UNIVERSITY OF THESSALY

Algorithms and System-level Support for Agent Placement and Migration in Wireless Sensor Networks.

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in
Computer and Communication Engineering
by

Nikolaos Tziritas

July 2011

UNIVERSITY OF THESSALY

Algorithms and System-level Support for Agent Placement and Migration in Wireless Sensor Networks.

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in
Computer and Communication Engineering
by

Nikolaos Tziritas

July 2011

Dissertation Committee:

Prof. Alexis Delis
Prof. Eleni Karatza
Prof. Spyros Kontogiannis
Prof. Leandros Tassiulas
Associate Prof. Spyros Lalis (Primary Supervisor)
Assistant Prof. Iordanis Koutsopoulos
Assistant Prof. Thanasis Loukopoulos (Secondary Supervisor)

The Dissertation of Nikolaos Tziritas is approved by :

Dr. Spyridon Lalis, Primary Supervisor

Dr. Athanasios Loukopoulos, Secondary Supervisor

Dr. Iordanis Koutsopoulos

University Of Thessaly
July 2011

Dedicated to my family:
my father Charalampos, my Mother Olga, my brother Ioannis and my soon-to-be sister-in-law Dimitra.

Acknowledgements

I would like to express my gratitude to Dr. Spyros Lalis for his excellent supervision/mentoring and invaluable support throughout these years. Without his guidance, constructive criticism and truly scientific attitude, to name a few of his contributions, this doctoral study would not have been possible.

I'm also greatly indebted to Dr. Thanasis Loukopoulos who brought unique perspectives to my research, enriching it greatly. His co-supervision/mentoring and invaluable contribution, along with the endless discussions made also this work possible.

I wish to thank Dr. Petros Lampsas who never stopped helping and encouraging me during the entire PhD thesis. I'm also very appreciative for his fruitful and stimulating conversations.

I also would like to express my thanks to my colleagues Giorgis Georgakoudis, Manos Koutsoumpelias, Apostolos Apostolaras and Dr Dimitris Syrivelis for making my study breaks very enjoyable. I would like to mention that Manos and Giorgis contributed to the implementation of the agent manager. Giorgis also helped me to conduct experiments to evaluate the performance of the agent creation and migration protocols.

My sincerely thanks go to the members of my Dissertation Committee for accepting to review my thesis.

A million thanks to my beloved family and friends for their moral support. I want to especially thank Manos Chatzidakis, Giannis Portokalakis, Manthos Sarris, Giorgos Mpoulios, and Lefteris Mamoulakis.

I very much acknowledge the financial support of the Alexander S. Onassis Public Benefit Foundation.

Finally, I want to acknowledge the financial support of POBICOS, FP7-ICT-223984, research project.

Abstract

Wireless sensor systems have drawn much attention from a considerable part of scientific community during the last years. The advances in this field range from the design of battery-powered embedded nodes to the development of software (i.e. operating system, middleware, etc) especially designed to run on such resource-constrained devices. One of the most challenging parts for both hardware and software oriented work is to maximize the lifetime of such nodes. This thesis focuses on the design and implementation of mobile code placement and migration algorithms for distributed applications in order to reduce the amount of application-level communication performed over the network. Since the largest part of a node's energy expenditure is attributed to the wireless communication (not code execution), reducing the energy consumption becomes of paramount importance, leading in that way to an increased system lifetime. In the sequel, we give a brief overview of the application model, the algorithms and the middleware designed and implemented in the context of this thesis.

The model adopted in this work is inspired by the POBICOS [91] platform, where the application is organized as a set of software entities (agents) that communicate with each other to implement the desired functionality. An agent can be "non-generic" or "generic". Non-generic agents use special resources of a node, e.g. a sensor measuring a physical quantity or an actuator controlling a device or function. On the contrary, generic agents perform computational tasks and decision making at a higher level, without relying on special resources.

Chapter 1 introduces the agent migration problem stated as follows: given an application that is deployed in a sensor network, perform generic agent migrations in order to reduce the data exchanged over the network due to the application-level communication between agents. We propose fully distributed algorithms that migrate an agent towards its center of gravity (in terms of communication load), thereby reducing the network cost. Also, two protocols are presented for handling the case of nodes with storage constraints (for hosting agents).

Chapter 2 examines the same (above) problem, with the difference that it considers migrations of agent groups instead of single agent migrations. The algorithms in question deal with co-located agents that are "mutually dependent", which in the case of the simpler algorithms may hinder migration, leading to noticeably inferior placements.

Chapter 3 discusses the competitiveness of the aforementioned algorithms versus the optimal algorithm. Also, it presents an enhancement of the group migration algorithms in order for them to produce an optimal agent placement (in terms of the network cost incurred by the application). It should be stressed that this enhancement guarantees optimality only if nodes do not have storage constraints, else the problem is NP-complete.

Chapter 4 proposes fully distributed algorithms for the problem of generic agent migrations for resource-constrained nodes, introducing the concept of "evictions". Specifically, agent migrations are considered that are not beneficial in their own right but free space which can be used to perform 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.

In Chapter 5 we focus on the problem that the aforementioned algorithms are not able to "guess" whether a (group) migration will turn out to actually reduce the network cost. They simply assume that the structure and communication pattern of the application remains stable for a "sufficiently" long time, so as to amortize the migration cost. As a consequence, frequent changes in the application-level load may lead to frequent agent migrations, thereby increasing the network cost (instead of reducing it). For example, an agent may continuously "oscillate" between two nodes due to periodic changes of the communication load with other agents (changing its center of gravity), before the respective migration cost is amortized. For this reason, we propose online algorithms, along with a discussion of their competitiveness versus the offline optimal algorithm.

In Chapters 6 and 7 we propose centralized algorithms tackling more complex problems. Specifically, chapter 6 addresses the problem of reducing the network cost through migrations of both generic and non-generic agents, considering that the nodes of the system have storage capacity limitations. The proposed algorithms use graph coloring techniques. In Chapter 7, a two-dimensional problem is considered, the objectives being: (a) to maximize the number of agents hosted by the nodes of the system; and (b) to maximize the network lifetime (maximize the lifetime of the first node that depletes its battery). We propose algorithms solving each dimension (sequentially) in an independent way, along with a branch-and-bound algorithm

tackling the problem concurrently in both dimensions. Regarding the first dimension of the problem, a considerable part of the algorithms involves the de-fragmentation of the nodes' storage capacity, through agent migrations.

Chapter 8 describes the implementation of the component of the POBICOS middleware that provides full-fledged, distributed, agent management functionality, on top of the TinyOS embedded operating system. Specifically, we describe: i) the mechanism for creating agents on eligible nodes; ii) the mechanism for transporting agent-level messages; iii) the mechanism detecting and destroying "orphan" agents; and iv) the mechanism for the migration of generic agents with full transparency for the application.

Finally, Chapter 9 discusses works related to this thesis, while Chapter 10 includes an overview of this dissertation and future directions.

Περίληψη

Τα τελευταία χρόνια ένα σημαντικό μέρος της επιστημονικής κοινότητας έχει στρέψει το ενδιαφέρον της προς τα ασύρματα δίκτυα αισθητήρων (wireless sensor networks). Οι τεχνολογικές εξελίξεις σε αυτό τον τομέα ξεκινούν από την σχεδίαση ενσωματωμένων κόμβων που ρευματοδοτούνται μέσω μπαταρίας και φτάνουν μέχρι την ανάπτυξη λογισμικού (λειτουργικών συστημάτων, ενδιάμεσου λογισμικού, κλπ) ειδικά σχεδιασμένου για να μπορεί να εκτελείται με τους περιορισμένους πόρους αυτών των συσκευών. Η μεγιστοποίηση της διάρκειας ζωής των κόμβων αποτελεί πρόκληση τόσο σε επίπεδο υλικού όσο και σε επίπεδο λογισμικού. Η παρούσα διατριβή αφορά στην σχεδίαση και ανάπτυξη αλγορίθμων τοποθέτησης και μετανάστευσης κώδικα κατανεμημένων εφαρμογών με στόχο την μείωση του φόρτου επικοινωνίας της εφαρμογής που πραγματοποιείται πάνω από το ασύρματο δίκτυο. Καθώς το μεγαλύτερο μέρος της ενέργειας των κόμβων ξοδεύεται συνήθως στην επικοινωνία (όχι στην εκτέλεση κώδικα), με αυτό το τρόπο μειώνεται η κατανάλωση ενέργειας και αυξάνεται η διάρκεια ζωής των κόμβων του συστήματος. Στη συνέχεια, παραθέτουμε μια σύντομη περιγραφή του μοντέλου εφαρμογής, των αλγορίθμων και του ενδιάμεσου λογισμικού που σχεδιάστηκαν και αναπτύχθηκαν στα πλαίσια της διατριβής.

Το μοντέλο που υποθέτει η εργασία είναι εμπνευσμένο από την πλατφόρμα POBICOS [91], όπου η εφαρμογή σχεδιάζεται ως ένα σύνολο από τμήματα λογισμικού (πράκτορες) που επικοινωνούν μεταξύ τους για να υλοποιήσουν την επιθυμητή λειτουργικότητα. Οι πράκτορες διαχωρίζονται σε «ειδικούς» και «γενικούς». Οι ειδικοί πράκτορες χρησιμοποιούν ειδικούς πόρους ενός κόμβου, π.χ. έναν αισθητήρα που δίνει τιμές για ένα φυσικό μέγεθος ή ένα ελεγκτή μιας συσκευής ή λειτουργίας. Αντίθετα, οι γενικοί πράκτορες πραγματοποιούν λειτουργίες επεξεργασίας και λήψης αποφάσεων σε πιο ψηλό επίπεδο, χωρίς να απαιτούν ειδικούς πόρους. Οι πράκτορες της εφαρμογής κατανέμονται (δυναμικά) στους κόμβους του συστήματος ανάλογα με τους πόρους που αυτοί διαθέτουν.

Το κεφάλαιο 1 εισάγει το πρόβλημα μετακίνησης πρακτόρων που διατυπώνεται ως εξής: δοθήσας μιας εφαρμογής που έχει αναπτυχθεί σε ένα δίκτυο αισθητήρων, το ζητούμενο είναι να πραγματοποιηθούν μετακινήσεις γενικών πρακτόρων ώστε να μειωθεί το κόστος δικτύου λόγω της επικοινωνίας σε επίπεδο εφαρμογής. Προτείνουμε πλήρως κατανεμημένους αλγορίθμους με στόχο την μετακίνηση του κάθε πράκτορα προς το κέντρο βάρους του (όσον αφορά το κόστος επικοινωνίας), πράγμα που ελαχιστοποιεί και το συνολικό κόστος επικοινωνίας σε επίπεδο δικτύου. Επίσης παρουσιάζονται δύο πρωτόκολλα για τον χειρισμό της περίπτωσης όπου οι κόμβοι διαθέτουν περιορισμένη αποθηκευτική χωρητικότητα για την φιλοξενία πρακτόρων.

Το κεφάλαιο 2 αφορά στο ίδιο (παραπάνω) πρόβλημα, με τη διαφορά ότι εξετάζει μετακινήσεις από ομάδες πρακτόρων αντί από μεμονωμένους πράκτορες. Οι εν προκειμένω αλγόριθμοι αντιμετωπίζουν την «αμοιβαία εξάρτηση» πρακτόρων που φιλοξενούνται στον ιδιο κόμβο και επικοινωνούν μεταξύ τους, πράγμα που, στους πιο απλούς αλγορίθμους, μπορεί να εμποδίσει την μετακίνηση τους, έχοντας ως αποτέλεσμα μια (σημαντικά) χειρότερη τοποθέτηση.

Το κεφάλαιο 3 εξετάζει την ανταγωνιστικότητα των παραπάνω αλγορίθμων σε σχέση με τον βέλτιστο αλγόριθμο. Επίσης, παρουσιάζει μία τροποποίηση που αφορά τους αλγορίθμους ομαδοποίησης έτσι ώστε αυτοί να καταλήγουν στη βέλτιστη τοποθέτηση των πρακτόρων (αναφορικά με το συνολικό φόρτο επικοινωνίας της εφαρμογής πάνω από το δίκτυο). Να τονιστεί πως αυτή η τροποποίηση καθιστά τους αλγορίθμους ομαδοποίησης βέλτιστους μόνο όταν δεν εξετάζουμε κόμβους με περιορισμένη χωρητικότητα, διαφορετικά το πρόβλημα είναι NP-complete.

Το κεφάλαιο 4 προτείνει πλήρως κατανεμημένους αλγορίθμους για την τοποθέτηση πρακτόρων σε κόμβους περιορισμένης αποθηκευτικής χωρητικότητας, εισάγοντας την έννοια της «έξωσης». Πιο συγκεκριμένα, εξετάζονται μετακινήσεις πρακτόρων που είναι μεμονωμένα ασύμφορες αλλά μπορεί να απελευθερώσουν χώρο που στη συνέχεια μπορεί να χρησιμοποιηθεί για την μετακίνηση άλλων πρακτόρων. Βεβαίως, ο απώτερος σκοπός εξακολουθεί να είναι η μείωση του κόστους επικοινωνίας, επομένως απαιτείται το συνολικό όφελος των μετακινήσεων να υπερβαίνει το κόστος των ασύμφορων μετακινήσεων.

Στο κεφάλαιο 5 εστιάζουμε στο πρόβλημα του ότι οι προαναφερθέντες αλγόριθμοι δεν έχουν την ικανότητα να «μαντέψουν» αν μία (ομαδική) μετακίνηση θα αποβεί τελικά προσοδοφόρα η όχι. Απλά υποθέτουν ότι ο αριθμός των πρακτόρων και ο φόρτος επικοινωνίας μεταξύ τους θα παραμείνουν σταθερά για ένα «αρκετά» μεγάλο χρονικό διάστημα, έτσι ώστε να αποσβεσθεί το όποιο κόστος μετακίνησης των πρακτόρων. Επομένως, συχνές αλλαγές στο σχήμα και φόρτο επικοινωνίας της εφαρμογής μπορεί να οδηγήσουν τους παραπάνω αλγορίθμους σε συχνές

μετακινήσεις πρακτόρων που τελικά αυξάνουν το κόστος επικοινωνίας πάνω από το δίκτυο (αντί να το μειώνουν). Για παράδειγμα, ένας πράκτορας μπορεί να «παλινδρομεί» συνεχώς μεταξύ δύο κόμβων, λόγω περιοδικών αλλαγών στο φόρτο επικοινωνίας με άλλους πράκτορες (αλλάζοντας το κέντρο βάρος του), χωρίς ποτέ να αποσβένεται το κόστος μετακίνησης. Για αυτό το λόγο, προτείνουμε online αλγορίθμους, δείχνοντας επίσης πόσο ανταγωνιστικοί είναι σε σχέση με τον offline βέλτιστο αλγόριθμο.

Στα κεφάλαια 6 και 7 προτείνουμε κεντρικοποιημένους αλγορίθμους που λύνουν πιο πολύπλοκα προβλήματα. Ειδικότερα, το κεφάλαιο 6 καταπιάνεται με το πρόβλημα της ελαχιστοποίησης του κόστους δικτύου μέσω μετακινήσεων πρακτόρων όχι μόνο γενικού αλλά και ειδικού τύπου, όταν οι κόμβοι διαθέτουν περιορισμένη αποθηκευτική χωρητικότητα. Οι αλγόριθμοι που προτείνονται κάνουν χρήση τεχνικών χρωματισμού γράφου. Στο κεφάλαιο 7 εξετάζεται ένα πρόβλημα δύο διαστάσεων, όπου το ζητούμενο είναι (α) να φιλοξενηθούν όσο γίνεται περισσότεροι πράκτορες στους κόμβους του δικτύου, και (β) να αυξηθεί η διάρκεια ζωής του συστήματος (δηλαδή να μεγιστοποιηθεί ο χρόνος ζωής του πρώτου κόμβου που θα εξαντλήσει τη μπαταρία του). Προτείνονται αλγόριθμοι που λύνουν το πρόβλημα ξεχωριστά (σειριακά) σε κάθε διάσταση, μαζί με ένα αλγόριθμο branch-and-bound που λύνει το πρόβλημα ταυτόχρονα και στις δύο διαστάσεις του. Ένα σημαντικό τμήμα των αλγορίθμων ως προς την πρώτη διάσταση του προβλήματος αφορά στην αποκερματοποίηση του αποθηκευτικού χώρου στους κόμβους του δικτύου, μέσω μετακινήσεων πρακτόρων.

Το κεφάλαιο 8 περιγράφει την υλοποίηση του τμήματος του ενδιάμεσου λογισμικού POBICOS που παρέχει μια ολοκληρωμένη, κατανεμημένη, διαχείριση των πρακτόρων της εφαρμογής, πάνω από το ενσωματωμένο λειτουργικό σύστημα TinyOS. Συγκεκριμένα, περιγράφονται: i) ο μηχανισμός δημιουργίας νέων πρακτόρων σε κόμβους με τους κατάλληλους πόρους, ii) ο μηχανισμός ανταλλαγής μηνυμάτων μεταξύ πρακτόρων, iii) ο μηχανισμός ανίχνευσης και καταστροφής «ορφανών» πρακτόρων, και iv) ο μηχανισμός μετακίνησης γενικών πρακτόρων με πλήρη διαφάνεια μετακίνησης σε επίπεδο εφαρμογής.

Τέλος, το κεφαλαιο 9 αναφέρει εργασίες που είναι συναφείς με την παρούσα διατριβή, ενώ το κεφάλαιο 10 παρέχει τα γενικά συμπεράσματα για την παρούσα δουλειά.

Contents

List of Figures

List of Tables

List of Equations

Chapter 1

On Deploying Tree Structured Agent Applications in Networked Embedded Systems

1 Introduction

Mobile code technologies for networked embedded systems, like Aggila [36], SmartMessages [52], Rovers [27] and POBICOS [91], allow the programmer to structure an application as a set of mobile components that can be placed on different nodes based on their computing resources and sensing/actuating capabilities. From a system perspective, the challenge is to optimize such a placement (through migrating the mobile components) taking into account the message traffic between application components. It should be stressed that this work focuses on non-highly volatile environments, e.g., home or office environments. Therefore, we can expect that: (i) the arrival of new applications is rather infrequent; (ii) an application is expected to be resident for a fairly large amount of time (enough to offset any potential migration overhead).

This chapter presents distributed algorithms for the dynamic migration of mobile components, referred to as agents, in a system of networked nodes with the objective of reducing the network load due to agent-level communication. The proposed algorithms are simple so they can be implemented on nodes with limited memory and computing capacity. Also, modest assumptions are made regarding the knowledge of routing paths used for message transport. The algorithms rely on information that can be provided by even simple networking or middleware logic without incurring (significant) additional communication overhead.

The contributions of this work are the following: (i) we identify and formulate the agent placement problem (APP) in a way that is of practical use to the POBICOS middleware but can also prove useful to other work on mobile agent systems with placement constraints, (ii) we present a distributed algorithm that relies on minimal network knowledge and extend it so that it can exploit additional information about the underlying network topology (if available), (iii) we evaluate both algorithm variants via simulations and discuss their performance.

2 Application and System Model, Problem Formulation

This section introduces the type of applications targeted in this work and the underlying system and network model. It then formulates the agent placement problem (APP) and the respective optimization objectives.

2.1 Application model

We focus on applications that are structured as a set of cooperating agents organized in a hierarchy. For instance, consider a demand-response client which tries to reduce power consumption upon request of the energy utility. A simplified possible structure is shown in *Fig 1.1*. The lowest level of the tree comprises agents that periodically report individual device status and power consumption to a room agent, which reports (aggregated) data for the entire room to the root agent. When the root decides to lower power consumption (responding to a request issued by the electric utility), it requests some or all room agents to curve power consumption as needed. In turn, room agents trigger the respective actions (turn off devices, lower consumption level) in the end devices by sending requests to the corresponding device agents.

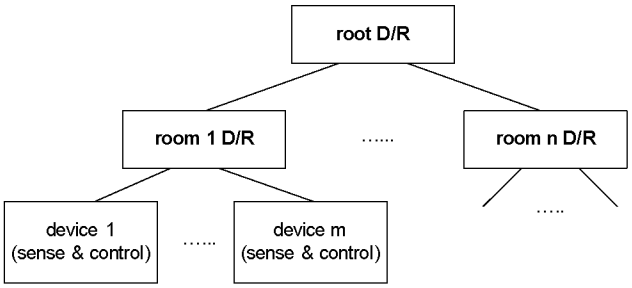

Fig 1.1 Agent tree structure of an indicative sensing/control application.

Leaf (sensing and actuating) agents interact with the physical environment and must be placed on nodes that provide some specific resources (e.g. sensing or actuating capabilities), hence are called "non-generic". On the other hand, intermediate agents perform their tasks using just general-purpose computing resources which can be provided by any node; thus we refer to these agents as "generic". In *Fig 1.1*, device agents are non-generic while all other agents are generic.

Agents can migrate between nodes to offload their current hosts or to get closer to the agents they communicate with. In our work we consider migration *only* for generic agents because their operation is location- and node-independent by design, while non-generic agents remain fixed on the nodes where they were created. Still, the ability to migrate generic agents creates a significant optimization potential in terms of reducing the overall communication cost.

2.2 System model

We assume a network of capacitated (resource-constrained) nodes with sensing and/or actuating capabilities. Let n_i denote the i^{th} node, $1 \leq i \leq N$ and $r(n_i)$ its resource capacity (processing power or memory size). The capacity of a node imposes a generic constraint to the number of agents it can host.

Nodes communicate with each other on top of a (wireless) network that is treated as a black box. The underlying routing topology is abstracted as a graph, its vertices representing nodes and each edge representing a bidirectional routing-level link between a node pair. In this work we consider *tree-based routing*, i.e., there is exactly one path for connecting any two nodes. Let D be a $N \times N \times N$ boolean matrix encoding the routing topology as follows: $D_{ijx}=1$ iff the path from n_i to n_j includes n_x, else $D_{ijx}=0$. Since we assume that the network is a tree $D_{ijx} = D_{jix}$. Also,

D_{iii}=1, D_{ijj}=1 and D_{iij}=0. Let h_{ij} be the path length between n_i and n_j; equal to 0 for i=j. Obviously, $h_{ij} = h_{ji}$.

Each application is structured as a set of cooperating agents organized in a tree-like structure, the leaf agents being non-generic and all other agents being generic. Assuming an enumeration of agents whereby generic agents come first, let a_k be the k^{th} agent, $1 \leq k \leq A+S$, with A and S being equal to the total number of generic and non-generic agents, respectively. Let $r(a_k)$ be the capacity required to host a_k. Agent-level traffic is captured via an $(A+S)\times(A+S)$ matrix C, where C_{km} denotes the load from a_k to a_m (measured in data units over a time period). Note that C_{km} need not be equal to C_{mk}. Also, C_{kk}=0 since an agent does not send messages to itself.

2.3 Problem formulation

For the sake of generality we target the case where all agents are already hosted on some nodes, but the current placement is non-optimal.

Let P be an $N\times(A+S)$ matrix used to encode the placement of agents on nodes as follows: P_{ik}=1 iff n_i hosts a_k, 0 otherwise. Let l_{ijk} (*Eq. 1.1*) denote the load associated with agent a_k hosted at node n_i for a neighbor node n_j; specifically, this load involves the volume of data exchanged between a_k and the agents using n_j as either a hosting or routing node to communicate with a_k.The total network load L incurred by the application for a placement P can then be expressed by *Eq. 1.2*:

$$l_{ijk} = \sum_{m=1}^{A+S} (C_{km} + C_{mk}) d_{ixj}, P_{xm} = 1 \qquad \text{Eq. 1.1}$$

$$L = \sum_{k=1}^{A+S} \sum_{m=1}^{A+S} C_{km} \sum_{i}^{N} \sum_{j}^{N} h_{ij} P_{ik} P_{jm} \qquad \text{Eq. 1.2}$$

A placement P is valid iff each agent is hosted on exactly one node and the node capacity constraints are not violated:

$$\sum_{i=1}^{N} P_{ik} = 1, \ \forall k, \ 1 \leq k \leq A+S \qquad \text{Eq. 1.3}$$

$$\sum_{k=1}^{N} P_{ik} r(a_k) \leq r(n_i), \ \forall i, \ 1 \leq i \leq N \qquad \text{Eq. 1.4}$$

Also, a migration is valid only if starting from a valid placement P it leads to another valid agent placement $P^{'}$ without moving any non-generic agent:

$$P_{ik} = P'_{ik},\ \forall k,\ A \le k \le A+S \qquad \text{Eq. 1.5}$$

The agent placement problem (APP) can then be stated as: starting from an initial valid agent placement P^{old}, perform a series of valid agent migrations, eventually leading to a new valid placement P^{new} that minimizes *Eq. 1.2*. In that sense the agent placement problem (APP) can be renamed to the agent migration problem (AMP). The decision for migrating a_{k} from n_{i} to n_{j} is taken iff l_{ijk} is greater than the total load with all other neighbors of n_{i} plus the local load associated with a_{k}:

$$l_{ijk} > l_{iik} + \sum_{x \neq i,j} l_{ixk},\ h_{ij} = h_{ix} = 1 \qquad \text{Eq. 1.6}$$

The intuition behind *Eq. 1.6* is that by moving a_{k} from its current host n_{i} to a neighbor n_{j}, the distance for the load with n_{j} decreases by one hop while the distance for all other loads, including the load that used to take place locally, increases by one hop. If *Eq. 1.6* holds, the cost-benefit of the migration is positive, hence the migration reduces the total network load as per *Eq. 1.2*.

Note that the resulting optimal placement of APP may be an unreachable placement, meaning that starting from an initial (sub-optimal) placement the optimal one can be reached by only performing a non-feasible "swap" of agents (the involved nodes cannot perform this "swap" because they don't have enough free capacity). A similar feasibility issue is discussed in [78] but in a slightly different context. Also, *Eq. 1.2* does not take into account the cost for performing a migration. This is because we target scenarios where the application structure, agent-level traffic pattern and underlying routing topology are expected to be sufficiently stable to amortize the migration costs.

3 Uncapacitated 1-hop Agent Migration Algorithm

This section presents an agent migration algorithm for the case where nodes can host any number of agents, i.e., without taking into account capacity limitations. In terms of routing knowledge, each node knows only its immediate (1-hop) neighbors involved in transporting inbound and outbound agent messages; we refer to this as *1-hop network awareness*. This information can be provided by even a very simple networking layer. A node does not attempt

to discover additional nodes but simply considers migrating agents to one of its neighbors. An agent may nevertheless move to distant nodes via consecutive 1-hop migrations.

Description. The *1-hop agent migration algorithm* (AMA-1) works as follows. A node records, for each locally hosted agent, the traffic associated with each neighboring node as well as the local traffic, due to the message exchange with remote and local agents, respectively. Periodically, this information is used to decide if it is beneficial for the agent to migrate to a neighbor.

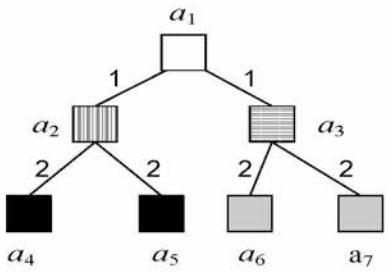

Fig 1.2 Application agent structure

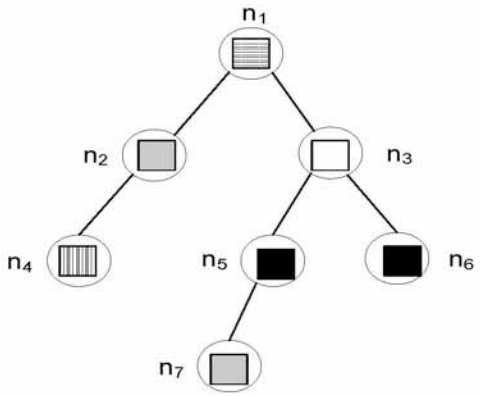

Fig 1.3 Agent placement on the network

Consider the application depicted in *Fig 1.2* which comprises four non-generic agents (a_4, a_5, a_6, a_7), two intermediate generic agents (a_2 a_3) and a generic root agent (a_1), and the actual agent placement on nodes shown in *Fig 1.3*. Let each non-generic agent generate 2 data units per time unit towards its parent, which in turn generates 1 data unit per time unit towards the root (edge values in *Fig 1.2*). Assume that n_1 runs the algorithm for a_3 (striped). The load associated with a_3 for the neighbour node n_2 and n_3 is l_{123}=2 respectively l_{133}=3 while the local load is l_{113}=0. According to *Eq. 1.6* the only beneficial migration for a_3 is for it to move on n_3. Continuing the example, assume that a_3 indeed migrates to n_3 and is (again) checked for migration. This time the relevant loads are l_{313}=2, l_{353}=2, l_{363}=0, l_{333}=1, thus a_3 will remain at n_3. Similarly, a_1 will remain at n_3 while a_2 will eventually migrate from n_4 to n_2 then to n_1 and last to n_3, resulting in a placement where all generic agents are hosted at n_3. This placement is stable since there is no beneficial migration as per *Eq. 1.6*.

Implementation and complexity. For each local agent it is required to record the load with each neighboring node and the load with other locally hosted agents. This can be done using a $A^{'}\times(g+1)$ load table, where $A^{'}$ is the number of local generic agents and g is the node degree

(number of neighbors). The destination for each agent can then be determined as per *Eq. 1.6* in a single pass across the respective row of the load table, in $O(g)$ operations or a total of $O(gA')$ for all agents. Note that the results of this calculation remain valid as long as the underlying network topology, application structure and agent message traffic statistics do not change.

Convergence. For the time being, the algorithm does *not* guarantee convergence because it is susceptible to live-locks. Revisiting the previous example, assume that the application consists only of the right-hand sub-tree of *Fig 1.2*, placed as in *Fig 1.3*. Node n_1 may decide to move a_3 to n_3 while n_3 may decide to move a_1 to n_1. Later on, the same migrations may be performed in the reverse direction, resulting in the old placement etc.

We expect such livelocks to be rare in practice, especially if neighboring nodes invoke the algorithm at different intervals. Nevertheless, to guarantee convergence we introduce a coordination scheme in the spirit of a mutual exclusion protocol. When n_i decides to migrate a_k to n_j it asks for a permission. To avoid "swaps" n_j denies this request if: (i) it hosts an agent $a_{k'}$ that is the child or the parent of a_k, (ii) it has decided to migrate $a_{k'}$ to n_i, and (iii) the identifier of n_j is smaller than that of n_i ($j<i$). Else, n_j grants permission to n_i and does not consider migrating any child or parent of a_k to n_i before the granted migration completes. It is important to note that any migration 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.Convergence is guaranteed since it is no more possible to perform swaps and each migration that is not a swap reduces the network load as per *Eq. 1.2*. It is worth pointing out that such a protocol can be implemented quite efficiently by piggybacking requests and replies on other messages that need to be exchanged anyway in order to perform the actual migration.

4 Uncapacitated *k*-hop Agent Migration Algorithm

This section introduces an extension of the 1-hop algorithm for the case where a node is assumed to know the routing topology within a *k*-hop radius. We refer to this as *k-hop network awareness*. Note this information may be collected in a lazy fashion, incurring a minimal communication overhead, by piggybacking the *k* most recent node identifiers when a (small) message travels through the network. In fact, this information comes for free by employing a naming scheme that encodes path information into node identifiers (e.g., as in ZigBee networks with hierarchical routing).

Description. The *k-hop agent migration algorithm* (AMA-*k*) is a straightforward extension of AMA-1 that exploits *k*-hop awareness. The difference is that for each agent a_m hosted at node n_i, AMA-*k* considers as possible candidates all nodes up to *k-hops* away from n_i which are involved in the message traffic of a_m.

The algorithm chooses the destination for a_m by iteratively evaluating *Eq. 1.6* for neighbour nodes, starting from 1-hop neighbours and working its way to more distant neighbours, following the most beneficial outbound direction. Each iteration determines whether it is beneficial to move a_m to a node that is 1 hop further away from n_i assuming a_m were hosted on the node picked in the previous iteration. The algorithm stops after *k* iterations or earlier when it is no longer beneficial to migrate a_m. AMA-*k* is expected to lead to fewer migrations than AMA-1 because an agent can (directly) move on a distant node in a single migration; as opposed to performing several 1-hop migrations to reach the same destination.

Returning to the previous example of *Fig 1.3*, assume that node n_4 runs AMA-5 for agent a_2. The first iteration will determine that a_2 should migrate (from n_4) to n_2, the second iteration will determine that a_2 should migrate (from n_2) to n_1, the third iteration will determine that a_2 should move on n_3, and finally the fourth iteration will decide that it is not beneficial for a_2 to migrate any further. At this point the algorithm stops, suggesting the migration of a_2 from n_4 to n_3.

Implementation and complexity. AMA-*k* requires the same type of load information as AMA-1 but for all *k*-hop instead of just 1-hop neighbors, rendering g^k the space complexity of AMA-1 (note that a refined, asynchronous, implementation, could store only the loads of the neighbors that are relevant for the computation of each iteration, requiring the same amount of memory as AMA-1). The destination for an agent is chosen in up to *k* iterations, each time evaluating *Eq. 1.6* for the relevant, up to *g*, neighbor nodes, yielding a total time complexity of $O(kg)$ for determining the most beneficial destination for a local agent, i.e., AMA-*k* is *k* times slower than AMA-1.

Convergence. It is straightforward to infer that the algorithm converges provided that race conditions are tackled as per AMA-1.

5 Handling Capacity Constraints

This section discusses how AMA-1 and AMA-*k* can be extended to handle node capacity constraints. When running the algorithms, some assumptions must be made regarding the free capacity of remote nodes to drop infeasible solutions. Notably, these assumptions could be invalid and must be confirmed in order to actually perform a migration. In this work we investigate two different schemes, as follows.

Inquire-Lock Before (ILB). Before running the algorithm, a request is sent to *all potential destinations*, 1-hop or *k*-hop neighbours depending on the algorithm, inquiring about their free capacity and requesting to reserve up to the amount needed to host *all* locally hosted agents that could be selected for migration. Nodes reply with their available free capacity, if any, which they reserve until further notice. The selection of the destination for each locally hosted agent is done as described in the previous subsections, having a *consistent and guaranteed view* of node capacities. When the destinations are chosen, all other nodes are informed to release the reserved capacity, while destinations release the capacity that is left over after accepting the agents assigned to them.

Inquire-Lock After (ILA). The algorithm runs based on a previous, *possibly outdated*, view of free node capacities. Destinations are then contacted to reserve the capacity needed for hosting the agents assigned to them. Initially, all nodes are assumed to have an infinite free capacity. This view, along with the nominal capacity of each node, is updated based on the replies received for each request. To avoid excluding destinations due to outdated information, with a certain probability nodes are assumed to have their full nominal capacity free, independently of the local view. Of course, this means that a migration might be decided based on invalid information, in which case the destination will send a negative reply when contacted to actually reserve capacity (and perform the migration).

Algorithmic adaptations. When AMA-1 picks a destination for a locally hosted agent, the migration is performed only if that node indeed has sufficient free capacity. Else, the agent is not considered for migration because all other destinations are guaranteed to lead to a load increase; *Eq. 1.6* holds for at most one 1-hop neighbor or put in other words there can be at most one beneficial migration direction in a tree network. In contrast, AMA-*k* can fall back to the next best option in that path. For instance, in *Fig 1.3*, n_4 would consider first n_3, then n_1 and finally n_2 as destinations for the migration of a_2. Notably, the destinations chosen by ILB are

guaranteed to be able to host the agents assigned to them, while ILA may pick destinations that turn out not to have sufficient free capacity to host the agent(s) assigned to them.

Notably, **both** schemes are subject to **starvation** due to locking collisions. To reduce the probability of such live-locks, each node invokes the algorithm in random intervals (within a larger time period). The random selection of algorithm invocation, guarantees convergence, but only eventually (the probability that convergence is reached at some point becomes one for infinite time) and without an apriori bound on communication. Convergence can also be achieved more conservatively, by adding a simple rule such as: "stop migration attempts after c collisions", which obviously guarantees convergence, even with "systematic" collisions.

6 Experiments

This section presents an experimental evaluation of the algorithms based on simulations performed on top of NS2 [85]. First we describe the experimental setup and then we present and discuss the results of indicative experiments.

6.1 Setup

Two types of networks are considered with 20 and 50 nodes placed randomly in a 80×80 and 120×120 plane, respectively. Nodes are in range of each other if their Euclidean distance is less than 30. The tree-based routing topology is obtained by calculating a spanning tree over the connectivity graph. Five topologies are generated for each network type. Each experiment is performed on all topologies. The average diameter for the 20- and 50-node networks is 6 and 15, respectively.

The application structure is generated as follows. Starting from an initial set of non-generic (leaf) agents, agents are split in disjoint groups of 5, and for each group 2-5 agents are randomly chosen, removed from the set, and labeled as children of a new generic agent that is added to the set. This process is repeated until the set comprises a single agent which becomes the root (we check to make sure that this is indeed a generic agent). Three application structures are generated with (50, 22), (25, 12) and (10, 5) (non-generic, generic) agents, referred to as app50, app25 and app10, respectively. The initial agent placement on nodes is random.

In terms of application-level traffic, we let each non-generic (leaf) agent send 10-50 messages per time unit to its parent and each generic (intermediate) agent send to its parent the average of the load received from its children (perfect aggregation). Also, each parent agent sends 1 message per time unit to its children (representing a heartbeat protocol). For simplicity, all messages are of equal size. The traffic pattern is stable throughout the whole duration of the experiments.

Nodes invoke the algorithm every T time units. Each node starts its periodic invocation with a different offset, randomly set between 0 and T. If an attempted migration fails due to resource constraints, the node backs-off for a number of periods T, chosen randomly between 1 and 5. Finally, in ILA, the probability for considering a node assuming that its full nominal capacity is free (as opposed to its free capacity according to the local view) is set to 20%.

As the main metric for our comparison, we measure the network load that corresponds to the agent placement produced by the algorithms vs. the load of the initial random placement but also vs. the optimal solution obtained via an exhaustive search algorithm (only for small-scale experiments). For experiments without capacity constraints, convergence is inferred when all nodes invoke the algorithm without attempting any migration. In experiments with capacity constraints, where algorithms employ the ILB or ILA scheme and convergence is not guaranteed, the simulation is stopped when each node invokes the algorithm 4 consecutive times without managing to perform a migration. The overhead of algorithms is captured via the number of agent migrations performed to reach the final placement as well as the number of (control) messages exchanged to avoid swaps and to reserve and release capacity.

6.2 Results without capacity constraints

In a first experiment we compare the placements obtained by the uncapacitated algorithms for the 20-node networks and one app10 application. *Table 1.1* summarizes the results for different degrees of network awareness (average values for the 5 different topologies). All algorithms perform close to optimal, even though the initial random placement is very bad, incurring more than twice the load of the optimal solution.

Table 1.1 Performance in the uncapacitated case (20 nodes, app10).

Algorithm	Total Load	Migrations	Control Msgs
Initial	106,6	-	-
AMA-1	45	10	20
AMA-2	44,4	6,8	13,6
AMA-3	44,8	5,2	10,4
AMA-4	44,8	5	10
Optimal	43,6	-	-

The (slightly) inferior placement achieved by AMA-1 is due to the fact that it forces distant migrations to occur in iterations, moving agents one hop at a time. In the meantime, other agents that communicate with the agent "under migration" might migrate too, leading to a suboptimal lock-in. Greater network awareness reduces the probability of such lock-ins but does not guarantee their absence, e.g., note that AMA-3 and AMA-4 produce a (slightly) worse placement than AMA-2.

As expected, greater network awareness leads to fewer migrations because agents can be placed directly on nodes further away from their original hosts, if desired. Notice that the number of control messages (in this case generated to avoid swaps) equals twice the number of migrations, indicating that no migration was turned down.

6.3 Results with capacity constraints – small scale experiments

In a second experiment, for capacitated nodes, we compare AMA-1 and AMA-2 vs. the optimal solution for the same topology and application as before, for both ILB and ILA schemes. All agents have identical capacity requirements. The results are plotted as node capacity is increased so that each node can host 1, 2, 3 and 4 additional agents compared to the initial placement.

As it can be seen in *Fig* 1.4, all algorithms produce sub-optimal results when node capacity is scarce, but the gap shrinks quite rapidly as capacity becomes abundant, approaching the results of the exhaustive search algorithm. Once again, the placements achieved by AMA-2 are better than those of AMA-1. Somewhat surprisingly, ILB consistently outperforms ILA only for AMA-1 but not for AMA-2. When capacity is tight, AMA-2 produces better results with ILA than ILB, even though ILA works with possibly outdated node capacity information. This can be explained due to the greedy locking approach of ILB which leads to more collisions compared to ILA, as network awareness increases and a node can receive capacity reservation requests from a larger number of nodes.

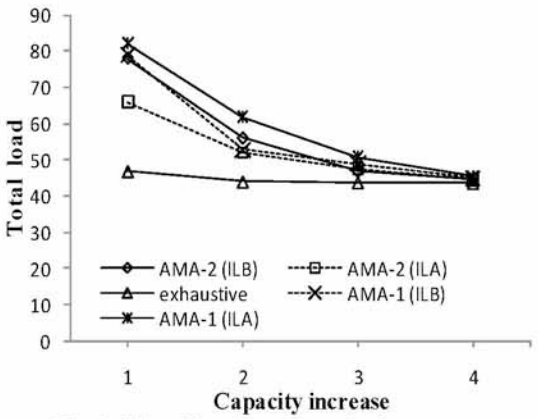

Fig 1.4 Total load vs. capacity increase (20 nodes, app10)

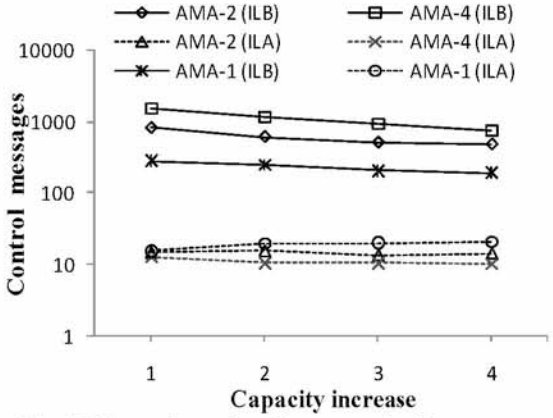

Fig 1.5 Control overhead vs. capacity increase (20 nodes, app10)

Another negative effect of ILB is shown in *Fig* 1.5 which plots the number of generated control messages. ILB clearly incurs a significantly higher overhead compared to ILA, by 1.5–2 orders of magnitude. This is due to the fact that ILB pro-actively inquires about and attempts to reserve free capacity on *all* neighbor nodes within a *k*-hop radius, while ILA mainly relies on information acquired through previous communications and tries to lock *only* the nodes that are actually selected as destinations.

6.4 Results with capacity constraints – large scale experiments

We also performed experiments for the 50-node networks and an application mix of five instances of app10, app25 and app50. We compare the performance of AMA-*k*, for k = 1, 2, 5, 10. Given the bad scalability of ILB, obvious from the previous results, only ILA is used. In the spirit of the previous experiments, the algorithms were tested for the case where each node is capable of hosting 5, 10, 20 and 40 additional agents compared to the initial random placement.

Fig 1.6 and *Fig 1.7* depict the load corresponding to the placements achieved (the initial placements amounted to an average load of 11,000) and the number of migrations performed to reach them, respectively. As expected, greater network awareness results in better placements and fewer migrations. The differences in placement quality are more pronounced for limited capacity and shrink as capacity increases, while the opposite trend holds for the number of migrations. Note that capacity constraints have a greater impact for smaller values of k. This is because, as discussed in Sec. 6.2, low network awareness is more likely to lead to suboptimal lock-ins, but now this may also waist capacity that could have enabled more beneficial migrations. Indeed this effect is more visible when capacity is scarce and diminishes as capacity increases.

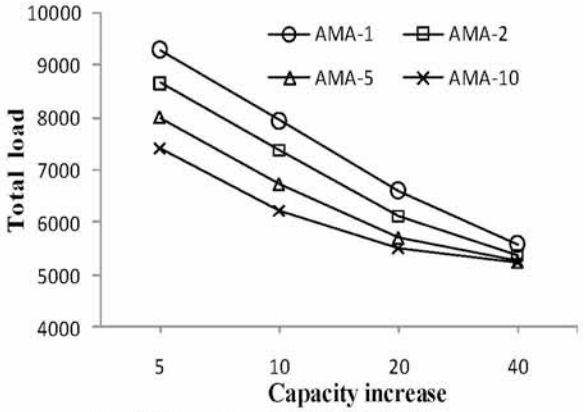

Fig 1.6 Total load vs. capacity increase (50 nodes, application mix)

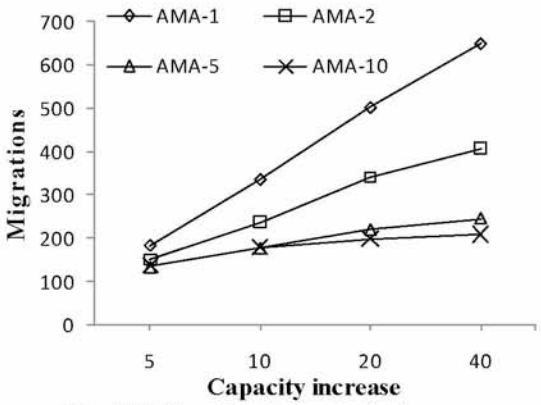

Fig 1.7 Migrations vs. capacity increase (50 nodes, application mix).

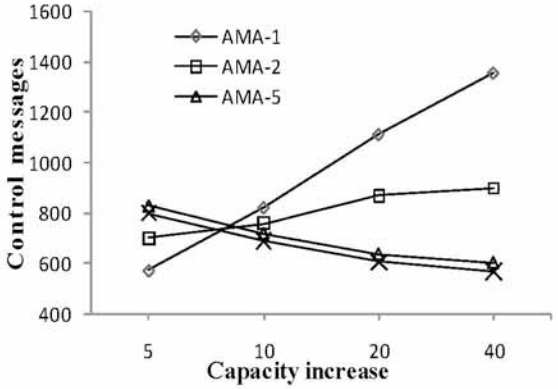

Fig 1.8 Control overhead vs. capacity increase (50 nodes, application mix)

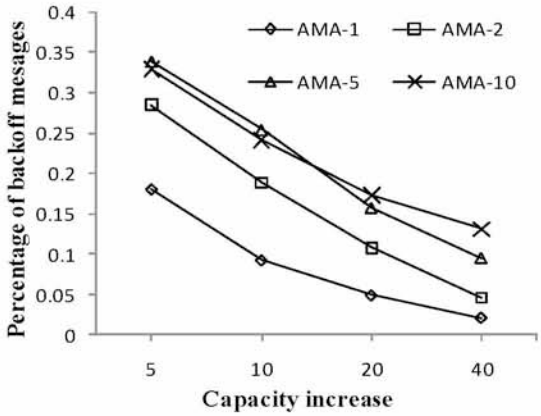

Fig 1.9 Back-offs vs. capacity increase (50 nodes, application mix)

The number of control messages is plotted in *Fig 1.8*. AMA-1 and AMA-2 follow opposite trends compared to AMA-5 and AMA-10, with the first pair incurring less overhead when capacity is tight, but then increasingly more as capacity becomes abundant. This is due to two reasons. On the one hand, the number of migrations, and that of (successful) capacity reservations in ILA, increases more steeply for low network awareness, as shown in *Fig 1.7*. On the other hand, the number of unsuccessful reservations, initially larger for the greater awareness, generally decreases with increasing capacity. This is confirmed in *Fig 1.9* which shows the percentage of control messages that resulted in a back-off. The net effect results in the observed behaviour.

6.5 Result summary

Based on the presented results we can state that: (i) AMA-*k* achieves close to optimal performance when there are no capacity constraints; (ii) with capacity constraints, AMA-*k* considerably improves agent placement from an initial random placement; (iii) greater network awareness leads to better placements while requiring fewer migrations, but this performance advantage shrinks rather quickly for larger values of *k*; (iv) the ILA scheme scales better than ILB, and in fact leads to better placements for increased network awareness when node capacity is scarce.

7 Conclusions

In this work we formulated the problem of placing cooperating mobile agents on nodes as to minimize the network load due to agent-level message traffic under node capacity constraints. We proposed and evaluated corresponding distributed algorithms for agent migration that can take advantage of basic routing-level information. Given their simplicity, these algorithms are suitable for resource constrained embedded systems. AMA-*k* combined with the ILA capacity inquiry and reservation scheme is a particularly attractive candidate since it achieves good results for relatively small (compared to the network diameter) values of *k*, incurring a modest communication overhead and being quite efficient in terms of memory and runtime complexity.

Part of this work has been published in the following conference:

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

Chapter 2

GRAL: A Grouping Algorithm to Optimize Application Placement In Wireless Embedded Systems

1 Introduction

This chapter considers the agent placement/migration problem (introduced in the previous chapter) in a more sophisticated way against the aforementioned simple algorithms. Specifically, it pinpoints the problem induced when having groups of "mutually" dependent agents (communicating heavily with each other), whereby the involved agents are located on their center of gravity in their own right, but not when considered as a whole. Therefore, migrating such a group of agents towards its center of gravity, network load reduction is further achieved.

Of course, the challenge is to identify such unbalanced groups of "mutually" dependent agents and then migrate them towards their center of gravity. To this end, a fully distributed *grouping algorithm* (GRAL) is proposed which considers both single and group agent migrations to minimize the network traffic. Given unlimited general-purpose resources, the algorithm utilizes only information available locally at each node, while in the more realistic constrained case, the resource status of potential destinations must be discovered/estimated.

The contributions of this work include the following: (i) we present two versions of the GRAL migration algorithm each assuming different network knowledge, given unlimited resources at nodes; (ii) we discuss various mechanisms to tackle migrations towards storage/resource constrained nodes; (iii) we evaluate the different approaches through simulation experiments,

comparing their performance against: a) optimal assignment derived through exhaustive search; b) AMA [113] which is an algorithm we have proposed in the previous chapter tackling the same problem in a different approach.

2 Application Model, System Model and Problem Formulation

This section is identical with the respective section of Chapter 1, with the difference that the system model is a little bit further extended as follows. An edge is called *local edge* when its incident agents are co-located, otherwise this edge is named *remote edge*. A collection/group of co-located generic agents is called *non-partitioned* when all the agents participating into that collection are connected with each other through local edges. Let h_{ij} denote the distance in hops between n_i and n_j.

3 Motivation example

Consider the example depicted in *Fig 2.1* where an application of three agents has been deployed into a network of two nodes; with white and black rectangles representing generic and non-generic agents, respectively. The number beside an edge denotes the communication load (per time unit) between the involved agents (e.g. in the example $C_{12}+C_{21}=20$). As it can be observed both a_1 and a_2 are located on their center of gravity, with that placement yielding a cost of 10. However, there is a group of "mutually" dependent agents (a_1, a_2), which is not located on its center of gravity, since the network cost could be reduced at zero if both a_1 and a_2 migrated towards n_1. Recall that this work assumes only generic agents can migrate, hence a_3 cannot migrate towards n_1. It should be stressed that AMA doesn't consider group migrations, thus we propose an algorithm tackling this case.

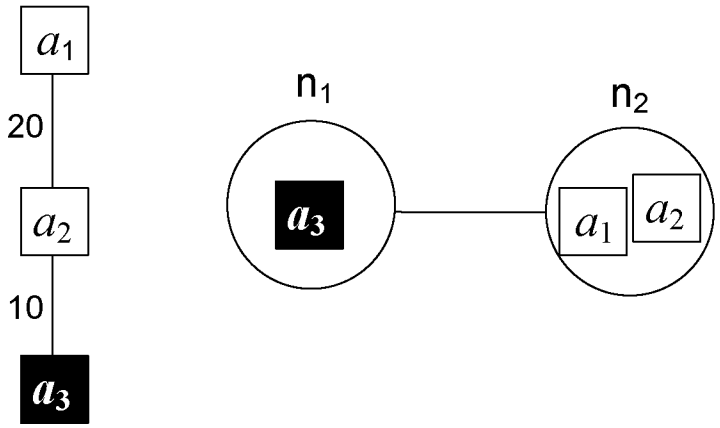

Fig 2.1 Application placement

4 GRAL Migration Algorithm

This section presents GRAL for the case where nodes can host any number of agents without taking into account capacity limitations. In terms of routing information, a node knows only its immediate (1-hop) neighbours involved in transporting both inbound and outbound agent messages. This information can be typically provided by even a simple networking layer. GRAL is a completely different approach against AMA [113], with the former considering migrations in a grouping manner taking into account agent dependencies, in contradistinction to latter where the migrations are performed in a single agent fashion.

4.1 Beneficial single agent migrations

GRAL performs single agent migrations in the same way as AMA algorithm (described in the previous chapter).

4.2 Beneficial group migrations

The algorithm first identifies disjoint application sub-trees hosted locally, and for each sub-tree produces a group (that may be a subset of the sub-tree). For each group, a single destination is chosen as a host for all agents that are part of the group. More specifically, the algorithm works in several steps, as follows:

Sub-tree Identification. First, one or more disjoint sets of communicating locally-hosted generic agents (belonging to the same application) are identified. Each such set corresponds to a part of the application tree, henceforth referred to as a *sub-tree*. Specifically the sub-tree identification takes place as follows: i) create a sub-tree rooted on a locally-hosted generic agent not belonging to an already identified sub-tree; ii) add to this sub-tree each locally-hosted generic agent adjacent to one of the agents belonging already to this sub-tree. Repeat phase (ii) till no agent can expand this sub-tree. After the expansion of a sub-tree completes, repeat phase (i) and (ii) accordingly, till all generic agents have been considered. Note that each sub-tree consisting of only one agent is discarded, since this agent will be considered by the single agent migration mechanism. Observe that each of the remaining sub-trees is a non-partitioned collection of co-located generic agents.

Selection of destination. For each sub-tree, the most *promising 1-hop destination node* is determined by comparing the load between subtree's agents and that node versus all other neighbours, as well as the load with (immobile) locally hosted non-generic agents. Let $l_{ijk}(A)$ and $l_{ijk}(S)$ denote the components of l_{ijk} due to the local communication of a_k with generic respectively non-generic agents hosted at n_j. Then, both *Eq. 2.1* and *Eq. 2.2* must hold true to select n_j as a destination for a subtree G hosted at n_i:

$$\sum_{\forall k:a_k \in G} l_{ijk} > \sum_{\forall k:a_k \in G} l_{ixk} \mid h_{ij} = 1 \wedge h_{ix} = 1, \forall x \neq i, j \qquad \text{Eq. 2.1}$$

$$\sum_{\forall k:a_k \in G} l_{ijk} > \sum_{\forall k:a_k \in G} l_{iik}(S) \mid h_{ij} = 1 \qquad \text{Eq. 2.2}$$

Namely, *Eq. 2.1* says that the aggregate load between the agents of the sub-tree and the destination n_j should be greater than the respective load for any other neighboring node. The aforementioned aggregate load involves the data exchanged between the agents of the sub-tree and the agents using n_j as either a hosting or routing node to communicate with the former ones .While *Eq. 2.2* says that this load should be also greater than the locally incurred one due to the communication with (immobile) non-generic agents hosted at n_i.

Partial benefit calculation. Having chosen the best promising 1-hop destination pn_j, the respective *affinity* and *partial benefit* value is computed for each agent a_k of the sub-tree:

$$aff_{ijk} = l_{ijk} - l_{iik}(S) - \sum_{\forall x \neq i,j} l_{ixk} \mid h_{ij} = 1 \wedge h_{ix} = 1 \qquad \text{Eq. 2.3}$$

$$pb_{ijr} = aff_{ijr} - l_{iir}(A) \mid h_{ij} = 1 \qquad \text{Eq. 2.4}$$

$$pb_{ijm} = aff_{ijm} - l_{iim}(A) + 2(C_{vm} + C_{mv}) \mid pr_m^v = 1 \wedge h_{ij} = 1 \quad \text{Eq. 2.5}$$

The affinity aff_{ijk} is equal to the load associated with a_k for pn_j minus (i) the local communication load in terms of local (immobile) non-generic agents; and ii) the respective load for all other neighbors. It provides an upper bound on the positive impact the migration of a_k from n_i to n_j can have, provided that the entire subtree moves to n_j. If all agents have negative affinity then no beneficial group migration exists within the subtree. Else, the partial migration benefit is calculated for each agent in a top down fashion. *Eq. 2.4* is used to calculate the partial benefit of the root a_r of the sub-tree, which corresponds to the benefit if only a_r migrates to pn_j while all other agents of the subtree it communicates with (i.e. its children) remain on n_i. To calculate the partial benefit of every other agent a_m of the subtree we make use of *Eq. 2.5*, with pr_m^v being equal to 1 if a_v is the parent of a_m (in terms of that subtree), otherwise 0. Specifically, this equation corresponds to the load impact if both a_m and its parent a_v migrate to pn_j while all other agents a_m cooperates with (i.e., its children) remain on n_i.

By construction, these values can be used to calculate the actual benefit obtained by migrating on pn_j any part of the subtree. Specifically, the actual benefit for migrating any agent a_m together with all its predecessors (in the path) up to the root a_r is equal to the sum of the respective partial migration benefit values. Also, the benefit of migrating any agent a_m together with all its predecessors up to agent a_u $(u \neq r)$ is equal to the sum of the partial benefits minus two times the load between a_u and its parent (that does not belong to the part being considered for migration).

Group Selection. The algorithm processes the subtree by merging leafs with their parent in a bottom-up fashion. Each merge produces a so-called *group* node with a respective migration benefit. The best grouping combination is recorded and updated correspondingly. Nodes with a negative benefit value are pruned. The grouping phase terminates when a single group node remains, and the best grouping is returned.

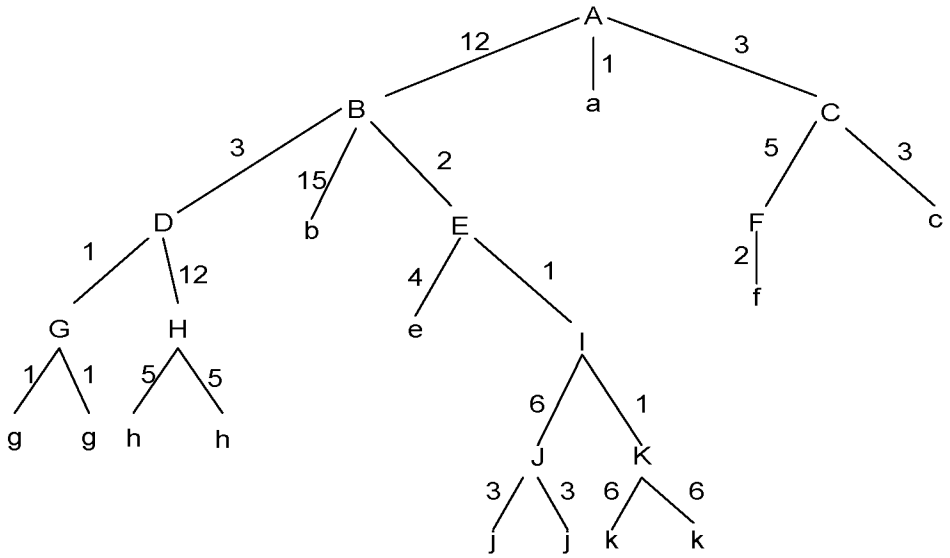

Fig 2.2 Application structure

As an example, consider the application tree shown in *Fig 2.2*, where generic agents are denoted in capitals and (multiple instances of) non-generic agents in small case letters. Edge values stand for the communication load between two agents.

Let the application be deployed on a network as illustrated in *Fig 2.3*. Two disjoint subtrees are hosted at n1: *(A, B, C, D, F, G, H)* and *(I, J, K)*, hence two groupings will be produced, one for each subtree (note that AMA cannot improve the placement depicted in *Fig 2.3*).

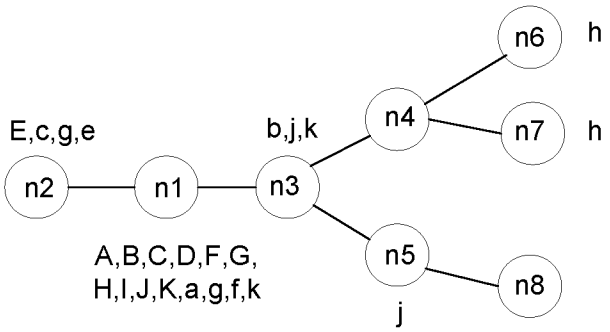

Fig 2.3 Agent placement

In the sequel we illustrate this process for the first sub-tree *(A, B, C, D, F, G, H)*. *Table 2.1* gives the relevant load components for these agents, i.e., the load coming from each neighbour of n_1, i.e., n_2 and n_3, together with the load from n_1 itself (local load). The last load is split into the load due to communicating with generic agents (n_1(A)) and the load due to communicating with non-generic agents (n_1(S)). For instance, [*C*, n_1(A)] is 8 due to the local communication with generic agents *A* and *F* on n_1, [*F*, n_1(S)] is 2 due to the local communication with non-

generic agent *f* on n_1, and [*H*, n_3] is 10 due to the remote communication with agents *h* on n_6 and n_7 (via n_4).

Table 2.1 Load components

Agent	n1(A)	n1(S)	n2	n3
A	15	1	0	0
B	15	0	2	15
C	8	0	3	0
D	16	0	0	0
F	5	2	0	0
G	1	1	1	0
H	12	0	0	10

First, the destination for *(A, B, C, D, F, G, H)* is chosen. The two possible options are n_2 and n_3. Based on the given loads, the best destination as per *Eq. 2.1* and *Eq. 2.2* is n_3, since it accounts for an aggregated load of 25 as opposed to 6 for n_2, and this load is greater than the total local load incurred between the entire sub-tree and the non-generic agents hosted at n_1, which is equal to 4.

Then, the partial benefits are computed by starting from the root of the subtree, in this case A which has an affinity of -1 ($aff_{13A} = l_{13A} - l_{11A}(S) - l_{12A} = 0 - 1 - 0 = -1$). The partial migration benefit of A as per *Eq. 2.4* is –16 ($pb_{13A} = aff_{13A} - l_{11A}(A) = -1-5 = -16$). The partial benefits of all other agents are calculated as per *Eq. 2.5*; for instance this is 22 for *B*: ($pb_{13B} = aff_{13B} - l_{11B}(A) + 2*(C_{BA} + C_{AB}) = 13 - 15 + 12*2 = 22$). The results are shown in *Fig 2.4* with node values denoting the respective partial migration benefits. The sub-tree is then processed to produce the best grouping option. *Fig 2.5* depicts the result of the first iteration, which leads to the creation of group nodes DH and CF.

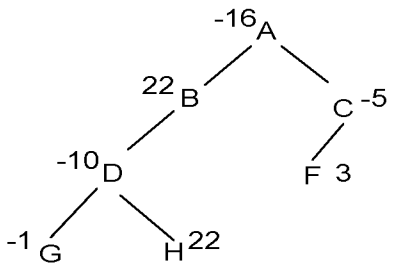

Fig 2.4 Tree construction phase for group (A, B, C, D, F, G, H)

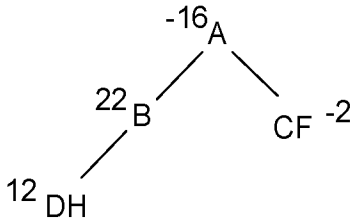

Fig 2.5 Tree contraction phase

Notice that G was pruned since it had negative partial benefit. In the second iteration CF will be pruned and BDH created with a benefit of 34. Finally, A will be merged with BDH and the resulting group node ABDH will have an actual benefit of 18.

Convergence. Notably, for the time being the algorithm *does not* guarantee convergence as it is susceptible to race conditions. Consider for instance two communicating node neutral agents residing at neighboring nodes. If the load between them is high enough and the nodes invoke the algorithm with the same period it is possible that these agents will swap places at one period, only to re-swap back to their original positions at the next period and so on so for.

We expect such live-locks to be rare in practice, especially if neighboring nodes invoke the algorithm at random intervals. To guarantee convergence though, we introduce a coordination scheme between nodes in the spirit of a mutual exclusion protocol. Namely, when n_i decides to migrate a *group* to n_j it asks n_j for a permission. In turn, n_j rejects such a request if all the following is true: (i) it hosts an agent a_k that is the child or parent of an agent belonging in the group to be transferred, (ii) it has decided to migrate a_k to n_i and has requested a respective permission, and (iii) the id of n_j is smaller than that of n_i *(j<i)*. Else, n_j grants permission to n_i and does not consider migrating any agent to n_i which has parent or child relation with an agent of the group in question till the later completes its migration. Convergence is guaranteed because conflicting migrations cannot be performed concurrently and each (non-conflicting) migration reduces the network load.

Complexity. For each locally hosted generic agent, one needs to record the load with each neighbor node as well as the load aggregates for local generic and non-generic agents. This requires a *A'x(N'+2)* table, where *A'* and *N'* is the number of local generic agents and neighbors, respectively, in the spirit of *Table 2.1*. In addition, parent-child loads must be recorded for each pair of locally hosted cooperating generic agents. This can be done via a separate tree structure for each subtree, with pointers to the respective locations of the load table, requiring *O(A')* memory in total.

The destination for each subtree can be chosen in one pass of the corresponding tree structure and respective load table entries, in *O(A'N')* for all subtrees. The calculation of the affinity and partial benefit values requires one more pass. Similarly, the grouping of each subtree can be done in a single pass of the tree structure in O(A'), while the best grouping combination can be updated in *O(1)* for each step. Hence the asymptotic time complexity of GRAL is *O(A'N')*.

5 Handling increased network knowledge

In this section we consider the case where each node not only knows its immediate neighbors, but also every node within *k*-hops. Such *k*-hop information may be collected without significant extra communication, e.g., by (occasionally) piggybacking node identifiers as a message travels through the network, or by employing a naming scheme that directly encodes path information into node identifiers as done in ZigBee for the case of hierarchical routing [129]. We proceed by presenting a variation of GRAL that explores such increased knowledge, referred to as GRAL-k. GRAL-k extends GRAL to: (i) take advantage of *k*-hop awareness, and (ii) potentially assign different parts of the group to different destinations (i.e., suggesting that some agents of the group migrate to different nodes).

For each subtree G, all neighbors within k hops of the local host and which are involved in the load associated with G are considered as potential destinations. The respective affinity and partial benefit values for each destination node n_i are calculated in the spirit of GRAL, however Eq. *2.3*, Eq. *2.4* and Eq. *2.5* are adjusted to consider the fact that n_j need not be a neighbour of n_i:

$$aff_{ijm} = l_{ijm}h_{ij} - l_{iim}(S)h_{ij} - \sum_{\forall x \neq i,j : D_{ijx}=0 \wedge h_{ix}=1} l_{ixm}h_{ij} - \sum_{\forall x \neq i,j : D_{ijx}=1} (l_{ixm} - l_{ium})(h_{jx} - h_{ix}) \,|\, D_{xju} = 1 \wedge h_{xu} = 1 \wedge u \neq i,j \quad \text{Eq. 2.6}$$

$$pb_{ijr} = aff_{ijr} - l_{iir}(A)h_{ij} \quad \text{Eq. 2.7}$$

$$pb_{ijm} = aff_{ijm} - l_{iim}(A)h_{ij} + 2(C_{vm} + C_{mv})h_{ij} \mid pr_m^v = 1 \quad \text{Eq. 2.8}$$

Where in *Eq. 2.6* n_u is the next hop node in the path from n_x to n_j, in *Eq. 2.7* a_r is the root of the sub-tree, and in *Eq. 2.8* a_v is the parent of a_m in the sub-tree. Recall, that the affinity aff_{ijm} represents the benefit of migrating am from n_i to n_j assuming the entire sub-tree also moves on n_j. Once again, three load components are considered: (i) the load associated with a_m that goes through n_j, minus (ii) the load due to the local communication with non-generic agents hosted at n_i and (iii) the additional load going through other nodes n_x. The first two components remain the same as in *Eq. 2.3*, multiplied by the hop distance between n_i and n_j. The third component now comprises two terms, handling two different cases. If n_x is a neighbor of n_i in a different outbound direction than n_j, the load for n_x remains the same and is multiplied by the extra distance travelled (the distance between n_i and n_j). Else, if n_x is between n_i and n_j, the additional penalty is the difference between the load for n_x and the next hop node n_u towards n_j multiplied

by the corresponding hop difference; this actually corresponds to a benefit, if n_x is closer to n_j than n_i. Note that *Eq. 2.6* maps to *Eq. 2.3* in case n_j is a neighbour of n_i (the last term disappears). The partial benefit formulas *Eq. 2.7* and *Eq. 2.8* are straightforward extensions of *Eq. 2.4* and *Eq. 2.5*, taking into account the distance between n_j and n_i.

The partial benefit values for each agent and destination node can be stored using a single tree structure, where the partial benefit of an agent is a vector; each element indicating the partial benefit for a different destination node. The grouping process follows the same principle as in GRAL, but when merging two nodes the best destination for the leaf is selected for each destination option of the next-level node, producing an equal number of combined placements and partial benefit values for the resulting group node. The most beneficial vector's entry of final contracted node is chosen.

During the grouping phase, the benefit values are calculated based on the fact that each merge "links" the parent agent a_m in the leaf node with the parent agent a_n in the next-level node (a_m is the child of a_n, both hosted at n_i). Let t_m and t_n denote the nodes (forming a sub-tree) that contain these agents, and $t_{n,m}$ denote the node that results after merging t_m with t_n. Also, let pbt_{ium} be the partial benefit of t_m if a_m moves from n_i on n_u, and pbt_{ivn} the partial benefit for t_n if a_n moves from n_i on n_v. Then the corresponding combined partial benefit for $t_{n,m}$ is:

$$pbt_{ivn,ium} = pbt_{ivn} + pbt_{ium} - (C_{mn} + C_{nm})(-h_{iv} + h_{iu} + h_{uv}) \mid pr_m^v = 1 \qquad \text{Eq. 2.9}$$

To explain the third term, recall that pbt_{ivn} is calculated as per *Eq. 2.7* assuming that a_m (a child of a_n) remains at n_i while, a_n moves from n_i on n_u, and pbt_{ium} is calculated as per *Eq. 2.8* assuming that a_n (the parent of a_m) will also migrate from n_i on n_u together with a_m. If this is not the case *($v \neq u$)*, the benefit must be adjusted by (i) crediting the cost $(C_{mn}+C_{nm})h_{iv}$ assumed in *Eq. 2.7*, (ii) subtracting the benefit $(C_{mn}+C_{nm})h_{iu}$ assumed in *Eq. 2.8*, and (iii) subtracting the load $(C_{mn}+C_{nm})h_{uv}$ that will actually be incurred between a_m and a_n from their new hosts. Note that the third term disappears for $v=u$ in which case the partial benefit of $t_{n,m}$ equals the sum of the individual partial benefits, as usual.

We illustrate how the algorithm works by revisiting the previous example (of *Fig 2.2*and *Fig 2.3*) for $k=2$. Assume that n_1 invokes the algorithm for the sub-tree *(I, J, K)*. The candidate destination nodes, involved in the message traffic associated with one or more agents of this sub-tree, are n_2, n_3 and n_5 (n_4 incurs no load and is omitted). *Table 2.2* lists the load and affinity value for each agent and destination candidate. For instance, [*J*, l_{15}] is 3 due to the

communication with agent *j* on n_5, and [*J*, aff_{15}] is 6 ($aff_{15J} = l_{15J}h_{15} - l_{11J}(S)h_{15} - l_{12J}h_{15} - (l_{13J} - l_{15J})(h_{13} - h_{35}) = 3*2 - 0*2 - 0*2 - (6-3)*(1-1) = 6$).

Table 2.2 Load coefficients for the subtree

Agent	(l_{12}, aff_{12})	(l_{13}, aff_{13})	(l_{15}, aff_{15})
I	(1, 1)	(0, -1)	(0, -2)
J	(0, -6)	(6, 6)	(3, 6)
K	(0, -12)	(6, 0)	(0, -12)

Fig 2.6 depicts the initial state of the sub-tree where each node is associated with 3 different partial benefit values, one for each of the candidate destinations nodes (listed below the respective values). For instance, the partial benefits for n_5 are: –16 for *I* ($pb_{15I} = aff_{15I} - l_{11I}(A)h_{15} = -2 - (6+1)*2$) as per *Eq. 2.7*; 18 for *J* ($pb_{15J} = aff_{15J} - l_{11J}(A)h_{15} + 2(C_{JI}+C_{JI})h_{15} = 6 - 6*2 + 2*6*2$) as per *Eq. 2.8*; –10 for *K* ($pb_{15K} = aff_{15K} - l_{11K}(A)h_{15} + 2(C_{KI}+C_{KI})h_{15} = -12 - 1*2 + 2*1*2$) as per *Eq. 2.8*.

Each merge produces 3 combinations whereby each agent is separately assigned to a destination. *Fig 2.7* shows the result of merging tree node *K* with *I* into a group node *IK*. Put in other words, if *I* migrates on n_2 the best destination for *K* is n_3 yielding a combined partial benefit of –7, if *I* moves on n_3 the best destination for *K* is n_3 with a partial benefit of –7, and if *I* moves on n_5 the best destination for *K* is n_3 with a benefit of –15. The vector of final contracted node *IKJ* becomes <11, 11, 3>, with the agent assignment on nodes being < (n_2, n_3, n_5), (n_3, n_3, n_5), (n_5, n_3, n_5)>. Hence, the algorithm will choose either the first or second entry, since both carry the same migration benefit. For instance, if the first entry is chosen as the most beneficial one, this means that *I*, *K*, *J* will migrate to n_2, n_3, n_5, respectively; with the actual migration benefit being 11.

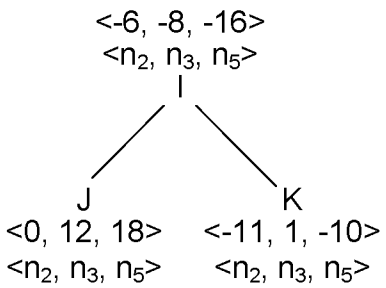

Fig 2.6 Tree construction phase

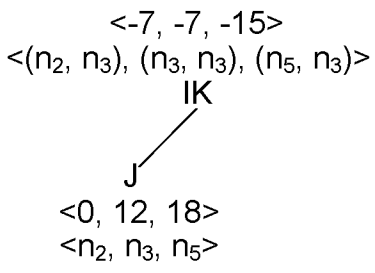

Fig 2.7 Tree contraction phase

In terms of space complexity, for each group node the partial benefit vector is *O(N')* in size, and $O(N'^2)$ of space is required to store the various placement combinations, where *N'* is the number

of nodes that are k hops away from the local node. Thus the aggregate space complexity is $O(A'N'^2)$. The time complexity of the algorithm is dominated by the grouping phase because each merge involves calculating the partial benefits of $O(N'^2)$ combinations, each individual calculation done in $O(1)$. This yields a total of $O(A'N'^2)$ for all locally hosted subtrees.

6 Handling capacity constraints

This section discusses how GRAL can be extended to tackle node capacity constraints. In a nutshell, four main elements must be added: (i) infeasible migrations must be dropped; (ii) the available free capacity of nodes must be "discovered" dynamically; (iii) capacity reservations must be made before initiating a migration; (iv) we keep in a special vector the most beneficial feasible merged node along with its actual migration benefit when considered as a standalone entity.

GRAL checks the capacity constraint during the grouping phase of a subtree. If a leaf contains agents that exceed the capacity of the destination, it is pruned. When running the algorithm, some assumptions must be made regarding the free capacity of remote nodes. These assumptions are then used to drop infeasible solutions. Obviously, these assumptions may be invalid and must be confirmed in order to actually perform a migration. In terms of (iv), each time two nodes are merged, we update the special vector to keep the merged node with the best actual migration benefit. Finally, when the contraction phase completes, the merged node with the best actual migration benefit is returned. The motivation behind this is that the finally merged node may be an infeasible solution, therefore in that way we are able to choose the most beneficial feasible solution. The algorithms are enhanced with the two locking schemes proposed in the previous chapter (ILA and ILB).

7 Experiments

The setting for the experimental setup took place in the same way as in Chapter 1.

7.1 Results without capacity constraints

In a first experiment we compare the placements obtained by the GRAL and AMA variants, and the optimal algorithm without taking into account capacity constraints. Due to time complexity owed to the exhaustive algorithm, we choose small-scale experiments (20-nodes, app-10).

Table 2.3 Peformance for Lavg and the 20-node network

Algorithm	Total Load	Migrations	Control Msgs	Conv. Time
initial	173.6	N/A	N/A	N/A
AMA-1	65.6	10.4	20.8	2.4
GRAL-1	58.2	14.8	22.8	2.6
AMA-2	61	7	14	2.4
GRAL-2	58.2	9.8	15.6	2.4
optimal	58.2	N/A	N/A	N/A

Table 2.3 summarizes the results of the aforementioned algorithms for an initial (random) placement and the *lavg* model. The first observation is that the initial placement is quite bad, incurring more than twice the total load of the optimal solution. In fact, both grouping variants GRAL-1 and GRAL-2 consistently achieve an optimal result. In case of AMA, the 2-hop variant produces better placements than the 1-hop variant, illustrating that in this case greater network awareness is less prone to suboptimal lock-ins compared to lower awareness. This is because the latter must perform hop-by-hop migrations in order for an agent to reach its final destination, while former can transfer it through 2-hop jumps.

As expected, the 2-hop variants perform a smaller number of migrations compared to their 1-hop counterparts, because they allow agents to move further away from their original hosts in a single migration. The grouping migration (GM) algorithms result in more migrations than their AMA counterparts, hinting to the fact that grouping avoids suboptimal lock-ins to which single agent algorithms are vulnerable. Further attesting to this fact is the observation that GM algorithms exhibit a lower control message per migration ratio than AMA, showing that it was indeed possible to form groups assigned to the same destination (2 control messages are needed per destination due to the protocol for avoiding swaps). In absolute numbers, however, GM algorithms result in slightly more control messages than AMA algorithms, which is due to the

larger number of migrations performed. A final observation is that the algorithms have much the same convergence time (~2.5 periods).

The same experiment was conducted for the rest load models (*lsum* and *lmix*), with the optimal algorithm being marginally better than GRAL variants. Specifically, the optimal algorithm achieved 1% and 5‰ better performance against GRAL over *lsum* and *lmix*, respectively. However, it was interesting to notice that AMA variants did not manage to bear fruit in *lsum* and *lmix* load models, yielding an enough inferior performance of 30% and 22%, respectively, against grouping variants. This is due to the fact that in these models the "bonds" among the relative agents become stronger, especially when the load between generic agents and their parents is relatively heavy; exposing in that way the drawback of considering migrations in a single agent manner, like AMA variants do. Hereafter, in the experiments we will always be using the *lavg* model.

Finally, the experiment was also repeated for the special case where *all generic agents are initially placed on the same node* (chosen randomly). GM algorithms once more achieved very good results, close to optimal (compared to the exhaustive algorithm). AMA algorithms were particularly bad due to their inherent lock-in problem.

7.2 Small-scale experiments

In the second set of experiments we compare the performance of the AMA-1, AMA-2, GRAL-1 and GRAL-2 algorithms for the ILB and ILA schemes versus the optimal solution obtained by exhaustive search. To reduce simulation time (for the exhaustive algorithm), we choose again the experimental setup to consist of a 20 node network and app10. The evaluation is performed for varying levels of "tightness" of the capacity constraint. More specifically, we start with the nodes having just enough capacity to store the agents defined in the initial placement and add additional capacity to hold 1, 2, 3, 4 extra agents at each node.

Fig 2.8 depicts the percentage of load reduction achieved by the 1-hop variants. All algorithms reduce significantly the load by more than 40% even in the case where the capacity constraint is tight. Comparing against exhaustive search, we notice that the performance difference between the algorithms and the optimal solution rapidly decreases as more capacity becomes available. For instance, with surplus capacity of 1 the difference between GRAL-1 (ILB) and the optimal result is more than 15% while with a surplus capacity of 4 it is less than 5%. This is due to the fact that when capacity is scarce it is also more likely that nodes will be filled. A filled node

essentially acts as a bottleneck separating the tree network into two parts. For 1-hop algorithms this means that these network parts cannot exchange agents; therefore, it is more likely to reach a suboptimal solution. Obviously, exhaustive search doesn't perform any real migrations in order to find the optimal solution, thus doesn't suffer from the effects of bottleneck nodes. The increase in load reduction for exhaustive search should be attributed to the "generally improved" optimization potential as the capacity of nodes increases and the setting gradually shift towards the unconstrained case.

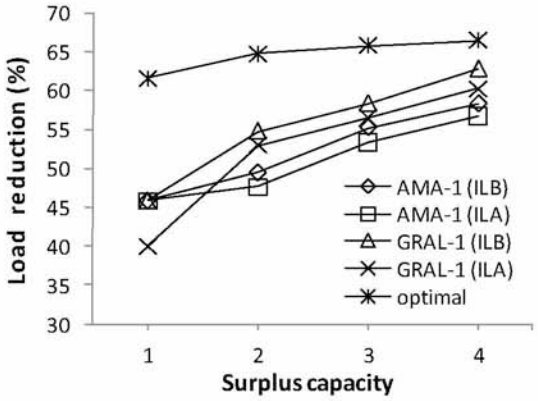

Fig 2.8 Load reduction
(20 nodes, app-10)

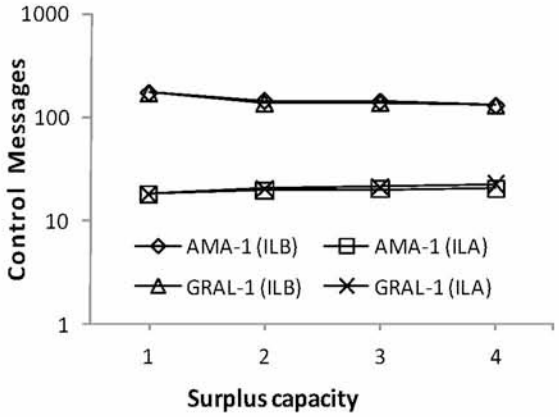

Fig 2.9 Control messages exchanged
(20 nodes, app-10)

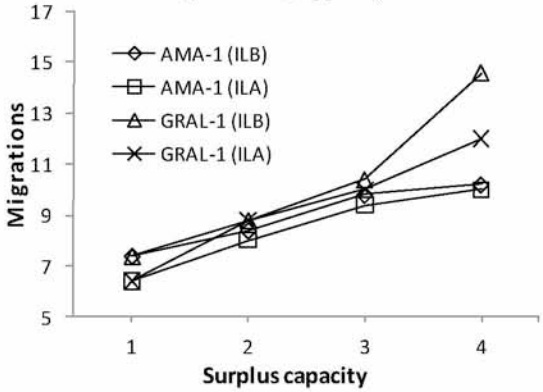

Fig 2.10 Migrations performed
(20 nodes, app-10)

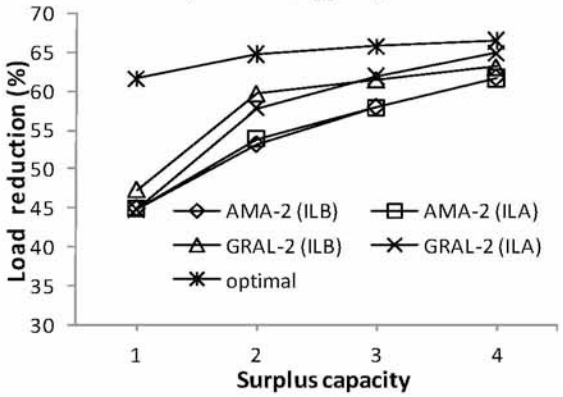

Fig 2.11 Load reduction
(20 nodes, app-10)

Concerning the relative performance of the 1-hop algorithms in *Fig 2.9* we can observe the following: (i) ILB achieves better placements than ILA for both AMA-1 and GRAL-1, and (ii) GRAL-1 consistently outperforms AMA, except in the case of ILA and surplus capacity of 1. Both observations are due to the fact that ILA works with estimates about the free capacity of nodes, thus it may be impossible to perform the decided migrations. While through each failed migration (and capacity reservation) attempt ILA updates its capacity information, this also leads to a back-off. This delay might prove vital since in the meantime a bottleneck node could

be created. This particularly affects GRAL-1 (ILA) because group migrations (two or more agents destined for the same node) are more likely to fail due to outdated capacity information when capacity is tight. AMA-1 (ILA) is less vulnerable to this effect because it only considers single agent migrations. However, it is worth noting that the negative performance impact of ILA in both AMA-1 and GRAL-1 applies only when capacity is scarce and diminishes when capacity increases.

Unfortunately, the performance of the ILB scheme comes at a non-negligible cost. *Fig 2.9* plots the control messages generated in order to reserve capacity and avoid swaps. It can be seen that ILB requires roughly one order of magnitude more messages compared to ILA. This is because ILB greedily attempts to obtain locks from all neighbors, before running the actual algorithm that determines the destinations for agent migrations, i.e., regardless whether these nodes will be chosen as migration targets or not. On the contrary, ILA tries to lock capacity only at the nodes that have been selected as destinations for one or more agent migrations. It is also worth noting that AMA-1 and GRAL-1 generate roughly the same amount of control messages. Another interesting observation is that the number of control messages for ILB tends to decrease as capacity increases. This is attributed to the fact that with larger free capacity a larger number of migrations will succeed without experiencing back-offs or lock-ins due to filled nodes and it is more likely to reach a good placement where agents will not need to move away from their hosts. This is in line with *Fig 2.11* which plots the number of migrations. As it can be seen, the number of migrations rises as capacity increases. It can also be seen that when capacity becomes abundant, the GM algorithms are able to perform more migrations than the SAM algorithms which suffer from lock-ins.

Fig 2.11 shows the results for the 2-hop variants, i.e., AMA-2 and GRAL-2. Most of the general trends discussed for the 1-hop variants hold here too, so we choose to not show the figures about control messages and migrations. Note, however, that the performance difference between ILA and ILB becomes minimal for both AMA-2 and GRAL-2. This is a very encouraging result considering the fact that ILB is very expensive in terms of control messages. To explain this note that with 2-hop network awareness the number of capacity reservation conflicts for ILB increases as a node can receive requests from a larger number of nodes. Thus, it is likely that some agents will not migrate to the destination(s) assigned to them but rather to a (less optimal) one hop neighbor of it, or not at all. This induces a similar effect to the one observed for ILA for 1-hop awareness. Namely, once back-offs occur and agent migrations are delayed, node capacity may be filled with other agents thereby hindering migrations that would be more

beneficial overall. In fact, notice that ILA quickly closes on and eventually overtakes ILB for GRAL-2 as capacity increases, approaching an optimal result.

7.3 Large-scale experiments

In this set of experiments we generated networks of 50 nodes. Also, we deploy 5 applications for each application structure (app-10, app-25, app-50) to synthesize a mix of 15 applications (app-mix). The algorithms being evaluated remain the same (excluding optimal) under a different range of surplus capacity (2, 5, 10, 20). The performance of the algorithms in this setting a little bit different compared to the previous one; specifically the load reduction (*Fig 2.12*) ranges between 10 and 50 instead of 40 and 60 percentage units, respectively. This is because in the previous setting, the proportion of agents needing migration per total surplus capacity is less than this one, thus leading in a more tight placement, and therefore in less migrations. Also, taking a look at the differences of locking schemes between *Fig 2.12* and *Fig 2.8*, we notice that ILB scheme deteriorates with the increase of the network topology (we attribute this to the increasing reservation conflicts).

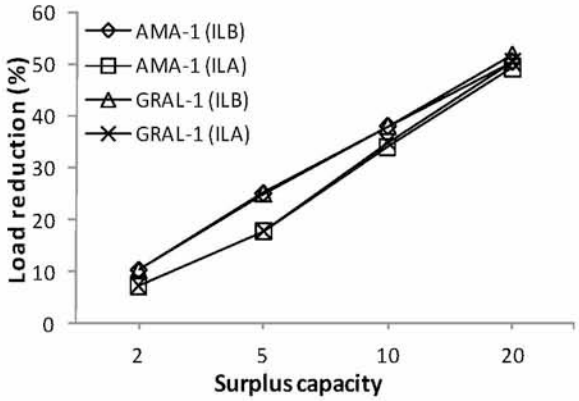

Fig 2.12 Load reduction
(50 nodes, app-mix)

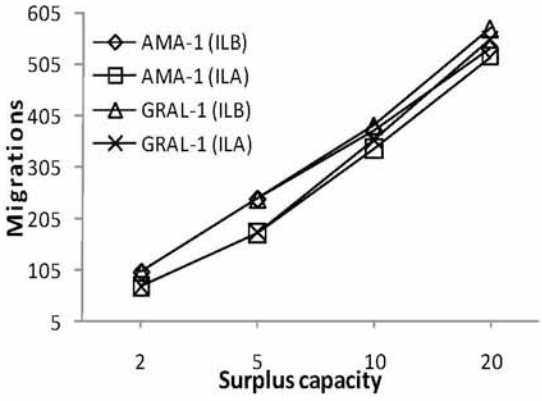

Fig 2.13 Migrations performed
(50 nodes, app-mix)

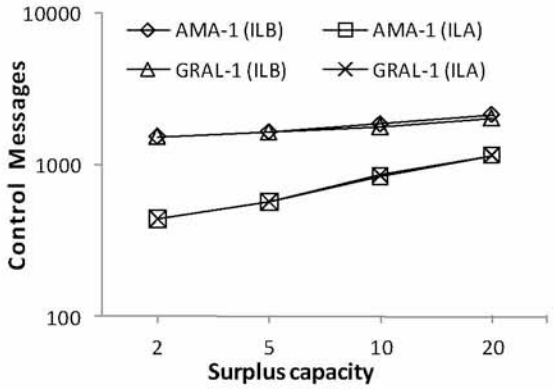

Fig 2.14 Control messages exchanged (50 nodes, app-mix)

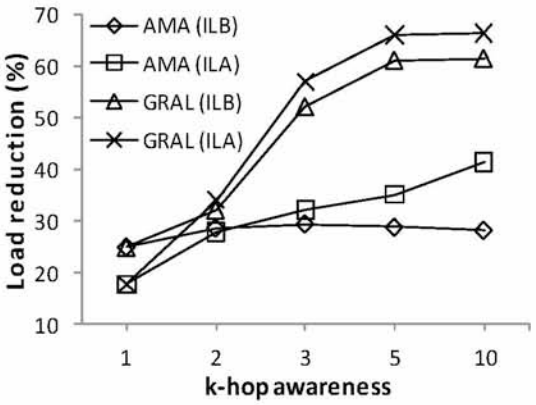

Fig 2.15 Load reduction (50 nodes, app-mix)

In terms of control messages, we notice in *Fig 2.13* that ILB continues sending much more messages against ILA, with ILA having a more steep inclination compared to ILB. This is elaborated through the following remarks: i) the number of migrations increases in a linear fashion (*Fig 2.14*) and the number of control messages, concerning ILA, are exactly twice the number of migrations (request/reply messages); b) in ILB scheme, the number of locking messages is amortized as the number of parallel migrations per node increases, since a node will not send double reservation messages in case it tries to migrate concurrently more than one agents. Summing up the aforementioned remarks, the control messages sent over ILA are linear to the number of migrations, in contradistinction to ILB which is not the case as discussed earlier. Therefore the aforesaid remarks explain ILA's bland increase. Also, due to the fact that we have observed a much similar behavior between 1-hop and 2-hop variants we chose to not show the figures accounting for the control messages and migrations.

For the last experiment we fix the surplus capacity to 5 agents per node and vary nodes' awareness to be between 1 and 10 hops. As we can see in *Fig 2.15*, GRAL achieves by far the better performance compared to AMA, in both ILA and ILB schemes. More specifically, for k=5 GRAL achieves roughly double the load reduction of AMA. The reduction itself is also quite impressive (roughly 65% of the random initial placement). Notably, the load reduction increases rapidly as hop awareness increases from 1 to 2 and 3 hops, stabilizing from 5 onwards. This means that modest network awareness (in this case, 1/3 of the network diameter) is sufficient to reach good solutions, which is also quite important considering the corresponding memory and runtime complexity implications.

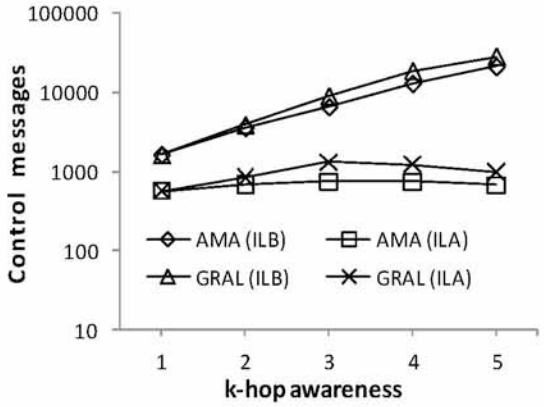

Fig 2.16 Control messages exchanged (50 nodes, app-mix)

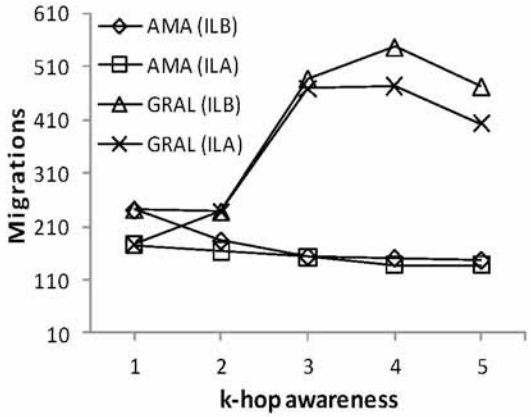

Fig 2.17 Migrations performed (50 nodes, app-mix)

Concerning the capacity reservation schemes, ILA clearly outperforms ILB for both AMA and GRAL with the difference becoming more pronounced as network awareness increases. *Fig 2.16* also shows that ILB exhibits an exponential trend with regards to control messages rendering this scheme inherently non-scalable.

Looking at *Fig 2.17*, which plots the number of migration performed, note that GRAL exhibits a rapid increase as hop awareness increases from 1 to 3, then stabilizing and dropping afterwards. The trend up to 3-hop awareness is due to the fact that increased hop awareness enables the flexible placement of even more agents at even better destinations. Once a good placement is reached, a further increase in hop awareness does not considerably enhance placement quality (see plateau in *Fig 2.15*) but only has the effect of decreasing the number of performed migrations (or more precisely, the consecutive migrations an agent must do in order to reach a good destination; a trend which is more clear for AMA). The above indicate an essential property of k-hop aware algorithms, namely that significant load reduction can be achieved with a relatively small value for k. Even larger k-hop awareness is not entirely without a positive effect, since it results in a reduced number of migrations and a smaller number of control messages for ILA.

A final remark concerns that the larger number of migrations performed by GRAL (ILB) compared to GRAL (ILA) for k=3,4,5, actually leads to an inferior agent placement. We attribute this to capacity reservation conflicts which become more likely for ILB as hop awareness increases. Such conflicts may lead to a suboptimal mapping of agents on nodes, with increasing probability as hop awareness increases, on nodes that are further away from their ideal destinations (and closer to their original hosts). In turn this may create bottlenecks that hinder more beneficial migrations, without necessarily blocking them completely. Even though

ILA can miss opportunities due to outdated capacity information, with increasing probability as hop awareness increases, hence is likely to perform a smaller number of migrations than ILB, precisely for that reason it is also less vulnerable to reservation conflicts thus is more likely to perform migrations that are more beneficial/effective than those of ILB. The net effect seems to be in favor of ILA even when performing fewer migrations compared to ILB.

7.4 Discussion

Summarizing the above, we can conclude on the following: (i) ILA is the more promising locking scheme, in fact, ILB is only applicable for small network awareness; (ii) algorithms using grouping outperform their counterparts in most cases but the ones where no network awareness exists and the capacity is very restricted; (iii) network awareness especially when applied to grouping algorithms together with ILA, drastically increase the quality of the produced placement, while performing comparably fewer migrations and control message exchange compared to non-network aware algorithms; (iv) in the unconstrained case GRAL-1 and GRAL-2 achieve optimal or close to optimal performance.

8 Conclusions

In this work we tackled (as in the previous chapter) the problem of placing the agents comprising an embedded application to the available nodes. We proposed distributed asynchronous algorithms to tackle both uncapacitated and capacitated versions of the problem, considering agent migrations in the form of a group instead of standalone entities (Chapter 1). Algorithms based on group migrations, outperform the ones considering migrations in a single agent manner, with their performance being optimal in most cases when the nodes have no capacity limitations; and near-optimal when nodes have enough capacity to host more than one agents (group of agents). Also, grouping algorithms are in the process of being implemented in POBICOS middleware, bestowing an extra quality on it against other similar systems, since such an attribute proves to be of great importance regarding the energy depletion.

Part of this work has been published in the following conference:

* 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*.

Chapter 3

Identifying the worst-case bounds for AMA and GRAL, and devising an optimal algorithm

1 Introduction

In this chapter we give an extensive analysis through lemmas and theorems about the approximation ratios of AMA and GRAL against the optimal algorithm. Specifically we prove that the worst-case scenario of AMA against the optimal algorithm is not bounded. As regards the approximation ratio of GRAL against the optimal algorithm, it proves to be that expressed by *Eq. 3.1*. With *G* denoting the number of generic agents into our system, while *B/2* being the maximum number of the data an agent can send towards another one, each time the respective network routine is called. It should be stressed that the aforementioned approximation ratio is expressed *Eq. 3.4*, under the restriction that an agent cannot communicate with more than *N* other agents (at most N incident edges). Also, we give some details as to why GRAL is not optimal, and introduce a modification of GRAL (called GRAL*) which proves to be optimal.

Section 2 describes the application and system model as usual. Section 3 proves that the communication cost difference between AMA and the optimal algorithm tends to infinity. In Section 4 we modify GRAL into GRAL* and prove that the later is optimal when having no capacity limitations; while Section 5 provides two worst case bounds of GRAL, with the first one concerning the case where an agent can have an arbitrarily large number of relatives; while the second one considering the case that an agent is allowed to have at most *N* relatives. Finally Section 6 concludes our work.

$$\frac{1}{(G-2)*2B} \quad \text{Eq. 3.1}$$

$$\frac{1}{(G-\left\lceil \frac{2G-N-3}{N} \right\rceil -2)*2B} \quad \text{Eq. 3.2}$$

2 Application and System Model

The application and system model continue being the same as that of Chapter 1, with the system model being a little bit extended. Let $h(a_i)$ be the hosting node of a_i. G and NG denote the number of generic and non-generic agents, respectively. e_{xy} equals 1 when there is an edge between a_x and a_y, otherwise 0. q_{xy} captures the data exchanged between a_x and a_y ($q_{xy} = q_{xy} = C_{xy}+C_{yx}$). Let *Eq. 3.3* represent the data exchanged between a_1 and the non-generic agents it communicates with. *Eq. 3.4* and *Eq. 3.5* capture the data exchanged between an agent (let a_j) and the generic agents it communicates with, with *Eq. 3.5* excluding the co-located agents communicating with a_j. M_{ijk} denotes the migration of a_k from n_i towards n_j.

$$Q_{S,j} = \sum_{a_i \in NG} q_{ij} \mid e_{ij} = 1 \quad \text{Eq. 3.3}$$

$$Q_{N,j} = \sum_{a_i \in G} q_{ij} \mid e_{ij} = 1 \quad \text{Eq. 3.4}$$

$$Q_{N',j} = \sum_{a_i \in G} q_{ij} \mid h(a_i) \neq h(a_j) \wedge e_{ij} = 1 \quad \text{Eq. 3.5}$$

Let D be the diameter of the network, while $B/2$ be the maximum data an agent can send towards another one, each time it calls the respective network routine, assuming that there is an edge connecting those agents. Therefore the maximum volume of data can pass through an edge at any instance of time is equal to B, with this happening when both involved agents send towards one another B/2 data simultaneously. We say that an agent is *individually balanced* if located on its center of gravity; otherwise we say that this agent is *unbalanced*. An agent (let a_x) is considered *totally balanced* if thefollowing hold: a) it is individually balanced; b) there is no subtree that contains a_x after the contraction phase completes (see Sec. 4 in the previous chapter). From now on we will interchangeably use the terms *stabilization* and *balance*, rendering the same meaning.

Definition1. The same equations/properties apply for either an individual agent or a group of agents.

A group of co-located agents can be thought of as a super-agent, provided that these agents are non-partitioned. The construction of a super-agent occurs by merging all the agents of the group as follows: a) each edge that originates from any agent of the group and ends up on another one not belonging to that group becomes an incident edge to that super-agent (originates from the super-agent instead of that agent); b) we ignore any edge that originates from and ends up on an agent belonging to that super-agent (internal edges). Without loss of generality we can assume a super-agent is possessed of the same properties (equations) holding true for a regular agent. Let A_s denote such a super-agent. From now on, we will use interchangeably the terms super-agent and group of agents.

3 Identifying the worst-case bound of AMA

Initially we prove AMA cannot be optimal through the following lemma.

Lemma 1: If the agents of an application are individually but not totally balanced, then it could be found a migration of group of agents, which reduces the total network communication cost.

Proof. Assume a_1 and a_4 (non-generic agents) are hosted by n_1 and n_3, respectively; while a_2 and a_3 (generic agents) are hosted by n_2, with the application and network structure being illustrated in *Fig 3.1* and *Fig 3.2*. Now assume the following hold: $q_{23} > q_{12}, q_{34}$ and $q_{34} > q_{12}$. Observe that there is no beneficial single agent migration, since a_2 and a_3 are stabilized due to the fact that there is no n_j such that: $l_{2j2} > l_{222} + \sum_{x \neq j,2} l_{2x2}$ and $l_{2j3} > l_{223} + \sum_{x \neq j,2} l_{2x3}$. Therefore, AMA cannot migrate any generic agent, with the total network cost being $q_{12} + q_{34}$. However if we were able to migrate both a_2 and a_3 towards n_3 then the total network cost would become $2q_{12} > q_{12} + q_{34}$. Hence, the total network cost could be further reduced by considering a group agent migration.

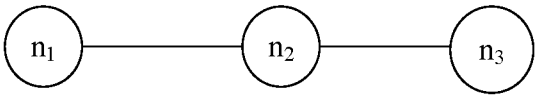

Fig 3.1 Network structure

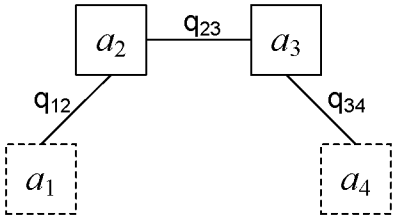

Fig 3.2 Application structure

Corollary 1. AMA is not optimal.

Proof. It stems directly from *Lemma 1*.

Lemma 2. A placement of agents is optimal iff there is no agent or super-agent (group of agents) being unbalanced (i.e., the placement is totally balanced/stabilized).

Proof. It is similar to show that if a placement of agents (let P_s) is totally balanced/stabilized, then there is no other placement reducing further the total communication cost. Proof tries to show this through contradictions of three assumptions described further down. Specifically, the three assumptions to be contradicted, provided that the initial placement is totally balanced, are: a) there is an agent/group migration that results in another balanced placement where the network cost is reduced; b) there is at least one agent/group migration which leaves intact the network cost on its own right, but reduces it via the help of other ones; c) the same as (b) with the difference that there is at least one agent/group migration increases the network cost instead of leaving it intact. For simplicity, to show the above contradictions we make use of only single agent migrations (not groups), without loss of generality due to Definition 1.

Assumption A: Consider a migration of an agent a_k from n_x towards an 1-hop neighbor n_u, under the assumption that the new placement is also totally balanced. Assume also that this migration reduces the total network cost. Since a_k is balanced independently of whether it is located on n_x or n_u, then the following hold:

$$l_{xuk} = l_{xxk} + \sum_{m \neq u,x} l_{xmk} \qquad \text{Eq. 3.6}$$

$$l_{uxk} = l_{uuk} + \sum_{m \neq u,x} l_{umk} \qquad \text{Eq. 3.7}$$

$$l_{xuk} > l_{xxk} + \sum_{m \neq u,x} l_{xmk} \qquad \text{Eq. 3.8}$$

Since we claimed that the migration in question reduces the total communication load, then the equation *Eq. 3.8* must hold true, which comes in contradiction with *Eq. 3.6*. Therefore a migration like that cannot reduce the communication cost on its own, hence we consider the case where such a migration cause other migration(s), where all of them reduce network cost. It is straightforward to show that these case holds true when n_u is a k-hop neighbor.

Assumption B: Consider the previous case with the difference that instead of n_u we have an k-hop neighbor (let n_z) and M_{xzk} leaves the network cost intact. Assume also that after M_{xzk} takes place, a_m migrates due to stabilization issues (changes in load patterns), which means that a_m has an edge towards a_k. However, a_m cannot be located on any node other than n_x, due to the following facts: i) if a_m was located on any node (like n_y depicted in *Fig 3.3*) across the path between n_x and n_z, then M_{xzk} would increase the network cost, which contradicts with our assumption that M_{xzk} does not increase/decrease the cost; ii) the same would hold if a_m was located on any node (like n_f in *Fig 3.3*) using n_y as a router to reach n_x, with the restriction that this node must not use n_z as a router; iii) if a_m was located on any node (like n_b) before the path between n_x and n_z, or after that path (like n_a), then a_m would not initially be balanced (contradiction); iv) if a_m was hosted by n_z, then a_m could not migrate anywhere since a_m would be eventually co-located with a_k (contradiction). Therefore, we result in the fact that a_m should be hosted by n_x. However the migration of both a_m and a_k cannot reduce the network cost; since that would mean that a group of agents is not stabilized which contradicts with the assumption of an initial totally balanced placement. It is self-evident that the same holds in case we have more than one agent (like a_m) to be migrated.

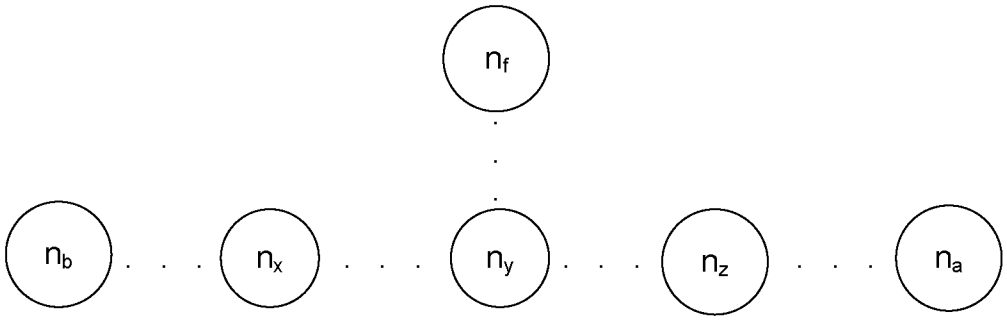

Fig 3.3 Network

Assumption C: We omit the case where M_{xzk} increases the communication cost without causing any other migration, since in that case the final network cost will increase (our proof is based on network cost reduction). Hence, assume M_{xzk} increases network cost, with that migration causing an extra migration of the agent a_m (with a_m having an edge towards a_k, as previously)

due to load changes, which finally reduces the initial network cost (i.e. the cost before M_{xzk} takes place). However such a situation could not happen since it is obvious that in the best case (assume that there are only two agents into our system a_m and a_k) a_m would amortize the cost caused by M_{xzk}. While in the worst case either M_{xzk} should be "revoked" (by performing M_{zxk}), or the migration of a_m would cause other migrations like the ones caused by M_{xzk}. As we can observe such a kind of migrations may be performed in a recursive fashion till the boundaries of the application tree are reached (root agent and non-generic agents), however without eventually reducing the final network cost.

Summing up: It was shown that if an algorithm results in a totally balanced placement, then there is no migration or a series of migrations of agents (or super-agents by making use of *Definition 1*) that can take place to reduce the network cost further more. Therefore, we conclude that a placement is optimal when all the agents are totally balanced (there is no agent/group being unbalanced).

Theorem 1. The worst-case bound between AMA and the optimal algorithm is $(G-1)*2B*D*T$, with T denoting the maximum number of times the agents can send $B/2$ data units over the network.

Proof. Assume that in our system there are no unbalanced individual agents (without loss of generality since AMA always results in a placement of no unbalanced individual agents), while there exist one unbalanced group (let super agent A_s). Since AMA cannot identify such a group to migrate it to its center of gravity, the worst-case scenario is for A_s to be as farther from its centre of gravity as possible (it is obvious that the optimal algorithm will decide to migrate this group of agents to its centre of gravity). In order for AMA to incur as large network cost as possible, while for optimal as small as possible, we need to decide which node will host A_s and which node(s) will host the adjacent agents to A_s. The best case for optimal algorithm is for the adjacent agents of A_s to be co-located on the same node (let n_r), hence the optimal algorithm will take the decision to migrate A_s towards n_r, with the network cost being zero. The worst-case for AMA results by consulting the following function:

$$f(i,j,s) = l_{ijs}h_{ij} - l_{iis}h_{ij} - \sum_{\forall x \neq i,j : D_{ijx}=0 \wedge h_{ix}=1} l_{ixs}h_{ij} - \sum_{\forall x \neq i,j : D_{ijx}=1} (l_{ixs} - l_{ius})(h_{jx} - h_{ix}) \mid D_{xju} = 1 \wedge h_{xu} = 1 \wedge u \neq j \qquad \text{Eq. 3.9}$$

Let *Eq. 3.9* be a function yielding the benefit/cost of migrating A_s from n_i to n_j. The first factor concerns the benefit of M_{ijs} due to the load associated with A_s and directed to n_j when n_i hosts A_s.

The second factor captures the cost of migrating A_s due to the communication load with its co-located agents. Third factor concerns the cost of moving A_s towards an opposite direction against the nodes hosting agents communicating with A_s (excluding n_i). Finally, the forth factor signifies the benefit/cost — benefit when n_x is closer to n_j; while cost when it is closer to n_i — of moving A_s towards n_j in terms of the nodes located in-between n_i and n_j.

We demand that *Eq. 3.9* be as large as possible in order for AMA to incur as large communication cost as possible. Looking carefully on *Eq. 3.9* it is obvious that when the first factor increases then also the cost increases, while second and third factor contribute negatively to the communication cost. In terms of 4th factor, note that the following equation holds always true $h_{ix} - h_{jx} < h_{ij}$, hence it is worse (in terms of cost) to host an agent on n_j instead of n_x. Therefore the worst-case is to have the relative agents of A_s located on n_j, and the hop-distance between n_i and n_j to be as large as possible, i.e., D.

Assumption A: Assume all generic agents participate into A_s, therefore all adjacent agents to A_s should be non-generic agents. Let n_i be the initial hosting node of A_s. Note that in order for each agent a_k participating into A_s to be individually balanced *Eq. 3.10* should hold true:

$$Q_{Sk} + Q_{N'k} \leq \sum_{\forall x: e_{xk}=1} q_{xk} \mid a_k, a_x \in A_s \qquad \text{Eq. 3.10}$$

$$\sum_{\forall k: a_k \in A_s} (Q_{Sk} + Q_{N'k}) D \qquad \text{Eq. 3.11}$$

$$Q_{Sk} + Q_{N'k} = \sum_{\forall x: e_{xk}=1 \wedge a_x \in A_s} B, \ \forall k : a_k \in A_s \qquad \text{Eq. 3.12}$$

Specifically *Eq. 3.10* denotes that, Q_{Sk} and $Q_{N'k}$ (considering that a_k participates in the unbalanced group) should be equal to or less than the accumulated communication load between a_k and each generic agent belonging into the unbalanced group. *Eq. 3.11* represents the network cost induced by AMA due to the external load (Q_{Sk} + $Q_{N'k}$) of each agent a_k participating into A_s. From now on, any reference to network cost will be inextricably linked with the fact that any agent is able to send *B/2* data units (towards each adjacent agent of its own) at most one time, *T = 1*. The factor D in *Eq. 3.11* is justified by an earlier remark that the adjacent agents to A_s should be hosted by a node that is D hops away against the node hosting A_s. Therefore the worst-case scenario for AMA is for Q_{Sk} and $Q_{N'k}$ to be as large as possible, since AMA cannot identify a migration to save this load/cost. Namely, given that: a) at most B data units can travel over an edge (*B/2* for each agent incident to that edge); b) each q_{xk} should be as large as possible

(provided that a_k and a_x belong into A_s); we conclude that each such q_{xk} should be equal to B, hence *Eq. 3.10* becomes *Eq. 3.12*.

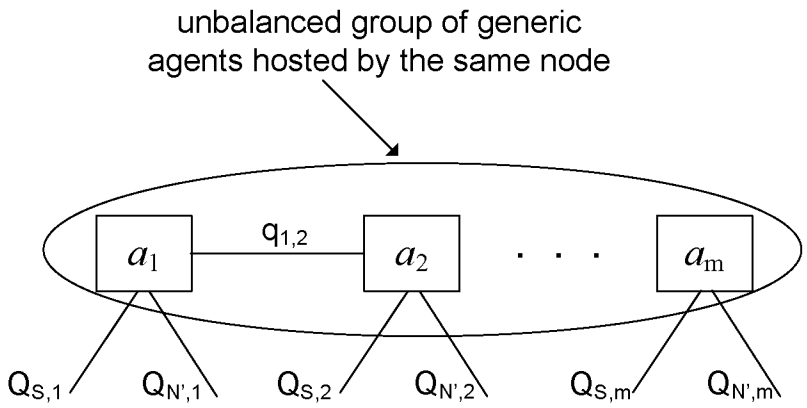

Fig 3.4 Unbalanced group A_s

Initially we assume that only two generic agents do exist into our system (m = 2 in *Fig 3.4*). Therefore, *Eq. 3.11* becomes equal to $2B*D$, since $Q_{S1}+Q_{N'1}=q_{12}=B$ and $Q_{S2}+Q_{N'2}=q_{12}=B$. In case of 3 generic agents (m = 3 in *Fig 3.4*) AMA results in a placement where the network cost is equal to $4B*D$, since $Q_{S1}+Q_{N'1}=q_{12}=B$, $Q_{S2}+Q_{N'2}=q_{12}+q_{23}=2B$, and $Q_{S3}+Q_{N'3}=q_{23}=B$. For 4 generic agents the network cost becomes $6B*D$, and so on. Hence, we observe that for each internal edge of A_s AMA incurs $2B*D$ additional cost. Therefore, due to the fact that our application is structured as a tree (in a tree of G nodes G-1 edges there exist), the largest difference (in terms of cost) between AMA and the optimal algorithm is equal to (G-1)*$2B*D$. Note also that the cost incurred by AMA is independent of how the agents belonging into A_s are connected with each other, since it depends only on the number of A_s' internal edges.

Assumption B: In the sequel we proceed with the case that $\{a_1..a_{G-1}\}$ belong to A_S, while a_G not to. Therefore, a_G is either located on the same node hosting A_s or on another one.

Assumption B1: In the first case the optimal algorithm will decide to migrate A_s from n_i towards the center of gravity (let n_j). Note that the ideal case for the optimal algorithm is to migrate also a_G onto n_j, however it could not make such a decision since in that case a_G would belong into A_s which comes in contradiction with *Assumption B*. This means that the load Q_{SG} cannot come from n_j, also Q_{NG} cannot be greater than Q_{SG} (since in that case a_G would belong into A_s). The worst case for AMA (best for the optimal algorithm) is the Q_{NG} to be as high as possible, and a_G to migrate as close to n_j as possible (let this node be n_x) with $h_{jx}=1$, and with

n_x being the source for the external load Q_{SG}. As a result, the cost incurred by the optimal algorithm is that of the communication between a_G and the agents belonging to A_s (*Eq. 3.13*). While AMA incurs cost equal to the external load of both A_s and a_G multiplied by the hops traversed (h_{ij} and h_{ix} respectively); the external load of A_s is equal to (G-1)*2B, while the external load of a_G cannot be greater than $\sum_{i=1}^{G-1} q_{iG} \mid e_{iG} = 1 \wedge a_i \in A_s$ due to the fact that a_G must be individually balanced on n_i (our initial assumption). Note also that a_G cannot be connected with more than one agent belonging to A_s due to the fact that: a) the application is structured as a tree; and b) the agents belonging to A_s must be non-partitioned (Definition 1). Therefore the external load of a_G cannot be greater than $q_{iG} \mid e_{iG} = 1 \wedge a_i \in A_s$. As a result the communication cost of AMA is represented by *Eq. 3.14*, which is less than (G-1)*2B*D.

$$h_{xj} \sum_{i=1}^{G-1} q_{iG} \mid e_{iG} = 1 \wedge a_i \in A_s \qquad \text{Eq. 3.13}$$

$$(G-2)*2B*h_{ij} + B*h_{ix} = (G-2)*2B*D + B*(D-1) \qquad \text{Eq. 3.14}$$

$$(G-2)*2B*h_{ij} + B*h_{ij} = (G-2)*2B*D + B*D \qquad \text{Eq. 3.15}$$

Assumption B2: Consider now the second case where a_G is initially hosted by a node other than n_i. In order for the optimal algorithm to pay no cost, the best-case scenario is for n_j to initially host a_G. The worst-case scenario for AMA is for a_G to be as far away in terms of n_i as possible (i.e on n_j). Following the same rationale as that of *Eq. 3.14*, with the difference that h_{ix} must be replaced by h_{ij}, we end up on *Eq. 3.15* which is less than (G-1)*2B*D.

Summing up: the worst-case scenario of AMA is that described in *Assumption A* (all the generic agents belong into the unbalanced group). Note also that in case that T tends to infinity the communication cost difference between AMA and the optimal algorithm tends also to infinity.

4 GRAL*: Modifying GRAL to become optimal

Lemma3. The way GRAL chooses the destination node (the most promising neighboring node) for a potential migrating group may lead GRAL to result in a sub-optimal placement.

Proof. When GRAL runs on a node it chooses for each non-partitioned collection of locally-hosted generic agents (disjoint subtree) the most promising destination node in order to proceed with the subtree construction/contraction phase. Specifically, when GRAL runs on a node (let n_x) records: a) for any neighboring node (let n_y) the communication load exchanged between the generic agents located on node n_x and the generic or non-generic agents located on either n_y or on a node using n_y as a router; b) the load exchanged between n_x's locally-hosted generic and non-generic agents (without taking into account the load between locally-hosted generic agents). For example, assume that only two nodes does exist into our system (n_1 and n_2). Consider that n_1 hosts the agents a_1, $a_{1'}$, a_2 and a_3 depicted in *Fig 3.5*, while $a_{2'}$ and $a_{3'}$ are hosted by n_2. Note that the current network cost is equal to 20, while in case of migrating both a_2 and a_3 towards n_2 the network cost becomes 1. Note that GRAL is not able to identify this beneficial migrating group due to the fact that when GRAL runs on n_1 it finds out that the most promising destination node for any potential migrating group is n_1 itself; since the accumulated local load is totaled 100 which is greater than the accumulated remote load associated n_2, which amounts to 20. A solution to this drawback is for GRAL to proceed with a tree construction/contraction phase for all the neighboring nodes of the node it runs on.

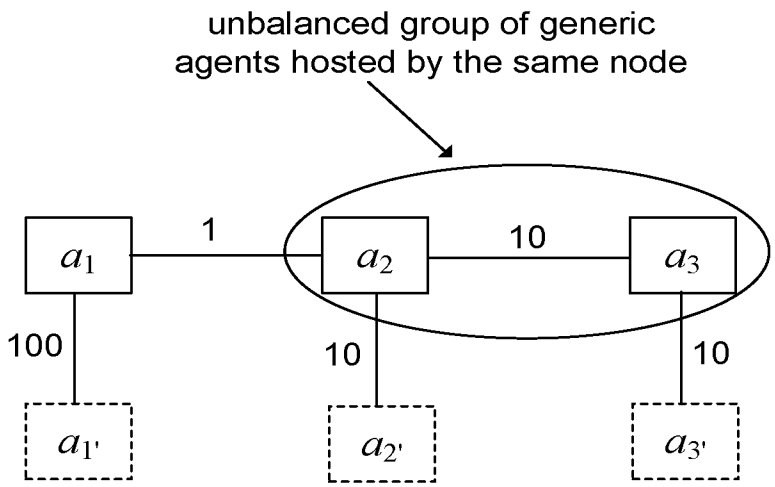

Fig 3.5 Unbalanced group of 3 agents

Lemma 4. The fact that the root agent of a sub-tree cannot be pruned (when contraction phase takes place) may lead GRAL to result in a sub-optimal placement.

Proof. The only agent that cannot be pruned when contraction phase takes place is the root agent. Consider again the example of *Lemma 3* where GRAL is possessed of the ability to proceed with the tree construction/contraction phase for all the neighboring nodes of the running

node. Assume again the example described previously and depicted in *Fig 3.5*. Let GRAL construct a sub-tree (rooted on a_1) for the potential destination node n_2 (*Fig 3.6*). Observe that when the contraction phase completes the final merged/contracted node has negative benefit which leads GRAL to cancel the decision to migrate this group. However if GRAL constructs a sub-tree rooted on a_2 (*Fig 3.7*), it is able to identify a beneficial migrating group and result in a better placement.

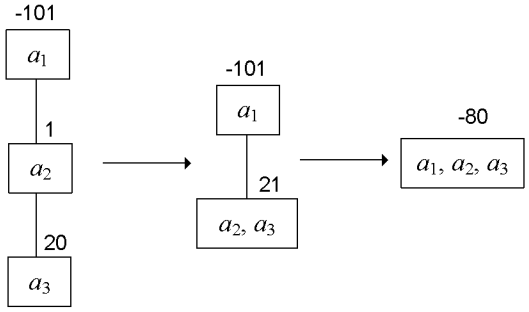

Fig 3.6 subtree rooted on a_1

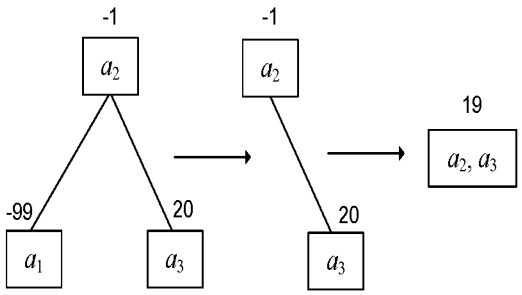

Fig 3.7 subtree rooted on a_2

Lemma 5. If the root agent of an identified sub-tree belongs to an unbalanced group (let A_s) of agents, then after the contraction phase GRAL will identify a beneficial migrating group (and migrate it in an optimal way) which is identical to A_s.

Proof. Recall that in the construction phase of a sub-tree, each agent belonging to that sub-tree is assigned a partial benefit for its migration towards a promising destination (let pn_x). Due to the nature of partial benefit calculation, if an agent is located at the bottom level of the sub-tree, then the partial benefit of that agent corresponds to the upper bound benefit of its migration towards pn_x.

Assumption 1: Consider that GRAL decides to construct a sub-tree of only two levels based on pn_x. Assume also that a part of this sub-tree represents an unbalanced group of agents where the optimal algorithm will decide to migrate it towards n_x (with the root agent belonging to the unbalanced group). This part of the sub-tree is called optimal migrating group.

If there are agents at the bottom level of the sub-tree, which have partial benefit equal to or less than zero, then GRAL will take the decision to prune them. Note that the optimal algorithm will decide to remove these agents from the optimal migrating group as well, since their upper bound migration benefit will be less than or equal to zero. Therefore GRAL's decision to prune them is correct. The rest bottom-level agents have positive partial benefit each, which means that if their parent (in terms of the sub-tree structure) migrate also, then their actual migration benefit should be greater than zero. It is obvious that the optimal algorithm will decide to

include these agents into the optimal migrating group, with the latter being consisted of the agents in question plus the root agent. According to GRAL's contraction phase, these agents will be merged (along with their partial benefits) with the root agent. Of course, in order for GRAL to perform the migration of the group represented by the final contracted node, the partial benefit of the latter should be greater than zero. Note that this partial benefit is equal to the actual benefit of migrating the group represented by that node. According to the optimal algorithm this actual migration benefit is positive, therefore GRAL will take the optimal decision to migrate that group.

Assumption 2: The same as *Assumption 1* with the difference that the sub-tree is consisted of three levels instead of 2. In the first step GRAL will proceed with the merge/pruning of the bottom level of the tree. As said earlier, each leaf contributing negatively will be pruned (the optimal algorithm will take the same decision). The rest bottom-level agents will be merged with the next upper-level nodes (which nodes represent agents), since they have positive partial benefit. Therefore, we result in a case identical to that of *Assumption 1*, with the difference that some of the bottom-level nodes may represent super-agents instead of individual ones. By making use of Definition 1, we conclude that GRAL will migrate the same agents with the optimal algorithm.

Assumption 3: The same as *assumption B* with the difference that the sub-tree is consisted of 4 levels instead of 3. Following the same rationale as previously, we conclude that this case is reduced to the case of *Assumption 2*. Therefore, GRAL again takes the optimal decision.

Iteratively, in general the case where the sub-tree is consisted of n levels is always reduced to *Assumption 1*. As a result, we conclude that if the root agent of a sub-tree belongs to an unbalanced group, then GRAL will identify and migrate this group in an optimal way.

Definition 2. GRAL* is a modification of GRAL tackling the drawbacks brought out by *lemma 3* and *lemma 4* through the following way. For each possible pair (A_s, n_d)— where A_s is an identified sub-tree and n_d is the potential destination node of that sub-tree — GRAL* constructs as many sub-trees (containing the same agents with A_s) as the number of the agents belonging to A_s, with each such sub-tree being rooted on a different agent.

Lemma 6. Each unbalanced group is contained in one of GRAL's sub-trees.

Proof. According to the sub-tree identification phase (described in Chapter 2, sec 4.2), GRAL organize all the locally hosted agents into sub-trees (co-located non-partitioned generic agents).

Since an unbalanced group is consisted of non-partitioned locally hosted agents, it is obvious that each unbalanced group should be contained by some GRAL's sub-tree.

Theorem 2. GRAL* results always in an optimal placement.

Proof. According to *Lemma 6* and *Definition 2*, for each unbalanced group there is always a GRAL*'s sub-tree where a) it includes this unbalanced group, and b) the root agent of this sub-tree belongs to this unbalanced group. Combining the above with *Lemma 5*, we conclude that GRAL* will identify all the unbalanced groups and take the optimal decision for them. This means that GRAL* will always result in a totally balanced placement, which in combination with *Lemma 2* prove GRAL*'s optimality.

5 Identifying the worst-case bound of GRAL

Theorem 3. The approximation ratio between GRAL and the optimal algorithm is $\frac{1}{(G-2)*2*B}$ for $G>2$, otherwise GRAL is optimal.

Proof.

Part A: In this part we prove that GRAL is optimal when $G \leq 2$ GRAL. It is obvious that when $G=1$ AMA is optimal (it stems from *Lemma 2*) so GRAL is optimal too, so we need to consider only the case where $G=2$. Since sub-optimality of GRAL is attributed to *Lemmas 3* and *4* (GRAL* becomes optimal by overcoming the drawbacks brought out by these lemmas), we only need to show that these lemmas do not hold true for the case of $G=2$. Getting started with *Lemma 3*, we can observe that in case GRAL cannot identify any promising neighboring node, then either both generic agents are individually and totally balanced, or one of them misleads GRAL to take the decision that the promising destination node is the local one. This means that only one generic agent is unbalanced, in which case both AMA and GRAL are able to identify such an individually unbalanced agent, and hence *Lemma 3* does not hold true. Proceeding with *Lemma 4* we predicate that it also does not hold true. As regards *Lemma 4*, it doesn't hold true (in terms of $G=2$) as well; this is due to the fact that when the identified sub-tree is rooted on an agent not belonging to an unbalanced group, then it is obvious that only individually unbalanced agents there can be. Hence, it is self-evident that *Lemma 4* does not hold true.

Part B: Here we see the case where $G > 2$. We extend *Fig 3.5* into *Fig 3.8* in order for both data exchanged between agents and the number of agents to be arbitrarily large. Let *loc*($Q_{S,i}$) and *ext*($Q_{S,i}$) denote the local respectively remote load attributed to the data exchanged between a_i and its adjacent non-generic agents. Assume the case where only 3 generic agents there exist, and they are hosted by some node n_x. Let $m = 2$ in *Fig 3.8*, then a_1 and a_2 will belong to the unbalanced group, while a_3 will be totally balanced. Note that in our case $Q_{N'i}$ ($\forall i$) is equal to 0, since all generic agents are assumed to be co-located. Also in order for GRAL to not be able to identify that unbalanced group (for the sake of proof) we need *Eq. 3.16* to hold true.

$$Q_{S,m+1} + Q_{N',m+1} \geq (G-2) * 2B \quad \text{Eq. 3.16}$$

$$\sum_i ext(Q_{S,i}) + q_{2,3} * D \quad \text{Eq. 3.17}$$

$$\sum_i ext(Q_{S,i}) + 2B * D \quad \text{Eq. 3.18}$$

$$\frac{\sum_i ext(Q_{S,i}) + q_{2,3} * D}{\sum_i ext(Q_{S,i}) + 2 * B * D} \quad \text{Eq. 3.19}$$

$$\frac{1}{(G-2) * 2B} \quad \text{Eq. 3.20}$$

According to those discussed in the previous paragraph, the optimal algorithm and GRAL will result in a placement where the communication cost is equal to that expressed by *Eq. 3.17* and *Eq. 3.18*, respectively; with $2B*D$ stemming from *Eq. 3.12*, by following the same rationale as that of *assumption A* in *Theorem 1*. Therefore their ratio is given by *Eq. 3.19*, which lessens in case both external loads and $q_{2,3}$ are equal to zero. However $q_{2,3}$ could not be equal to zero since in that case GRAL would identify the unbalanced group, resulting in that way in an optimal placement. Hence, $q_{2,3}$ is set to 1 (the minimum feasible value) with the worst-case bound becoming 1/2B. Following the same rationale as above we conclude that for $G = 4$ ($m = 3$ in *Fig 3.8*) the worst-case bound becomes 1/4B, while for $G = 5$ we have 1/6B. Finally, by reduction to an arbitrarily large G, we conclude that the worst-case ratio is given by *Eq. 3.20*.

Following the same reasoning as in *Theorem 1* we conclude that a) the network cost incurred by the resulting placement of GRAL is independent of how the agents belonging into the unbalanced group are connected with each other; b) the more the agents belonging to the unbalanced group the more the network cost incurred by the resulting placement of GRAL; c)

the totally balanced agent (a_{m+1}) should be adjacent with only one agent of the unbalanced group. Else, the collection of $a_1..a_m$ would be partitioned due to the tree-structured application; in other words, the number of the agents belonging to the unbalanced group could not be greater than G-1, lessening in that way the worst-case bound (not desired).

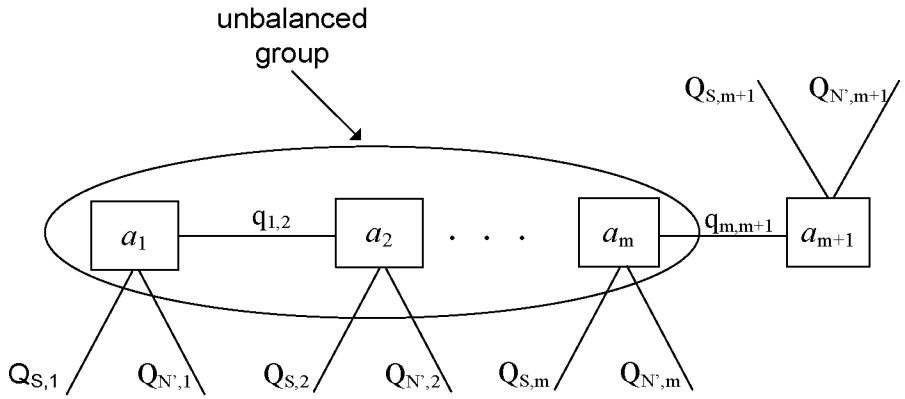

Fig 3.8 Unbalanced group of G agents

Theorem 4. The approximation ratio of GRAL is equal to $\frac{1}{(G-\left\lceil \frac{2G-N-3}{N} \right\rceil -2)*2B}$, under the following restrictions: i) the number of incident edges to an agent is at most N and at least 3; and ii) $G>2$.

Proof. In order for GRAL to result in a non optimal placement (the same rationale as in previous proof), it is required *Eq. 3.21* to hold true. Also, due to the fact that an agent cannot have more than N adjacent agents, *Eq. 3.22* should hold true as well. Equating *Eq. 3.21* with *Eq. 3.22* we get *Eq. 3.23*. Specifically, the positive part of the latter equation specifies the external load of the agents participating into the unbalanced group, while the negative part concerns the local load of the totally balanced agent. However, *Eq. 3.23* does not hold true for any possible combination (G, N), thus enabling GRAL to identify the unbalanced group depicted in *Fig 3.8* (which spoils the proof). So it is needful for *Eq. 3.23* to be modified to hold true for any combination (G, N).

This can be achieved by having the negative part of *Eq. 3.23* to always surpass the positive one. Note that the positive part of that equation decreases when the number of agents participating into the unbalanced group decreases; of course this means that the number of agents belonging to the totally balanced group increases proportionally. Therefore, assuming $G-1-k$ agents participate into the unbalanced group instead of $G-1$, *Eq. 3.21* is transformed into *Eq. 3.24*.

Note that the number of totally balanced agents (called totally balanced group) is $k+1$, instead of 1, consequently *Eq. 3.22* is transformed into *Eq. 3.25*. *Fig 3.9* illustrates such a scenario, with the unbalanced agents being $m=G-1-k$, while the totally balanced ones being $n-(m+1)=k+1$. Note that the second part of *Eq. 3.25* is attributed to the fact that when the totally balanced group is consisted of $k+1$ agents, then all these agents are able to have a total of at most $\sum_{i=1}^{k}(N-2)+(N-1)$ adjacent non-generic agents. Putting these all together, we infer that for each agent being transferred from the unbalanced group to the totally balanced one, the positive part of *Eq. 3.21* decreases by $2B$ since the internal edges of the unbalanced group decrease by 1; while the negative part decreases by $(N-2)*B$. As a result, *Eq. 3.21* becomes *Eq. 3.26* (by equating *Eq. 3.24* with *Eq. 3.25*), with k denoting the agents transferred from the unbalanced group to the totally balanced one.

$$Q_{S,m+1}+Q_{N',m+1}\geq(G-2)*2B \quad \text{Eq. 3.21}$$

$$Q_{S,m+1}+Q_{N',m+1}\leq(N-1)*B \quad \text{Eq. 3.22}$$

$$(N-1)*B\geq(G-2)*2B\Rightarrow(G-2)*2B-(N-1)*B\leq 0 \quad \text{Eq. 3.23}$$

$$\sum_{i=1}^{k+1}(Q_{S,m+i}+Q_{N',m+i})\geq(G-2-k)*2B \quad \text{Eq. 3.24}$$

$$\sum_{i=1}^{k+1}(Q_{S,m+i}+Q_{N',m+i})\leq\sum_{i=1}^{k}((N-2)*B)-(N-1)*B \quad \text{Eq. 3.25}$$

$$(G-2-k)*2B-(N-1)*B-\sum_{i=0}^{k}(N-2)*B\leq 0 \quad \text{Eq. 3.26}$$

$$k=\left\lceil\frac{(G-2)*2B-(N-1)*B}{2B+(N-2)*B}\right\rceil=\left\lceil\frac{2G-N-3}{N}\right\rceil \quad \text{Eq. 3.27}$$

$$\frac{1}{(G-\left\lceil\frac{2G-N-3}{N}\right\rceil-2)*2B} \quad \text{Eq. 3.28}$$

What it remains is to decide the value of k in order for *Eq. 3.26* to hold always true, keeping at the same time this equation as close to zero as possible in order for GRAL to be as worse as possible. This value is given by *Eq. 3.27* which says that *assuming an unbalanced group of* $G-1$ *agents and a totally balanced group of only 1, how many agents we need to transfer from the unbalanced group to the totally balanced one in order for that equation to become equal to*

or less than 0. Recall that for each such transfer, the positive part of *Eq. 3.26* decreases by $2B$, while the negative one decreases by $(N-2)*B$, giving a total decrease of $2B+(N-2)*B$. Putting all these together, we infer that the unbalanced group is able to have at most $G-k-2$ internal edges. Following the same rationale as in *Theorem 3* — that is, for each internal edge ($G-k-2$ in total) inside unbalanced group, GRAL incurs network cost equal to $2B$ — we conclude that the approximation ratio is given by *Eq. 3.28*.

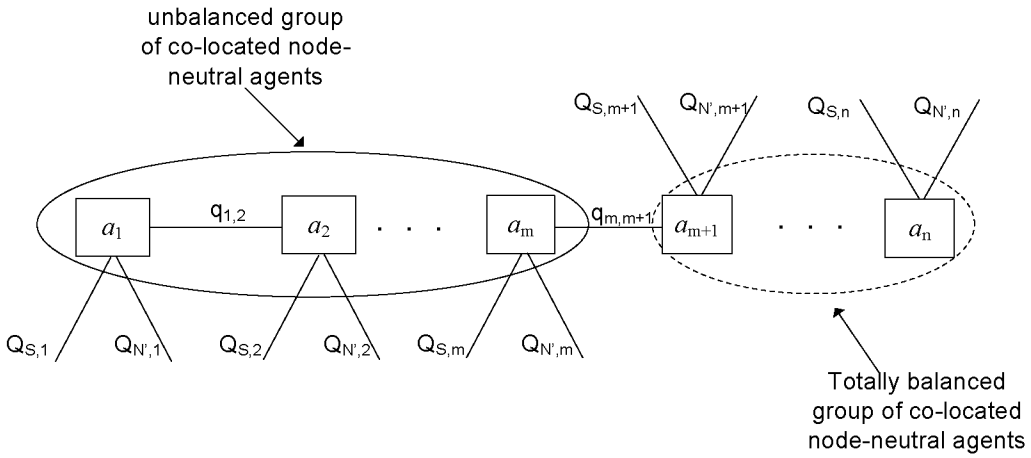

Fig 3.9 Unbalanced and totally balanced groups

6 Conclusions

In this chapter we discussed the bounds for the algorithms proposed in Chapter 1and Chapter 2, and showed that AMA can not be bounded. We also proposed two simple changes for the GRAL algorithm, making it in that way to result always in the optimal placement.

Part of this work is going to be submitted in the following journal:

* N. Tziritas, T. Loukopoulos, S. Lalis and P. Lampsas, "Identifying the worst-case bounds for AMA and GRAL, and devising an optimal algorithm," to be submitted in IEEE Transactions on Parallel and Distributed Systems.

Chapter 4

Introducing Agent Evictions to Improve Application Placement in Wireless Embedded Systems

1 Introduction

In the previous chapter we proved that GRAL can be transformed into an optimal algorithm (GRAL*), provided that there are no storage-constrained nodes. However, the agent migration problem continues being intractable for the case where the nodes of the system have storage capacity limitations. Specifically, we prove that the agent migration problem is NP-complete through its reduction to the well-known knapsack problem [56], considering no capacious (in terms of memory) nodes. The algorithms proposed in this chapter are designed in a more sophisticated way against the solutions proposed in previous ones.

This work introduces the concept of "evictions". Specifically, the term "eviction" represents a migration of an agent without aiming at reducing network cost, but at increasing the free storage capacity of the current hosting node. Of course such a migration it does not come for free (network cost increases), since it is distanced from its center of gravity. It should be stressed that an agent eviction takes place iff there is a guarantee that the induced network cost will be amortized by some other migration. The former migration is also called space-effective, while the latter one is named cost-effective.

The algorithms proposed take the decision for migrating an agent based on a fully distributed manner. Specifically, a cost-effective migration is considered in the same way as a migration in Chapter 1, while the decision for a space-effective one is taken in a different way described in

the following sections. Note that these algorithms are enhanced with 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 a *radio silence mechanism*, and can be also applied to ILA and ILB protocols to decrease considerably their control messages.

The rest of this work is structured as follows: section 2 describes the problem formulation, application and system model; section 3 presents the proposed distributed algorithms; section 4 section provides the experimental evaluation of the proposed algorithms through simulation.

2 Application, System Model and Problem Formulation

Since the application model of this chapter is the same with the previous ones we referred the reader to Chapter 1.

2.1 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 it can host.

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 α_k, $s(\alpha_k)$, $h(\alpha_k)$ be the k^{th} agent in the system, its 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 α_k may exchange messages with its relatives (parent or children) in the application tree, let 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 α_k to α_m, i.e., the number of data units α_k sends to α_m over a specific period (note that, in the general case, $T_{km} \neq T_{mk}$).

2.2 Problem formulation

The objective is to reduce the amount of wireless traffic between nodes due to the application-level communication, i.e., the messages exchanged between agents. 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 in order 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, each denoted with o_i. Let s_i, v_i be the size and value of o_i and S the size of the knapsack. The knapsack problem consists of finding the collection of objects of maximum value V that fits in the knapsack. We can transform any such statement to an equivalent statement of the agent placement problem studied in this work, as follows. The application tree consists of the root and two more levels. In the first level, k generic agents (let a_i) exist, corresponding one to one to the knapsack objects. In the second level, k non-generic agents (let $a_{i'}$) exist, such as each generic agent a_i communicates with exactly one non-generic agent $a_{i'}$ and vice versa.

The communication cost between the tree root and the generic agents is set to be e, where $e \leq \min(v_i)$, and between the generic agent a_i and the non generic $a_{i'}$ is set to be v_i-e. Two nodes exist in the network n_1 and n_2. All the generic agents initially rest at n_1, while n_2 holds all non-generic agents together with the application root. The size of a generic agent a_i is set to the corresponding knapsack's object size (s_i), the size of the root agent is set to: $1+\sum_{\forall i} s_i$, while the size of the non-generic agents can be any positive number. Finally, the capacity of n_1 is set to $\sum_{\forall i} s_i$, i.e., just enough to hold the generic agents allocated 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 agents of the first level (that rest in n_1) communicating with the root agent and the agents of the second level (that rest at n_2). The total load of this assignment is $\sum_{\forall i}(v_i - e) + \sum_{\forall i} e = \sum_{\forall i} v_i$. In order to minimize this load the only possible migrations involve the agents of the first level moving from n_1 to n_2. This is due to the fact that the agents of the second level are non-generic (thus cannot move), while the root agent has size greater than the capacity of n_1. It is easy to see that each migration of a_i from n_1 to n_2, decreases the network cost by v_i 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 the initial knapsack instance.

2.3 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 that a_k incurs into the system if hosted by n_i can be expressed as follows:

$$l_i^k = \sum_{a_m \in RS_k} (T_{km} + T_{mk}) r_{ih(a_m)} \quad \text{Eq. 4.1}$$

Let M_{ij}^k refer to the migration of α_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:

$$B_{ij}^k = l_j^k - l_i^k \quad \text{Eq. 4.2}$$

For M_{ij}^k to be eligible, α_k should be node-neutral and the destination node n_j should have enough free capacity:

$$a_k, 1 \le k \le NA \quad \text{Eq. 4.3}$$

$$c(n_j) \ge s(a_k) + \sum_{m}^{NA+SA} s(a_m) \mid h(a_m) = n_j \quad \text{Eq. 4.4}$$

Each migration M_{ij}^k leads to a new placement, which 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*. But note that not all beneficial migrations are eligible, due to the capacity constraint (*Eq. 4.4*).

2.4 Evictions

To alleviate this 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. In order to

achieve this, evictions must be followed by at least one migration with a benefit that outweighs their penalty.

In the sequel we give an example to illustrate this scenario. Assume the application depicted in *Fig 4.1*, which 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 the application is deployed in a network of seven nodes as shown in *Fig 4.2*, where each node has enough capacity to host only one agent.

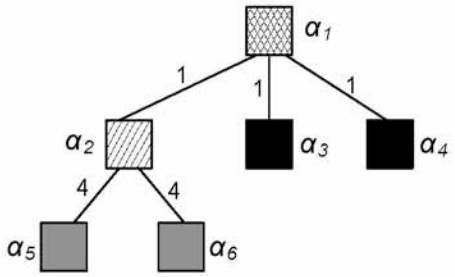

Fig 4.1 Application structure and traffic

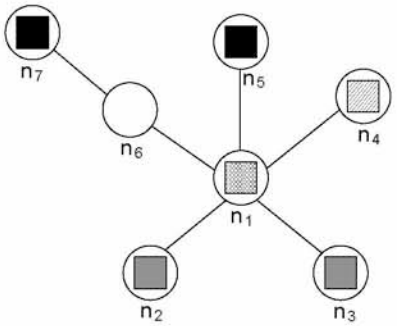

Fig 4.2 Initial application placement

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 *Eq. 4.2*. 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 *Eq. 4.2*. But note that M_{41}^2 is not feasible due to the capacity constraint (*Eq. 4.4*) 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.

3 Heuristics

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

3.1 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, its size and the benefit of the migration as per *Eq. 4.2*.

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, it 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, or 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. This ensures that it will be possible to perform the respective migration, if the node that issued the hosting request decides to proceed. Such 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 the nodes that replied positively.

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

Table 4.1 Pseudocode description of SP

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 }
}
```

```
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 }
      }
   }
}
```

3.2 Network flooding algorithm (FL)

In 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 this problem, we propose an algorithm where the agent to be evicted is chosen based on the smallest "total" penalty of this action.

The main difference 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.

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 which agent(s) it has to evict to which nodes.

For example, consider an application that is deployed in a network of nodes as shown in *Fig 4.3*. Assume that each node is able to host one agent, and that all agents depicted in *Fig 4.3* 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 is as listed in *Table 4.2*.

Table 4.2 Benefit/penalty per migration

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

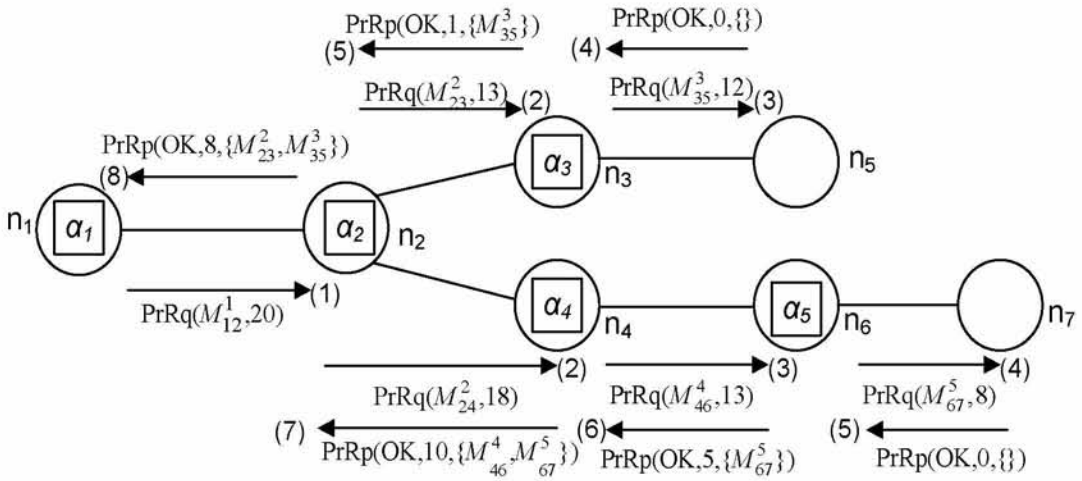

Fig 4.3 Example with probe requests/replies

Given that the only beneficial migration is that of a_1 from n_1 to n_2, node n_1 will send a probe request to n_2 with a benefit value of 20. Since 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 n_5 has sufficient free capacity to host a_3, thus sends back a positive reply with a penalty of 0 and an empty eviction list. When n_3 receives this reply, it sends to n_2 a positive reply with the cumulative penalty of 1 and an updated eviction list that includes M_{35}^3. Similarly, n_2 will receive from n_4 a positive reply with a cumulative penalty of 10 and the respective eviction list $\{M_{46}^4, M_{67}^5\}$. n_2 will chose the reply with the smallest penalty, i.e., that of n_3, and will reply positively to 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 4.3*). 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 several different requests at the same time. This 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, which in turn may result in many failed hosting requests, a hosting request cannot concern an agent involved in another pending hosting requests 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 any 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).

3.3 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. Hence, assuming a stable communication pattern between the agents of the application 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 is also bounded by the

network diameter (there are no cycles). It follows that the total number of migrations is bounded, therefore, eventually, there will be no more migrations (or evictions) to perform.

It is important to note that a beneficial migration as per *Eq. 4.2* 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. Else, it would be possible to have a never ending loop of "swaps". The algorithms can be easily extended to satisfy this constraint, e.g., by notifying the relatives of an agent before commencing with the actual migration process.

3.4 Radio silence

Both algorithms are extended with a mechanism that stops the respective protocols from producing messages (ad infinitum) once convergence is reached. This 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 its block list, and forwards the update to its neighbors.

Due to convergence, eventually, no more migrations will take place. The source(s) of the last beneficial migration(s) will issue update messages due to the hosting capacity that is 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 henceforth suppress the generation of new requests due to the blocking policy. When this final communication phase is over, there are no nodes that can generate any new update messages or hosting/probe requests, hence radio silence is achieved.

4 Evaluation

The settings for the experimental setup took place in the same way as in Chapter 1.

4.1 Reference algorithms

As a reference for the results achieved by SP and FL, we run the ILA algorithm, described in detail in Chapter 1. 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 their 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. Then, the best candidate, as per *Eq. 4.2*, 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 combinations of agents on nodes, subject to their hosting capacity. 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 (*Eq. 4.4*). This means that the corresponding network cost represents a lower bound on what could be achieved even by an optimal algorithm.

4.2 Experiments

In a first set of experiments we compare the placements obtained for the 20-node networks and one app-10 application, as 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 (25 runs). No large variances were recorded.

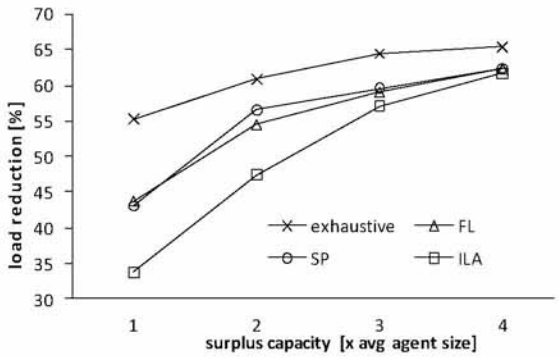

Fig 4.4 Load reduction vs. additional capacity (20 nodes, app-10).

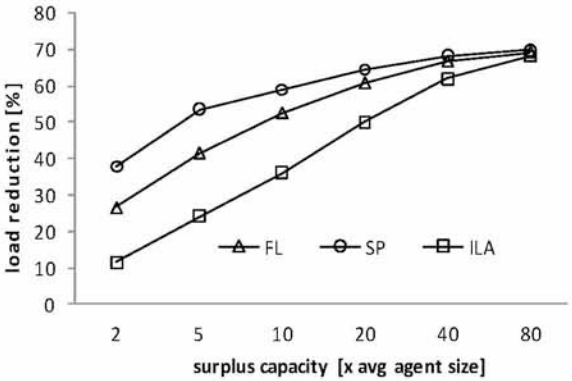

Fig 4.5 Load reduction vs. additional capacity (50 nodes, 15 applications).

Fig 4.4 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. Also, when the extra free capacity is (just) 2 times the average agent size, SP and FL 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 as free capacity increases, hence good placements can be reached without (any) agent evictions.

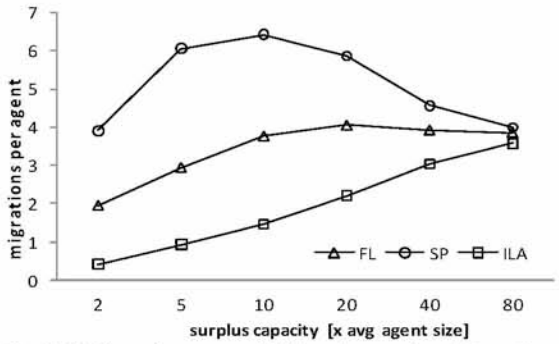

Fig 4.6 Migrations vs. additional capacity (50 nodes, 15 applications).

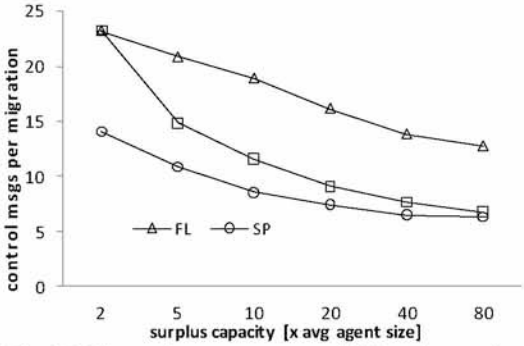

Fig 4.7 Control messages vs. additional capacity (50 nodes, 15 applications).

In the next experiments, we run the algorithms in the 50-node networks where we deploy a mix of fifteen applications (five app-50, five app-25 and five app-10 applications). 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 runtime complexity.

As it can be seen in *Fig 4.5*, the trend is similar to the one observed in the small-scale experiment. However, the improvement of SP and FL vs. ILA becomes impressive, from 320% to 220%, when the hosting capacity of nodes is limited. What is equally important to note is that SP performs better placements than FL. In fact, when capacity is tight SP produces placements that are almost 1.5x better compared to FL, which in turn produces placements that are close to 2.5 times as efficient compared to the ones produce by ILA.

The inferiority of FL vs. SP is attributed to the contention introduced by its flooding mechanism. In a large-scale system, it is very likely that several migrations and evictions will be attempted concurrently, which in turn leads to a large number of conflicts, where beneficial migrations are hindered by less beneficial ones (including evictions). Also, 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 4.6*, 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 the free capacity of nodes 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 scare, 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 4.7* 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. Moreover, for each beneficial migration, FL floods the network with probe requests and replies in order to find the best possible series of evictions, whereas SP picks a single path.

The high per-migration protocol overhead of ILA is also due to the fewer migrations accomplished compared to SP and FL. This is clearly visible when free capacity is tight. However, ILA continues to exhibit a non-negligible overhead even when nodes have abundant free capacity and the number of migrations performed is close to that of SP and FL. The reason is that even if a node is found occupied, ILA will still consider it (with 0.2 probability) as a

possible destination for a beneficial agent migration. As a result of contacting nodes in this optimistic way, the number of unsuccessful migration attempts remains high.

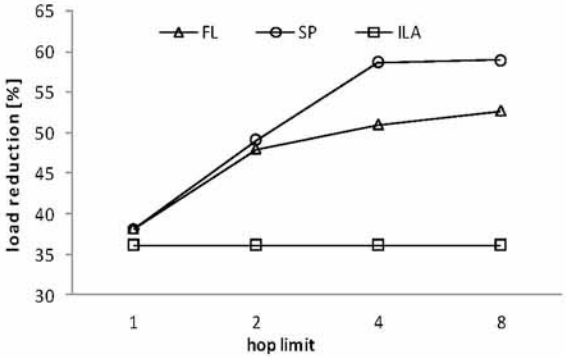

Fig 4.8 Load reduction vs. hop limit (50 nodes, cap +10, app-mix).

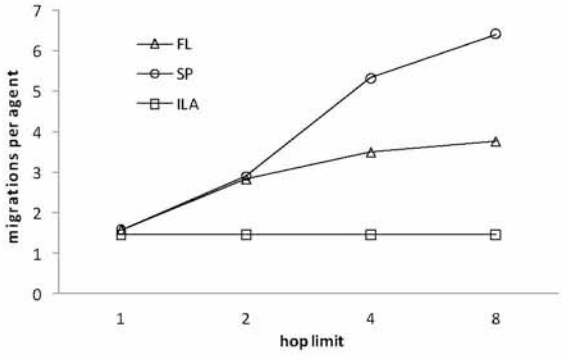

Fig 4.9 Migrations vs. hop limit (50 nodes, cap +10, app-mix).

In a final set of experiments, we measure the impact of limiting the hops of hosting and probe requests in SP and FL. We use again the 50-node networks and application mix of the previous experiments, while fixing the extra free node capacity to 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 *Fig 4.8*, *Fig 4.9* and *Fig 4.10*, respectively, with the hop limit varying from 1 to 8. The behavior of ILA is not affected by this parameter (the algorithm only issues 1-hop requests for beneficial migrations).

Both algorithms exhibit a similar performance for small hop limits. As the hop limit increases, SP clearly outperforms FL, due to the growing negative effects of the flooding approach. 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. Therefore 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). Worthwhile noting is also the fact that 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 grow at larger hop limits. On the other hand, the per-migration overhead of FL increases steadily due to the scalability problems of the flooding approach.

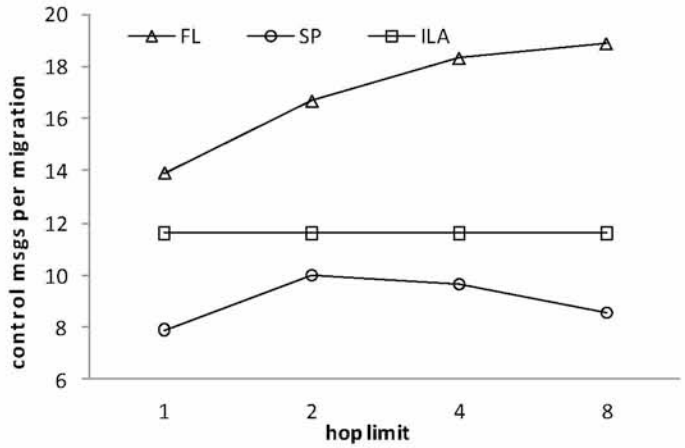

Fig 4.10 Control msgs vs. hop limit (50 nodes, cap +10, app-mix).

4.3 Result summary

Both SP and FL produce significantly better placements than ILA 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.

5 Conclusions

Here we described distributed algorithms for migrating agents between the nodes of a wireless embedded system in order to reduce application-level network traffic. Our approach introduces migrations that are non-beneficial on their own but free enough space on nodes in order to enable beneficial migrations, which can eventually lead to an overall better placement. We 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.

Part of this work is goint to be submitted in the following conference:

* N. Tziritas, P. Lampsas, S. Lalis, T. Loukopoulos, "Introducing Agent Evictions To Improve Application Placement in Wireless Embedded Systems" ICPADS *2011*.

Chapter 5

Online Algorithms for the Agent Migration Problem in Wireless Embedded Systems

1 Introduction

Previous works turn their attention to migrate agents without i) laying emphasis on the changes of traffic patterns, and ii) taking into account the cost of the migrations performed. However, assuming that traffic patterns are not static (they are subject to changes with the pass of time), performing migrations without taking their cost into consideration may prove crucial to the energy spent over the network. So we focus on the intractable problem of taking online decisions to migrate agents in order to reduce the overall network cost, considering the energy spent through the process of migrating an agent.

The difficulty of this problem lies in the fact that a decision should be made in advance of any knowledge about the future load/traffic changes. The implications of making bad decisions are that: i) the agent may be migrated far way from its center of gravity, paying in that way the cost of the wireless communication with its distanced relative agents; ii) the network will be burdened with the energy spent for mistakenly (due to a bad estimation) transfering an agent from some node to another one.

In this chapter we propose two online algorithms to decide which is the point in time that an agent should migrate to reduce its communication cost over the network, taking also into account its migration cost. Commonly, the algorithms proposed in the context of online decision problems are accompanied with their competitive ratios. The competitiveness is used to

compare the output of online algorithms when coming up against an input chosen by an adversary, to the output generated by the offline optimal algorithm. When the competitive ratio approximates 1, it means that the behavior of the online algorithm considered comes closer to optimal. Therefore, following the current, we evaluate the performance of the proposed algorithms by providing for each of them its competitive ratio along with a comprehensive proof. Specifically, the first algorithm achieves 1/3 competitive ratio assuming infinite capacity, while the second one 1/4 (no assumptions about infinite capacity).

This work is organized as follows. In Section 2 we describe the application and system model. Section 3 provides the proposed algorithms and discusses their competitiveness in a detailed way. In Section 4 the experimental setup is described along with a thorough evaluation of the proposed algorithms. This section also discusses the way we implemented a static offline optimal algorithm serving as a yardstick for the quality of our algorithms. Last, Section 5 concludes our work.

2 Application and System Model

The application and system model is much the same as the one described in Chapter 1. Below, we repeat the most relevant elements of the model, and introduce some extensions that are used to describe the algorithms and give the worst-case bound proofs.

Let l_{ijk}^{st} be the number of bytes exchanged between a_k hosted by n_i and other agents hosted by n_j in the time-interval $[s,t]$. Let $P_{ik}^{st} = \sum_{\forall j \neq i} l_{ijk}^{st} h_{ij}$ be the network communication cost due to the data exchanged between a_k hosted by n_i and the agents that are not co-located with a_k, under the time-interval $[s,t]$. Let M_{ijk} specify the migration of a_k from n_i towards n_j. Let B_{ijk}^{st} be the benefit/cost of M_{ijk}, subject to the collected message traffic statistics in the time interval $[s,t]$. The cost of M_{ijk} at time unit t is captured by MC_{ijk}^{t}. We assume that the time when a migration is performed is independent of the migration cost, therefore $MC_{ijk}^{t} = MC_{ijk}^{s}, \forall s,t$.

Let d and D specify the hop-awareness of an algorithm and the diameter of the network, respectively. If the cost of migrating an agent towards an 1-hop neighbor is equal to X, then the cost of migrating the agent in question towards an d-hop neighbor is equal to $d*X$, which is

formally stated by $MC^t_{ijk} = h_{ij} MC^t_{zmk} \mid h_{zm} = 1$. For simplicity, we assume that a migration occurs "instantly" and that the data traffic within the respective time interval $[t^-, t^+]$ is zero. *MT* denotes the migration threshold, i.e., the minimum required benefit for taking the decision to migrate an agent towards its center of gravity.

3 Algorithms

In this section we present three algorithms addressing the problem of taking into account the network cost incurred when migrating an agent. We prove also the competitiveness of each algorithm against the optimal algorithm.

3.1 Online algorithm based on discrete-time events (ADE)

The first algorithm is designed based on the (unrealistic) assumption that we have infinite memory. Based on this assumption, the algorithm, called ADE, can calculate the benefit/cost of migrating an agent based on any (sample) time interval ranging from the most recent point in time to any point in time in the past.

Let P1 be a property which forces this algorithm to migrate agents iff there is a time interval $[p, z^-]$ such that $B^{pz^-}_{ijk} \geq 2MC^z_{ijk} \mid i \neq j$ (this is referred to as "migration threshold"). The drawback of this algorithm is the increased memory complexity, since it needs to keep information about the exchanged data (volume of data, source/destination node that sent/received the data in question) in a discrete-time fashion; in order to be able to identify any $[p, z^-]$ where P1 is satisfied. Note that z^- should always map to the most recent point in time, while *p* can be any point in time past (that's why this algorithm needs infinite memory).

Theorem 1. ADE is 1/3-competitive.

Proof:

Consider that only one agent does exist into our system (a_k), which is hosted on n_i. Initially we let ADE perform a migration iff there is any $[p, z^-]$ such that $B^{pz^-}_{ijk} > 0$ (P1'). Obviously, when both ADE and the optimal algorithm perform no migrations, the competitive ratio is equal to 1. Therefore we focus on the case where the loads are such that ADE chooses to perform

migrations. Assume there is a time interval $[x, y^-]$ such that $B_{ijk}^{xy^-} > 0$, which means that ADE will perform M_{ijk} (y^- is the most recent point in time). In this case, the optimal algorithm may decide to perform or not to perform M_{ijk}.

Assumption 1

Let's start with the case where ADE decides that M_{ijk} should not be performed. In this case, the competitive ratio between the optimal algorithm and ADE becomes:

$$\frac{P_{ik}^{sx^-} + P_{ik}^{x^+y^-} + P_{ik}^{y^+c}}{P_{ik}^{sx^-} + P_{ik}^{x^+y^-} + P_{jk}^{y^+c} + MC_{ijk}^{y}} \quad \text{Eq. 5.1}$$

We now consider when this ratio becomes as small as possible (the worst case). We observe that the smaller the values of $P_{ik}^{sx^-}, P_{ik}^{x^+y^-}, P_{ik}^{y^+c}$ the smaller the ratio. Note that $P_{ik}^{x^+y^-} \geq h_{ij}$ otherwise M_{ijk} would not be performed, which contradicts our assumption that migrations are performed when P1' holds true. Also, we notice that when $P_{jk}^{y^+c}$ increases the ratio decreases. In combination with the fact that the value of enumerator should be kept as small as possible, we conclude that only $I_{jik}^{y^+c}$ should be greater than zero; else $P_{ik}^{y^+c}$ could not be equal to zero hence the enumerator (and the ratio) would increase. Of course this means that there is a time interval $[y^+, c]$ such that $B_{jik}^{y^+c} > 0$, which means that ADE will perform an additional migration, in the reverse direction, namely M_{jik}, as dictated by our assumption that ADE performs a migration when P1' holds true. Therefore the ratio is expressed by *Eq. 5.2*, which equation implies that the ratio is independent of the hops between n_i and n_j.

$$\frac{h_{ij} * 1}{h_{ij} * 1 + h_{ji} * 1 + h_{ji} MC_{gmk}^{y'} + h_{ij} MC_{gmk}^{y}} = \frac{1}{2 + 2MC_{gmk}^{y}} \mid h_{gm} = 1 \quad \text{Eq. 5.2}$$

If we assume that ADE additionally performs X such back-and-forth migrations, as the previously discussed case, then the ratio becomes:

$$\frac{X}{2X + 2X * MC_{gmk}^{y}} = \frac{1}{2 + 2MC_{gmk}^{y}} \mid h_{gm} = 1 \quad \text{Eq. 5.3}$$

In other words, the worse-case ratio is independent of the number of back-and-forth migrations.

Assumption 2

Now let's consider the case where optimal algorithm decides that M_{ijk} should be performed. In this case, the ratio becomes:

$$\frac{P_{ik}^{sx^-} + P_{jk}^{x^+c} + MC_{ijk}^{x}}{P_{ik}^{sx^-} + P_{ik}^{x^+y^-} + P_{jk}^{y^+c} + MC_{ijk}^{y}} \qquad \text{Eq. 5.4}$$

It is obvious that the worst case scenario is that $P_{ik}^{sx^-}, P_{jk}^{x^+c}$ are as small as possible, while $P_{ik}^{x^+y^-}, P_{jk}^{y^+c}$ as large as possible. However we note that the optimal algorithm will "immediately" decide for this migration, before ADE collects the "necessary" load information. It follows that $[y^+, c] \subset [x^+, c] \Rightarrow P_{jk}^{y^+c} < P_{jk}^{x^+c}$, and thus $P_{jk}^{y^+c}$ should be as small as possible too.

Here we make an extra assumption: Namely, that the maximum application-level message size is smaller than the cost to perform any migration M_{jik}.

Based on this assumption, $P_{ik}^{x^+y^-}$ cannot be larger than the maximum application-level message, because ADE takes the decision to perform M_{jik} as soon as P1' holds true, and this condition is checked each time a message is sent/received. Therefore, the ratio becomes:

$$\frac{MC_{gmk}^{y}}{2MC_{gmk}^{y}} = \frac{1}{2} \mid h_{gm} = 1 \qquad \text{Eq. 5.5}$$

Note that this is smaller than *Eq. 5.2*. Also note that in this case, naturally, since both ADE and the optimal algorithm decided for a migration, *Eq. 5.5* is independent of the number of migrations. Consequently, the worst case is when the benefit $B_{ijk}^{xy^-}$ is such that optimal algorithm does not perform migration, but ADE decides for a (back-and-forth) migration.

Fine-tuning the migration threshold

Let's see if we are able to improve the performance of ADE, by fine-tuning the migration threshold. Consider the case when ADE performs a migration iff there is any $[p, z^-]$ such that $B_{ijk}^{pz^-} \geq MT$ (P1''). For sake of simplicity we assume that $MT = h_{ij} * x$, hence the aforementioned ratios described by *Eq. 5.2* and *Eq. 5.5* become:

$$\frac{x}{2x + 2MC^{y}_{gmk}} \mid h_{gm} = 1 \qquad \text{Eq. 5.6}$$

$$\frac{MC^{y}_{gmk}}{x - MC^{y}_{gmk} + 2MC^{y}_{gmk}} \mid h_{gm} = 1, x \geq MC^{y}_{gmk} \qquad \text{Eq. 5.7}$$

Note that *Eq. 5.6* decreases when *MT* increases to the point that $x = 2MC^{y}_{gmk}$ (note that the enumerator cannot be greater than $2MC^{y}_{gmk}$, independently of the value of *x*). Also, *Eq. 5.7* increases when *MT* increases, and remains smaller than *Eq. 5.6* as long as $x < 2MC^{y}_{gmk}$, and becomes equal to *Eq. 5.6* if $x = 2MC^{y}_{gmk}$. Also, note that in *Eq. 5.7* $x \geq MC^{y}_{gmk}$ due to our assumption that the optimal algorithm performs M_{ijk}. Therefore, due to the equality of *Eq. 5.6* and *Eq. 5.7*, the competitive ratio between the optimal algorithm and ADE is given by *Eq. 5.8*, provided that ADE takes the decision for migration iff there is any $[p, z^{-}]$ such that $B^{pz^{-}}_{ijk} = 2MC^{z}_{ijk}$ (in practice, the decision could be taken if $B^{pz^{-}}_{ijk} > 2MC^{z}_{ijk}$, due to the transmission/arrival of a large application-level message). We should point out that *Eq. 5.8* is independent of the number of migrations. Therefore our assumption of only one agent into our system is valid.

$$\frac{MC^{y}_{gmk}}{3MC^{y}_{gmk}} = \frac{1}{3} \mid h_{gm} = 1 \qquad \text{Eq. 5.8}$$

3.2 Algorithm based on sliding window and discrete-time events (ADE-SW)

Since ADE is non-applicable due to the assumption of infinite memory, we resort to a modified version of it to bound the memory needed for keeping message traffic statistics. ADE-SW uses for each generic agent (a_k) a *sliding window* (w_k) of maximum size S_k to keep the data exchanged between this agent and its relatives. Let w^{i}_{k} and $t(w^{i}_{k})$ denote the ith entry of w_k and the point in time this entry was inserted into w_k, respectively; then $t(w^{i}_{k}) > t(w^{j}_{k}) \mid i < j, \forall i, j$, in other terms jth entry was inserted into w_k prior to ith one. An entry w^{i}_{k} represents a tuple (a_m, *vd*), whereby *vd* is the volume of data exchanged between a_m and a_k at $t(w^{i}_{k})$. Putting it otherwise, w^{i}_{k} represents the size of the message sent/received by a_k at $t(w^{i}_{k})$, provided that a_m

is the destination/source agent. Let mr_k $(1 \le mr_k \le S_k)$ be the *sliding marker* for w_k, which points to an entry of w_k. $s(mr_k)$ denotes the number of entries this marker leaves behind (including the one it points to) when it slides towards the most aged entry.

Each w_k is implemented as a list. Each time a generic agent (a_k) sends/receives a message it pushes at the back of the list an entry, provided that the new size of the list is not greater than S_k. Otherwise, the first entry of the list is removed before inserting the new one at the end of the list. Initially the marker is set to $s(mr_k)^{th}$ entry of w_k, provided that this entry does exist; otherwise is set to the most ancient entry. The entries lying behind the sliding marker (including the one it points to) are deemed marked. ADE-SW considers whether the migration threshold of an agent has been reached or not, by taking into account only the information associated with the marked entries. Each time it decides that a_k cannot migrate anywhere (no benefit), it slides the respective marker by $s(mr_k)$ entries (i.e. $mr_k = mr_k + s(mr_k)$), and reconsiders whether the migration threshold of a_k has been exceeded or not. This procedure repeats itself till the marker points to the most aged entry, where if the corresponding agent cannot be migrated then the sliding marker is reset to $s(mr_k)^{th}$ entry.

In case ADE-SW decides to implement a migration, then the respective agent (a_k) is migrated along with only the marked entries. There are two reasons for doing so: i) if we don't transfer this information and an agent cannot migrate directly towards its center of gravity (limited hop-awareness), then each intermediate migration of its own will be delayed (due to P1); ii) if we resort to transfer all entries of the window associated with the migrating agent, then it is probable for that agent to migrate back-and-forth, due to outdated information.

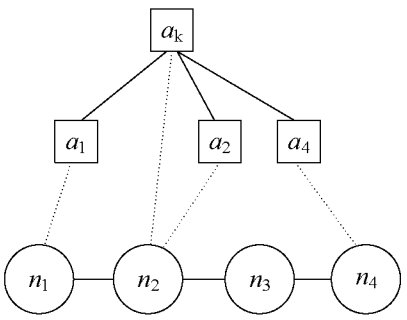

Fig 5.1 Application deployment

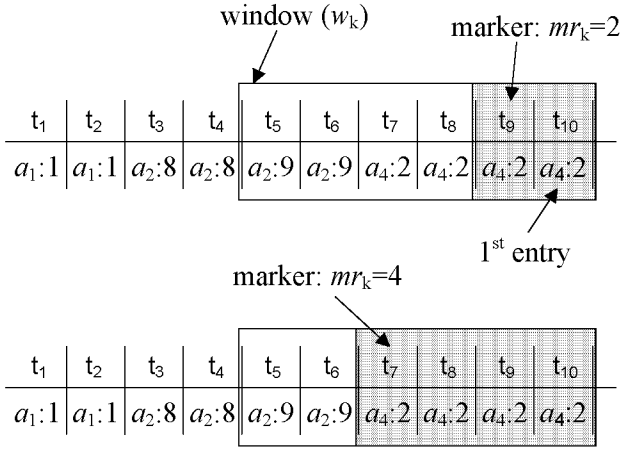

Fig 5.2 Sliding window and marker

Consider the example illustrated in *Fig 5.1*, whereby a_k is a generic agent while the rest non-generic ones. A solid edge means that the involved agents communicate with each other, while a dashed edge represents the hosting node of the involved agent. Let the cost of migrating a_k towards an 1-hop away neighboring node be 4 ($MC^{y}_{gmk} = 4, | h_{gm} = 1, \forall y, k$), $s(mr_k) = 2$ and $S_k = 6$. In *Fig 5.2* we set out an example of how an agent is migrated, making use of the respective sliding window and marker. A column in an upper row represents the point in time where a_k sent/received a message towards/from an adjacent agent of its own, with t_1 representing the most aged message, while t_{10} the most recent one. A column in a bottom row signifies the respective message size along with the involved destination/source agent. Recall that the window slides towards the most recent messages (i.e., $t_5..t_{10}$).

Initially, mr_k is set to 2; however due to P1, whereby $B^{t_9,t_{10}}_{23k} = 4 < 2 * MC^{y}_{23k} = 8$, M_{23k} is considered non beneficial and the sliding marker is set to 4. Therefore, in the next iteration $B^{t_7,t_{10}}_{23k} = 8$, which means that a_k is forced to migrate towards n_3 (along with the marked entries $t_7 \ldots t_{10}$). It is worth noticing that without the sliding marker M_{23k} cannot be identified (since $B^{t_5,t_{10}}_{23k} = -10$). In the sequel, the same steps are followed on n_3, forcing a_k to eventually migrate towards n_4 (along with the marked entries $t_7 \ldots t_{10}$), without even needing to collect any extra information. It should be stressed that if a_k migrated towards n_3 a) without the entries of the respective window then M_{34k} would be procrastinated till there is a t_n such that $B^{t_m,t_n}_{23k} >= 8, t_{10} \le t_m < t_n$ (case i); b) along with all entries of the respective window, then n_3 would decide to migrate a_k back to n_2, since when $mr_k = 6$ then $B^{t_5,t_{10}}_{32k} = 10$. This back-and-forth migration would continue in a perpetual way, till new messages arrived (case ii).

3.3 Algorithm based on aggregation of events (AGE)

AGE is designed to reduce the memory requirements of ADE-SW, whereby the information about the collected events of same affinity is kept aggregated. Specifically, for each generic agent a_k hosted on node n_i and each m-hop neighbor node n_j, where $0<=m<=d$, where d is equal to the network awareness, a load variable R_{ijk} is used to record the accumulated message traffic associated with a_k between n_i and n_j as follows:

1) if $0<=m<d$, R_{ijk} records the accumulated load between a_k and all agents that reside on n_j;

2) if $m=d$, R_{ijk} records the accumulated load between a_k and (a) all agents that reside on n_j and (b) all agents that reside on nodes that are more than m hops away from n_i and communicate with n_i via n_j.

Note that if d is equal to the network diameter D, i.e., in case of "full network awareness", (2) becomes equivalent to (1) because no two nodes can be more than d hops away from each other, so there can be no case (2b). Finally, R_{iik} records the load between a_k and all agents that are co-located with it on n_i. Due to the fact that the load is stored in an aggregated fashion, the benefit of M_{ijk} is now represented by $B_{ijk}^{z^-} > 0$ (instead of $B_{ijk}^{sz^-} > 0$), where z^- is the most recent point in time.

The algorithm works as follows: Initially, when a_k is created on n_i, the load variables R_{ijk} for each (relevant) neighbor node n_j is initialized to *0*. From that point onwards, R_{ijk} is updated by adding the number of bytes sent/received by a_k to/from node n_j.

Each time R_{ijk} is updated, the following checks/actions are made/taken:

i) If $B_{ijk}^{z^-} \geq 2MC_{ijk}^{z}$, M_{ijk} is performed.

ii) If $B_{iik}^{z^-} > 0$, the load variables are reset to 0 (on the current host).

iii) Else, if $\sum_{\forall j} R_{ijk} \geq RT_k$, the load variables are reset to 0 (on the current host n_i); RT_k is referred to as the so-called reset threshold

Note that the resetting of the load variables in (ii) and (iii) corresponds to a form of "aging", making sure that a recent change in the application traffic pattern will be considered promptly, instead of waiting until it "overrules" the aggregated load history.

Theorem 2. AGE is 1/4 competitive, when $d=D$ and $RT_k \approx 3.2MC_{gmk}^{y} \mid h_{gm} = 1$.

Proof:

Consider AGE without (ii) and (iii).

We initially assume that AGE performs only one migration M_{ijk} at time unit y. The performance of AGE worsens as the value of $l_{iik}^{sy^-}$ increases. This is because (i) must hold true, which means that the network cost produced by $l_{ijk}^{sy^-}$ (i.e., $l_{ijk}^{sy^-} h_{ij}$) must become equal to $2MC_{ijk}^{y} + l_{iik}^{sy^-} h_{ij}$.

When h_{ij} increases the performance of AGE worsens, so we conclude that the worst-case scenario is to have $h_{ij} = D$.

Similarly, the performance of AGE keeps worsening the larger the value of (another load variable) $l_{ifk}^{sy^-}, f \neq i, j$; now, in order for $B_{ijk}^{sy^-} = 2MC_{ijk}^{y}$, the network cost produced by $l_{ijk}^{sy^-}$ should be equal to $2MC_{ijk}^{y} + l_{iik}^{sy^-} h_{ij} + l_{ifk}^{sy^-} h_{fj}$. For the sake of proof, we need h_{fj} to be as large as possible. Note that h_{fj} would be equal to D iff n_f: (a) is not in the path between n_i and n_j; and (b) does not use any node in the path between n_i and n_j (including n_j) as a router for data towards n_i. This is not possible, though, since then h_{fj} would have to be equal to D+1 (we assumed $h_{ij} = D$). However, it is feasible for h_{fj} to be equal to D-1 (see *Fig 5.3*) which is the next largest possible value. This is the case when: (a) n_f is in the path between n_i and n_j, provided that h_{if}= 1; or (b) n_f uses a node n_u as a router for data towards n_j, provided that $h_{iu} = 1$ and $h_{uj} = D-1$.

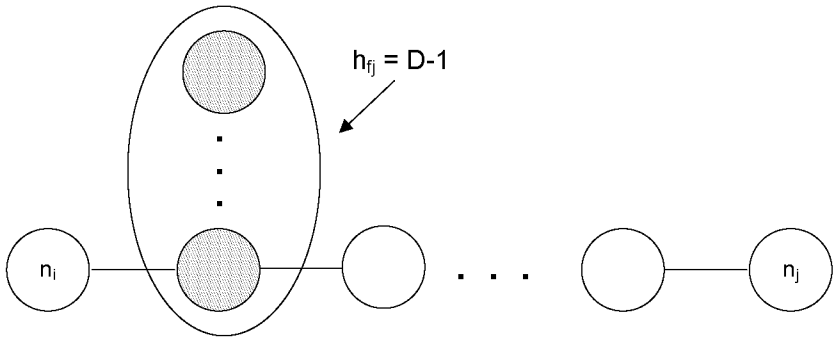

Fig 5.3 when h_{jf} becomes equal to D-1

Note that since $l_{ifk}^{sy^-}$ can be arbitrarily large, without loss of generality, we can assume that all other load variables $l_{if'k}^{sy^-}, f' \neq i, j, f$ are equal to zero.

Consequently, in order for AGE to perform M_{ijk} (i.e., for (i) to hold true), n_j must produce network cost of $l_{ijk}^{sy^-} = l_{ifk}^{sy^-}(D-1) + l_{iik}^{sy^-} D + 2MC_{ijk}^{y}$. The worst case for AGE is for the optimal algorithm to decide M_{ijk} before n_j starts producing any network cost ($l_{ijk}^{sz^-} = 0$), thus incurring only the migration cost plus $l_{ifk}^{sy^-}$ which is unavoidable (for both algorithms). Therefore the ratio vs. the optimal algorithm (which decides for that migration before AGE, ideally when $l_{ijk}^{sz^-} = 0$), becomes:

$$\frac{I_{ifk}^{sy^-} + MC_{ijk}^{z}}{I_{ifk}^{sy^-} + I_{ijk}^{sy^-} + MC_{ijk}^{y}} = \frac{I_{ifk}^{sy^-} + MC_{ijk}^{z}}{I_{ifk}^{sy^-} + I_{ifk}^{sy^-}(D-1) + I_{iik}^{sy^-} D + 3MC_{ijk}^{y}} \quad \text{Eq. 5.9}$$

Note that *Eq. 5.9* is independent of the number of migrations, since for each additional migration (following the same rationale) the cost for each algorithm is doubled. Therefore our initial assumption of only one migration does not affect the competitive ratio.

We observe that when $I_{ifk}^{sy^-} = 0$ and $I_{iik}^{sy^-}$ tends to infinity, the ratio tends to 0 (provided that $I_{iik}^{zy^-} = 0$). We avoid this case by applying (ii). Then, $I_{iik}^{sy^-}$ cannot be greater than $I_{ifk}^{sy^-}$, hence the worst case is to have $I_{ifk}^{sy^-} = I_{iik}^{sy^-}$ (provided that $I_{ifk}^{zy^-} = 0$ and $I_{iik}^{zy^-} = 0$, with z being the point in time where the optimal algorithm performs M_{ijk}). Let also $I_{iik}^{sy^-} = X$ for the sake of readability. As a result, the ratio becomes equal to that expressed by *Eq. 5.10* (also taking into account that $h_{ij} = D$, hence $MC_{ijk}^{y} = D * MC_{gmk}^{y} \mid h_{gm} = 1$):

$$\frac{X + D * MC_{gmk}^{z}}{2X * D + 3D * MC_{gmk}^{y}} \mid h_{gm} = 1 \quad \text{Eq. 5.10}$$

$$\frac{MC_{gmk}^{z}}{2X + 3MC_{gmk}^{y}} \mid h_{gm} = 1 \quad \text{Eq. 5.11}$$

When D tends to infinity the competitive ratio worsens, therefore we reformulate *Eq. 5.10* into *Eq. 5.11*. By applying (iii) and setting a finite RT_k: (a) X cannot be arbitrarily large, which means that *Eq. 5.11* decreases; and (b) AGE becomes reactive to load changes. It is prudent to choose RT_k greater than the double cost of migrating a_k towards 1-hop neighbor ($RT_k > 2 * MC_{gmk}^{y} \mid h_{gm} = 1$), else we compromise the performance of AGE (load variables will be reset before being able to decide for any migration). We also note that when resetting the load variables there is a case of resetting a variable R_{ijk}, while some $B_{ijk}^{y^-} > 0$. The greater the value of RT_k the greater the loss of AGE vs. the optimal algorithm, however $B_{ijk}^{y^-}$ cannot be greater than $2MC_{ijk}^{y}$ due to (i).

$$(RT_k - 2MC_{gmk}^{y} - 1)/2 \mid h_{gm} = 1 \quad \text{Eq. 5.12}$$

$$(RT_k - 2MC_{gmk}^{y} - 1)/2 + 2MC_{gmk}^{y} - 1 \mid h_{gm} = 1 \quad \text{Eq. 5.13}$$

Therefore, the worst case now becomes that of having the previous scenario with the difference that AGE is forced to reset the load variables as many times as possible before deciding for M_{ijk}, provided that when these resets take place $B_{ijk}^{z^-}$ is as large as possible. We stress that the worst point in time (let z^-) of resetting the load variables for AGE is when $B_{ijk}^{z^-} = 2MC_{ijk}^{z} - 1$, else AGE would perform M_{ijk}. We should also point out that in order to reset load variables when $B_{ijk}^{z^-} = 2MC_{ijk}^{z} - 1$, we need from some nodes to incur a load given by *Eq. 5.12*, and for n_j to incur a load given by *Eq. 5.13*. Note that the nodes contributing to *Eq. 5.12*: (a) cannot be in the path between n_i and n_j, and (b) cannot use any node in the path between n_i and n_j (including n_j) as a router for data towards n_i, else *Eq. 5.13* would be increased and property (iii) would be violated. It follows that the only node that can contribute to *Eq. 5.12* is n_i, else AGE would create cost greater than that of *Eq. 5.12*.

Summing up, the performance ratio between AGE and the optimal algorithm becomes:

$$\frac{D^* f(RT_k - 2MC_{gmk}^{y} - 1)/2 + MC_{ijk}^{y}}{D^* f[(RT_k - 2MC_{gmk}^{y} - 1)/2 + 2MC_{gmk}^{y} - 1] + 3MC_{ijk}^{y}} \mid h_{gm} = 1 \wedge h_{ij} = D \qquad \text{Eq. 5.14}$$

with f denoting the number of resets. For simplicity, we eliminate the "-1"s (without loss of generality since the ratio worsens). We can observe that the ratio changes with the variation of f and RT_k. For the case where RT_k is less than $3MC_{ijk}^{y}$ the ratio worsens when f tends to infinity. In terms of case where RT_k is equal to or greater than $3MC_{ijk}^{y}$ the ratio worsens when f tends to zero. However we omit the case where f tends to zero since in that case *Eq. 5.14* is dominated by *Eq. 5.11*, which means that the worst-case ratio is given by *Eq. 5.11*. As a result *Eq. 5.14* becomes:

$$\frac{RT_k - 2MC_{gmk}^{y}}{RT_k + 2MC_{gmk}^{y}} \mid h_{gm} = 1 \qquad \text{Eq. 5.15}$$

Due to the fact that X should be as large as possible without enabling the resetting of loads, we conclude that the resetting threshold should be expressed by *Eq. 5.16*. Therefore *Eq. 5.11* is transformed into *Eq. 5.17*.

$$RT_k = l_{iik}^{sy^-} + l_{ifk}^{sy^-} + 2MC_{gmk}^{y} + 1 \approx 2X + 2MC_{gmk}^{sy^-} \mid h_{gm} = 1 \qquad \text{Eq. 5.16}$$

$$\frac{MC^{y}_{gmk}}{RT_k + MC^{y}_{gmk}} \mid h_{gm} = 1 \qquad \text{Eq. 5.17}$$

We recall that $RT_k > 2MC^{y}_{gmk} \mid h_{gm} = 1$, else we could not be able to perform migrations across the whole network, and therefore the performance of AGE would decrease. We observe that when RT_k increases *Eq. 5.15* decreases, while *Eq. 5.17* increases. Given the above, and due to the fact that the competitive ratio is given by the smaller equation between *Eq. 5.15* and *Eq. 5.17*, we turn to equating them to get the value(s) of RT_k which maximize(s) the competitive ratio. Therefore, two roots result from that operation, the negative and the positive one. Of course the negative one is out of consideration, since RT_k cannot be negative. The positive root is roughly equal to $3.2MC^{y}_{gmk} \mid h_{gm} = 1$, with the competitive ratio being approximately equal to $1/4$.

4 Experiments

This section presents an experimental evaluation of the algorithms based on simulations performed on top of NS2 [85]. First we describe the experimental setup and then we present and discuss the results of indicative experiments.

4.1 Setup

The network topologies and application structures were produced in the same way as for the previous chapters. Five different network topologies were generated, while 3 different application types were produced with (50, 22), (25, 12) and (10, 5) (non-generic, generic) agents, referred to as app50, app25 and app10, respectively. For each application type we produced 5 different application structures. The initial agent placement on nodes was random, while agents were assigned sizes randomly selected between 100 and 1,000 bytes. For each combination network topology and application structure an experiment was conducted (75 in total) taking the average of them.

Contrary to the previous chapters, in this one we consider traffic patterns that are not stable throughout the duration of an experiment. Unless otherwise stated, we assume that a non-generic agent can change between two modes M_H and M_L, signifying a change in the frequency

of the messages are sent out by the respective agent. Specifically, when an agent is in M_H mode then it sends 10 times more messages against than M_L mode. We consider four different types of traffic pattern T(UH), T(H), T(L) and T(UL); with these reflecting that an agent changes between M_H and M_L modes in ultra high, high, low and ultra low rate, respectively. Specifically, in terms of T(UH) each agent remains in a mode from 1 to 10 time periods, chosen randomly. As far as T(H), T(L) and T(UL) are concerned, their corresponding periods range between (1, 100), (50, 500) and (100, 1000), respectively. We differentiate between three application families F_1, F_2 and F_3, whereby at most 1, 2 and 3, respectively, agents belonging to the same parent can be in L_H mode simultaneously.

As the main metric for our comparison, we use the network load incurred by the resulting placements of our algorithms. We also devise a static offline optimal algorithm serving as a yardstick for the quality of the solutions derived by the proposed algorithms. In order to get the static offline optimal solution, we resort to GRAL* of which the input is chosen to be slightly different against the online algorithms. Specifically, GRAL* takes as input the static load associated with each application edge. Specifically, the static load of an edge represents the volume of data that would be exchanged between the incident agents to this edge, if we let the involved agents exchange messages for a specified time according to an adopted type of traffic pattern, e.g., T(H).

We observed that ADE-SW variants have different trend when the traffic is based on T(UH) pattern compared to the rest ones, so we chose to plot the results separately for each case.

4.2 Considering T(H), T(L) and T(UL)

ADE-SW can be parameterized into two dimensions, with the first one being the migration threshold, which is common for both algorithms; while the second one being the number of window entries marked each time the marker slides towards the most aged entries. From now on a variant of ADE-SW will be referred to as ADE-SW-(MT, s(mr)); with MT and s(mr) reflecting the first and second dimension, respectively. AGE is also parameterized into two dimensions, with the first one being also the migration threshold, while the second one being the reset threshold. From this time forward a variant of AGE will be referred to as AGE-(MT, RT); with RT reflecting the reset threshold. This set of experiments is based on F_1 application family.

Fig 5.4 concerns the case where the size of the sliding marker varies between 1 and 500, considering all types of traffic patterns excluding T(UH). As observed, the performance of

ADE-SW variants deteriorates as the size of the sliding marker increases. This is expected since such an increase means that the migration decisions will be based on further aged information, rendering in that way ADE-SW slower in identifying changes in message traffic pattern. Another remark is that the gap between variants is growing as the changes in traffic pattern become less intense. This is explained by the fact that the benefit of migrating an agent towards a direction is continuously growing as long as its center of gravity does not change into another direction. Specifically, an agent's center of gravity change more vigorously in T(H) pattern, rendering some migrations less fruitful, since in that case it is almost the same for an agent to remain in a node instead of migrating back and forth due to load changes.

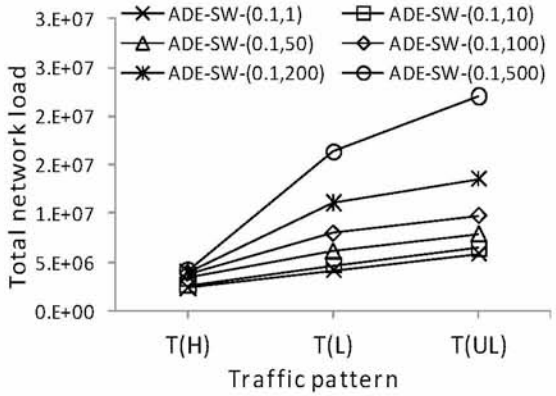

Fig 5.4 ADE-SW behavior when varying the size of sliding marker

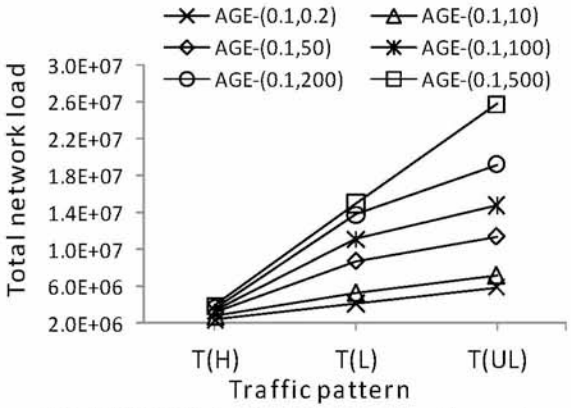

Fig 5.5 AGE behavior when varying the reset threshold

Fig 5.5 shows the behavior of AGE when varying the reset threshold between 0.2 and 500. It is observed that the performance worsens when increasing the reset threshold. This is anticipated since such an increase incurs a proportional delay when deciding to perform a migration. Specifically, an increase to the reset threshold means that the migration decisions are based on more outdated information, so the delay is attributed to the time the algorithm needs to offset this outdated information and finally take the decision to perform a migration.

Note that we conducted the same experiment for both ADE-SW and AGE keeping fixed the size of the sliding marker and the reset threshold at 1 and 0.2, respectively; while varying the migration threshold. The results showed that the performance of both algorithms worsens as the migration threshold increases. Hence we conclude that the best variants are ADE-SW-(0.1, 1) and AGE-(0.1, 0.2). The observation that the variants are more distanced with each other when the changes in traffic pattern become less intense is explained through the respective remark in the previous paragraph.

4.3 Considering T(UH)

In this set of experiments the application family continues being F_1, however the traffic patter type considered is T(UH). *Fig 5.6* shows the behavior of ADE-SW when varying the size of sliding marker. It is shown that ADE-SW variants have an opposite trend against the previous case (*Fig 5.4*). This is due to frequent changes in traffic pattern, increasing in that way the probability of not amortizing the cost of the agent migrations performed. Therefore it is not lucrative for an agent to be reactive to all those changes. Note that, as discussed earlier, an agent becomes less reactive to changes when increasing the size of the sliding marker, therefore the "variant-500" achieves the best performance with the "variant-200" following closely. This performance is attributed to the fact that the greater the size of the sliding marker the less reactive the algorithm to traffic changes, and therefore the less the migrations performed (*Fig 5.7*).

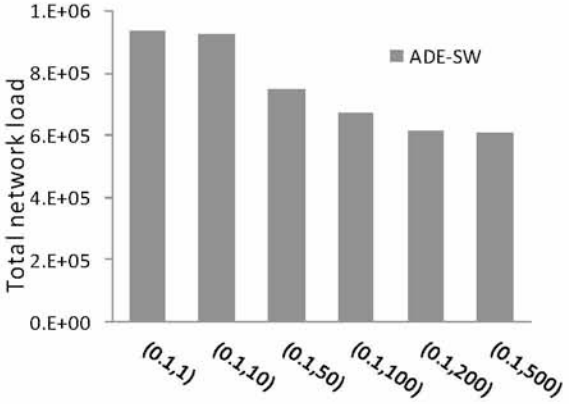

Fig 5.6 ADE-SW behavior when varying the size of sliding marker (the migration threshold is kept fixed at 0.1).

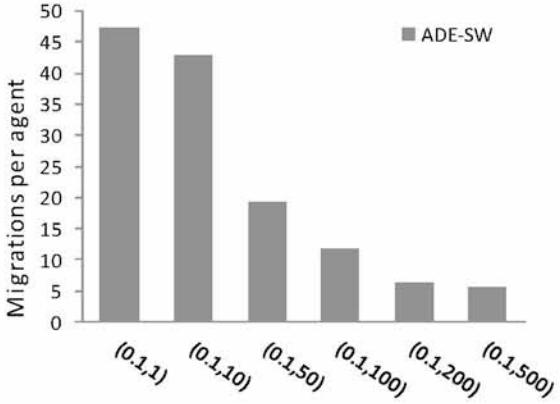

Fig 5.7 migrations performed by ADE-SW when varying the size of sliding marker (the migration threshold is kept fixed at 0.1).

We conducted the same experiment with that depicted in *Fig 5.6* with the difference that we kept the size of the sliding marker fixed at 500 (best variant), while varied the migration threshold. The results of this experiment are shown in *Fig 5.8*, whereby the performance of ADE-SW degenerates when increasing the migration threshold. This is due to the fact that the benefit of migrating agents is kept in low levels due to frequent load changes, therefore an increase to the migration threshold may lead to migrations that their cost is hardly (or cannot be) amortized.

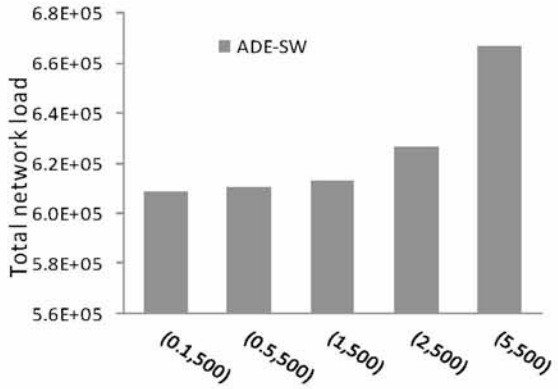

Fig 5.8 ADE-SW behavior when varying the migration threshold (the size of the sliding marker is kept fixed at 500).

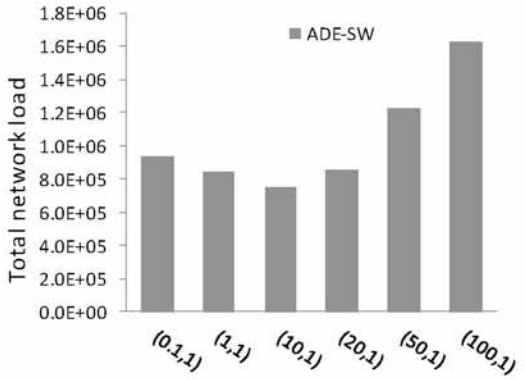

Fig 5.9 ADE-SW behavior when varying the migration threshold (the size of the sliding marker is kept fixed at 1).

We also carried out the same experiment with that depicted in *Fig 5.8*, with the difference that the size of the sliding marker is kept fixed at 1 instead of 500. In *Fig 5.9* there are two observations (i) the trend of this experiment is opposite to the previous one as long as the migration threshold is less than or equal to 10; (ii) while these trends coincide when the migration threshold is equal to or greater than 20. The first observation is explained by the fact that ADE-SW becomes enough reactive to load changes when the size of the sliding marker is 1; as a result the migration threshold serves as a repressing factor regarding the reactiveness of the algorithm to those changes. The second observation is attributed to the fact that when the migration threshold becomes enough large, then an agent may be not migrated even in the case where all the relative agents of its own belong to the same direction. This is witnessed in *Fig 5.10*, where it can be seen that the number of migrations lessens rapidly when increasing the migration threshold. It should be stressed that among all these cases, the best results are obtained through ADE-SW-(0.1, 500).

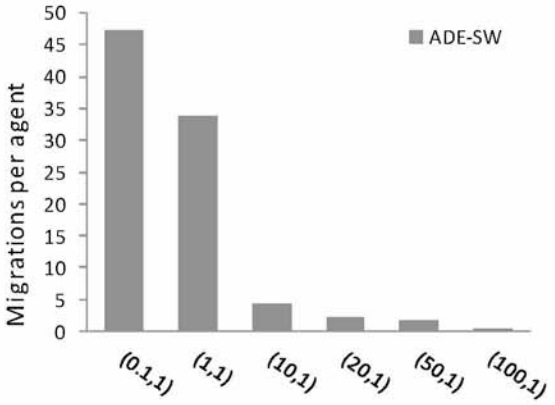

Fig 5.10 Migrations performed by ADE-SW when varying the migration threshold (the size of sliding marker is kept fixed at 1).

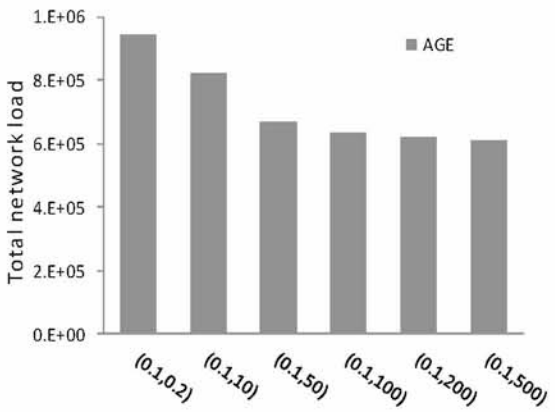

Fig 5.11 AGE behavior when varying the reset threshold (the migration threshold is kept fixed at 0.1).

Fig 5.11 shows the performance of AGE when varying the reset threshold, while keeping fixed the migration threshold at 0.1. As it can be observed, AGE becomes more fruitful when increasing the reset threshold. This increase means that AGE becomes less reactive to frequent load changes, thus yielding placements wasting less resources in terms of the wireless communication. Note that we decided to omit the rest experiments conducted for AGE, since the observations were exact the same as previously. It should be noticed that AGE-(0.1,500) outperforms all AGE variants.

4.4 Comparing our algorithms to the offline optimal algorithm

In this set of experiments we pick the best variants of AGE and ADE-SW for each type of traffic pattern and draw a parallel between them and the static offline optimal algorithm (i.e., GRAL*).

For the first experiment (*Fig 5.12*) the application family keeps being F_1. A first observation is that the performance of AGE and ADE-SW is identical. This is expected since in case of (i) T(UH) both algorithms gather enough information in order to decide whether a migration is beneficial or not; (ii) T(H), T(L) and T(UH) both algorithms take the decision to migrate an agent as early as possible. Another remark is that the offline optimal algorithm outperforms AGE and ADE-SW when the load changes take place in a rapid fashion. This is why in such a situation it is difficult for an online algorithm to decide whether a migration will bear fruits or not. Therefore the best decision is to perform only the essential migrations, however such a decision is only applicable in an offline fashion. This is illustrated in *Fig 5.13*, where in T(UH) plot both online algorithms try to perform as less migrations as possible. Of course our

algorithms are able to adjust their thresholds in such a way to become almost identical with the offline algorithm.

It is also observed that online algorithms achieve up to 80 percent load reduction against the offline optimal algorithm, provided that the type of traffic pattern is other than T(UH). This is attributed to the fact that when load changes become less frequent then online algorithms are able, due to their nature, to perform more beneficial migrations than the static offline optimal algorithm. This is partly explained through *Fig 5.13*, whereby online algorithms perform by far more migrations against the static offline optimal algorithm, given that traffic changes take place in a slower pace than T(UH).

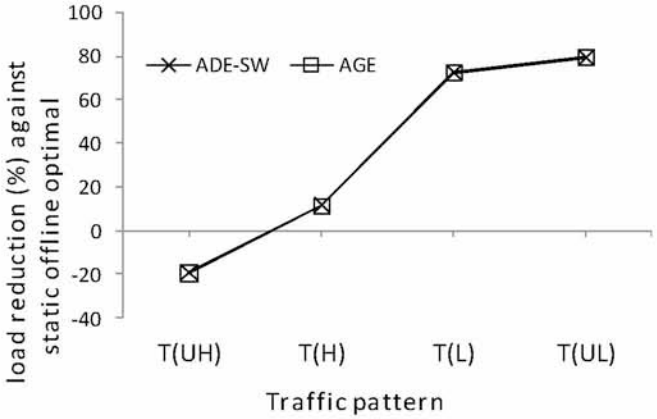

Fig 5.12 AGE and ADE-SW against the optimal offline algorithm (the application family is kept fixed at F_1).

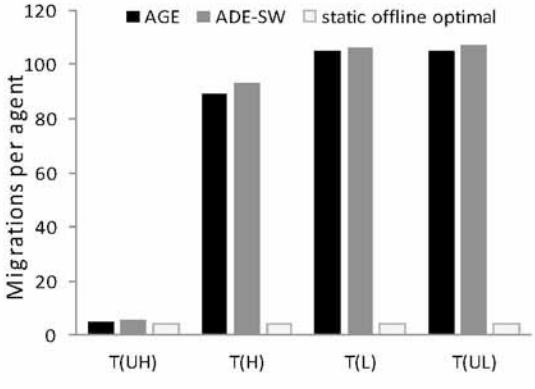

Fig 5.13 Migrations performed (the application family is kept fixed at F_1).

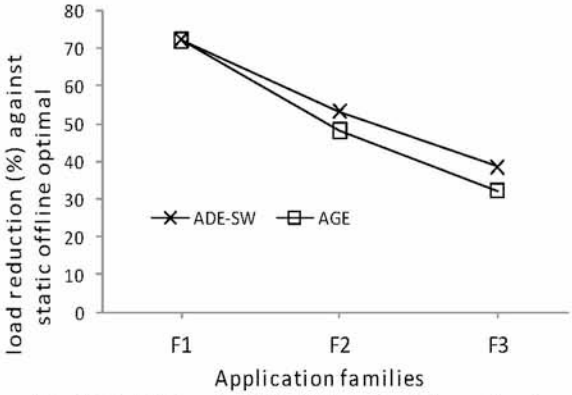

Fig 5.14 AGE and ADE-SW against the optimal algorithm when varying the application families (T(L) is kept fixed).

Last, we ran another experiment where the application family is varied among F_1, F_2 and F_3. Taking a look at *Fig 5.14*, we can see that both algorithms are getting worse when going from

F_1 to F_2, and finally to F_3. As discussed earlier, the index of an application family reflects the maximum number of the sibling agents that can be simultaneously in M_H state. Hence, the probability of a migration to become less beneficial is increased. Also another remark is that the performance of AGE becomes less gainful against ADE-SW. This is ascribed to the inferiority of AGE to promptly identify a beneficial agent migration when the involved agent receives simultaneously data from more than one relatives of its own. Actually the proof of Theorem 2 is based on such a scenario, whereby AGE fails to identify a beneficial migration in a prompt manner due to threshold reset.

5 Conclusions

In this work we introduced the problem of deciding which is the point in time that a migration should be performed to reduce the total network cost, taking into account the network cost when performing a migration. We proposed two online algorithms solving the problem without knowing in advance the future traffic changes. The competitive ratios of the proposed algorithms are also discussed thoroughly, giving in that way a flavor of the quality of each algorithm. Experiments were conducted to take an insight about the performance of our algorithms against the static offline optimal algorithm. This work differs from the previous ones in that the migration decisions are taken in an online way taking also into account the migration cost.

Part of this work is going to be submitted in the following conference:

* N. Tziritas, T. Loukopoulos,P. Lampsas, S. Lalis, “Online Algorithms for the Agent Migration Problem in Wireless Embedded Systems” IPDPS *2012*.

Chapter 6

On Reconfiguring Embedded Application Placement on Smart Sensing and Actuating Environments

1 Introduction

In this chapter we introduce the *agent reconfiguration problem* (ARP), in light of a smart home or smart office environment with a central monitoring entity, e.g., a desktop computer or a set-top box. This entity is responsible for deciding about the agents' placement, having full knowledge of the present placement scheme, the network, and the respective smart node capabilities. The goal is to place agents in nodes having the required resources (generic or non-generic), so that communication traffic is minimized, thus reducing battery consumption and saving bandwidth. The main differences with the previous chapters are: i) that non-generic agents are able to migrate, taking into account their non-generic resource demands; and ii) that the reconfiguration decision (migrations) is made in a centralized way (on central monitoring entity); iii) the application is structured as a general graph instead of a tree.

This work is modeled as a graph coloring problem; where the proposed algorithm is based on to perform agent exchanges (i.e., migrations) between nodes to eventually reduce the total network cost. It should be stressed that the graph is modeled in such a way to include the migration cost, favoring in that way agent migrations of small size. Note also that the knapsack component [56] is used to check feasibility issues involving the agent exchanges between nodes.

The rest of this chapter is organized as follows: the rest of Section 1 illustrates the application model; Section 2 provides the system model and problem formulation ; Section 3 illustrates two

algorithmic approaches solving ARP, with the first one being based on the graph coloring problem while the second one on greedy techniques In Section 4 both algorithms are evaluated on small- and large-scale experiments, where in the former ones an exhaustive algorithm takes place for comparison reasons; while in Section 5 we give our conclusions.

1.1 Application Model

In this chapter we use a roughly different application model against the previous ones. Specifically, the agents participating into an application may communicate with agents of other applications for reusability reasons. Consider two applications are to be deployed into a network, with the first one needing to create humidity and temperature gathering agents, while the second one brightness and temperature ones. Assume the first application is deployed as usual by creating the humidity and temperature agents. It is prudent, in light of scarce resources provided by such a network, to force the second application to not create temperature agents but use the already existing ones. However, the middleware may set a limitation on the number of applications an agent can participate to, due to overloading an agent.

2 Problem Definition

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

2.1 System model

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

A non-generic agent is not only dependent on the computational resources at the destination; it requires also that *non-generic resources* be provided by the destination node (i.e. sensing or actuating capabilities). A binary $N \times A$ eligibility matrix L is used to encode whether a node has the required *non-generic resources* (thus is eligible to hold the agent) as follows: $L_{ik}=1$ if n_i

provides the required by a_k specific resources, 0 otherwise. Recall also that non-generic agents belonging to the same application and providing the same functionality (e.g. temperature gathering agent) must not reside at the same node. We model it through an $A \times A$ binary *mutual exclusion* matrix F, whereby $F_{kw}=1$ if a_k must not reside at the same node with a_w, 0 if no such requirement is necessary.

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

2.2 Problem formulation

A binary $N \times A$ matrix P is used to encode agent placement at nodes as follows: $P_{ik}=1$ if a_k is in n_i, 0 otherwise. The APR problem can then be stated as follows: given an initial placement P^{old} of application agents on nodes, define a new placement P^{new} so that the overall network load due to agent communication is minimized. As a secondary optimization target we also require that the network cost due to the migrations performed in order to switch from the initial placement P^{old} to the new one P^{new} is also minimal. The network load T due to agent communication is given by *Eq. 6.1*. Thus, the benefit in agent communication terms by switching from P^{old} to P^{new} described by *Eq. 6.2*.

A single migration incurs a cost proportional to the agent size and the hop distance between the start and destination node. We assume that there exists a single monitoring node (let n_m) which also acts as an entry point for the arriving agents in the system (e.g., for security reasons) and keeps an immutable copy of all agents' code. Migrations are performed by sending a copy of the agent's code from n_m and the agent's status from the node where the agent currently resides. For simplicity, we assume that the size of the status is negligible, compared to the code size, which is denoted by s_k. Therefore, given an initial placement P^{old} and the one that must be implemented P^{new}, the total migration cost M can be computed by *Eq. 6.3*.

$$T = \sum_{k=1}^{A}\sum_{m=1}^{A}(C_{km}\sum_{i=1}^{N}\sum_{j=1}^{N}h_{ij}P_{ik}P_{jm}) \qquad \text{Eq. 6.1}$$

$$B = T^{old} - T^{new} \qquad \text{Eq. 6.2}$$

$$M = \sum_{k=1}^{A} \sum_{i=1}^{N} P_{ik}^{new} (1 - P_{ik}^{old}) h_{mi} s_k \quad \text{Eq. 6.3}$$

Minimizing agent communication cost (*Eq. 6.1*) and migration cost (*Eq. 6.3*) are conflicting, since *Eq. 6.3* is minimized if P^{new} is the same as P^{old}. Intuitively, *Eq. 6.3* acts as an overhead which can be fully or partially offset by the reduction in agent communication cost (*Eq. 6.2*), depending on whether P^{new} will remain unchanged for a sufficient large time. Let a be a constant depicting the importance of migration cost over agent communication. Then the APR problem can be stated as: given an initial agent placement P^{old} find a new placement P^{new} such as *Eq. 6.4* is optimized, with respect to constraints described by *Eq. 6.5*,*Eq. 6.6*,*Eq. 6.7*, and *Eq. 6.8*.

$$\max \; D = B - yM \quad \text{Eq. 6.4}$$

$$\sum_{k=1}^{A} P_{ik}^{new} r(a_k) \le r(n_i), \; \forall i \quad \text{Eq. 6.5}$$

$$\sum_{i=1}^{N} P_{ik}^{new} = 1, \; \forall k \quad \text{Eq. 6.6}$$

$$P_{ik}^{new} (1 - L_{ik}) = 0, \; \forall i,k \quad \text{Eq. 6.7}$$

$$F_{kw} P_{ik}^{new} P_{iw}^{new} = 0, \; \forall i,k,w \quad \text{Eq. 6.8}$$

Eq. 6.5 states that node capacity constraints should not be violated. *Eq. 6.6* enforces that each agent should be placed at exactly one node. In addition, this placement must be eligible in terms of specific resources (*Eq. 6.7*) and there should not be conflicts with other agents residing at the same node (*Eq. 6.8*). By *Eq. 6.5* it is easy to see that the relevant ARP decision problem is NP-complete having (among others) a knapsack component [56]. In the following section we present heuristics to tackle it.

3 Algorithms

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

3.1 The ARP problem with 2 nodes

Consider the ARP problem for the case where the system consists of two nodes n_1 and n_2 and a monitoring node n_m. All nodes are assumed to have 1-hop distance between each other. Assume a total of 5 agents are already placed at the system's nodes as follows: a_1, a_2 and a_3 are placed at n_1 and a_4, a_5 at n_2. *Table 6.1* depicts the load generated due to agent communication, as well as the agents' resource requirements.

Table 6.1 Agent communication load and resource requirements

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

Let the capacity of the two nodes (resource wise) be: $r(n_1)$=7 and $r(n_2)$=5. Assuming migrations incur no cost and that no specific resources or mutual exclusion constraints do exist, ARP can be transformed into a graph coloring problem as follows. In a first phase, the agent communication graph $G(V, E)$ is constructed, whereby the vertices of the graph correspond one to one with the agents, and an edge (a_k, a_w) exists if a_k and a_w communicate with each other. Each edge has a weight $w(a_k, a_w)$ which equals the communication cost between a_k and a_w across both directions, i.e., $w(a_k, a_w) = C_{kw} + C_{wk}$. Furthermore, each vertex has a weight $w(a_k)$ equaling the amount of generic resources a_k demands. Let *Fig 6.1* represent such a graph in terms of the agents hosted by n_1 and n_2.

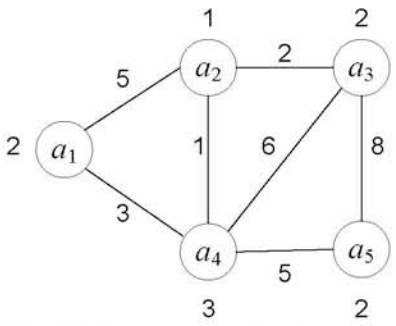

Fig 6.1 Agent communication graph

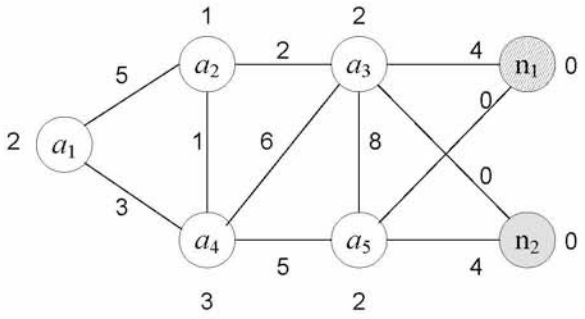

Fig 6.2 Extending the communication graph

In a second phase, graph G is extended by adding two vertices, with these vertices corresponding to the node pair hosting the agents represented by G. These vertices have 0 weight and are colored through a 2-color scheme (e.g. red, black). Note that the rest vertices

(agent-vertices) remain uncolored for the time being. In order to take into account the cost of migrating an agent, for each agent-vertex there are two extra edges towards the two node-vertices. Since such an edge represents the migration cost, its weight is set to zero if the incident agent-vertex to that edge is not hosted by the incident node-vertex; otherwise the weight equals to the cost of migrating the agent represented by the incident agent-vertex, from the monitoring node towards the node not hosting it. *Fig 6.2* illustrates the above extension, only for the agents a_3 and a_5, assuming that all agent sizes is 8 and that the constant a=0.5. Red vertex (n_1) is shown striped, while black vertex (n_2) is shown grayed. Since all migrations are assumed to be performed via the monitoring node (hop distance of 1 against n_1 and n_2), all edges whereby the migration cost must be charged have a weight of 4 (equals α*agent size*hop distance).

The specific resources demands (in terms of an agent) are included in the model by coloring the respective agent-vertex. For instance, if in the example $L_{21}=0$, then a_1 vertex will be painted in red, i.e., a_1 will be forced to stay at n_1 (red vertex). Finally, mutual exclusion constraints are included by adding coloring constraints for the corresponding agents. For example, in modeling that $F_{kw}=0$, it is equivalent to say that a_k and a_w vertices must have different colours.

Putting all these together, an agent that is differently colored against its current hosting node, should migrate towards the other node in the system (same-coloured). Hence, ARP can be re-stated as follows: try to paint each agent-vertex in one of the available colours, with respect to our constraints, in such a way that the network communication cost is minimized.

3.2 The agent exchange algorithm

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

The algorithm works in iterations. In each iteration, the edge with the highest weight is selected and the vertices it connects with are merged, since this weight represents a benefit. Specifically if the incident vertices to that edge: i) are both agent-vertices, then this benefit comes from placing the agents, included on that vertices, on the same node (they communicate heavily); and ii) are an agent-vertex and a node-vertex, this benefit comes from placing the agent(s) represented by agent-vertex on node represented by node-vertex. In case the merged vertices have a mutual exclusion constraint, the merging is not performed and the edge connecting them is colored grey (i.e., not to be considered further). Otherwise, the new vertex has the cumulative

weight of the previous ones and their remaining edges. If any of the vertices belonging to the merged vertex is colored then the merged vertex will also be colored (with the same color). In case the two vertices to be merged are colored with different color each, merging is not performed and the respective edge becomes grey. *Fig 6.3* shows the resulting graph by merging a_3 with a_5.

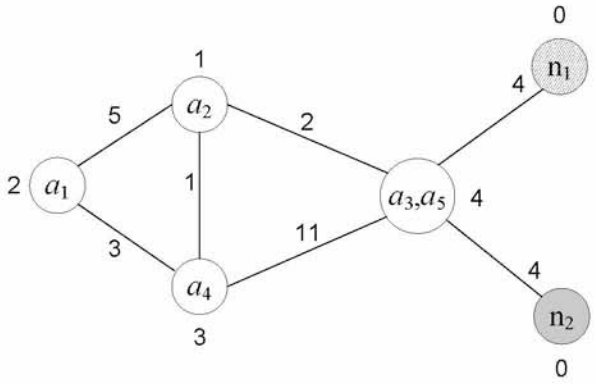

Fig 6.3 Resulting graph after merging.

Each time two vertices are merged, AXA attempts to find if a feasible vertex coloring does exist in the new graph. To this end it solves knapsack two times, once for n_1 and once for n_2, with the candidate objects being the a_k vertices (the size of each object being the weight of the vertex). In the previous example (*Fig 6.3*), by solving knapsack on n_1 (the red node) we get the following objects to be placed: $\{a_1, a_2, \{a_3, a_5\}\}$, filling the resource capacity of n_1 which is 7. Having obtained a knapsack solution for n_1, the algorithm checks if the remaining objects fit in n_2. In the example only a_4 remains which fits in n_2 since $r(n_2)$ was assumed 5. If so, the algorithm keeps the merged vertex without coloring it and proceeds with the next iteration. Otherwise, the algorithm attempts to find a valid placement by solving knapsack for n_2 (the black node) and checking whether the remaining objects fit at n_1. If after trying both knapsack solutions AXA is unable to find a valid placement involving all the objects, it backtracks to the graph state before merging, marking the edge under consideration as grey.

The algorithm continues in the same fashion till either all the remaining agent-vertices are colored, whereby performs the corresponding migrations; or edges are colored in gray, where no migrations are performed.

3.3 Extending to N nodes

Tackling the case of $N>2$ nodes is done with the *pair-wise reconfiguration algorithm* (PRA), the pseudocode of which is shown in *Fig 6.4*.

```
found:=true;
while (found)
   found:=false;
      for i=1 to N
       for j=1 to N
           apply AXA over (ni, nj) pair;
           if D>0 then found:=true; keep AXA changes;
              else discard AXA changes;
           endif
       endfor
   endfor
endwhile
```

Fig 6.4 Pseudocode of PRA

PRA iterates through all node pairs applying AXA. If during an iteration AXA manages to define a better placement according to *Eq. 6.4*, the process reiterates, otherwise it ends producing the final agent placement. In order for AXA to successfully optimize locally, i.e., within a node pair, the agent placement, adaptations are required to the way agent communication load and migration costs are modeled. We illustrate them through an example.

Assume the network of *Fig 6.5*, with 7 nodes plus the monitoring node n_m. Let the agents of *Table 6.1* be already placed on n_2 and n_5 as follows: n_2 has a_1, a_2, a_3 and n_5 has a_4 and a_5. In other terms, n_2 and n_5 in this example have the same role as n_1 and n_2 in the example of Sec. 3.1. Assuming only these agents exist in the network, the equivalent graph colouring problem is similar to the one in *Fig 6.2*, with the exception being that the hop count must be taken into account both on edges representing agent communication (a_k, a_w) and on edges representing migration cost (n_i, a_k). So all $w(a_k, a_w)$ edge weights will be multiplied by a factor of 3 (the hop distance between n_2 and n_5), while all edge weights $w(n_i, a_k)$ will be multiplied by the hop distance between n_m and the node of the opposite color with which n_i was painted. For instance, $w(n_2, a_2)$ will remain 4 since the distance between n_m and n_5 (the black node) is 1, while $w(n_5, a_4)$ will now be 16 since h_{m2}=4. Fig. 7 depicts the resulting problem graph. For clarity, the edges between (a_1, a_2, a_4) and (n_1, n_2) are omitted, as previously.

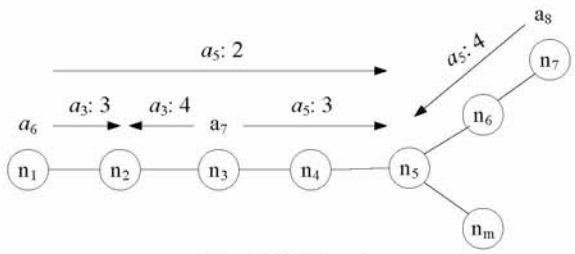

Fig 6.5 Network

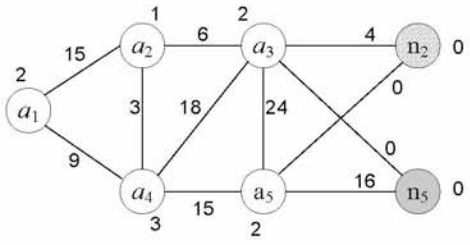

Fig 6.6 Resulting problem graph

In the general case, agents placed on nodes other than the pair in question (n_2, n_5) might generate load towards some of the agents placed on the pair. *Fig 6.5* gives an example, whereby 3 more agents exist, namely: a_6 which is placed at n_1, a_7 at n_3 and a_8 at n_7. The figure also shows the load these agents generate towards the ones placed at n_2 and n_5, specifically: $C_{63}+ C_{36}=3$, $C_{65}+ C_{56}=2$, $C_{73}+ C_{37}=4$, $C_{75}+ C_{57}=3$ and $C_{85}+ C_{58}=4$.

Such external (to the node pair) load must be incorporated to the graph coloring model in order for it to map to ARP correctly. This external load can be viewed as another form of node related cost in the problem graph, as was the case with migration. Consider for instance the migration of a_5 from n_5 to n_2. Aside from the migration cost of 16 to transfer a_5 from n_m to n_2 there will also be a change on the cost in terms of the external load directed to/from a_5. For instance, the load generated by (a_5, a_8) communication will not incur a cost of 8, but rather a cost of 20 since the hop distance between the two agents will increase from 2 to 5. In order to incorporate the above case in the problem graph it suffices to augment: i) $w(n_5, a_5)$ by the network cost incurred if a_5 moved to n_2, i.e., 20; and ii) $w(n_2, a_5)$ by the incurred load if a_5 stayed in n_5, i.e., 8. Repeating the process for all the external loads of a_5 results in $w(n_5, a_5)$ being augmented by a factor of: 20 (a_8's load) + 3 (a_7's load) + 2 (a_6's load) for a total of 25, and $w(n_2,a_5)$ being augmented by: 8 (a_8's load) + 6 (a_7's load) + 8 (a_6's load) for a total of 22. *Fig 6.7* illustrates the final graph coloring transformation for the example of Fig. 6. Again, to avoid cluttering, only edges between n_2, n_5 and a_3, a_5 are shown.

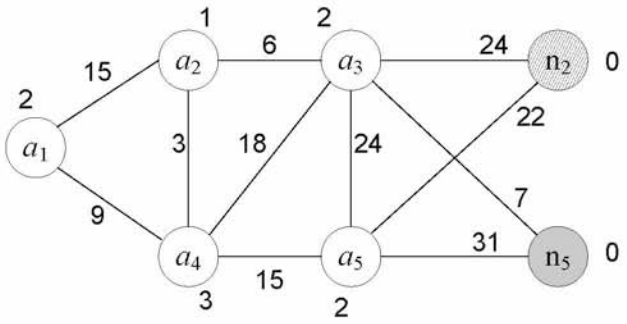

Fig 6.7 Resulting problem graph

However, a subtle change must be made to AXA in order for it to function properly. Recall, that AXA selects the link of highest weight and attempts to merge the incident vertices to that link. The rationale for the decision is to attempt to place together agents communicating heavily with each other. So, if a_3 and a_5 are placed together, then the communication load among them will be alleviated and a benefit of $w(a_3, a_5)=24$, will occur. However, the same is not true when considering edges involving a node-vertex. For instance, if a_5, n_5 are merged the actual benefit will not be 31, but rather the cost difference between placing a_5 at n_5 and at n_2. According to

this, AXA may begin the coloring/merging process from a less beneficial edge, thus leading to inferior solutions. For instance, in this example AXA will begin with (a_5, n_5) having actual benefit equal to 9 (31-22), instead of (a_5, a_3) having actual benefit equal to 24. For this reason, at the sorting step of AXA all edges of the form (n_i, a_k) do not participate with their weights, but rather with the weight difference: $w(n_i, a_k) - w(n_j, a_k)$, assuming n_i and n_j are the system nodes for which AXA runs.

3.4 Greedy algorithmic approach

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

Starting from the initial placement, *Greedy* iteratively selects an agent to migrate and performs the migration. Specifically, at each iteration all *A*N* possible migrations are considered and the one that optimizes *Eq. 6.4* the most, subject to the constraints *Eq. 6.5* - *Eq. 6.8*, is selected. The process is repeated until no further beneficial migration can be defined.

4 Experiments

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

4.1 Experimental setup

Due to the fact that the POBICOS middleware is currently under development we conducted the experimental evaluation using simulation experiments. The details of the simulation setup are briefly discussed below.

Network generation. Two types of networks were constructed, one with 7 and one with 30 nodes. In both networks an extra node played the role of the monitoring node. Nodes were

placed randomly in a 100×100 2D plain and assumed to be in range of each other if their Euclidean distance was less than 30 distance units. In the resulting network topology graph, a spanning tree was calculated and acted as the corresponding tree-based routing topology.

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

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

Other parameters. The size of agents varied uniformly between 100 and 1,000 data units. All the non-generic agents that have the same parent were assumed to share one common special resource requirement and had a mutual exclusion constraint among them. Non-generic agents with different parents were assumed to differ in at least one special resource requirement. In the experiments we begin with an initial placement and run the algorithms to define a better one. This initial placement is derived by placing the non generic agents first. Specifically for every group of non-generic agents with the same parent (let *ng* in cardinality), $(1+\beta)ng$ nodes (randomly selected) were assumed to have adequate special resources to hold the agents, i.e., for a node n_i and an agent a_k such as above, $L_{ik}=1$. Unless otherwise stated, constant β takes a value

of 0.5. In the initial placement the non-generic agents were placed randomly to nodes having the required functionality in such a way so as to respect mutual exclusion constraints as well. Having placed the non-generic agents, generic agents were placed afterwards, again in a random fashion. Last, in the experiments we assume that the computational resource of interest is memory and that all nodes start with an initial capacity equaling the size of the agents assigned to them by the initial placement.

4.2 Comparison against the optimal algorithm

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

First we recorded the performance of the algorithms regarding the quality of the placement scheme they reach, as a percentage of the optimal performance. Assuming that in the initial placement *init* communication load is incurred per time unit, that in the optimal scheme *opt* communication load is incurred and that in the placement calculated by the algorithms *alg* communication load is incurred, the percentage of the optimal performance achieved by an algorithm is characterized by the ratio: (*init-alg*)/(*init-opt*), i.e., how much load reduction an algorithm achieves compared to the optimal. *Table 6.2* presents the results for PRA and Greedy for two different load types: *lavg* and *lsum*. We also varied the amount of extra free capacity available at the system nodes. So for instance *lavg(2)*, means that each node had just enough capacity to hold the agents allocated there in the initial placement, plus extra space equaling 2 times the average agent size.

Table 6.2 Solution quality compared to the optimal

	lavg(1)	***lavg(2)***	***lavg(3)***	***lsum(1)***	***lsum(2)***	***lsum(3)***
Greedy	*81.7%*	*88.4%*	*95.4%*	*86.8%*	*86.9%*	*86.9%*
PRA	*85.5%*	*100%*	*100%*	*89.8%*	*100%*	*100%*

We can observe from *Table 6.2* that PRA constantly outperforms the simpler Greedy algorithm. In fact, the difference between PRA and the optimal scheme is not large when capacity is tight (plus one extra space for an agent), while with a less tight constraint, PRA achieves the optimal performance. It is also worth noting that the Greedy algorithm never achieves an optimal performance.

Table 6.3 Migrations performed

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

We also recorded in *Table 6.3* the number of migrations performed by each of the algorithms. Recall, that the application type used was app-10, involving 10 non-generic agents and roughly 6 generic, for a total of 16 agents. Results here are mixed, with PRA doing more or less migrations compared to Greedy depending on the scenario. However, the fact that in certain cases where PRA achieves the optimal, e.g., *lavg(3)*, *lsum(3)*, PRA also performs less migrations compared to Greedy, illustrates even more the merits of our approach.

4.3 Experiments with a larger network

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

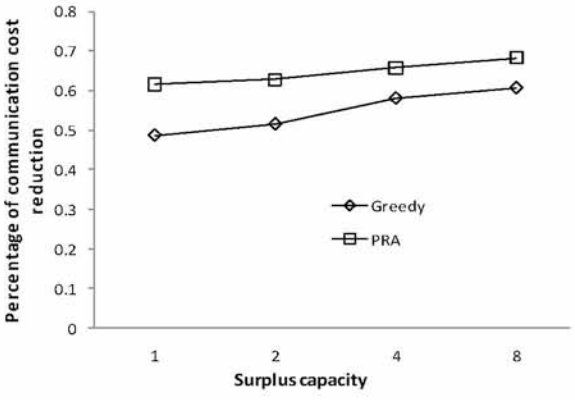

Fig 6.8 Performance of the algorithms against increased node capacity

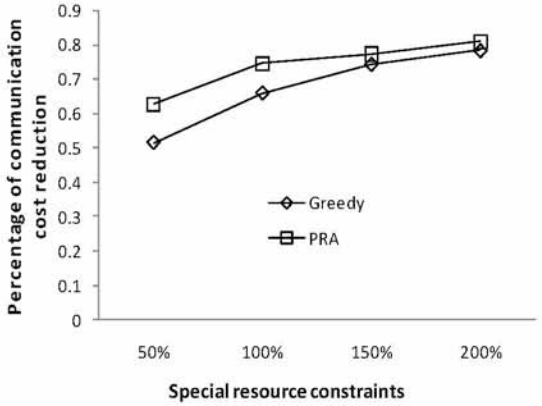

Fig 6.9 Performance of the algorithms when relaxing special resource constraints

Fig 6.8 demonstrates the performance of the algorithms as more capacity is added at each node e.g., the value of 4 in the x-axis means that each node has capacity equaling the necessary one to hold the agents initially placed there, plus 4 times the average agent size. The first thing to notice, is that the achievable saves by both algorithms increase to the surplus capacity at the

nodes, which is expected since with tighter capacity agents that should have been placed together might not be able to do so. Notice that PRA manages to reduce the initial load by more than 60% in all cases and by roughly 10% more compared to Greedy, a fact that further reinforces the viability of our approach.

Last, in *Fig 6.9* we measure the performance of the algorithms as the special resource constraints become less tight. Recall from Sec. 4.1 that each non-generic agent group having the same parent is assumed to require the same special resource. Assuming *ng* is the group size (5 in our case) then $(1+\beta)ng$ nodes are assumed to provide such a special resource. In the x-axis of Fig. 10 we vary the constant β by 50%, 100%, 150% and 200% essentially increasing the number of possible hosts (special resource wise) from 5 to 7.5, 10, 12.5 and 15.

As expected, with more candidate locations available for each agent, there is an increased optimization potential compared to the random initial placement. Both PRA and Greedy exploit this potential resulting in a performance increase (PRA achieves roughly 80% savings by the end of the plot). Again PRA outperforms Greedy with their difference becoming small in the 150% and 200% case. In a sense, this result means that as the nodes of the system become more homogeneous, Greedy might be a viable alternative, whereas for heterogeneous networks PRA is a clear winner.

4.4 Discussion

Summarizing the experiments we can state the following: (i) judging from the optimization margin left by the initial placement, any random solution to ARP will probably be particularly inefficient; (ii) PRA achieves performance close to optimal particularly if the computational capacity constraint is not very tight; and (iii) simpler algorithms based on a pure greedy paradigm cannot achieve equivalent performance compared to PRA, particularly in networks with a heterogeneity degree as is usually the case in a smart home environment.

We would also like to mention that the increased performance offered by PRA does not involve a prohibitive runtime cost. All the experiments were run in an ordinary laptop carrying an Intel Pentium Dual CPU T3200 processor at 2GHz with 3GB of memory. Even in the larger setup of Sec. 4.3 the running time of PRA never exceeded a couple of seconds.

5 Conclusions

In this work we tackled the APR problem by iteratively solving it for node pairs. To do so we illustrated a graph coloring problem transformation, and proposed an algorithm (AXA) to derive a solution for the equivalent problem. Through simulation experiments the final algorithmic scheme (PRA) was found to outperform a simpler greedy approach, while achieving the optimal solution in many cases. The main differences of this work against the previous ones are: i) the application structure is structured as a graph (instead of a tree); ii) besides the generic agents, the non-generic ones are migratable provided that the destination nodes have the required non-generic resources.

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

Part of this work has been published in the following book chapter:

* N. Tziritas, S.U. Khan, T. Loukopoulos, “On Reconfiguring Embedded Application Placement On Smart Sensing and Actuating Environment”, in Intelligent Decision Systems in Large-Scale Distributed Environments, Springer, New York, USA, 20011, ISBN 978-3-642-21270-3, Chapter 11.

Chapter 7

Algorithms for Energy-Driven Agent Placement in Wireless Embedded Systems with Memory Constraints

1 Introduction

In this chapter, we address the basic problem of placing a single new agent (software component) in a network of nodes taking into account both the available memory and remaining battery of each node. Priority is given to agent acceptance while maximizing the lifetime of the first node that will run out of battery. As it turns out, the problem of accepting a new agent, without paying any attention to the communication and battery costs, is quite challenging in itself. The reason is that even if no single node has enough memory to host a new agent, it may still be possible to free sufficient space at some node by migrating one or more agents to other nodes.

Our solutions are centralized, assuming a single point of entry, which has sufficient computing and energy resources and decides about agent placement having a global overview of the system state. For the POBICOS system, this could be a set-top box or a desktop computer which acts as the coordinator of the home network, keeping track of the applications deployed in the system in order to take good agent placement decisions. We assume that the node network topology and communication traffic between agents is known to the coordinator; in reality, this information would have to be collected at runtime using some kind of monitoring protocol – but this does not change the core of the problem investigated here.

The rest of this work is organized as follows: Sec. 2 formulates the agent placement problem; Sec. 3 presents algorithms that accept a new agent without making any lifetime optimization;

Sec. 4 gives two greedy reconfiguration algorithms to optimize node lifetime once an agent is successfully placed in the system; Sec. 5 illustrates two branch and bound heuristics that accept a new agent while optimizing node lifetime in a "simultaneous" way; Sec. 6 describes how to implement the defined placements efficiently; experimental evaluation is included in Sec. 7; finally, Sec. 8 includes the concluding remarks.

2 Problem Definition

2.1 System model

Let the system comprise of N nodes denoted by n_i, $1 \le i \le N$ and let $m(n_i)$ be the memory capacity of the i^{th} node measured in abstract data units. The agents to be deployed in the system are denoted by a_k, $1 \le k \le A$, each having size $m(a_k)$. A binary $N \times A$ matrix P is used to encode agent placement at nodes as follows: $P_{ik}=1$ if a_k is hosted by n_i, 0 otherwise. Obviously, a node can host agents only up to its memory capacity. The communication between agents is captured via an $A \times A$ matrix C, where C_{kw} denotes the data units sent on average from agent a_k to a_w per time unit. Let R be a $N \times N \times N$ routing table where an element R_{ijx} denotes the percentage of traffic from n_i to n_j that passes through n_x. Multiple routing and network topology scenarios can be captured using R. The model and consequently the algorithms in this work do not make any particular assumptions on either of them.

2.2 Battery consumption and node lifetime

Let $b(n_i)$ be the battery level of node n_i, measured as the data units a node can send before its battery is depleted. Data transfer consumes the battery of the source and destination nodes where the communicating agents reside, but also the battery of all intermediate nodes that act as routers. Let β denote the ratio between the cost of sending and receiving a data unit. So, for instance if $\beta=0.5$ it means that the receiving cost is 50% of the sending cost. We assume that the cost of routing is equal to the cost for receiving plus the cost for sending data. For simplicity, we ignore the communication cost between co-located agents and the cost of local agent execution.

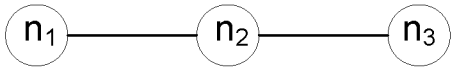

Fig 7.1 An example network

As an example of how battery consumption is captured in our model, consider the topology shown in *Fig 7.1*. Let n_1 send a message of K data units to n_3 every time unit. Assuming that the battery levels of all nodes are B and that β=0.5, n_1 will deplete its battery after B/K time units, n_3 after $2B/K$ *time* units and n_2 after $2B/3K$ time units.

More formally, let L_i denote the lifetime of n_i. This depends on the communication load incurred at n_i, which in turn comprises of three components: (1) the load due to the data sent by agents located on n_i (let X_i), (2) the load due to the data received by agents located on n_i (let Y_i), and (3) the load due to n_i acting as a router (let Z_i):

$$X_i = \sum_{k=1}^{A}\sum_{w=1}^{A} P_{ik}(1-P_{iw})C_{kw} \quad \text{Eq. 7.1}$$

$$Y_i = \sum_{k=1}^{A}\sum_{w=1}^{A} P_{ik}(1-P_{iw})C_{wk} \quad \text{Eq. 7.2}$$

$$Z_i = \sum_{k=1}^{A}\sum_{w=1}^{A} (1-P_{ik})(1-P_{iw})R_{xyi}C_{kw} \mid P_{xk}=1 \wedge P_{yw}=1 \quad \text{Eq. 7.3}$$

$$L_i = \frac{b(n_i)}{X_i + \beta Y_i + (1+\beta)Z_i} \quad \text{Eq. 7.4}$$

2.3 Adding a new agent

The addition of a new agent requires that sufficient memory space be found at some node or be created through agent migrations. For instance, *Fig 7.2* shows two nodes with a memory capacity of 20 units each, which host five agents in total, leaving 2 units of free space at n_1 and 3 units at n_2. Assume that a new agent of size 5 arrives. Clearly, neither n_1 nor n_2 have sufficient free space to host the agent. It is however possible to merge the two free memory fragments into a single bigger chunk, e.g., by swapping a_5 with a_2 and a_3, in order to make space for the new agent to be hosted at n_2 as depicted in *Fig 7.3*.

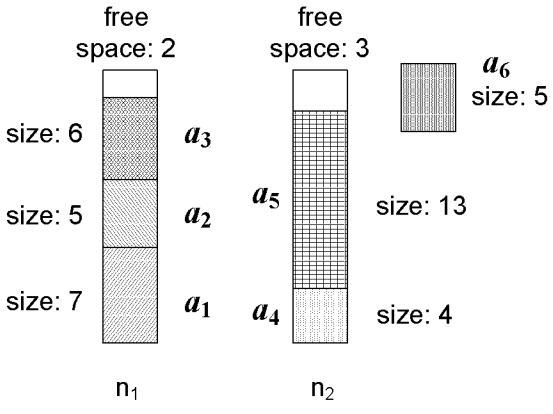

Fig 7.2 Placement (a)

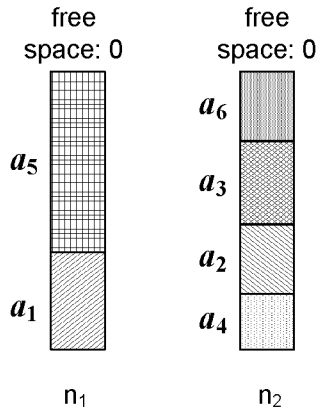

Fig 7.3 Placement (b)

The operations that can be used to alter the agent placement scheme are transfers (migrations) and deletions. Let T_{ijk} denote the transfer of a_k from n_i to n_j and D_{ik} the deletion of a_k at n_i. In order for a transfer T_{ijk} to be feasible, the destination n_j must have enough free space to hold a_k. Note that, given this restriction, it is not possible to implement the transition shown from the placement of *Fig 7.2* into that of *Fig 7.3*, because to perform any agent transfer one must first perform a deletion. We discussed similar feasibility issues in placement transitions in [78]. Tackling them in combination with memory and energy optimization exceeds the scope of this work. Therefore, we assume that the entry point maintains a repository with the code of all agents that have been injected in the system. Thus, the suggested transition could be implemented by deleting a_5 from n_2, transferring a_2 and a_3 from n_1 to n_2, and then transferring a_5 to n_1 and the new agent (a_6) to n_2 from the entry point, corresponding to the sequence $\{D_{25}, T_{122}, T_{123}, T_{e15}, T_{e26}\}$ where n_e is the entry point.

Deletions incur no cost. On the contrary, the cost for performing a transfer is proportional to the agent size, affecting source, destination and the intermediate routers. Specifically, the cost incurred at n_i for a transfer T_{xyk} is given by *Eq. 7.5*.

$$S_i(T_{xyk}) = \begin{cases} m(a_k),\ i = x \\ \beta m(a_k),\ i = y \\ (1+\beta)m(a_k)R_{xyi},\ i \neq x, y \end{cases} \qquad \text{Eq. 7.5}$$

Since an agent migration incurs a communication cost, it also affects the lifetime of nodes in the system. Assuming that at a given point in time the battery level of n_i is equal to $b(n_i)$, and that a series of transfers and deletions are performed to place a new agent, the lifetime of n_i for the new system configuration (including the new agent which introduces additional communication cost due to its interaction with one or more existing agents) is given by *Eq. 7.6*.

$$L_i = \frac{b(n_i) - \sum_{\forall T_{xyk}\ performed} S_i(T_{xyk})}{X_i + \beta Y_i + (1+\beta) Z_i} \quad \text{Eq. 7.6}$$

Notably, our agent migration model ignores for simplicity the cost required for transferring the state of an agent (considered negligible compared to the agent code size). However, it is quite straightforward to take this aspect into account too, by splitting an agent's transfer into two parts: its status only obtainable by the hosting node, and its code obtainable by the hosting node as well as the entry point.

2.4 Problem statement

Let P^{old} be the existing placement of agents on the nodes of the system and P^{new} the placement reached after accepting a new agent (if possible). For modeling purposes we let P^{old} and P^{new} be $(N+1)\times(A+1)$ matrices, where $n_{\text{N+1}}$ is the entry point and $a_{\text{A+1}}$ is the new agent to be placed in the system; whose code is initially available only at node $n_{\text{N+1}}$, i.e., $P^{old}_{N+1,A+1} = 1 \wedge P^{old}_{i,A+1} = 0 \; \forall 1 \le i \le N$. Also, the routing matrix R is extended to include $n_{\text{N+1}}$.

The first target of the agent placement problem (APP) is to define a feasible schedule of agent migrations (transfers and deletions) such that, starting from P^{old}, one reaches a placement P^{new} where $a_{\text{A+1}}$ is placed at some node (besides $n_{\text{N+1}}$), i.e., $\sum_{i=1}^{N} P^{new}_{i,A+1} = 1$. The second target is to maximize the lifetime of the first node that will deplete its battery resources, as per *Eq. 7.6*. Thus, the agent placement problem (APP) can be stated as: Given an initial placement P^{old} of A agents at N nodes and a special entry point node $n_{\text{N+1}}$ that holds the code of all agents as well as the code of a new agent $a_{\text{A+1}}$, define a series of transfers and deletions leading to a new placement P^{new} where $a_{\text{A+1}}$ is placed at some node n_i, $1 \le i \le N$, while maximizing min(L_i).

Notably, APP decision is NP-complete even for the first criterion only, i.e., accepting a new agent with no concern for node lifetimes. We sketch an informal proof by reduction to the Bin Packing-decision (BP-dec) problem which has the following statement: given A objects of size s_i and bins of size K, is there an assignment of objects to bins using V bins?

Proof of NP-completeness: For each BP-dec instance we build an APP-dec statement as follows. The network consists of $V+1$ nodes, the first V of which have capacity K, while $n_{\text{V+1}}$ acts as the entry point. Furthermore, for each object in BP-dec there exists a corresponding agent of same size. In P^{old} the agents exist only at the entry point, while in P^{new} they must be accepted (placed)

at nodes n_1 to n_V. Clearly, a solution for accepting all agents exists if and only if the equivalent BP-dec has a solution with V bins. Therefore, APP-dec is NP-complete.

3 Algorithms for Accepting Agents

Accepting an agent works in two steps. The first step is to check whether some node has sufficient free space to host the agent. If so, the agent is placed at that node. In case multiple candidates exist, the one that results in the longest minimum lifetime as per *Eq. 7.6* is chosen. If no node has sufficient memory to hold the new agent, the second step is to create enough space at some node, by performing a series of transfers and deletions as discussed in Sec 2.3. The respective heuristics employ a component for solving the knapsack problem through dynamic programming [56].

3.1 Pairwise checking algorithm (PCA)

The node with the largest free memory is more likely to provide the space needed for hosting a new agent, by moving one or more of its local agents to another node. Conversely, if some agents must be moved away from a node, it is easier to do so if the destination has relatively ample free space. This is the intuition behind the pairwise-checking algorithm (PCA), the pseudocode of which is shown in *Fig 7.4*.

Algorithm PCA

```
L:=sort nodes in decreasing order of available memory
while (L has at least two nodes)
   n1:=L→head; //most capacious node
   n2:=n1→next; //second most
   while (n2≠NIL && availMem(n1)<requiredSpace)
      openSpace(n1, n2);
      reinsert(L, n1); reinsert(L, n2);
      if (n1=L→head && n2=n1→next) n2:=n2→next;
      else if (n1=L→head) n2:=n1→next; //n2 changed
      else break; //n1, n2 changed, restart process
      endif
   endwhile
   if (availMem(n1)≥ requiredSpace) return; //success
   endif
   if (n2=NIL) delete(L, n1); //list traversed
   endif
endwhile
```

openSpace(node: n1, node: n2)

```
maxSpace:=maxFreeSpace(n1, n2);
bestsol:=current placement;
A:=set of agents located at both n1 and n2;
sol1:=knapsack(n1, A) and remaining agents at n2;
sol2:=knapsack(n2, A) and remaining agents at n1;
if (maxFreeSpace(sol1, n1, n2) > maxSpace)
   bestsol:=sol1;
   maxSpace := maxFreeSpace(sol1, n1, n2) ;
endif
if (maxFreeSpace(sol2, n1, n2) > maxSpace)
   bestsol:=sol2;
   maxSpace := maxFreeSpace(sol2, n1, n2) ;
endif
implement bestsol ;
```

Fig 7.4 Pseudocode for PCA

Specifically, PCA maintains a list of nodes sorted in decreasing order of their remaining free space. It takes the first node (with the most free space) and attempts to open even more space by considering the second node of the sorted list as a partner for exchanging agents. If enough space is opened at any of the two nodes, the algorithm terminates successfully. Else, the first node is checked against the third node etc., until the last node in the list is checked. Then, the first node is removed from the list and the process is repeated (starting with the new first node), until either enough free space is opened at some node or the list is empty. After each attempt to open space the list is updated with the new free space values (and placements) of the participating nodes. If during the process either the first or second node changes position in the list, the iteration restarts with the new first and second nodes.

Agent rearrangement at each considered node pair (openSpace function) is done with the goal to maximize the free space at one of the nodes. This is achieved by solving two different instances of the knapsack problem, with the storage capacity of the first and respectively second node as the knapsack size; the set of agents to be placed in the knapsack being the union of agents hosted at both nodes, and the benefit of each agent being equal to its size. The two solutions are compared to each other and with the initial placement, and the one with the largest free space at a node is chosen.

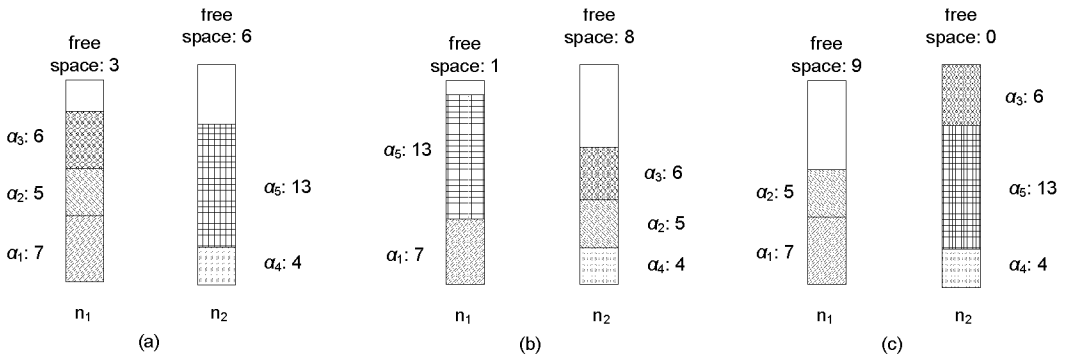

Fig 7.5 Example of knapsack runs: (a) initial state; (b) run on n_1; (c) run on n_2

As an example, consider *Fig 7.5a* which continues the example of *Fig 7.2* but with the capacity of n_1 and n_2 being 21 and 23 data units, respectively, leaving 3 units of free space at n_1 and 6 at n_2. Assume 9 units of free space are needed to place a new agent. The knapsack run on n_1 (*Fig 7.5b*) produces a placement whereby agents a_1, a_5 are located at n_1 while a_2, a_3, a_4 are located at n_2, resulting in a contiguous free space of 8 units at n_2. For the run on n_2 (*Fig 7.5c*), agents a_3, a_4, a_5 are placed at n_2 while a_1, a_2 are placed at n_1, leaving a free space of 9 units at n_1. Thus, the placement resulting from the second run is chosen.

3.2 Greedy bin packing algorithm (GBPA)

The second algorithm follows a bin packing approach. Starting with all nodes initially empty, GBPA iteratively attempts to place all agents, including the newcomer. In the first iteration, knapsack is run *N* times, once for each node, and the solution that leaves the least free space on a node, i.e., fills a node as much as possible, is chosen. The agents selected by that knapsack run are placed on that node, and the process is repeated for the rest of the agents and nodes. The algorithm continues until either all agents or all nodes have been considered. In the first case the generated placement can be used to accommodate the new agent whereas in the second case a solution could not be found. *Fig 7.6* illustrates the pseudocode of the algorithm.

```
Algorithm GBPA

N:=all nodes;
A:=all agents including the newcomer;
bestspace:=INFTY; bestnode:=NIL; bestagents:=NIL;
while (A and N not empty)
   for all nodes ni at N
      knapsack(ni, A);
      if (free space at ni<bestspace)
         bestspace:=free space at ni;
         bestnode:=ni;
         bestagents:=agents assigned to ni by knapsack;
      endif
   endfor
   remove bestnode from N;
   remove bestagents from A;
endwhile
if (A=NIL) implement the assignments produced;
endif
```

Fig 7.6 Pseudocode for GBPA

One can expect that GBPA will alter the initial placement scheme more drastically than PCA, because all agents are placed on the nodes essentially from scratch. PCA changes the placement of node pairs and starts doing so using the most promising ones (the ones with the largest free space), hence the initial placement scheme could be left relatively unmodified. However, given its packing-oriented nature, GBPA is also more likely to reach a solution compared to PCA. For comparison reasons we also experiment with two well known bin packing algorithms, FirstFit (FF) and BestFit (BF).

4 Optimizing Node Lifetime

Once the goal of placing a new agent is accomplished, one may adjust the placement in order to maximize the lifetime of the node that will first deplete its battery. The key component of the above optimization is an agent swapping process among node pairs that attempts to move agents that communicate heavily "closer" to each other; ideally on the same node. Following we give details of the process and introduce two algorithms that optimize lifetime based on agent swapping.

4.1 Agent swaps

Given a pair of nodes and the agents assigned to them, the problem of redefining the placement so that the minimum node lifetime is increased is tackled as follows. For each agent the benefit (in node lifetime terms) of migrating it to the other node of the pair is calculated. The agent with the highest benefit attempts to migrate first. If the destination node has sufficient free space, the migration succeeds. Else, the process attempts to define a group of agents at the destination, such that if the group is swapped with the agent, enough free space opens. If such a group exists and the overall placement remains beneficial, the exchange is performed. The process is repeated for the next most beneficial agent and so on. After a migration attempt is successfully accomplished, the benefits are updated. The process terminates, when all agents are considered. *Fig 7.7* shows its pseudocode.

```
swapAgents(node: n1, node: n2)
oldlife:=calculate min lifetime //as per Eq. 7.6
for all agents a_k in n1 and n2
   life[k]:=min lifetime if a_k changed node;
   benefit[k]:=life[k] - oldlife;
endfor
while (exists a_k: benefit[k]>0)
   candidate:=max benefit agent;
   if (free capacity at opposite node>size of candidate)
      place candidate at opposite node;
   else
      g:=group of agents from opposite node such that enough free space is opened;
      newlife:=min lifetime if candidate and agents in g were swapped;
      if (newlife>oldlife)
         swap candidate and agents in g;
      endif
   endif
   oldlife:=newlife;
   recalculate life[], benefit[];
endwhile
```

Fig 7.7 Pseudocode for swapping agents in a node pair

4.2 Reconfiguration algorithms

All the reconfiguration algorithms (RAs) we consider, work in a greedy fashion by iteratively applying the swapAgents method (Sec. 4.1).

ggRA. The first algorithm called greedy global reconfiguration algorithm (ggRA) considers at each iteration, all node pairs ($O(N^2)$) and for each of them computes agent swapping as per Sec. 4.1. The pair for which the application of swapAgents yielded the maximum benefit with respect to minimum node lifetime is selected and the induced agent transfers are performed. The algorithm then continues by checking again the agent swapping at all node pairs, selecting the best candidate and so on so for, until at some iteration the application of swapAgents results in zero or negative benefit at all node pairs. At this point the algorithm stops and the final placement is produced. *Fig 1.2* shows the pseudocode of the algorithm.

Algorithm ggRA

```
oldlife:=calculate min lifetime //as per Eq. 7.6
found:=true;
while (found)
   found:=false;
   bestlife:=0;
   for all node pairs (ni, nj)
      sol:=swapAgents(ni, nj);
      newlife:=calculate min lifetime of sol;
      if (newlife>bestlife)
         bestsol:=sol; bestlife:=newlife;
      endif
   endfor
   if (bestlife>oldlife)
      implement bestsol;
      oldlife:=bestlife;
      found:=true;
   endif
endwhile
```

Fig 7.8 Pseudocode for ggRA

glRA. The second algorithm we consider called greedy local reconfiguration algorithm (glRA) works in a similar manner to ggRA. Again at each iteration it computes swapAgents for node pairs. However, contrary to ggRA which must check all node pairs before deciding the best one, glRA selects the first pair that incurs a positive benefit in swapAgents, perform the required transfers and reiterates.

Comparing the two reconfiguration algorithms we expect that glRA will be considerably faster compared to ggRA, without however, achieving the same solution quality.

5 Accepting Agents and Optimizing Lifetime Simultaneously

The algorithms presented so far can be used to tackle APP in a two step fashion: first the new agent is placed at some node, using the algorithms of Sec. 3 to create enough space if necessary; then some RA to optimize the resulting placement in terms of node lifetime. The algorithms presented here are based on the branch and bound paradigm and combine both steps at the same time.

sBBA. The simple branch and bound algorithm (sBBA) works as follows. Beginning with the initial placement (excluding the new agent), a solution tree is built. At the first level, all node pairs are considered, and sBBA runs for each node pair the openSpace process (Sec. 3). Then, it selects the best candidates, which are expanded to produce the next level of the tree, by adding one of the remaining nodes. *Fig 7.9* depicts the structure of such a tree. Whenever a partial solution (tree node) with i nodes is expanded to produce a partial solution with i+1 nodes (e.g., from a pair to a triplet) the agent placement is updated by running openSpace for opening space among the node that was added to produce the expansion and the node with the largest free space in the previous solution.

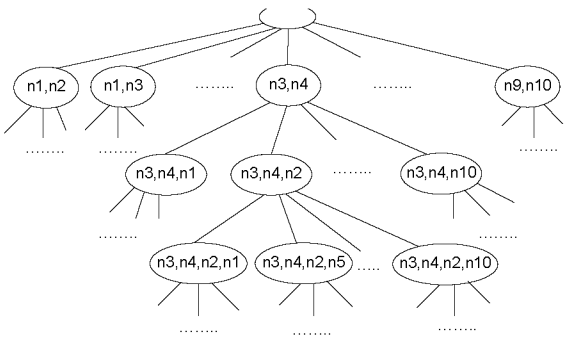

Fig 7.9 Solution tree with 10 nodes

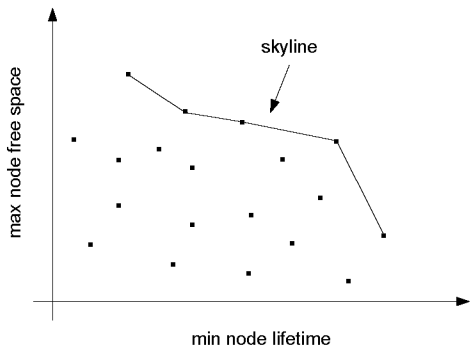

Fig 7.10 Skyline example

sBBA decides which partial solutions (tree nodes) to expand by evaluating them across two metrics: the maximum free space at a node belonging to the partial solution and the minimum node lifetime in all nodes of the network. At each tree level, only partial solutions at the skyline (no other solution is better in both dimensions) of the above two dimensional space are considered for expansion (see *Fig 7.10*). *Fig 7.11* illustrates the algorithm in pseudocode.

Algorithm sBBA

```
P:=all partial solutions consisting of single nodes;
c:=1://minimum cardinality of a partial solution at P
while (requiredSpace not opened)
   while (P contains partial solutions of cardinality c)
      pi:=a partial solution of cardinality c;
      while (not all nodes considered for expansion)
         expand pi with node nj;
         for all nodes nx∈ pi
               openSpace(nx, nj) //as per Fig 7.4
         endfor
         if (requiredSpace opened)
               finalsol:=expanded pi + all nodes ∉ pi
               return;
         endif
      endwhile
      subtract pi from P
      add pi's expansions to P
   endwhile
   prune from P partial solutions not belonging to the skyline
   c:=c+1;
endwhile
```

Fig 7.11 Pseudocode for sBBA

iBBA. The improved branch and bound algorithm (iBBA) follows the same general procedure with sBBA, nevertheless, it differs in two major ways. The first one concerns the way a final solution (involving all nodes) is defined, once in a partial solution (tree node) the required free space is opened. sBBA stops at this point and leaves the placement on the nodes not belonging in the final solution untouched. So, for instance in *Fig 7.9* if the partial solution $<n_3, n_4, n_2, n_5>$ opens the required space, the final solution of sBBA will consist of the placement described at the partial solution for the nodes $<n_3, n_4, n_2, n_5>$ and the initial placement at the remaining nodes $<n_1, n_6,.., n_{10}>$. This might be inefficient liefetime-wise, since in the remaining nodes optimization possibilities might exist. iBBA takes advantage of such optimization potential by defining the final solution as follows. It adds to the partial solution e.g., $<n_3, n_4, n_2, n_5>$ one by one all remaining nodes in a random order (in the example 6 in total). At each such addition swapAgents (*Fig 7.7*) is run between the agent that is added and the existing agents at the partial solution.

iBBA also differs in another way compared to sBBA. Namely, while sBBA stops if a partial solution involves the desired free space, iBBA continues exploring further possibilities. To do so, the partial solution that opened the desired space, as well as all its successors do not take part in the skyline criterion. To bound the running time, iBBA stops after k such alternative solutions are defined and implements the best among them.

6 Implementing a New Placement

A subtle issue concerns how the computed placements are actually implemented. Recall that all algorithms start from an initial placement P^{old} and try to define a new one P^{new} that includes the new agent. It is possible to trace the execution steps of the algorithms to perform the corresponding agent migrations, albeit at a high implementation cost. This is especially true if the algorithms run as a "pipeline", e.g., ggRA on top of GBPA, since the placement produced by the algorithm that runs first, will be altered afterwards. Instead, we tackle the implementation of P^{new} as a separate problem, which can be stated as: given P^{old} and the P^{new} derived by the algorithm(s) of Sec. 3-5, perform a series of agent transfers and deletions so that P^{new} is reached with the lowest possible cost.

In [78] we explored various algorithms for a similar problem where multiple copies must be created for a given object. Here we adopt the following variation. Starting from the set of all required agent migrations (agent a_{k} must move to n_{i} if $P_{ik}^{old} = 0$ and $P_{ik}^{new} = 1$), a migration is picked randomly and performed by transferring the agent code from a suitable source. Two sources may exist for fetching the code of an agent: the node that hosted the agent in P^{old} (provided the agent has not been deleted), and the entry point which keeps a copy of all agents. If both options apply, the algorithm selects the source corresponding to the transfer path that contains the node with the longest minimum lifetime. In case the destination does not have enough free space, the algorithm randomly deletes one or more agents that must not be hosted at that node according to P^{new}. Finally, having performed all the required transfers, to reach P^{new}, the algorithm deletes any superfluous copies of agents (at their old hosts).

7 Experiments

The presented algorithms were evaluated through simulations for a network of 31 nodes (one being the entry point). A total of 5 different networks were generated as follows. The nodes were randomly placed in a 100x100 2D plane and assumed to be in range of each other if their Euclidean distance was less than 30. Based on the resulting connectivity graph, the minimum (hop-wise) spanning tree was defined as the routing topology. Nodes were assumed to have a battery lifetime enough to transfer/receive (both costs assumed equal) 1GB of data (roughly the case of an Imote2 platform supported with 3 AAA alkaline batteries [1]) and 256KB of

memory. The size of agents varied uniformly between 10KB and 150KB (unless stated otherwise). Each agent communicated with 5 other randomly selected agents, generating a load uniformly distributed between 10 and 100 bytes per time unit per agent.

In the following experiments we discuss the performance of PCA, GBPA, sBBA, iBBA, BF and FF, both as standalone algorithms and in conjunction with ggRA and glRA (denoted as PCA+ggRA etc.). Unless otherwise stated, the maximum number of final solutions explored by iBBA was set to 5. Each experiment was repeated 4 times per generated network (total 20 times), each with a different agent setup and results were averaged. As a reference, we also include results obtained for a naïve algorithm (RAND) which randomly places a new agent as long as there is a node with enough space to host it.

7.1 Performance on acceptance criterion

Starting from an empty system, we investigate the scenario where one new agent arrives every 100 time units, for 500 agents. The algorithms do not stop when the first agent is rejected, but continue until all agents have been considered (in their arrival sequence).

Table 7.1 shows the sequence number (average of 20 runs) of the first agent that was rejected by each algorithm. It shows that RAND, BF and FF start dropping agents earlier on, with a value between 92 and 93, while BBAs, GPBA and PCA are able to place roughly 4 more agents before rejecting the first one. Among them, GBPA has the best performance with the relevant differences being small. This experiment was also performed with all the algorithms' combinations with ggRA and glRA. Results showed that the application of RAs had a negligible (mostly positive) effect to the acceptance metrics of all algorithms but RAND, whereby it results in performance deterioration. This is because RAND never changes the placement of agents, hence cannot "repair" possible fragmentation of free space caused by RA in its attempt to optimize node lifetime.

Table 7.1 also shows the number of agents that were rejected, while the total free fragmented space was greater than their size (tentative wrong rejections). Also, the ratio of the respective agent sizes to the total free memory at the point of rejection is shown, as a measure of difficulty for the placement that failed. It can be seen that GBPA is almost optimal with only 1 agent being a tentative wrong rejection for the total of the 20 runs (0.05 average) while the total available space was barely enough to host it (0.97 ratio).

Table 7.1 Acceptance metrics

	first rejection	tentative wrong rejections	agent size / available memory
RAND	92.5	298.9	0.53
RAND+ggRA	88.05	318.75	0.48
RAND+glRA	88.4	300.35	0.49
PCA	96.3	4.15	0.89
GBPA	96.6	0.05	0.97
sBBA	96.05	30.7	0.78
iBBA	96.3	21.25	0.81
BF	92.65	294.3	0.53
FF	92.7	300.1	0.52

To further evaluate the algorithms concerning their acceptance capability we performed a "domination" test. An algorithm A is said to dominate another algorithm B if any sequence of agent arrivals that is accepted by B, is accepted by A as well. In order to test algorithm domination, we recorded in the previous experiment all the agents accepted by each algorithm. Recall that the simulation didn't terminate upon an agent's rejection but continued until all 500 agents were considered. Therefore, different algorithms accepted (most likely) different agents in each of the 20 runs conducted. We used the agents accepted by an algorithm as input to the others and recorded whether the sequence was accepted or not.

Table 7.2 gives the percentage of the sequences that were accepted by another algorithm. Table columns depict which algorithm's accepted agents were used as an input sequence to the algorithm mentioned in the relevant row. Each value represents the result of all 20 such sequences. So, for instance PCA accepted only 15% (0.15 value in the relevant cell) of the 20 sequences involving the agents accepted by GBPA, while all algorithms obviously have a domination percentage of 1 against themselves. RAND was excluded from the experiment since it was dominated by all others.

Table 7.2 Domination percentage

	PCA	GBPA	sBBA	iBBA	BF	FF
PCA	1	0.15	0.7	0.6	1	1
GBPA	0.95	1	0.95	0.95	0.95	1
sBBA	0.35	0.05	1	0.5	1	1
iBBA	0.2	0.05	0.55	1	1	1
BF	0	0	0	0	1	0.25
FF	0	0	0	0	0.75	1

The first thing to notice is that no algorithm dominates absolutely all others. The second thing is that GBPA offers the highest domination ratio accepting 19 out of 20 sequences corresponding to PCA and BBAs (0.95 value in the table). A peculiar result is that while BF and FF are totally dominated (value of 1) by PCA and BBAs, GBPA fails to accept one of the BF sequences (0.95 value). By delving into the experimental data, we found out that there is only one agent GBPA rejected, which agent is accepted by PCA, BBAs and BF. Nevertheless, the domination rate of GBPA is still the highest. Furthermore, what is more important is to observe the domination of the other algorithms versus GBPA. PCA accepts only 15% of GBPA's sequences, while BBAs accept only 5%. This reinforces our intuition in Sec. 3.2 that GBPA is the most powerful algorithm in opening space to accommodate new agents. *Table 7.2* also shows PCA coming second followed by BBAs, while BF and FF being particularly bad, unable to accept any of the remaining algorithms' sequences.

Table 7.3 Average algorithm behavior in the domination test

	domination percentage	**rejected agents**	**size of rejected agents**
PCA	0.69	0.31	10.15
GBPA	0.96	0.04	5.96
sBBA	0.58	0.42	17.31
iBBA	0.56	0.46	19.97
BF	0.05	2.53	233.7
FF	0.15	2.22	224.43

Table 7.3 records the average domination percentage of an algorithm against the sequences of all others (5*20=100 total), together with the average number of rejected agents per sequence and their size. One thing that deserves explanation is the fact that iBBA has a slightly smaller average domination behaviour compared to sBBA. This is an acceptable tradeoff, since iBBA results in placements more optimized towards energy efficiency against sBBA. Overall, *Table 7.3* confirms the previous remarks concerning the relevant algorithm performance on accepting agents, i.e., GBPA is first, followed by PCA, followed by BBAs, while BF and FF are particularly bad with the latter being better than the first. Henceforth, RAND, BF and FF will be mostly omitted from the experiments.

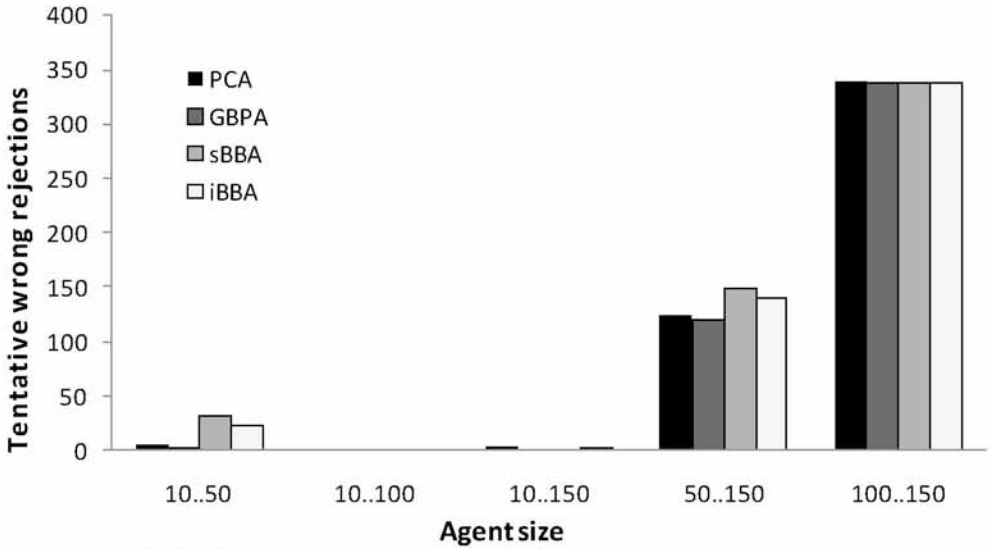

Fig 7.12 Number of tentative wrong rejections for various agent sizes

As a last test for the ability of the algorithms to accept a newcomer agent, *Fig 7.12* shows the number of tentative wrong agent rejections for 5 different agent size (uniform) distributions. In the cases where agent size could take both small and large values (10..100 and 10..150 distributions), all algorithms had almost zero tentative wrong rejections. This is a particularly encouraging result indicating that the algorithms achieved the optimal performance. In the 10..50 case the best performance was by GBPA followed by PCA. For the 50..150 and 100..150 distributions, where the maximum agent size is greater than half node capacity, a significant number of tentative wrong rejections appear, their number increasing with the average agent size. This behaviour is expected because the problem of creating enough space to fit an average sized agent becomes harder. GBPA either outperforms or is equal to the rest, which further confirms its merits in accepting agents. Notice, that the high rejection rate observed is a bit misleading. In the 100..150 case all algorithms left a total (at all nodes) free space of merely 179.95 i.e., enough to place one additional agent with the largest size, while in the 50..150 case the total free space left varied from between 64 (GBPA) and 88.7 (sBBA), i.e., enough to store one agent of the smallest size.

7.2 Performance on energy criterion

In order to evaluate the algorithms in terms of maximizing the lifetime of the first node that depletes its battery, we stop our simulation when the first agent is rejected by some algorithm (on average at the 96th agent). At that point, all placements are guaranteed to contain the same

(communicating) agents, and thus can be fairly compared as to the energy consumption criterion.

Fig 7.13 shows the minimum node lifetime for GBPA, PCA, sBBA, iBBA and their glRA variants. When executed as standalone, iBBA achieves the best results with a performance difference of more than 20% compared to the second best which is sBBA. Standalone PCA outperforms GBPA by roughly 12.7%, however, both algorithms result in marginally inferior solutions agenst iBBA and sBBA (by more than 141%). These results confirm the premise of BBAs, i.e., that they can tackle both acceptance and energy optimization criteria at the same time. Next, observe that the application of glRA considerably improves the performance of all algorithms by between 30.8% in iBBA and 227% in PCA. The best combination is sBBA+glRA, with iBBA+glRA coming second, PCA+glRA third and GBPA+glRA last. An interesting thing to notice is that standalone iBBA outperforms GBPA+glRA by 30.1% and loses to PCA+glRA by 13%. As it will become apparent in Sec. 7.3, the application of glRA affects significantly the running time of the algorithms. Therefore, when a compromise between running time and energy efficiency is needed, standalone iBBA is a valid choice. Finally, we would like to mention that the apparently low performance of GBPA, even after the application of glRA is rather expected since GBPA redefines the total placement from scratch each time it accepts an agent, therefore it makes it harder for glRA to optimize the placement and also requires more (costly) migrations to do so.

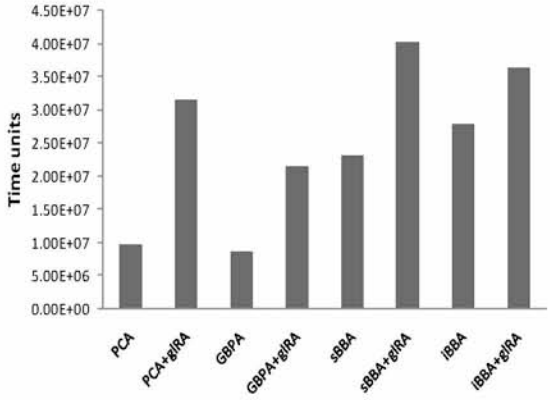

Fig 7.13 Minimum node lifespan

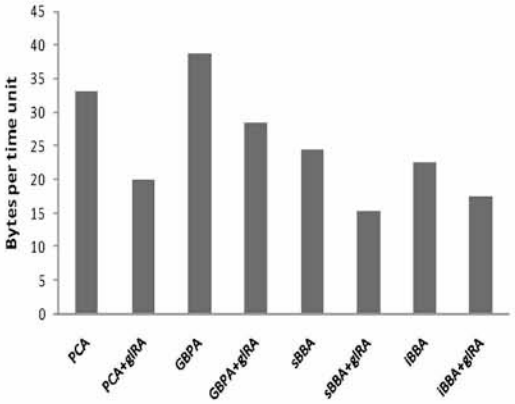

Fig 7.14 Average node battery consumption per time unit

To further characterize the algorithms in terms of energy efficiency, *Fig 7.14* shows the average battery consumption at each node per time unit, measured from the time when the first agent is accepted up to the time where the first node runs out of battery. Again, iBBA achieved the best performance among standalone algorithms, while sBBA+glRA was the best combination with

iBBA+glRA following closely. This means that BBAs and their combinations are not only superior in maximizing the lifetime of the first dying node, but also in minimizing energy consumption across the whole network (iBBA had 32.3% less consumption against PCA and 42% against GBPA).

7.3 Other experiment and metrics

Thus far we presented results with glRA as the reconfiguration algorithm. *Fig 7.15* shows the relevant performance differences between ggRA and glRA when applied over PCA, GBPA, sBBA and iBBA. Concerning the main energy related metrics, i.e., min node lifetime and average battery consumption at all nodes, ggRA gives mixed results. For instance, BBAs+ggRA is better at improving the lifetime of the first node that dies compared to BBAs+glRA (by less than 10%), while when applied over PCA and GBPA the results are the opposite, i.e., the glRA combination is superior (negative values in the plot). However, glRA is faster than ggRA, regardless of the algorithm applied to, and is therefore a more viable option.

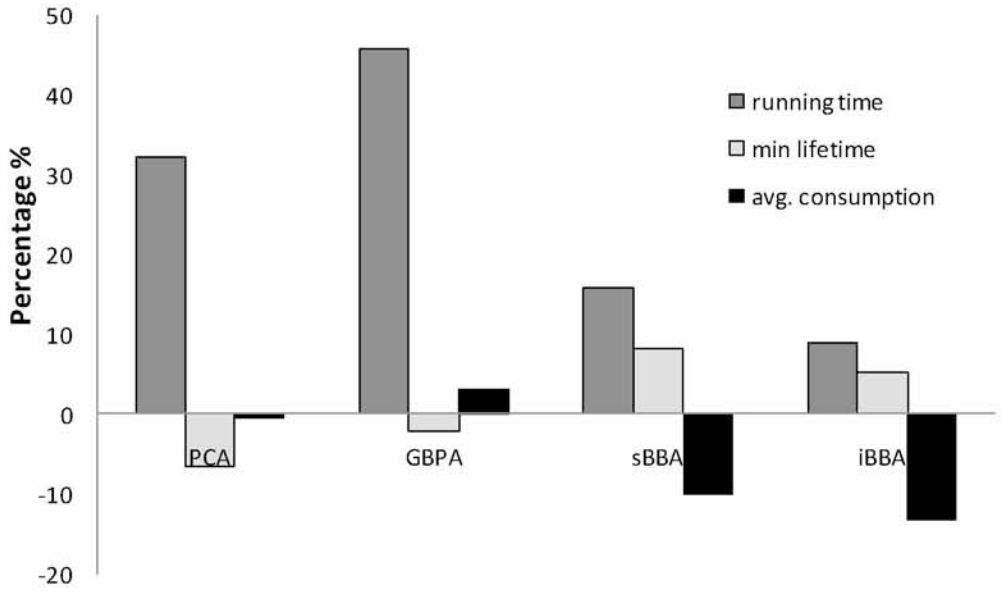

Fig 7.15 Comparison between ggRA and glRA

Next, we evaluate iBBA's performance with regards to the number of the final solutions (that is, the *k* variable we are referred to in last paragraph of Sec. 5) the algorithm is allowed to explore before terminating. *Fig 7.16* plots the achievable node lifespan, while *Fig 7.17* plots the average running time for accepting/rejecting a single agent of iBBA. Concerning the later we can notice that it increases linearly to the number of final solutions the algorithm outputs, while

performance on the node lifetime criterion (*Fig 7.16*) exhibits a knee. The results mean that after a certain number of final solutions are achieved, the relevant performance gains by continuing the exploration of the solution space are small and might be offset by the corresponding increase in the running time of the algorithm. In all our experiments we used the value of the knee, i.e., 5 final solutions as a stopping criterion of iBBA.

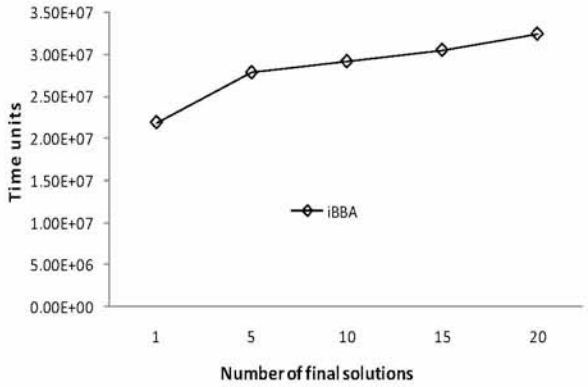

Fig 7.16 Minimum node lifespan achieved by different iBBA versions

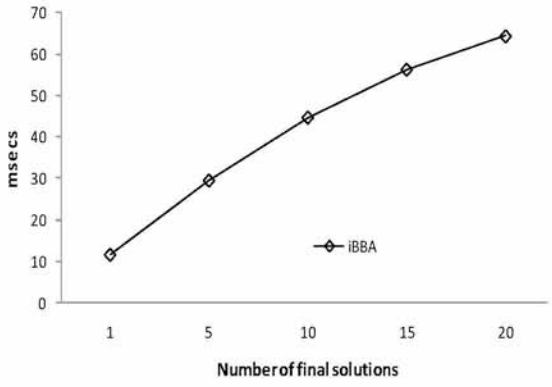

Fig 7.17 Running time of different iBBA versions

Last, we discuss two more performance parameters. The first parameter is the communication cost incurred by the algorithms due to agent migrations, which is essentially the overhead for achieving the resulting placement. *Fig 7.18* shows the number of migrations and *Fig 7.19* the percentage of migration cost in the total communication load (including agent-level traffic) for three different battery levels: 1, 0.5 and 0.1 GB. In all cases the placement overhead (*Fig 7.19*) increases as the battery level decreases, because nodes (and agents) die sooner and as a consequence the system cannot amortize the agent migration cost paid. glRA variants incur significantly higher overhead compared to standalone algorithms due to the increased number of migrations performed; more than an order of magnitude compared to standalone algorithms as shown in *Fig 7.18*. Among the standalone algorithms GBPA is the most expensive migration wise. This confirms the assumption that bin packing alters considerably the existing placement, making it much harder for RAs to optimize it.

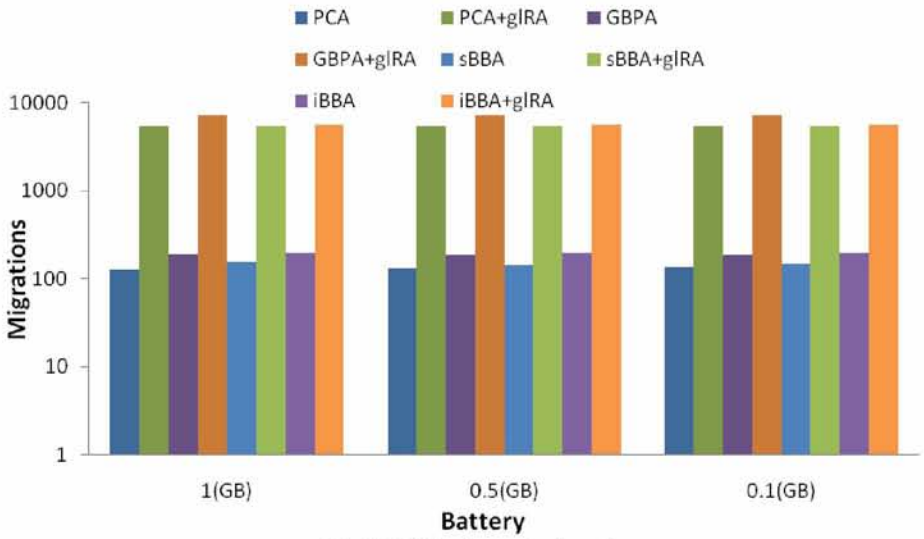

Fig 7.18 Number of migrations

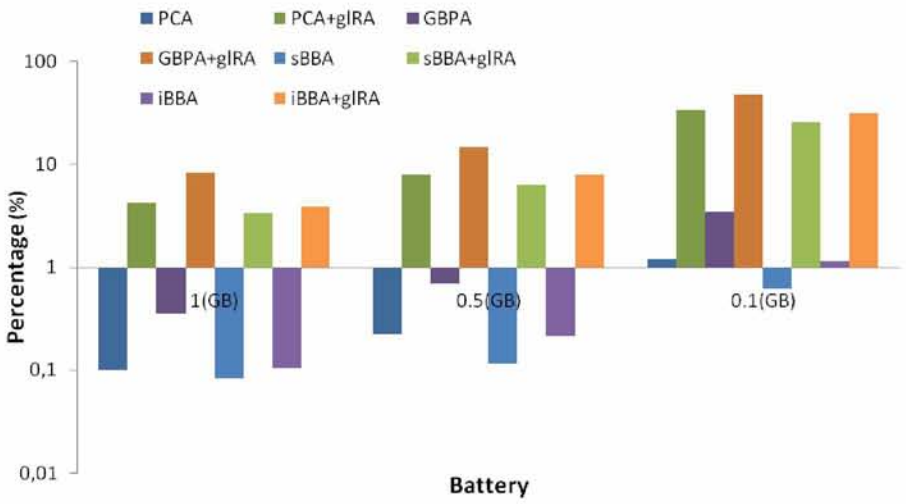

Fig 7.19 Placement overhead as a percentage of total network load

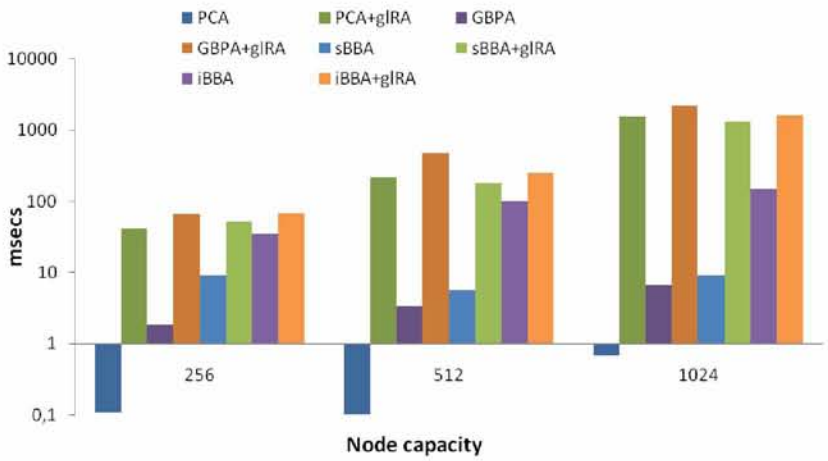

Fig 7.20 Average running time (msecs) for accepting/rejecting a single agent

Finally, we measure the running time of the algorithms. *Fig 7.20* presents the average time it took for the algorithms to accept (reject) one agent, measured for three distinct node sizes: 256KB, 512KB, 1024KB. Note, that the running time of most algorithms increases for larger node sizes (more agents accepted). Among the standalone variants, PCA is the fastest, with GBPA second and sBBA, iBBA following in that order. It is interesting however to notice that iBBA is faster compared to all algorithms that achieve comparable performance on the lifetime metric with it, i.e., glRA variants (*Fig 7.13*). Overall glRA increases the running time of all standalone versions by between 1 and 3 orders of magnitude. Nevertheless, the actual values even for the slowest combination (about 3 secs for GBPA+glRA) are still small enough for a real-world system.

7.4 Discussion

Summarizing we can state the following: (i) the classic bin packing solutions BF and FF, as well as the random algorithm have noticeably inferior performance compared to GBPA, PCA and BBAs, accepting fewer agents; (ii) GBPA is better in accepting agents than PCA and BBAs but has higher running time and is less able to save energy; (iii) BBAs are the most energy efficient algorithms, achieve comparable (but smaller) to GBPA and PCA performance on the agent acceptance criterion, but have higher running times compared to them; (iv) among BBAs, iBBA is slower compared to sBBA, but achieves considerably better performance on the lifetime criterion; (v) PCA is a tradeoff between GBPA and BBAs concerning acceptance and energy management, while being considerably faster compared to them; (vi) RAs improve the energy efficiency of all algorithms without affecting the acceptance criterion much, at the expense of a higher running time; (vii) among the RAs, glRA offers the better trade-off between running time and solution quality.

Thus, whenever the acceptance criterion is the absolute determining factor GBPA (and possibly GBPA+glRA or GBPA+ggRA) is the algorithm to choose, whereas if energy efficiency is equally important iBBA (and possibly sBBA+glRA) offer viable alternatives. Finally, PCA (and possibly PCA+glRA) is a good choice whenever a decent trade-off between acceptance, energy optimization and computation time is required.

8 Conclusions

In this chapter we introduced the agent placement problem (APP) which has two different components: (i) finding/creating enough space for hosting an agent and (ii) optimizing energy consumption due to agent communication and migration. Heuristics were proposed for tackling the two performance aspects both independently (GBPA, PCA, RAs) and simultaneously (BBAs). Through simulated experiments, different tradeoffs were identified (BBAs offered a particularly promising one), while all algorithms outperformed two well known bin packing heuristics (best and first fit) as well as random placement. In previous works the objective function was the reduction of the energy spent over the network, while this chapter does not take this optimization into account at all.

Part of this work has been published in the following workshop and journal proceedings:

* N. Tziritas, T. Loukopoulos, S. Lalis and P. Lampsas, "Agent Placement in Wireless Embedded Systems: Memory Space and Energy Optimizations," in Proc. 9th Int. Workshop on Performance Modeling, Evaluation, and Optimization ofUbiquitous Computing and Networked Systems (*PMEO2010*), *IPDPS workshops*.

* N. Tziritas, T. Loukopoulos, S. Lalis and P. Lampsas, "Algorithms for energy-driven agent placement in wireless embedded systems with memory constraints," *Simulation Modelling Practice and Theory (Elsevier)*, 2011

Chapter 8

Agent Manager System Implementation and Evaluation

1 Introduction

POBICOS [91] is a platform that focuses on applications running on top of a networking system consisting of cooperating objects in the field of wireless embedded systems. An application consists of a number of mobile code entities (called agents) structured in a tree-like manner. The main targets of POBICOS is to provide: i) a user-friendly environment to install/un-install/monitor applications without needing the presence of an expert; ii) an opportunistic programming model enabling the application programmer to write an application of its own preference without knowing in advance which objects will host the application in question, and also the connectivity graph of that objects.

The core of this project is the middleware lying between the application(s) and the operating system (TinyOS). Specifically the most significant components of the middleware are shown in: i) the *runtime* which is responsible for executing the code of an agent; ii) the *agent manager* whose functionality is to enable the interaction between agents either they are co-located or not; iii) the *code transport* which is invoked by agent manager to download agent binaries; iv) the *network abstraction* which is responsible for the communication between objects. In the sequel we give a coarse-grained description about the basic functionalities of network and runtime component which are central to the agent manager functionality.

The networking layer provides two different messaging services: reliable and best-effort. In the case of former, a datagram is (re-)transmitted from the source node to the destination one, till either an acknowledgment travels back from the destination node to the source one to confirm that the datagram has been successfully delivered; or the maximum number of retransmissions has been reached, where the delivery is declared unsuccessful. Datagram ordering and filtering of duplicates is handled by that service. As far as the best-effort service is concerned, a datagram is sent towards the destination node without retransmission attempts, and therefore without guarantees that the datagram will be ever delivered. This service provides neither ordering nor filtering of duplicates.

The agent manager interacts with the runtime component through commands/events in order for the former to: i) issue a request (via a command) about the allocation and removal of an agent instance; ii) request the suspend/resume of the execution flow of an agent instance when needed (e.g. performing an agent migration); inquire about locally available (generic and non-generic) resources and the local node descriptor.

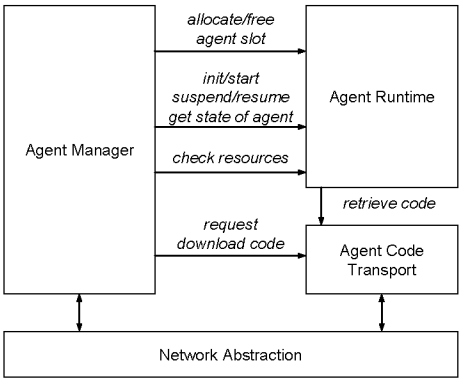

Fig 8.1 Key middleware components and interactions for supporting agent mobility.

2 System Implementation

The POBICOS middleware is developed for TinyOS v2.1 running on Crossbow iMote2 nodes at 104MHz. Thanks to a component that provides transparent access to external memories (e.g., Flash), the core RAM requirements can be kept below 8KB, which makes it possible to port the

middleware to more resource-constrained devices. Wireless communication is via an external ZigBee modem from the Z430-RF2480 demo kit of Texas Instruments [110].

We should notice that AGE has been developed into the agent manager, besides the protocols elucidated further down.

2.1 Data types and data structures

This section gives an overview of the data types and data structures of the POBICOS middleware that are relevant for the purpose of agent management. Data structures are specified in a high-level fashion, without focusing on any implementation details.

2.1.1 Agent identifiers

The identifiers of agents are 4-byte unsigned integers. The most significant 2 bytes are set equal to the address of the node where the agent is created. The least significant 2 bytes are assigned the value of an agent seed number, which is incremented each time a new agent is created. This number is stored in persistent memory to guarantee uniqueness of agent identifiers despite node reboots.

2.1.2 Agent descriptors

For each locally hosted agent, a descriptor is used to keep all relevant information, such as the agent's identifier, the node address and identifier of its parent, as well as the node addresses, identifiers and group identifiers of its children. Agent descriptors are stored in volatile memory. When a node reboots, this information (along with all runtime information associated with agents) is lost. (Note: POBICOS agents are not persistent.)

2.1.3 Creation request descriptors

For each agent creation request issued by a locally hosted agent, a descriptor is used to keep all relevant information, such as the identifier of the agent that issues the request, the parameters of the request, the remaining lifetime of the request, and the current state of the request. Creation request descriptors are stored in volatile memory. When a node reboots, this information is lost.

2.1.4 Message queues, sequence numbers, epoch numbers

For each node, a message queue is maintained where agent-level (and other special) messages are placed for transmission in FIFO order over the network. Each queue is associated with a local sequence number that is increased for each message sent via the queue, and with a remote

sequence number that is updated each time a message from that node is received. Message queues and their sequence numbers are stored in volatile memory, hence do not survive reboots.

A message queue is initialized when the node reboots or the respective remote node becomes "unreachable" (according to the network abstraction layer), in which case both the local and remote sequence numbers are reset to 0. To let remote nodes infer such resets, each message queue is also associated with a local and remote epoch number. The local epoch is attached (together with the local sequence number) to all messages which must be delivered in FIFO order. The local epoch number is stored in persistent memory and increases each time the node reboots. It is also increased when a remote node is declared "unreachable", in which case the epoch of the corresponding message queue is updated (the epoch numbers of other queues are left intact). The epoch numbers of remote nodes do not need to be stored in persistent memory. They are initialized when the first a message is received from that node and are updated when a message arrives carrying an epoch that is greater the previously recorded value (indicating a reset in the remote sequence numbering).

2.1.5 Report lists

For each report list created by a local agent, a corresponding data structure is maintained for storing and retrieving reports. These data structures are all kept in volatile memory. When a node reboots, this information is lost.

2.2 Host Candidate Discovery Protocol

This protocol is used to discover the nodes that are candidates for hosting an instance of a given agent type, subject to size constraints and (for non-generic agents) the non-generic resource requirements and the object qualifier expression provided by the application.

2.2.1 Description

To find candidates for hosting an instance of a given agent type, the middleware broadcasts a *HostProbeRequest* message to the POBICOS network and waits for *HostProbeReply* messages for a certain amount of time. *HostProbeRequest* messages carry information about the agent type, size and non-generic requirements as well as the object qualifier specified by the application and the application's priority. Due to the limited size of broadcast messages, it may be possible to sent only part of the non-generic requirements and/or object qualifier, in which case this first phase will produce "inaccurate" results (i.e., false positives).

Each *HostProbeReply* received is added in a candidate list. When the waiting time elapses, the candidate list is traversed to find the "best" candidate. Notably, there is no need to wait for the entire waiting time to elapse, and waiting can be terminated as soon as a "good enough" candidate replies. This is implementation specific.

When the middleware receives a *HostProbeRequest* message, it checks whether the locally available generic computing resources are sufficient to host the agent's code and static data. Also, if the agent type is non-generic, it checks whether the local node matches the object qualifier expression and meets the corresponding non-generic resource requirements; also that there is no other locally hosted non-generic agent of equal or higher priority that employs a conflicting non-sharable primitive. If all checks are successful, a *HostProbeReply* is sent back to the sender of the request, carrying the matching result (this can be further processed to pick the "best" reply). Notably, a reply serves just as a hint, i.e., the replying node does not reserve any local resources.

The *HostProbeRequest* message is broadcast as an unreliable datagram while the *HostProbeReply* message is sent as a reliable datagram, using the corresponding service of the networking abstraction layer. A simple sequence numbering scheme is used to verify that a *HostProbeReply* message corresponds to the most recently sent *HostProbeReply* message.

2.2.2 Message sequence diagram

The prototypical interaction for this protocol is as follows:

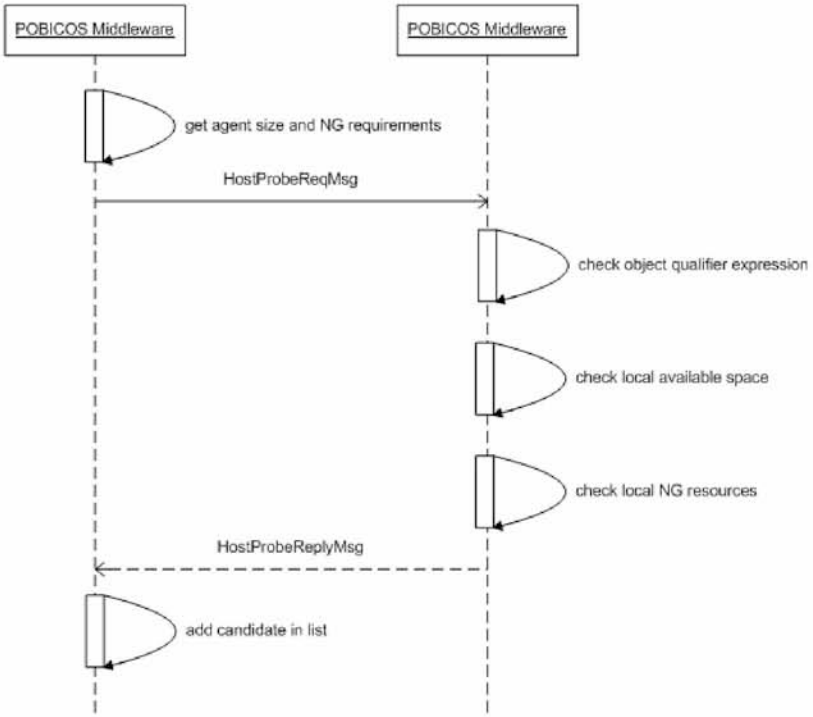

Fig 8.2 Message diagram for the Host Candidate Discovery protocol

2.3 Agent Creation Protocol

This protocol is used to create a new agent instance on a (specific) candidate node.

2.3.1 Description

Having picked a candidate for hosting an agent to be created (see Host Candidate Discovery Protocol), the middleware sends an *AgentCreationRequest* message to it and waits for an *AgentCreationReply* message. If the reply is positive, the child information of the local parent agent is updated and the agent is notified accordingly about child creation. If the reply is negative, the next candidate (if any) is considered.

When the middleware receives an *AgentCreationRequest* message, it checks that the object qualifier (if any) matches against the local object descriptor. Then, it fetches the code (if not already locally available) and the configuration settings for that agent type. Finally, it checks whether the local generic computing resources are sufficient to host the agent type, and, if the agent type is non-generic, whether the local node meets the corresponding non-generic resource requirements, and that there is no other locally hosted non-generic agent of equal or higher priority that employs a conflicting non-sharable primitive. If these checks are successful, a new

agent of the requested type is created locally (via calls to the runtime) and an *AgentCreationReply* message is sent back to the sender of the request carrying the identifier of the newly created agent. If any of these checks fail or the transfer of the agent code or its configuration settings fail or the runtime failed to instantiate the requested agent instance, the value zero (0) is returned instead of an agent identifier.

The *AgentCreationRequest* and *AgentCreationReply* messages are sent as reliable datagrams using the corresponding service of the networking abstraction layer (it is assumed that the entire information of a request fits within a reliable message; note that requests do not carry the non-generic requirements since these are extracted locally by the host, once the agent code is fetched). A simple sequence numbering scheme is used to verify that an *AgentCreationReply* message corresponds to the most recently sent *AgentCreationRequest* message.

While waiting for a reply from a node, *Ping* messages are sent periodically to it in order to check its operation, making sure that it (still) makes sense to wait for a reply. If the network reports that it was unable to deliver a *Ping* message to the destination, the next candidate (if any) is considered.

2.3.2 Message sequence diagram

The prototypical interaction for this protocol is as follows:

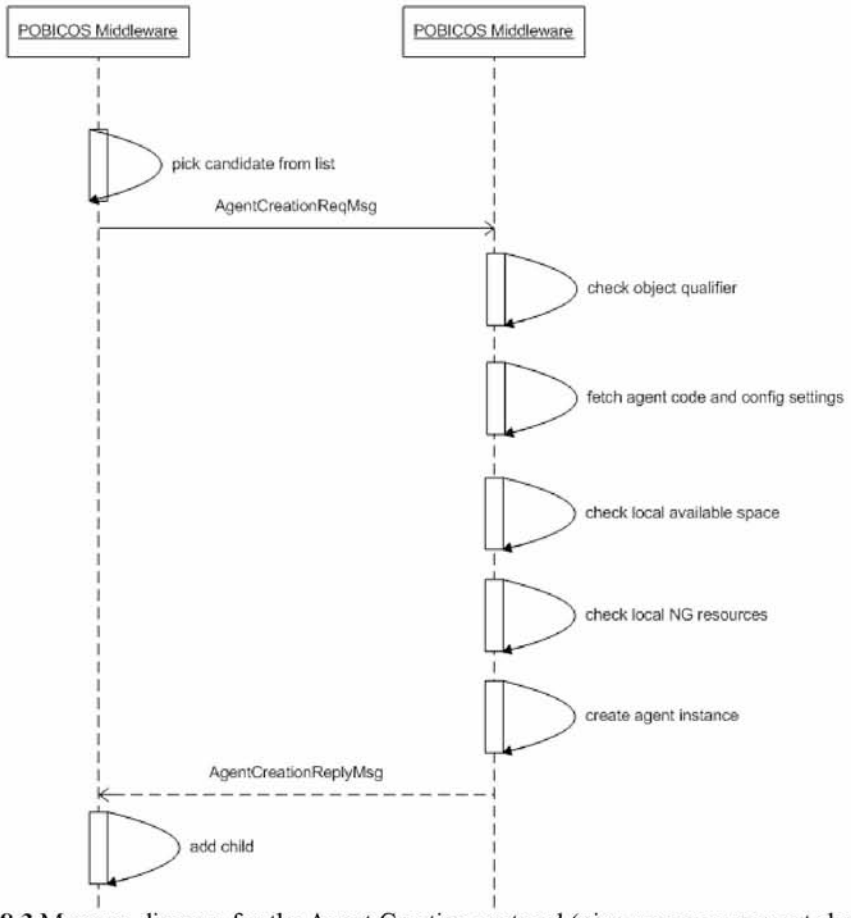

Fig 8.3 Message diagram for the Agent Creation protocol (ping messages are not shown)

2.4 Heartbeat Protocol

This protocol is used to refresh the lifetime of child agents as well as to detect the fact that an agent (parent or child) is unreachable.

2.4.1 Description

The liveness of agents is explicitly confirmed by periodically transmitting a *Heartbeat* message from the parent to its children. The middleware does this automatically, without any explicit request from the application.

When the middleware receives a *Heartbeat* message from the parent of a local agent, it extends the lifetime of that agent by a certain amount of time. If the lifetime of a local agent expires, i.e.,

a "sufficiently" long period of time passes by without having received a heartbeat from its parent, the agent is declared orphan. Consequently it is finalized and removed.

If the middleware receives a *Heartbeat* message for an agent that is not hosted locally, a corresponding *NAck* message is sent back to inform the sender that the agent does not exist, carrying some information about its non-existence, if possible. When the middleware receives a *NAck* message for a child of a local agent, it notifies the agent that the child is unreachable.

Heartbeat messages are sent as reliable datagrams whereas *NAck* messages are sent as unreliable datagrams. To avoid causal inconsistencies, *NAck* messages are delivered in a FIFO manner behind agent-level messages and thus carry corresponding sequencing information (epoch and sequence numbers). The sequencing logic is discussed in the sequel, as a part of the agent-level message transport protocol.

2.4.2 Message sequence diagrams

The prototypical interaction for this protocol is as follows:

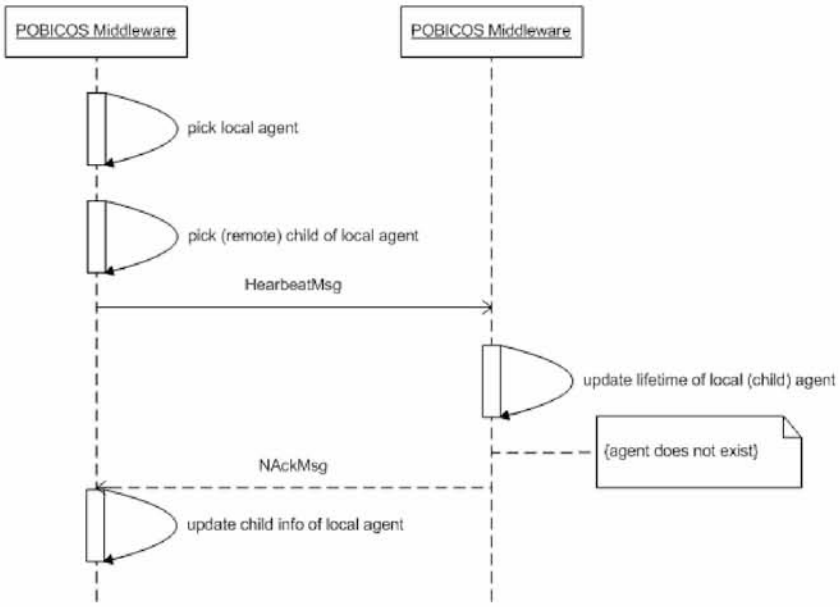

Fig 8.4 Message diagram for the Heartbeat protocol

2.4.3 Node-level heartbeats

A single or multiple local agents may have created several children on the same node. To avoid sending several heartbeat messages to the same node, each heartbeat (or application-level message) sent from a local parent to a child on a node also serves (i) as a heartbeat from that

parent to any other of its children that happen to be on that node, as well as (ii) a heartbeat from all other local parents to all of their children on the same node.

2.5 Agent-level Message Transport protocol

This protocol is used to transport (reliable and unreliable) agent-level messages (commands and reports) as well as negative acknowledgement messages.

2.5.1 Description

For each (reliable or unreliable) agent-level message, the middleware prepares a corresponding *AgentMsg* message and queues it up for transmission towards the node where the destination agent is hosted.

An *AgentMsg* message also serves as a heartbeat (see previous section). This means that the recipient is expected to generate a *NAck* message if the destination agent does not exist, just like for a *Heartbeat* message (see heartbeat protocol). Note that in this case, a *NAck* message may be issued towards a parent (indicating, as in the heartbeat protocol, that the child does not exist) as well as towards a child (indicating that the parent does not exist).

2.5.2 Sequencing

To achieve FIFO delivery, every message queue is associated with local and remote sequence number. The local sequence number is incremented each time an *AgentMsg* (or *NAck*) message is added in the queue, and the sequence number is also attached to the message itself.

The queue is traversed to forward messages to the network layer for transmission. Message transmission is suspended when a reliable message is handed over to the network layer, until its delivery is explicitly confirmed or the network layer reports a problem (see failure handling below).

When the middleware receives an *AgentMsg* (or *NAck*) message it checks its sequence number and accepts it only if it is greater or equal to the next expected sequence number for that (remote) node. Else the message must be dropped.

Due to the transmission policy on the sending side, it is impossible for an unreliable message to overtake a reliable message. As a consequence, only unreliable messages may arrive out of order, and can be dropped without violating the application-level delivery semantics. Nevertheless, a clever implementation can buffer out of order (unreliable) messages and wait for “late” messages to arrive.

AgentMsg messages are sent as reliable or unreliable messages, as requested by the application. *NAck* messages are also sent as unreliable datagrams. To avoid causal inconsistencies, *NAck* messages are queued behind agent-level messages and thus carry corresponding sequencing information.

2.5.3 Network and node failures

When the network reports that it was not possible to successfully transmit a reliable datagram, the corresponding remote node is declared "unreachable". In this case, messages queued up for transmission towards this node are dropped. Also, all agents known to be hosted on that node are declared "unreachable". If such an agent is the father of a local agent, the child is considered orphan and is terminated/finalized and removed. Else, if such an agent is the child of a local agent, the child is removed from the child list and the agent is notified about the child being "unreachable".

To deal with network failures and reboots, the middleware maintains a local epoch number. Each message queue is associated with a local and remote epoch number. When the middleware initializes (the local node boots) it increments its epoch number and assigns this value to each message queue. When a remote node is declared unreachable, the local epoch is incremented and assigned to the local epoch of the corresponding message queue while the local sequence number is reset to 0. The local epoch number associated with a message queue is attached together with the sequence number to all *AgentMsg* and *NAck* messages sent via that queue.

When the middleware receives from a node a message with a smaller than expected epoch number, it drops it. Messages with the expected epoch number are processed as usual (see sequencing). Finally, if a message with a greater than expected epoch number is received, the middleware knows that the remote node has declared the local node as unreachable, handles this case appropriately (as if it had also declared that node unreachable, but without increasing the local epoch), updates the epoch for that remote node and resets the corresponding sequence number to 0.

Notably, this approach allows a node to safely declare another node as unreachable, using whatever criterion is considered more realistic, without causing any serious inconsistency even if the node is actually alive. The price for doing this too "eagerly" is that nodes (and agents) can be declared as unreachable even if this is not the case in reality. It is up to the middleware implementation to decide when to declare a node as unreachable, e.g., when the network fails to deliver a reliable message to the destination node (after some number of attempts or a timeout).

2.5.4 Message sequence diagram

The prototypical interaction for this protocol is as follows:

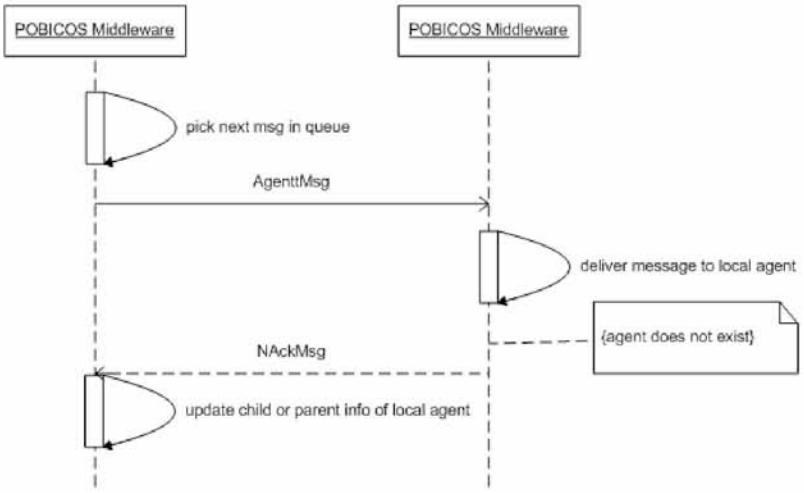

Fig 8.5 Message diagram for the Agent-level Message Transport protocol

2.6 Agent Migration Protocol

This protocol is used to move a locally hosted (generic) agent to a specific (given) remote node in a transparent fashion. The protocol works in multiple phases: (i) acquisition of the agent code and configuration settings; (ii) notification of the agent's parent and children that the migration starts; (iii) actual migration; (iv) notification of the agent's parent and children that the migration finished. The last phase also serves as a tie-break, in case migration succeeds but the old host nevertheless believes (due to a network partition or message transmission failure) that migration has not been completed successfully, letting the parent act as a common synchronization point.

Notably, this protocol does not address the problem of finding a suitable destination for a locally hosted agent, which is the subject of the so-called Agent Migration Algorithm (several options are discussed in the respective chapter of this document).

2.6.1 Description

When the middleware wishes to move a locally hosted agent to a given destination, it performs a series of communication rounds, as follows.

In a first step, the agent's host sends a *GetCodeRequest* message to the destination and waits for a *GetCodeReply* message. When the middleware receives a *GetCodeRequest* message, it fetches the code of the agent to be migrated (see reusable data item transfer protocol), if not already available, as well as the configuration settings for that agent. The result is communicated back to the host of the agent via a *GetCodeReply* message. The reachability of the destination is monitored (via *Ping* messages, as in the agent creation protocol). In case of a failure, the migration is aborted.

In a second step, the agent's host informs the nodes of the agent's parent and children about the (planned) migration via a *MigNotify* message and waits for corresponding *MigNotifyAck* replies. When the middleware receives a *MigNotify* message, it starts buffering all messages towards that agent (except heartbeats) until further notice, and replies with a *MigNotifyAck* message. If any of the nodes hosting the agent's parent or children become unreachable (again, this is detected via the periodic transmission of *Ping* messages), the migration is aborted, and the agent's host sends *MigFailed* messages to the nodes of the agent's parent and children.

When the agent's host receives all *MigNotifyAck* replies, it suspends the agent and retrieves its runtime state via the proper calls to the local runtime. Then, it waits until all outgoing messages issued by that agent are sent over the network.

In a third step, a *MigRequest* message is sent to the destination node, followed by one or more *AgentState* messages[1] carrying the full state of the agent (i.e., pending creation requests, children information, report lists and their contents, and runtime state). Upon receipt of these messages, the destination (to become the agent's new host) fetches the code and configuration settings of that agent type, creates a new instance, and initializes it using the state received. The result is reported via a *MigReply* message. If the *MigReply* is negative, the old host of the agent sends *MigFailed* messages to the nodes of the agent's parent and children. Else, if the *MigReply* is positive it simply removes the agent.

If the *MigReply* is positive, in a fourth step, the new host sends a *MigDone* message to the node of the agent's parent and waits for a *MigAck* or *MigNAck* reply. Upon receipt of a *MigAck* message, it sends *MigDone* messages to the hosts of the agent's children and resumes the execution of the agent (including the transmission of heartbeats to its children). Else, if a *MigNAck* message is received, indicating that the old host believes the migration has not been

[1] The reason for this fragmentation (sending the migration request and agent's state using different messages and splitting the state in more than one messages) is that the current network abstraction does not support arbitrarily large reliable datagrams neither does it provide a reliable stream abstraction.

completed and already notified the parent about this, the new host removes the agent (still in a suspended state).

The old host monitors the reachability of the new host (via *Ping* messages) until it receives a *MigReply*. In case of a failure, it (conservatively) assumes that migration has not been completed successfully. To "complete" the migration, it sends a *MigDone* message to the node that hosts the agent's parent (advertising its own address), and waits for a *MigAck* or *MigNAck* message. Upon receipt of a *MigAck* message, it sends *MigDone* messages to the hosts of the agent's children and resumes the execution of the agent. Else, if a *MigNAck* message is received, indicating that the new agent successfully completed the migration and notified the parent about this, the old host removes the agent (still in a suspended state).

When a node receives a *MigFailed* message it resumes agent-level message transmission to it. When a node receives a *MigDone* message, it does the same after adjusting the agent's node address. In addition, if the node is the agent's parent, it sends a *MigAck* message as a confirmation, before resuming normal message transmission. If the parent receives an unexpected *MigDone* message (for a child that is not under migration), it replies with a *MigNAck* message.

All messages are sent as reliable datagrams. Also, all messages except the ones related to the agent code transfer phase (*GetCodeReq* and *GetCodeReply*) carry a sequence number that is used to drop old messages (generated as a part of a previous instance of the migration protocol). Finally, *MigNotifyAck* messages are sent using the FIFO transport mechanism used for *AgentMsg* messages, so that their receipt also serves as a guarantee that there are no other *AgentMsg* messages in transit for the agent to be migrated. In the same spirit, the *MigDone* messages towards the children are also sent via the FIFO transport mechanism used for *AgentMsg* messages, so that they are guaranteed to precede any messages sent by the agent once it is resumed.

2.6.2 Message sequence diagram

The prototypical interaction for this protocol is as follows:

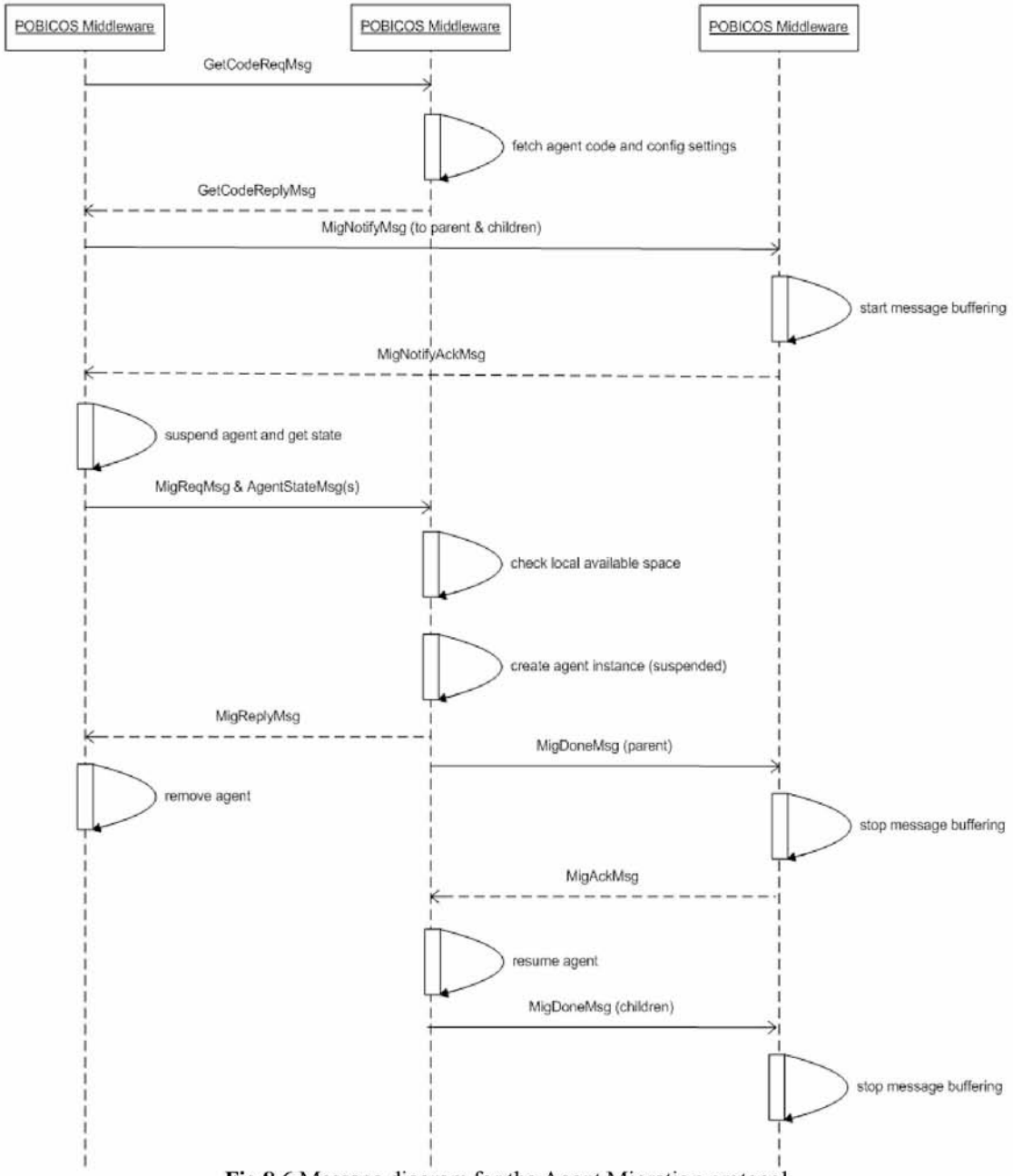

Fig 8.6 Message diagram for the Agent Migration protocol

2.7 Migration algorithms

We have implemented the *k*-hop variant of AGE algorithm, which assumes knowledge about the network routing structure within a *k*-hop radius and picks migration destinations in this range. Note that in ZigBee tree networks, routing information can be gained without extra communication, by exploiting the addressing scheme [86]; in essence, a node that receives a message can reconstruct the path to the source based on its address.

3 Middleware Evaluation

The evaluation of the agent manager takes place through conducting measurements about (i) the performance of the agent creation and migration mechanism (ii) the load reduction achieved when using agent migrations in context of a real application.

3.1 Performance measurements

This section presents measurements on the performance of agent creation and migration. The network topology is a 4-node chain, with the ZigBee coordinator at the one end as the source and other nodes as the destinations of the mobility operations.

The protocol cost is reported in bytes both for the Network Abstraction and ZigBee modem interface; the difference is due to datagram fragmentation. As a reference for the reported delays, the 1-hop throughput via the Abstract Network (incl. headers) is about 26Kbps and 15Kbps for unreliable and reliable datagrams, respectively. This poor performance is attributed to delays in accessing the CC2480 chip via SPI, but also middleware overheads, such as datagram fragmentation and software retransmission.

3.1.1 Agent creation overhead

In a first set of experiments, we measure the overhead for creating a non-generic agent with just one special resource need (e.g., a user activity sensor). The results for generic agents are similar. The delay for creating an agent locally is about 1ms.

Table 8.1 Agent creation cost breakdown and overhead for different agent sizes.

agent code size (B)	code transport protocol cost (B)		signaling protocol cost (B)		relative protocol overhead (B)	
	abstract	ZigBee	abstract	ZigBee	abstract	ZigBee
300	352	484	71	107	41%	97%
600	684	912	71	107	26%	70%
900	1032	1392	71	107	23%	67%

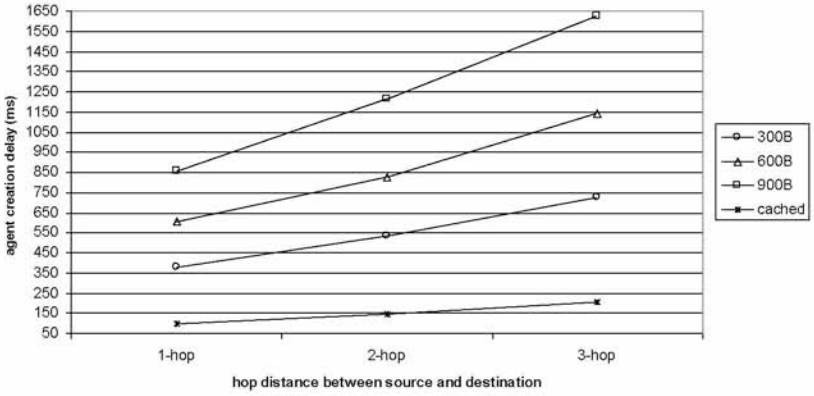

Fig 8.7 Agent creation delay as a function of hop distance for different agent sizes.

Table 8.1 analyzes the protocol cost for different agent sizes. The signaling overhead is constant and relatively low, corresponding to one host probe and one agent creation request-reply interaction. Clearly, the dominating part is the code transfer cost, which grows as expected to the agent size. The relative protocol overhead drops as code size increases, but the conversion of datagrams to ZigBee packets costs 35-40%.

Fig 8.7 plots the creation time, including the host probe phase, as a function of the hop distance between the source and the destination node for different agent sizes. It can be seen that the routing overhead is non-negligible. Naturally, the delay rises as the code size increases, yet with an economy of scale: about 21% and 24% for a 600B and a 900B agent vs. a 300B agent. Code transfer requires 3, 5 (+2) and 8 (+3) reliable datagrams (chunks) for 300B, 600B and 900B, which is why the creation of the 600B agent is slightly faster in relation.

We also performed measurements when the agent binary is cached at the destination node. In this case, the cost is solely due to the signaling protocol as per *Table 8.1*. The respective delay,

shown in *Fig 8.7*, is considerably smaller, yielding an average speedup of 3.7x, 5.8x and 8.4x for a 300B, 600B and 900B agent, respectively.

3.1.2 Agent migration overhead

In a second set of experiments, we measure the migration overhead for a generic agent that is co-located with its parent and has one child on a remote node to which it migrates directly. The runtime state is fixed at 256B. The delay for a corresponding agent suspend-create-init-resume cycle is about 2ms when performed locally.

The breakdown of the protocol cost is listed in *Table 8.2*. Naturally, the code transfer numbers

Table 8.2 Agent migration cost breakdown and overhead for different agent

agent code + runtime size (B)	code transport protocol cost (B)		signaling + state trans. protocol cost (B)		relative protocol overhead (B)	
	abstract	ZigBee	abstract	ZigBee	abstract	ZigBee
300+256	352	484	387	543	33%	85%
600+256	684	912	387	543	25%	70%
900+256	1032	1392	387	543	23%	67%

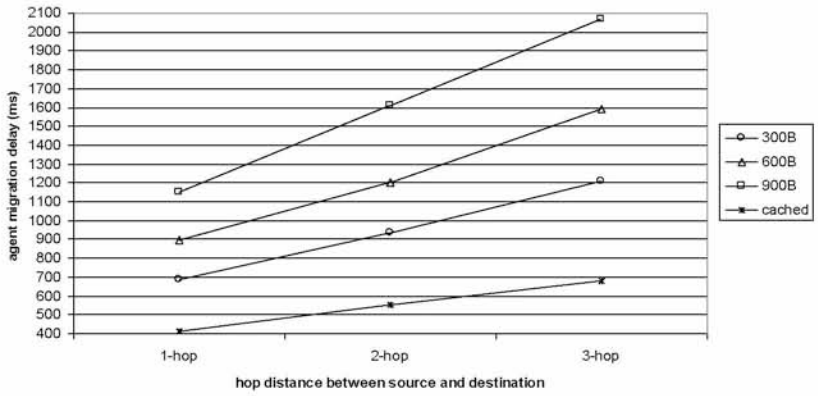

Fig 8.8 Agent migration delay as a function of hop distance for different agent sizes.

are the same as for agent creation. The signaling cost is much higher though because it includes the synchronization with the agent's parent and child, but also the transfer of the 256B state. As a result, the code transfer cost is less dominant compared to agent creation, amounting to 47% (vs. 82%), 67% (vs. 89%) and 72% (vs. 92%) of the protocol cost for a 300B, 600B and 900B agent, respectively.

Fig 8.8 plots the agent migration time as a function of the hop distance for different agent sizes. The trends are the same as for agent creation with the respective delays being longer due to the increased signaling and state transfer cost. The delay rises to the code size, but with a greater economy of scale compared to agent creation, about 35% and 43% for a 600B and a 900B agent vs. a 300B agent, due to the higher signaling cost. For the same reason, while caching reduces

the migration time, the speedup is less impressive: 1.7x, 2.2x and 2.9x for a 300B, 600B and 900B agent.

It is worth noting that a 2-hop migration is 32% faster compared to two 1-hop migrations, and a 3-hop migration is 40% faster than three 1-hop migrations. This holds even more if agent binaries are cached, the savings being 33% and 45%, respectively. This clearly speaks in favor of performing a single long-distance migration vs. several shorter-distance ones.

We also measured the migration time for a 600B agent with 256B runtime state for a varying number of its children residing on different 1-hop neighbors (using a star topology). The delay is 843ms, 874ms, 945ms and 974ms for 1, 2, 3 and 4 children, respectively (345, 400, 430 and 485 for a cached agent), rising due to the extra signaling needed for each child. The non-linearity from 2 to 3 children is due to the increase in the child information which happens to exceed the datagram payload limit, requiring an additional reliable transmission during the state transfer.

3.1.3 Summary

The results show that agent creation is fast enough to support the build-up and evolution of the application tree at runtime. Creation is very quick if a node has the binary cached (e.g., because it hosted such an agent in the past). Agent migration is also reasonably fast. Most importantly, since agents remain fully operational during the code transfer phase, the application is affected only by the signaling and state transfer delay; well under 1 second in our experiments (see the values reported for caching). This is acceptable for the applications we wish to support using our middleware, which have rather slack and soft real-time requirements. Note that an agent will notice the delay of a migration only if it expects to receive a message at the same point in time. Finally, the 1-hop throughput of the agent mobility operations, implemented largely using reliable datagrams, is 12-14Kbps. This is close to the throughput of our communication subsystem, which seems to be the main bottleneck. The practically instantaneous local creation and suspend-create-init-resume operations further attest to this fact.

3.2 Application scenario

In this section we put the benefit and cost of agent mobility in context of a concrete application scenario. Both the application and the network are kept simple in order to easily follow the operation of the POBICOS middleware. Still, the results are indicative of the potential gains in more complex and larger scale scenarios.

3.2.1 Application, network topology and test scenario

The test scenario involves an application to infer user absence based on all possible user activity sensors in a home: the root agent (R) creates a generic agent (I) for inferring user inactivity, which in turn creates an open number of non-generic user activity sensing agents (A). *Fig 8.9a* illustrates the corresponding tree structure.

As long as a sensing agent does not detect activity, it sends to the inference agent a 1-byte report every 5 seconds. When user activity is detected, the reporting frequency rises to 1 report per 2 seconds. Based on the reports received from its children, the inference agent sends a 1-byte

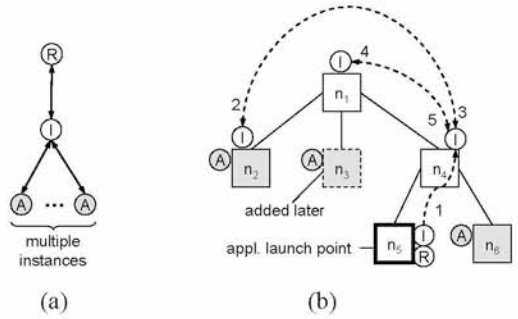

Fig 8.9 Experiment setup: (a) application tree; (b) nodes, network topology, and agent placement at different stages of the test scenario.

status report to the root every 10 seconds. The size of the root, inference, and user activity sensing agent is 50B, 240B and 24B, respectively.

Fig 8.9b shows the object/node network used to deploy and run the application. Nodes n_2, n_3 and n_6 represent objects with a user activity sensor, which can host a user activity sensing agent. The root remains fixed on n_5 from where the application is launched, while the generic inference agent can be placed on any node.

The initial node/network topology is that of *Fig 8.9b* without n_3, which is added and removed at later stages. The relevant stages of the test scenario are as follows:

The application is launched from n_5. The root and the inference agent are created on n_5, while user activity sensing agents are created on n_2 and n_6. Since the traffic with its children is larger than the traffic with its parent, the inference agent migrates on n_4.

The agent on n_2 detects user activity and starts reporting at a higher frequency. In turn, this increase in traffic drives the inference agent to migrate on n_2.

User activity stops, and the sensing agent on n_2 reverts to the normal reporting frequency. Consequently, the inference agent moves back on n_4.

Node n_3 (with a user activity sensor) is added to the network, leading to the creation of a sensing agent on it. As a result of this new child, the traffic for the inference agent via node n_1 becomes larger than the traffic with n_5 and n_6, so the inference agent migrates on n_1.

Finally, n_3 is removed, the local user activity sensing agent is killed, and the inference agent moves back on n_4.

Fig 8.9b shows the migrations and placements of the inference agent for each stage.

3.2.2 Results

Table 3 lists the results. It can be seen that the migration of the inference agent leads to considerable savings in network traffic, also at a cost that can be recovered within a relatively short amount of time of stable operation. Moreover, when the inference agent returns to a node where it was previously hosted (stages 3 and 5), caching halves the migration cost, also shortening the respective amortization time.

Table 8.3 Cost and benefit for each migration of the inference agent in the test scenario, as well as the time of stable operation required in order to amortize each migration.

scenario stages	migration of inference agent	migration cost (B x hops)	absolute traffic reduction (B x hops / min)	relative traffic reduction (%)	migration amortization (mins)
1	$n_5 \rightarrow n_4$	873	558	30%	1.5
2	$n_4 \rightarrow n_2$	1495	522	22%	2.7
3	$n_2 \rightarrow n_4$	769	486	27%	1.6
4	$n_4 \rightarrow n_1$	1007	174	8%	5.8
5	$n_1 \rightarrow n_4$	511	270	17%	1.9

In terms of real-time performance (not shown here), the average delay for creating a remote user activity agent is about 200ms. The migration delay for the inference agent is 620ms on average vs. 390ms when the code is cached at the destination. In both cases, migration delays were

unnoticeable at the application level, and are too insignificant to affect the respective amortization times.

Of course, a migration may turn out to be non-beneficial if the agent tree or traffic pattern changes fast. In our implementation we have two criteria for suppressing migrations that are unlikely to be beneficial, namely a migration is not performed unless it (i) reduces the amount of network traffic above a threshold and (ii) can be amortized within a certain amount of time, assuming stable operation. These checks can be computed locally. The network traffic after a migration can be computed based on the known agent message traffic while an estimate of the migration cost can be calculated using an analytical formula. Both checks are disabled in the experiment; they simply lead to fewer migrations, depending on the threshold settings.

4 Conclusions

In this chapter we briefly described how micro-agents are to be managed internally by POBICOS middleware. We also discussed, in a comprehensive way, the protocols used for the corresponding interaction between different instances of the middleware residing on different nodes. A number of experiments was conducted to evaluate the performance of agent creation and migration protocol, which comprise the most heavy (in terms of messages exchanged) functionalities of not only the agent manager, but also the POBICOS middleware. Among the algorithms proposed in the previous chapters, we chose AGE to be implemented in POBICOS middleware due to its eminent features: i) it is a fully distributed algorithm; ii) it needs only a small amount of both computational and storage resources; iii) it makes a decision to migrate an agent in an online manner. Finally, an indicative experiment was conducted to see AGE behaviour in a real system

Chapter 9

Related work

1 Systems that Support Mobile Code/Agents

Mobile code based systems are subsumed in the general category of systems that afford programming abstractions for WSNs [84]. Mate [66] is a general event-driven stack-based architecture allowing a user to select the bytecodes and execution events in order to build a virtual machine of his own preference. It focuses on simplifying application development via a high-level program representation, which allows the nodes of a network to be reprogrammed in a dynamic fashion. Rovers [27] is a middleware for tiny resource-constrained communicating nodes. Its agent-based programming model aims at freeing the programmer from the concept of the physical node by providing ontology-driven representation of sensors and actuators and implicit resource discovery.

One.world [41] is an architecture designed from the ground up to provide system support for pervasive application development. One of the system services, afforded by one.world is migration that moves or copies an environment (represents units of local computation) and all its contents to a different device. In [52] a system based on mobile code units, called Smart Messages, is described. Smart Messages (SMs) correspond to agents in our terminology. A key operation in the SM programming model is multi-hop migration, which implements routing using tags. An SM names the nodes of interest by tags, and then calls a high-level migrate function to route itself to a node that has the desired tags through multiple one hop migrations.

Agilla [36], adopts a mobile agent-based paradigm where programs are composed of agents that can migrate across nodes. A context manager determines the node location and maintains the

list of reachable neighbours. Migration is accomplished either by reliably relocating the agent, or by cloning it. Both strong and weak mobility is supported. Strong mobility ensures that the execution state is retained across movement, enabling the agent to resume execution right after the migration instruction. Instead, weak mobility moves only the agent code, whose execution restarts from scratch. A similar approach is adopted by Olympus framework [92] a high-level programming model for Active Spaces (i.e. a physically-bounded collection, such as a room of devices, objects, users, services and applications), while in SensorWare [20] only weak mobility is supported.

The above systems support user triggered/defined agent placement/migration. A small number of systems exist that automatically partitions an application into components (agents) and decide on their placement. The Pleiades compiler [60] performs data-flow analysis to partition the program in independent execution units called nodecuts, each running on a single node. The compiler assigns nodecuts to nodes based on the expected communication cost for accessing variables at remote nodes. MagnetOS [73] automatically and transparently partitions applications into components and dynamically places them on nodes to reduce energy consumption. The MagnetOS runtime also provides an explicit interface by which application writers can manually direct component placement. DFuse [96] is an architectural framework for dynamic application-specified data fusion in sensor networks. It can be used for developing advanced fusion applications (aggregation on data of possibly different types) that take into account the dynamic nature of applications and sensor networks. One of its main components is the distributed algorithm for fusion function placement and dynamic relocation that attempts to optimally place the fusion functions in the network nodes so that communication is minimized.

Summarizing, many systems provide support for mobile code and migration (strong mobility), adopting a 1-hop or k-hop network awareness (and migration) model. Some systems, such as Agilla, one.world, Smart Messages, Olympus, SensorWare, Pushpin and Mobile-C, let placement and/or migration be defined/triggered by the programmer. Other systems, such as MagnetOS, Pleiades and DFuse, automatically place and move code between nodes based on some optimization objective, typically related to the reduction of communication that takes place over the network. However to the best of our knowledge none of the systems reviewed considers: i) the case of storage constrained nodes; ii) the migration of a group of agents; iii) online algorithms to migrate agents; iv) maximize the network lifetime.

2 Data Placement and Replica Placement Problems

Algorithmic wise the agent migration problem (AMP) belongs to the general family of placement problems, whereby given a set of possible hosting entities and a set of objects, the problem is to place the objects at the entities so that performance is optimized. Placement problems have been studied in various fields, some of them not directly related to computer science, e.g., the facility location problem in operation research [101].

2.1 Data placement

In computer science one of the first placement problems to attract research interest was the so called file allocation problem (FAP). The first problem statements date the late 60's. [24] is one of the pioneering works tackling the problem of assigning files (single copies) to computers in a multi-computer environment in order to minimize the cost of answering user requests (read only) under storage constraints. They prove that under their formulation the problem experiences monotonic behaviour, i.e., each assignment reduces the cumulative cost and propose a branch and bound algorithm to solve it optimally. Extensions to the basic formulation included considering multiple file copies (replication), update requests, distinguishing between code and data allocation etc. A survey of early works in FAP can be found in [29]. [10] considers distributed FAP with read and write requests. An online Steiner-tree is built on which requests and replica creations are performed. To achieve competitiveness the algorithm bounds the cost of updates by deleting all object replicas when a write request is issued.

The above early works on allocation/placement are not directly related to the work on AMP we present here. However, they do provide a background as far as constructing a useful cost model concerns.

With the advent of the Internet and the World Wide Web, placement problems got renewed interest. Two main problem families were studied. The first aimed at placing network entities optimally. Papers in this subject include: [48], [67], [94], [107]. [67], [94] and [107] aimed at placing Web proxies at the network in order to improve user experienced response time, while [48] aimed at placing monitoring tools at the Internet in order to be able to estimate all-pair host distances based on accurately measuring a small portion of them. Typically, these papers use variations of the k-median problem [115] which can be briefly stated as: given a network graph with node weights representing user requests and link weights denoting a distance cost between

nodes, place k servers at the nodes so that the total cost of satisfying user requests (a function of node weights and distance), is minimized. For the case of a tree network, transitive distance cost, i.e., if d(u,v) is the distance between nodes u and v and d(v,w) is the distance between nodes v and w, then d(u,w)=d(u,v)+d(v,w), and linear target function of distance*node weight, exact solutions can be obtained by dynamic programming [115], [48].

Overall, we found k-median formulations less than useful, both for the centralized and the distributed problem versions we tackle. Nevertheless, it was important at the designing step to consider similar formulations even if we didn't adopt them.

2.2 Replica placement

The second family of placement problems that was extensively studied in the Web context was related to FAP with multiple file copies, often called replica placement. Solutions to the replica placement problem included both static centralized algorithms and dynamic distributed ones. [53] consider the problem of allocating Web objects at distributed Web servers with the aim of minimizing the background network traffic. The solution proposed was static and based on the greedy paradigm. [76] considered a similar static model and proposed a genetic algorithm to decide on object placement. They also gave extensions to the basic genetic algorithm which targeted at incrementally altering object allocation whenever slight changes in user request patterns occur. [55] evaluates different replica placement heuristics with the aim being network traffic cost, user response time, or server load balancing. [128] illustrates algorithms that decide separately on the number of object replicas and the locations they should be placed. Finally, [50] considers object placement in a tree-network with read and write requests. They give a dynamic programming algorithm for the uncapacitated case and prove that the problem is NP-hard when servers have capacity constraints.

Combinations of problems where object/replica placement is considered as a component have also been studied. [13] discussed the combined effects of static replica placement together with LRU caching. Their aim was to define the optimal split of storage space for long term replication and on demand caching. [77] tackled placement together with the implementation cost of it. Their goal was to define placements that are not expensive to implement, i.e., do not require many object transfers. The same authors also studied the implementation of placement by means of transfers and deletions as a separate problem in [78]. Notice, that in the centralized AMP once a newcoming agent is accepted, agent placement is altered in order to maximize the minimum node lifetime. This involves agent transfers which consume battery and affect the

optimization target function. Thus, the problem has a similar component to the one discussed in [78]. In fact RA adopts a simple algorithm to decide upon transfers that is inspired from [78]. Last, [107] considers both proxy and data placement in a k-median inspired manner.

The above works on object/replica placement make static assumptions regarding user request patterns and lead to centralized solutions. Although close in spirit to our work for the centralized AMP, AMP is different in a number of ways. The most important perhaps is the fact that the primary goal is to find/open space for a new agent. In the previous work creating an object replica is rather optional and is done only if it helps improving the optimization function. In our case accepting an agent is almost compulsory, therefore, the bin packing aspect of the problem is more important than the pure placement one.

A number of works exist on the dynamic/distributed replica placement. [109] proposes and compares static versus dynamic greedy heuristics for replica placement. [118] introduces the ADR algorithm, which creates, migrates and deletes replicas depending on the traffic direction and the relevant read to write ratio. [95] proposed a distributed algorithm that attempt to reduce simultaneously both the network traffic and server load imbalance. The core idea is that aside from deciding what to replicate where, a request routing scheme must be defined to judiciously distribute the load at the created replicas.

Some of the ideas used in the algorithms for distributed AMP are also found in the above works, namely, the migration towards the center of gravity of the communication load, or single hop migrations [118]. However, we differ from the above works in many ways. The most important one is that in the above works load is considered to originate from system nodes. In our case we might consider that the traffic between non-generic and generic agents is essentially traffic between nodes and generic agents, however, we also have traffic between generic agents or, put it in another way, the objects to be placed communicate with each other. As a result, algorithms that consider for migration each object separately are less powerful compared to the ones that form groups of objects/agents.

3 Energy Driven Algorithms

This section gives a flavour of the most related problems against the one introduced in Chapter 7. Specifially, our work is related to the greater area of energy management in wireless embedded systems, which is attracting much research interest. Most of the papers are dealing

with the problem of energy-aware routing. Existing work attempts to optimize power consumption mostly at routing level.

A wide number of papers address the problem of minimum energy routing [12], [97], [63], [32] to name but a few. [7] deals with the data-centric routing and proposes an algorithm building a special rooted broadcast tree with many leaves. By doing so, this algorithm keeps active only the relaying nodes while it turns off the radio on the leaves. Benerjee and Misra [16] argue that minimum-energy routing algorithms should not be based solely on the energy spent in a single transmission but on the total energy spent for a packet to be delivered to its final destination. [51] investigates the problem of energy-efficient broadcast routing over wireless static ad hoc network. It provides a globally optimal solution to the problem maximizing a static network lifetime through a graph theoretic approach. The case of power-aware georouting, whereby routing is done based on location and not address (thus no need to maintain routing information) is the objective of [26].

The approach of the above works is to minimize the energy spent on the network. However, there are a lot of papers, in the context of power-aware routing, aiming at maximizing the network lifespan [1], [69], [88], [22], [116] to name a few. In [1] the problem of maximizing system's lifespan (measured as the time when the first node dies) was formulated as a linear program. An optimal probabilistic data propagation algorithm maximizing network lifespan was proposed in [92], while [69] tackles the case where energy is replenished in a dynamic fashion. In [88] the authors study the impact of cooperative routing for maximizing the network lifetime in sensor networks. Chang and Tassiulas proposed a shortest cost path routing algorithm which uses link costs that reflect both the communication energy consumption rates and the residual energy levels at the two end nodes. Differently from previous solutions, the purpose of [116] is to maximize network lifetime by exploiting sink mobility. Specifically, the authors give a linear programming formulation for the joint problems of determining the movement of the sink and the sojourn time at different points in the network that induce the maximum network lifetime.

Other papers related to energy-driven algorithms are [17], [47], [49], [35], [46], [64]. [17] studies the problem of reducing energy dissipation by losslessly compressing data prior to transmission. The authors in [47] present an algorithm which automatically maps the IPs/cores onto a generic regular Network on Chip (NoC) such that the total communication energy is minimized. At the same time they try to not violate the constraints in terms of bandwidth reservation. [49] describes DE-MAC which is based on media access control technique. Specifically, DE-MAC treats the nodes having scarce energy resources differently in a

distributed manner, i.e., a weaker node should be used less frequently in a routing in order to accomplish load balancing. [35] shows how applications can dynamically modify their behavior to conserve energy. An energy-aware spanning tree algorithm is proposed by Lee and Wong [64] in the context of data aggregation. Specifically, this algorithm constructs a spanning tree based on the residual energy on nodes. The authors in [114] introduce energy-aware fault-tolerance heuristics in the context of real-time systems. [87] considers the problem of average throughput maximization per total consumed energy in sensor networks.

There are also papers that target at power management to achieve reduction in energy spent on machines. Sharma et al [99] investigate adaptive algorithms for dynamic voltage scaling in QoS-enabled Web servers to minimize energy consumption subject to service delay constraints; while [46] proposes power-aware algorithms that adapt its voltage and frequency setting to achieve reduction in energy dissipation with minimal impact on performance.

The papers dealing with the problem of minimizing the energy dissipation in wireless sensor networks are close to our works (excluding Chapter 7). However there are a lot of differences ranging from node storage constraints up to mutual agent dependencies. In terms of the aforementioned papers the ones related to the network lifespan maximization are rather similar to the work described in Chapter 7. Again the scope of our work is rather different from these papers since we try to both i) maximize the lifespan of the network by changing the placement of the application components; ii) make defragmentation in order for the nodes to be able to host as many application components as possible.

4 Load Balancing Problems

A significant part of the literature focuses on migrating jobs to distribute workload across multiple workstations (commonly known as load-balancing). An important part of the load-balancing strategy is the migration policy, which determines when migrations occur and which processes are migrated. There are two kind of strategies, the first one involves the preemptive migration where an active process may be suspended and migrate to another host [65], [61], [10] [15]; the second one concerns the non-preemptive load distribution which is based on initial placement of processes on the machines [106], [7], [28], [71], [82]. Another part concerns the selection of a new host for the migrated process, where [127] and [62] claim that the target

host should be the one with the shortest CPU run queue. [127] and [105] study load-balancing policies using a priori information about job lifetimes.

Most recent papers turn focus on distributing traffic among a set of diverse paths (i.e., routing with load balancing), especially when the target is a wireless sensor network due to its limited resources in terms of bandwidth. In [90] the authors provide an analytic model for evaluating the load balance as regards single shortest path routing in an ad hoc network. In terms of multi-path routing, they assume that load is uniformly distributed without considering the number of paths used and the way these paths are chosen. [38] is a relevant work to [90], where the authors propose an analytic model showing that multi-path routing results in a better load balance compared to single-path routing in case there is a very large number of paths between any source-destination pair nodes. While [45] proposes a load-balancing routing algorithm that lowers the bandwidth blocking rate to maximize network utilization. Last, in [44] two distributed algorithms are proposed for routing and load balancing in dynamic communication networks. Specifically The first algorithm is based on round trip routing agents that update the routing tables by backtracking their way after having reached the destination; while the second one relies on forward agents that update routing tables directly as they move towards their destination.

The papers dealing with the problem of migrating jobs to balance the load in a system are the most relative to our works, since the jobs and machines can be viewed as agents and nodes, respectively. However, our works differs from the above ones in that the agents are structured as a tree/graph and that there are two kind of agents (generic and non-generic).

5 Online Decision Problems

This section is directly related to Chapter 5, and discusses a lot of online algorithmic problems. The difficulty of the online decision problems lie in the fact that the input is only partially available because some relevant input data arrives in the future and is not accessible at present. Therefore an online algorithm should take a decision without knowledge of the entire output. The quality of such an online algorithm is usually evaluated using competitive analysis. The idea of competitiveness is to compare the output generated by an online algorithm to the output produced by an offline algorithm that knows the entire input data in advance and can compute

an optimal output. The better an online algorithm approximates the optimal solution, the more competitive this algorithm is [4]. A survey on online algorithms is given by [5].

One from the most renowned problems in the context of online decision problems is that of deciding which pages to keep in a memory of k pages in order to minimize the number of faults (i.e., paging problem). Sleator and Tarjan [102] provide two theorems, whereby the first one says that LRU and FIFO are k-competitive; while the second shows that no deterministic online algorithm for the paging problem can achieve a competitive ratio smaller than k. [33] proposes a randomized algorithm, called marking algorithm, and shows that it is $2H_k$-competitve where H_k denotes the kth harmonic number.

Scheduling has received a lot of research interest in the context of online strategies. Specifically, the problem is to assign jobs on machines in such a way as to minimize the makespan, which is the completion time of the last job that finishes in the schedule. Graham [25] proposed the elegant Greedy algorithm and analized its performance. Specifically, this algorithm assigns a new job to the least loaded machine and is (2-1/m)-competitive, where m represents the machines. Graham also showed that the competitive ratio of Greedy is not smaller than 2-1/m. In recent years the research community has focused on devising algorithms that achieve a competitive ratio asymptotically smaller than 2 [98], [3], [54], [34].

Online load-balancing can be viewed as a type of scheduling problem, where we have to minimize the maximum load instead of minimizing the makespan. [9] Studies the problem of minimizing the load on machines for the case where the tasks have limited duration. While [11] study the same problem provided that the task durations are not known upon their arrival. The authors prove also that the competitive factor of their algorithm is at most 4c, provided $c \geq 5$. Caragiannis et al [21] introduce the problem of how much the quality of load balancing is affected by selfishness and greediness. They prove that for any $e > 0$, greedy load balancing has competitiveness at least 17/3 - e, while greedy load balancing on identical servers has competitiveness at most $2/3\sqrt{21}+1$.

The most recent years a significant part of research community has turned its attention towards online decision problems in the context of WSNs. Even though the online routing problems have received a lot of interest [37], [8], [83], [68], [117]; there are also works that focus on other issues related to WSNs. For example, in [18] the authors analyze the theoretical complexity for the problem of gathering data in WSNs in a distributed fashion, and devise online algorithms solving this problem. [23] provides an online algorithm for the time interval

top-k query optimization to maximize the network lifetime through striking balance between the total energy consumption and the maximum energy consumption. Another work [58] presents an online algorithm to minimize the total transmission energy in a broadcast network by dynamically adjusting each node's transmission power and rate on a per-packet basis. Finally, [75] deals with the problem of mining frequent sensor value sets from a large sensor network.

None of the above papers is related to the problem of deciding in an online fashion whether the cost of an agent migration would be amortized in the future or not. However, they provided us some useful insights into devising the competitiveness of our algorithms. Last, if we were asked to say which is the most relevant work to ours, then we would refer to the papers dealing with the problem of online load balancing on machines (see previous section).

6 Query Optimization in Distributed Databases and WSNs

Generic agents in the POBICOS programming model carry the core application decision logic. Commonly, such logic involves filtering and aggregation of the data collected from sensors. Therefore, from the standpoint of data processing our work is related to database research on query optimization in general distributed systems [31] and (more recently) sensor networks [103].

Centralized query processing aims at defining the optimal sequence of filtering operations (WHERE clause in an SQL statement) as well as JOIN operations so that the query is answered in the minimum possible time. In doing so, the key parameters involved are table sizes and the selectivity of each filtering operation. Query answer time is usually assumed to be a function of the involved table sizes and the cost of the operation(s) on them. In distributed query processing network delays are taken into account. Each node only stores a portion of the database scheme. Answering a query might involve multiple nodes. An optimal query plan must decide whether a node should send its data elsewhere or must acquire data from other nodes and perform a partial join. [6] discussed the problem of optimally placing table fragments (data) in distributed nodes, assuming a fixed query plan that involves fetching all the necessary fragments to compute joins. [70] considered allocation when query plans involve partial join computations at intermediate nodes. For surveys on query optimization techniques including distributed query optimization the interested reader is referred to [39] and [59].

In the context of sensor networks much work has been done on how to efficiently perform aggregations, e.g., [79], [92], [30] to name a few. Usually, the aim is to reduce the amount of data sent by a node through either using special data structures, or delaying transmissions until absolutely necessary. Under the context of acquisitional query processing [80] discusses how often data must be propagated to query operators that (aside from simple aggregations) might involve complex filtering and joins. Reducing the cost of join operators in sensor networks has also seen much research activity, e.g., [123] while multiple query optimization in this context has been tackled in [120]. The above papers fall in the category of optimizing the execution of specific operators and/or joins. In doing so, they usually assume a fixed operator placement and discuss execution strategies that reduce the amount of transmitted data.

Perhaps the closest to our works from the query optimization literature are the ones tackling operator placement, e.g., [103], [1], [124] to name a few. In [103] the authors consider the problem of optimally placing query operators on the nodes of a sensor network given estimated operator costs. [1] proposed a greedy algorithm to solve the same problem, while [124] considered caching at intermediate nodes to reduce the fetching requirements of operators. The authors of [103] also studied the operator placement problem in the context of Web Services giving an optimal algorithm to perform Select-Project-Join queries [104]. In the POBICOS framework each operator is implemented as a generic agent. Therefore, at a first glance the two problems, i.e., of placing operators and of placing agents appear to be very similar. There is however, one important difference between works on operator placement and our work on AMP. Namely, there is a difference in scope. In our case, we attempt to optimize the placement of agents (operators) the behaviour of which is not known in advance since it is up to the user to decide. Therefore, we can only view the agents as black boxes and decide on their placement not according to their functionality, but rather according to their interaction with their environment, i.e., the load they incur.

7 Agent/Task Migrations

The concept of migrating agents instead of moving raw data to processing elements for data integrations is discussed in [93]. [119] takes a step forward, in the context of the previous work, by introducing the problem of computing a route for the mobile agent in terms of maximizing the received signal strength while keeping path loss and energy consumption low. They propose

a genetic algorithm to solve this problem since it turns out to be NP-hard. The same problem is considered in [121] and [122], with the difference that they focus on dynamic mobile agent planning techniques, which are distributed and dynamic in nature. They evaluate their algorithm's performance using three metrics: energy consumption, network lifetime, and the number of hops. [100] is a quite different work against the previous ones, since they consider agent migrations in the context of increasing efficiency of systems. However, the objectives of the above papers are completely different with the works discussed in this thesis.

The task allocation problem can be also viewed as the agent migration problem, since a migrating task may be represented by a migrating agent. In [43], [62], and [74] the authors consider the problem of mapping communicating tasks to homogeneous computing nodes in order to minimize execution time, while [108] considers the same problem in a heterogeneous environment. In [2] the authors tackle task allocation in an underlying torus network with the target of reducing both task communication and network congestion. In [42] the authors address the problem of finding a robust task allocation absorbing large changes of the environment without needing reallocation.

The fact that in the task allocation problem the tasks communicate with each other brings this problem closer to our works against the aforementioned agent migration problems which have no similarity to our ones. However these papers differ from the problems studied in this thesis either in the network and application structure assumed [43], [62], [74], [108], [2] as well as in scope [42].

8 Summary

Even though the area of energy management in sensor network systems is attracting a lot of research interest, there is no other work on either distributed or centralized agent migration algorithms aiming at bringing the communicating agents close to each other to reduce the energy spent over the network by considering solutions where i) the agents can migrate in groups; ii) the nodes have limited storage constraints; iii) agents can be evicted to create room to other agents which can eventually reduce the total network load; iv) the migration decisions are taken in an online fashion. Also, to the best of our knowledge there is no work that considers maximizing at the same time both the number of agents that a sensor network can host, and the lifespan of the later.

Chapter 10

Conclusions

1 Overview

The most crucial factors in terms of the sustainability of an application running on a sensor network are: i) the energy spent over the embedded nodes due to the communicating agents, since wasteful energy consumption may render a battery-powered node no operational; ii) as well as the limited resources the embedded nodes provide, since they may remove the right from an agent to be placed/migrated on a desired node. Inspired by the aforementioned issues, this thesis focuses on the agent migration problem to handle them in an efficient way.

In the first two chapers we proposed fully distributed algorithms to alleviate the total energy spent over the network, by performing beneficial migrations of agents or group of thems towards their center of gravity. The proposed algorithms are enhanced with two locking schemes to deal with the resource-constrained nodes. The third chapter discusses the bound of the proposed algorithms in a detailed way, and provides a modification of the grouping algorithm to make agent migration decisions in an optimal way.

In the sequel, though, we realized that when the nodes of the system provides scarce resources,then there is a lot of room for improving the solutions produced by the locking schemes studied in Chapter 1 and 2. This insight came through the fact that GRAL performed migrations in an almost optimal way when the nodes of the system provided a considerable amount of resources, while in the opposite case there was a discernible difference when comparing the placements resulted by GRAL and the optimal algorithm. Therefore we resorted

to the agent evictions (i.e. possibly no beneficial migrations) to enable a beneficial agent migration which eventually reduces the total network cost (taking into account the cost of the former ones). The algorithms proposed in Chapter 4 are based in the concept of agent evictions and they result in agent placements that are by far better in terms of energy dissipation against the aforementioned locking schemes.

Chapters 5 and 6 deal also with the problem of energy consumption but taking also into account the cost of the migrations performed. However, there are a lot of differences between them, since Chapter 5 proposes distributed algorithms that migrate only generic agents in an online fashion, while Chapter 6 focuses on offline centralized solutions based on the graph coloring problem, which aim at the network load reduction through migrating both generic and non-generic agents. Also, in Chapter 6 we adopt applications structured as a graph instead of a tree, unless otherwise stated.

Chapter 7 differs from the above works, since it formulates the agent migration problem for the two optimization goals of accepting a new agent and maximizing network longevity.

Finally, Chapter 8 discusses the implementation issues of how an agent migration/creation can take place in a POBICOS-enabled sensor network. AGE has also been implemented in POBICOS middleware, proving that some of the algorithms proposed in this thesis can be implemented in resource-constrained embedded systems.

2 Future Work

The distributed solutions dealing with the energy minimization problem assume tree-like structures for both the application and the underlying network. It would be challenging to devise new distributed algorithms along with their bounds when both application and network are organized as a graph. Also another future work would be to enhance the proposed distributed algorithms to consider non-generic agent migrations.

As regards the online decision problem discussed in Chapter 5, we plan to investigate algorithms that automatically learn to recognize patterns and make intelligent decisions based on their learning experience. Actually, such an algorithm could dynamically change parameters like migration threshold or reset threshold, which parameters turn out to be crucial for the

performance of the online algorithms. It would be also interesting to develop online algorithms for the problem of maximizing the network lifetime.

As part of our plans is the investigation of a distributed protocol that takes advantage of the distributed nature of PRA when considering migrations between node pairs. There is also a lot of room for the problem discussed in Chapter 7, since it would be quite challenging to deal with it through distributed solutions.

The root agent of an application may experience delay from the time the data are sensed till they are accessible (in a fused manner) to it. However, such a delay may prove crucial for the functionality of a real-time application. Therefore a future direction could be the energy minimization without violating some pre-specified delay constraints. Also, this problem could be investigated in its own right (without considering the energy minimization aspect), through algorithms that dynamically reform the application structure to meet delay constraints.

References

[1] Abrams Z. and Liu J., "Greedy is good: On service tree placement for in-network stream processing," in Proc. of ICDCS, 2006.

[2] Agarwal T., Sharma A., Laxmikant A. and Kale L.V., "Topology-aware task mapping for reducing communication contention on large parallel machines," in Proc. 20th Int. Parallel and Distributed Processing Symp. (IPDPS 06).

[3] Albers S., "Better bounds for online scheduling", SIAM J. Comput. 29, pp. 459-473, 1999.

[4] Albers S., "Competitive online algorithms", OPTIMA: The Mathematical programming Society Newsletter, 1997.

[5] Albers S., "Online algorithms: A survey", Mathematical Programming, 2003.

[6] Apers P.M.G., "Data Allocation in Distributed Database Systems", ACM Transactions on Database Systems, 13(3), 1988.

[7] Artsy Y., Finkel R., "Designing a process migration facility: The Charlotte experience", IEEE Comput. 22(1), 198l.

[8] Aslam J., Li Q., Rus D., "Three power-aware routing algorithms for sensor networks", Wireless Communications and Mobile Computing", 3(2), 2003.

[9] Aspenes J., Azar Y., Fiat A., Plotkin S., Waarts O., "On-line machine scheduling with application to load balancing and virtual circuit routing", In Proc. Annual ACM Symposium on Theory of Computing", 1993.

[10] Awerbuch B., Bartal Y., Fiat A., "Optimally-Competitive Distributed File allocation," 25th Annual ACM Symposium on Theory of Computing (STOC), 1993.

[11] Azar Y., Kalyanasundararm B., Plotking S., Pruhs K., Waarts O., "On-line load balancing of temporary tasks", In Proc. Workshop on Algorithms and Data Structures, 1993.

[12] Baker D.J., Ephremides A., "The architectural organization of a mobile radio network via a distributed algorithm", Transactions on Communications, 1981.

[13] Bakiras S., Loukopoulos T., "Increasing the Performance of CDNs Using Replication and Caching: A Hybrid Approach," IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2005.

[14] Balter M.H., Downey A.B., "Exploiting Process Lifetime Distributions for Dynamic Load Balancing", in Proc TOCS, 1997.,

[15] Barker K.J., Chrisochoides N.P., "An evaluation of a framework for the dynamic load balancing of highly adaptive and irregular parallel applications", In Proc ICS, 2003

[16] Banerjee S., Misra A., "Minimum Energy Paths for Reliable Communication in Multi-Hop Wireless Networks", in Proc Mobihoc, 2002.

[17] Barr K.C, Asanovic K., "Energy-Aware Lossless Compression", ACM Transactions on Computer Systems, 24(3), 2006.

[18] Bonifaci V., Korteweg P., Marchetti-Spaccamela A., Stougie L. "The distributed wireless gathering problem", In Proc. ICAMWN, 2008.

[19] Boukerche A., Cheng X., Linus J., "Energy-Aware Data-Centric Routing in Microsensor Netowkrs", in Proc MSWIM, 2003.

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

[21] Caragiannis I., Flammini M., Kaklamanis C., Kanellopoulos P., Loscardelli L., "Tight bounds for selfish and greedy load balancing", In Proc. ICALP, 2006.

[22] Chang J.H., Tassiulas L., "Maximum lifetime routing in wireless sensor networks", Transactions on Networking, 12(4), 2004.

[23] Chen B., Liang W, Xu Yu J.,"Online time interval top-k queries in wireless sensor networks", In Proc. ICMDM, 2010.

[24] Chu W., "Optimal File Allocation in a Multiple Computer System," IEEE Transactions on Computers, 18(10), 1969.

[25] Crossbow, "Imote2 Hardware Reference Manual", available at: http://www.cse.wustl.edu/wsn/images/9/90/Imote2_hardware_ref.pdf.

[26] Das S., Nayak A., Ruhrup S. and Stojmenovic I., "Semi-Beaconless Power and Cost Efficient Georouting with Guaranteed Delivery Using Variable Transmission Radii for Wireless Sensor Networks," in Proc. MASS 2007.

[27] Domaszewicz J., Roj M., Pruszkowski A., Golanski M., Kacperski K., "ROVERS: Pervasive Computing Platform for Heterogeneous Sensor-Actuator Networks". Proc. WoWMoM 2006.

[28] Douglis F., Ousterhout J., "Transparent process migration: Design alternatives and the Sprite implementation" Softw. Pract. Exper. 21(8), 1991.

[29] Dowdy L., Foster D., "Comparative Models of the File Assignment problem," ACM Computing Surveys, 14(2), 1982.

[30] Edara P. and Ramamritham K., "Asynchronous In-Network Prediction: Efficient Aggregation in Sensor Networks," in ACM TOSN, vol. 4(4):25, Aug. 2008.

[31] Epstein R., Stonebraker M. and Wong E., "Distributed query processing in a relational data base system," in Proc. ACM SIGMOD 1978.

[32] Falls N., "On the Problem of Energy Efficiency of Multi-Hop vs One-Hop Routing in Wireless Sensor Networks", In Proc IANAW, 2007.

[33] Fiat A., Karp R.M., Luby M., McGeoch L.A., Sleator D.D., Young N.E., "Competitive paging algorithms", Journal of Algorithms, 12(4), 1991.

[34] Fleischer R., Wal M., "Online scheduling revisited", J. Scheduling 3, pp. 343-353, 2000.

[35] Flinn J., Satyanarayanan M., "Managing battery lifetime with energy-aware adaptation", ACM Transactions on Computer Systems, 22(2), 2004.

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

[37] Fontanelli D., Palopoli L., Passerone R., "On the global convergence of a class of distributed algorithms for maximizing the coverage of a WSN", in Proc CDC/CCC, 2009.

[38] Ganjali Y., Keshavarzian A., "Load balancing in ad hoc networks: single-path routing vs. multi-path routing", In Proc INFOCOM, 2004.

[39] Graefe C., "Query evaluation techniques for large databases," in ACM Computing Surveys, vol. 25(2), pp. 73–170, 1993.

[40] Graham R.L., "Bounds for certain multiprocessor anomalies", Bell Syst. Technical J, 1966.

[41] Grimm R., Davis J., Lemar E., Macbeth A., Swanson S., Anderson T., Bershad B., Borriello G., Gribble S., Wetherall D., "System support for pervasive applications", ACM Transactions on Computer Systems, 22(4), 2004.

[42] Gu D., Drews F. and Welch L.R., "Robust task allocation for dynamic distributed real-time systems subject to multiple environmental parameters," in Proc. 25th Int. Conf. on Distributed Computing Systems (ICDCS 05).

[43] Heiss H.U. and Schmitz M., "Decentralized dynamic load balancing: The particles approach," in Proc. 8th Int. Symp. on Computer and Information Sciences (ISCIS 95).

[44] Heusse M., Snyers D., Guering S., Kuntz P., " Adaptive agent-driven routing and load balancing in communication networks", ACS, 1(2), 1998.

[45] Hsiao P.H., Hwang A., Kung H.T., Vlah D., "Load-balancing routing for wireless access networks", In Proc INFOCOM, 2001.

[46] Hsu C.H., Feng W.C, "A Power-Aware Run-Time System for High-Performance Computing", in Proc SC 2005.

[47] Hu J., Marculescu R, "Energy-Aware Mpapping for Tile-based NoC architectures under performance constraints", in Proc ASP-DAC 2003.

[48] Jamin S., Jin C., Jin Y., Riaz D., Shavitt Y., Zhang L., "On the Placement of Internet Instrumentation," IEEE INFOCOM, 2000.

[49] Kalindindi R., Kannan R. Iyengar S.S., Ray L., "Distributed Energy Aware Mac Layer Protocol for Wireless Sensor Networks", in Proc ICWN 2003.

[50] Kalpakis K., Dasgupta K., Wolfson O., "Optimal Placement of Replicas in Trees with Read, Write and Storage Costs," IEEE Transactions on Parallel and Distributed Systems (TPDS), 12(6), 2001.

[51] Kang I., Poovendran R., "Maximizing Static Network Lifetime of Wireless Broadcast Ad Hoc Networks", in Proc ICC 2003.

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

[53] Kangasharju J., Roberts J., W. Ross K., "Object Replication Strategies in Content Distribution Networks", Computer Communications, 25(4), 2002.

[54] Karger D.R, Phillips S.J., Torng E. J. Algorithms 20, pp. 400-430, 1996.

[55] Karlsson M., Karamanolis C., "Choosing Replica Placement Heuristics for Wide-Area Systems", IEEE International Conference on Distributed Computing Systems (ICDCS), 2004.

[56] Kellerrer H., Pferchy U. and Pisinger D., Knapsack Problems Springer, Oct. 2004.

[57] Kim H.S., Abdelzaher T.F. and Kwon W.H., "Dynamic Delay-Constrained Minimum-Energy Dissemination in Wireless Sensor Networks," in ACM Trans. on Embedded Computing Systems, Vol. 4 (3), pp. 679-706, 2005.

[58] Kompella R.R., Shoeren A.C., "Practical lazy scheduling in sensor networks", In Proc SenSys, 2003.

[59] Kossmann D., "The state of the art in distributed query processing", ACM Computing Surveys, 32(4), 2000.

[60] Kothari N., Gummadi R., Millstein T., Govindan R., "Reliable and efficient programming abstractions for wireless sensor networks", ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2007.

[61] Krueger P., Livny M., "A comparison of preemptive and non-preemptive load distributing", In Proc ICDCS 1988.

[62] Kunz T., "The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme," in IEEE Transactions on Software Engineering., Vol. 17 (7) , pp. 725-730, 1991.

[63] Lee H., Lee K., "Energy Minimization for Flat Routing nd Hierarchical Routing for Wireless Sensor Networks", In Proc ICSTA, 2008.

[64] Lee M., Wong V.W.S., "An Energy-Aware Spanning Tree Algorithm for Data Aggregation in Wireless Sensor Networks", in Proc PACRIM 2005.

[65] Leland W.E., Ott T.J., "Load-balancing heuristics and process behavior" In Proc of Sigmetrics, 1986.

[66] Levis P. and Culler D., "Mate: A Tiny Virtual Machine for Sensor Networks," in Proc. ASPLOS 2002.

[67] Li B., Golin M., Italiano G., Deng X., "On the optimal placement of web proxies in the Internet", IEEE INFOCOM, 2000.

[68] Li Q., Aslam J., Rus D., "Online power-aware routing in wireless ad-hoc networks", In Proc MobiCom, 2001

[69] Lin L., Shroff N. and Srikant R., "Asymptotically Optimal Power-Aware Routing for Multihop Wireless Networks with Renewable Energy Sources," in Proc. INFOCOM 2005.

[70] Lin X., Orlowska M., Zhang Y., "On Data Allocation with the Minimum Overall Communication Costs in Distributed Database Design", 5th IEEE International Conference on Computing and Information (ICCI), 1993.

[71] Litskow M., Linvy M., Mutca M., "Condor – A hunter of idle workstations" In Proc ICDCS 1988.

[72] Littman M.L., Boyan J.A., "A Distributed Reinforcement Learning Scheme for Network Routing", In Advances in Neural Information Processing Systems, vol. 6, pp.670-678, 1993.

[73] Liu H., Roeder T., Walsh K., Barr R., Sirer E.G., "Design and implementation of a single system image operating system for ad hoc networks", 3rd International Conference on Mobile Systems, Applications and Services (MOBISYS), 2005.

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

[75] Loo K.K., Tong I., Kao B., "Online algorithms for mining inter-stream associations from large sensor neworks", Lecture Notes in Computer Science, 2005.

[76] Loukopoulos T., Ahmad I., "Static and Adaptive Data Replication Algorithms for Fast Information Access in Large Distributed systems", 20th IEEE International Conference on Distributed Computing Systems (ICDCS), 2000.

[77] Loukopoulos T., Lampsas P., Ahmad I., "Continuous Replica Placement Schemes in Distributed Systems", International Conference on Supercomputing (ICS), 2005.

[78] Loukopoulos T., Tziritas N., Lampsas P., Lalis S. : Implementing Replica Placements: Feasibility and Cost Minimization. Proc. IPDPS 2007.

[79] Madden S., Franklin M., Hellerstein J. and Hong W., "TAG: A tiny aggregation service for ad-hoc sensor networks," in Proc. 5th USENIX Symposium on OSDI, Dec. 2002.

[80] Madden S., Franklin M., Hellerstein J., Hong W., "The design of an acquisitional query processor for sensor networks," in Proc. ACM SIGMOD 2003.

[81] Manolache S., Eles P., Peng Z, "Fault and Energy-Aware communication mapping with guaranteed latency for applications implemented on NoC", in Proc DAC 2005.

[82] Milojicic D.S., "Load distribution: Implementation for the Mach microkernel. Ph.D. dissertation, Univ. of Kaiserslautern, Kaiserslautern, Germany, 1993.

[83] Minhas M.R., Gopalakrishnan S., Leung V.C.M., "An online multipath routing algorithm for maximizing lifetime in wireless sensor networks", in Proc ITNG, 2009.

[84] Mottola L., Picco G.P., "Programming Wireless Sensor Networks: Fundamental Concepts and State of the Art", to appear in ACM Computing Surveys (2010).

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

[86] Pan M.-S., Fang H.-W., Liu Y.-C., Tseng Y.-C.: Address Assignment and Routing Schemes for Zigbee-based Long-thin Wireless Sensor Networks. 67th IEEE International Conference on Vehicular Technology (VTC), 2008.

[87] Pandana C., Liu K.J.R., "Near-Optimal Reinforcement Learning Framework for Energy-Aware Sensor Communications", In IEEE Journal on Selected Areas in Communications, vol. 23, pp. 788-797, 2005.

[88] Pandana C., Siriwongpairat W.P., Himsoon T., " Distributed Cooperative Routing Algorithms for Maximizing Network Lifetime", in Proc WCNC, 2006.

[89] Pearlman M.R., Haas Z.J., Sholander P., Tabrizi S.S., "Mobile and ad hoc networking and computing, In Proc MobiHOC, 2000.

[90] Pham P.P., Perreau S., "Performance analysis of reactive shortest path and multi-path routing mechanism with load balance", In Proc INFOCOM, 2003

[91] POBICOS project web site, http://www.ict-pobicos.eu.

[92] Powell O., Leone P. and Rolim J., "Energy Optimal Data Propagation in Wireless Sensor Netowrks," in Elsevier JPDC, vol. 67(3), pp. 302-317, 2007.

[93] Qi H, Iyengar S., Chakrabarty K., "Multiresolution data integration using mobile agents in distributed sensor networks", IEEE Transactions on Applications and Reviews, 2001.

[94] Qiu L., Padmanabhan V., Voelker G., "On the Placement of Web Server Replicas", IEEE INFOCOM, 2001.

[95] Rabinovich M., Rabinovich I., Rajaraman R., Aggarwal A., "A dynamic object replication and migration protocol for an Internet hosting service", 19th IEEE International Conference on Distributed Computing Systems (ICDCS), 1999.

[96] Ramachandran U., Kumar R., Wolenetz M., Cooper B., Agarwalla B., Shin J., Hutto P., Paul A., "Dynamic data fusion for future sensor networks", ACM Transactions on Sensor Networks, 2(3), 2006.

[97] Rodoplu V., Meng T.H., "Minimum Energy Mobile Wireless Networks", In Proc. INFOCOM ,1997.

[98] Rudin J.F., "Improved bounds for the on-line scheduling problem", Ph.D. Thesis, the university of Texas, Dallas, 2001.

[99] Sharma V., Thomas A., Abdelzaher T., Skadron K., Zhijian L., " Power-aware QoS management in web servers", In Proc RTSS, 2003.

[100] Shehory O., Sycara K., Chalasani P., Jha S. "Agent cloning: an approach to agent mobility and resource allocation", Communications Magazine, 1998.

[101] Shmoys D., Tardos E., Aardal K., "Approximation algorithms for facility location problems", 28th ACM Symposium on Theory of Computing (STOC), 1997.

[102] Sleator D.D., Tarjan R.E., "Amortized efficiency of list update and paging rules", ACM Commun, 1985.

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

[104] Srivastava U., Munagala K., Widom J. and Motwani R., "Query optimization over web services," in Proc. VLDB 2006.

[105] Svensson A., "History, an intelligent load sharing filter", In Proc ICDCS 1990.

[106] Tanenbaum A., Renesse V.H, Staveren V.H., Sharp G.,"Experiences with the Amoeba distributed operating system", ACM Commun., 1990

[107] Tang X., Xu J., "QoS-Aware Replica Placement for Content Distribution", IEEE Transactions on Parallel and Distributed Systems (TPDS), 16(10), 2005.

[108] Taura K. and Chien A., "A Heuristic Algorithm for Mapping Communicating Tasks on Heterogeneous Resources," in Proc. 9th Heterogeneous Computing Workshop (HCW 00).

[109] Tenzakhti F., Day K., Olud-Khaoua M., "Replication Algorithms for the Word-Wide Web", Journal of System Architecture, 50, 2004.

[110] Texas Instruments: Z-Accell Demonstration Kit, http://focus.ti.com/docs/toolsw/folders/print/ez430-rf2480.html.

[111] Tian Y., Boangoat J., Ekici E. and Özgüner F., "Real-time task mapping and scheduling for collaborative in-network processing in DVS-enabled wireless sensor networks," in Proc. 20th Int. Parallel and Distributed Processing Symp. (IPDPS 06).

[112] Tziritas N., Loukopoulos T., Lalis T., Lampsas P. : Agent placement in wireless embedded systems: memory space and energy optimizations. Proc. 9th Int. Workshop on Performance Modeling, Evaluation and Optimization of Ubiquitous Computing and Networked Systems (PMEO-UCNS10).

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

[114] Unsal O.S., Kren I., Krishna C.M.," Towards Energy-Aware Software-based Fault Tolerance in Real-Time Systems", in Proc. ISLPED 2002.

[115] Vigneron A., Gao L., Golin M., Italiano G., Li B., "An algorithm for finding a k-median in a directed tree," Information Processing Letters, 74, 2000.

[116] Wang Z.M., Basagni S., Melachrinoudis E., Petrioli C., "Exploiting Sink Mobility for Maximizing Sensor Networks Lifetime", In Proc HICSS, 2005.

[117] Watfa M., Yaghi L., "An efficient online-battery aware geographic routing algorithm for wireless sensor networks", International Journal of Communication Systems, 23(1), 2010.

[118] Wolfson O., Jajodia S., Huang Y., "An Adaptive Data Replication Algorithm", ACM Transactions on Database Systems, 22(4), 1997.

[119] Wu Q., Rao N. S. V., Barhen J., et al., "On computing mobile agent routes for data fusion in distributed sensor networks," in IEEE Transactions on Knowledge and Data Engineering, Vol. 16 (6), pp. 740–753, 2004.

[120] Xiang S., Lim H. B., Tan K. L. and Zhou Y., "Two-Tier Multiple Query Optimization for Sensor Networks," in Proc. ICDCS 2007.

[121] Xu Y., Qi H, " Mobile agent migration algorithms for collaborative processing", In Proc WCNC, 2006.

[122] Xu Y., Qi H, " Mobile agent migration modeling and design for target tracking in wireless sensor networks", In Proc WCNC, 2006.

[123] Yang X., Lim H. B., Özsu M. T. and Tan K. L., "In-Network Execution of Monitoring Queries in sensor Networks," in Proc. SIGMOD 2007.

[124] Ying L., Liu Z., Towsley D. and Xia C.H., "Distributed Operator Placement and Data Caching in Large Scale Sensor Networks," in Proc. INFOCOM 2008.

[125] Younis M., Youssef M., Arisha K., "Energy-Aware Routing in Cluster-based Sensor Networks", In Proc. MASCOT 2002.

[126] Youssef M., Younis M., Arisha K., "A Constrained Shortest-Path Energy-Aware Routing Algorithm for Wireless Sensor Networks", in Proc. WCNC 2002.

[127] Zhou S., Wang J., Zheng X., Delisle P., "Utopia: A load-sharing facility for large heterogeneous distributed computing systems., Softw. Pract. Exper. 23(2), 1993.

[128] Zhuo L., Wang C-L, Lau F. C. M., "Document Replication and Distribution in Extensible Geographically Distributed Web Servers", Journal of Parallel and Distributed Computing, 63(10), 2003.

[129] Zigbee Alliance: Zigbee specification (2006), http://www.zigbee.org.