WISEMAN
Ver. 0.91

Data Communications Group
Department of Electrical and
Computer Engineering -
The University of British Columbia

Prepared by:
Sergio González-Valenzuela

December 8, 2008

ABSTRACT

We introduce WISEMAN (Wireless Sensors Employing Mobile AgeNts), a mobile-code system for the programmable control of distributed tasks in wireless sensor networks. WISEMAN's architecture and language constructs decouple the coordination element of distributed processes from the actual data processing in order to produce ultra-compact agents that may help to reduce bandwidth utilization. WISEMAN also allows the dynamic creation of labelled links between WSN nodes, which facilitates agent navigation and propagation. We present the rationale behind WISEMAN's, and important details of its implementation in Crossbow® Micaz.

1 INTRODUCTION

The use of code mobility has gained significant attention as a plausible alternative to address energy-conservation issues in Wireless Sensor Networks (WSNs). The selling point for the use of code mobility here is twofold. First, mobile code allows the network to be conveniently re-tasked according to the user needs. Second, a programmable approach enables the data computation element of an application to be re-located to the site where relatively large amounts of data were collected, enabling potentially high energy-savings.

Existing agent approaches for WSN attempt to achieve a balance between the degrees of functionality incorporated into the actual code interpreter, and the one provided to the agents. On the one hand, a coarse-grained agent system incorporating a high degree of functionality in the interpreter would require simple constructs on the agent side in order to accomplish a certain task. For example, a coarse-grained language construct might look like: <execute task A; execute task B; end program>. On the other hand, a fine-grained code interpreter with little or no functionality would require agents with a language construct comprehensive enough to cover all of the node's own machine code functionalities. As an example, a fine-grained language construct might look like: <move X,Y; add Y,2; XOR X,Y; … >. Thus, the degree of granularity that language constructs will be provided should clearly be a function of the intended WSN's application.

Given the application-specific and task-centered nature of WSNs, it makes little sense to create MAS that promote overly detailed control of the system's functionalities if the tasks to be performed are consistently repetitive. In other words, since the objective of the WSN nodes is all the same, intuitively, MAS deployed here should provide the necessary functionality to support coordination of distributed tasks, whatever that might be.

WISEMAN stems from an earlier system that addresses this issue very effectively. In fact, the *Wave* system can be considered one of the earliest precursors of code mobility in data networks, with its foundations lying on the idea of efficient task coordination in distributed environments. As a result, WISEMAN inherits many of Wave's original traits, including its compact language, which is highly appealing in WSN. In the following sections, we describe in detail the architecture of WISEMAN, which is a condensed version of the original Wave system, its language constructs and its functionalities.

2 SYSTEM ARCHITECTURE FOR WISEMAN

The initial implementation and testing of WISEMAN was carried out in the OMNeT++ Discrete Event Simulator, and was later ported to NesC for running in TinyOS. WISEMAN's architecture is shown in Figure 1, and is based on a significantly scaled-down version of the original Wave interpreter. The data flow within the interpreter is depicted by bold arrows, whereas dashed lines show the interactions between its various elements. In essence, the WISEMAN's interpreter is comprised of: an incoming agent queue, a codes' parser, an instruction execution block, an engine block, and an agent dispatcher.

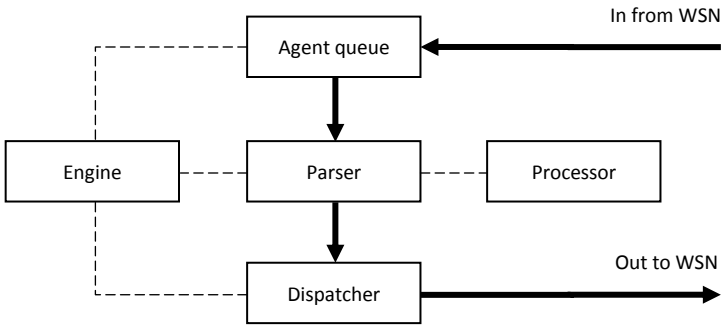

Figure 1 WISEMAN's architecture

The incoming agent queue accepts agents arriving either from other nodes through the wireless link and works in a simplistic first-input first-output fashion. All arriving agents are immediately inserted to the back of the queue, whereas agents ready for processing are removed from the head.

The parser manipulates WISEMAN codes that are written in text-string form. Initially, the parser removes a single agent from the incoming queue and separates code and data. The code is then separated into two segments, hereto referred as head and tail. The head is the first fragment of code that appears before either an operation or precedence delimiter as defined by the language constructs is encountered. Consequently, the rest of the code that follows is referred to as the tail. The head is recursively unwrapped from within any precedence delimiters or top-syntactic operations, stopping once a single indivisible operation is found. This indivisible operation is subsequently sent to the execution block for further processing. The parser continues to send the next indivisible operation as indicated by the execution block if the previous one is successfully processed.

The processing block's task is to carry out any operation indicated in the codes. The outcome of an operation is defined either as success or failure, and is signalled back to the parser. In addition, the processing block handles any function calls in response to an agent's need for information processing at the current node. To this regard, the agent must know in advance the function name or identifier that the execution block shall use. The agent can employ the value or values returned by the function, if any, in order to make further decisions that affect the future execution and coordination steps of the distributed process.

The agent execution process is halted when an operation cannot be successfully completed, a node hop operation is encountered, or explicit process termination is indicated. In the first case, the agent' tail is discarded and the agent simply terminates executing, unless the head is contained within a language construct that instructs the parser to proceed otherwise. Else, the parser sends the next indivisible operation to the execution block and the process continues as defined by the agent's code construct. In the second case, the agent may instruct the execution block to forward the agent to another node, and so the tail of the agent is sent to the propagation block for dispatching. For the third case, the agent may simply instruct the interpreter to explicitly halt the current process and terminate the agent as needed.

The dispatcher block is responsible for forwarding agents between nodes. In doing so, this block arranges for all of the required data and code to be marshalled together before the agent propagates. When an agent is forwarded, the dispatcher first ensures that the agent's transfer is actually possible by looking up the destination node in a local table that contains a list of the neighbouring nodes. The agent will thus be dispatched out of the interpreter only if the intended target node's ID number is found on the local table. Otherwise, the agent is discarded. The dispatcher block signals the parser once an agent has been forwarded, which in turn may proceed to remove the next agent from the incoming queue, if one exists.

The Engine block does not necessarily perform the operations normally observed in similar elements of other software systems as the name implies. However, it implements a number of procedures and algorithms that do not belong elsewhere in the interpreter. Thus, the Engine block provides general-purpose functionalities and serves as a repository of information that is required by the other blocks of the interpreter at different stages during the codes' processing. This block is also employed as the interface between the interpreter and the node's sensor functionalities.

The maintenance of the neighbouring nodes' table is to be performed by the interpreter's interaction with the local node's communications system through conventional means, but can also be updated by the agents as needed. We also note that the neighbouring table not only indicates whether a node is reachable, but it may also be used to establish semantic relationships from the current node. The purpose of this functionality is to enable the creation of virtual links that the agents can employ in their WSN navigation process.

3 LANGUAGE CONSTRUCTS

WISEMAN's language constructs are comprised by: variables, operators, control statements, and delimiters, most of which are derivations from the original Wave language. However, several changes have been made in order to facilitate codes' parsing and processing, and to simplify the structure of the interpreter itself, leading to a smaller memory footprint. The following subsections explain in detail the particular aspects of these constructs.

3.1 Variables

In WISEMAN, the interpreter employs five kinds of variables, all of which employ pre-assigned memory segments in the local node (there are no provisions for dynamic memory reservation). Table 1 depicts the type of variables available in WISEMAN.

Table 1 WISEMAN variables

Identifier	Description
N	Numeric
C	Character
M	Mobile
B	Clipboard
P	Predecessor
L	Link
I	Node ID

- ***Numeric***. Numeric variables are represented by the letter *N*. Each WSN node has its own pool of numeric variables. However, given memory constraints in WSN nodes, the actual number of Numeric variables is predefined to a certain value, for instance from *N0* to *N12*, where *N* stands for Numeric and the adjacent number is a simple identification index for the variable. The actual meaning of the variable depends on what the programmer decides to use it for. Value assignment to Numeric variables follows the traditional form, (e.g., "*N3=1*"), but compound arithmetic operations are strictly binary, as in "*N6+2*", which adds 2 units to the value already stored in variable *N6* (if any). Therefore, expressions such as "*N2=N1+5*" are invalid. However, multiple variables can be employed to achieve the same result, if needed. This helps to reduce the complexity in both the parser and processing modules at the expense of a slightly lengthier agent codes. Numeric variables are semantically similar to public variables defined in object-oriented programming, meaning that all

agents that arrive to the interpreter have access to them. Manipulation of Numeric variables at the local node has no effect on the variables of remote nodes. In addition, all variables are expected to maintain their semantic meaning across the WSN according to how they are individually manipulated by programmers through the agents.

- ***Character***. This kind of variables is similar to Numeric, except that these are defined for storing single characters through the letter *C* (e.g., "*C7=d*"). Arithmetic operations over characters are not defined/supported. Otherwise, the same rationale of Numeric variables applies.
- ***Mobile***. These variables are represented by the letter "*M*" (e.g., "*M2=2*"), and they resemble the role of private numeric variables in object-oriented programming. Mobile variables accompany agents as they hop from node to node. An agent may modify its mobile variables as needed, but cannot modify the values of other agent's mobile variables. These variables are temporarily stored at the current node when an agent arrives, and they are erased as soon as the agent hops to another node. Therefore, an agent's manipulation of its Mobile variables has no effect on other agents' Mobile variables. All agents carry with them the same amount of Mobile variables. This amount is predetermined, and must be adjusted at all WSN nodes if changes are made to the interpreter's default values.
- ***Clipboard***. The *Clipboard* variable *B* can be employed by agents or by the interpreter to temporarily store numeric, as explained later in this manual.
- ***Environmental***. The WISEMAN interpreter is provided with two environmental variables. The *Link* variable "*L"* can be employed to establish semantic relationships between WSN nodes. This variable can be set with the desired value before an agent hops to another node (e.g., "*L=s*"). Upon encountering a *Link* assignment operation, the interpreter will add the corresponding label to the local table as an additional identifier for the node in question. At the same time, the interpreter appends this new label to the agent before it hops to the next node so that, upon arrival, a corresponding label can be assigned resulting in a symmetric label assignment. Finally, the "*L*" variable needs to be reset to 0. In other words, the "*L*" variable is turned on when setting up labelled paths between nodes, and then turned off when the process is finished. The *Predecessor* variable "*P*" simply holds the identification number of the last node visited by an agent. Both of these variables accompany the agent as it hops from node to node, although the *Predecessor* variable is an only-read type, whereas the Link variable can be either read or written to. Both of these variables are re-written at the local interpreter every time an agent is removed from the

incoming queue for processing. Identity *I* is a read-only variable, whose content is defined by the local node's identification number (i.e., 1, 2 …).

3.2 Operators

Table 2 shows the operators defined for WISEMAN. The first five operators are standard comparison operators, which like the previous two return *true* or *false* Boolean values that can be readily evaluated (e.g. "*M3<45*"). The next four operators can be employed to perform simple arithmetic operations that agents use on for simple calculations of values read by the local sensors, when updating hop counts or for time-related calculations. For example, "*N3*4*" multiplies the existing value of *N3* by 4, and the result is re-assigned to *N3*. The same applies for all arithmetic operators. The assignment operator "*=*" provides standard functionality for assigning values to variables.

Table 2 WISEMAN operators

Identifier	Description
<	Less than
<=	Equal or less than
==	Equal to
=>	More than or equal to
>	More than
!=	Different to
+	Add
-	Subtract
*	Multiply
/	Divide
=	Assign
#	Hop
$	Execute
!	Halt
@	Local broadcast
^	Insert script

The *Hop* operator is employed to indicate that the agent needs to be forwarded to the node specified on the right-hand side of the "*#*" character, or to a subset of nodes that are logically inked to the local node by means of a predefined label as indicated on the left-hand side of the operator. For example, the code segment "*#2*" indicates a direct hop to node 2 from the local node. On the other hand, the segment "*s#*"

indicates that the agent will be forwarded to all nodes logically linked to the local node with the label "*s*". This latter usage implies multicasting capabilities for WISEMAN. So, for example, if node 1 is physically tethered to nodes 2, 3 and 4, then an agent can setup a virtual link (or logical association) to, say, nodes 3 and 4 by employing a label "*s*". Later on, other agents need not know the identities of the nodes they are supposed to hop to, but instead may use the corresponding label to reach nodes 3 and 4 from node 1 by employing the code "*s#*". Evidently, the semantic connotation of the letter "*s*" is defined by the programmer. In any case, the interpreter will automatically clone the agent with as many copies as outgoing virtual links exist in accordance to the left-hand operand. That is, if there are 3 such virtual links labelled as "*s*", then 3 identical copies will be forwarded by the dispatcher. Alternatively, a copy of the agent may be the broadcast to all immediate neighbours by employing the *@* operator. The *Execute* operator "*$*" instructs the interpreter to call upon a local function. In this case, the function identifier is specified on the left-hand side of the operator, whereas the right-hand side can be used to pass a parameter. For instance the code "*l$r*" instructs the interpreter to access the hardware interface of the local node's and switch on the red LED. Table 3 shows the functionalities available in the current WISEMAN version. The "*!*" operator indicates an explicit process halt of the current agent with success if the right-side operand is 1 (i.e., "!1"), or failure if the operand is 1 (i.e., "*!0*"). Finally, the local injection operator "^" indicates that a local agent string with the identifier defined in the right-hand side of the operator will be injected at the local node after a delay of *t* seconds (e.g., "*2^0"*).

3.3 Control Statements

Table 4 illustrates the identifiers for the compound operators defined in WISEMAN. The *Repeat* "*R*" control statement indicates that the codes embraced by curly brackets will be repeatedly executed (e.g. *"R{segment1;segment2;…}*"). The *Or* and *And* control statements are provided as a way to manipulate the execution of an agent's constructs by testing whether the code delimited by square brackets yields a true or false value for every code segment it includes. Therefore, an "*O[…;…;…]*" compound segment indicates that the individual codes separated by semicolons "*;*" will be sequentially executed, stopping as soon as one of these segments results in a true value. Otherwise, the whole construct returns false and the agent's process stops. The same applies for the *And* rule "*A[…;…;…]*", except that all of the segments must return true in order for the whole rule to return true as well. Finally, codes embraced by round brackets "*(…)*"are executed in a compound fashion, as shown in the examples later.

Table 3 Local functions accessed through the execute parameter

Function	Parameter	Description
l	r	Switch on the red LED
	y	Switch on the yellow LED
	g	Switch on green LED
	e	Switch off red LED
	l	Switch off yellow LED
	i	Switch off green LED
	d	Toggle red LED
	w	Toggle yellow LED
	n	Toggle green LED
p	0-n	Enable/disable the photo sensor. If argument is >0, a timer with period 'n' is triggered to read the corresponding value. If argument is 0, the timer is disabled and the sensor is switched off.
r	p	Get the raw value of the latest photo sensor reading. The reading is stored in the clipboard 'B'.

Table 4 WISEMAN control statements

Identifier	Description
R	Repeat
O	Or
A	And

3.4 Code Delimiters

As seen in examples above, the main delimiter employed to separate code segments is the semicolon ";". The use of round brackets indicates compound segments' execution, whereas square and curly brackets have been designed to delimit code segments whose execution depends on a control statement construct, as explained before. Distinct types of brackets are employed since they facilitate the Parser's task when tokenizing nested code segments for processing (e.g., "*R{…O[…(…)…]…}*"). The

sole use of a single type of brackets (e.g., "*(…)*") would have implied extended requirements in the Parser's functionality, and therefore, added overhead.

4 PROGRAMMING IN WISEMAN

4.1 Program Structure

As mentioned before, WISEMAN programs follow a simple structure in which code segments are separated by a semicolon (e.g., "*segment 1;segment 2;…*"). The simplest type of program that can be created involves the use of simple operations at a local node (e.g., "*N1=1;N1<2;l$r*"). However, these programs serve no useful purpose. In reality, WISEMAN programs will almost always involve the use of the hop operator "*#*" and the use of control statements that provide the means to introduce some meaningful functionality to the programs. In general, programs with a well-defined execution flow usually feature the use of one or more "*And*/*Or*" control statements. Alternatively, programs with non-deterministic execution flows employ the "*Repeat*" control statement, and run continuously until certain condition is met and the program terminates. An important note is that WISEMAN's control statements have precedence levels, and can only be combined according to the following rules:

a) Illegal codes statements will cause the WISEMAN interpreter behave erratically, or halt altogether.
b) Nesting control statements is not allowed. For example, program segments like: "*R{…;R{…};…}*" or "*O[…;O[…];…]*" are illegal.
c) *Repeat* has precedence over *And/Or*, and *And/Or* have precedence over *(…)*. Therefore, enclosing a control statement of greater precedence inside one of lower precedence is illegal. (as in "*O[…;R{…};…]*").
d) The *hop* operator can only be used without control statements, or within *Repeat*. Enclosing the *hop* operator in any of the other control statements is illegal.

Though the constraints introduced by these rules might seem counterintuitive, they help to simplify the architecture of the interpreter significantly, which in the current WISEMAN version yields less around 17KB of ROM memory. This leaves space for the implementation of other local functions or data processing algorithms as needed.

4.2 WISEMAN Installation

The current WISEMAN implementation is comprised by a single project named *Virtual Machine*. The binary image of the Virtual Machine project can be installed in as many additional Crossbow motes as desired. It is recommended that the motes be programmed with the lowest RF power setting (i.e., 3) to save battery power when running experiments. It is assumed that the programmer knows the physical

location of the motes when running experiments, so that the approximate WSN topology can be inferred. This is necessary when creating programs where explicit hop operations are employed (e.g., "*#1;…;#4*"). Once the binary image is installed, all of the motes with an ID greater than 0 remain idle, waiting for agents to arrive. Upon being switched on, the base mote (i.e., with ID 0) will load and process pre-programmed agents, which will forwarded onto other motes accordingly if so indicated.

4.3 Programming Example

> Note: The current WISEMAN implementation only supports agent scripts no longer than 170 characters.

Multiple WISEMAN agents can be sequentially injected into the WSN from the gateway node. These codes need to be pre-programmed into the AgentQueueM module module around line 130, and can be changed with new ones as desired. For instance, we can program the base mote with the following codes:

"*L=a;#2;L=b;#3*"

"*l$r;a#;l$y;b#;l$g*"

When base mote is powered up, the first agent is injected in the WSN. The *L* variable is set with the label '*a*', and the agent sets up a virtual link with this label between motes 1 and 2. Then, the *L* variable is reset to label '*b*' and it is used to set up another virtual link between motes 2 and 3. One second later, the next agent is injected also by the base mote 0 into the WSN through mote 1. This agent will navigate through the virtual links just set up and turn on local LEDs as indicated. The user should change these values to other arbitrary values to understand this process better.

In another example, the agent represented by the codes "*R{M0+1;M0<100;l$w;#}*" will first instruct the base mote to add 1 to the current value of the mobile variable M0. Then, if the value in M0 is less than 100, the agent instructs the interpreter to toggle the yellow LED just before hoping onto another mote. Assuming there is only one mote in radio proximity, the agent will be continuously forwarded back and forth from between the corresponding mote pair until the "M0<100" is not met, and the agent is simply discarded. Subsequent agents can be injected into the WSN for further processing. These agents are removed from the queue at a pre-programmed rate of 1 per second (but this value can be changed).