, respectively. A detailed discussion about the convergence issues of the strategies is given in Section VII. The strategies are experimentally evaluated in Section VIII. Finally, in Section IX we provide concluding remarks.

II. RELATED WORK

This work is in the context of the agent-based embedded middleware POBICOS [19]. There are many other WSN middleware platforms that support code mobility and migrations [1], [2], [4], [6], [7], [8], [9], [10], [11], [16], [17]. Unfortunately, enough of them do not support automatic code placement, assigning in that way the complex task of manual code placement to the application programmer. This

is a main drawback, because there are environments where the network topology changes dynamically, rendering these systems completely unsuitable to those environments. On the other extreme, some of the aforementioned platforms [10], [11], [16] support dynamic agent placement.

In [20], [22] and [23] we address the agent placement problem in WSNs. The decisions are taken in a distributed and autonomous way, without coordination when performing agent migrations. The agent placement problem can also be thought of as the operator placement problem in in-network processing. Operators process data that finally reach a sink node within the system. Operators play the role of generic-agents, while sinks represent the non-generic agents. In [13], the objective of operator tree placement is to minimize the total communication overhead. The problem is solved in a distributed way in an uncoordinated fashion. The authors in [5] proposed an optimal fully distributed solution for the in-network processing problem. Their solution is based on Fermat node and it concerns only one operator and therefore there cannot be guaranteed optimality in case of multiple operators. The aforementioned problem is also tackled by [12] in an optimal way for multiple operators.

Similar problems are studied in the field of virtual machine placement, where instead of agents one deals with virtual machines (VMs). For instance, [14], [3], [24] tackle the problem of VM placement taking into account VM communicational dependencies. The problem is solved in a centralized way. On the other extreme, the aforementioned problem is solved in a fully distributed way in [21] and [18].

All of the aforementioned works perform migrations in an uncoordinated fashion, leading in that way to oscillations as well as increased network overhead. To the best of our knowledge there is no work in the literature proposing different fully and semi-coordinated schemes and evaluating them versus the uncoordinated ones.

III. SYSTEM MODEL AND PROBLEM DEFINITION

A. System Model

In this paper we investigate the problem of finding a placement scheme for each application in order to minimize the total network overhead. We assume that the agents of each application are strictly application-dependent and so they do not communicate with agents of other applications. Agents of an application form a communication tree with the root agent lying at the top of the tree. Therefore each agent has one parent and a number of children except the root agent and the leaf-agents where the former has not a parent and the latter have no children. Here, we introduce the concept of generic and non-generic agents. Generic agent is a piece of code that is not dependent on node's characteristics and is able to migrate in any node of a given network. The idea of a generic agent is to act as an aggregator to collect data from each own child and in the sequel to apply a compression function on these data. On the other side, a non-generic agent is dependent on node's capabilities, thus, is hardly movable and will be considered immobile for convenience. From this point onwards, we will use the term *nodes* and *sensors/actuators* interchangeably in the paper.

Fig. 1 shows an indicative agent tree for a temperature control application. In this case, the application employs three generic agents (the root and two aggregator agents) and five non-generic 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).

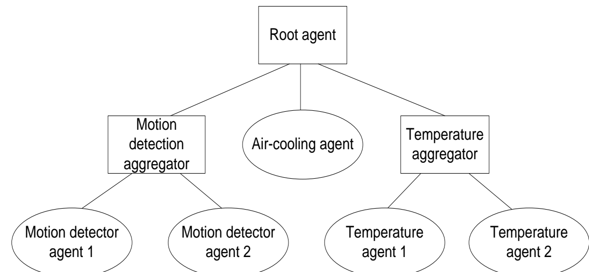

Figure 1. Tree structure of the temperature control application.

Once a generic agent is created, the system may decide at runtime to move it to another node. This can only be done for generic agents since they are the only ones that are mobile. 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.

The objective of this paper is to provide algorithms that aim at reducing the communication cost of applications. The whole idea is to migrate each agent (group of agents) as close to its children as possible. This can be done: (i) by coordinating all of the agents in a top-down fashion (full-coordination strategy); (ii) by having each group of agents acting autonomously while coordinating specific group members (semi-coordination strategy); and (iii) by having each agent acting autonomously (uncoordinated strategy). The three migration strategies are detailed further down.

B. Problem Definition

The variables n_i, a_k denote the i-th node and the k-th agent, respectively. Let Parent(a_k) and Host(a_k) denote the parent and the host of k-th agent, respectively. The agent assignment is captured by $N \times A$ matrix, denoted by f, with N denoting the number of nodes and A the number of agents. Specifically, if n_i hosts a_k then f_{ik} equals one, otherwise equals zero. The variable $l(a_{k,} a_m)$ specifies the communication load between k-th and m-th agent. On the other hand, $l_i(a_k)$ denotes the communication overhead exchanged between n_i and Host(a_k) for the communication between a_k and its parent/children. Let h_{ij} be the distance between i-th and j-th node measured in hops. The amount of computing requirements of an agent a_k is denoted by $c(a_k)$, while the total computing capacity of a node n_j is signified by $c(n_j)$.

A migration of a_k from n_i to n_j incurs a network cost proportional to h_{ij} and the size of a_k (let $size(a_k)$). The objective function for minimizing the communication

overhead is given by (1), while the one for minimizing the network overhead due to migrations is defined by (2). It must be noted that in (2) the total time for the transition of an old assignment scheme towards a new one is split into T slices, such that each slice represents the maximum possible time for performing an agent migration. Note also that $f_{ik}^{(t)}$ records whether the k-th agent is hosted by i-th node at t-th slice. The fact that the agents hosted by a node must not violate its computing capacity is captured by (3).

$$comm(F) = \sum_{\forall i}^{N} \sum_{\forall j}^{N} \sum_{\forall k}^{A} \sum_{\forall m}^{A} l(a_k, a_m) f_{ik} f_{jm} h_{ij} \quad (1)$$

$$mig = \sum_{\forall t}^{T-1} \sum_{\forall i}^{N} \sum_{\forall j}^{N} \sum_{\forall k}^{A} size(a_k) f_{ik}^{(t)} f_{jk}^{(t+1)} h_{ij} \quad (2)$$

$$c(n_i) \leq \sum_{\forall k}^{A} c(a_k), \forall i \quad (3)$$

The problem is formally stated as: *Given an old assignment scheme of generic and non-generic agents and communication dependencies between them, find a new assignment by migrating the generic agents so that the total network overhead and the agent migration oscillations are minimized, while the transition from the old scheme towards the new one is realized at the minimum possible time.*

IV. Strategies for Single Agent Migrations

As we discussed in the previous section there are three strategies that try to solve the problem of minimizing the total network overhead. In the full-coordination strategy the migration procedure starts in a bottom-up manner which means that the process starts from the lowest level of the application tree and goes on the upper ones taking into consideration that the agents belonging to a given level are able to start the migration only if the agents of the lower level have completed their migration process. In the semi-coordination strategy each agent may start the migration process only if its children and parent are not in the process of migration. In the uncoordinated strategy each agent acts autonomously without considering the state of other agents.

The problem of finding the node that minimizes the communication load for a given agent can translate into the problem of finding the center of gravity of an agent by considering its neighboring agents as a set of components with each component corresponding to a parent-child relationship; the greater the communication load and the number of hops for a parent-child relationship, the greater the force exerted by the component corresponding to this relationship. Therefore, a generic agent migrates to the node acting as the center of gravity of agent's components.

Fig. 2 gives the pseudocode for a function *findBestHost* that finds the node acting as the center of gravity for a given agent. The function computes (lines 4-6) for each 1-hop neighboring node the load that goes through it and is associated with the agent for which we wish to find the best host. A better host is available if there exists a 1-hop neighbor having an associated load greater than the summation of the loads from all the remaining 1-hop neighbors together with the load at the node where the agent currently resides (*oldHost*).

```
1:   int findBestHost(a_k) {
2:       oldHost=Host(a_k);
3:       nodeSet={1-hop neighbors of oldHost, oldHost}
4:       for each n_i ∈ nodeSet {
5:           compute l_i(a_k);
6:       }
7:       newHost= n_i ≠ oldHost : l_i(a_k) > Σ_{x≠i}^{nodeSet} l_x(a_k);
8:       if (newHost==NULL) return oldHost;
9:       else return newHost;
10:  }
```

Figure 2. Pseudocode for identifying best host.

This criterion is justified in the following manner. By (1) the cost incurred between two agents is proportional to the hop distance of their respective hosts, as well as agent communication load. By migrating the agent towards a neighboring node (*newHost*) the agent reduces its distance (by one) towards all the agents it communicated through *newHost* and increases (by one) its distance towards all other agents, including the ones residing at *oldHost*. Therefore for the migration to be beneficial the load coming or going through *newHost* must be greater than the load coming from all other 1-hop neighbors plus the load originating from *oldHost*. In case a *newHost* satisfying the above requirement exists the function returns it, otherwise, it returns the *oldHost* since there exists no beneficial migration for the agent in question.

Before migration takes place, the node having the agent to be migrated takes action as a proxy in order to forward control messages and raw data from other agents in relationship to the agent in question. This is because the neighboring agents (in terms of application structure) are not able to keep track of the moving agent; so they keep sending control messages and raw data to the proxy node till the moving agent stabilizes to a specific node, at which point the neighboring agents are informed about the new hosting node and the proxy node is released. Notice, that the agent that moved to *newHost*, might be further migrated if deemed beneficial. Each time this happens it sends its new location to the proxy. The proxy is only released once the agent reaches its final destination, i.e., the *findBestHost* function returns the same host it currently resides.

A. Full Coordination Strategy (FCS)

In this strategy the root agent sends an initiation message to its generic children. An agent receiving the message: (i) forwards it to its own children, (ii) waits until all its children respond and (iii) once it obtains an answer from all its children initiates the migration process by computing its best hosting node. Thus, the first agent that starts computing its best hosting node is a generic agent that has no generic children. After migrating to the best hosting node or deciding that there is no need for migration, the generic agent responds to its parent with its new hosting node or with its current hosting node (in case of no migration), respectively.

This procedure iterates till the root agent is able to calculate its best hosting node. The intuition of the method is that by forcing migrations to be performed in a bottom-up fashion, whereby an agent decides once all its children have decided, a parent's migration decision is likely to be more stable, i.e., less likely to change in the future, resulting potentially to reduced convergence time. Fig. 3 presents the pseudocode of the algorithm.

```
1:  while(1) {
2:      receive(msg="start migration", addressOf(Host(Parent(this))));
3:          if(any of my children is generic){
4:              for each generic  a_i ∈ children of this {
5:                  send("start migration", Host(a_i));
6:              }
7:              for each generic a_i ∈ children of this {
8:                  receive(msg="new host", Host(a_i));
9:              }
10:     }
11:     newHost=findBestHost(a_k);
12:     if(newHost!=Host(a_k)){
13:         migrate a_k towards newHost;
14:         Host(a_k)=newHost;
15:         send("Host(a_k)", addressOf(Host(Parent(this))));
16:     }
17: }
```

Figure 3. Pseudocode for agent a_k (main thread).

It must be noted that there is also an assistant thread (its pseudocode is omitted due to space limitation) that calls periodically the *findBestHost* function to identify the best host for each hosted agent. If the function returns the current hosting node, then nothing happens. Otherwise, the hosting node sends a message to its parent. The message is iteratively forwarded from parent to parent until the message reaches the root agent. When the above happens, the "start migration" initiation message is triggered.

B. Semi-Coordination Strategy (SCS)

As we discussed earlier, in this strategy any group of agents acts autonomously without taking into account the actions of any other agent not included in the group. Any generic agent is able to form a group which consists of agents in direct relationship to the agent in question (parent or/and children). The leader of each group is the agent that both acts as parent and child. This agent is allowed to proceed on its migration if all the agents of the group are not in the process of migration. Each agent is consisted of two running threads. The first one is responsible for (i) checking if migration is needed; (ii) sending *block* messages (i.e. a message carrying a block command) to show intention that this agent is going to migrate; (iii) sending *unblock* messages (i.e. a message carrying an unblock command) to declare that either the migration procedure for this agent is done or it gives priority to another agent due to a tie; (iv) starting migration procedure. The second running thread has less power since it blocks in a receive function and waits till a *block* or an *unblock* message arrives. As discussed earlier a tie may arise when a parent and one or more children show intention for migration. We set out two approaches for breaking a tie: (i) the parent gives priority to its children; (ii) a child gives priority to its parent. The aforementioned approaches are detailed further down. Each approach uses a common function called *needMigration* which examines if an agent should migrate in order to reduce its communication overhead related to its children (see pseudocode in Fig. 4).

```
1:      boolean needMigration(a_k) {
2:          newHost=findBestHost(a_k);
3:          if (newHost!=Host(a_k)) return true;
4:          else return false;
5:      }
```

Figure 4. Pseudocode for needMigration function.

1) Approach with parent priority (SCS-PP)

In this approach the main thread works as follows (Fig. 5 gives the pseudocode). An agent forms a group consisting of its parent (if any) and its generic children (if any). It then waits until the *needMigration* function returns true, and sends *block* messages to each member of the group. When it receives all the acknowledgements, it checks if agent's parent has shown intention for migration, where two kind of ties may appear: (i) a parent-tie which means that the agent has a conflict with its parent; (ii) child-tie which means that the agent is in conflict with one or more children of its own.

```
1:      startMigration = false;
2:      ga_k={parent(a_k), generic children of a_k, a_k }
3:      while(1){
4:          while(!startMigration) {
5:              startMigration = needMigration(a_k);
6:              wait(n);
7:          }
8:          agentSet = ga_k\a_k;
9:          for each agent a_i ∈ agentSet {
10:         send(" block",Host(a_i));
11:         receive(msg, Host(a_i));
12:     }
13 :        if(blockCmd!=0){
14:         if (blockFromParent==true){
15:             send("unblock",Parent(a_i));
16 :                wait till blockFromPrarent become false;
17:             agentSet ={Parent(a_i)};
18 :                goto line 9;
19:         }else wait till blockCmd becomes 0;
20:     }
21:     oldHost = Host(a_i);
22:     Host(a_i)=findBestHost(a_i,oldHost);
23 :        agentSet =ga_k\a_k ∪ non-generic children of a_k;
24:     for each agent  a_i ∈ agentSet {
25 :            send(" unblock+Host(a_k)",Host(a_i));
26:         receive(msg, Host(a_i));
27:     }
28:     release oldHost from proxy service;
29: }
```

Figure 5. Pseudocode for agent a_k (parent priority-main thread).

In case of parent-tie the agent gives priority to its parent by sending to it an *unblock* message. Next, it blocks in a waiting function till an *unblock* message arrives from its parent. When the waiting function returns, the procedure goes back to the sending of *block* messages phase and sends a *block* message only to its parent. In child-tie case, we wait

in a function till all children have sent an *unblock* message. If both of the ties appear concurrently then we first deal with the parent-tie and then with the child-tie. When there are no ties by either a parent or a child the agent proceeds to its migration. Thereafter, it sends to each non-generic child a message about the new hosting node; also it sends to its parent and each generic child of its own a message about the new hosting node and an *unblock* command.

```
1:    while(1){
2:        receive(msg);
3:        if(msg.data==”block”) {
4:            blockCmd++;
5:            if(msg.sender==Parent(a_k)) {
6:                blockFromParent=true;
7:            }
8:            send(“ack”,msg.sender);
9:        }
10:   else {
11:       blockCmd--;
12:       if(msg.sender==Parent(a_k)) {
13:           blockFromParent=false;
14:       }
15:       update msg.newhost for msg.sender;
16:       send(“ack”,msg.sender);
17:   }
18: }
```

Figure 6. Pseudocode for agent a_k (parent priority-assistant thread).

An assistant thread (see Fig. 6) is responsible for handling *block* and *unblock* messages from agent’s children and parent. This thread blocks in a receive function till a message arrives. The message may be either a *block* message or an *unblock* message. In both cases an acknowledgement message is sent back. Each time a *block* message arrives, we increase a variable named *blockCmd* and check if the message originates from the parent agent in which case an auxiliary variable *blockFromParent* is used to represent the fact. In *unblock* message case, we decrease the variable that counts the *block* messages and check if the message was sent by the parent agent in order to write down this in the *blockFromParent* variable.

2) Approach with children priority (SCS-CP)

This approach is much the same with the aforementioned one. The difference is that the agent now gives priority to their children. Therefore, if a parent-tie appears then the main thread waits till the parent sends an *unblock* message. However, if one or more child-ties arise then the procedure is as follows: (i) it sends to each child registered in the list an *unblock* message and enters the child in a set named *agentSet*; (ii) it waits till the *blocking-list* (a list consisted of child-agents which have sent a *block* message) becomes null which means that there is no child in process of migration; (iii) it goes back to the phase where the sending of blocking messages takes place and sends a *block* message to each agent found in *agentSet*. As in previous approach the procedure iterates till there are no ties. Fig. 7 and Fig. 8 give the main and assistant thread pseudocode respectively.

```
1:    startMigration = false;
2:    ga_k={parent(a_k), generic children of a_k, a_k }
3:    while(1){
4:        while(!startMigration) {
5:            startMigration = needMigration(a_k);
6:            wait(n);
7:        }
8:        agentSet = ga_k\a_k;
9:        for each agent a_i ∈ agentSet {
10:       send(“ block command”,Host(a_i));
11:       receive(msg, Host(a_i));
12:   }
13 :      if(blockCmd!=0){
14:       agentSet={};
15:       for each agent a_i ∈blocking-list{
16:           send(“unblock command”, a_i);
17:           agentSet=agentSet ∪ a_i;
18:       }
19:       if (agentSet ≠ ∅){
20:           wait till blocking-list becomes null;
21:           goto line 9;
22:       } else wait till blockCmd becomes 0;
23:   }
24:   oldHost = Host(a_k);
25:   Host(a_k)=findBestHost(a_k, oldHost);
26 :      agentSet = ga_k\a_k ∪ non-generic children of a_k;
27:   for each agent A_i agentSet {
28:       send(“ unblock command+Host(a_k)”, Host(a_i));
29:       receive(msg, Host(a_i));
30:   }
31:   release oldHost from proxy service;
32: }
```

Figure 7. Pseudocode for agent a_k (children priority-main thread).

```
1:  while(1){
2:      receive(msg);
3:      if(msg.data==”block command”){
4:          blockCmd++;
5:          if(msg.sender is a child){
6:              add msg.sender in blocking-list;
7:          }
8:          send(“ack”,”msg.sender);
9:      }else{
10:     blockCmd--;
11:     remove msg.sender from blocking-list if it exists;
12:     send(“ack”,”msg.sender);
13: }
14:  }
```

Figure 8. Pseudocode for agent a_k (children priority-assistant thread).

C. Non-Coordination Strategy (NCS)

In non-coordination strategy, each node can act autonomously when taking decisions about migrations. However, when an agent is to be migrated, a message must be sent towards the parent of the respective agent. This message is called “swap message” and its role is to ensure that there will be no swap between agents with a parent-child relationship. This is crucial for convergence, else a race condition might apply, where two agents swap at one time slice only to swap-back again in the next time slice, etc.

```
1:    while(1){
2:        newHost=findBestHost(a_k,oldHost);
3:        if(newHost!=Host(a_k)){
4:            send("swap message", addressOf(Host(Parent(this))))
5:            recv(msg, addressOf(Host(Parent(this))))
6:            if(msg=="nok") {
7:                wait(n);
8:                continue;
9:            }
10:       perform migration of a_k;
11:       Host(a_k)=newHost;
12:       agentSet={Parent(a_k), children of a_k};
13:       for each agent a_i ∈ agentSet {
14:           send("Host(a_k)",Host(a_i));
15:       }
16:   }
17:   wait(n);
18: }
```

Figure 9. Pseudocode for agent a_k (NCS-main thread).

The aforementioned process is shown in Fig. 9, lines 5-9. It must be noted that there is also an assistant thread that is responsible for replying "ok" or "nok" in swap messages. Specifically, the assistant thread checks whether there may be cases of swaps and replies accordingly. The assistant thread is also responsible for updating changes of the hosts of remote agents having parent-child relationships with the agents hosted in the current node.

V. Extending Strategies to Support Group Agent Migrations

It has been already proved that performing only single agent migrations may result in suboptimal agent placements [20]. Therefore, a strategy that does consider group agent migrations may adversely affect the total network overhead. It must also be noted that the above will also affect the network longevity because sensor batteries are largely depleted when transmitting data through wireless chip [15]. For the above reason, the proposed strategies must be adapted to consider also group agent migrations.

In [20] we have shown that GRAL* is an algorithm that performs group agent migrations in an optimal manner. Therefore, it is of paramount importance to update the aforementioned strategies to consider group agent migrations. Before proceeding with the updates of the strategies, we first explain the basic steps that GRAL* performs to identify a beneficial group of agents to be migrated. For more information the reader is referred to [20]. At a first step, each sensor/actuator within the system identifies (in an independent manner) groups of disjoint co-located agents. Each such group is structured as a tree, with each node of the tree representing one or more agents. At a second step, the agents of a group are iterated in a bottom-up manner to judge whether a leaf agent must be pruned from the tree or merged with its parent. The second step is repeated until only one node in the tree is left. Normally, the aforementioned node represents more than one agent. If the migration benefit of the aforementioned node is positive, then a migration group has been identified. Otherwise, the identified group of agents is not considered for migration. The sensors/actuators within the system must record the communication load between agents so as to be able to perform the aforementioned steps of GRAL*. When a sensor/actuator identifies a beneficial group of agents, then it proceeds with the creation of a hyper-agent which is described below.

Hyper-agent transformation: The transformation starts by removing all of the edges that connect agents belonging to the targeted group. The above edges are also called *internal edges*, while the edges that connect agents of the targeted group with agents not belonging to that group are called *external edges.* All of the external edges of the targeted group become edges of the hyper-agent to be formed. In case the hyper-agent in question acquires more than one edge towards an external agent, then we merge all of these edges into one, with its weight being equal to the sum of the weights of the merged edges (see Fig. 10 for an example).

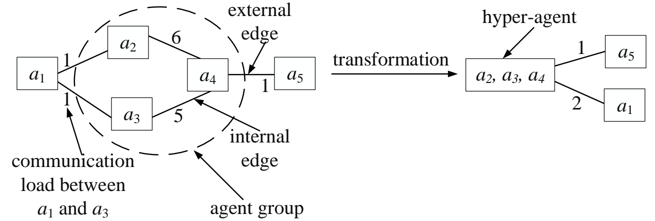

Figure 10. Transforming a group of agents into a hyper-agent.

After transforming a beneficial group of agents into a hyper-agent, we remove the corresponding threads of the agents participating in the hyper-agent in question. Then, a new main (and possibly assistant) thread representing the hyper-agent is created. The aforementioned thread/threads are in accordance to the threads shown in Fig. 3, and Figs. 5-9. The only difference is that they represent and act upon a group of agents instead of a single one.

VI. Computing Capacity Constraints

The aforementioned strategies do not take into account constraints on the computing capacity of sensors/actuators within the system. However, because of the limited capabilities of such tiny devices, it would not be prudent not to consider computing limitations. Therefore, we consider that the hosted agents of a sensor/actuator within the system cannot exceed the computing capacity of the respective node. It is assumed heterogeneity in computing capacities of sensors/actuators, as well as in computing requirements of agents. Unfortunately, when considering computing capacity constraints the problem becomes NP-hard [22].

To tackle the problem of computing capacity constraints on nodes, we enforce that each node within the system broadcasts its available computing capacity towards its 1-hop neighbors. The above step takes place only when there is a change in the current available computing capacity of

the respective node (agent departure/arrival). When a node decides to offload single agents, then it just sends a control message towards the target node, along with the size of each agent to be migrated as well as with its benefit. When dealing with groups of agents, then we modify GRAL* not to return a group but a set of groups for different levels of computing capacity. In the sequel, the node sends a control message containing the whole set of groups, along with their computing capacities as well as their benefits.

The nodes process the hosting requests at the end of a pre-specified period. Hence the incoming hosting requests are queued on the respective node until the period expires. In the sequel, the corresponding node runs the knapsack algorithm with the following parameters. The current available capacity of the respective node plays the role of the knapsack size. The computing requirements of an agent or group of agents correspond to the weight of an object, while the benefit of an agent represents the object benefit. The agents contained in knapsack are chosen to be hosted by the respective agent. For those agents a positive feedback is sent back, while all other agents get a negative feedback.

VII. Convergence Issues

In this section we provide informal proofs on the convergence properties of the algorithms of Section IV.

Lemma 1. *Allowing agents having parent-child relationships to swap may result in perpetual agent migrations.*

Proof. Consider the case where the network consists of two nodes (n_1 and n_2) and two agents (a_1 and a_2). Let a_1 be parent of a_2, with a_1 being hosted at n_1 and a_2 hosted at n_2. Assume the communicational dependencies between a_1 and a_2 are 10 data units. Therefore the total network overhead is 10 data units. By performing NCS strategy allowing agent swaps, a_1 will decide to migrate towards a_2 and the opposite. However, if the above happens, then a_2 will be hosted by n_1 and a_1 by n_2. Because the above swaps may happen ad infinitum, there is no guarantee for convergence. □

Theorem 1. *NCS achieves convergence after a finite number of migrations owing to swap prevention mechanism*.

Proof. Since agent swaps are prevented in NCS, each agent migration leads to a network overhead reduction of at least 1 data unit. The above is guaranteed because of Lemma 1 and the fact that a migration is only performed to reduce the network overhead. Given a finite network overhead of *N*, then convergence is achieved after performing at most *N* agent migrations. □

Theorem 2. *SCS-CP and SCS-PP converge after a finite number of migrations.*

Proof. Both SCS-CP and SCS-PP do not allow concurrent migrations of agents having parent-child relationship due to the inherent mechanism of coordinating parent-child migrations. Therefore, following the same rationale as that of Theorem 1, i.e., that each migration decreases network overhead, we conclude that convergence is guaranteed. □

Theorem 3. *FCS converges after a finite number of migrations.*

Proof. FCS coordinates the whole process of migrations in a level-by-level fashion. As a result, swapping agents having parent-child relationship is inherently not allowed in FCS. Following the same rationale as that of Theorems 1 and 2, we conclude that convergence is guaranteed. □

Theorem 4. *Each node within the system stops the transmission of control messages after resulting in a stable agent placement.*

Proof. According to Theorems 1-3, for the transition of an old placement to a new stable one, it is only needed a finite number of migrations for all of the proposed strategies.

Therefore, given no changes in the communication patterns between agents, after a point in time all of the agents within the system are stabilized. A node within the system sends a control message only when there is a pending migration or there is a change in its current available computing capacity. Note that changes in the available computing capacity of nodes happen only when agents migrate. Therefore, after convergence has been achieved there will be no node within the system issuing control messages. □

VIII. Evaluation

A. Experimental Setup

We conducted the experimental evaluation using ns2 [25]. The details of the simulation setup are briefly given below.

Network generation. Ten different network topologies have been generated for conducting the experiments. The nodes of the topologies range between 40 and 60. The nodes are scattered randomly in planes between 100×100 and 120×120 distance units. Nodes are assumed to be in range of each other if their Euclidean distance was less than 30 distance units. The corresponding tree-based routing topology is obtained by constructing a spanning tree, whereby each pair of nodes is connected via a single path.

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 five, and for each group two to five agents are randomly chosen as children of a new generic agent. In next iterations, orphan (non-generic and generic) agents are (again) randomly split in groups of five and the process of parent creation is repeated, until a single agent remains which becomes the root of the application. Three different application structures are generated with (50, 22), (25, 12) and (10, 5) (non-generic, generic) agents, referred to as app-50, app-25 and app-10, respectively. The initial placement of agents on the network is random, unless stated otherwise.

Application traffic. We let each non-generic agent send one to five messages per simulation time unit to its parent. For the load from a generic agent towards its parent load from a generic agent towards its parent we consider three cases: **(a)** *lavg*: the agent sends the average of

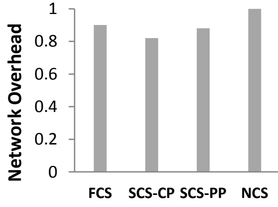

Figure 11. Network overhead

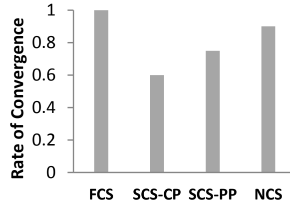

Figure 12. Rate of convergence

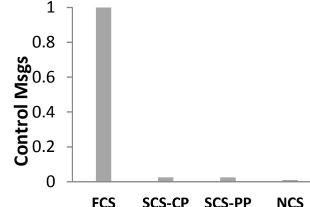

Figure 13. Control msgs per migration

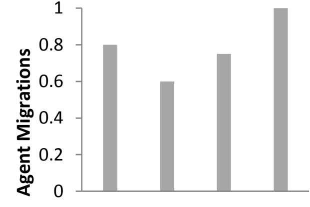

Figure 14. Number of agent migrations

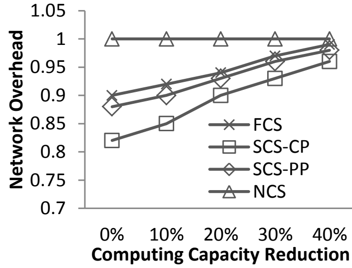

Figure 15. Network overhead

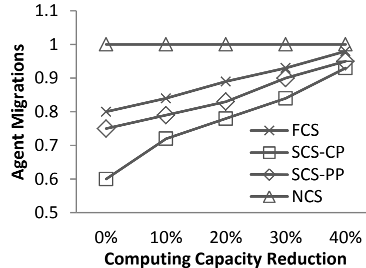

Figure 16. Number of agent migrations

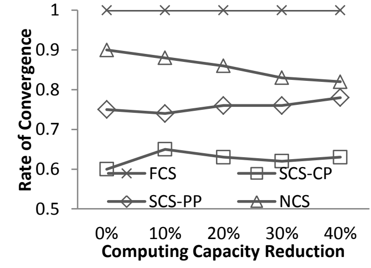

Figure 17. Rate of convergence

the load received from its children, corresponding to a data aggregation scenario, **(b)** *lsum*: the agent sends to its parent the sum of the loads received from its children, corresponding to a forwarding scenario, and **(c)** *lmix*: half generic agents (randomly chosen) generate load according to *lavg* and the other half according to *lsum*. All messages are of equal size. The application-level traffic pattern remains static to allow the algorithms to converge.

B. Results without Computing Capacity Limitations

In the first set of experiments, we consider no computing capacity limitations on the nodes within the system and compared FCS, SCS-CP, SCS-PP, and NCS strategies. We chose network topologies of 50 nodes, and a mixed application of 5 applications of each application type (app-10, 25, 50). In the experiments we record the normalized algorithm performance according to the worst strategy.

In Fig. 11 we show the network overhead in the system. As can be seen, SCS-CP achieves the best performance, with SCS-PP and FCS following. Comparing SCS-CP and SCS-PP, we observe that allowing children to take migration decisions first, before their parents, leads to superior results. This is due to the fact that children are closer to immobile non-generic agents. FCS has the second worst performance as it takes a lot of time from the point where a migration needed to be performed until it is actually performed. This is because the coordination is performed through the root agent, making the migration process less re-active than SCS-CP and SCS-PP. Lastly, even though NCS is more re-active than other strategies, it yields the worst results, because the lack of coordination between parents and children leads to inefficient migrations and oscillations.

Next, we compare the speed of convergence of the proposed strategies. The superiority of SCS-CP is clearly shown in Fig. 12. The second best results are achieved by SCS-PP, with NCS and FCS following behind. The reason that SCS-CP results in the highest rate of convergence is attributed to the stability of the migrations it decides. The aforementioned stability stems from the fact that as the migration decisions are firstly taken by the children it is more unlikely to have oscillations. FCS results in the worst performance since coordination starts from the root agent and migrations are performed in a level-by-level fashion.

The control messages exchanged between nodes to take migration decisions are shown in Fig. 13. As expected, FCS results in a large amount of control messages. The above is due to the number of messages exchanged across the whole application tree to coordinate the migration process. On the other extreme, NCS produces the fewest messages as there is no coordination when taking a decision for agent migrations. SCS strategies follow closely behind NCS. It must be noted that FCS results into almost two orders of magnitude more messages compared to other strategies.

In Fig. 14, we show the agent migrations performed by each strategy. As observed, the biggest number of migrations is performed by NCS. The above is because of the oscillations occurred due to the fact that NCS does not provide any coordination among migrations. On the other extreme, SCS-CP achieves the least number of migrations, with SCS-PP and FCS following behind. The oscillations occurred in FCS strategy is due to the fact that the migrations take place in a level-by-level fashion.

C. Results with Computing Capacity Limitations

We have also performed experiments taking into account the computing capacity limitations of the system nodes. Particularly, when the previous experiments were conducted we captured the computing utilization of the nodes within the system during the transition from the old placement towards the new one. Then, according to the recorded utilizations we performed extra experiments by reducing the total available capacity of the nodes by 10%, 20%, 30%, and 40%.

In Fig. 15 we compare the network overhead among the strategies. As it can be seen, the performance ranking of the

algorithms remains the same with the one in the previous subsection. However, we can observe that when reducing the computing capacity of the nodes by a large amount the network overhead of all of the strategies tend to be the same. The above is reasonable, since by reducing the computing capacity of the nodes the probability of performing a migration decreases. This is clearly illustrated in Fig. 16 which plots the number of migrations.

Finally, Fig. 17 illustrates the rate of convergence. As it can be observed, all algorithms are almost unaffected when reducing the computing capacity of nodes except NCS. The above is quite remarkable since the rate of convergence of NCS is mainly affected by the oscillations when performing migrations. Consequently, when the computing capacity of nodes decreases the number of migrations also decreases, resulting in fewer oscillations. In terms of control messages per agent migration, the figure is omitted because there is almost no change when decreasing the computing capacity. The above is because each strategy consumes an almost fixed number of control messages per migration.

IX. CONCLUSIONS

In this work, we tackled the agent migration problem to minimize network overhead and agent migration oscillations, as well as to maximize convergence rate. Fully-coordinated (FCS) as well as semi-coordinated (SCS) strategies were proposed. Convergence guarantees were given via informal proofs. All algorithms were experimentally evaluated against the existing uncoordinated approach (NCS). A particular variation of semi-coordination (SCS-CP) was found to be a winner across four metrics of interest namely: network overhead, convergence rate, control messages and agent migrations, thus, consists our major contribution.

ACKNOWLEDGEMENTS

This work was supported in part by the China National Basic Research Program (973 Program, No.2015CB352400) and NSFC under grant U1401258.

REFERENCES

[1] F. Aielo, G. Fortino, R. Gravina, A. Guerrieri "A Java-based Agent Platform for Programming Wireless Sensor Networks," *Computing*, vol 54(3), pp. 439-454, 2010.

[2] I. Ayala, M. Amor, L. Fuentes, "An Agent Platform for Self-Configuring Agents in the Internet of Things," *Infrastructures and Tools for Multiagent Systems*, 2012.

[3] O. Biran, A. Corradi, M. Fanelli, L. Foschini, A. Nus, D. Raz, E. Silvera, "A Stable Network-Aware VM Placement for Cloud Systems," *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2012.

[4] S. Bosse, "Design and Simulation of Material-Integrated Distributed Sensor Processing with a Code-Based Agent Platform and Mobile Multi-Agent Systems," *Sensors*, vol. 15(2), pp. 4513-4549, 2015.

[5] G. Chatzimilioudis, A. Cuzzocrea, D. Gunopoulos, N. Mamoulis, "A Novel Distributed Framework for Optimizing Query Routing Trees in Wireless Sensor Networks via Optimal Operator Placement," *Journal of Computer and System Sciences*, vol. 79, pp. 349-368, 2013.

[6] J. Domaszewicz, M. Roj, A. Pruszkowski, M. Golanski, K. Kacperski, "ROVERS: Pervasive Computing Platform for Heterogeneous Sensor-Actuator Networks," *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, 2006.

[7] G. Fortino, A. Guerrieri, M. Lacopo, M. Lucia, W. Russo, "An Agent-Based Middleware for Cooperating Smart Objects," *Highlights on Practical Applications of Agents and Multi-Agent Systems Communications in Computer and Infromation Science*, vol. 365, pp. 387-398, 2013.

[8] C. L. Fok, G. C. Roman, C. Lu, "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications," *International Conference on Distributed Computing Systems*, 2005.

[9] E. Jung, I. Cho, S. M. Kang, "iotSilo: The Agent Service Platform Supporting Dynamic Behaviour Assembly for Resolving the Heterogeneity of IoT," *International Journal of Distributed Sensor Networks*, 2014.

[10] N. Kothari, R. Gummadi, T. Millstein, R. Govindan, "Reliable and Efficient Programming Abstractions for Wireless Sensor Networks," *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[11] H. Liu, T. Roeder, K. Walsh, R. Barr, E. G. Sirer, "Design and Implementation of a Single System Image Operating System for Ad Hoc Networks," *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2005.

[12] Z. Lu, Y. Wen, "Distributed and Asynchronous Solution to Operator Placement in Large Wireless Sensor Networks," *IEEE International Conference on Mobile Ad-hoc and Sensor Networks*, 2012.

[13] Z. Lu, Y. Wen, R. Fan, S.-L. Tan, J. Biswas, "Toward Efficient Distributed Algorithms for In-Network Binary Operator Tree Placement in Wireless Sensor Networks," *IEEE Journal on Selected Areas in Communications*, vol 31(4), 2013.

[14] X. Meng, V. Pappas, L. Zhang, "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement," *IEEE Conference on Computer Communications (INFOCOM)*, 2010.

[15] G.J. Pottie, W.J. Kaiser, "Wireless Integrated Network Sensors, *ACM Communications*, vol 43(5), pp. 51-58, 2000.

[16] U. Ramachanran, R. Kumar, M. Wolenetz, B. Cooper, B. Agarwalla, J. Shin, P. Hutto, A. Paul, "Dynamic Data Fusion for Future Sensor Networks," *ACM Transactions on Sensor Networks*, vol 2(3), 2006.

[17] A. Ranganathan, S. Chetan, J. Al-Muhtadi, R.H. Campbell, M.D. Mickunas, "Olympus: A High-Level Programming Model for Pervasive Computing Environments," *IEEE International Conference on Pervasive Computing and Communications (PERCOM)*, 2005.

[18] J. Sonnek, J. Greensky, R. Reutiman, A. Chandra, "Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration," *International Conference on Parallel Processing (ICPP)*, 2010.

[19] N. Tziritas, G. Georgakoudis, S. Lalis, T. Paczesny, J. Domaszewicz, P. Lampsas, T. Loukopoulos, "Middleware Mechanisms for Agent Mobility in Wireless Sensor and Actuator Networks," *International Conference on Sensor Systems and Software*, pp. 30-44, 2012.

[20] N. Tziritas, S. U. Khan, T. Loukopoulos, S. Lalis, C.-Z. Xu, P. Lampsas, "Single and Group Agent Migration: Algorithms, Bounds, and Optimality Issues," *IEEE Transactions on Computers*, vol. 63(12), pp. 3143-3161, 2014.

[21] N. Tziritas, S. U. Khan, C.-Z. Xu, T. Loukopoulos, S. Lalis, "On Minimizing the Resource Consumption of Cloud Applications Using Process Migration," *Journal of Parallel and Distributed Computing (JPDC),* vol 73(12), pp. 1690-1704, 2013.

[22] N. Tziritas, P. Lampsas, S. Lalis, T. Loukopoulos, S. U. Khan, C.-Z. Xu, "Introducing Agent Evictions to Improve Application Placement in Wireless Distributed Systems," *International Conference on Parallel Processing (ICPP)*, 2012.

[23] N. Tziritas, T. Loukopoulos, S. Lalis, P. Lampsas, "On Deploying Tree Structured Agent Applications in Networked Embedded Systems," *International Euro-Par Conference (EUROPAR)*, 2010.

[24] N. Tziritas, C.-Z. Xu, T. Loukopoulos, S. U. Khan, Z. Yu, "Application-Aware Workload Consolidation to Minimize Both Energy Consumption and Network Load in Cloud Environments, "*International Conference on Parallel Processing (ICPP)*, 2013.

[25] Network Simulator2 (ns2), http://www.isi.edu/nsnam/ns/.