Programming distributed applications with symbolic reasoning on WSNs

S. Gaglio, G. Lo Re, G. Martorella, D. Peri

In Proceedings of the 2015 International Conference on Computing, Networking and Communications (ICNC)

Programming Distributed Applications with Symbolic Reasoning on WSNs

Salvatore Gaglio
salvatore.gaglio@unipa.it

Giuseppe Lo Re
giuseppe.lore@unipa.it

Gloria Martorella
gloria.martorella@unipa.it

Daniele Peri
daniele.peri@unipa.it

Abstract

Programming Wireless Sensor Networks (WSNs) is a complex task for which existing approaches adopt rigid architectures that are only suitable for specific application fields. In previous papers we introduced a programming methodology and a lightweight middleware based on high-level programming and executable code exchange for distributed processing on WSNs. In this paper, we show how high-level programming can be effectively used on WSNs to implement symbolic reasoning. In order to prove the feasibility of our approach, we present a Fuzzy Logic system where the value updates and the rule evaluations are performed in a distributed way. Through the proposed methodology, we discuss the development of an Ambient Intelligence application. In particular, we describe how the nodes of a WSN may compute an estimation of the user thermal comfort by exchanging symbolic rather than numerical data, and control an HVAC (Heating, Ventilation and Air Conditioning) system accordingly.

1 Introduction

Wireless Sensor Networks (WSNs) are composed of nodes with constrained resources in terms of power, and processing capabilities. Although in literature several contexts for WSN applications have been proposed [1], the real adoption of such networks is hindered by the difficulties in their practical implementation. Existing approaches characterized by rigid architectures are only suitable for specific application fields, and cannot be easily extended to support a broader range of application domains. However, the tiny devices composing the network may be also used to interact with people and to treat high level knowledge [2], although nodes' constraints leave the system designers many challenges to face, especially when distributed applications are considered [3]. Providing abstractions that can be used to incorporate high-level processing of more structured and symbolic data than those resulting from mere logging of a few key physical quantities may prove impractical with the standard programming methodologies. One possible answer to this issue is to provide the application with an interpreter for some high-level language [4]. Unfortunately, such a strategy, if carried out with current WSN programming environment, would dramatically increase the processing load on the on-board microcontrollers and detach the application from the hardware. As an added consequence re-uploads of the application code could not be excluded should the hardware-abstraction layer require modifications. In some previous works we showed that programming WSNs with an extensible high-level language without compromising code execution efficiency is feasible by adopting a programming methodology based on Forth and we introduced a lightweight middleware for WSNs based on it [5, 6]. In this paper we show how our middleware can be easily extended with the ability to process symbolic data. Then we describe the implementation of a distributed Ambient Intelligence (AmI) application to control an HVAC system according to the user thermal comfort.

2 Methodology

Unlike current WSN programming models, our methodology is based on interactive development directly on wireless nodes. Interactivity avoids cross-compilation and proves time-saving since code draft and test phases collapse into a single step. The twofold nature of our approach integrating interpretation and compilation in the development cycle makes provisions for extensions of the language to accommodate new syntactic constructs while permitting interactive programming of nodes through the command line. We believe that the possibility to program nodes interactively is particularly useful when designing intelligent applications, as code can be tested on the real hardware and with real sensory data. However, the adoption of a high-level language does not prevent simulations from being carried on. Indeed, the same high level code can be made run on simulation hardware. In our work we adopted the Forth programming model. Forth is a stack-based language that is easily extensible by defining new *words* in the so-called *dictionary*. Forth is both interpreted and compiled, and it is possible to extend both the interpretation and compilation semantics of the language. Words can be tested interactively to verify their correctness. A Forth program thus simply consists in a sequence of words that can be made similar to natural language sentences. The symbolic approach is therefore inherent in the programming paradigm we adopted since each program is a sequence of "executable symbols". Following this approach, we implemented a software platform for WSNs that permits to define procedures and services according to the application target, and simultaneously test them. Nodes can be reprogrammed when needed, even after network deployment. Even though our programming environment currently targets the IRIS mote WSN platform, due to the high-level programming approach, porting it to other hardware is straightforward. A simple but powerful mechanism that is central to our methodology is executable code exchange. A node can send code to another node or broadcast it to the network through the following construct:

`( destination-node -- )` `tell:` ⟨*code*⟩ `:tell`

Code between `tell:` and `:tell` keywords is sent to the destination node whose address is expected on the top of the stack, as can be seen in the Forth standard parentheses surrounded *stack effect notation* in the synopsis. The constructs works in Forth's interpreted mode. A compilable version is shown in Listing 2 (line 9).

The word `bcst` can be used to leave the broadcast address on top of the stack. Code is sent as textual strings as payload of IEEE 802.15.4 MAC frames [6]. On the receiver side the payload is interpreted as if it had been typed at the command line interface. We exploited this feature in the implementation of a Timing-Sync protocol [5], and to remotely program deployed nodes in an interactive manner through a command line interface [6].

3 Distributed Symbolic Reasoning with Fuzzy Logic

Integrating readings of all the nodes is one of the main tasks of WSN applications, though processing is often carried out in a centralized manner. In our approach we alternatively propose to make full use of the computing resources of nodes and implement distributed applications with our high-level programming methodology. In order to demonstrate how the concepts outlined in the previous section can be used to incorporate new abstractions in our programming environment here we recur to a system based on Fuzzy Logic. We decided to use the Fuzzy Logic formalism since it is particularly suited to implement qualitative and imprecise reasoning, and has been widely used in AmI applications [7, 8]. When information is expressed through crisp data it is possible to model the convergence to a single shared value through successive modification of the initial state toward the average of all the values held by the nodes:

$$x_t(t+1) = \sum_{j=1}^{N} W_{i,j} x_j(t). \tag{1}$$

where $W_{i,j}$ represents the weight related to the information held by the j-th node. Averaging is widely used in distributed WSNs applications to solve several problems, such as clock synchronization[9]. On the contrary, our approach permits to achieve consensus among nodes through the exchange of symbolic and qualitative description of the observed phenomenon, as in the natural language [10]. There are many implementations of fuzzy controllers in literature with diverse formalisms. In this work we adopted the formalism described in [11] extending it to support both local and remote operations on fuzzy values.

Listing 1: Remote definition of the fuzzy variable `temp` and creation of the membership functions with the specified labels. In this example the message is broadcast using the special `bcst` address through a bridge node

```
bcst tell:
  0 50 fvar temp
```

Table 1: Summary table of used words. Words are indicated together with their “stack effects”

Word (before -- after)	**Description**
translate-IR (--)	Process the last received IR signal to update the user comfort level estimate, as described in Fig. 4
temp-read (-- temperature-val)	Leave the temperature sensory reading on the stack
apply (crisp fuzzy-var --)	Fuzzify the crisp input of a certain fuzzy variable.
conclude (fuzzy-var -- crisp)	Defuzzify and return the crisp output
fvar (min-val max-val ”name” --)	Define a new fuzzy variable with the provided validity range
member (lm lt rt rm --)	Define a new membership function according to the control points on the stack
favg# (--)	Signal that the update will concern the number of samples of the average
favg (fuzzy-var ”name” --)	Define a new structure to store the fuzzy average of truth values concerning the fuzzy variable on the stack
favg-update (fuzzy-val pointer --)	Updates either the number of samples or the average value bound to the membership function
fvar-update (fuzzy-val membership-addr --)	Store the fuzzy truth value of the membership on top of stack
fvar-remote-update (dest-addr fuzzy-var --)	Send a message to *dest-addr* to update the fuzzy truth values of the linguistic labels referring to the fuzzy variable *fuzzy-var*.
favg-local-update (fuzzy-var favg-addr --)	Locally updates the fuzzy average accordingly to the fuzzy-var membership values.
favg-remote-update (dest-addr favg-addr --)	Send a message to *dest-addr* containing the code to update the fuzzy average on the remote node.

```
3   0  0 15 22 member temp.low
4  20 25 25 30 member temp.medium
5  28 35 50 50 member temp.high
6 :tell
```

Due to the hardware limitations of the microcontroller, fuzzy truth-values are represented with integer numbers. This is not a limiting factor, though as arbitrarily precise computations can be carried out with fixed point values adopting integer scaling factors. In this case, following the convention in [11] the maximum fuzzy truth value is 255. By exploiting executable code exchange a fuzzy variable definition can be easily distributed among nodes even after their deployment. For instance, listing 1 reports the command line input that broadcast (`bcst`) the code to define the fuzzy variable `temp` and its support (line 2) in the range [0,50] Celsius degrees, along with three trapezoidal membership functions (lines 3-5). Four control points are used to define the shape of a membership function (Fig. 1). After such definition all the nodes can evaluate, exchange and apply rules to fuzzy temperature values. For instance, a node can measure its local temperature and fuzzify it with the code:

```
temp-read temp apply
```

then make the others update their fuzzy temperature values accordingly:

```
bcst temp fvar-remote-update
```

The command evaluates the `temp` variable and broadcasts a message containing code that updates the three membership values. The word `fvar-remote-update` creates a message whose content is the repetition of the pattern

```
<truth> <membership func> fvar-update
```

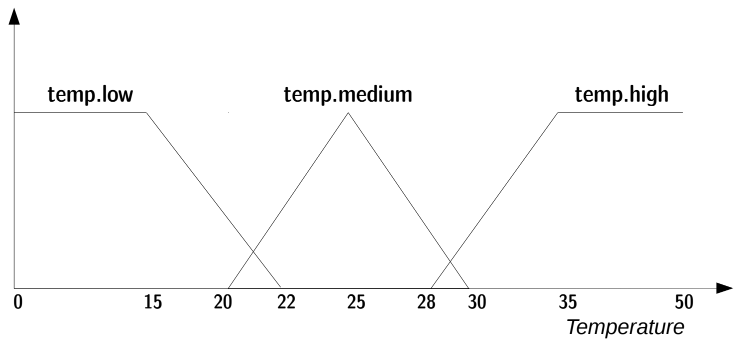
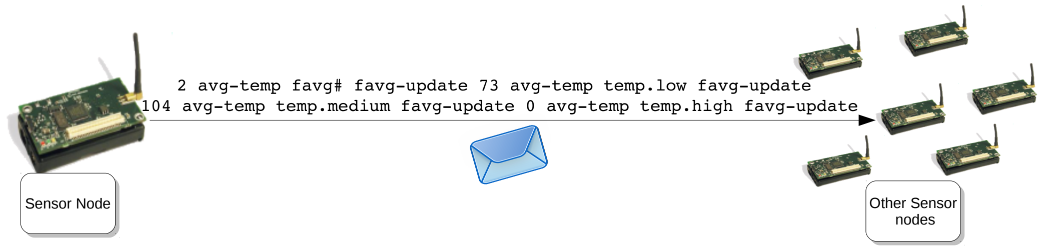


Figure 1: The three membership functions of the fuzzy variable `temp`. Four control points (bottom-left, top-left, top-right, and bottom-right) are used to define the shape of a membership function so that triangular and asymmetric trapezoidal shapes are defined by repeating one control point value as shown in Listing 1.



```
 2 avg-temp favg# favg-update 73 avg-temp temp.low favg-update
104 avg-temp temp.medium favg-update 0 avg-temp temp.high favg-update
```

Figure 2: Executable code contained in the message sent by the word `favg-remote-update`. The word `favg-update` is used to update the number of samples (`favg#`) or the average truth value of a given membership function.

for each membership function of the argument fuzzy variable– `temp` in this case. When a node receives the message, it interprets the command updating the truth values of its local `temp.low`, `temp.medium` and `temp.high` membership functions. Once the basics of fuzzy values exchange and evaluation are set, it is quite easy to use fuzzy symbol processing for distributed processing. Let us suppose we intend to build a thermostat application in which fuzzy temperature values from the network nodes are averaged and used as input to the control rules. A *fuzzy average* variable is then defined on all the nodes as:

```
bcst tell: temp favg avg-temp :tell
```

The word `favg` defines the new variable `avg-temp` binding it to the previously defined variable `temp` to store the fuzzy average temperature. The variable `avg-temp` then stores an average truth value for each membership function of the bound variable – `temp` in this case. The average truth value bound to a membership function is updated by stating its new value:

```
30 avg-temp temp.low favg-update
```

and the new number of samples of the average

```
   1 avg-temp favg# favg-update
```

When the local update is completed it is propagated to the other nodes:

```
bcst avg-temp favg-remote-update
```

The word `favg-remote-update` builds a message containing the code to update the remote `favg` values as shown in Fig. 2. The strength of our approach is that all the remote values update messages are actually sent as a textual symbols so that no other internal representations and, consequently, translations are needed. All the values are represented in the chosen formalism throughout the computation. As shown in [5] and [6], symbol processing is quite effective in our implementation since what is being sent is actually executable code that is either interpreted or compiled as it is received. A summary of all the words implementing the formalism is provided in Table 1.

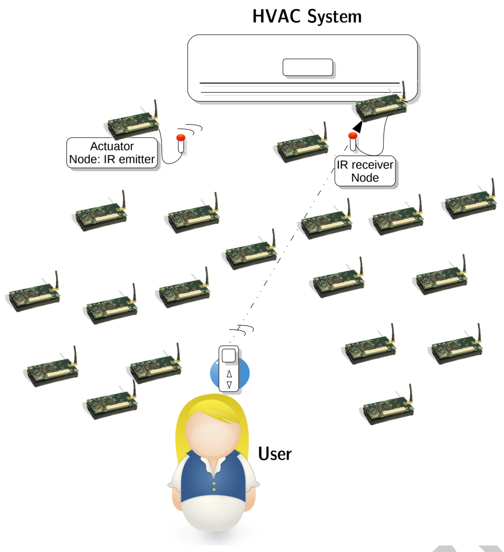


Figure 3: The AmI application scenario. The network is composed of temperature sensing nodes, a node able to detect the IR signals from the remote control to provide the network with *implicit user feedbacks*, and an actuator node equipped with an IR emitter to send commands to the HVAC system.

4 An AmI application to control an HVAC system by estimating the user comfort level

In order to provide a practical application of our methodology to a real problem, we consider an AmI scenario constituted by a sensory infrastructure that acquires temperature readings and reasons in a distributed way to control an HVAC system according to the user preferences (Fig. 3). Each node runs our middleware with the fuzzy extension described in the previous section. Programmers can also send directly to the nodes code to be executed via a command line interface provided by a bridge node connected to a serial terminal on a host PC. All the nodes are IRIS motes. The temperature sensing nodes are equipped with the MTS300CB sensing board, while the IR-receiver node and the IR-emitter node are provided with custom IR devices connected to the expansion lines of an MDA100CA prototyping board. Let us assume that the nodes are already calibrated and share the same fuzzy temperature categorization (low, medium, high) with the user. The fuzzy variables `temp`, `avg-temp`, along with the related fuzzy linguistic terms, are those discussed in section 3. The goal of this section is to prove the feasibility of our approach rather than to describe and evaluate a fuzzy controller in all its details. Thus, in the remainder of the section we limit the discussion to the main tasks of the application.

4.1 Detecting the user's comfort level

In order to represent the user thermal comfort, we introduce a fuzzy variable that we called `user-comfort-level` and three membership functions (`ucl.cold`, `ucl.comfortable` and `ucl.hot`). These membership functions are known in advance and are the same for each node.

Listing 2: Words defined on the IR receiver node

```
: comfort-level-bcst ( -- )
  bcst user-comfort-level
  fvar-remote-update ;

: comfort-level-remote-update
  translate-IR
  comfort-level-bcst
  bcst [tell:] user-agreement-mode
   [:tell] ;
```

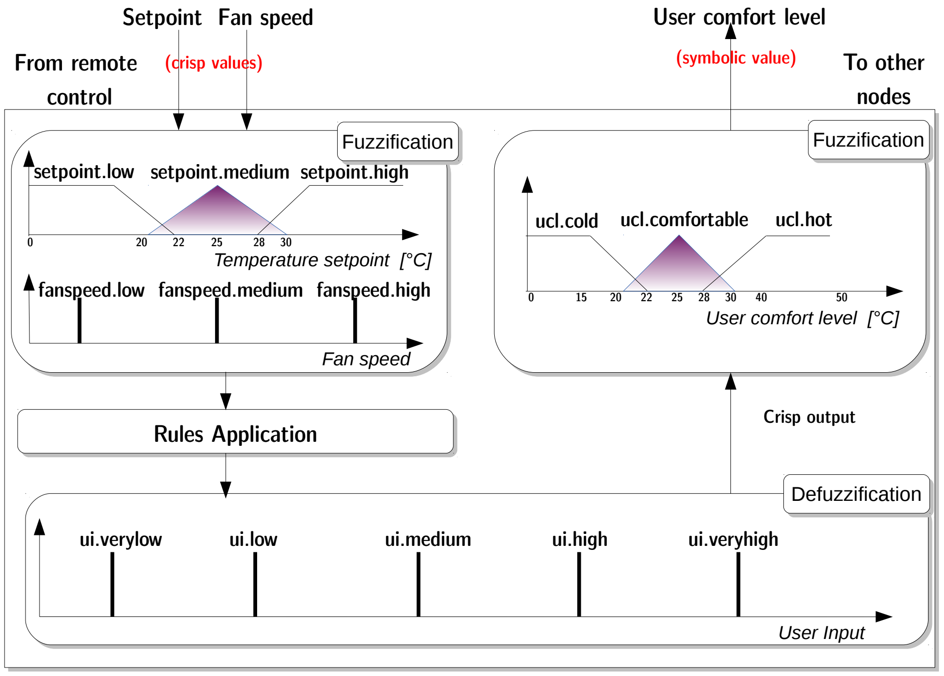


Figure 4: The IR receiver node processes the IR input signal to obtain the numerical values corresponding to the fan speed and the setpoint set by the user. These values are then fuzzified and combined into a representation of user input through fuzzy rules. Then the user-input is fuzzified and the IR receiver node computes a truth value for each membership function of the fuzzy variable `user-comfort-level`. Eventually, the symbolic values are broadcast to the WSN network.

The IR receiver operation is described quite compactly by the code in Listing 2. The word `comfort-level-remote-update` implements the whole process described in Fig. 4. In our experimental setup the control command contains, among other codes related to the HVAC operating mode, a temperature setpoint value and the fan speed value. For the latter, only three values, low, medium and high, are possible. Thus, the word `translate-IR` (line 6) decodes the temperature setpoint and fan speed in the last received IR command. Then fuzzifies these crisp inputs and applies rules to obtain a crisp value representing the user input. Finally, fuzzifies the user input value and stores the obtained fuzzy truth values for `ucl.cold`, `ucl.comfortable`, and `ucl.hot`. The word `comfort-level-bcst` broadcasts the updated value of the user comfort level using the word `fvar-remote-update` with the `bcst` MAC address. After the execution of `translate-IR` and `comfort-level-bcst`, the network is switched into the *user agreement mode* (Line 9), which is described in subsection 4.2. The mechanism through which the IR receiver node updates the *user comfort level* on the rest of the network is shown in Figure 5.

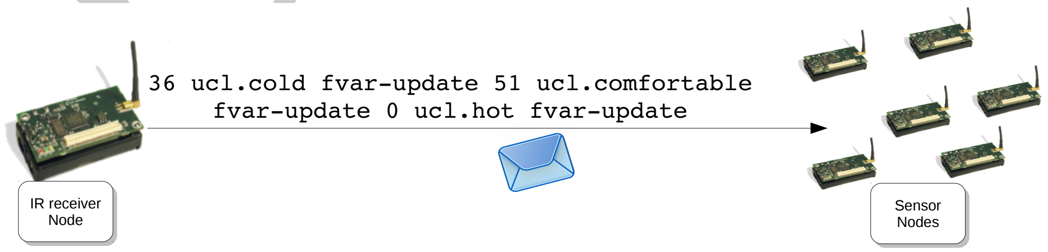

36 ucl.cold fvar-update 51 ucl.comfortable fvar-update 0 ucl.hot fvar-update
IR receiver Node
Sensor Nodes

Figure 5: Remote update of the `fvar user-comfort-level`. The IR receiver node executes `comfort-level-bcst` once it detects an IR signal (Listing 2). This causes the execution of `fvar-remote-update`, which creates a message whose content is the repetition of the pattern: *truth-value linguistic-value* `fvar-update` for each membership function of the fuzzy variable. Once the message is received, the WSN node updates the fuzzy truth values related to the fuzzy values `ucl.cold`, `ucl.comfortable`, and `ucl.hot`.

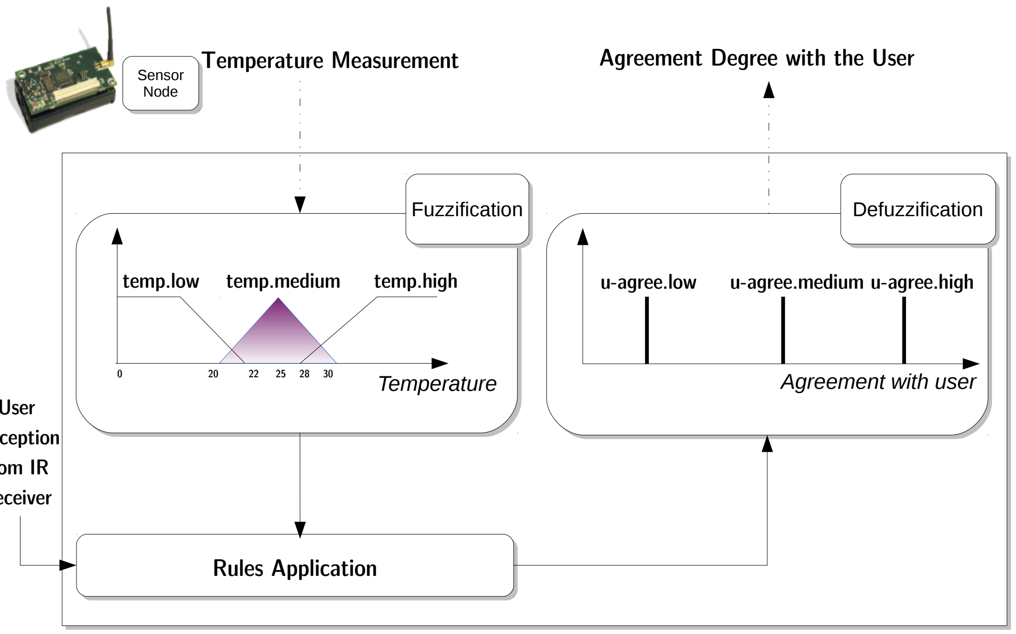


Figure 6: Description of the user-agreement mode. After performing the temperature measurement each node fuzzifies the crisp input and combines it with the user user comfort level estimate to obtain a crisp value representing the agreement degree with user. Fuzzy rules to evaluate the agreement with the user are described in Table 2.

Table 2: Fuzzy rules used to evaluate the agreement with the user.

user-comfort-level / temp	ucl.cold	ucl.comfortable	ucl.hot
temp.low	node-usr-agreement.high	node-usr-agreement.medium	node-usr-agreement.low
temp.medium	node-usr-agreement.medium	node-usr-agreement.high	node-usr-agreement.medium
temp.high	node-usr-agreement.low	node-usr-agreement.medium	node-usr-agreement.high

4.2 Agreement with the user

In order for the application to work properly it is necessary to provide each node with the capability to assess whether it has the same perception of the environmental conditions as the user. After the node has measured the temperature, the reading is fuzzified and processed by fuzzy rules together with the last user input received from the IR receiver and eventually defuzzified to obtain a crisp output representing the agreement with the user. The process is based on the mechanism of implicit feedbacks, as described in [12] and is shown in Fig. 6. After agreement with the user has been evaluated the network switches to the *network agreement mode*, described in subsection 4.3

4.3 Agreement with the network on temperature and comfort level estimation

Analogously to the agreement with the user, each node can assess its agreement degree with the temperature perceived by the network. Each node, before transmitting, locally updates the fuzzy average by adding its fuzzy truth value to the sum it has received, increments the number of nodes, calculates the average, and sends the sum of fuzzy truth values to the other nodes (Listing 3).

Table 3: Fuzzy rules used in the evaluation of the agreement with other nodes

avg-temp / temp	temp.low	temp.medium	temp.hot
temp.low	nodes-agreement.high	nodes-agreement.medium	nodes-agreement.low
temp.medium	nodes-agreement.medium	nodes-agreement.high	nodes-agreement.medium
temp.high	nodes-agreement.low	nodes-agreement.medium	nodes-agreement.high

Listing 3: Code implementing the shared averaging of the fuzzy temperature value

```
1 : average&bcst
2   temp avg-temp favg-local-update
3   bcst avg-temp favg-remote-update
4 ;
5 : start-averaging
6   id @ on-timer ['] average&bcst once fire ;
```

The word `favg-local-update` (line 2) updates the fuzzy average according to the values of `temp`. The word `favg-remote-update` sends the updating message to the other nodes. The word `start-averaging` controls the shared averaging process by making the word `average&bcst` be executed only once, after a time interval proportional to the node's ID value has elapsed. The word `network-agreement-assess` (Listing 4) implements the procedure to compute the agreement degree with the network. The temperature measurement is the crisp input to the fuzzification process. The resulting fuzzy values are combined with the user comfort level sent by the IR receiver and then defuzzified, obtaining a numerical value representing the agreement of the node with the rest of the network, as shown in Fig. 7. The word `network-agreement-rules` executes the rules described in Table 3 and then `conclude` defuzzifies the output fuzzy variable `network-agreement` leaving on the stack the resulting crisp output. The agreement with the other nodes can be used to define new behaviors according to the agreement degree. For instance, if the agreement degree is low, a node can decide to retract its contribution to the average and inform the network.

Listing 4: Code to assess agreement of node with the network

```
: network-agreement-assess ( -- crisp-output)
  temp-read temp apply
  network-agreement-rules
  network-agreement conclude
```

4.4 Actuate on the environment

In order to meet the user's thermal comfort, it is necessary a periodical control of the HVAC system. Since the membership functions representing the user comfort level are known to the nodes in advance, the average temperature classification can be combined with the former to control the HVAC system. Output values are the corrections in fan speed and temperature setpoint to be sent to the HVAC as IR signals. Before sending an IR command, the actuator sends to the IR-receiver node a command to disable IR detection and processing. This ensures that only IR signals coming from the user operated remote are interpreted as user inputs. After the signal is sent the actuator re-enables IR signal reception.

5 Building and testing the application

Our methodology supports and encourages interactive and incremental programming. Following this coding practice, we thus now describe the steps it takes to build and test our experimental AmI application. All nodes receive the same code –with the few exceptions of the specific code for the IR receiver and transmitter nodes– in the initial *code distribution* phase that can be carried on in either batch or interactive way. In the latter case, the programmer sends nodes definitions for:

- the fuzzy variable `temp` and the related membership functions (Listing 1),
- the fuzzy average variable `avg-temp`,
- the action to be taken when a IR signal is detected (Listing 2),
- the shared averaging of the fuzzy temperature value (Listing 3),
- the assessment of the agreement of node with the network (Listing 4).

When all the necessary definitions have been provided to nodes, the fuzzy temperature averaging can be started:

```
bcst tell: start-averaging :tell
```

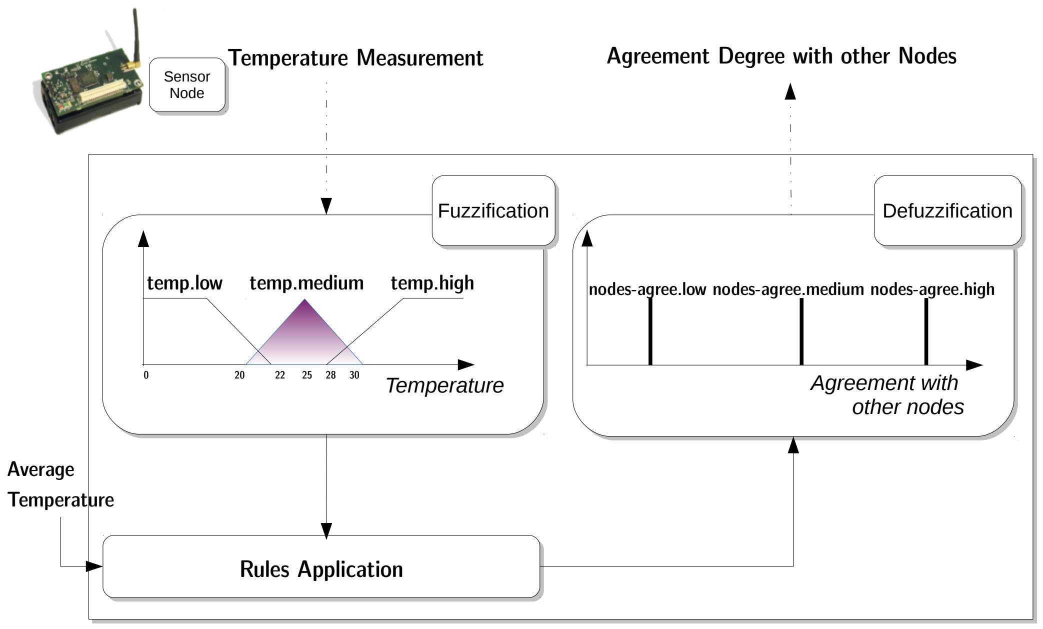


Figure 7: The network agreement task. Each node can evaluate its accordance with the network computed comfort level, that is the average temperature truth value resulting from the aggregation process. Nodes perform the fuzzification-rules-defuzzification loop to obtain a crisp output representing its agreement degree with the rest of the network.

and the handler responsible for the IR signal interpretation can be installed on the IR receiver node:

```
<IRreceiverID> tell: on-IRreception
' comfort-level-remote-update :tell
```

Periodically, the node with the IR emitter might choose one of its neighbor WSN nodes to ask the network estimated comfort level resulting from the averaging process, and use it to actuate on the environment. The control time interval must be adequate to ensure that the nodes have already calculated the degree of agreement with the network at least once. A comprehensive description of the concurrent application execution is provided in Figure 8. The *inner interpreter loop* executes the commands received from either a UART or a radio link. As the execution model is event-driven, other tasks can be running on nodes and paused when the tasks of the application are to be executed. This mechanism is quite effective as it is based on hardware interrupts.

6 Conclusions

In this paper we presented a methodology for interactive high-level programming on WSN following our previous work in which we had drawn the foundations of the methodology based on executable code exchange and interactive programming, and introduced a lightweight middleware platform for WSNs. Even though WSN nodes are resource-constrained, we showed how abstractions and symbolic expression evaluation can be efficiently incorporated into a programming model for such networks by exploiting both interpretation and compilation of code. We then introduced a Fuzzy Logic formalism supporting distributed update of values and evaluation of rules. In spite of the high level symbolic programming allowed, our software platform is quite compact. Including the fuzzy reasoning primitives, it is composed of 454 Forth words, and its overall memory footprint is less than 8 Kb of Flash and 1 Kb of RAM. This leaves plenty of room for complex AmI applications on resource constrained devices. To give a reference example, we discussed the development of an AmI application through the proposed methodology and platform. Future work will focus on extensions of distributed definition and evaluation of symbolic expressions, for instance to make the network learn and reshape the membership functions, or to include other symbolic formalisms.

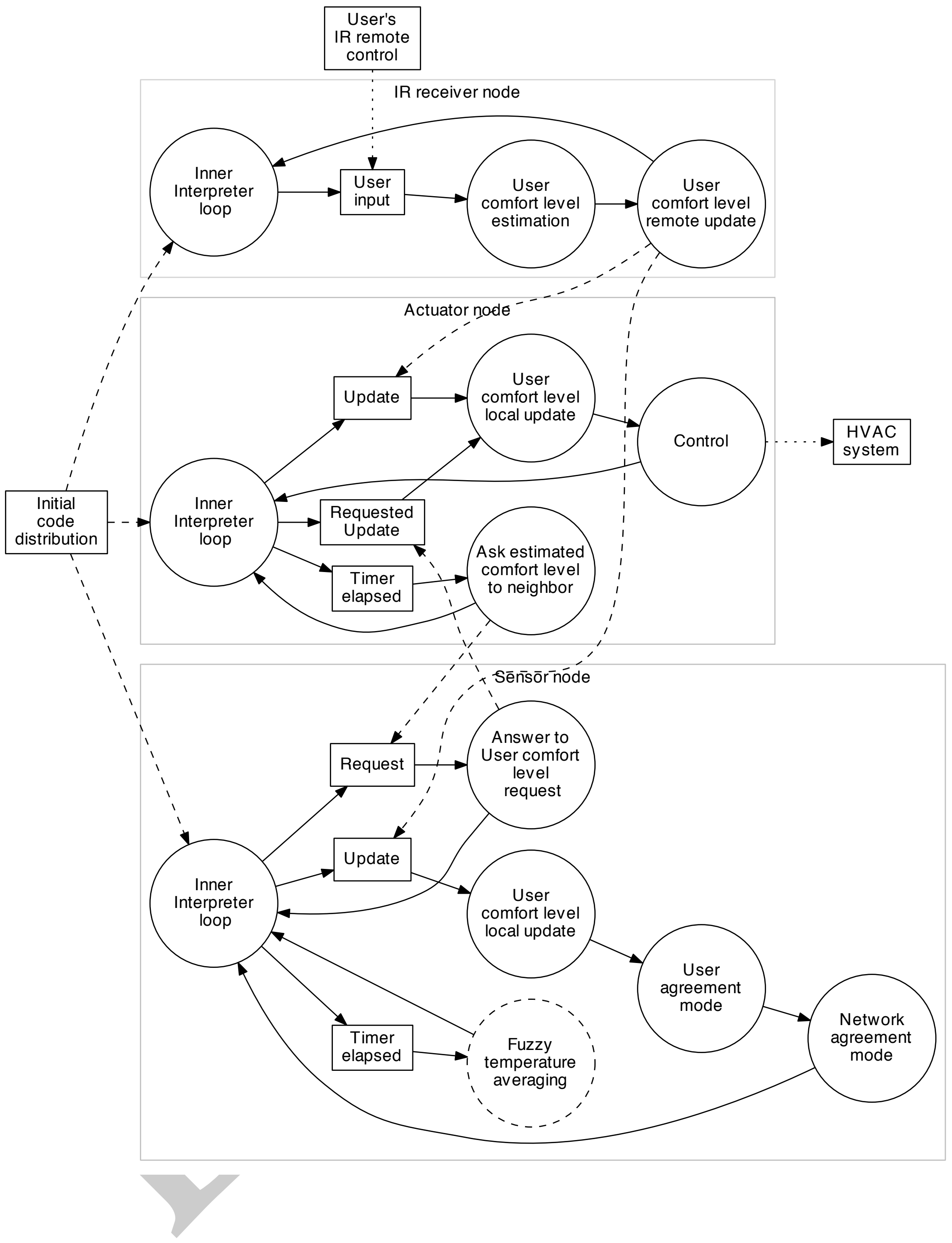


Figure 8: Concurrent execution of the AmI distributed application. Continuous arrows indicate execution of the task composing the application, which are drawn in circles. Signals to trigger task executions –in boxes– are represented by dashed arrows. These signals are actually messages containing executable code that are sent between nodes. The *Fuzzy temperature averaging* task is enclosed in a dashed circle to denote its distributed execution, as described in subsection 4.3 (Listing 3). The dotted arrows from the *IR Remote Control* and to the *HVAC System* represent remote control IR signals.

References

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.

[2] Bin Guo, Daqing Zhang, Zhiwen Yu, Yunji Liang, Zhu Wang, and Xingshe Zhou. From the Internet of Things to Embedded Intelligence. *World Wide Web*, 16(4):399 – 420, 2013.

[3] Gloria Martorella, Daniele Peri, and Elena Toscano. Hardware and Software Platforms for Distributed Computing on Resource Constrained Devices. In *Advances onto the Internet of Things*, volume 260 of *Advances in Intelligent Systems and Computing*, pages 121–133. Springer International Publishing, 2014.

[4] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 86–93, New York, NY, USA, 1998. ACM.

[5] Salvatore Gaglio, Giuseppe Lo Re, Gloria Martorella, and Daniele Peri. A Lightweight Middleware Platform for Distributed Computing on Wireless Sensor Networks. *Procedia Computer Science*, 32(0):908 – 913, 2014. The 5th International Conference on Ambient Systems, Networks and Technologies (ANT-2014), the 4th International Conference on Sustainable Energy Information Technology (SEIT-2014).

[6] S. Gaglio, G. Lo Re, G. Martorella, and D. Peri. A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, pages 1–8, Sept 2014.

[7] Xinrong Zhang and Bo Chang. Research of Temperature and Humidity Monitoring System Based on WSN and Fuzzy Control. In *Electronics and Optoelectronics (ICEOE), 2011 International Conference on*, volume 4, pages V4–300–V4–303, July 2011.

[8] Alessandra De Paola, Giuseppe Lo Re, and Antonio Pellegrino. A Fuzzy Adaptive Controller for an Ambient Intelligence Scenario. In *Advances onto the Internet of Things*, volume 260 of *Advances in Intelligent Systems and Computing*, pages 47–59. Springer International Publishing, 2014.

[9] M.K. Maggs, S.G. O'Keefe, and D.V. Thiel. Consensus Clock Synchronization for Wireless Sensor Networks. *Sensors Journal, IEEE*, 12(6):2269–2277, June 2012.

[10] Gilles Mauris, Eric Benoit, and Laurent Foulloy. Fuzzy Symbolic Sensors-From Concept to Applications . *Measurement*, 12(4):357 – 384, 1994.

[11] R. VanNorman. Fuzzy Forth. *Forth Dimensions*, 18:6–13, March 1997.

[12] A. De Paola, M. Ortolani, G. Lo Re, G. Anastasi, and S.K. Das. Intelligent Management Systems for Energy Efficiency in Buildings: A Survey. *ACM Computing Surveys*, 47(1):38, 2014.