Design, Implementation and Case Study of WISEMAN: WIreless Sensors Employing Mobile AgeNts

Sergio González-Valenzuela, Min Chen, and Victor C.M. Leung

Department of Electrical and Computer Engineering
The University of British Columbia
2332 Main Mall, Vancouver BC, V6Z1T4, Canada
{sergiog,minchen,vleung}@ece.ubc.ca

Abstract. We describe the practical implementation of Wiseman: our proposed scheme for running mobile agents in Wireless Sensor Networks. Wiseman's architecture derives from a much earlier agent system originally conceived for distributed process coordination in wired networks. Given the memory constraints associated with small sensor devices, we revised the architecture of the original agent system to make it applicable to this type of networks. Agents are programmed as compact text scripts that are interpreted at the sensor nodes. Wiseman is currently implemented in TinyOS ver. 1, its binary image occupies 19Kbytes of ROM memory, and it occupies 3Kbytes of RAM to operate. We describe the rationale behind Wiseman's interpreter architecture and unique programming features that can help reduce packet overhead in sensor networks. In addition, we gauge the proposed system's efficiency in terms of task duration with different network topologies through a case study that involves an early-fire-detection application in a fictitious forest setting.

Keywords: Mobile agents, wireless sensor networks, performance evaluation.

1 Introduction

The topic of Wireless Sensor Networks (WSN) continues to draw significant research interest at present. These investigations include studies on the feasibility of dynamically re-tasking sensor nodes in the face of continuous changes in the underlying environment. To tackle this issue, code mobility can be used as a potentially efficient approach [1]. While some of these approaches have been proposed for operation over devices with plentiful hardware resources, we are rather interested in the type of agent-based re-tasking systems targeted at low-end sensor devices that are characterized by severe memory and processing constraints. The main motivation for studying WSN agent-based re-tasking is that it enables rapid modifications to the pre-programmed behaviour of sensor nodes responding to a predefined type of events. Clearly, dynamic WSN re-tasking provides flexibility and convenience to its operators. Still, further studies are needed to determine whether this mechanism is not only practicable from an engineering perspective, but also whether the system that implements it fulfills performance expectations in terms of resources employed and response time.

Initial accomplishments in the area of WSN re-tasking were achieved by schemes such as Deluge [2], Impala [3] and Maté [4]. These approaches introduced basic forms of code mobility to reprogram WSN nodes with a certain degree of flexibility and were soon followed by enhanced approaches. For instance, Agilla is a popular scheme that employs code mobility in the form of programmable agents to perform WSN re-tasking too [5]. The Agilla framework introduced significant improvements over existing proposals by employing a more robust agent interpreter capable of handling certain tasks in devices severely constrained in hardware resources. Several lessons were learned during the implementation and testing of Agilla, including issues related to memory management and the agents' programming language [6].

In a previous paper, we contested that middleware design for agent systems in WSN is highly dependent on several issues, which includes (1) the targeted system application and (2) the network navigation methodology implemented into the mobile codes to address the issue in question [7]. In addition, the use of mobile codes in WSNs is mostly warranted when collecting and/or aggregating data that is dispersed in various nodes. To this regard, we have previously proposed MAWSN and MADD as itinerary-planning schemes aimed at reducing the overhead incurred by the corresponding process [8, 9]. One shortcoming with these and other proposed paradigms is that once the agent is dispatched by the sink node, its itinerary cannot be revised, which becomes an issue should the underlying environment conditions change after the agent has been dispatched [10]. By the same token, if on-the-go itinerary changes are needed, then this functionality needs to be supported by the agent middleware.

To address the previous issue, we designed and implemented Wiseman: "WIreless Sensors Employing Mobile AgeNts". Our proposed scheme overcomes the source-based itinerary issue, enabling changes by any WSN node in the pre-defined path assigned to an agent. Our novel agent middleware system incorporates: (1) a high-level text-based language system that acts as a compact action script, (2) the ability to dynamically modify agent itineraries that can employ hop-by-hop planning, and (3) a virtual-link navigation capability that mimics multicast routing through labelled paths. A combination of the latter 2 schemes is also possible. In addition to this, Wiseman is based on a framework that does not necessitate a program counter and execution stack to function. Instead, Wiseman incorporates a self-depleting command execution model that discards agent instructions once they are no longer needed. This implies that the agent size is variable, and may be structured to progressively shrink as it completes its task.

The rest of the paper is organized as follows: In Section 2, we briefly introduce the original predecessor of the Wiseman system and describe its adaptation into a middleware system amenable to WSN use. Section 3 describes Wiseman's simplified architecture and main system modules. In Section 4, we describe Wiseman's language construct design based on the needs of the overall system. Practical implementation aspects of the system are discussed in Section 5. In Section 6, we illustrate a sample application scenario to evaluate the performance of the system in terms of agent migration delay. In Section 7, we discuss the results of our experiments. Finally, we present our conclusions in Section 8.

2. System Foundations

Existing mobile agent approaches for WSN attempt to achieve a balance between the degrees of functionality incorporated into the actual code interpreter, and the one provided to the agents. On the one hand, a coarse-grained agent system for WSN that incorporates a high degree of functionality in the interpreter requires simple constructs on the agent side in order to accomplish a certain task (e.g., <*run task A*>, <*run task B*>, <*end*>). On the other hand, a code interpreter tailored for fine-grained agent language construct leads to larger programs that describe in detail the task to perform (e.g., <*mov 1 x*>, <*and x 0xFB*>, …) Once again, the degree of granularity used in the language constructs of the agent system should be a direct function of the intended WSN's application. Given the specific nature of WSNs, it results intuitive to think of coarse-grained language constructs as a more suitable approach. In other words, it makes little sense here to provide agents with excessive control of the node's data processing functionalities if the tasks to be performed are consistently repetitive. In fact, the case for code mobility in the form of agents hardly holds if the WSNs tasks they are set to solve are rather deterministic, or if they require minimal changes. The use of mobile agents in a WSN is therefore justified by the need of flexibility in the evolution of a system process. In such case, whereas the application of a WSN might be very well defined, external factors driven by the underlying environment might require different strategies to deal with the problem. For this reason, we propose an alternative approach to incorporate programmability in distributed tasks based on a simplified version of the Wave system for incorporation into WSN.

The *Wave* system can be considered one of the earliest precursors of code mobility in data networks, with its foundations lying on the idea of efficient task coordination in distributed environments [11, 12]. To this effect, Wave's high-level language construct allows creating highly compact programs that encompass a suitable degree of distributed coordination. This approach results highly appealing to WSNs, since it promotes the use of existing functionalities in the node, instead of creating agents that repeatedly perform the same task on every node they visit. In fact, overall system efficiency can arguably be improved by promoting local data processing through algorithms that run on native code. As a result: (1) the process coordination part of the distributed application is decoupled from the data processing part and is now left for the agents to perform; (2) additional agent compactness can be achieved by defining language constructs that are sufficient to describe the desired coordination methodology of the mobile process; and (3) a condensed language construct translates into simplified interpreter's architecture, smaller memory footprint, and reduced forwarding overhead in terms of delay and bandwidth. In the next section we describe Wiseman's architecture as a significantly simplified adaptation of the original Wave system.

3 The Wiseman System Architecture

The Wiseman interpreter is comprised of an incoming agent queue, a code parser, a processor block, and an agent dispatcher, as shown in Fig. 1. The incoming agent queue works in a simplistic first-in-first-out fashion, and accepts agents either arriving from other nodes, or those injected at the local node. The parser is in charge of processing agents as they are received from the wireless interface of the node.

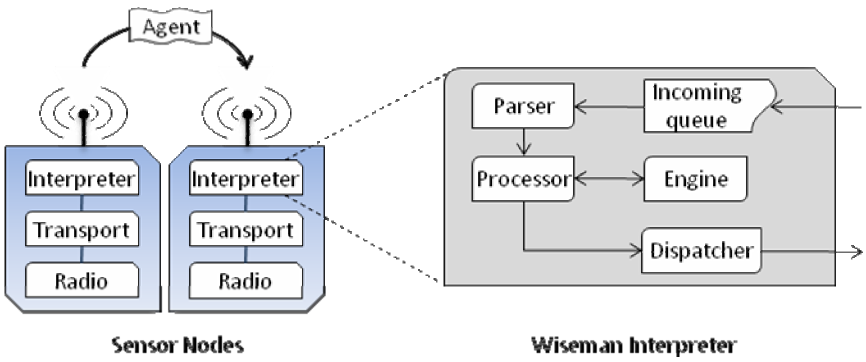

Fig. 1. Wiseman's System architecture

Agents are removed from the incoming queue one at a time and fragmented into their respective codes and data fields. Agent codes are further separated into two segments, hereto referred as *head* and *tail*. The *head* is the first code fragment defined by the language constructs, whereas the rest of the codes that follow are referred to as the *tail*. The *head* is then unwrapped from any delimiters until a single indivisible operation is found by the parser, and is subsequently passed onto the processor block for execution. When finished, the process control is returned to the parser, which then extracts the next operation from the agent's *tail* if the previous operation was successfully processed. The new segment becomes the *head* that is processed in the exact same fashion. This way of processing agents implies that the agent's size is systematically depleted as operations are being performed, and can help save forwarding time and bandwidth when hoping across the WSN.

The processor performs the operation indicated by the *head*, whose outcome is determined as Boolean value (i.e., its execution is either successfully completed, or not). This outcome is employed to make decisions that affect the subsequent execution of the distributed process, as noted before. The agent execution process is halted if any of the following conditions occur: (1) an operation yields an unsuccessful outcome, (2) an agent-hop operation is encountered, or (3) explicit process termination is indicated. In the first case, the agent's *tail* is discarded and the agent simply terminates executing, unless the *head* is contained within a language construct that instructs the parser to proceed otherwise. If the operation is successful, then the parser sends the next operation to the execution block and the process continues as defined by the agent's codes. In the second case, the agent instructs the execution block to migrate the agent to another node, and so the *tail* is sent to the dispatcher block for subsequent forwarding to another node or set of nodes. In the third case, the agent may simply instruct the interpreter to explicitly halt the current process execution as needed. The process sequence in Wiseman's architecture is shown in Fig. 2.

An agent may hop to a node or set of nodes that are associated by sharing a single-character label used as a unique identifier. If the agent's codes specify a hop through one of these labelled (virtual) paths, then the dispatcher first ensures that the hop is actually possible by looking up the corresponding label entry in a local table that contains a list of the neighbouring nodes associated to it. The agent will be dispatched

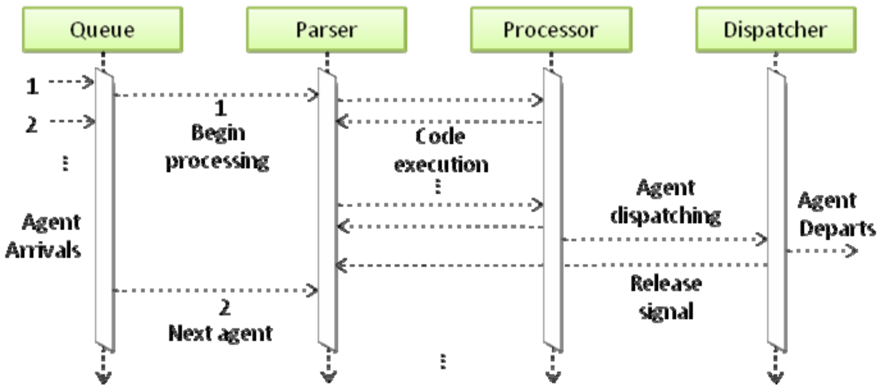

Fig. 2. Agent processing sequence in Wiseman

out of the interpreter only if the corresponding label exists in the table. Otherwise, the agent is discarded. However, the agent may also be unicast to a specific node, or broadcast if no particular destination is specified. Finally, the dispatcher signals the parser once the agent has been forwarded, so that the next agent in the incoming queue can be processed, if one exists.

Another novel feature of Wiseman is defined by its metamorphic capability. Since Wiseman agents are transmitted and processed in their raw text-string from, the architecture effectively allows the interpreter to substitute portions of the agent's code with other codes. This characteristic introduces a degree of flexibility unmatched by existing agent approaches in WSNs. For example, a target-tracking agent may initially carry and execute the corresponding code of an energy-efficient algorithm that works best for slow moving objects. However, if the object's rate of mobility increases, then the agent may instruct a WSN node to replace its target tracking algorithm with one that is more energy-demanding, but otherwise necessary to keep up with fast-moving objects.

4 Language Constructs

Wiseman's constructs are simplified versions of Wave's original operators, variables, rules, and delimiters, so that they can be employed in WSN nodes:

4.1 Variables

The Wiseman interpreter supports three kinds of fixed-type variables, all of which employ pre-assigned memory segments in the local node. The first type is coined as *Numeric*, represented by the letter *N*, and is predefined as being of floating point type. The second kind is the *Character* variable, intended for use with single characters through the letter *C*. Both of these variable types are semantically similar to public variables defined in object-oriented programming, meaning that all agents that arrive to the interpreter have access to them. In addition, agents carry with them *Mobile* variables, which are accessed through the letter *M*. Mobile variables' role resembles that of private variables in object-oriented programming. Thus, manipulation of an agent's own *Mobile* variables has no effect on other agents' *Mobile* variables. Similarly, the manipulation of *Numeric* and *Character* variables at the local node has no effect on the variables of remote nodes. However, all variables are expected to maintain their semantic meaning across the network according to how they are individually manipulated by programmers through the agents. Agents always carry with them associated *Mobile* variables, which are temporarily stored in predefined memory locations when visiting a node. The interpreter also implements the *Clipboard* variable *B* to temporarily store data. The execution environment also defines three extra *Environmental* variables. *Identity I* is a read-only variable, whose content is defined by the local node's identification number (i.e., 1, 2 …) The *Predecessor P* contains the identification number of the node where the agent being processed hopped from. Finally, the *Link* variable *L* holds the label of the virtual link through which the agent arrived from. The interpreter's dispatcher appends these variables to the agent's control field as soon as it arrives from the wireless channel.

4.2 Operators

Wiseman defines a mix of general purpose and system-specific operators. For instance, standard arithmetic operators are provided to allow simple calculations at the nodes (i.e., +, -, *, / and =). Ordinary comparison operators that return Boolean values that agents evaluate are also supported (i.e., <, <=, ==, =>, > and !=). The *hop* operator is employed to indicate that the agent needs to be forwarded either to the node specified in the right-hand side of the *#* character, or through the virtual link specified in its left-hand side that is associated with a certain subset of adjacent nodes. For the later case, this operator provides the functionality of automatically cloning the agent with as many copies as outgoing virtual links exist to the corresponding nodes. That is, if there are 3 such virtual links labelled with the letter *s*, then an identical number of agent copies will be forwarded by the dispatcher. Moreover, if a hop operation is met by the interpreter when processing the codes, then the agent is forwarded and execution resumes at the next operation where the process had been previously suspended. This functionality provides Wiseman with a basic form of *strong mobility* that does not require any form of program counter or execution state that needs to be forwarded with the agent. Alternatively, a copy of the agent may be locally broadcast to all immediate neighbours by employing the @ operator. The execution operator *$* indicates to the interpreter that a local function is to be called as specified by its left- and right-hand side parameters. The code injection operator ^ indicates the insertion of a locally stored code(s) segment into the agents structure. Finally, the halt operator *!* indicates the explicit termination of the current agent with success if the right-side operand is 1 or failure if the operand is 0.

4.3 Rules

In Wiseman, the *Repeat* rule *R* indicates that codes embraced by the corresponding delimiters will be continuously executed. However, Wiseman processes cycling codes by extracting them from the delimiters and re-inserting them before the original *Repeat* construct. Therefore, an agent segment that possesses the corresponding structure for repeating code *R{...}* yields the modified structure *...;R{...}*. This process can be repeated consecutively until a certain condition is met. In addition, *Or O* and *And A* rules are defined as a way to manipulate execution of the agent by testing whether the code embraced within square brackets yields a true or false value for every code segment it includes. Therefore, an *O[...;...;...]* construct indicates that the code segments delimited by semicolons are to be sequentially executed, stopping as soon as one of these segments results in a true value. Otherwise, the *Or* rule returns a false value and the whole agent's process stops. A similar logic applies for the *And* rule, except that all of the segments must return a true value in order for the whole construct to succeed.

4.4 Delimiters

The main delimiter employed to separate code segments is the semicolon. In addition, round, square and curly brackets are designated to delimit code segments whose execution depends on a rule construct, as explained before. Distinct types of brackets are employed since they facilitate the parser's task when tokenizing nested code segments prior to being processed (i.e., *R{...O[...(...)...]...}*). The use of a single type of bracket would have implied significantly more parsing functionalities, and therefore, added processing overhead.

5 Practical Implementation Aspects

Wiseman was initially evaluated employing the OMNeT++ Discrete Event Simulator [13] to verify the correctness of the design after undergoing significant changes from an earlier system proposal [14]. This preliminary evaluation step was crucial in streamlining its implementation over actual hardware devices given the difficulty of debugging firmware programs. After verification, Wiseman was ported to the NesC language and subsequently improved for more efficient operation over Crossbow Micaz [15]. The NesC programming language employed to code TinyOS ver. 1.1 programs is in fact a modified version of the C language [16]. We decided to employ this wireless sensor platform since it provides an ideal example of a severely-constrained hardware device type whereby our proposed system could be put to the test. In particular, these *motes* have 128Kbytes of instruction memory and 4Kbytes of data memory. Therefore, developing an agent system for this hardware platform is challenging due primarily to the limited amount of memory space. A mote attached to a Crossbow MIB510 interface board is used as a data sink node, whereas the MIB510 itself interfaces data between the WSN and a regular laptop computer. Wiseman spans approximately 2400 lines of NesC code divided into modules that reflect the system architecture shown in Fig. 1. An additional module that includes a few uncommon string manipulation functions was also incorporated as required by the parser to process agent codes. Wiseman can be currently obtained from [17].

As mentioned before, Wiseman's binary image occupies 19Kbytes and just over 3Kbytes of RAM space to operate in its current form. A good portion of the RAM space usage reflects the space reserved to manipulate agents that occupy a maximum of 170 bytes. Depending on the type of agent program created, this amount of reserved memory suffices for our experiments. However, for agents that incorporate the

Repeat rule *R*, we needed to ensure that there was enough memory space since this construct may effectively duplicate the size of the agent. For example, an agent with codes "*R{...;#1}*" yields the string "*...;#1;R{#1;...}*", or simply "*#1;R{...}*" just before the agent is set to hop to node 1. However "*R{#1;...}*" leads to "*#1;...;R{#1;...}*", nearly doubling the agent code's size before being forwarded to node 1. It is also evident that agents can be larger than the data payload of the Zigbee packets used by the Micaz to communicate with other nodes. Consequently, we implemented a simple data forwarding mechanism that follows the signalling sequence illustrated in Fig. 3.

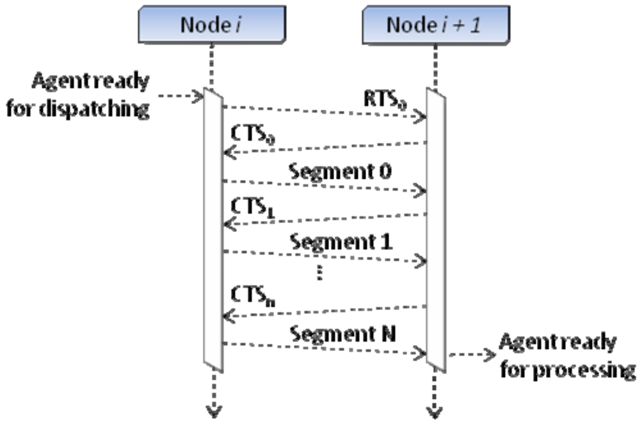

Fig. 3. Forwarding sequence of Wiseman agent segments

As seen here, every node wishing to forward an agent first must request permission to initiate the agent migration process by sending a *Request-To-Send* (RTS) packet to the intended destination node to ensure that it is currently available, since it might possibly be involved in another agent forwarding process with a different node. Subsequent permission to send packets is granted upon receiving a *Clear-To-Send* (CTS) acknowledgement from the destination. These session initiation and control packets are comprised by three fields, which include: a source node ID number, a randomly generated session number, and the current segment number. CTS packets always indicate the segment number the target node is expecting next. Any discrepancy between the expected segment number, the session number, or the source node identifier resets the process and all of the previous segments are discarded. This procedure is done to ensure agent forwarding correctness and to avoid possible confusions with packets that may arrive from other nodes. A timeout process that expires after 300mS is always initiated at the sender's side, and a maximum of 3 transmission attempts may take place. If unsuccessful, the interpreter first discards the current agent and then attempts to transmit the next agent in the outgoing agent queue, which may be destined to a different node. To this regard, two different agent queues are kept: one that holds a maximum of 3 incoming agents, and one for a maximum of 5 outgoing agents. This signifies that the interpreter may receive and queue up to 3 agents for future processing if it is currently busy with another agent. If the incoming queue is full, then additional RTS signals received are left unanswered. In addition to this, since the receiving end implements no timeout procedure during an agent forwarding process, if the current forwarding process fails, then the receiver will keep its last session values. Later, when another node attempts to begin transmissions, the old values at the receiver will trigger an immediate failure, and the current session will be reset. In this case, the sender will initiate the forwarding packets until the second attempt 300mS later. Agents are always forwarded according to the values received as parameters in a *hop* operation (#). Currently, no routing functionalities are incorporated into the system. Instead, the WSN operator may create virtual multicast trees by explicitly labelling links between nodes. The benefits of this approach will become apparent in the next section, which explains a sample deployment scenario where Wiseman can be conceivably employed.

6 An Example Application: Early Detection of Forest Fires

6.1 Rationale

As an illustrative example to exhibit the capability of Wiseman system, we consider an application in the prevention of forest fires. Forest fires, also known as wild fires, are often uncontrolled events that occur in natural settings and cause significant damage to both the environment, and to man-made infrastructure. Glitho et al. [18] have already addressed the weather monitoring issue, and Fok et al. [5] have addressed the issue of tracking fire while it is spreading. By comparison, this application study considers early detection/prevention of an unwanted event, since this can reduce damage, and perhaps even human and wild-life casualties. Forest fires usually happen when two main conditions meet simultaneously: high temperature and low humidity. As exemplified in Fig. 4, a fictitious forest area is separated into three sections: A, B and C. We propose realizing early detection of forest fires by dividing the task into two stages. In the first stage, we collect temperatures for each forest section, in which the corresponding temperature sensor node (e.g., 1, 2 or 3 in Fig. 4) is regarded as the cluster-head. If the temperature is higher than a certain critical threshold, then

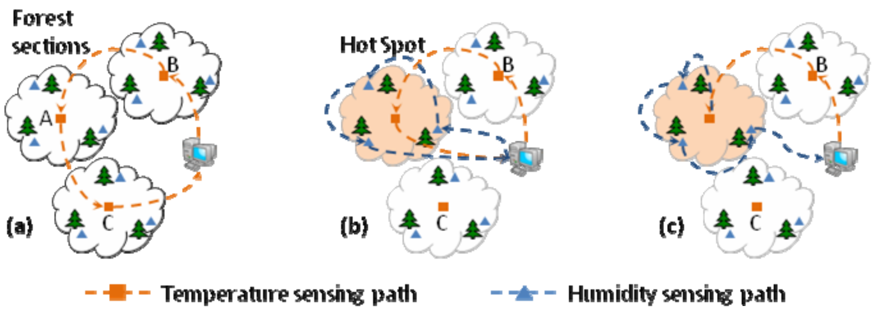

Fig. 4. Wiseman for early detection of forest fires: (a) routine temperature checking; (b) primary temperature monitoring task; (c) secondary humidity monitoring task

the second stage of the task is triggered. In this second stage, the humidity readings of the corresponding forest section are collected. If one of these readings is smaller than a certain threshold, then an alert signals is raised to inform personnel of the current hazardous situation.

During normal circumstances, a mobile agent will collect temperature for each monitored area in the path, as shown in Fig. 4 (a). The agent's migration itinerary for the first stage is planned according to the priority of different forest areas. In the case shown in Fig. 4, forest area A has the highest priority, which means the trees in this area are the easiest to catch fire, and/or that such tree species might be the most precious.

Occasionally, an abnormal reading may be detected by the mobile agent, which means that the temperature in a certain forest area exceeds a certain critical threshold. The forest section with abnormal temperature/humidity readings is circled, as shown in Fig. 4 (b) and (c). At this moment, there are three schemes that can be employed to initiate the task of stage 2:

Scheme A**:** The current agent does not know how to handle the special event since there is no corresponding action script carried with it. Thus, the agent returns to the sink node immediately, which in turn dispatches the second agent whose task is to obtain the minimum humidity reading.

Scheme B**:** The current agent already carries the action script to handle the emergent case. Thus, the agent will perform the task for stage 2 starting from node 2, as shown in **Fig. 4**(c).

Scheme C**:** The current agent does not carry the processing code to handle the special case. Instead, it retrieves an action script already stored in node 2, and it replaces the old action script with the new one. This strategy enables the agent to obtain the mobile codes that handle the emergent case locally. The agent itinerary is the same as that of *Scheme B*. However, the energy consumption incurred while otherwise transmitting the action script portion that is only useful when handling the emergent event is avoided, and the corresponding agent migration delay is decreased as well.

Note that in our previous agent-based approaches [8, 9] only *Scheme A* and *Scheme B* were supported. This is because the itinerary must be planned by the sink node in advance. Thus, two agents are dispatched in *Scheme A* that individually carry the possible itineraries for the temperature checking path, and for the emergent humidity data collecting task. By comparison, Wiseman enables changes on demand (e.g., at node 2 in Fig. 4) in the pre-defined path assigned to an agent, as seen in *Scheme C*.

6.2 Experimental Setup

Fig. 5 shows the four topologies used in our experiments. Compared to the scenario shown in Fig. 4, we incorporate humidity sensor nodes in the hot spot assigned to forest section 2 of Fig. 5(a), and omit them in the other forest sections (1 and 3). In this sample scenario, node 2 is the cluster-head of the area comprised by sensor nodes 6, 7, 8 and 9. In Fig. 5(b), we add two more humidity sensor nodes to the hot spot in order to depict a larger forest area, whereas topology 3 in Fig. 5(c) adds two extra temperature sensor nodes compared to topology 1. Finally, we add two more humidity sensor nodes in Fig. 5(d) (compared with topology 3).

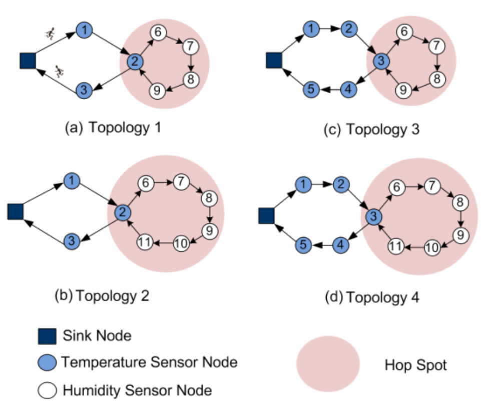

Fig. 5. Topologies employed for our experiments

Table 1 shows the agents are that were employed to set up the environment that the agents in Table 2 will encounter in accordance to our experiment setting for Schemes B and C. In these cases, the task of the agents in Table 1 is to set up labelled paths for subsequent agent navigation, creating the type of virtual links previously described. Links in the main temperature-reading circuit are labelled with the character *a* by employing the corresponding link assignment operation *L=a* before migrating to another node, whereas links in the humidity-reading circuit are labelled with the character *b*. Once these virtual paths are set, other agents can use them as they traverse the network. In addition, agents toggle-on the green LEDs as a visual aid to verify their itinerary through the WSN by means of the *l$n* operation. In this operation, character *l* on the left-hand side signifies an LED operation, and character *n* on the right-hand side signifies that the green LED is toggled-on. It can also be seen that the interpreter's Clipboard *B* is set to a fictitious value of 45 at either node 2 or 3 depending to the current topology being employed, which signifies that the temperature at that particular node has reached the specified value. Finally, the numbers on the right-hand side of the hop operator indicate the identity of the node to which the agent is set to migrate next (e.g., #1 migrates the agent to node 1). In accordance to the way agents are processed, all operations that have been already executed are removed from the agent's code, and only the trailing operations are migrated (i.e., the *tail* of the agent, as described in Section 3).

Table 1. Environment-setting agents

Topology	Agent Script for Itinerary Labelling
1	l$n;L=a;#1;l$n;#2;l$n;B=45;L=b;#6;l$n;#7;l$n;#8;l$n;#9;l$n;#2;L=a;#3;l$n;#0;l$n
2	l$n;L=a;#1;l$n;#2;l$n;B=45;L=b;#6;l$n;#7;l$n;#8;l$n;#9;l$n;#10;l$n;#11;l$n;#2;L=a;#3;l$n;#0;l$n
3	l$n;L=a;#1;l$n;#2;l$n;#3;l$n;B=45;L=b;#6;l$n;#7;l$n;#8;l$n;#9;l$n;#3;L=a;#4;l$n;#5;l$n;#0;l$n
4	l$n;L=a;#1;l$n;#2;l$n;#3;l$n;B=45;L=b;#6;l$n;#7;l$n;#8;l$n;#9;l$n;#10;l$n;#11;l$n;#3;L=a;#4;l$n;#5;l$n;#0;l$n

Table 2. Agents that implement distinct migration strategies

Scheme	Agent	Script
A	1	l$n;M0=1;R{#M0;I!=0;M0+1;l$n;r$t;O[B<40;M1=I];O[M0<6;M0=0]}
	2	l$d;#1;#2;#3;M0=6;R{#M0;M0+1;l$d;I!=0;O[(I==9;M0=3);(I==5;M0=0);!1]; r$h;O[B>20;M1=I]}
B	1	a#;R{l$w;I!=0;O[(I<4;r$t;B>40;M2<1;M2=1;M0=I;b#);(I>3;r$h;B<20;M1=I;!0) ;a#;b#]}
C	1	R{a#;l$w;I!=0;O[(B>40;M0=I;2^0);!1]}
	2	b#;R{l$d;I!=0;O[(I>5;r$h;B<20;M0=I;!0);a#;b#]}

According to our experiment setup, *Agent 1* (59 bytes long) in Scheme A explores the cluster-head circuit comprised by nodes 1-3 (topologies 1 and 2 in Fig. 5) or 1-5 (for topologies 3 and 4 in Fig. 5). Since this scheme does not rely on virtual links, the value of mobile variable *M0* is sequentially incremented (*M0+1*), and is then employed to determine the next agent hop destination once the current *repeat* rule cycles (#M0). Upon reaching the corresponding node, the agent toggles the green LED and reads the locally sensed temperature (*l$n;r$t*). If the temperature is less than the specified threshold (*B<40*), then the execution thread continues at the second *Or* rule. Otherwise, the ID of the local node is stored in mobile variable M1 (*M1=I*) to be returned to the sink node. Finally, the value of variable M0 will be set to 0 at the end of the itinerary, and the *I!=0* operation will ensure that *Agent 1* terminates when it gets back to the sink (ID 0). On the other hand, *Agent 2* is dispatched in response to the value in *M1* brought by Agent 1. It can be seen that the initial itinerary of *Agent 2* (85 bytes long) is set deterministically for topology 2. Here, the agent enters the humidity-sensing circuit at node 3, traversing nodes 6 through 9, and exiting back to node 3 as indicated by the value in M0, which is in turn modified by the preceding operations (*O[(I==9;M0=3);(I==5; M0=0);!1*)]. The sequence of events is similar as before, and M1 will be set to the value of the current node's ID if the humidity reading exceeds a predefined threshold (*r$h;O[B>20;M1=I]*).

In contrast to *Scheme A*, *Scheme B* relies on a virtual path previously set by a preceding agent (according to the current topology, as shown in Table 1). In this particular experiment, the path-setting agent needs to be executed only once for all subsequent agents to use. We can see that the chain of operations is fairly similar to those agents in *Scheme A*, with the main difference that hoping is made through virtual links labeled with letter *a* for the main temperature-reading circuit, and with letter *b* for the humidity-sensing circuit. To this regard, the label identifier is set on the left-hand side of the hop operator (i.e., *a#*, or *b#*), and the hoping sequence within these circuits is controlled by the *Or* rules. Consequently, a single agent (79 bytes long) is needed for this scheme. Finally, *Scheme C*, employs the local injection operator (^) that is used by *Agent 1* (36 bytes long) when the temperature reading exceeds the predefined threshold. Therefore, *Agent 1* does not need to carry the associated code that specifies when to switch its navigation path to labeled circuits *a* or *b* when needed. Instead, a second agent (id = 0) will be injected with a 2-second delay (*2^0)* to the humidity-sensing circuit from the node whose temperature value exceeded the corresponding threshold. Thus, *Agent B* (46 bytes long) needs to be already available at predefined

WSN nodes. Each of these approaches addresses the event-prevention objective proposed before, and they have their own advantages and disadvantages as evidenced by the results obtained through the experiments that we discuss next.

7 Experiment Results

We have conducted experiments for the four topologies described in Section 6.2. For each topology, this section will evaluate the performance of *Schemes A, B* and *C* in terms of task duration and incurred packet overhead. The task duration (or itinerary completion delay) indicates the time at which the sink node dispatched the first agent to the time when the task of the last stage is finished.

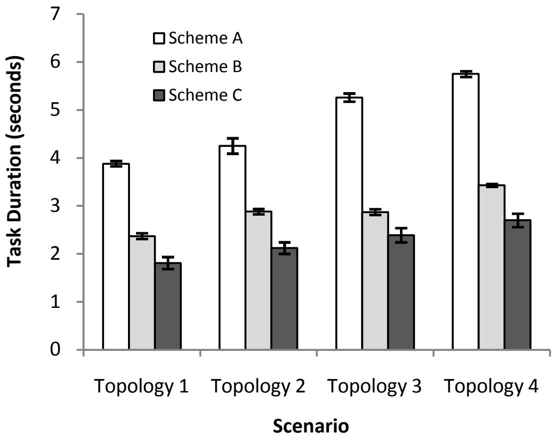

Fig. 6. Itinerary completion delay

As shown in Fig. 6, for each scheme, the task duration is the lowest in Topology 1, and it reaches the maximum value in Topology 4. This is because Topology 1 and 4 yield the shortest and the longest agent itineraries respectively, and the task duration is proportional to the length of the itinerary. We can also observe in Fig. 6 that *Scheme A* always causes the largest task duration among the three schemes in each topology. This is because two agents are dispatched to perform the task in *Scheme A*, as illustrated in Section 6.1. Compared to *Scheme B*, *Scheme C* sees a decreased task duration since this smaller action script is stored locally.

Table 3 illustrates the total number of Zigbee packets incurred by each scheme, and the results for individual agents 1 and 2 (denoted by A1 and A2) are shown in brackets below each corresponding sum for schemes A and C. The overhead incurred by the environment-setting agents is not accounted for. While it is evident that *Scheme C* performs better than *A* and *B*, there needs to be a mechanism that allows locally existing agents to be modified at will to fully exploit the capabilities of the system. It is also evident that the label-based approach is the key to achieving performance improvements in terms of both itinerary completion delay and bandwidth used (number of packets).

Table 3. Number of Zigbee packets incurred by the agents in each scheme

Scheme	Topology 1	Topology 2	Topology 3	Topology 4
A	161 [44(A1)+117(A2)]	187 [44(A1)+143(A2)]	209 [66(A1)+143(A2)]	235 [66(A1)+169(A2)]
B	135	165	165	195
C	99 [36(A1)+63(A2)]	117 [36(A1)+81(A2)]	126 [54(A1)+72(A2)]	144 [54(A1)+90(A2)]

8 Conclusions

We have presented Wiseman, a mobile code approach for WSN. We described Wiseman's architecture, its language constructs, and its features as a suitable system for use in WSNs. We also described important implementation and programming details of our scheme. The design of our system is based on an earlier implementation of the Wave system for mobile processing in wired networks. However, a number of changes were introduced as required by the scarcity of hardware and energy resources that characterize WSN devices. Among these changes are: elimination of dynamic memory allocation, redefinition of the language constructs, simplification of the interpreter's architecture, and simplification of the agents' program structure. Wiseman has been implemented and tested in the Crossbow Micaz running TinyOS ver. 1.1, which yielded a binary image of only 19Kbytes and RAM usage of around 3Kbytes. The benefits of employing Wiseman's agents were readily evident as shown by their ultra-compact size, leading to very low bandwidth usage. In addition, Wiseman facilitates the creation of overlay networks, which also helps shorten the overall size of the agents and reduces operation overhead.

Acknowledgments. This project was supported by the National Sciences and Engineering Research Council of the Canadian Government under grant STPGP 322208-05.

References

1. Bellavista, P., Corradi, A., Stefanelli, C.: Mobile Agent Middleware for Mobile Computing. IEEE Computer 34(3) (2001)
2. Hui, J., Culler, D.: The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale. In: Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, Baltimore, USA (2004)
3. Liu, T., Martonosi, M.: Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In: Proceedings of ACM SIGPLAN, San Diego, USA (June 2003)
4. Levis, P., Culler, D.: Maté: A Tiny Virtual Machine for Sensor Networks. In: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, USA (October 2002)
5. Fok, C.-L., Roman, G.-C., Lu, C.: Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS), Columbus, USA (June 2005)

6. Fok, C.-L., Roman, G.-C., Lu, C.: Mobile Agent Middleware for Sensor Networks: An Application Case Study. In: Proc. Of the 4th Int'l Conf. Information Processing in Sensor Networks. IEEE Press, Los Alamitos (2005)
7. Chen, M., Gonzalez-Valenzuela, S., Leung, V.C.M.: Applications and Design Issues of Mobile Agents in Wireless Sensor Networks. IEEE Wireless Communications 14(6), 20–26 (2007)
8. Chen, M., Kwon, T.K., Yuan, Y., Choi, Y.H., Leung, V.C.M.: Mobile Agent Based Wireless Sensor Networks. Journal of Computers 1(1) (2006)
9. Chen, M., Kwon, T.K., Yuan, Y., Choi, Y.H., Leung, V.C.M.: Mobile-agent-based Directed Diffusion (MADD) in Wireless Sensor Networks. In: EURASIP Journal on Applied Signal Processing, vol. 2007(1) (January 2007)
10. Qi, H., Iyengar, S.S., Chakrabarty, K.: Multiresolution Data Integration Using Mobile Agents in Distributed Sensor Networks. IEEE Transactions on Systems, Man and Cybernetics – Part C: Applications and Reviews 31(3), 383–391 (2001)
11. Sapaty, P.: A Wave Language for Parallel Processing of Semantic Networks. Computers and Artificial Intelligence 5(4) (1986)
12. Sapaty, P.: Mobile Processing in Distributed and Open Environments. John Wiley & Sons, Chichester (2000)
13. The OMNeT++ Discrete Event Simulator, `http://www.omnetpp.org`
14. González-Valenzuela, S., Vuong, S.T., Leung, V.C.M.: A Mobile Code Platform for Distributed Task Control in Wireless Sensor Networks. In: Proceedings of 6th ACM MobiDE, Chicago, USA (2006)
15. Crossbow Technology, `http://www.xbow.com`
16. TinyOS for wireless embedded sensor networks, `http://www.tinyos.net`
17. The Wiseman Agent System for Sensor Networks, `http://www.ece.ubc.ca/~sergio/wiseman`
18. Glitho, R., Olougouna, E., Pierre, S.: Mobile Agents and Their Use for Information Retrieval: A Brief Overview and an Elaborate Case Study. IEEE Network Magazine 16(1) (2002)