

Programmable Middleware for Wireless Sensor Networks Applications Using Mobile Agents

Sergio González-Valenzuela · Min Chen ·
Victor C. M. Leung

Published online: 17 April 2010
© Springer Science+Business Media, LLC 2010

Abstract We describe the design, implementation and performance evaluation of Wiseman: a middleware platform developed for interpreting mobile agent scripts in Wireless Sensor Networks (WSN). Inspired by an earlier agent system originally devised for the coordination of distributed process in wired networks, we developed a simplified interpreter that can be embedded in resource-limited wireless sensor devices for processing text-based codes that realize diverse WSNs tasks. We describe in detail the foundations of our proposed approach, as well as the distinctive programming features that reduce its operation's overhead. In its present form, the Wiseman interpreter implementation occupies 19Kbytes of embedded code and 3Kbytes of SRAM. We also study the effects of employing the different agent migration methodologies supported by the interpreter, and report performance evaluations that gauge the codes' consumed bandwidth and migration delay, which can take as little as 235 mS per hop.

Keywords mobile agents · wireless sensor networks · performance evaluation · embedded software

S. González-Valenzuela (✉) · V. C. M. Leung
Department of Electrical and Computer Engineering,
The University of British Columbia,
Vancouver, BC V6T 1Z4, Canada
e-mail: sergiog@ece.ubc.ca

V. C. M. Leung
e-mail: vleung@ece.ubc.ca

M. Chen
School of Computer Science and Engineering,
Seoul National University,
Seoul 151-742, South Korea
e-mail: minchen@ieee.org

1 Introduction

The creation of compact electronic devices that sense environmental parameters and implement low-power wireless communication schemes to share data has driven a myriad of research efforts aimed at finding solutions that optimize their coordinated operation. Given that a sizable portion of the commercially available WSN hardware is battery-powered, most WSN research efforts deal with energy efficiency as the core of the schemes being proposed [1]. Additionally, many popular WSN hardware platforms provide limited resources in terms of available memory and processing power. Therefore, designing WSN applications that make efficient use of the corresponding hardware becomes a challenging task, along with the possibility of WSNs being deployed in hard-to-reach areas, making it difficult to plan for frequent battery changes and/or middleware updates if needed.

The aforementioned circumstances motivate further research into creating a resource-conscious solution that enables dynamic programmability of WSNs and rapid updates to the operation of sensor nodes. Providing WSNs with this dynamic re-tasking ability enhances the network's flexibility and operation convenience. However, extensive investigations are also needed to determine whether the proposed scheme is viable from an engineering perspective. To this effect, it is necessary to determine whether a programmable-tasking system embedded into resource-constrained sensor devices delivers the expected performance level.

One way of enabling programmable-tasking is by means of virtual machines that interpret and process text programs coded in a high-level language, and then perform lower-level processes as specified by these programs. In fact, virtual machines made their incursion into the realm of

WSNs in recent years. Maté [2], Impala [3] and Deluge [4] were the earliest approaches that enabled programmable-tasking of WSNs, and were soon followed by more innovative and complex schemes, such as SensorWare [5], SmartMessages [6], and Agilla [7]. In particular, the latter approaches made possible the deployment of interpretable codes that could move from one node to another of a multi-hop WSN. This mobility attribute provides the means for dispatching programs, also known as agents, which roam a WSN and re-task its nodes as specified.

Before these advances in code mobility aimed at WSNs were actually achieved, extensive investigations had already been conducted in the 1990s to better understand the advantages of choosing the mobile agent approach over classic message passing schemes in the overall performance of WSNs [8]. However, we have previously contested that the efficiency of an agent system in WSNs also depends heavily on the design of the enabling middleware [9]. To this effect, mobile agent interpreters in WSNs should provide the necessary functionalities to: (1) efficiently support the intended application of the networked system, and (2) implement the necessary mechanisms that minimize bandwidth and power consumption. A closer look at the existing mobile agent-based systems for WSNs reveals that their designs rather fit a general-purpose scheme. Consequently, their middleware architecture is prone to incurring increased overhead. For instance, SensorWare's architecture is only suitable for higher-end sensor devices with relatively plentiful resources, which enables the inclusion of a variety of functionalities that also incur extra overhead. Similarly, Agilla relies on a distinctive tuple-space mechanism to implement WSN re-tasking. Nonetheless, any quantifiable advantages over alternative approaches that can be directly attributed to the use of the tuple-space scheme remain unclear.

Our proposed middleware system for programmable-tasking in WSNs differs from its predecessors in that it implements specific functionalities to support well-defined operations and tasks. In other words, we depart from the general-purpose scheme otherwise attributed to WSNs by its counterparts, and tailor the functionalities of the system as-needed. We coin our system *Wiseman* (Wireless Sensors Employing Mobile AgeNts) [10, 11]. The contributions of our work are described as follows: (1) we present the architecture of our proposed system, which supports different migration mechanisms, and promotes a flexible execution flow; (2) we introduce an alternative language for implementing text-based mobile codes that yield compact action scripts for the programmable-tasking of WSN nodes; (3) we promote an execution model based on self-depleting strings of codes that simplifies their processing complexity and reduces inter-node forwarding overhead; and (4) we describe the implementation of *Wiseman* in hardware

devices as a proof-of-concept and provide all the relevant performance evaluations.

The rest of this paper is organized as follows. In Section 2, we introduce the original system upon which *Wiseman*'s architecture is based, and describe its adaptation into a middleware solution that is amenable for WSN use. In Section 3, we describe *Wiseman*'s architecture and operation principles. In Section 4, we introduce *Wiseman*'s language and constructs. In Section 5, we provide a detailed account of the codes' migration methodologies supported by *Wiseman*. In Section 6, we describe implementation aspects relevant to its practical implementation over commercially-available sensor network hardware. In Section 7, we present performance evaluations centered on bandwidth utilization, as well as migration and execution delay results. In Section 8, we describe a sample application scenario to evaluate the performance of the system for an early-fire-detection application in a fictitious forest setting. Finally, Section 9 concludes this paper.

2 System foundations

As mentioned before, the scarcity of hardware resources prevalent in several commercial sensor network devices prompts middleware engineers to find a trade-off balance between system functionality and performance. This same precept remains true when designing a mobile agent interpreter for WSNs. On the one hand, implementing a coarse-grained agent system implies that agents will be coded using simple language constructs that instruct the interpreter to perform compound-type operations to accomplish a certain task (e.g., `<run task A>`, `<run task B>`, `<end>`). Alternatively, mobile agents can be provided with a fine-grained language that enables them to describe detailed operations when performing a task (e.g., `<mov 1 x>`, `<add x 0xFB>`, ...) In the former case, it is evident that the interpreter would handle more compact agents, although with restrained flexibility in their ability to perform a detailed execution thread, whereas in the latter case larger agent programs would be able specify the exact operation flow for a certain task. In the end, the degree of granularity used for programming these agents should depend on the intended WSN's application. Consequently, it is instinctive to think of coarse-grained approaches as being as more suitable for agent systems built specifically for WSNs. In other words, it results counterintuitive to provide agents with an unnecessary control level of the node's processing functionalities if the tasks that the system is expected to performed are consistently repetitive. Moreover, the incentive behind using mobile codes is lost if the WSNs tasks they perform are deterministic, or if they require minimal flexibility. Instead, using mobile agents is warranted by the

WSN applications' need of having a programmable-tasking feature as the means to respond to external factors of the underlying environment, which might require applying altogether different strategies to deal with a problem. This reason motivated our considering the adaptation of a simplified version of the Wave system for use in WSNs.

The Wave system can be deemed as one of the earliest precursors of code mobility in computer networks [12]. In particular, Wave's language was specifically designed to handle compact programs for efficient task coordination in distributed environments [13]. This approach is appealing to mobile agent systems targeted at WSNs, since it relies on the execution of local data processing algorithms, contrary to deploying mobile codes that perform repetitive tasks. As a result of this, the agent system benefits from the following:

- 1) The process coordination and data processing parts of a distributed application become decoupled, and the former is left as the agents' main duty to perform.
- 2) Mobile agents can be further compacted by defining a language that is sufficient to describe the coordination methodology that the operator wishes to perform.
- 3) A language comprised by a condensed instruction set yields a simplified interpreter's architecture, and consequently a smaller memory footprint. Moreover, compact codes are quicker to forward between WSN nodes, and helps reduce delay and bandwidth overhead.

In the next section we describe Wiseman's architecture as a significantly simplified adaptation of the original Wave system.

3 Wiseman's architecture and operation principles

Wiseman's middleware was designed to occupy a limited amount of memory to make it amenable to WSN hardware. As a result, Wiseman implements only four components: an *Incoming Queue*, a code *Parser*, a *Processor* block, and an agent *Dispatcher*; as well as two helper components: an *Engine*, and a *Session Warden*. Even though the Engine block implements several data processing and miscellaneous maintenance functions required by the interpreter, it functions rather as an auxiliary component to the Processor. By the same token, the Session Warden's function is to assist in the segmentation and reassembly of larger agents when being forwarded from node to node through the corresponding sensors' radio interfaces. Figure 1 illustrates these components and their interactions. We note that although the outgoing queue is shown as a separate module, in practice it is implemented within the dispatcher.

The Incoming Queue receives agents that have been reassembled after arriving from the wireless interface for immediate processing in a first-come-first-served manner. However, agents may also be injected locally as needed.

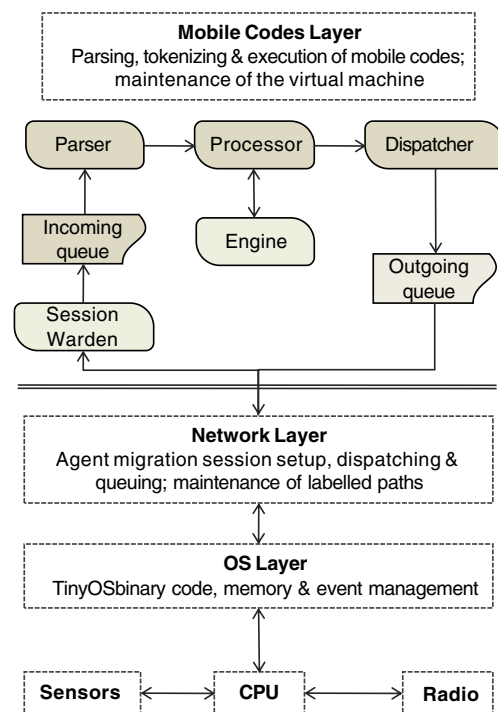
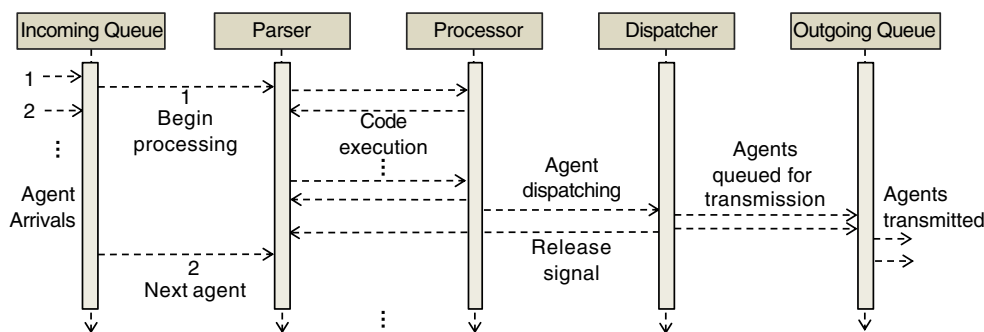


Fig. 1 Wiseman's architecture and layered structure of the sensor node system

Wiseman does not implement any multi-threading capabilities, since we designed the interpreter for use in WSN nodes with stringent hardware limitations. Consequently, agents are removed and executed from the Incoming Queue one at a time. The job of the Parser is to tokenize individual program instructions by first splitting the agent into two main segments: a *head* and a *tail*. The head is a single code segment that is sent to the Processor module for immediate execution, and the tail is comprised by the remainder of the codes that will be subsequently processed depending on the outcome of the head's execution. The Processor may rely on the Engine block to perform certain operations or maintenance of the Interpreter's variables while executing the corresponding head instruction.

The Parser regains control of the process immediately after the current instruction finishes executing, and the next instruction is obtained by tokenizing the first code segment from the tail if the outcome of the head's execution is successful. In this case, the Parser sends the next instruction to the execution block and the process continues. Figure 2 illustrates the execution sequence of Wiseman agents as performed by the interpreter. The continuous execution of codes is halted at any time if: (1) the current operation yields an unsuccessful outcome, (2) a hop operation is specified, or (3) an explicit termination operation is indicated. In the first case, the remainder of the agent's codes (i.e., the tail) is discarded. In the second case, the control of the execution process is passed to the Dispatcher,

Fig. 2 Agent processing sequence of the Wiseman interpreter



which forwards the agent’s tail to one or more neighbouring nodes. In the third case, the agent may explicitly instruct the interpreter to halt the current process execution. This execution sequence implies that the agent’s codes are gradually depleted, as seen in Fig. 3, although some portions of the codes may be skipped depending on how the agent itself is structured. Other aspects that are relevant to the interpreter’s main modules will be explained in Section 4 within the context of the Wiseman language. Similarly, aspects pertaining to the practical implementation of the Session Warden will be explained in detail in Section 6.

4 Wiseman’s instruction set and language constructs

Wiseman’s language is comprised of variables, rules, operators, and delimiters that are derived from the original Wave system. However, in contrast to Wave’s original syntax, Wiseman language directives are further condensed to reduce the size of agents as they migrate through the WSN. As mentioned before, Wiseman processes codes implemented as text scripts. This approach has the main advantage of allowing agents to be dynamically modified as needed, while making the codes human-readable. Additionally, the Parser’s use of an existing string-manipulation library greatly simplifies its overall structure. Table 1 illustrates the programming elements available in Wiseman, as explained next.

4.1 Variables

Wiseman implements three kinds of variables that employ a pre-assigned memory space. The first type is called

Numeric, and is defined as being of floating point type that is accessible through the letter *N* (e.g., *N1*, *N2*, etc.) The second type is called *Character*, defined for use with single characters referenced by the letter *C* (e.g., *C1*, *C2*, etc.) Both Numeric and Character variables are semantically similar to public variables seen in object-oriented programming. Therefore, all agents can access them. To this effect, the manipulation of Numeric and Character variables at any given node has no effect on the variables of other nodes. Additionally, agents may bring along *Mobile* variables as they migrate, which are accessed through the letter *M* (e.g., *M1*, *M2*, etc.), which are semantically similar to private variables in object-oriented programming. Consequently, agents can only access their own Mobile variables, which are temporarily stored in predefined memory locations when visiting a node. It is important to note that the WSN programmer is responsible for ensuring that all variable types maintain their semantic meaning throughout the WSN (i.e., the purpose of variable *N1* in node *i* should be the same as that of *N1* at node *j*). The interpreter can also temporarily store data that agents reference through the *Clipboard* variable *B*. Finally, Wiseman defines three *Environmental* variables that provide certain information about the current execution environment. First, the *Identity* variable *I* holds a read-only value of the local node’s identification number (i.e., 1, 2 ...) Second, the *Predecessor* variable *P* holds the identification number of the node that sent the agent currently being processed. Finally, the *Link* variable *L* holds the label identifier of the virtual link that the agent used for hopping, as discussed in the next section. The local node obtains the corresponding values for the Predecessor and the Link environmental variables from the agent’s control field as provided by the sender node.

4.2 Operators

Wiseman provides a variety of both general purpose and system-specific operators. All of the standard arithmetic (i.e., +, -, *,./and =) and comparison operators (i.e., <, <=, ==, =>, > and !=) are supported. The *Hop* operator indicates that the remainder of the agent’s codes (i.e., the tail) will be forwarded to a node specified on the right-hand

Step	Agent’s span				
t ₀	Head	Tail			
t ₁		Head	Tail		
t ₂			Skipped	Head	Tail
t _n

Fig. 3 Execution sequence of Wiseman codes

Table 1 The WISEMAN language

Lexeme	Name	Type	Description
N	Numeric	Variable	Local storage of numeric values
M	Mobile		A numeric value carried by an agent
C	Character		A character value stored locally
B	Clipboard		Temporary storage of a numerical value
I	Identity	(environmental)	Holds the local ID node value
P	Predecessor	(environmental)	Holds the ID value of the source node that dispatched the agent
L	Link	(environmental)	A character value used to label virtual links
O	Or	Rule	Yields true if <i>any</i> of the embraced commands is executed successfully
A	And		Yields true if <i>all</i> of the embraced commands are executed successfully
R	Repeat		Cycles through the commands embraced until a false outcome is encountered
+ - */=	Arithmetic	Operator(s)	Used to perform regular arithmetic operations on variables
< <= == ==> !=	Comparison		Standard operators to evaluate values and variables
#	Hop		Indicates that the agent will hop to another node or set of nodes
@	Broadcast		Broadcast agent to 1-hop neighbours
\$_	Execute		Performs local operation as indicated by parameters
!_	Halt		Stops execution with success or fail outcome as indicated
^	Insert		Inserts locally-stored agent
_?	Label query		Tests whether a labelled path exists in local node
;	Semicolon	Delimiter	Used to separate individual expressions
{...}	Curly bracket		Used to delimit expressions encompassed by a <i>Repeat</i> rule
[...]	Square bracket		Used to delimit expressions encompassed by an <i>And/Or</i> rule
(...)	Round bracket		Used to perform compound operations

side of the # character, or to the nodes associated to the virtual link value appearing on its left-hand side. In the latter case, the Hop operator automatically clones the agent with as many copies as destination nodes associated to the virtual links exist. For instance, if the virtual label g has 4 neighbouring nodes associated to it, then an equal number of agent clones are forwarded. Upon hopping to another node, the agent's execution thread resumes its operation at the point where the process had been suspended in the sender node. Consequently, Wiseman supports a basic form of strong mobility that eliminates the need of using a program counter and/or execution state that needs to accompany the agent. The *Query* operator allows agents to test whether a given label is already being used in the local node. For example, the instruction " $g?$ " tests whether label g exists. As an alternative to the Hop operator, the *Broadcast* operator forwards a local copy of a short agent to all immediate neighbours by using the @ character. The code *Injection* operator ^ can be employed to insert locally stored agents into the Incoming Queue module for subsequent execution. The *Halt* operator ! can be used to signal the explicit termination agent that is executing. The *Execution* operator \$ allows the programmer to call a local function. Its left-hand side value indicates the function being called, whereas the right-hand side passes a single runtime parameter. The

necessary functionalities to manipulate the state of onboard LEDs (i.e., on, off), to read values from the temperature/light sensors, and to change the frequency channels and transmission power of the radio chip have been implemented.

4.3 Rules

Wiseman implements a limited number of rules to manipulate the execution flow of an agent. First the *Repeat* rule R indicates that the codes delimited by curly brackets will be cyclically executed extracting and re-inserting them before the whole Repeat construct (i.e., the structure " $R\{...\}$ " yields " $...;R\{...\}$ "). This sequence can be continuously executed until a certain condition is found. Additionally, the *And/Or* rules defined by the letters A and O respectively, manipulate the execution of an agent by checking whether the codes delimited by square brackets yield a true or false value for each code segment. Thus, the construct specified by the Or rule " $O[...;...;...]$ " stops executing as soon as one of these segments yields a true value. Otherwise, the Or rule itself returns a false value, which halts the agent's execution. A similar logic applies for the And rule, with the key difference that all of the embraced segments must result true for the whole construct to succeed.

5 Wiseman's agent migration methodologies

Wiseman supports three agent migration techniques: explicit-path, variable-based, and label-based; each of which has its advantages and disadvantages, and may influence the agent's network load overhead, as described next.

5.1 Explicit path hopping

It is customary for Mobile Agent Systems in general to provide the necessary means that enable the explicit definition of the path that an agent will follow. This method implies that, prior to leaving the source node, an agent is provided with a pre-established itinerary of the nodes it plans to visit. In the case of agent systems in WSNs, this approach is reasonably justified if the conditions being monitored at the deployment setting are expected to remain for the most part stable [14, 15]. Wiseman enables explicit path hopping by means of the Hop operator introduced in the previous section, which uses the value on its right-hand side to decide where to hop. For instance, the instruction “#1” indicates an explicit hop from the current location to node 1. It then follows that the codes in the script “#1;#2;#3;#4;...” define a hopping sequence that takes the agent from node 1 through 4, at which point other operations may be performed.

5.2 Variable-target hopping

While convenient, the previous agent hopping technique has limited applicability since the agent's itinerary must be known in advance. However, the agent's itinerary might need to be revised after being dispatched from the source node in response to changes in the WSN's monitored environment. Wiseman supports this functionality by using one or more Mobile or Numeric variables on the right-hand side of the Hop operator as part of a customized algorithm implemented in the agent's constructs. For instance the operation “#N1;...” executed by the interpreter indicates that the ID of the target node where agent will hop to is stored in the Numeric variable N1, which must be set in accordance to the current circumstances. This implies that, since N1 is a local variable, any change made to its contents will have an effect on the hop target of an agent that visits the node afterwards. However, Wiseman agents may also resort to using Mobile variables in order to support variable-target hopping. For example, the operation “#M2...” would have the same effect as in the numeric variable case, except that its value can only be updated by the agent that owns it as it hops through the WSN.

5.3 Labelled path hopping

The main shortcoming of the previous agent migration methods is that they do not support forwarding multiple

copies of an agent to distinct nodes at once (i.e., in a multicast fashion). To solve this issue Wiseman also supports a functionality to create labelled paths that can be used to emulate multicast transmissions from the local node. To accomplish this, the interpreter implements a simple 2-dimensional table, in which node ID numbers are individually assigned to a character letter (i.e., a label). Thus, an agent can hop through labelled paths by passing the corresponding identifier as the left-hand operator of the same Hop operation (e.g., “a#”). Upon encountering a labelled-path Hop operation, the interpreter's processor fetches the IDs of the nodes associated with the corresponding label, and dispatches an agent copy to each of these nodes. Consider the following example in which node 0 has nodes 1 through 4 as neighbours, and the WSN programmer wants to create two multicast groups: one for the odd-numbered nodes, and one for the even-numbered nodes. To accomplish this, nodes 1 and 3 are assigned to label ‘a’ with the following codes: “L=a;#1;#P;#3”. We can see here that the labelled path is established by setting the Link environmental variable L to the corresponding value, whereas the predecessor environmental variable P is used to have the agent return to node 0 after the first label is set. Therefore, a simple agent that employs explicit path hopping through nodes 0-1-0-3 realizes this task, as shown in Fig. 4. A second agent script “L=b;#2;#P;#4” is subsequently dispatched with the corresponding values to set up the labelled path ‘b’ in the same fashion. After the previous paths have been set, other agents can be later dispatched using the instruction “a#”, or “b#” to reach nodes 1 and 3, or 2 and 4 respectively. One important advantage of following this approach is that subsequent agents arriving at node 0 do not need to know in advance the identities of the destination nodes in the labelled path. It also follows that labelled path hopping itself can be of either explicit-hopping or variable-target type. In the explicit-hopping case, a predetermined label is used (e.g., “c#”), whereas the variable-target case requires the use of Character variables (e.g., “C5#”), whose value can be modified as needed. This labelled-based approach enables creating shorter agents that can help reduce delay and bandwidth overhead, as seen in later.

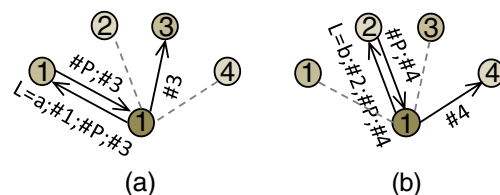


Fig. 4 Labelled-path setting example: **a** neighbors 1, 3 are marked with label ‘a’; and **b** neighbors 2, 4 are marked with label ‘b’

6 Practical implementation aspects

In this section, we describe aspects related to Wiseman’s implementation in sensor hardware. In its first conception, Wiseman was evaluated employing the OMNeT++ Discrete Event Simulator [16] to verify the correctness of its design after undergoing considerable changes from a much earlier proposal. This initial evaluation step was critical in expediting its implementation over actual hardware devices, given the complexity of debugging embedded programs. After verification, Wiseman was ported to the NesC language to produce a TinyOS ver. 1.1 binary image [17] for the Crossbow Micaz hardware [18]. This popular platform for creating WSNs provides a perfect example of hardware devices with severe resource limitations whereby our proposed scheme could be tested. In particular, the Micaz nodes provide 128Kbytes of instruction memory and 4Kbytes of SRAM memory. Wiseman is comprised of nearly 2,400 lines of NesC code divided into several modules that reproduce the system architecture shown in Fig. 1, and can be obtained from [19]. As mentioned before, Wiseman’s binary image yields 19Kbytes and around 3Kbytes of SRAM space, the latter of which reflects the space reserved to handle agents that occupy a maximum of 170 bytes. This memory amount has proven sufficient for all of our experiments.

An important issue that we had to address was that most middleware platforms used in sensor devices implement limited networking capabilities for data transfer between nodes. As a result, networking protocols are left as a task for the WSN programmer to put into operation. In Wiseman, a basic agent segmentation and reassembly mechanism is implemented by the Session Warden module. Agents that are segmented into various pieces for forwarding are reassembled using information contained in their corresponding header fields as marshalled by the source throughout the session. As seen in Fig. 5, this header contains the following information fields:

- 1) The current segment number, which is gradually incremented with every segment being sent.
- 2) The last segment indicator that is set to indicate the end of the current session.
- 3) The source node identifier.
- 4) A session number identifier that contains a pseudo-random number computed at the source in the beginning of each agent forwarding process.

All these values ensure that current agent forwarding session is able to recover from errors occurring in the wireless channel. (Fig. 6)

Every agent forwarding session begins with the sender’s issuing a *Request-To-Send* (RTS) packet to the target node to ensure first that the latter is currently available to receive data (since it might be involved in another agent forwarding process with a different node). The sender is granted permission to begin forwarding an agent upon receiving a *Clear-To-Send* (CTS) acknowledgement packet from the destination. To this effect, CTS packets indicate the segment number that the target node is expecting next. A discrepancy between the expected segment number, the session number, or the source node identifier annuls the whole process, and all of the previous agent segments are discarded. This measure ensures that agents are correctly forwarded by engaging in a sequential exchange of RTS and CTS signals until all of the segments have been forwarded. The agent forwarding session ends when a flag in the last segment field of a packet header is found, at which point, both the sender and the receiver reset their corresponding session values. To avoid possible deadlocks, a timeout signal is scheduled to trigger after 300mS at the sender’s side if no CTS acknowledgement is received, and retransmissions are attempted up to a maximum of 3 times. If the session setup process is unsuccessful, then the interpreter discards the current agent and the interpreter attempts to transmit the next one in the outgoing agent queue (if any). To this end, the Incoming Queue is set to hold a maximum of 3 agents, and the Outgoing Queue implemented by the Dispatcher module may hold a maximum of 5. If the Incoming Queue is full, then all additional RTS signals received are left unanswered until space to accommodate at least one more agent becomes available. Additionally, if the current forwarding process fails, then the receiver maintains its last session values given that the receiver implements no timeout procedure during an agent forwarding process. When another node attempts to initiate a forwarding session, the obsolete values at the receiver force a reset of the session process, and the operation resumes 300mS later (after the timeout expires). No routing functionalities are incorporated into the system. Instead, the WSN operator may create virtual multicast trees by explicitly labelling links between nodes using explicit hop operations. The benefits of this approach will

Fig. 5 Wiseman’s transfer session packet

Segment number	Last segment	Source	Session number	Wiseman Agent
Predecessor	Link	Type	Mobile variables	Codes

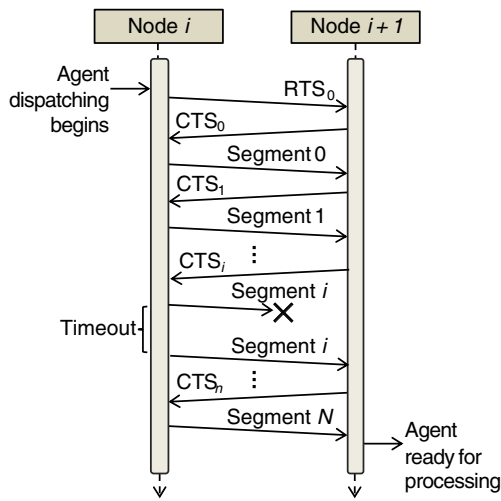


Fig. 6 Forwarding sequence of a Wiseman agent

become clear in Section 8, which explains a sample deployment scenario for Wiseman.

7 Performance evaluations of the Wiseman interpreter

We performed a number of experiments to evaluate the efficiency of the Wiseman interpreter implemented in TinyOS version 1.1 over Crossbow Micaz motes, as mentioned before. Performance was measured in terms of agent migration delay and bandwidth consumption (as network load incurred by agents).

Figure 7 illustrates the execution time averaged over 1,000 runs that each individual Wiseman instruction incurs, clearly showing that the hop operation sustains the shortest execution time, whereas arithmetic operations yield the longest one. The processing delay shown for the execute operator \$ was obtained by averaging over LED, transmission power and frequency channel change operations,

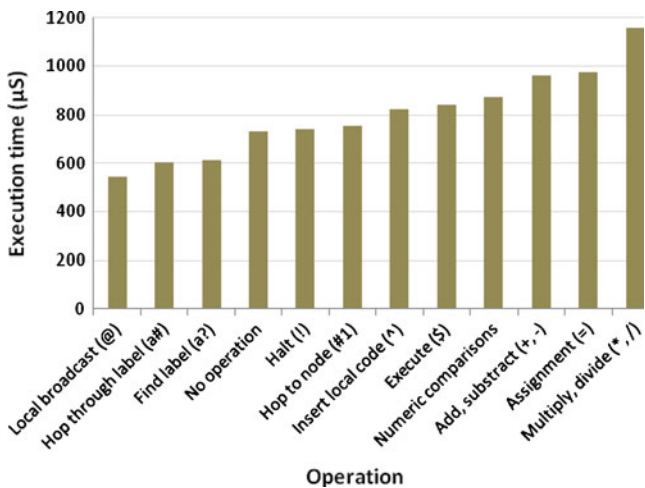


Fig. 7 Execution time of Wiseman operations

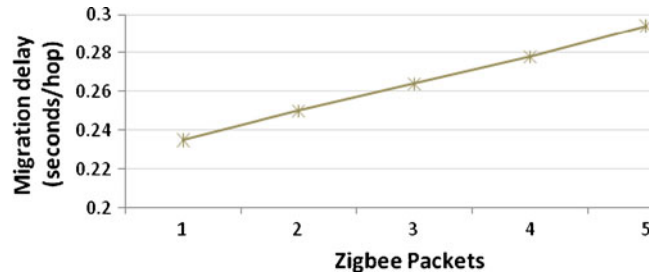


Fig. 8 Agent migration delay as a function of Zigbee packets incurred

which are already built into TinyOS. Overall, the average execution time hovers around the 800 μs mark, which is greater than Agilla’s. We attribute this result to the parsing and tokenizing delays of Wiseman text-based codes, as mentioned in Section 2. However, we deem the average execution delay as being within reasonable margins, since they do not pose any significant disadvantage to the overall system’s efficiency.

Figure 8 illustrates the migration delay of a sample Wiseman agent measured as a function of the actual number of Zigbee packets that each experiment spans. The agent employed for the initial test was coded to fit into a single Zigbee packet. Then, its size was gradually increased by padding it with arbitrary instructions in order to gradually fill in additional Zigbee packets. The 5-packet case sets the limit for the current Wiseman implementation that accommodates up to 170-bytes of code. These results are averaged over 100 runs, and provide a good delay estimate that can be used as a reference when coding Wiseman programs for deployment in WSNs formed by Crossbow Micaz. We note that these delays also correspond to a WSN whereby the forming devices were separated 10 cm from one another, and their transmission power was set to -15 dbm in a radio environment shared by multiple IEEE 802.11 WLANs indoors.

The delay results for a path length of up to 7 hops that correspond to the explicit path, variable-target, and labelled path migration techniques advanced in the previous section are shown in Fig. 9. As expected, the label-hopping technique yields the shortest migration delay, whereas the

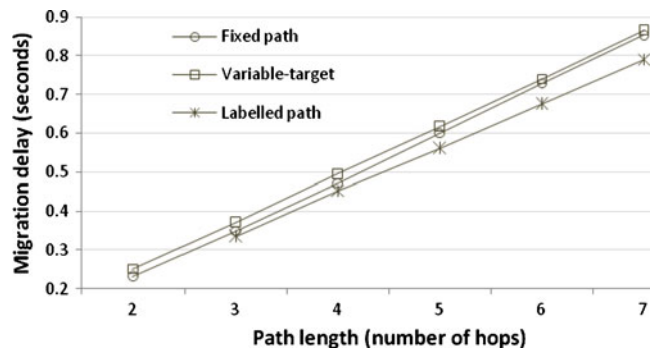


Fig. 9 Forwarding delay for distinct agent migration techniques

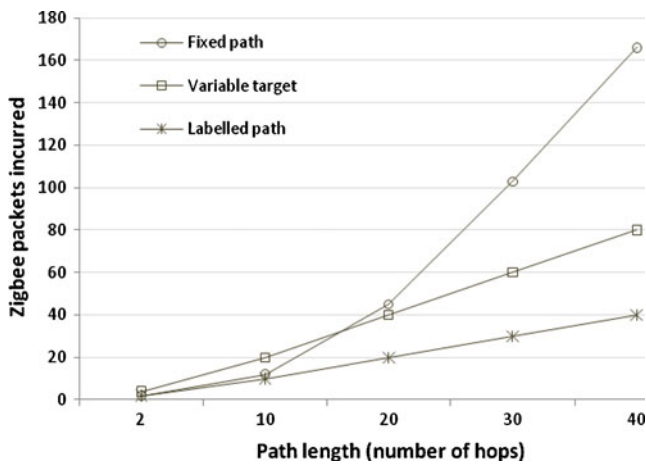


Fig. 10 Number of Zigbee packets incurred for distinct agent migration techniques

performance of the variable-target method rivals the one given by the fixed-path hopping methods as the path length nears 8. This result is explained by the growing agent size in the fixed-path approach with the increasing number of explicit hop instructions being added to the codes (e.g., “...#1...”, followed by “...#1;#2...”, and then “...#1;#2;#3...”, and so forth.) Eventually, the delay of the latter surpasses the one yielded by the former, implying that if agents are expected to traverse longer paths, then the variable-target hopping method should be implemented, if feasible.

Figure 10 shows the number of Zigbee packets incurred by each of the three types of migration techniques implemented by the corresponding agents for up to 40 hops, which were numerically calculated. The 40-hop limit is set by the maximum agent size of 170 bytes, which is reached by codes implementing the explicit path method, and is followed by the agents that implement the remaining methods for fairness. These results provide a compelling reason for resorting to the labelled-path hopping technique as supported by the WSN application. We can also see that the bandwidth overhead of an agent implementing the labelled based variable-target hopping technique is roughly half of the one obtained with the fixed path approach.

These results are consistent with the migration delay evaluation illustrated in Fig. 9 for 7 hops, indicating that a larger number of explicit hop-by-hop instructions have a direct effect on both the migration delay and bandwidth overhead due to the agents occupying additional Zigbee packets. Nonetheless, the cost of implementing explicit hop-by-hop migration becomes competitive when an agent traverses increasingly shorter paths.

It could be argued that the text-based approach used for coding agents can be regarded as a disadvantage in terms of execution delay, since code delimiters become a larger proportion of the agent’s text script. A straightforward solution to this issue can be achieved by defining fixed-length instructions (for instance, codes can be defined to span 2 bytes, where the first byte represents the instruction to execute, and the second byte is the value to act over. In other words, the agents adopt an assembly programming-like structure that requires no delimiters and can help save bandwidth and processing overhead, as seen in the Agilla system [7]. On the other hand, Agilla’s programs cannot be modified once they have been injected into a WSN, which limits the flexibility of their agents, and requires forwarding the corresponding program’s execution state, thus offsetting the bandwidth savings that had been introduced.

8 Case study: early forest fire detection

8.1 Experiment’s rationale

We consider a WSN application targeted at the prevention of forest fires as a descriptive example to showcase the capability of the Wiseman system. Forest fires that frequently occur in natural settings can cause significant environmental damage, and threaten the integrity of man-made infrastructure in the presence of high temperature and low humidity. Whereas the using WSN applications for weather monitoring [18] and forest fire tracking have already been explored [5], we focus on the issue of early detection and/or prevention. Figure 11 illustrates a sample

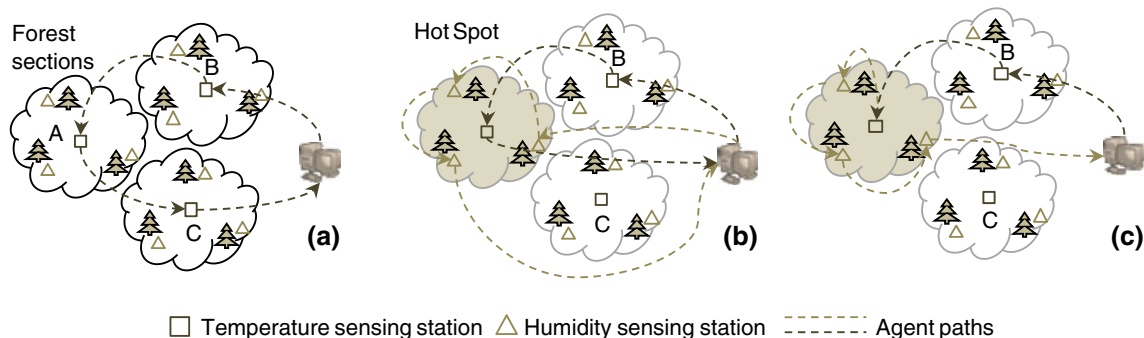


Fig. 11 Deploying Wiseman agents in the early detection of forest fires for: a routine temperature checking; b a primary temperature monitoring task; and c a secondary humidity monitoring task

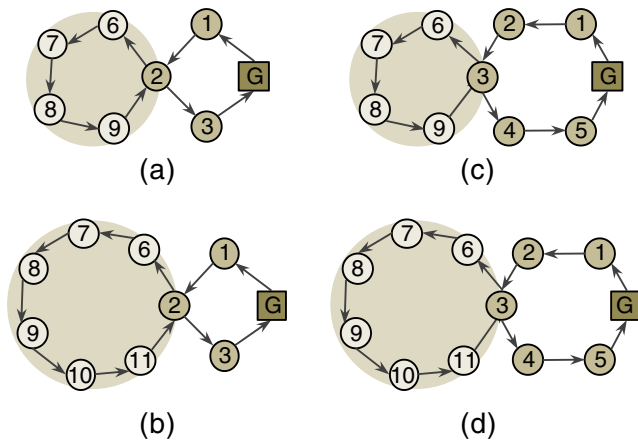


Fig. 12 Test topologies showing nodes 1–3/5 in the temperature sensing circuit, and nodes 6–9/11 in the humidity sensing circuit

forest region that is divided into sections *A*, *B* and *C*, whereby the proposed early forest fire detection application can be realized by dividing the task into two stages. In the first stage of the monitoring process, temperatures readings are collected at each of the forest sections through their corresponding temperature sensor node, which we regard as cluster-head. If the temperature exceeds a predefined threshold, then a secondary stage of the task follows, in which humidity readings for the corresponding forest area are also collected. If either of these readings is lower than a certain critical threshold, then an alert signal is sent to the monitoring centre.

Under regular operation, a Wiseman agent collects temperature readings along the predefined path for each of the monitored areas, as depicted in Fig. 11 (a), whose itinerary may be planned according to the hazard priority. From time to time, a mobile agent might detect an abnormal temperature reading that exceeds the critical threshold, as shown in Fig. 11b and c. At this point, one of three different approaches can be employed to initiate stage 2:

Scheme A: The primary agent returns to the sink node immediately after the temperature reading anomaly is found, and in turn dispatches a second agent whose task is to obtain the minimum humidity reading.

Scheme B: The primary agent possesses the required action script that handles the emergent case,

and performs the secondary task directly starting at the node that recoded the anomaly, as shown in Fig. 11b.

Scheme C: The primary agent does not carry the action script to handle the special case. However, the secondary agent that handles task 2 is already stored at all nodes, and so it is invoked locally. In this scheme, the agent’s itinerary is the same as in Scheme B.

Note that in our previous agent-based approaches [14] and [15], only Scheme A and Scheme B were supported since the itinerary must be planned in advance by the sink node. Conversely, Wiseman enables dynamic itinerary changes (e.g., at node 2 in Fig. 12a, b) for Scheme C.

8.2 Case-study setup

The four topologies that emulate the forest settings in our experiments are shown in Fig. 12. We emulated the placement of humidity sensor nodes in the fictitious hot spot forest Section 2 of Fig. 12a, b; and in Section 3 of Fig. 12c, d. In this sample setting, node 2 is the cluster-head temperature sensor of the forest area monitored by humidity sensor nodes 6, 7, 8 and 9. In another experiment, two more humidity sensor nodes monitor the hot spot depicted in Fig. 12b in order to depict a larger forest area. In contrast, the topology in Fig. 12c incorporates two extra temperature sensor nodes as compared to topology 1, whereas two more humidity sensor nodes are added in topology 4, as shown in Fig. 12d.

Table 2 shows the agent scripts that set up the corresponding environment and labelled paths for subsequent agent navigation for the respective experiments, whereas the agent scripts shown in Table 3 carry out the actual monitoring tasks in accordance to Schemes *A*, *B* and *C*. The character ‘*a*’ is assigned to the temperature-reading labelled path by using the corresponding link assignment operation “ $L=a$ ”. Similarly, the character ‘*b*’ is assigned to the labelled path in the humidity-reading circuit. Green LEDs are switched on by means of the “ $I\$n$ ” operation as a visual aid to verify that agents complete their itinerary as expected. The interpreter’s Clipboard B value is set to a value of 45 at either node 2 or 3 depending on the scheme

Table 2 Environment-setting agents for the case-study tests

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

Table 3 Agent scripts that implement distinct migration strategies

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

being tested to emulate the event of the local temperature reaching this value. Finally, the values on the right-hand side of the hop operator indicate the node number to which the agent is set to migrate next.

In our experiment setup for Scheme A, a 59 byte-long agent first explores the cluster-head route formed by nodes 1–3 (topologies (a) and (b) in Fig. 12) or 0–5 (for topologies (c) and (d) in Fig. 12). A variable-target migration method is employed here since this scheme does not rely on virtual links, and so the value of mobile variable *M0* is sequentially incremented (“*M0+1*”) and employed to determine the next hop destination (“*#M0*”). Upon reaching the next node, the agent toggles the green LED on and reads the sensed temperature (“*l\$nr\$t*”). If the temperature value is lower than the predefined threshold set beforehand (“*B < 40*”), then the execution continues at the following Or rule. However, a false outcome indicates that the local temperature has exceeded the threshold, and so 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 migration itinerary, and the “*I!=0*” operation ensures that Agent 1 terminates upon returning to the sink. This process is then continued by Agent 2, which is dispatched in response to the value in *M1* previously returned by Agent 1. It can be seen that 85-byte long Agent 2’s itinerary is set deterministically for topology 2, whereby the agent enters the humidity-sensing route at node 3, and traverses nodes 6 through 9 before exiting back

to node 3, as indicated by *M0*’s value. As before, the value of the current node’s ID is stored in *M1* if the humidity reading exceeds a predefined threshold by the script block “*r\$h;O[B > 20;M1=I]*”.

Unlike Scheme A, Scheme B defines a virtual path set beforehand by the corresponding agent (according to the current topology), which can be subsequently employed by other agents implementing the labelled-based hopping method, and using the letter ‘a’ for the temperature-reading circuit, and the letter ‘b’ for the humidity-sensing circuit (i.e., “a#”, or “b#”). Consequently, a single 79-byte long agent is needed to this carry out the corresponding task. Finally, a 36-byte long agent employs the local injection operator (^) to implement Scheme C if the temperature reading exceeds the predefined threshold. Since this agent does not carry the associated code for specifying the path switching rule during its navigation, a 46-byte long agent needs to be already available at the WSN nodes for injection from the node whose temperature value exceeded the corresponding threshold to the humidity-sensing circuit after a 2-second delay (“*2^0*”).

8.3 Case study results

We conducted the corresponding experiments that implement Schemes A, B and C for the topologies described earlier. We evaluated the performance of Wiseman in terms of task duration and packet overhead. To this effect, the

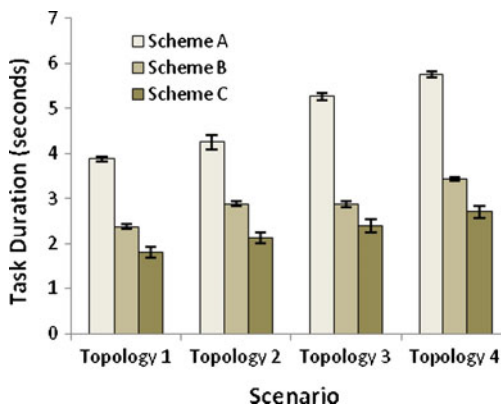


Fig. 13 Itinerary completion delay for schemes A, B and C

Table 4 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)]

task duration bracket (i.e., the itinerary completion delay) starts at the time that the sink node dispatches the first agent, and ends when the task of the last stage finishes.

Figure 13 shows that the task shortest task duration always occurs in Topology 1, whereas Topology 4 yields the maximum, as expected, since the task duration is proportional to the length of the itinerary. In addition, Fig. 13 shows that Scheme A always incurs the longest task duration among the three schemes, regardless of the topology tested, which is attributed to the fact that two agents are needed to perform the task in Scheme A. On the other hand, Scheme C sees the shortest task duration since only one agent is deployed, which in turn invokes a smaller action script that is stored locally for execution. Table 4 illustrates the total number of Zigbee packets incurred by Schemes A, B and C, in which individual results for agents 1 and 2 (denoted by $A1$ and $A2$) are shown in brackets below the corresponding sum in schemes A and C, unlike Scheme B that employs only one agent. (The overhead incurred by the environment-setting agents is not taken into account.) Evidently, Scheme C outperforms both A and B because of its label-based approach introducing improvements in terms of itinerary completion delay and bandwidth used (number of packets).

9 Conclusions

We have provided a detailed account of our experiences developing the Wiseman system for deploying mobile codes in WSNs. Wiseman's design is based on an earlier implementation of the Wave system for mobile processing in wired networks. However, several changes were introduced to cope with the scarcity of memory, processing, and energy resources intrinsic to WSN hardware. The results presented in this investigation support our claims that Wiseman is a viable solution to enabling programmability in this type of networks. Our performance evaluations also show that the system's efficiency is highly dependent on how the program is designed, which can have a significant impact in the migration delay and bandwidth overhead incurred. In addition, the overall system's performance will also depend on the WSN's size. Particularly, the sole use of the explicit path-hopping technique for agent migration may result detrimental to the bandwidth usage for large WSNs, whereas the impact of employing this technique in smaller WSNs is equivalent to the other ones supported. We also note that the efficiency of the labelled-path hopping approach also depends on whether the underlying conditions of the WSN's deployment setting warrant constant label-maintenance sub-tasks. If labelled-paths maintenance is expected to be performed in a continuous basis, then the variable-target hopping can potentially provide the best

solution. However, the effectiveness of variable-target hopping itself also depends on how an agent's codes are structured, at which point the skills of the WSN programmer may come into play. Moreover, it is possible that the task being performed by the WSN is sufficiently complex to warrant the concurrent use of multiple agents, especially if the WSN devices' memory limitation impedes the creation of a single larger agent to realize all the required goals. In such case, a combination of distinct migration techniques and execution flows may be the best approach to follow.

Acknowledgment This project was supported by the National Sciences and Engineering Research Council of the Canadian Government under grants STPGP 322208-05 and 365208-08. In addition, this work was supported in part by NAP of Korea Research Council of Fundamental Science & Technology.

References

1. MacRuairi R, Keane MT, Coleman G (2008) A wireless sensor network application requirements taxonomy. Proceedings of the second international conference on sensor technologies and applications, SENSORCOMM, Cap Esterel, France, 25–31 August 2008
2. Levis P, Culler D (2002) Maté: a tiny virtual machine for sensor networks. Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, USA, October 2002
3. Liu T, Martonosi M (2003) Impala: a middleware system for managing autonomic, parallel sensor systems. Proceedings of ACM SIGPLAN: Symposium on Principles and Practice of Parallel Programming, San Diego, USA, June 2003
4. Hui J, Culler D (2004) The dynamic behavior of a data dissemination protocol for network programming at scale. Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems, Baltimore, USA, November 2004
5. Boulis A, Han C-C, Srivastava M (2003) Design and implementation of a framework for efficient and programmable sensor networks. Proceedings of the First International ACM Conference on Mobile Systems, Applications and Services, San Francisco, USA May 2003
6. Kang P, Borcea C, Xu G, Saxena A, Kremer U, Iftode L (2004) Smart messages: a distributed computing platform for networks of embedded systems. The Comput J Special Issue on Mobile and Pervasive Computing, Oxford Journals 47(4):475–494
7. Fok C-L, Roman G-C, Lu C (2005) Rapid development and flexible deployment of adaptive wireless sensor network applications. Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS), Columbus, USA, June 2005
8. Qi H, Iyengar SS, Chakrabarty K (2001) Multiresolution data integration using mobile agents in distributed sensor networks. IEEE Trans Syst Man Cybern C Appl Rev 31(3):383–391
9. Chen M, Gonzalez-Valenzuela S, Leung VCM (2007) Applications and design issues of mobile agents in wireless sensor networks. IEEE Wireless Commun 14(6):20–26
10. González-Valenzuela S, Chen M, Leung VCM (2009) Design, implementation and case study of WISEMAN: wireless sensors employing mobile agents. Proceedings of the 2nd International ICST Conference on MOBILE Wireless MiddleWARE, Operating

- Systems, and Applications (MobilWare), Berlin, Germany, April 28–29
11. González-Valenzuela S, Chen M, Leung VCM (2009) Programmable re-tasking of wireless sensor networks using WISEMAN, to appear at the International Workshop on Advanced Sensor Integration Technology (ASIT), Niagara Falls, Canada, September 23–25
 12. Sapaty P (1986) A wave language for parallel processing of semantic networks. *Comput Artif Intell* 5(4)
 13. Sapaty P (2000) *Mobile processing in distributed and open environments*. Wiley
 14. Chen M, Kwon T, Yuan Y, Choi Y, Leung V (2007) MADD: mobile-agent-based directed diffusion in wireless sensor networks. *EURASIP Journal on Applied Signal Processing*. doi:10.1155/2007/36871
 15. Chen M, Leung V, Mao S, Kwon T, Li M, (2009) Energy-efficient itinerary planning for mobile agents in wireless sensor networks. *Proceedings of IEEE ICC*, Dresden, Germany, June 14–18
 16. The OMNeT++ Discrete Event Simulator. <http://www.omnetpp.org>.
 17. TinyOS for deeply embedded wireless sensor networks. <http://www.tinyos.net>.
 18. Crossbow Technology. <http://www.xbow.com>.
 19. The Wiseman Agent System for WSNs. <http://www.ece.ubc.ca/~sergio/wiseman>.