Computer Communications 36 (2013) 1011–1023

Contents lists available at SciVerse ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

Improving the dynamism of mobile agent applications in wireless sensor networks through separate itineraries

Estanislao Mercadal [a,*], Carlos Vidueira [a], Cormac J. Sreenan [b], Joan Borrell [a]

[a] Department of Information and Communications Engineering, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain
[b] Department of Computer Science, University College Cork, Cork, Ireland

ARTICLE INFO

Article history:
Available online 13 October 2012

Keywords:
WSN mobile agent applications
Agilla
Separate itineraries
Emergency scenarios

ABSTRACT

This paper introduces the deployment of a new type of mobile agent application in wireless sensor networks (WSNs) that implements the construct of separate itinerary, a classic concept from Concordia, a mobile agent system in conventional distributed environments. A separate itinerary is a completely separate data structure from the agent itself, providing a simple mechanism to flexibly define and track how an agent travels. Our contribution is twofold: First, the adaptation of separate itineraries to a highly resource-constrained environment, i.e. Agilla agents on TelosB nodes. Second, the demonstration of this adaptation using a new application in the field of emergency scenarios – dynamic Mobile Agent Electronic Triage Tag. This application shows the impact of fault-tolerance through the use of several agents with different migration strategies from separate itineraries. By considering the itinerary separately from the agent code, we gain an additional level of adaptability and reactivity in Agilla applications, similar to that already available in WISEMAN applications.

1. Introduction

A strategy to increase dynamism (reactivity) of WSNs is reprogramming them over the wireless network. Through this reprogramming WSNs no longer have to be static in their operation, but they can adapt their behavior, reacting to the changing conditions of their environment or the changing needs of their deployers.

A couple of good surveys on WSN reprogramming systems can be found in [1,2]. These systems can be classified according to how the decision to reprogram is made. In a number of systems (based either on code execution, e.g. [3,4], or code interpretation, e.g. [5,6], or both, e.g. [7]) the reprogramming decision is determined centrally at a base station, often by a human operator. However, maximum reactivity is reached when reprogramming decisions can be made locally and autonomously in any node of the WSN. To this end, a number of mobile agent middlewares for WSNs have been proposed [8,1,9–11,2] allowing each WSN node to execute different code (different agents), and allowing the evolution of this code.

Mobile agents are special processes that can autonomously migrate or clone from node to node while maintaining their state and are able to communicate and coordinate with other agents. Mobile agents need an execution environment (agency) inside every node and mechanisms to support agent communication. We agree with other authors [12,2,1] that the paradigm of mobile agents (which has been used in conventional distributed systems since 1996 [13]) can provide great benefits in the context of WSNs. In particular, at the application level, mobile agents can be used as design and programming abstractions through which WSN applications can be effectively designed and implemented.

A key issue when designing WSN mobile agent applications [14] is agent itinerary planning. WSN itinerary planning includes both the selection of the set of WSN nodes to be visited by the mobile agents of these applications, and the determination of the node-visiting sequence in an energy-efficient manner WSN itinerary planning is categorized in [14] as:

Static, where the agent itinerary is totally determined by the sink node or base station before the agent is dispatched.

Dynamic, where the mobile agent autonomously determines the nodes to be visited and the node-visiting sequence, according to the current WSN status.

Hybrid, where the set of nodes to be visited is decided by the sink, and the visiting sequence is determined dynamically by the mobile agent.

Our interest is focused on WSN mobile agent applications, and particularly on those applications developed over middlewares that have been tested on actual nodes. Thus, we conducted a survey of applications running over Agent Framework [8], Agilla [15,1], ActorNet [9], In-Motes [10], WISEMAN [16,11], and MAPS [2].

Among these analyzed middlewares, only WISEMAN provides migration methodologies to support static, dynamic, and hybrid agent itinerary planning. Itineraries in all the other middlewares

* Corresponding author. Tel.: +34 935813577.
E-mail addresses: emercadal@deic.uab.cat (E. Mercadal), cvidueira@deic.uab.cat (C. Vidueira), cjs@cs.ucc.ie (C.J. Sreenan), jborrell@deic.uab.cat (J. Borrell).

0140-3664/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.comcom.2012.09.017

are either static or merely unplanned (arbitrary). Nevertheless, despite this difference in the supported category of planning, in all these analyzed middlewares itinerary specification is included inside the mobile agent code. In all middlewares there is at least one migration operator (# in WISEMAN, *smove* in Agilla, etc.), which receives the nodes of the itinerary through its arguments.

The same situation (itinerary included in the code of the agents) was common in the first mobile agents systems in conventional distributed environments, e.g. Telescript [13]. In such systems, without extensive analysis of the code composing an agent and some knowledge of what runtime conditions were like, it was difficult to predict the behavior of that agent. A solution was provided by Concordia system [17] by separating agent's itinerary from agent's code. A *separate (Concordia) itinerary* is a completely separate data structure from the agent itself, providing a simple mechanism to define and track how an agent travels. See Section 2 for a description of how flexibility of separate itineraries (their ability to be modified at runtime) has increased from the original proposal.

The goal of this paper is twofold. First, to adapt separate itineraries to a highly resource-constrained environment, i.e. Agilla agents on TelosB nodes [18], to improve the dynamism of the migration methodologies of Agilla applications, similarly to what happens in WISEMAN applications. Second, to show how separate itineraries allows us to get a new Agilla application in which agents move globally in a WSN while deciding the node-visiting sequence from these itineraries (in a form of hybrid planning [14]).

This application is used to bring dynamism to one of our previous JADE [19] mobile agent applications of classification of victims in emergency scenarios (MAETT: Mobile Agent Electronic Triage Tag). The general description of MAETT can be found in [20], and the overall architecture of Dynamic MAETT is described in [21]. A summary of all this background can be found in Section 3.

In Section 4 we analyze whether our proposal of including separate itineraries in the Agilla middleware is both feasible and useful. We have to adapt conventional separate itineraries to the new environment, finding room to store the itineraries, and simplifying them to the capabilities of the middleware. This adaptation is then used in our agent-based application Dynamic MAETT. Regarding usefulness, we also shown in Section 4 that our application can easily have a reactive fault-tolerant behavior just programming its agents according to a hybrid planning. To this end, our agents autonomously determine their node-visiting sequence in order to (a) cope with network partitioning, and (b) skip failed or removed nodes, and thus minimizing the probability of all of them being caught inside a failing node.

The experimental evaluation of our proposal can be found in Section 5, including TOSSIM/TinyViz [22] simulations, tests on a testbed with up to 27 real TelosB nodes, and a field trial of our application in a scenario that closely resembles a mass casualty incident (MCI) situation. Energy consumption of our application is also analyzed and found to be in line with other similar applications in MCI scenarios, e.g., [23]. From this evaluation we can conclude that the adaptation of separate itineraries to Agilla mobile agents is both feasible (in space and time), and useful to improve the reactivity of our application to failing nodes and partitioned networks

2. Related work

2.1. WSN mobile agents: Middlewares and applications

As mentioned above, we are interested in those middlewares that have been developed and tested on actual nodes, and specifically interested in their applications. For each analyzed middleware we enumerate below the platforms on which has been developed, the provided programming language for applications, the mechanisms for communication and coordination among its agents, and a list of its developed applications.

Agent Framework[8]	
Platforms:	Mica2dot
Prog. language:	Maté [24] TinyScript
Coordination:	Shared memory, network messages
Applications:	Global data collection
	Gradient search
	Event tracking
ActorNet[9]	
Platforms:	Mica2
Prog. language:	High level functional-oriented
Coordination:	Shared memory, network messages
Applications:	Gradient search
In-Motes[10]	
Platforms:	Mica2dot
Prog. language:	Micro-programming
Coordination:	Tuple spaces, agent facilitators
Applications:	Data gathering
WISEMAN[16,11]	
Platforms:	MicaZ
Prog. language:	Text-based codes
Coordination:	Local (node) variables
Applications:	Early forest fire detection
Agilla[15,1]	
Platforms:	TelosB, Mica2,
	MicaZ and Tyndall 25 mm
Prog. language:	Micro-programming
Coordination:	Tuple spaces [25]
Applications:	Fire detection and tracking
	Monitoring cargo containers
	Navigation in a dynamic environment
MAPS[2]	
Platforms:	Sun SPOT
Prog. language:	Java
Coordination:	Network messages
Applications:	Remote sensor monitoring

Recently, Agilla agents are also supported by the Servilla platform [26]. Servilla has been developed for heterogeneous nodes (from Telosb to imote2) and provides different services depending on the node.

2.2. Mobile agent itineraries

Although the itinerary concept is the same in all mobile agents systems (route followed during mobile agent migration), there is a significant difference when using this concept in conventional distributed environments or in WSNs. In conventional environments, itinerary planning is basically related to the description and control of the migration (behavior) of the agents. On the other hand, in WSNs, itinerary planning is also a way to minimize the energy consumption of the nodes, by computing the optimal route used by agents when traversing the WSN.

The current form of itineraries in conventional environments [27–29] was initially proposed by Concordia [17], as a separate data structure to hold the information of the locations to be visited by the agent. Apart from migration instructions, agent's itineraries include specific code and data for each visited host. First separate itineraries were sequential, in the sense that all platforms were visited one after the other, in the order initially specified by the programmer. To allow the programmer to define alternative routes, flexible itineraries were introduced in [30]. Flexible separate itineraries are composed of different types of recursive entries allowing agents to make decisions about their travel plan at runtime. As an example, three entry types were defined in [30]: the *sequence*, where the agent has only one possible destination after the current platform, so no routing decision needs to be made; the *alternative*, where the agent can choose its next destination from a predefined set of platforms; and the *set*, where the agent has to visit all the platforms of a predefined set in any order.

As a new step in the evolution of conventional mobile agent itineraries, [31] presented a proposal to define and protect separate itineraries for *free-roaming agents*, which involves discovering the location of one or more destination platforms at runtime.

Regarding WSN mobile agents itineraries, optimization of energy consumption in their planning is paramount. As the problem of finding optimal itineraries in WSNs is NP-hard [32], a lot of research has been devoted to this problem, surveyed in [33]. Different heuristics have been proposed, from the simplest ones in [34], based on genetic algorithms in [35] or [36], to the more elaborated ones in [33]. Multiple mobile agents' itinerary planning is also considered in recent proposals [37,33] to allow the scalability of the solutions to large WSNs.

As stated in Section 3, the limited size of the WSN in our intended application downplays the effect of optimizing the energy consumption in the algorithm used to compute the initial itineraries of our agents. On the other hand, our separate itineraries are used to increase the adaptability of our application, as they facilitate changes in the node-visiting sequence of our agents, in a migration methodology that supports hybrid planning [14].

3. Background

In this section we summarize the description of our previous JADE based MAETT application. A brief description of the Agilla WSN mobile agent middleware and of the architectural components of Dynamic MAETT is also included.

3.1. Mobile Agent Electronic Triage Tag (MAETT)

MAETT (Mobile Agent Electronic Triage Tag) [20] is a system providing early resource allocation during emergencies when no network infrastructure is available.

The foundation of the system is mobile agent technology [28], which allows information to be directly transported from terminal to neighboring terminal regardless of the status of the rest of the network at that particular time. Handheld devices run an execution environment for JADE agents, the platform, where mobile agents can be created, executed and forwarded to other terminals. Are the agents themselves who decide the route to follow depending upon the available information on the neighbors.

The main actors of the system are the victims, the triage personnel or first responders, and the rescue teams (see scenario in Fig. 1, where colored smileys represent JADE agents). Assuming victims are scattered over an arbitrarily large area of emergency. The triage personnel scour all this area looking for victims and triage them according to standard methods. The result of this triage is written on a physical tag and placed visibly on the victim. Finally, the rescue teams collect all the victims, prioritizing depending on triage results. The Emergency Coordination Center (ECC) coordinates all actions. Triage personnel, rescue teams and the ECC wear handheld wireless devices with a JADE mobile agent platform and a GPS receiver.

Triage personnel leave the ECC, and have an estimation on when they will get back (Time To Return – TTR). When a victim

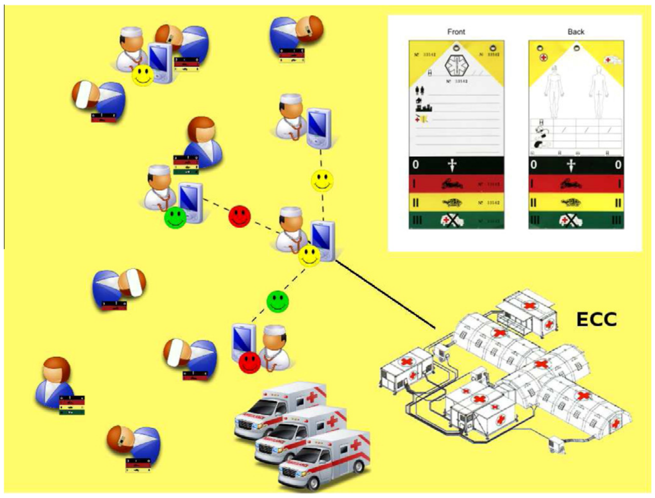


Fig. 1. MAETT triaging scenario showing the classic cardboard triage tags.

is found, they use the standard START method [38] and place a cardboard triage tag (see Fig. 1) with an integrated RFID on the neck of the victim with their evaluation written on it. At the same time, an agent is created containing the information in the tag, plus the GPS position of the victim and the RFID of the tag. All this information will be used later in the ECC to optimize the route of the rescue teams. This agent is transmitted to neighbor devices only if the bearer has a lesser TTR. This is to make sure that moving the information is never going to make it arrive later. Consequently, all handheld devices carried by triage personnel are used to create agents with information about found victims, and also to forward agents corresponding to other victims. When agents arrive to the ECC, the ECC sends the rescue teams with a detailed schedule of the route based on the GPS position of victims as well as their medical condition.

3.2. Agilla WSN mobile agent middleware

Agilla [1] is a mobile agent middleware designed to support self-adaptive applications in wireless sensor networks. Agilla's model is shown in Fig. 2. Each node supports multiple mobile agents that can move or clone across nodes while carrying their state. To facilitate agent interactions, each node provides two data structures, a neighbor list and a tuple space [25], a type of shared memory accessed via pattern-matching that enables a decoupled style of communication. Agilla also provides specialized reaction primitives that enable agents to efficiently respond to changing state. Prior Agilla versions addressed WSN nodes by their location, though this restriction was removed in version 3.0.

3.3. Dynamic MAETT architecture

The problem with MAETT is that changes in victims' medical conditions, which have a great impact on the subsequent rescue planning, are never conveyed to the ECC. We added dynamism to MAETT by placing a wireless node equipped with medical sensors in every victim, in addition to the triage tag in the MAETT scenario.

Any device able to run Agilla agents, for example those of [39], or [23], or that in Fig. 3, manufactured by Maxfor,[1] can be used to create a WSN among neighboring victims. Agilla agents inside this WSN can be used to dynamically update the medical status of every

[1] http://maxfor.co.kr.

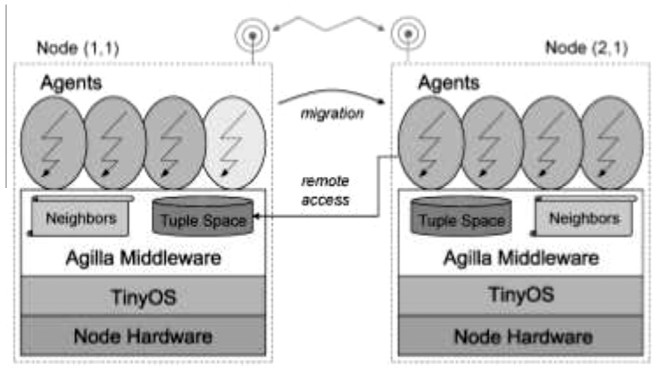


Fig. 2. The Agilla model (from [1]).

Fig. 3. Watch type body monitoring device.

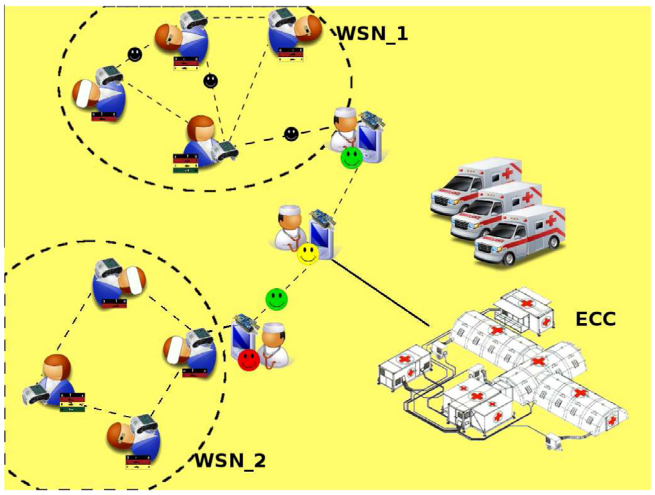


Fig. 4. Dynamic MAETT triaging scenario.

victim and to communicate any significant change to the emergency personnel nearby, should they are in range.

The new scenario for dynamic MAETT can be seen in Fig. 4, where black small smileys represent Agilla agents and colored big smileys represent JADE agents. See [21] for a full description of Dynamic MAETT architecture.

To communicate the WSN and the handheld devices of triage or rescue members we also need these members carry a WSN node (Fig. 5) attached to their handheld device, also a TelosB compatible node from Maxfor.

As every nearby victim is both paper tagged and electronically tagged, neighboring wireless nodes belonging to the same triage team member wirelessly connect creating a growing WSN of victims.

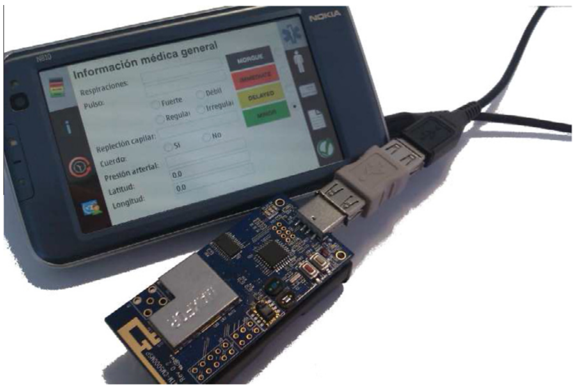


Fig. 5. A WSN node connected to a Nokia N810 handheld.

When every victim in the vicinity is tagged and each body sensor node monitors its own victim, the triage team member ends the creation of his WSN. The number of nodes of each WSN is limited to lighten the medical personnel bags. Notice that every health monitoring sensor weights about 70 g including the required two 1.5 V AA batteries. Assuming that the triaging personnel carries the handheld device and the paper tags, they may agree to carry the extra weight of 20–25 sensor nodes, that is, the weight of a small laptop (i.e. ~1.5 Kg).

With this limited number of nodes, and thanks to the fact that the topology of the WSN is known (the handheld device records the GPS position of every tagged victim), the handheld device itself can easily compute a path through the newly created WSN by using an optimized version of the Depth-First-Search (DFS) algorithm [40,41]. The DFS computed path tries to minimize the number of visited nodes and the radio transmission. See [41] for the complete analysis of the results of DFS execution in our handheld device. Any other algorithm for itinerary planning [33] could be used to compute this path, albeit the gain in energy efficiency would be small due to the reduced size of our WSN.

The architecture presented above is the platform where our system of mobile agents with separate itineraries is developed. In the following section this development and its inclusion in a specific application (Dynamic MAETT) is described.

4. Separate itineraries in Dynamic MAETT application

In this section, the core of our paper, the adaptation of separate itineraries to Agilla middleware on TelosB WSN nodes is shown. This adaptation has to consider memory and programming limitations in this highly resource-constrained environment. This adaptation is then used in our agent-based application Dynamic MAETT. First of all, an overview of the application is provided, to subsequently enter in its details, describing how the application is designed, with all its composing parts, i.e., the Agilla agents, the use of separate itineraries in its mobile agents, and the memory requirements and limits for the application. We conclude the section showing how separate itineraries allow to improve the fault tolerance reactive behavior of our application.

4.1. Separate itineraries for Agilla mobile agents

Computationally restricted mobile agent technologies, i.e., those working on low power devices, e.g. WSN nodes, have not yet benefited from the advantages separate itineraries can provide. These devices can also make the most of itineraries, opening new possibilities on WSN applications.

In order to develop separate itineraries for Agilla mobile agents we had to consider two issues:

1. Where to store this separate itinerary since the memory of the nodes is very limited.
2. How to adapt the itineraries of conventional agents to the limited execution environment of the nodes.

In regard to memory, Agilla provides three different storage constructions:

The tuple space is a shared memory space where data is structured as tuples that are accessed via pattern-matching. In Agilla it is primarily used for communication between agents, either coexisting in the same node or not.
The stack is a common data structure which works as a LIFO queue and provides only two operations `push` and `pop`. It is fundamentally used to store application runtime variables and instruction return values. By default it can store up to 105 bytes.
The heap is a random-access storage area that allows each agent to store (by default) 20 variables of 16 bits each. We can access any position of the heap with the `setvar` and `getvar` Agilla instructions, previously pushing the desired position of the heap onto the stack.

It is this third data structure, with its random access method and its storage size, the most suitable place to store the itinerary. See Section 4.5.1 for a more detailed analysis of memory limits and maximum itinerary length inside Agilla.

As for the adaptation of the itineraries, we made two simplifications regarding those of conventional agents. The first one is to consider only entries of type *sequence* and *alternative*. *Set* type entries should be implemented cloning an agent a number of times, followed by gathering the results of cloned agents. This is a complex issue that must be studied in more detail. Thus, a WSN mobile agent itinerary will be a sequence of node identifiers stored in the heap. The agent will move to the next node in the itinerary, and in case the next node being unreachable, or if the agent code decides otherwise, the agent will move to an alternative node.

The second simplification is to consider that the same code will be run on each node. If the code had to be different on each node it should be stored on the heap, and the nodes should have a capacity similar to Java reflection,[2] a mechanism out of place inside those nodes. However, despite having the same code on all nodes, it is easy to adapt this code to specific nodes, e.g. those at the ends of the itinerary.

4.2. Dynamic MAETT: An overview

In the first run of the first responders, every victim receives the classic triage tag from the triaging method START [38], and it is also equipped with a wireless body monitoring sensor node with an Agilla agent (*Victim* agent). This agent periodically reads victim's vital signs from the sensor device and runs a triaging algorithm on them, maintaining an up-to-date state of the health condition of the victim. These readings and behavior are simulated in the current version of our application as we are more interested in agent cooperation than in the actual sensing. We recognize that developing a fully functional version of the *Victim* agent is far from trivial. This agent has to monitor and process vital signs and this is a complex task, as [23,39] point out. Moreover, according to the study of memory requirements that we perform in Section 4.5.1 (biggest agent code), the *Victim* agent should be encoded in a maximum of 5808 bytes. We are currently porting Agilla to TinyOS v.2 as a prior step to address this development.

[2] http://docs.oracle.com/javase/tutorial/reflect/.

When every victim in the vicinity is tagged and each body sensor node monitors its own victim, the triage team member ends the creation of his WSN. Moreover, the first responder handheld computes a DFS path [41] that can be used as the itinerary of another Agilla agent (*Traveler* agent), this time being mobile, to permanently travel around the WSN. The unique purpose of the *Traveler* agent is to collect, if needed, the changes in victims' condition, and distributing these changes to the other WSN nodes.

Later on, when a new rescue or triage team member approaches the WSN, and its handheld device contacts a node in the WSN, the node attached to the handheld device uses an *Extractor* agent to get the aggregated changes of all the victims' status form the *Victim* agent. As every node of the WSN has the most up-to-date state as possible, any node of the WSN can indistinctly provide this information. The handheld device then creates a JADE mobile agent containing the new status of the victims, and routes it to the ECC, as was done in MAETT.

4.3. Dynamic MAETT: Agent-based design

Our application is designed around three types of agents: the *Victim* agents, the *Traveler* agents, and the *Extractor* agents. A *Victim* agent resides permanently into each node of the WSN, periodically reading victim's node sensors and uses those readings to compute the general health condition of the victim. To save batteries it only gets the sensors readings and computes the summary when a *Traveler* agent is in the same node, thus sleeping the rest of the time.

A *Traveler* agent is that in charge of going over every node of the WSN collecting, aggregating and sharing victim's health status calculated by *Victim* agents. A *Traveler* agent uses the itinerary computed beforehand using the first responder handheld device, which contains a valid route through the WSN. The agent jumps from node to node reading the general health condition of the victim previously calculated by the *Victim* agent. A *Traveler* agent also writes the changes found in the previous nodes, thus maintaining an up-to-date log of the whole WSN in every node. Note that a *Traveler* agent only updates the already visited nodes of the ongoing trip, it does not update node information from previous runs.

Finally, an *Extractor* agent is that residing into a first-responder handheld, responsible for the communication and information exchange with the victim's WSN when connected.

The communication between agents is done by means of Agilla reactions, a method provided by the middleware itself that makes the agent respond to the presence of a specific tuple in the tuple space, preventing the agent from doing continuous polls. See Fig. 6 for an example of reaction code. Reactions consist of a template, a label to the callback function, and a block of code. To register a reaction with the `regrxn` instruction, we first need to specify when the reaction should fire, i.e., we need to specify the form of the tuple. This is done by pushing the template into the agent's stack (`pusht VALUE`) and specifying its number of fields (`pushc 1`). Then we have to state the name of the callback function, i.e., indicate where to jump (`pusch DORXN`).

The callback function is then executed and finishes its execution with the `endrxn` instruction, which returns the control to the main program. In the sample code the callback function pops the tuple values from the stack (`2*pop`) and lights the red led (`pushc 25` & `putled`).

In our application, when the *Traveler* mobile agent arrives to a node it inserts a tuple in the node tuple space to notify of its arrival and to instruct the *Victim* agent to start reading the sensors and dump the results back to the *Traveler* agent. Afterwards, the *Traveler* agent inserts another tuple with the information of the rest of the WSN, to which the *Victim* agent responds with a `FIN` signal when it has stored all the information. Fig. 7 depicts the communication between *Traveler* and *Victim* agents illustrating how reactions work.

```
pusht VALUE          DORXN    pop
pushc 1                       pop
pushc DORXN                   pushc 25
regrxn                        putled
                              endrxn
```

Fig. 6. Sample reaction code.

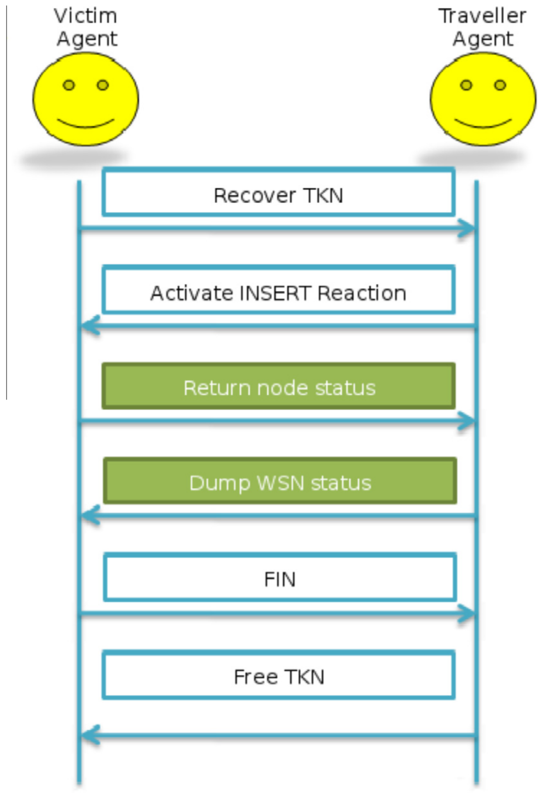


Fig. 7. Communication between Traveler and Victim agents using reactions.

The extraction of the collected data is started by the third type of agent of our application, the *Extractor*. This agent starts its operation when gets in contact with any node of any WSN. The *Extractor* agent initiates the communication with the residing *Victim* agent of a given node by pushing a tuple into the remote node's tuple space. The victim agent then dumps the whole WSN status to the *Extractor* agent tuple space. When the *Extractor* has read the dumped tuple it responds with a `FIN` signal, which releases the communication between both agents. Fig. 8 pictures this recollection of WSN's status by the *Extractor* agent.

4.4. Separate itineraries in Dynamic MAETT

Regarding our application, we designed and implemented an itinerary construction for our WSN nodes using as little as 8 bits per node, including the node identification (5 bits), an on/off bit and information about the state of the monitored victim (2 bits). It is worth noting that the itinerary is not computed to be circular, thus, to continuously move through the WSN, the *Traveler* agent has to traverse the itinerary forward and backward. With our construction and taking advantage of the random access of the heap, this can be done without having to reorder the carefully coded itinerary.

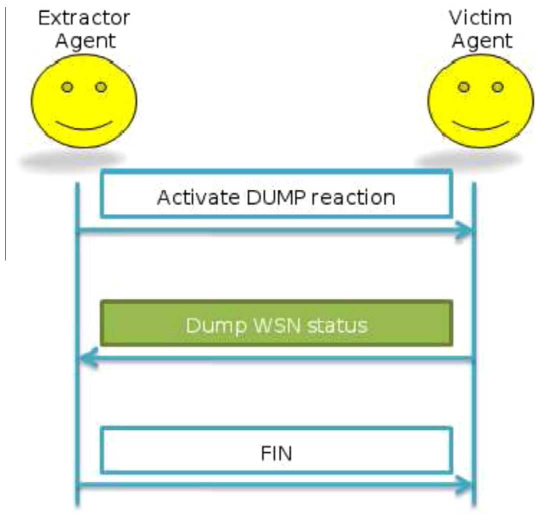


Fig. 8. Communication between Extractor and Victim agents.

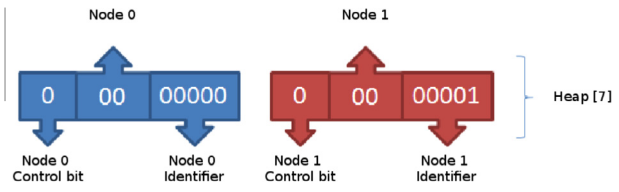


Fig. 9. A position of the heap containing two itinerary entries.

The itinerary is then loaded into the Agilla mobile agent and is injected into the WSN, starting its route through the sensor nodes. For the itinerary to fit in just 8 bits we had to take some important technical decisions, which are due to the maximum amount of sensor nodes first responders carry and thus, the maximum size of the WSN.

Our application uses the 8 bits of each itinerary position as follows (Fig. 9): 1 bit to determine if the node is active; 2 bits to store a summary of the sensor readings; and 5 bits for the node identifier.

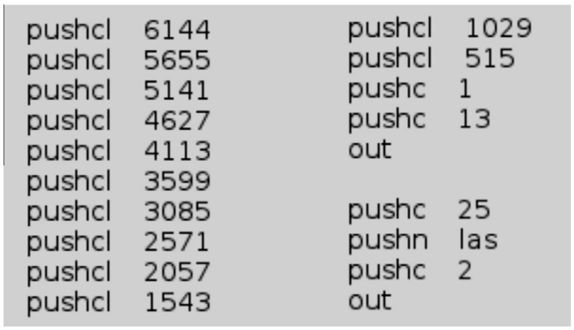
Using this codification we can easily fit the maximum amount of nodes we defined for our architecture (25) in just 13 positions of the heap, and we still have room for some other application-related purposes.

A sample coded itinerary of 24 nodes can be seen in Fig. 10, where each number in the first 13 instruction represents two nodes of the itinerary, that is loaded into the nodes' tuple space, staying there until is retrieved by the *Traveler* agent.

Moreover, using 8 bits to define every step of the itinerary we can fit two of this steps using just one position of the stack. Unfortunately, this complicates the reading of an itinerary step from the heap, forcing the division of this value and choosing the required step after pushing it onto the stack. This division is done by shifting the value to the right if the eight most significant bits are required.

4.5. Memory requirements

With our specifications the memory requirements of our application are very small, leaving room in the WSN node for a broad range of improvements. A basic Agilla installation of our application in a TelosB node takes up 3866 bytes out of 10 kB of RAM and 45308 default bytes out of 48 kB of ROM, increasing just 400 bytes of RAM from the Agilla installation. As for the changes done



```
pushcl  6144        pushcl  1029
pushcl  5655        pushcl  515
pushcl  5141        pushc   1
pushcl  4627        pushc   13
pushcl  4113        out
pushcl  3599
pushcl  3085        pushc   25
pushcl  2571        pushn   las
pushcl  2057        pushc   2
pushcl  1543        out
```

Fig. 10. A sample coded itinerary.

to the default Agilla installation to support our application the most important are contained in:

- $AGILLA/nesc/agilla/Makefile.Agilla
  - l. 1 -DAGILLA_NUM_AGENTS from 3 to 6
  - l. 2 -DAGILLA_NUM_CODE_BLOCKS from 12 to 60
  - l. 14 -DAGILLA_MAX_NUM_NEIGHBORS from 20 to 25
- $AGILLA/nesc/agilla/types/TupleSpace.h
  - l. 45 AGILLA_MAX_TUPLE_SIZE from 20 to 48
- $TOSDIR/types/AM.h
  - l. 65 #define TOSH_DATA_LENGTH from 29 to 36

These changes increase the maximum number of agents supported by a node from 3 to 6, the maximum memory used by agent's code from 12 to 60 code blocks (1 code block equals 22 bytes), the maximum number of neighbors per node from 20 to 25, the maximum size of a tuple from 20 to 48 bytes and the length of a TinyOS message from 29 to 36 bytes.

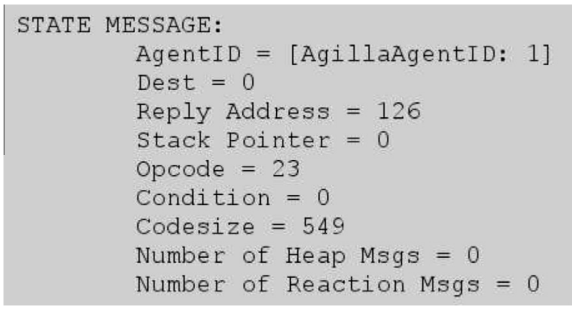
Agent's code size, in bytes, is shown when the agent is injected into a node (Fig. 11).

The injected agent is stored into the node's reserved RAM, occupying the memory needed by the code (26 blocks or 572 bytes) plus the stack, the heap and the agent registers (248 bytes).

4.5.1. Memory limits

We can use the remaining memory in the node for various purposes, either maximizing the code of the agent, maximizing the length of the itinerary, or maximizing the number of agents inside a WSN node.

The size of the code of an Agilla agent, stored in the RAM of the WSN node, is determined by the number of code blocks it occupies. We know that every basic Agilla instruction needs 1 byte, shared by the instruction identifier and the parameter. Moreover, Agilla supports an extended ISA, which needs two bytes for each instruction to operate. Adding the size of all the instructions of the agent's code we can compute the number of code blocks required for the



```
STATE MESSAGE:
        AgentID = [AgillaAgentID: 1]
        Dest = 0
        Reply Address = 126
        Stack Pointer = 0
        Opcode = 23
        Condition = 0
        Codesize = 549
        Number of Heap Msgs = 0
        Number of Reaction Msgs = 0
```

Fig. 11. Injection messages.

agent. This amount of memory is reserved in the node when programmed with Agilla and before injecting any agent. This will determine the maximum total size of the agents the node will accept.

Knowing that the size of an agent in Agilla is determined not only by the length of its code, but also by its stack, its heap and the tuple space, we found the maximum memory used by the mentioned three possible configurations:

4.5.1.1. Maximum itinerary length.

We calculated the maximum of addressable nodes using just one *Traveler* agent, i.e, setting the AGILLA_NUM_AGENTS variable to 2, the AGILLA_NUM_CODE_BLOCKS to 35 (26 for the Traveler and 9 for the Victim). This two variables can be found in lines 1 and 2 in

$AGILLA/nesc/agilla/Makefile.Agilla.

To increase the number of addressable nodes we have to increase the heap size (AGILLA_HEAP_SIZE in.

$AGILLA/nesc/agilla/types/Agilla.h), so all the itinerary fits in it. It is also advised to increase the AGILLA_MAX_NUM_NEIGHBORS variable, found in line 14 of $AGILLA/nesc/agilla/Makefile.Agilla to reflect these changes, which will allow the agents to identify a higher number of neighbors.

We found that the maximum length of the itinerary is limited by the maximum 551 positions of the heap, filling 10238 bytes of RAM (10kB), and allowing to address more than 2^9 nodes.

4.5.1.2. Biggest agent code.

The largest code an agent can have in a similar application is determined by the maximum number of code blocks that fit in the memory of the node. In our case we are limited by the 10kB of the TelosB WSN nodes.

We found that all the 10kB of RAM memory becomes full with an agent of 290 code blocks, i.e. 6380 bytes, using just one *Traveler* and one *Victim*, keeping the value of 248 bytes for stack, heap and tuple space.

4.5.1.3. Maximum number of agents.

The maximum amount of RAM of the TelosB nodes is reached with 9 agents: 8 *Travelers* and 1 *Victim*, also keeping the value of 248 bytes for stack, heap and tuple space. This 9 agents took 217 code blocks and a total of 10152 bytes of RAM.

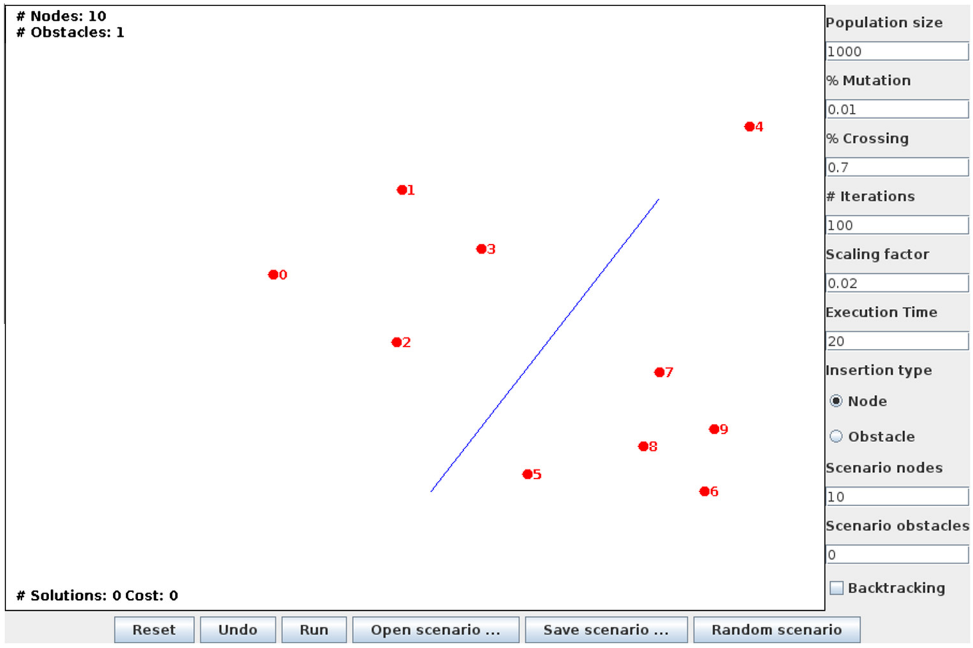
This is maybe the most interesting configuration for a triaging application. With 8 *Traveler* agents the fault tolerance of the application can be heavily improved using them with different itineraries or monitoring different sections of the WSN.

4.6. Fault tolerance

Fault tolerant mechanisms for our application are two-fold, and both based on separate itineraries. The first mechanism consists in moving to alternative entries in these itineraries in order to skip failed or removed nodes. The second mechanism consists in using several *Traveler* agents with different itineraries, to minimize the probability of all agents being caught inside a failing node, and to cope with network partitioning.

Regarding the first mechanism, when a *Traveler* agent finds that the next node in its itinerary is not available, either because it is failing or because it was attached to a removed victim, it has three different methods to move to alternative nodes. In the first implemented method, **jump-after-next**, the agent looks forward in the itinerary structure and tries to migrate to the following node if it is available and in range. If it is not, the *Traveler* agent decides by a simple coin-flipping to choose the second method (**random-jump**) or the third one (**reverse-itinerary**).

In the **random-jump** method, the *Traveler* agent gets the list of available neighbor nodes and randomly chooses one of them for its migration. After any of these migrations to alternative nodes the *Traveler* agent resumes its sequential route from the new node.


Nodes: 10
Obstacles: 1
Solutions: 0 Cost: 0

Population size: 1000
% Mutation: 0.01
% Crossing: 0.7
Iterations: 100
Scaling factor: 0.02
Execution Time: 20
Insertion type: [✓ Node] Obstacle
Scenario nodes: 10
Scenario obstacles: 0
Backtracking: ☐

Fig. 12. Critical node (No. 4) in a WSN configuration.

In the **reverse-itinerary** method the trip through the itinerary is simply reversed, just as it is done when arriving to the end node. Note that the third method is a useful strategy because **jump-after-next** tries always to go forward in the itinerary, and **random-jump** also goes forward from a new random node, and thus some backward nodes can be left unvisited in case of network partitioning.

As for the second fault tolerant mechanism, our application can use several (up to 8, according Section 4.5.1) *Traveler* mobile agents, where each of them starts its itinerary from a different node of the network and, preferably, follows a completely different itinerary. This adds another layer to the tolerance against failures of the application, increasing the robustness of the WSN in terms of partitioning. Now it is more unlikely to end up with a part of the WSN not being monitored due to a permanent failure of one, or some, critical nodes.

The alternative itineraries followed by the supplementary *Traveler* agents are also calculated in the first responders handheld, where we have all the information about the victim's location. In the handheld, using DFS and choosing different starting nodes, we can obtain different, but similarly effective, itineraries for all the mobile agents. To prevent first responders of having to move to several victims to inject the additional *Traveler* agents, we use the same strategy as in the **random-jump** fault tolerance method: each mobile agent looks for the actual node in the itinerary structure and follows its route from there.

The drawback of using more than one *Traveler* agent is that the integrity of the data is not guaranteed, i.e., an agent may update more recent data written by another *Traveler* agent. In our application, where every *Traveler* agent is updating victim's information at most every 80 s (see experimental roundtrip times in Section 5) this is not a critical issue.

A situation that we had to take into account is the meeting of two, or more, *Traveler* agents into the same sensor node. If this node fails then all *Travelers* inside will be lost. This is specifically serious if the node is the only link between two clouds of nodes, as in Fig. 12 (a snapshot of a GUI used to show DFS computed itineraries in [41]). To prevent this accumulation of mobile agents we use a token-like solution: a special tuple is stored in the node's tuple space. Then, when a *Traveler* agent wants to migrate to a new node, it looks for this tuple in the destination's node tuple space. If it is available, takes it and continues its normal execution. If another *Traveler* agent has already taken the tuple, the new agent tries to move to another node using one of the alternative methods detailed above.

Code of our application Dynamic MAETT can be downloaded from https://senda.uab.cat/wiki/dMAETT.

5. Experimental evaluation

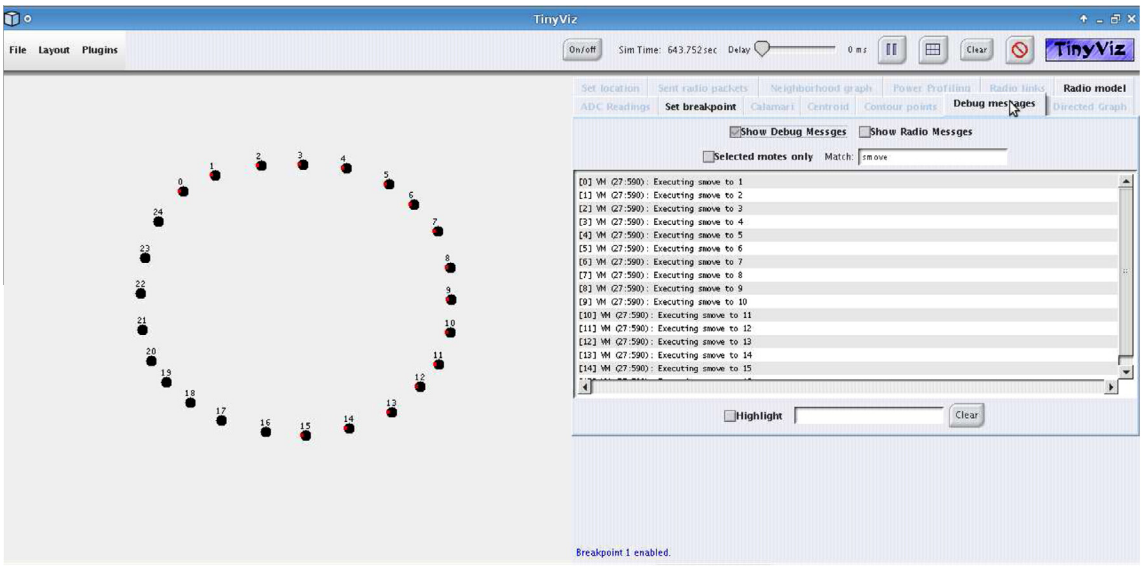
Experimental evaluation of our proposal of separate itineraries built in our Dynamic MAETT application for mass casualty incidents (MCI) includes TOSSIM/TinyViz simulations, tests on a testbed with up to 27 real TelosB nodes, and a field trial of our application in a scenario that closely resembles a MCI situation.

The goal of the evaluation is, apart from debugging and checking the correct working of every part of our application, to check the correct behavior of all the fault-tolerant mechanisms included in our separate itineraries, and to measure the migration times of our *Traveler* agents in different WSN configurations.

We used simulations first to debug the application using TOSSIM and TinyViz, the simulator and visualization GUI for TinyOS. Debugging using this tools is not straightforward but excruciating, since some of the messages provided by Agilla are not informative at all, being the most common `INVALID_TYPE` and `INVALID_SENSOR`.

Having checked the correct working of every part of our application we build a testbed with up to 27 TelosB nodes (Maxfor MTM-CM5000-MSP and MTM-CM4000-MSP) with Agilla v.3.1.1 over TinyOS, and ran our application there to check that the results obtained with the simulations were valid using real sensor nodes. This move forced us to make some important changes to our design, such as moving from reactions to active waiting in the *Traveler* agent, due to problems during their transmission, and reducing the size of the TinyOS messages.

After verifying that real sensor nodes performed as its simulated counterparts we moved our application to a real deployment where, with the help of some colleagues, we tested our application



Match: smove

```
[0] VM (27:590): Executing smove to 1
[1] VM (27:590): Executing smove to 2
[2] VM (27:590): Executing smove to 3
[3] VM (27:590): Executing smove to 4
[4] VM (27:590): Executing smove to 5
[5] VM (27:590): Executing smove to 6
[6] VM (27:590): Executing smove to 7
[7] VM (27:590): Executing smove to 8
[8] VM (27:590): Executing smove to 9
[9] VM (27:590): Executing smove to 10
[10] VM (27:590): Executing smove to 11
[11] VM (27:590): Executing smove to 12
[12] VM (27:590): Executing smove to 13
[13] VM (27:590): Executing smove to 14
[14] VM (27:590): Executing smove to 15
```

Breakpoint 1 enabled.

Fig. 13. TOSSIM simulation of a circular WSN.

with 15 nodes in a scenario which closely resembles a mass casualty incident (MCI) situation.

5.1. Simulations

The first simulation done with TOSSIM and TinyViz was done using a circular topology (Fig. 13) of 10, 20 and 25 nodes, with none of them failed. There we made the first tests to our application, solving little programming issues, problems with the stack and with the jumps when activating the reactions, due to the size of the code.

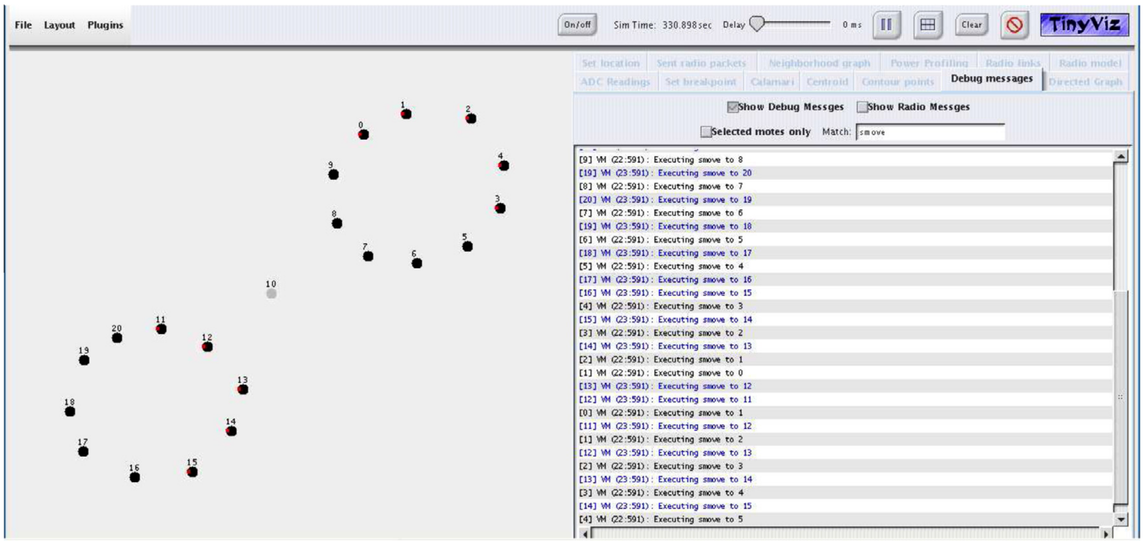
After verifying the correct working of the application without failed nodes we moved a step forward and tested our application against a more complicated scenario. There we used the same circular topology but forced the failure of some randomly chosen nodes. Fault tolerance methods behaved as expected, avoiding the problems of not finding the expected node and having to move to a different one, either jumping to the subsequent node of the itinerary or randomly jumping to a known neighbor. These methods also proved its correctness when the *Traveler* agent was able to continue operating after an alternative migration.

Finally we simulated our application with two *Traveler* agents where, in a partitioned WSN (Fig. 14), we tested its correct adaptation to a failure of an important node, a node that is the only link between two sections of the WSN. Both *Traveler* agents remained in its partition of the WSN, visiting the nodes either following the itinerary or using fault tolerance when not possible.

Simulations were used just to test the correct working of the application. Then we moved to a testbed of nodes to check its actual behavior.

5.2. Testbed runs

When moving the application to our testbed (Fig. 15), the first test consisted in verifying the application to work as in the simulations. First with all the nodes working and fully functional. Here we perceived the problem with the *Traveler* agent and its reactions. After running for an undetermined time, reactions in



Match: smove

```
[9] VM (22:591): Executing smove to 8
[19] VM (23:591): Executing smove to 20
[8] VM (22:591): Executing smove to 7
[20] VM (23:591): Executing smove to 19
[7] VM (22:591): Executing smove to 6
[19] VM (23:591): Executing smove to 18
[6] VM (22:591): Executing smove to 5
[18] VM (23:591): Executing smove to 17
[5] VM (22:591): Executing smove to 4
[17] VM (23:591): Executing smove to 16
[16] VM (23:591): Executing smove to 15
[4] VM (22:591): Executing smove to 3
[15] VM (23:591): Executing smove to 14
[3] VM (22:591): Executing smove to 2
[14] VM (23:591): Executing smove to 13
[2] VM (22:591): Executing smove to 1
[1] VM (22:591): Executing smove to 0
[13] VM (23:591): Executing smove to 12
[12] VM (23:591): Executing smove to 11
[0] VM (22:591): Executing smove to 1
[11] VM (23:591): Executing smove to 12
[1] VM (22:591): Executing smove to 2
[12] VM (23:591): Executing smove to 13
[2] VM (22:591): Executing smove to 3
[13] VM (23:591): Executing smove to 14
[3] VM (22:591): Executing smove to 4
[14] VM (23:591): Executing smove to 15
[4] VM (22:591): Executing smove to 5
```

Fig. 14. TOSSIM simulation of a partitioned WSN.

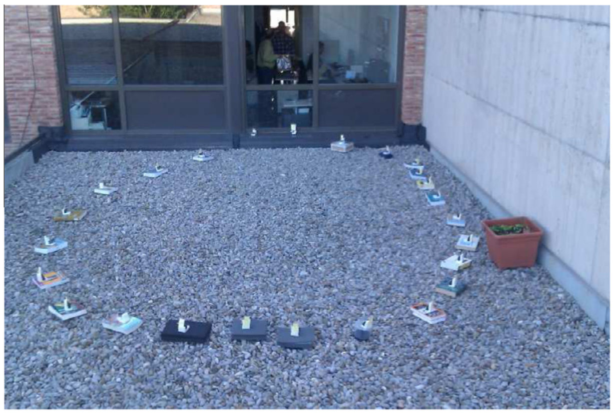


Fig. 15. Testbed scenario as a circular topology.

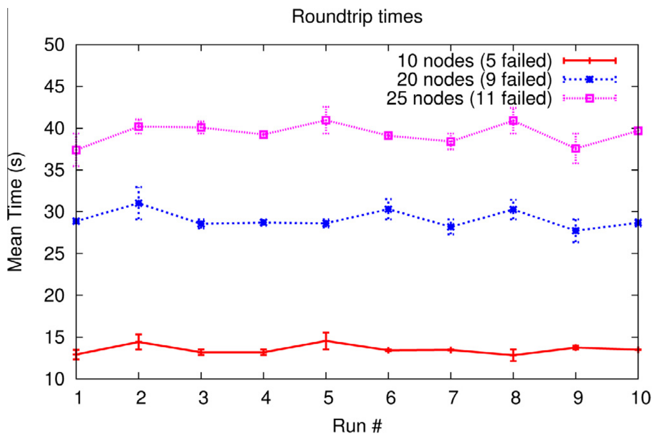


Fig. 17. Mean lap times after 10 runs with failed nodes.

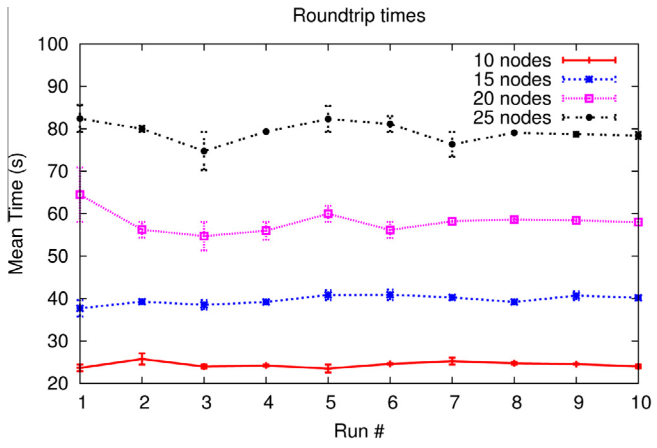


Fig. 16. Mean lap times after 10 runs.

the *Traveler* agent stopped transmitting with its bearer, resulting in the complete stall of the application.

To solve this issue we had to move the communications done by these reactions to an active polling method, where the agent looks for the expected data in the tuple space. With the reaction problem circumvented, we measured the amount of time the *Traveler* agent needs to visit each and every node of the network.

Fig. 16 depicts the running times of tests conducted on 10, 15, 20 and 25 nodes WSN. The graph samples are the mean of 5 runs of 10 roundtrips each, errorbars show the deviation of the data in each roundtrip sample. Computing the mean of the tests we have that our *Traveler* agent takes around 24.4 s to perform a full roundtrip of the 10 nodes WSN, that is about 1.2 s per node. In the 15 nodes WSN takes around 43.8 s, that is 1.46 s per node. In the 20 nodes WSN around 58.1, 1.45 s per node, and in the 25 nodes WSN around 79.3 s, that is 1.59 s per node.

From the data in Fig. 16 is worth noting that the time spent in each node and doing the migration increases as the number of nodes increase. This is due to the bigger amount of data the agent has to carry when the WSN enlarges, thus increasing the amount of data computed and transferred for each node.

After having measured the behavior of our application in a testbed with all the nodes operational, we tested our agents against a problematic WSN. First we benchmarked it against a WSN with some deactivated nodes, 5 in the case of the 10 nodes WSN, 9 for the 20 nodes one, and 11 for the 25 nodes network. This test was intended to measure the correct working of the **jump-after-next** fault tolerance method, thus never turning off two consecutive nodes.

The results (Fig. 17) showed that our application responds very well to problems with failing nodes, reducing the roundtrip times by nearly the half when compared with the network with all the nodes working. This happens because most of the time is spent doing migrations, thus the less nodes to visit, the less migrations to perform.

The last test we applied to the testbed was focused to measure the response of the application against network partitioning. To perform this test we deployed a network with two separated clouds of 10 nodes each, only connected by a single critical node. We injected two independent *Traveler* mobile agents and after that we turned off the critical node deliberately. Each agent was left in a different network partition, thus being disconnected from the other part of the network. We proved that having two independent *Traveler* agents with disjoint itineraries is a good solution for this kind of situations where the itinerary is not complete in any of the partitions and fault tolerance methods are heavily used. We also used this scenario to measure the time needed to visit every node of a partition, with the *Traveler* agent being forced to use every fault tolerance method implemented.

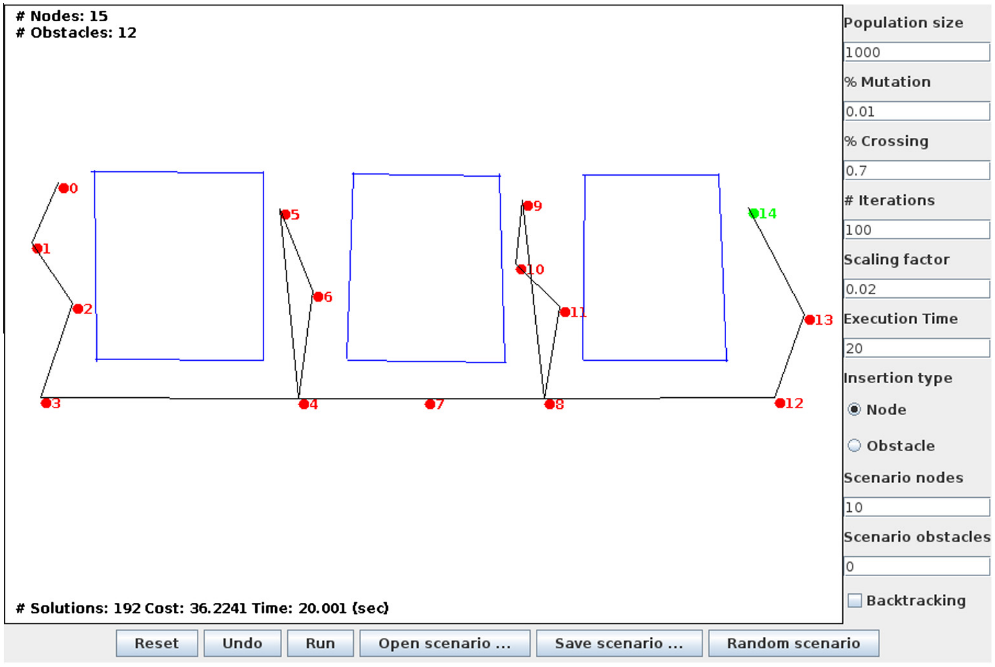
In this case we consider a roundtrip done when all the nodes of the cloud have been visited. Table 1 shows the values obtained with these tests, where the *Traveler* agent has to apply every fault tolerance method implemented. Random components of some of these methods lead to much more variable roundtrip times, useful only to confirm that the application is still working, but not suitable for benchmarking purposes.

5.3. Real world scenario

Finally we tested our application in a real world scenario, modeling a MCI outside the building of our faculty (Fig. 18). We used 15 sensor nodes for our tests and we checked the correct working of the application both with all the nodes working and deactivating some of them. Fig. 19 was taken during the experimentation where every individual acted as an MCI victim and carried a WSN node in his hands. Note that the photograph was taken using a fisheye lens, thus distances may appear distorted. Use this GPS positions for an

Table 1
21 node disconnected scenario with one *Traveler* per cloud.

Lap	Total time	Lap time
1	01:02.832	01:02.832
2	01:58.482	00:55.650
3	02:58.446	00:59.964
4	03:40.061	00:41.615
5	04:01.730	00:21.669



Nodes: 15
Obstacles: 12

Solutions: 192 Cost: 36.2241 Time: 20.001 (sec)

Population size: 1000
% Mutation: 0.01
% Crossing: 0.7
Iterations: 100
Scaling factor: 0.02
Execution Time: 20
Insertion type: [✓ Node] Obstacle
Scenario nodes: 10
Scenario obstacles: 0
☐ Backtracking

Fig. 18. Building scenario (same GUI as in Fig. 12).

accurate view of the area of our test: (41.499727, 2.112164), (41.500070, 2.113401).

Tests with all the nodes working carried out in this scenario prove that what was encouraging in the testbed is also applicable to a real scenario. The roundtrip times for the application in a network with all the nodes working also show that the values obtained in the testbed can be extrapolated to real scenarios. The top line in Fig. 20 shows the results of 10 continuous roundtrips to the WSN, being the mean of almost 39 s, i.e., a little bit more than one second per node, a value very similar to that obtained in the testbed.

To test the fault-tolerance overhead we conducted another test using the same building scenario where we randomly fail 5 nodes, never consecutive. Fig. 20 shows three graphs, one for the times of our application running on a 15 node WSN, another one showing a 10 node WSN run, and a 15 to 10 node run. In this third graph the first two values correspond to the WSN with all the nodes running, following a one node per run fail until reaching the final 10 node configuration. The figure shows that using the **jump-after-next** fault-tolerance method does not increase significantly the time needed to make the full roundtrip of the WSN, see last four samples. This is because the most time consuming task of our application are the transmissions required to migrate the *traveler* agent and its data.

Tests with consecutive failed nodes have been also performed, proving the usefulness of the application in these cases. Roundtrip times in these situations are variable and differ a lot from those seen in the other tests. As those in Table 1 they only confirm that the application is still working, but are not suitable for benchmarking purposes. Table 2 shows the times obtained when traveling the building scenario network.

5.4. Energy consumption

The TelosB compatible nodes used in our application are powered by two AA batteries (3 V) and consume roughly from 19,2 mA (RX) to 20,6 mA (TX) when active, according to their Maxfor product reference guides. Experimentally, we reached a mean continuous operation of our application (i.e., a continuous migration of our *Traveler* agents), in a scenario of 15 working nodes with standard AA alkaline batteries, of nearly 5 days. This lifetime

Fig. 19. Field deployment.

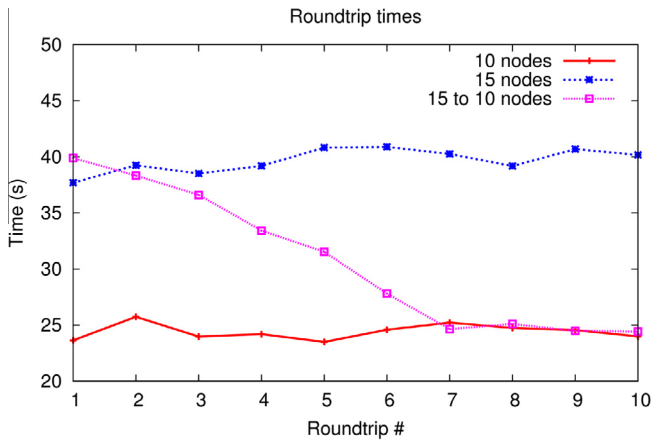


Fig. 20. Roundtrip times with nodes falling from 15 to 10.

Table 2
15 node building MCI with 5 consecutive failed nodes.

Lap	Total time	Lap time
1	2:26.246	2:26.246
2	4:30.442	2:04.196
3	5:47.146	1:16.704
4	6:20.464	0:33.318
5	8:02.186	1:41.722

is similar to that of other existing WSN applications for emergency scenarios [23], and fairly more time than the expected time to rescue triaged victims in a MCI.

5.5. Deployment issues

We observed an interesting behavior when injecting the agent in a newly closed WSN. Some times, when testing the network against node failures to measure the correct working of the **random-jump** method, the *Traveler* mobile agent got lost for a while, then following the pre-established itinerary as if nothing had happened. This issue raised important headaches to the team, forcing to reprogram the itinerary of the nodes, recalculate its values, etc. Finally, after some discouraging tests a light appeared in our testbed. Not in a node from the WSN, but in a foreign node, not used in the current test but powered on. The *Traveler* mobile agent, in one of its **random-jumps**, reached a neighbor node out of its WSN, and jumped to it.

After some more tests with the unused nodes disconnected everything ran as expected, finishing an, in the end, enriching experience.

At the moment, to avoid this issue we recommend to switch off the unused nodes before injecting the *Traveler* agent. This can be done either having all nodes disconnected from the beginning, and powering on just those necessary in the current WSN, or switching off the unused nodes after closing the WSN and before injecting the *Traveler* agent (either removing the batteries or with the on/off switch of MTM-CM4000-MSP nodes).

6. Conclusion

The paper shows how the adaptation of conventional separate itineraries to the Agilla middleware is both feasible (in space and time), and useful.

Regarding feasibility, in a highly resource-constrained environment such as Agilla running on TelosB nodes, we have developed an application in which we can store separate itineraries with a length from 2 to 512 positions. Moreover, we experimentally found that our mobile agent *Traveler* (of 820 bytes) is able to follow this itinerary with an approximated time of migration between nodes of 1.5 s.

Regarding usefulness, the inclusion of the separate itineraries (with their sequential and alternative entries) in Dynamic MAETT has shown the utility of these itineraries as they allow the application to be very fault tolerant, reacting in front of failing WSN nodes. WSN partitions can also be easily handled by the application just with two mobile agents following different itineraries.

All the fault-tolerant strategies have been tested and all worked properly, allowing to keep visiting all the nodes of all the partitions in a variable but limited time. This time depends on the necessary number of random jumps for the alternatives, and on the number of coincident nodes in the different itineraries of all the *Traveler* agents.

In addition, we found that the middleware Agilla (despite its painful coding) is flexible and robust enough to support a new application following a new approach with agents roaming the whole WSN according to separate itineraries. These itineraries have been easily incorporated to Agilla, and this leads us to believe that they can also be included in other WSN mobile agent systems running on less restrictive environments.

One of the drawbacks of our application is the need to leave the injecting node, the one attached to the handheld device, in connection range of the created WSN. This forces the medical personnel to use that node as part of the WSN, having to replace the injecting node every time a new WSN is created. Albeit the injecting node ends up working as any other node and the medical personnel does not have to carry any extra weight for this matter, the need of changing it every time a new WSN has to be created adds an extra task to the medical personnel. Not leaving the injecting node in connection range makes the application to behave strangely after some time, to finally end up totally motionless. Without having more Agilla internal information regarding this issue, we figured out that the injecting node is acting as a kind of a necessary *cluster head* for the WSN.

Moreover, our application is limited to operate with a single WSN. We are working on expanding it to operate with several independent WSN. At the moment we are adding network identifiers to the itinerary construction which will serve as the basis of a multi-network triaging system. These identifiers will as well prevent the issue of the *Traveler* agent jumping to external nodes when performing random jumps (see Section 5.5).

Improving the fault-tolerance of our system is in our future plans. We are working to reach the maximum number of agents allowed by Agilla to obtain more robust fault-tolerance mechanisms. These mechanisms would be based on eight *Travelers* with different itineraries. To reach optimal results these initial itineraries could be calculated according to more complex planning algorithms like those of [33].

We also plan to finish the deployment of our Dynamic MAETT application by adding actual body sensors to our TelosB nodes in order to conduct more realistic field trials.

As future work it could also be interesting to port our application to other WSN mobile agent middlewares such as WISEMAN or MAPS, to allow a direct comparison among those middlewares. Translating our application to a TinyOS (non-agent) environment (similar to [23]) could also be interesting to get an additional understanding of benefits and drawbacks of using mobile agents in WSNs.

Acknowledgments

This work was partly supported by the Catalan AGAUR 2009SGR-1224 project, the Spanish MICINN TIN2010-15764 project, and the Irish HEA PRTLI-IV NEMBES project.

We would like to thank Aitor López for his collaboration when implementing the DFS algorithm in [41].

References

[1] C.-L. Fok, G.-C. Roman, C. Lu, Agilla: a mobile agent middleware for self-adaptive wireless sensor networks, ACM Transactions on Autonomous and Adaptive Systems 4 (3) (2009) 1–26.
[2] F. Aiello, G. Fortino, R. Gravina, A. Guerrieri, A java-based agent platform for programming wireless sensor networks, The Computer Journal 54 (3) (2011) 439–454.
[3] J.W. Hui, D. Culler, The dynamic behavior of a data dissemination protocol for network programming at scale, in: Proceedings of the Second International Conference on Embedded Networked Sensor Systems, ACM Press, 2004, pp. 81–94.
[4] A. Dunkels, B. Gronvall, T. Voigt, Contiki – a lightweight and flexible operating system for tiny networked sensors, in: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, LCN '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 455–462.
[5] P. Levis, D. Gay, D. Culler, Active sensor networks, in: Proceedings of the Second Conference on Symposium on Networked Systems Design & Implementation, NSDI'05, vol. 2, USENIX Association, Berkeley, CA, USA, 2005, pp. 343–356.
[6] Y. Yu, L.J. Rittle, V. Bhandari, J.B. LeBrun, Supporting concurrent applications in wireless sensor networks, in: Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems, SenSys '06, ACM, New York, NY, USA, 2006, pp. 139–152.
[7] R. Balani, C.-C. Han, R.K. Rengaswamy, I. Tsigkogiannis, M. Srivastava, Multi-level software reconfiguration for sensor networks, in: Proceedings of the Sixth ACM & IEEE International conference on Embedded Software, EMSOFT '06, ACM, New York, NY, USA, 2006, pp. 112–121.
[8] L. Szumel, J. LeBrun, J.D. Owens, Towards a mobile agent framework for sensor networks, in: Proceedings of the Second IEEE Workshop on Embedded Networked Sensors, IEEE Computer Society, Washington, DC, USA, 2005, pp. 79–87.
[9] Y. Kwon, S. Sundresh, K. Mechitov, G. Agha, Actornet: an actor platform for wireless sensor networks, in: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '06, ACM, New York, NY, USA, 2006, pp. 1297–1300.
[10] D. Georgoulas, K. Blow, In-motes: an intelligent agent based middleware for wireless sensor networks, in: Proceedings of the Fifth WSEAS International Conference on Applications of Electrical Engineering, AEE'06, World Scientific and Engineering Academy and Society (WSEAS), Stevens Point, Wisconsin, USA, 2006, pp. 225–231.
[11] S. Gonzalez-Valenzuela, M. Chen, V. Leung, Programmable middleware for wireless sensor networks applications using mobile agents, Mobile Networks and Applications 15 (2010) 853–865.
[12] S. González-Valenzuela, M. Chen, V.C. Leung, Chapter 4 – Applications of Mobile Agents in Wireless Networks and Mobile Computing, Advances in Computers, vol. 82, Elsevier, 2011. pp. 113–163.
[13] J. White, Telescript technology: mobile agents, in: J. Bradshaw (Ed.), Software Agents, MIT Press, 1996, pp. 437–472.
[14] M. Chen, S. González-Valenzuela, V.C. Leung, Applications and design issues of mobile agents in wireless sensor networks, IEEE Wireless Communications 14 (6) (2007) 20–26.
[15] C.-L. Fok, G.-C. Roman, C. Lu, Rapid development and flexible deployment of adaptive wireless sensor network applications, in: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems, ICDCS '05, IEEE Computer Society, Washington, DC, USA, 2005, pp. 653–662.
[16] S. González-Valenzuela, S. Vuong, V.C. Leung, A mobile code platform for distributed task control in wireless sensor networks, in: Proceedings of the Fifth ACM International Workshop on Data Engineering for Wireless and Mobile Access, MobiDE '06, ACM, 2006, pp. 83–86.
[17] D. Wong, N. Paciorek, T. Walsh, J. DiCelie, M. Young, B. Peet, Concordia: an infrastructure for collaborating mobile agents, in: K. Rothermel, R. Popescu-Zeletin (Eds.), Mobile Agents, Lecture Notes in Computer Science, vol. 1219, Springer, Berlin/Heidelberg, 1997, pp. 86–97.
[18] J. Polastre, R. Szewczyk, D. Culler, Telos: Enabling ultra-low power wireless research, in: IPSN '05 Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks, ACM and IEEE, 2005, pp. 364–369.
[19] F.L. Bellifemine, G. Caire, D. Greenwood, Developing Multi-Agent Systems with JADE, Wiley, 2007.
[20] R. Martí, S. Robles, A. Martín-Campillo, J. Cucurull, Providing early resource allocation during emergencies: the mobile triage tag, Journal of Network and Computer Applications 32 (2009) 1167–1182.
[21] E. Mercadal, S. Robles, R. Martí, C. Sreenan, J. Borrell, Heterogeneous multiagent architecture for dynamic triage of victims in emergency scenarios, in: Nineth International Conference on Practical Applications of Agents and Multiagent Systems (PAAMS), 2011, pp. 237–246.
[22] P. Levis, N. Lee, M. Welsh, D. Culler, TOSSIM: accurate and scalable simulation of entire TinyOS applications, in: SenSys '03: Proceedings of the First International Conference on Embedded Networked Sensor Systems, ACM, New York, NY, USA, 2003, pp. 126–137.
[23] T. Gao, T. Massey, L. Selavo, D. Crawford, B. rong Chen, K. Lorincz, V. Shnayder, M. Welsh, The advanced health and disaster aid network: a light-weight wireless medical system for triage, IEEE Transactions on Biomedical Circuits and Systems 1 (3) (2007) 203–216.
[24] P. Levis, D. Culler, Maté: a tiny virtual machine for sensor networks, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS-X, ACM, New York, NY, USA, 2002, pp. 85–95.
[25] D. Gelernter, Generative communication in Linda, ACM Transactions on Programming Languages and Systems 7 (1985) 80–112.
[26] C.-L. Fok, G.-C. Roman, C. Lu, Servilla: a flexible service provisioning middleware for heterogeneous sensor networks, Science of Computer Programming 77 (6) (2012) 663–684.
[27] S.S. Manvi, P. Venkataram, Applications of agent technology in communications: a review, Computer Communications 27 (15) (2004) 1493–1508.
[28] J. Cucurull, J. Ametller, R. Martí, Agent mobility, in: F.L. Bellifemine, G. Caire, D. Greenwood (Eds.), Developing Multi-Agent Systems with JADE, Wiley, 2007, pp. 115–130.
[29] J. Cucurull, R. Martí, D. Navarro-Arribas, S. Robles, B. Overeinder, J. Borrell, Agent mobility architecture based on IEEE-FIPA standards, Computer Communications 32 (2009) 712–729.
[30] M. Straer, K. Rothermel, C. Maihöfer, Providing reliable agents for electronic commerce, in: Proceedings of the International IFIP/GI Working Conference on Trends in Distributed Systems for Electronic Commerce, Springer-Verlag, London, UK, 1998, pp. 241–253.
[31] C. Garrigues, S. Robles, J. Borrell, Securing dynamic itineraries for mobile agent applications, Journal of Network and Computer Applications 31 (2008) 487–508.
[32] Q. Wu, N.S.V. Rao, J. Barhen, S.S. Iyengar, V.K. Vaishnavi, H. Qi, K. Chakrabarty, On computing mobile agent routes for data fusion in distributed sensor networks, IEEE Transactions on Knowledge and Data Engineering 16 (6) (2004) 740–753.
[33] M. Chen, L.T. Yang, T. Kwon, L. Zhou, M. Jo, Itinerary planning for energy-efficient agent communication in wireless sensor networks, IEEE Wireless Communications 60 (7) (2011) 3290–3299.
[34] H. Qi, F. Wang, Optimal itinerary analysis for mobile agents in ad hoc wireless sensor networks, in: Proceedings of IEEE ICC, 2001, pp. 147–153.
[35] D. Massaguer, C.-L. Fok, N. Venkatasubramanian, G.-C. Roman, C. Lu, Exploring sensor networks using mobile agents, in: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS '06, ACM, New York, NY, USA, 2006, pp. 323–325.
[36] W. Cai, M. Chen, T. Hara, L. Shu, T. Kwon, A genetic algorithm approach to multi-agent itinerary planning in wireless sensor networks, MONET 16 (6) (2011) 782–793.
[37] X. Wang, M. Chen, T. Kwon, H. Chao, Multiple mobile agents' itinerary planning in wireless sensor networks: survey and evaluation, Communications IET 5 (12) (2011) 1769–1776.
[38] G. Super, S. Groth, R. Hook, START: simple triage and rapid treatment plan, Hoag Memorial Hospital Presbyterian, Newport Beach, CA, 1994.
[39] O. Chipara, C. Lu, T.C. Bailey, G.-C. Roman, Reliable clinical monitoring using wireless sensor networks: experiences in a step-down hospital unit, in: Proceedings of the Eighth ACM Conference on Embedded Networked Sensor Systems, SenSys '10, ACM, New York, NY, USA, 2010, pp. 155–168.
[40] S.S. Iyengar, N. Parameshwaran, V.V. Phoha, N. Balakrishnan, C.D. Okoye, Fundamentals of Sensor Network Programming: Applications and Technology, Wiley-IEEE Press, 2010.
[41] E. Mercadal, S. Robles, R. Martí, C. Sreenan, J. Borrell, Double multiagent architecture for dynamic triage of victims in emergency scenarios, Progress in Artificial Intelligence 1 (2) (2012) 183–191.