Programmable Re-tasking of Wireless Sensor Networks Using WISEMAN

Sergio González-Valenzuela[1], Min Chen[2], Huasong Cao[1], and Victor C.M. Leung[1]

[1] Department of Electrical and Computer Engineering
The University of British Columbia
2332 Main Mall, Vancouver BC, V6Z1T4, Canada
{sergiog,huasongc,vleung}@ece.ubc.ca
[2] School of Computer Science & Engineering
Seoul National University
Seoul, 151-744, Korea
mchen@mmlab.snu.ac.kr

Abstract. In this paper we present a flexible middleware platform for re-tasking Wireless Sensor Networks (WSNs) that we coin *WISEMAN*. Based on our previous experiences with mobile agents in computer networks, we developed a lightweight interpreter of text-based codes that enables their deployment in order to implement diverse WSNs tasks. WISEMAN occupies 19Kbytes of TinyOS embedded code, and 3 Kbytes of memory to operate in commercially available sensor nodes. We examine different agent migration methodologies, and present performance evaluations to gauge their efficiency in terms of delay and bandwidth with aims to determine which approach works best depending on the intended agent application. Our results indicate that WISEMAN agents can migrate as fast as 235 mS per-hop, which is comparable to existing approaches, while supporting the necessary code execution flexibility needed for the rapid implementation and deployment of WSN re-tasking programs.

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

1 Introduction

WSNs are created by small hardware devices that possess the necessary functionalities to measure and exchange a variety of environmental data in their deployment setting. Most WSN applications warrant the use of battery-enabled devices, to support their placement in locations where a wired electricity supply is either impractical to set up, or is simply unavailable. Consequently, WSN algorithms and communications protocols are designed to consume the least possible amount of power in order to extend the batteries' lifetime, and ultimately enable a more convenient, cost-efficient solution. To achieve this objective, WSN research in the past few years has produced novel ideas in the areas of signal processing, data aggregation, and wireless networking protocols, among others [1]. Regardless of its intended application, we note that it may be impractical to modify the operation of a

J. Zheng et al. (Eds.): ADHOCNETS 2009, LNICST 28, pp. 780–794, 2010.

WSN once its forming devices have already been deployed. Therefore, additional efforts have been put forward to enable dynamic WSN programmability, which is commonly known as re-tasking.

WSN re-tasking approaches such as Maté [2], Impala [3] and Deluge [4] were the first pioneers in this area, and were followed by others, such as SensorWare [5], SmartMessages [6], and Agilla [7]. Compared to their predecessors, the latter approaches introduced more sophisticated forms of code mobility. A closer look at all of these schemes reveals that middleware design in WSNs is heavily dependent on the targeted application of the system [8]. However, other factors also play important roles, such as the way in which codes are moved from one host to another. These mobile codes are often referred to as mobile agents, which are comprised of *interpretable* instructions with a granularity level that reflects a particular level of execution. Consequently, some agent systems support language constructs that afford a fine-grained process execution flow, whereas other systems implement coarse-grained languages to support high-level procedures and execute multiple instructions in compound. Ultimately, mobile codes systems targeted at WSN re-tasking should provide the necessary functionalities to: (1) efficiently support the main aspects of the networked sensor system application (e.g., data aggregation, node coordination, etc.), and (2) incur the least possible bandwidth and power consumption.

This paper presents follow-up work from our previous work on WISEMAN (WIreless Sensors Employing Mobile AgeNts) – a mobile codes approach for flexible WSN re-tasking [9]. WISEMAN is designed on the premise that WSNs are deployed to gather distributed information in settings where the behaviour of the underlying environment can constantly change. As a result, our proposed system is built to interpret codes in the form of compact text scripts that can be dynamically modified to provide enhanced flexibility. WISEMAN's main features can be summarized as follows:

A. *Compact language*. WISEMAN's language constructs are based on an earlier code mobility system that defines a high-level text-based language system acting as a compact script that can help save bandwidth and battery power by minimizing the number of packets needed for their transmission.
B. *Small memory footprint*. WISEMAN's interpreter implementation in TinyOS v1.1 occupies a mere 19Kbytes of memory, leaving plenty of space to incorporate additional functionalities as needed.
C. *Stateless execution model*. Unlike most programming models, WISEMAN implements an execution model based on self-depleting strings of codes that simplifies their processing and reduces inter-node forwarding overhead.
D. *Flexible code migration strategies.* Our scheme enables the ability to dynamically modify agent itineraries that can employ explicit hop-by-hop migration, variable-based hopping, and a virtual-link navigation capability that mimics multicast routing through labelled paths.

In a previous contribution, we described in detail WISEMAN's system architecture and language constructs, accompanied by results of a sample deployment scenario to illustrate its applicability [9]. The contributions of this paper are summarized as follows. We will provide a quick review of the WISEMAN system, including a description of its latest features, and a discussion of the advantages and disadvantages

of using particular language constructs. In addition, we will discuss in detail the features that make it a unique and effective system for the deployment of mobile codes in WSNs. We will also present generic performance evaluations of WISEMAN's codes execution and migration delay, as well as bandwidth consumption. In addition, we will describe programming aspects in more detail that can help to better estimate the performance of possible applications, and anticipate potential performance issues.

The rest of the paper is organized as follows. In Section 2, we describe WISEMAN's architecture, operation principles and language constructs. In Section 3, we provide a detailed description of WISEMAN's migration methodologies, its supported code execution flows, and discussions of the potential benefits and drawbacks obtained by employing each of these. In Section 4, we present performance evaluations that depict execution and migration delay, as well as bandwidth utilization of our mobile codes as processed by an interpreter programmed in TinyOS v1.1 over commercially-available sensor hardware. Finally, we present conclusions of our paper in Section 5.

2 Overview of WISEMAN

In this section, we revisit the most important aspects of WISEMAN's architecture and language constructs. A more detailed account of these can be found in [9]. We also describe recently added functionalities to the system, followed by a discussion of the advantages and disadvantages of employing our language structure.

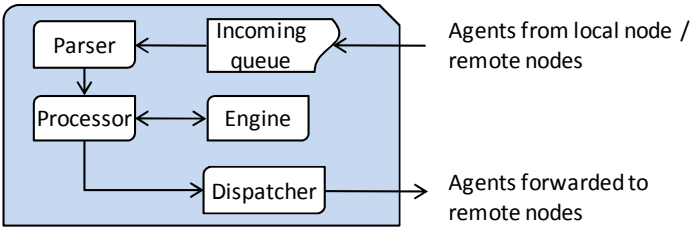

Fig. 1. Internal architecture of the WIESEMAN interpreter

2.1 WISEMAN's Architecture

One of the main objectives of WISEMAN is to ensure that the interpreter occupies the least possible amount of resources in the implementing hardware. For that reason, we define the interpreter's architecture as having only four basic components: an incoming agent queue, a code parser, a processor block, and an agent dispatcher, as shown in Fig. 1. We also define a supplementary component regarded as the system's Engine. Although this block might not necessarily implement the type of functionalities regularly attributed to software engines, it does implement a library of functions that encompass a good deal of the data processing capabilities provided by the interpreter as a whole, and thus the naming. An additional sub-block that is

responsible for setting up agent forwarding sessions is also described shortly. Fig. 1 shows that the Incoming Queue receives agents that arrive from other nodes in the WSN once they have been re-assembled. However, it is also possible to pre-load the Incoming Queue with agents that will be executed in an ongoing basis.

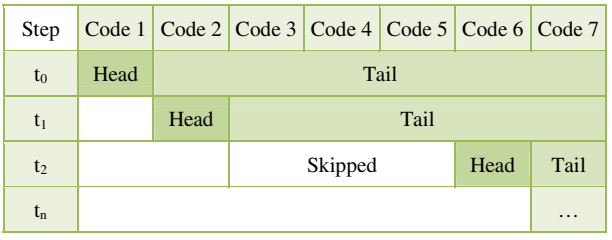

Fig. 2. Execution sequence of WISEMAN codes

Since WISEMAN is targeted at resource-limited WSN nodes, we assume that the implementing hardware does not have multi-threading capabilities, and so agents are removed one by one from the Incoming Queue. The Parser's main task is to tokenize each language instruction for subsequent handling by the processor module. Here, the text string that forms the codes is separated into a *head* and *tail*. The former is the code portion that will be immediately processed, whereas the latter contains the rest of the codes that may be subsequently processed, depending on the outcome of the head's execution. The Parser forwards each tokenized instruction to the Processor, which may in turn rely on the Engine block to perform any operation that has interpreter-wide implications.

After being serviced, the current code is immediately discarded, and the Parser regains control of the execution process. The next instruction is obtained from the *tail* (becoming the new *head*), and is immediately tokenized for subsequent processing. This execution sequence implies that the agent's codes are gradually depleted, although some portions of the codes may be skipped depending on how the agent itself is structured, as seen in Fig. 2. Additionally, an instruction may indicate the interpreter to forward the *tail* of the agent to one or more nodes, at which point the agent's tail is passed to the Dispatcher for immediate transmission.

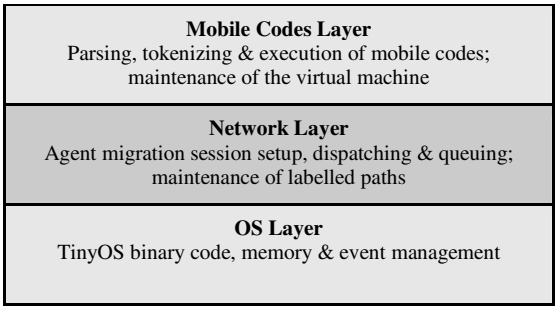

Fig. 3. Layered execution model of WISEMAN

Fig. 3 illustrates the layered structure of the WISEMAN system. At the top-most position, the Mobile Codes Layer is in charge of all agent-handling functions, and is comprised by the Parser, the Processor, and the Engine blocks of the interpreter as explained above. Maintenance of the environmental variables for the agent being currently executed is also performed at this level. The Network Layer is comprised by a Session Warden that is in charge of reassembling WISEMAN agents as they arrive from the wireless medium by using information embedded into the corresponding session fields, as seen in Fig. 4.

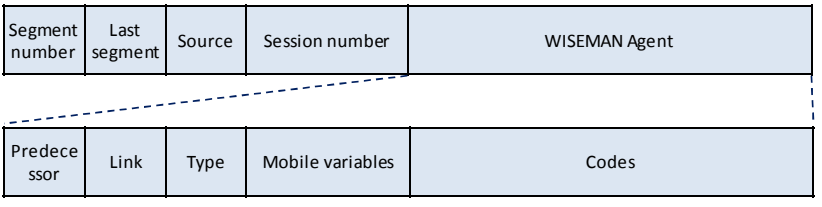

Fig. 4. WISEMAN's forwarding session packet

This agent forwarding process carries out the necessary steps for ensuring that an agent spanning multiple Zigbee packets is correctly received and reassembled at the destination node. To accomplish this, values that correspond to the current segment number, last segment indicator, source node, and the actual session number are employed as the header part of a session packet that wraps all WISEMAN agents during the forwarding process. This process consists of an exchange of simple Request-to-Send (RTS) and Clear-To-Send (CTS) signals, as detailed in [9]. These values help the system keep track of every single segment being transferred in case errors in the wireless channel occur. The segment field is a 1-byte value that is gradually incremented with every segment being sent. The last segment indicator field (1-byte) is a simple flag value that specifies whether the current segment is the last for the corresponding session. The source field contains the identification number of the source node. This value is used separately from the one referenced by the interpreter as the *Predecessor* value to help discriminate other agent segments that might be transmitted by nearby nodes. The actual session number is a pseudo-random number computed at the source at the beginning of each agent forwarding process that is kept fixed for the duration of the process, after which it is subsequently discarded. On the other hand, the Dispatcher block handles the necessary operations for setting up agent forwarding sessions that are also part of the Network Layer. Finally, the Operating System Layer is implemented by the TinyOS binary codes that WISEMAN is linked against when the actual interpreter is compiled. All memory and event management tasks that WISEMAN requires to operate are handled here.

2.2 WISEMAN's Language Constructs

Table 1 summarizes WISEMAN's language variable types, rules, operators, and delimiters. WISEMAN derives its language and architecture from the Wave system

that was originally introduced in 1986 [10, 11]. Compared to Wave's original syntax, WISEMAN further condenses its language directives in order to reduce the size of agents when migrating between nodes. As in the Wave system, our scheme also interprets and processes codes implemented as text scripts. This approach has the following advantages: (1) it makes the codes human-readable, unlike approaches like Agilla that rely on agents built in a byte-code style, (2) it allows agents to be modified more easily in a dynamic fashion, (3) it facilitates the parser's structure by employing a library of functions for string-manipulation, and (4) it allows for the implementation of the interpreter in other languages and systems that can process WISEMAN strings in the same fashion. For instance, this implies that a standard PC could pre-process WISEMAN codes by running the corresponding programmed in, say, Perl language, before injecting the agents into the WSN.

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 *any* of the embraced commands is executed successfully
A	And		Yields true if *all* 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 *Repeat* rule
[…]	Square bracket		Used to delimit expressions encompassed by an *And/Or* rule
(…)	Round bracket		Used to perform compound operations

The main disadvantages of our approach are that the language may be perceived as being cryptic, and that handling text strings in devices with severe processing limitations can add unwanted delay, as seen later. Another disadvantage is that code delimiters become a larger proportion of the overall agent text string when compared to the actual codes. The simplest way to overcome this problem is to define fixed-length instructions. For example, instructions could be set to span 2 bytes, where the first byte represents the command, and the second byte the value to act upon (i.e., in an assembly language-like fashion). By doing this, no explicit delimiters are needed, which can help save bandwidth and processing overhead. In fact, this approach is employed by the Agilla system, which can yield compact agents spanning tens of bytes [7]. However, a counterargument to this is that Agilla's programs cannot be changed once they have been injected into a WSN, which hinders the flexibility of their proposed scheme. In addition, Agilla's codes require forwarding all or a portion the whole program's execution state, which may offset the bandwidth savings that had been eliminated by not employing command delimiters.

Finally, WISEMAN has been recently enhanced by introducing a couple of extra functionalities. One of these is the the query operator '*?*'. Previously, agents had no way of knowing whether a given label had been already assigned to a hop between two or more nodes. As a result, an agent attempting to use an inexistent labelled-path would fail and its execution would immediately stop, having an unwanted effect on the overall application process. On the other hand, providing the agent with the necessary label-setting instructions meant added overhead. The introduction of the query operator allows agents to test whether a specific label has been assigned at the local node. For example, the instruction "*a?*" tests whether label '*a'* exists. Depending on the outcome of this query, the execution flow of the current agent can change as desired. Another functionality introduced is the ability to switch the local node's transmission power and/or the radio frequency channel as needed. These actions can be achieved by means of the execution operator '*$*'. By using the letter '*w*' on the left-hand side operand, and a valid numeric parameter on the right-hand side operand (e.g., "*w$7*"), an agent can increase or decrease the transmission power at will. (In the case of Crossbow Micaz hardware [12], valid numeric parameters are 3, 7, 11, 15, 19, 23, 27 and 31.) On the other hand, the node's radio frequency channel can be adjusted in the same fashion by using the letter '*f*' on the left-hand side operand, and a valid numeric parameter on the right-hand side operand. (For the Micaz, valid channel values are from 11 to 26 – e.g., "*f$15*".) These new functionalities built into the interpreter for enabling flexible topology reconfiguration of the WSN, can be readily employed to support diverse schemes being currently investigated in the area of multi-channel mesh networking. However, the WSN programmer must be careful in ensuring that the nodes are always reachable if the power and/or frequency operating values of one or more nodes are modified. In the next section, we elaborate on the agent migration procedures supported by WISEMAN's, and present concise examples to better understand its programming language and execution flow.

3 Migration Procedures and Agent Execution Flow

WISEMAN provides the means to migrate codes in three different ways: (1) by using explicit hop-by-hop codes, (2) by using values held in numeric variables, and (3) by means of a labelled (virtual) links. Each of these approaches has its advantages and

disadvantages, and will be explained here in detail. In addition, the execution flow of an agent can be conveniently controlled by means of WISEMAN's language rules, as explained shortly.

3.1 Code Migration Methodology

Explicit path hopping. Mobile agent systems targeted at WSNs (and otherwise) must provide the necessary means that allow agents to decide their hopping sequence through the network. In this case, it is assumed that agents are dispatched with an itinerary that meets the applications as defined by the agent's source node [13, 14]. This approach results practical in deployment settings where the conditions being monitored are for the most part stable. WISEMAN supports explicit path hopping by specifying the target node on the right-hand side of a hop operation. For example the instruction "*#1*" indicates an explicit hop to node 1 from the current location. By the same token, the segment "*#1;#2;#3;#4;…*" defines an execution sequence wherein the corresponding agent performs a hopping sequence that takes it to node 4 through nodes 1, 2 and 3, at which point a certain set of local operations are performed.

Variable-target hopping. Depending on different factors, it is possible that the agent's itinerary may need to be modified as it travels through the WSN. This implies that the hopping sequence might not be preset beforehand at the source node, but will be dynamically determined by an algorithm implemented into the agent's constructs instead. WISEMAN supports this functionality by allowing the use of either a mobile or a numeric (local) variable as the right-hand side of the hop operator. For instance the instruction "*#N1;…*" as parsed by the interpreter indicates that the agent will hop to the node whose numeric ID value is stored in the numeric variable *N1*. Evidently, the variable in *N1* must be set accordingly depending on the current circumstances, and may be updated by a separate agent/process. However, we note that this value is stored and maintained locally, and so modifying *N1* at, say, node 3 has no effect on *N1*'s value at, say, node 5. Nonetheless, WISEMAN provides the means to carry values that accompany the agent as it travels through the network. These values are referred to as mobile variables, as seen in Table 1, and are referenced in the same fashion when issuing a hop instruction (e.g., "*#M2*…"). Unlike numeric variables, mobile variables are temporarily stored at the local node where an agent arrives to. There, mobile variables may be modified by the agent that owns it, (e.g., "*M2+1*"), and their value is copied into the agent's header fields before being dispatched onto a different node. Whether numeric or mobile variables are used, the WSN operator must ensure that the values stored in the corresponding variables referenced by the agent to define their hopping sequence are properly maintained to avoid undesired effects.

Labelled path hopping. In certain cases, it may be desirable to allow an agent to forward copies of it to different nodes at once. For instance, a given node might have some semantic relationship with only a subset of neighbouring nodes to which one or more agents need to be forwarded. In this case, the numeric variable-target hopping explained before becomes inadequate since numeric and mobile variables can hold only one value. To address this issue WISEMAN supports the creation of labelled paths that can be employed to emulate multicast transmissions from the local node.

Labels are set in a pair-wise fashion between two nodes. However, one of these nodes can serve as a "pivot" node that can add the identity of a subset of nodes to the same label. For example, node 0 might have nodes 1 through 4 as neighbours, and the WSN operator may wish to create two multicast groups: one for the odd-numbered nodes, and one for the even-numbered nodes. In this case, the first labelled path can be set with the following codes: "*L=a;#1;#P;#3*". The labelled path is established by setting the *Link* environmental variable *L* to the corresponding value, and the predecessor environmental variable *P* is employed for defining the explicit path hopping sequence through nodes 0-1-0-3 as shown in Fig. 5. A similar agent is then dispatched with the corresponding values to set up the labelled path '*b*': "*L=b;#2;#P;#4*". After these initial steps, subsequent agents can be dispatched using the instruction "*a#*", or "*b#*" in order to reach the corresponding nodes 1 and 3, or 2 and 4. Note that labelled path hopping requires that the label identifier appears on the left-hand side of the operator. A clear advantage of this approach is that any agent that arrives later at node 0 does not need advance knowledge of the destination nodes' identities. It follows that labelled path hopping can be of either explicit-hopping or variable-target type. In the former case, a fixed label identifier is used ("*c#*"), whereas the latter case warrants the use of Character variables *C* (e.g., "*C3#*"), whose value can be modified as needed. It also follows that variable-based hopping can rely on different types variables, which are accessed by specifying their corresponding number at the right-hand side of it (e.g., "*N3*", "*M2*", "*C1*"). It is also evident that the labelled-based approach can be an advantageous approach to implement shorter agents. However, the overhead introduced by the label-setting process needs to be taken into account. In the end, a combination of the three agent migration procedures can be employed, and the resulting agents' codes should be evaluated to decide which approach is more desirable depending on the intended application. In the next subsection, we explain different ways to manipulate the process flow of an agent in WISEMAN.

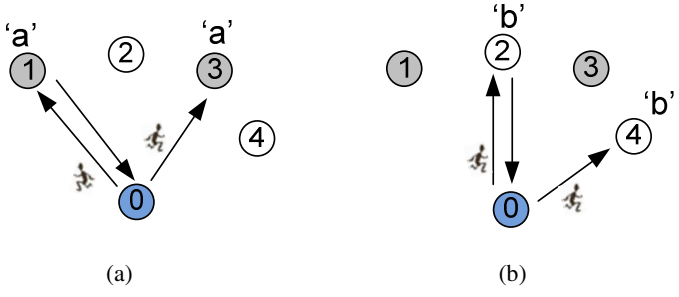

Fig. 5. Example of path-labeling for multicast transmissions: (a) neighbors 1, 3 are marked with label 'a'; (b) neighbors 2, 4 are marked with label 'b'

3.2 Agent Execution Flow

WISEMAN supports three basic forms of execution flow that can be programmed into the agents, and they are implemented by means of the *And*, *Or*, and *Repeat* rules of its language. These rules can be employed for condition-checking in order to make hopping decisions, as explained next.

Multiple condition fulfilment. The WISEMAN interpreter supports a language construct where all the instructions embraced by an *And* rule are evaluated to verify that they produce a successful outcome. If a single instruction within the rule returns false, then the whole rule fails and the agent's execution is aborted. For instance, the construct "*…A[N1>1;M2==9;a?]…*" will yield *true* only if numeric variable 1 is greater than 1, mobile variable 2 equals 9, and the labelled path 'a' exists at the local node. In practice, this rule has been seldom employed and is provided as legacy rule from the original Wave language.

Single condition fulfilment. Unlike the previous rule, single condition checking is carried out more often than not, and allows a programmer to implement widely-used if-else constructs by means of the *Or* rule. When employed, the agent's execution is continued as long as at least a single embraced instruction yields true. For instance, compared to the example seen in the multiple condition case shown above, the construct "*…O[N1>1;M2==9; a?]…*" will yield success if *any* of the embraced instructions yields true. Therefore, if the instruction "*N1>1*" is successful, then the remaining two instructions will not be evaluated. Else, the next instruction is subsequently evaluated, and so forth. If neither of the embraced instructions returns true, then the whole construct fails, and the agent stops executing.

Compound execution. Our experience has shown that an agent often needs to perform a specific instruction immediately after a certain condition is met. In addition, the action to take depends on which condition checking instruction yielded a *true* value. In these cases, compound operations can be employed to complement the operation of the previous rules. For instance, the codes "*…O[(M1> 3;#8);(N2=5;#7)]…*" indicate that, if the value contained in mobile variable 1 is greater than 3, then the agent will hop to node 8; otherwise, numeric variable 2 is set to 5, and the agent hops to node 7. It can be seen that the compound operators force the execution of all the embraced instructions. Otherwise, without using compound execution, the first instruction yielding a *true* outcome would prevent the rest of the following instructions from executing.

Condition cycling. This form of process flow implements a standard functionality that enables cycling through a set of instructions embraced within. However, WISEMAN defines an alternative means to execute cycles. In it, the codes embraced by a repeat rule *R* are extracted and inserted in front of the corresponding rule. Therefore, an agent program with the codes "*R{N1<20;a#}*" becomes "*N1<20; a#;R{N1<20;a#}*" when the repeat rule is encountered by the WISEMAN interpreter. The reason behind this approach is that it enables agent migration without having to worry about forwarding the agent's current execution state. In this particular example, once the first instruction is evaluated and the labelled hop instruction is processed, the only portion of the code that is actually forwarded to the next node is "*R{N1<20;a#}*", whereas the preceding codes that have already been executed are discarded. This self-depleting agent approach rids the system from having to forward program counters, register values, etc. that are normally associated with agent migration, which ultimately simplifies the operation of the interpreter, helps save bandwidth, and reduces agent forwarding delay.

4 Performance Evaluations

We performed a series of experiments to determine the performance of the WISEMAN interpreter implemented in TinyOS version 1.1 over Crossbow Micaz motes [15, 16]. These devices implement the Zigbee protocol for low-power communications, provide up to 128Kbytes of flash memory to store user programs, and 4 Kbytes of volatile RAM memory for variables' use. This hardware platform has as proven to be popular choice among researchers in the area of WSN, as it provides an ideal example of the resource limitations that have to be dealt with.

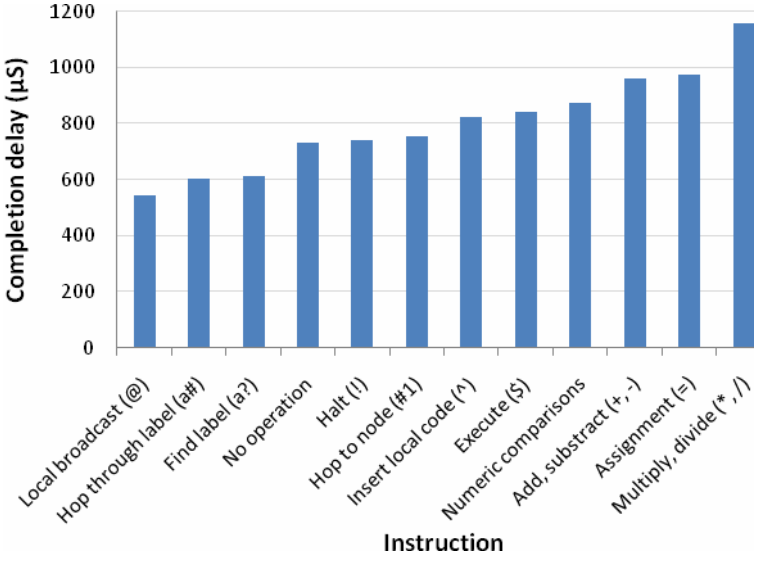

Fig. 6. Execution time for individual WISEMAN instructions

Fig. 6 illustrates the actual execution time incurred by individual WISEMAN instructions averaged over 1000 runs. We can see that the hop operations incur the lowest execution time, whereas the arithmetic operations yield the highest with an overall average execution time per instruction of around 800 µS. Compared to Agilla's instruction set, WISEMAN's instructions take longer to execute. We attribute this to WISEMAN's text-based codes' parsing and tokenizing delay, as mentioned in Section II. In addition, the lack of an arithmetic co-processor in the hardware of the Micaz mote is expected to introduce larger processing delays when the interpreter encounters codes that include the corresponding instructions. The processing delay shown for the execute operator *$* was obtained by averaging over LED, transmission power and frequency channel change operations, which are already built into TinyOS.

Fig. 7 shows the migration delay of a sample WISEMAN agent as a function of the number of Zigbee packets it spans. At first, the agent employed in our simulations was coded to fit in a single Zigbee packet. Then, the same agent was gradually

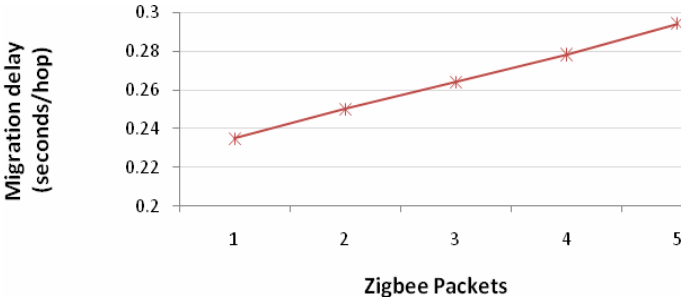

Fig. 7. Agent migration delay as a function of Zigbee packets incurred

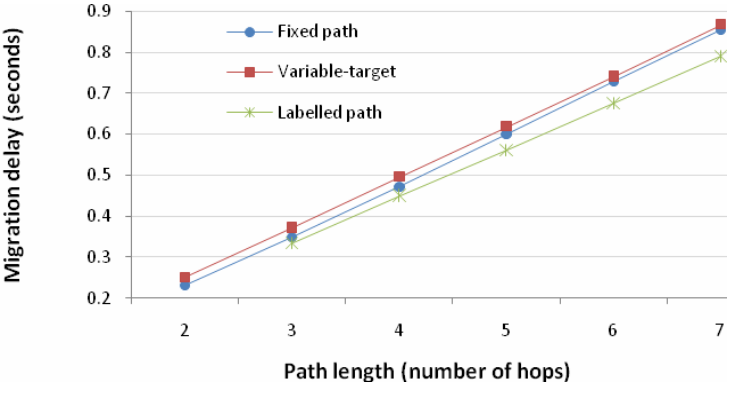

Fig. 8. Forwarding delay for distinct agent migration techniques

increased in size using arbitrary instructions to occupy up to 5 Zigbee packets in total, which is the maximum number that can be used before the 170-byte agent-size limit set for the current WISEMAN implementations is reached. The results shown are averaged over 100 runs for each case, and they provide a good estimate that can be referenced when creating WISEMAN programs for deployment in WSNs formed by Crossbow Micaz. We also note that the previous results were obtained by setting the transmission power of the sensor nodes to -15dbm, and the sensor nodes were placed at 10 cm from one another in a non-controlled environment with regards to the present RF signals.

Fig. 8 shows migration delay results for agents that employ the corresponding techniques explained in the previous section for a path length of up to 7 hops (that correspond to 8 Micaz sensors at hand). We see that the label-hopping technique yields the shortest migration delay, whereas the delay seen in variable-target method approaches the one experienced by the fixed-path hopping technique as the path length increases. It can also be observed that the performance of the variable-target technique approaches the one obtained using the fixed path method as a function of the number of hops, which reflects the growing agent size due to explicit hop instructions being added to the codes (e.g., "…*#1*…", followed by "…*#1;#2*…", and then "…*#1;#2;#3*…"), and so forth.

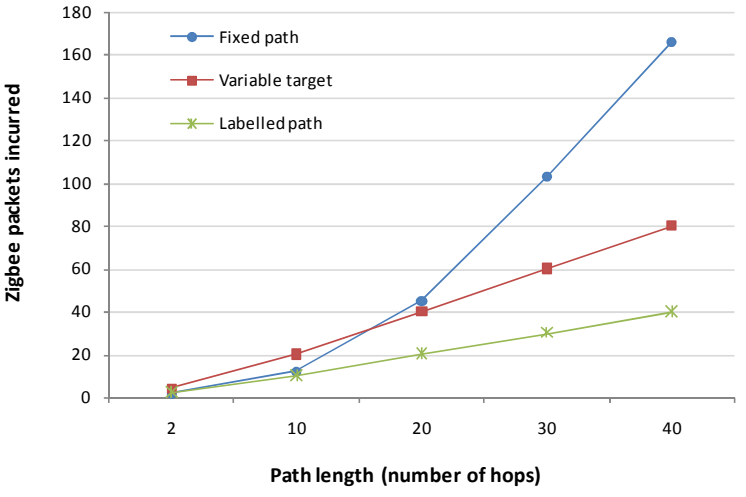

Fig. 9. Number of Zigbee packets incurred for distinct agent migration techniques

Fig. 9 depicts the number of Zigbee packets generated by forwarding by each of the three types of agents implementing the corresponding migration techniques being considered for up to 40 hops (limited by the maximum agent size of 170 bytes). These numbers were numerically calculated, and provide a compelling reason for using the labelled-path hopping technique whenever possible. We can also observe that the bandwidth overhead of an agent implementing the variable-target hopping technique is twice as long as the labelled based, with the fixed path approach yielding the worse performance. As expected, these results are consistent with the delay performance observations of Fig. 8. Thus, the increasing number of explicit hop-by-hop instructions has a direct effect on both the migration delay and the incurred bandwidth as the agents' size occupies additional Zigbee packets. The longer the explicit hop-by-hop sequence, the more Zigbee packets that are needed to accommodate the corresponding codes. Based on this figure, we can see that explicit hop-by-hop migration is not as costly when hopping through a short path, but its use becomes otherwise detrimental when hopping through longer paths, in addition to being a more inflexible approach that is unsuited for environments monitored by the WSN that may change unpredictably.

5 Conclusions

In this paper, we have provided a more detailed insight of the WISEMAN system for the deployment of mobile codes in WSNs, and described a number of features that make our proposed system a viable solution to enabling programmability in this type of networks. Our performance evaluations show that the efficiency in terms of migration delay and bandwidth consumption of a WISEMAN agent script may depend heavily on how the program is designed. This performance depends in part on the size of the WSN, and so the longer the path that an agent has to traverse, the higher the impact that its program structure will have on the overall WSN

performance. In particular, we have shown that employing the explicit path-hopping approach can have a significant impact in bandwidth usage for large WSN, whereas smaller WSNs are not as vulnerable to the migration methodology employed. However, even though labelled-path hopping provides the most energy-efficient approach to agent migration, it should be noted that a deployment scenario in which the underlying conditions warrant constant label-maintenance sub-tasks might incur increased bandwidth consumption overhead. If this is the case, then variable-target hopping rivals labelled-path hopping. On the other hand, the effectiveness of variable-target hopping depends on how an agent's codes are structured, thus putting into play the skill of the WSN programmer, who decides how WISEMAN's execution flow constructs are used. In addition, it is possible that the task being carried out by the WSN is complex enough to require injecting multiple agents, since memory availability hinders the creation of a single larger agent that accomplishes all the required objectives. In such case, a programmer may rely on distinct migration techniques and execution flows to accomplish the goal in the most efficient manner.

Acknowledgments. This project was supported in part by the National Sciences and Engineering Research Council of the Canadian Government under grants STPGP 322208-05 and 365208-08, and by KRCF and the Ministry of Knowledge Economy, Korea, under the Information Technology Research Center support program supervised by the IITA (grant number IITA-2009-C1090-0902-0006).

References

1. MacRuairi, R., Keane, M.T., Coleman, G.: A Wireless Sensor Network Application Requirements Taxonomy. In: Proceedings of the Second International Conference on Sensor Technologies and Applications, SENSORCOMM, Cap Esterel, France, August 25-31 (2008)
2. 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)
3. Liu, T., Martonosi, M.: Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems. In: Proceedings of ACM SIGPLAN: Symposium on Principles and Practice of Parallel Programming, San Diego, USA (June 2003)
4. 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 (November 2004)
5. Boulis, A., Han, C.-C., Srivastava, M.: Design and Implementation of a Framework for Efficient and Programmable Sensor Networks. In: 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.: Smart Messages: A Distributed Computing Platform for Networks of Embedded Systems. The Computer Journal, Special Issue on Mobile and Pervasive Computing, Oxford Journals 47(4), 475–494 (2004)

7. 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)
8. Chen, M., Gonzalez, S., Leung, V.: Applications and Design Issues of Mobile Agents in Wireless Sensor Networks. IEEE Wireless Communications Magazine 14(6), 20–26 (2007)
9. González-Valenzuela, S., Chen, M., Leung, V.C.M.: Design, Implementation and Case Study of WISEMAN: WIreless Sensors Employing Mobile AgeNts. In: Proceedings of the 2nd International ICST Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications (MobilWare), Berlin, Germany (April 2009)
10. Sapaty, P.: A Wave Language for Parallel Processing of Semantic Networks. Computers and Artificial Intelligence 5(4) (1986)
11. Sapaty, P.: Mobile Processing in Distributed and Open Environments. John Wiley & Sons, Chichester (2000)
12. Crossbow Technology, http://www.xbow.com
13. Min Chen, T., Kwon, Y., Yuan, Y., Choi, Y., Leung, V.: MADD: Mobile-agent-based Directed Diffusion in Wireless Sensor Networks. EURASIP Journal on Applied Signal Processing (2007), doi:10.1155/2007/36871
14. Chen, M., Leung, V., Mao, S., Kwon, T., Li, M.: Energy-efficient Itinerary Planning for Mobile Agents in Wireless Sensor Networks. In: IEEE ICC 2009, Dresden, Germany, June 14-18 (2009)
15. TinyOS for wireless embedded sensor networks, http://www.tinyos.net
16. The Wiseman Agent System for WSNs, http://www.ece.ubc.ca/~sergio/wiseman