Exploring Sensor Networks using Mobile Agents

Daniel Massaguer†, Chien-Liang Fok‡, Nalini Venkatasubramanian†,
Gruia-Catalin Roman‡, and Chenyang Lu‡

†Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA 92697, USA
{dmassagu, nalini}@uci.edu

‡Department of Computer Science and Engineering
Washington University in Saint Louis
Saint Louis, MO, 63105, USA
{liang, roman, lu}@cse.wustl.edu

Abstract

Today's wireless sensor networks have limited flexibility because their software is static. Mobile agents alleviate this problem by introducing mobile code and state. Mobile agents differ from traditional programs in that they are capable of moving and cloning throughout a network performing application-specific tasks. As they move, they carry their code and state, and are able to intelligently interact with each other. However, in order to realize the full benefits of mobile agents, new algorithms need to be developed for controlling their migration patterns. This paper presents a mechanism for ensuring efficient network exploration, e.g., that every node is visited. Network exploration has many applications ranging from data collection to network health monitoring. The mechanism uses a genetic algorithm to determine the number of agents and their itineraries, followed by techniques for intelligent in-network adaptation to unpredictable situations like node failure. This paper presents the genetic algorithm and its strategies for in-network adaptation. It also shows simulation results that demonstrate the technique's effectiveness, and an evaluation using a real wireless sensor network consisting of 25 MICA2 motes running Agilla, a mobile agent middleware platform.

I. Introduction

Wireless sensor networks (WSNs) have attracted a tremendous amount of interest in recent years as embedded systems, micro-sensors, and batteries improve. A WSN consists of numerous tiny sensors embedded within the environment that autonomously form wireless ad hoc networks. By increasing the density and scale of traditional sensing systems by several orders of magnitude, a WSN enables computer programs to gain higher levels of context-awareness than ever before. Current applications for WSNs include structural and habitat monitoring [1], [2], fire fighting [3], and precision agriculture [4].

WSN developers are confronted with a bewildering set of conflicting requirements, mostly related to software flexibility and efficiency. For example, many WSNs are deployed for long periods of time, which virtually guarantees that the user requirements will change. Unfortunately, most WSNs run statically installed software that is loaded onto the sensors prior to deployment, limiting network adaptivity to the tweaking of certain pre-defined parameters. Another conflicting property stems from the fact that WSNs are expected to cover a wide geographic area servicing many users that need to run different programs. Since the current software infrastructure consisting of TinyOS [5] and NesC [6] statically compiles the software prior to deployment, everyone's programs must be included *a priori*. Moreover, micro-sensors have extremely little memory (e.g., the MICA2 motes have a mere 128KB [7]), preventing multiple applications from co-existing, thus reducing the network's utility.

Mobile agents have recently been embraced as a solution to the problems mentioned above. Unlike traditional programs that remain statically installed on a single node, mobile agents are capable of intelligently migrating across the network carrying both code and state. As they migrate, they perform application-specific tasks like taking sensor readings, performing in-network aggregation, and coordinating with other agents to achieve a common goal. This enables the development of fluid applications capable of adapting to the intricacies of WSNs. For example, mobile agents may swarm around a fire forming a dynamic barrier for tracking purposes [8]. In addition to the well-established advantages that mobile agents provide like asynchronous operation and reduced bandwidth usage [9], mobile agents allow WSNs to be dynamically reprogrammed by enabling users to inject new agents into the network and allowing old ones to die. They also allow multiple applications to co-exist since multiple agents may reside on each node. In all, mobile agents can increase the flexibility and utility of both applications and the network.

Before mobile agents can be used, however, new algorithms must be created for controlling agent behavior. One topic of particular interest is *network exploration*. The goal of network exploration is to visit every node and to do so efficiently in terms of time and energy. This paper addresses, the specific issue of sensor network exploration using multiple mobile agents. Network exploration is used in many applications. They include data collection, network diagnostic and health monitoring, residual energy scanning, topology discovery, and global reprogramming. They all require that every node be visited, though the order and number of visitations does not matter. The network exploration mechanism must determine the number of agents and their itineraries, and the techniques for dealing with unexpected in-network circumstances, e.g., changes in topology.

WSNs impose several nonfunctional requirements like energy efficiency, timeliness, and fault tolerance that a network exploration must address.

- **Energy efficiency.** WSNs rely on non-renewable energy (e.g., batteries). Physical access to a node after it is deployed is not practical because of the sheer number and their embedded-ness. Thus, the network exploration algorithm must be energy efficient. In our approach, greater efficiency is achieved by minimizing the total number of agent migrations and reducing the agent size.
- **Timeliness.** Many applications require a result within a certain amount of time. For example, in a data collection application used for retrieving seismic data from sensors on the ocean floor, the quake data must be delivered in time for seismologists to determine and alert those who will be affected by the resulting tsunami. The network exploration algorithm must adhere to strict time deadlines in addition to being energy efficient, or face disastrous consequences. We developed techniques to support multiple concurrent explorations of the WSN for this purpose.
- **Fault Tolerance.** Since WSNs are composed of potentially thousands of embedded sensors that are exposed to the elements, many of them will fail mid-deployment. New sensors may later be added to replace them. To save power, weak unreliable radios are used, further complicating neighbor availability. All of this results in a highly dynamic network topology. The network exploration mechanism we propose must tailor agents to intelligently deal with dead nodes.

The remainder of this paper is organized as follows. Section II describes related work. Section III formally defines the network exploration problem and presents our solution. Section IV presents experiments that evaluate the solution. Section V describes future work. The paper ends with conclusions in Section VI.

II. Related Work

Several other studies have investigated the use of mobile agents in WSNs. In [10], the authors study the problem of routing agents carrying a single data packet from a source to a destination, as shown in Fig. 1. Agents independently choose the next hop based on the routing cost and the amount of energy remaining on the neighbor; it uses a heuristic based on the distributed local view available at every node. To avoid an explosion in the number of agents with the number of data packets, the system automatically merges mobile agents carrying redundant data.

In [11], the authors present a genetic algorithm for determining the route of a mobile agent that loops through the network delivering an aggregate result based on the information on all of the nodes it visited, as shown in Fig. 2. This approach is used in object tracking and detection applications where nodes sensing stronger signals are more important. The authors prove the NP-completeness of the problem, justifying the need of heuristics (e.g., a genetic algorithm) to solve it. However, they do not consider the use of multiple mobile agents operating in parallel to reduce the total latency, or possible interferences between different mobile agent routes. Furthermore, when a failure is detected, the routes are adapted by

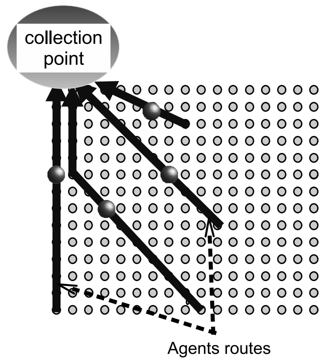

Fig. 1. One mobile agent per source and per unit of data.

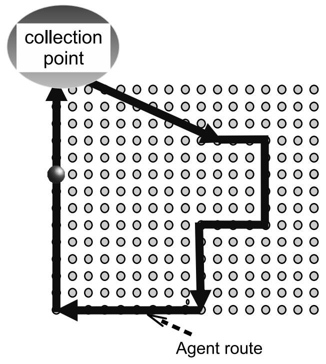

Fig. 2. One single mobile agent following a precomputed itinerary.

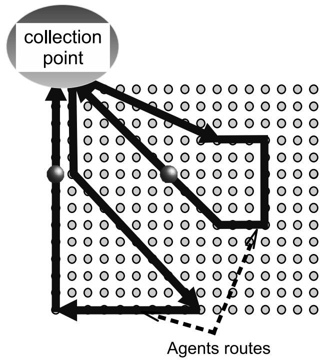

Fig. 3. Multiple Mobile agents collecting information.

the central collection point instead of having the agents independently, and intelligently, adapt – a key feature of all agent systems.

One possibility is to model sensor network exploration as a multi-Travelling Salesman Problem [12]. There are significant differences, however. For example, unlike the multi-TSP nodes may be visited more than once. Depending on the topology and the time constraints, the mobile agents routes may have to overlap to meet the time constraints. Consider a 4x4 grid where the collection point is at the lower-left corner. If agents routes start and end at the collection point, only one agent can be used to assure all routes are disjoint because the collection point has only three neighbors. In this case, we divide the WSN into non-overlapping regions where each region exploration is a separate TSP. Consequently, without route overlapping the minimum length of a path is sixteen hops. If route overlapping were allowed the minimum length of a path is six hops.

This paper proposes to use multiple mobile agents that simultaneously collect information from the WSN as shown in Fig. 3. By using multiple mobile agents, we can explore the network quicker than a single agent [11]. By allowing the agents to aggregate data from the nodes they have visited, we can achieve better energy efficiency than if an agent had been used for each data packet [10]. We developed a centralized genetic algorithm that plans the routes that the mobile agents, running in parallel, ought to follow. The paths must start and end at the collection point. They must be chosen so as to guarantee that all nodes are visited within a certain deadline while minimizing the energy spent. The mobile agents follow these routes and adapt them to unexpected events like network and node failures.

III. Problem Formulation and Solution

Using multiple mobile agents to explore a WSN offers the benefits of flexibility, load distribution, and asynchronous operation. However, since WSNs are large and resource-constrained, these mobile agents ought to be deployed so that energy usage is minimized, and real-time guarantees are met. This section introduces the necessary notation, formulates a minimization problem, and presents a solution.

A. *WSN Model and Assumptions*

A WSN consists of a multitude of nodes, each containing a set of sensors, microcontroller, external memory, radio transceiver, and power source. Nodes can sense real-world phenomena, perform local computations, store information, receive data, and transmit data. Receiving and transmitting data costs more energy than performing local computations. Moreover, we assume that the WSN is connected to the rest of the world via a single base station, or collection point.

We assume that the network exploration mechanism does not need to worry about maintaining a sleep schedule. Sleep schedules are often used to save energy by configuring the nodes to sleep most of the time, only briefly waking up to perform application-specific tasks. There are many components in the network stack that can handle sleep schedules without affecting higher-level components [13]–[16]. By relegating the sleep schedule to lower-level components, the only way the network exploration algorithm can save energy is by minimizing the number of bytes transmitted.

Since many different applications will use our network exploration agents, they have to be designed to allow any type of behavior to be plugged into them. This determines what the agent does on each node (e.g., remember the maximum temperature of all sensors visited), and is application specific. We assume that the code for this behavior is stored at, or is otherwise available to, the collection point. This allows the agent to start and finish at the collection point (otherwise, it would have to visit a separate code repository to pick up application-specific behavior).

Finally, we assume that the amount of time an agent spends processing data at each node and the time it takes to migrate an agent is predictable and known. In practice, such information may be estimated conservatively based on empirical measurements (see section IV below), where the measured *time per node* includes all: processing time, migration time, and node-sleeping time. This knowledge is necessary for our algorithm to ensure that the network is explored within a certain time.

B. Problem Formulation

Given a sensor network, Ω, with a single collection point, the ultimate goal is to explore the entire network by visiting every node within a certain amount of time, while minimizing energy consumption. This can be done using one or more mobile agents operating autonomously and in parallel following individual routes. These routes must start and end at the collection point.

There are many applications, however, that do not need every node to be visited, especially if the nodes are densely deployed. For example, it may not be necessary to visit two nodes separated by one meter if the task is to determine the average temperature along a path. To account for this, we relax the problem by introducing a density variable d. The density variable, d, specifies how many of the nodes within the network need to be visited for the network to be considered explored. Specifically, a network is considered explored if all nodes $m \in \Omega$ have either been visited, or have a d-hop neighbor that has been visited. Note that if $d = 0$, the exploration problem is restored to visiting every node.

The time constraint t is specified by the maximum length of a path, expressed as number of nodes. We do not use an actual time since the amount of time an agent spends on each node, and the amount of time it takes an agent to migrate is assumed to be predictable and known.

The cost of an agent, a, visiting a node, m, following a certain path, p, is denoted $C_a(m, p)$. It is defined in terms of the weight of the agent, $W_a(m, p)$ and the amount of energy remaining on the node, E_m, as follows:

$$C_a(m, p) = \frac{W_a(m, p)}{E_m} \tag{1}$$

The weight of agent a, $W_a(m, p)$, is a function that returns the weight, in bytes, of agent a, when visiting node m, following path p. The weight of an agent depends on the size of its code and on the size of the data it carries. The size of its code is fixed, whereas the size of the data varies every hop, and largely depends on what the application does upon visiting each node. For example, if the agent does pure-aggregation (e.g., find the maximum temperature), the size of the application-specific data remains constant (e.g., a single temperature reading). But, if the agent does pure-collection (e.g., return all of the temperatures), the amount of application-specific data increases with each hop. Regardless of whether the application-specific data changes, the network-exploration data always decreases with each hop. This is because as it traverses its path, it can forget the portion of the path that has already been fulfilled. This usually frees 2-8 bytes each hop, which is the size of a node's address [17]. Fig. 4 shows how the weight of an agent changes as it progresses along its path. Note that the pure-collection agent increases in size as

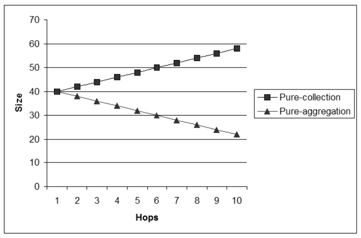

Fig. 4. $W_a(,m,p)$ of a pure-collection and pure-aggregation agent.

it picks up additional sensor data values (in this case, the additional sensor data outweights the decrease in path length). The pure-aggregation agent decreases in size since the amount application-specific data remains constant, while the path length decreases. In this examples, the pure-collection agent's weight increases two bytes per hop since it keeps the address and a two-byte sensed value of every node it visits. The pure-aggregation agent's weight decreases two bytes per hop, since we assume two-byte addresses.

Finally, let P be the set of all paths and P_m be the set of paths that include node m, where $P_m \subseteq P$, we can formulate the following minimization problem:

$$min \sum_{m \in \Omega} \sum_{p \in P_m} \frac{W_a(m,p)}{E_m} \tag{2}$$

such that

$$\forall p \in P : |p| \leq t$$

and the WSN is covered with density d.

Thus, we need to minimize the sum of all individual node costs, where the individual cost is the quotient of all the bytes transmitted by that node, and the amount of energy the node had initially. This formulation can also be extended to multiple rounds of network exploration. That is, for n rounds with time deadlines t_1, t_2, ..., and t_n, the problem still remains to minimize equation 2. Hence, the goal is to minimize the sum of all node costs based on an estimate of the initial energy remaining at each node, and the total number of bytes transmitted by that node.

The cost function is defined in terms of number of bytes transmitted and energy availability such that the less energy the node has, the more expensive it is to send each byte. This balances the remaining energy throughout the network better than, for instance, a more linear cost such as $C_a(m,p) = W_a(m,p) - E_m$.

C. *Genetic Algorithm for Multi Mobile Agents Route Planning*

Given the number of variables (e.g., the network topology, size and kind of mobile agent, time constraint t, and coverage density d) and the nature of the problem itself, the solution is indeed not trivial. Even in the simplified case where the network topology is a grid and each node has at most eight neighbors, doing an exhaustive search implies exploring a solution space of $O(2^{2^{3t}})$. That is, if paths are at most t long, and each node has up to eight neighbors, the set of all possible paths is a set of size $O(8^t)$. And since each solution is a set of paths, the search space becomes $O(2^{8^t}) = O(2^{2^{3t}})$. It is clearly unfeasible to find the solution by exhaustive search. Furthermore, not all $2^{2^{3t}}$ solutions are sufficient, since not all combinations of paths cover the entire WSN. The set of sufficient solutions is smaller than the set of all solutions. Defining the size of this set and of the optimal set is currently part of our on-going research and some promising simulation results regarding these sets are discussed in the *Evaluation* section.

Given the complexity of the problem and the size of the search space, a heuristic is necessary. In addition, the sensor network's topology may change due to nodes running out of energy, desynchronizing, or being affected by external events (e.g., being moved by the wind). Gathering and continuously monitoring the

```
GeneticAlgorithm($\Omega$,d,t,Wa(m,p))
  CreateInitialPopulation(N);
   For generation=1 to MaxGenerations
     Cross();
     Mutate();
     NaturalSelection();
     AddNewIndividuals(I);
   endFor
  return AgentRoutes
endGeneticAlgorithm
```

Fig. 5. The Genetic Algorithm.

topology is energy consuming. Thus, in order to solve the formulated minimization problem 2, we propose an algorithm based on two phases: an initial route planning phase done centrally by a genetic algorithm and an in-network route adaptation phase that is done by each agent autonomously.

First, a genetic algorithm plans the routes that the agents will follow. This is done at the collection point, which is assumed to have plenty of power (e.g., a laptop). A genetic algorithm is a heuristic local search method for discrete optimization inspired by real-life processes of genetics and evolution [18]. As in the Darwinian evolution, the original population (a set of solutions), evolves through reproduction, mutation, and natural selection. Reproduction (or crossing) is characterized by the random selection of two fit (i.e. close to the optimal) solutions and posterior crossing of their chromosomes (i.e. combination of the two solutions to get a new one). Mutation is based on randomly changing a given characteristic of some randomly selected solutions. And last, selection is based on the survival of the fittest;' only the solutions that are close to the optimal are kept. *Crossing* aims to combine two solutions in an effort to get closer to a local minimum. *Mutation* is used in an effort to avoid local minimums. As in evolution, the idea behind mutation is to provide variability that allows exploring a different region of the solution space, hopefully avoiding local minimums. *Selection* enables efficient memory usage by discarding bad solutions.

Because the network topology changes and determining the network topology at the collection point is time and energy consuming, the network view used by the genetic algorithm may be outdated. Thus, when following its route, every mobile agent makes its own local decisions to adapt its given route to topology changes.

Fig. 5 is the pseudo-code of the genetic algorithm (GA) proposed here. In particular, it takes as input the definition of a sensor network Ω (i.e. the topology), the time constraint t (i.e. the maximum number of nodes of the longest route), the density d, and the specification of the varying size of the agents $W_a(m,p)$. The outcome is the number of agents and their routes, such that the collection is achieved in an energy-aware manner and within a time deadline. More details on each of the GA following subparts follow: *Generation of initial population*, *Crossing*, *Mutation*, *Selection*, and *Adding of new individuals*.

1) Generation of Initial Population and Adding of New Individuals : Initially, an initial set, P, of solutions is generated. These solutions are a collection of random paths that cover the network, and are generated by repeatedly calculating paths until the whole network, Ω, has been covered with a density of d. Each path is selected by starting at the collection point and iteratively randomly selecting a neighboring node until the collection point is reached again. Only those neighboring nodes whose minimum distance (in number of hops) to the collection point is smaller or equal to the time remaining to meet the time deadline will be selected. As the time constraint is expressed in maximum number of nodes per path, comparing a nodes distance to the collection point with the time deadline is trivial. All feasible neighboring nodes are equally likely to be selected, and the initial size of P, $|P|$, is a design parameter. The pseudo-code for generating a new solution and for generating the initial population is in Fig. 6, 7, and 8. Fig. 7 and 8 also describe how to add new individuals at every iteration. Fig. 9 illustrates an example of creating a

```
CreateInitialPopulation(N)
   AddNewIndividuals(N);
EndCreateInitialPopulation
```

Fig. 6. CreateInitialPopulation

```
AddNewIndividuals(I)
   For individual=1 to I
      CreateNewIndividual();
   endFor
EndAddNewIndividuals
```

Fig. 7. AddNewIndividuals

```
CreateNewIndividual()
   While (not WSN.covered())
      //create a new path p
      Path p=null;
      p+ =BS; // BS = source node
      p+ =Unif(Neighbors(p.last));
      While(p.last != BS)
         p+ =Unif(Neighbors(p.last));
      EndWhile
      //add the new path
      Individual.add(p);
   EndWhile
   EvaluateIndividual();
EndCreateNewIndividual
```

Fig. 8. CreateNewIndividual

new individual.

2) Crossing and Mutation: At every iteration, a new generation is added to the population P by selecting x individuals to be crossed. Two individuals are crossed by selecting a subset of their paths to create a new individual. We can see an example of crossing in Fig. 9. Depending of how those individuals are selected, and which subset of their paths is crossed, we have different crossing variations. Ranging from *Random* to *Deterministic*, those variations are:

Random crossing (R). Randomly select $x\%$ of the best and cross them at each iteration by randomly selecting one path from each parent, until the WSN is covered with density d.

PseudoRandom crossing (PR). Randomly select $x\%$ of the best, and cross them at each iteration $i > 1$ by selecting the path that minimizes the new total cost. At iteration 1, the first path is randomly selected.

PseudoDeterministic crossing (PD). Randomly select $x\%$ of the best, and cross them at each iteration by selecting the path that minimizes the new total cost.

Deterministic crossing (D). Select all the $x\%$ best and cross them all at each iteration by selecting the path that minimizes the new total cost.

Mutation is done by self-crossing one individual with itself (Fig. 9), and the same crossing variations are applicable to the mutation operator.

3) Evaluation and Selection: All solutions are evaluated by:

$$f(individual) = \sum_{m\in\Omega} \sum_{p\in Pm} \frac{W_a(m,p)}{E_m}, \tag{3}$$

and sorted from the minimum to the maximum. An example evaluation is depicted in Fig. 9. Upon evaluation, $k_n\%$ of the worst individuals are randomly selected and removed from the population. Note that to avoid loosing the best solutions generated so-far, the five best solutions are never removed, irrespective of k_n.

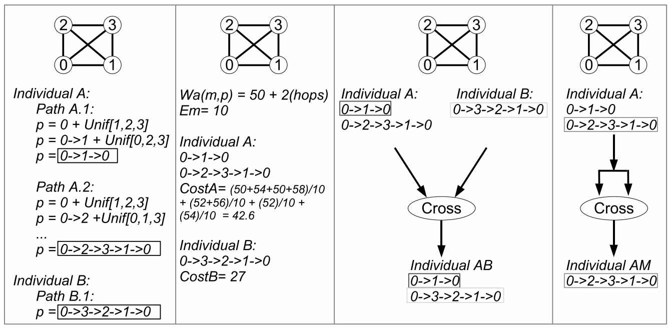

Fig. 9. An example of creating two individuals A and B, evaluating them, crossing them, and mutating A.

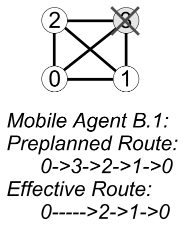

Fig. 10. A Mobile agent bypassing a node failure.

D. In-Network Mobile Agent Route Adaptation

Though not as dynamic as ad hoc networks, the topology of WSNs does change. Mobile agents follow the planned routes adapting them to network failures and topology changes. In particular, if an agent cannot successfully migrate from the i^{th} node of its proposed route, to the $(i+1)^{th}$, it incrementally tries to migrate to the j^{th} node such that $j > (i+1)$. Given a typically dense WSN, at least one of the nodes that follow the $(i+1)^{th}$ node on the agent itinerary may eventually be in the i^{th} node radio range. Which allows the agent to resume its exploration. Fig. 10 presents an example. Furthermore, if the underlying mobile agent middleware supports multi-hop routing (e.g., geographic routing) such as Agilla [19], this solution even more robust since an agent can migrate to any node in the WSN.

After following and adapting their routes bypassing all unreachable nodes, mobile agents return to the collection point with the collected or aggregated data from the nodes they managed to visit. Although mobile agents currently only adapt to individual network failures, they could also adapt to application specific events. For example, on a fire scenario, agents could adapt their routes either to advance towards the fire, or to avoid landing on nodes that are likely to get burned.

Therefore, we have a two-phase algorithm based on an initial centralized route planning, followed by a distributed dynamic route adaptation protocol. This achieves efficient multi-mobile agent route planning with real-time guarantees. In the next section, we present the implementation of the algorithm for a WSN running Agilla [19], a mobile agent middleware, as well as the results of several experiments and simulations.

IV. Evaluation

We evaluated our system through simulations and experiments on a real WSN consisting of MICA2 motes. We used the Agilla mobile agent middleware on the motes, enabling us to define concrete $W_a(m,p)$ functions based on an actual system.

A. Implementation platform

The system architecture is depicted in Fig. 11. The GA has been implemented in Java. It provides a GUI for inputting the parameters (e.g., the source mote, the time constraint, the IP address of the WSN gateway), and has been integrated into the JADE agent platform [20]. It also allows the user to input any solution (set of paths), which enables direct comparison between solutions generated manually and by the GA. The GA is executed off-line on a powerful machine connected to the Internet. The mobile agents that are generated by the GA are then remotely injected via Java RMI into a WSN consisting of MICA2 motes. This is possible since Agilla's base station has a Java application that supports remote injections via RMI. By relying on Java RMI and the Internet, the place where the GA runs and the WSN is do not need to be the same.

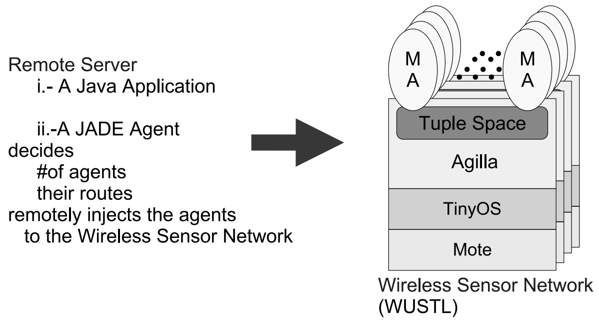

Fig. 11. The system architecture.

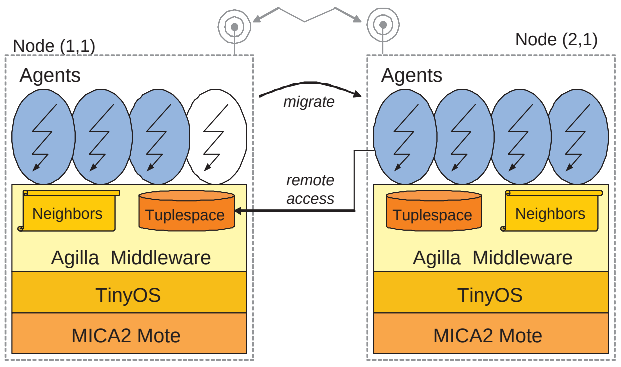

Fig. 12. The Agilla model

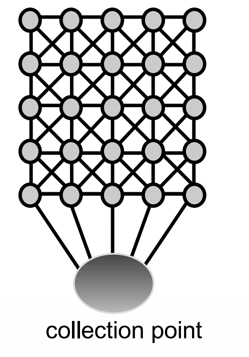

Fig. 13. Agilla network topology for a five-by-five WSN.

The MICA family of nodes is a popular WSN platform. A MICA2 node, called a “mote,” consists of a central microcontroller, radio transceiver, 512KB of flash RAM, two AA batteries (2000mA/hr), and a sensor board. The microcontroller is capable of up to 8MIPS at 12mA [7]. The transceiver throughput is a mere 40kbps (250kbps for the new MICAz motes), and its power consumption is 7mA for receiving, and 10mA for transmitting (19.7mA and 17mA for the MICAz, respectively). Thus, transmitting one byte of data is equivalent to computing hundreds of instructions, which is why minimizing agent size and number of migrations is vital.

Our Mica2 motes run Agilla [19], a virtual machine supporting mobile agents whose architecture is shown in Fig. 12. Agilla applications consist of several mobile agents that move and spread throughout the sensor network and coordinate through per-node tuplespaces. A tuplespace is a content-based shared memory that contains tuples, which are ordered lists of typed values. All agents residing on a node share the same tuplespace, but agents can also access tuplespaces remotely. Agilla embraces the spatial nature of WSNs by addressing nodes by their location. It maintains a list of one-hop neighbors on each node so agents can determine where to migrate to. Thus, when migrating or performing a remote tuplespace operation, the agent specifies the destination as a location. Agilla agents are written using a high-level byte code language. A modified stack architecture is used to limit the size of most instructions to a single byte (some instructions are 3 bytes – two bytes for a parameter). A full listing of Agilla’s instructions is available at [21].

We configured the network topology as a 5x5 grid, where the lower-left corner is at coordinate (1,1), and the collection point is one-hop away from all nodes in the first row, as shown in Fig. 13. The (x, y) location of each node corresponds to the (column,row) it is placed in within the grid.

Two basic types of agents are used throughout our simulations and experiments. The first type follows

```
01: SETUP  pushc -1
02          pushloc 0 0 //push route into the stack
03:         pushloc 1 1
04:         pushloc 1 2
05:         pushloc 1 1
06:         pushloc 0 0
07: LAND    pushc 2     //green LED on
08:         putled
09:         pushc 2
10:         sleep       //sleep 2/8 of a sec
11:         copy
12:         pushc -1
13:         ceq
14:         rjumpc BYE
15:         smove       //strongly move
16:         rjump LAND
17: BYE     halt
```

Fig. 14. Agilla mobile agent that sets the green LED at every mote.

its given route and sets the green LED on at every node it visits. Example code is shown in Fig. 14. The first six lines store the route into the operand stack. In this particular example, the agent route is $(0,0) \rightarrow (1,1) \rightarrow (1,2) \rightarrow (1,1) \rightarrow (0,0)$. This block of code is executed only once at the beginning. Lines 7-16 contain the code the agent iterates over. Every time the agent lands on a mote, it turns on the green LED, and sleeps for $.25s$ before moving to the next location on the path. If the agent fails to move, it attempts to move to the location after the next one. Note that if this location is multiple hops away, Agilla will attempt to migrate the agent using greedy geographic forwarding. This process is repeated until the value "-1" is on top of the stack. At this point, the itinerary is consumed, and the agent halts its execution. In this example, the agent always executes the same code irrespective whether it successfully migrates to the next node. More sophisticated agents that change behavior when it detects a failure may be used.

The second type of agent is the same as the first, except it uses the local tuplespace to remember how many times a node has been visited. Whenever an agent arrives, it increments a tuple that stores a counter. The lower-three bits of this counter are then displayed on the mote's LEDs.

Table I contains the $W_a(m,p)$ functions of each type of mobile agent. Note that, depending on what the application wants to do at each node, the agent may be pure aggregation, or pure collection. $W_a(m,p)$ depends on the fixed code size, and the variable data size. The fixed code size for the agents that turn on the green LED is 11 bytes, while the counting agents take 32 bytes. We use the notation $Ord(m,p)$ to denote the number of nodes located before node m in path p. Using this notation, the data size for *Pure Aggregation* agents is $(5(|p| - Ord(m,p)) + 2)$ bytes, while the data size for *Pure Collection* agents is $(5(|p| - Ord(m,p)) + 7 \cdot Ord(m,p))$ bytes. In Agilla, a location is 5 bytes, while a sensor reading is 2 bytes. This is why the *Pure Collection* agent increases its size by $(5(|p| - Ord(m,p)) + 7 \cdot Ord(m,p))$ at every hop; a five-byte location is discarded on the itinerary-data structure (the operand stack), and two byte sensor reading plus five byte location are added to the data memory.

B. Simulations

To evaluate the performance of the GA, we performed several simulations whose parameters are shown in Table II. In these simulations, we determined the cost distribution of the solutions with and without applying random mutations and crossings. The results, shown in Fig. 15 and 16, show that random mutations and crossings are clearly necessary. Fig. 15 shows that the probabilities of randomly generating both the optimal and bad solution are very low. Fortunately, most randomly solutions fall nearer to the good solutions than to the bad solutions. This suggests that a random solution will be closer to the best

TABLE I

$W_a(m,p)$ IN BYTES FOR FOUR DIFFERENT AGILLA MOBILE AGENTS.

Pure Aggregation	
Green LED on	$11 + 5(\|p\| - Ord(m,p)) + 2$
Visits counter	$32 + 5(\|p\| - Ord(m,p)) + 2$
Pure Collection	
Green LED on	$11 + 5(\|p\| - Ord(m,p)) + 7 \cdot Ord(m,p)$
Visits counter	$32 + 5(\|p\| - Ord(m,p)) + 7 \cdot Ord(m,p)$

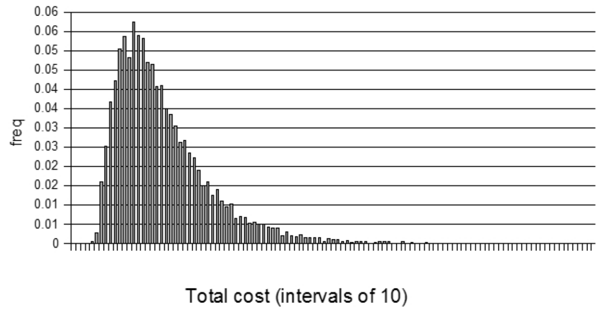

Fig. 15. Solution distribution with no mutation and crossing. Based on 10001 explored solutions.

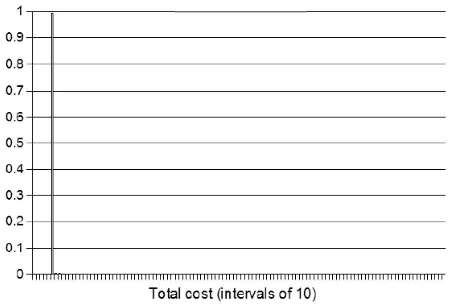

Fig. 16. Solution distribution with mutation and crossing. Based on 11411 explored solutions.

than to the worst. However, Fig. 16 shows that using the mutation and crossing operators significantly increases the probability of getting a solution close to the optimal. By using mutations and crossovers, fewer solutions need to be explored to probabilistically achieve the same solution quality.

We now evaluate the performance of the GA using various amounts of mutation and crossovers. The experiments range from being *totally random* (e.g., no mutation and crossovers) to *very aggressive* (20%-PD mutation and crossover, and 99% Selection). The parameters are shown in Table II. Furthermore, we compared the GA solutions to a solution to the multi-traveling salesmen problem (TSP). Recall that the multi-TSP does not allow a node to be visited more than once. The specific solution we used is shown in Fig. 17.

The simulation results are shown in Fig. 18 and 19. Fig. 18 plots the cost of the best solution found at

TABLE II

CONFIGURATION OF THE SIMULATIONS .

	Fig. 15	**Fig. 16**	**GA_noNew_Conservative**	**GA_Conservative**	**Random**	**GA_VeryAggressive**
Mutation	No	20%PD	20%PD	20%PD	No	20%PD
Crossing	No	20%PD	20%PD	20%PD	No	20%PD
Selection	No	No	20%	20%	No	99%
Initial population	9999	10	25	25	25	25
Iterations	1	20	54	54	54	54
New individuals	2	2	0	5	5	5
Time constraint t	22	22	22	22	22	22
Network size	5x5	5x5	5x5	5x5	5x5	5x5

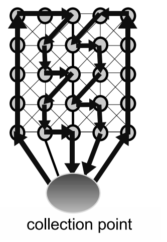

Fig. 17. A multi-TSP solution.

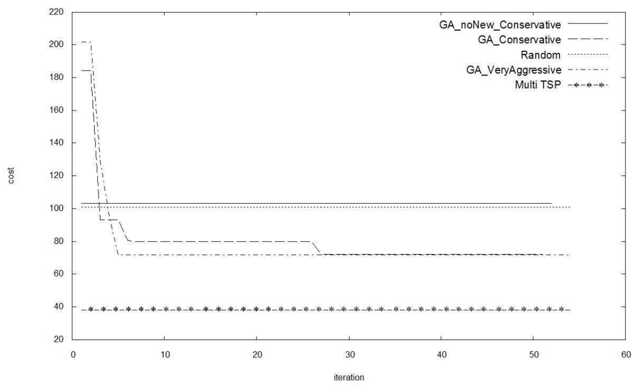

Fig. 18. Cost vs iteration.

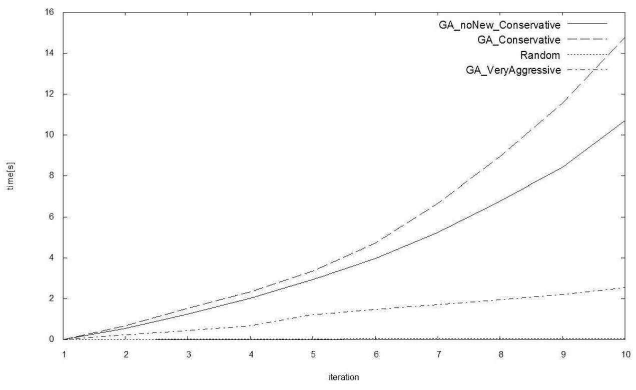

Fig. 19. Time vs iteration.

each iteration, while Fig. 19 plots the time at which each iteration began. Since the multi-TSP solution does not involve any iterations, we plot it as a constant. The results show that the maximum number of disjoint subregions is bounded by $\lfloor \frac{|CollectionPointNeighbors|}{2} \rfloor$. As discussed in Section III, the GA can achieve lower time deadlines than a multi-TSP solution by allowing paths to overlap. However, since the solution space is large, when the time constraint can be met by a multi-TSP solution, the GA solution may be worse than a multi-TSP solution, as depicted in Fig. 18. Fig. 18 also suggests that there is no gain when repeatedly applying the crossover and mutation operators on the same seed of random solutions. Adding randomly individuals at each iteration makes it possible for the solution to improve with each iteration. In fact, at every iteration, the best solutions found and the random individuals added at the end of the previous iteration are enough to keep the algorithm converging towards the optimal solution. All the other solutions can be discarded, which saves memory and time, as shown in Fig. 19. Comparatively, not using the crossover and mutation operators takes a very small amount of time per iteration (around $12ms$ vs. several minutes). However, to find a solution as good as one found by applying the crossover and mutation operators, more random solutions may need to be generated. This consumes more memory and is less predictable. Lastly, we note that the multi-TSP produces a solution with lower cost than the GA in all cases. However, as noted in Section II, there are situations when the time constraint is so tight that the multi-TSP algorithm simply cannot find a solution to but the GA can.

C. Experiments

The GA and the in-network route adaptation protocol have been tested on a real five-by-five WSN consisting of 25 MICA2 motes running Agilla. In these experiments, we measured the actual amount of time agent migration takes. If all the agents are injected at the WSN at the same time, given a time constraint t and the seconds, h, it takes to migrate an agent one hop, the real time rt it takes to get the agents back at the collection point should be:

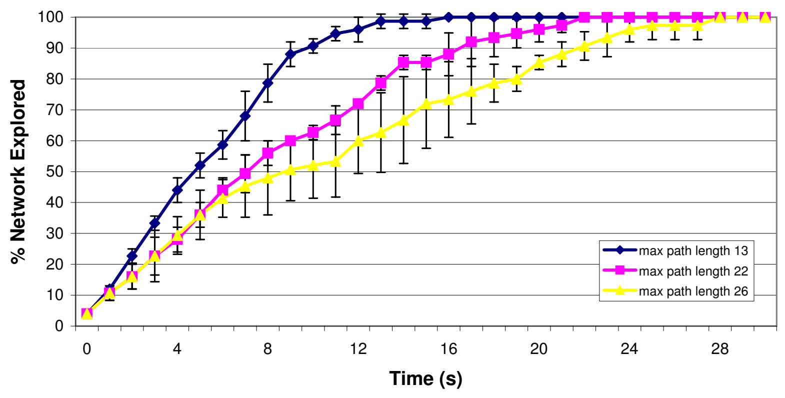

Fig. 20. Percent Network Explored vs. Time.

TABLE III
EMPIRICAL TIME AGENTS SPENT IN THE WSN FOR DIFFERENT TIME CONSTRAINTS.

time constraint (max hops/path)	**mean (time in WSN)**	**var (time in WSN)**
t=13	14 s	1 s^2
t=22	22.33 s	0.33 s^2
t=26	27.667 s	9.33 s^2

$$rt \leq (t-1) \cdot h. \tag{4}$$

And the total real time rtW the agents spent in the WSN (i.e. outside the collection point) is:

$$rtW \leq (t-3) \cdot h. \tag{5}$$

However, due to memory constraints, all of the agents cannot be injected simultaneously. If they were, the node connected to the collection point would suffer a memory overflow error. Thus, our implementation waits $850ms$ before injection to allow the previously injected agent to migrate away before injecting the next agent. Given this, the total real time, rt', it takes to get the agents back at the collection point is:

$$rt < rt' \leq (t-1) \cdot h + (\#agents) \cdot 0.850. \tag{6}$$

And the total real time, rtW', the agents spend in the WSN is:

$$rtW \leq rtW' \leq (t-3) \cdot h + (\#agents) \cdot 0.850. \tag{7}$$

We ran three different experiments with $t = 13, 22$, and 26. The GA executed in 20%-PseudoDeterministic mode for Crossing and Mutation. The GA generated twenty-five initial solutions, it executed for five iterations, and it added five new solutions every iteration thereafter. Each experiment was run three times. For each run, we used a Sony DCR-TVR18 digital camcorder to record what happened. The camcorder records video at 30fps with enough color resolution to tell which LEDs lit up. By analyzing the video off-line, we were able to determine how much of the network had been explored. For each experiment, we averaged the three runs and plotted the percent of the network explored versus time, as shown in Fig. 20, and summarized in Table III.

As expected, Fig. 20 shows that the genetic algorithm produces faster results with less variance when the max path length is shorter. With a time constraint of $t = 13, 22$, and 26, mobile agents spent in average

14s, 22.33s, and 24s in the WSN, respectively. From equation 7, the real time it takes to move an agent one hop is $1.01s$. Yet, according to the equation 5, it takes an average of $1.233s$. Since the user does not know *a priori* the number of paths, the conservative latter measurement should be used when translating the time constraint from the maximum number of nodes per path into seconds, even if it is slightly less accurate in this particular case. For example, if the network must be explored in 12.33 seconds, no path can exceed 10 hops.

V. Future Work

There are many areas of future work. One area of particular interest is on extending our algorithm to support multiple collection points that move. Our current algorithm only works with WSN with a single collection point that does not move. There are many applications that involve multiple users that move. For example, several fire fighters may be walking through the network carrying PDAs that inject network exploration agents into the network. These agents need to be able to explore the network and return to their respective fire fighters, who have moved since the agents were injected.

Supporting multiple static collection points requires minimal changes to the genetic algorithm. Instead of calculating the results over a single source and destination, they will be calculated over a set of sources and destinations. What remains to be seen, however, is the quality of the results and additional computational time required to calculate these results.

Enhancing the genetic algorithm to support mobile collection points is not difficult, especially if the motion of the collection point is predictable. If the motion profile of the collection point is known, the destination of each route is simply the location of the collection point based on the motion profile. Note that the real-time guarantees already provided by the genetic algorithm ensures that the agent will reach the collection point, provided the motion profile is accurate. To account for possible inaccuracies of the collection point's motion profile, additional intelligence must be added to the agent. Consider the case where an agent arrives at its collection point, but the collection point is no longer there. One possible solution to address this issue is for the collection point to leave a "scent" that the agent can follow. If the scent decays over time, the agent can find the collection point by following the direction in which the scent grows stronger.

Another area in which the network exploration algorithm can be improved involves the use of agent cloning. The current algorithm produces a fixed set of agents, each with a non-branching route. It is possible that allowing agents to clone, especially when a disabled node is encountered, can improve efficiency. We plan on investigating this possibility.

VI. Conclusions

Software for WSNs need to be more flexible for users to take full advantage of the benefits that the network has to offer. One approach is the use of mobile agents that can migrate carrying code and state across the nodes. New algorithms need to be developed to control agent behavior. One important problem that we have addressed in this paper is that of network exploration. Network exploration consists of visiting enough nodes in the network such that a certain density of visits has been reached. We present a genetic algorithm that determines the number of agents necessary to explore the network and their paths. It includes contingency plans to account for the dynamic nature of WSNs. We show through simulations and experiments on real motes that our mechanism is effective and that it is able to explore the network within the set deadline, while keeping the amount of energy used in check.

Acknowledgment

This research has been supported in part by the Generalitat de Catalunya, the Univ. of Girona, and the Univ. of California through a Girona fellowship, and by the US Office of Naval Research under MURI research contract N00014-02-1-0715.

REFERENCES

[1] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," in *Proc. of the ACM SenSys*, 2004.

[2] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proc. of the 1st ACM Workshop on Wireless Sensor Networks and Applications*, September 2002.

[3] http://fire.me.berkeley.edu/.

[4] http://www.intel.com/technology/techresearch/research/rs01031.htm.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104. [Online]. Available: citeseer.ist.psu.edu/382595.html

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," 2003. [Online]. Available: citeseer.ist.psu.edu/gay03nesc.html

[7] http://www.xbow.com.

[8] C.-L. Fok, G.-C. Roman, and C. Lu, "Mobile agent middleware for sensor networks: An application case study," Washington University in St. Louis Department of Computer Science and Engineering, Tech. Rep. WUCSE-04-73, 2004. [Online]. Available: mobilab.wustl.edu/projects/agilla

[9] D. B. Lange and M. Oshima, "Seven good reasons for mobile agents," *Commun. ACM*, vol. 42, no. 3, pp. 88–89, 1999.

[10] X. J. Long Gan, Jiming Liu, "Agent-based, energy efficient routing in sensor networks," in *AAMAS'04*, July 2004.

[11] Q. Wu, N. S. V. Rao, J. Barhen, S. S. Iyengar, V. K. Vaishnavi, H. Qi, and K. Chakrabarty, "On computing mobile agent routes for data fusion in distributed sensor networks," *IEEE Trans. Knowledge Data Eng.*, vol. 16, no. 6, jun 2004.

[12] W. Zhang, *State-Space Search: Algorithms, Complexity, Extensions, and Applications*. Springer, 1999.

[13] IEEE, "Wireless lan medium access control (mac) and physical layer (phy) specifications," *IEEE Standard 802.11*, 1999.

[14] B. Chen, K. Jamieson, H. Balakrishnan, and R. Morris, "Span: An energy-efficient coordination algorithm for topology maintenance in ad hoc wireless networks," in *Mobile Computing and Networking*, 2001, pp. 85–96.

[15] W. Ye, J. Heidemann, and D. Estrin, "An energy-efficient mac protocol for wireless sensor networks," in *Proceeding of Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies*, 2002, pp. 1567– 1576.

[16] T. van Dam and K. Langendoen, "An adaptive energy-efficient mac protocol for wireless sensor networks," in *Proc. of SenSys 2003*, 2003.

[17] J. Zheng and M. J. Lee, "Will ieee 802.15.4 make ubiquitous networking a reality?: A discussion on a potential low power, low bit rate standard," *IEEE Commun. Mag.*, vol. 42, no. 6, jun 2004.

[18] D. P. Bertsekas, *Network Optimization: Continuous and Discrete Models*. Athena Scientific, 1998.

[19] C.-L. Fok, G.-C. Roman, and C. Lu, "Rapid development and flexible deployment of adaptive wireless sensor network applications," Washington University in St. Louis Department of Computer Science and Engineering, Tech. Rep. WUCSE-04-59, 2004. [Online]. Available: mobilab.wustl.edu/projects/agilla

[20] (2004, Nov.) Jade java agent development framework. [Online]. Available: http://jade.tilab.com

[21] http://mobilab.wustl.edu/projects/agilla/.