A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks

Article

Accepted version

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

In Proceedings of the 19th International Conference on Emerging Technologies and Factory Automation (ETFA2014)

It is advisable to refer to the publisher's version if you intend to cite from the work.

Publisher: IEEE

NDS Research Group - Networking and Distributed Systems
University of Palermo, ITALY
http://www.dicgim.unipa.it/networks

A Fast and Interactive Approach to Application Development on Wireless Sensor and Actuator Networks

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

DRAFT

Abstract

In Wireless Sensor and Actuator Networks (WSANs) sensor and actuator devices are connected through radio links to perform tasks in many different contexts. Conventionally, applications for WSANs are developed using traditional operating systems which application code is linked with at the end of a cross-compilation process. We propose instead an alternative approach for building applications on WSANs that is based on interactivity and does not require time consuming cross-compilation phases. In our development methodology, it is possible to define procedures and services according to the application target, simultaneously test them and reprogram the nodes interactively when needed, even after network deployment. The main advantage of our approach is flexibility since it lets nodes exchange data and executable code, permits to define new syntactic constructs at runtime, and supports the creation of application-oriented languages.

1 Introduction

Wireless Sensor and Actuator Networks (WSANs) are gaining considerable interest related to the possibility of combining the benefits of sensing and monitoring of sensor devices with the ability to interact with and modify the environment, typical of actuation devices [1]. These distinctive features have been exploited in many applications in medical, industrial, environmental, and agricultural fields, as well as in ambient intelligence scenarios to control large-scale systems such as energy efficient buildings [2].

In these networks, the need of collaboration among the networked devices, which may differ both in terms of architecture and capabilities, gives rise to new challenges concerning the network programming and management [3, 4].

Usually, the application development for embedded systems with constrained resources, such as the devices composing WSANs, conventionally relies on the support of a general-purpose operating system providing basic services to applications. On the other hand, an operating system offers a rigid architecture which limits any modification of the source code meaning a deep knowledge of the operating system architecture and functionalities. The methodology for developing embedded applications using an operating system consists in a chain of phases that transform the source code into machine code by using compilers, assemblers, linkers and debuggers, and requires to upload the executable code on the target node by means of some wired in-system programming tool. This practice makes the programming phase “static” and time consuming since the nodes must be programmed in advance for any specific task and every change in the code implies to compile and re-upload the whole application. Application testing can be thus performed only at a late stage of development, and solving the errors highlighted in this phase implies to correct the code and redo the previous steps involving the generation of the machine code to be loaded on the node and its test.

Moreover, interaction among nodes, whether sensors or actuators, plays a crucial role in the adoption of such networks in several applications. Therefore, it is desirable to provide mechanisms for programming and config-

uring heterogeneous nodes in a homogeneous way, even after their deployment, instead of reprogramming them manually [5, 6].

These difficulties are more evident when programming WSANs, due to the intrinsic differences of the devices composing the network and to the absence of standard high-level abstractions simplifying the programming phases [1].

To overcome these issues, we propose an approach that is based on interactive development of applications on sensor and actuator nodes even after their physical deployment. The key idea is to have an interactive environment that runs on nodes, and that is provided with a high-level programming language and an essential but easy to expand set of functionalities. The simplicity of programming a heterogeneous network is thus achieved by combining homogeneous primitives for sensors and actuators with symbolic processing of data and code permitted by the high-level programming language. Our approach offers the programmer the possibility to define new syntactic rules, supporting the creation of application-oriented languages. Moreover, being interactive, the software development phase is faster than the traditional approaches based on cross-compilation.

In Section 2 we introduce our methodology and describe our experimental setup. In sections 3 and 4 we discuss the application of our development approach to sensing and acting tasks. Section 5 presents a sample case study consisting of a simple remote shell application to demonstrate how our methodology makes the application development for WSANs easy and fast. Finally, Section 6 reports our conclusions.

2 Interactive Methodology for Application Development

Since programming WSANs is far from being trivial, it is important to ease the application development maintaining at the same time a certain degree of flexibility. We propose a methodology that avoids cross-compilation and aims to minimize the time of development by resorting to interactivity. Developing applications in an interactive manner offers the programmer the possibility to design an application and simultaneously test it. Errors and failures can be thus highlighted during code drafting. This tends to reduce the time spent in programming because it avoids compiling and uploading the executable code to the target node before running and testing it.

The need of a language support that hides the hardware complexity and heterogeneity of WSANs, maintaining at the same time a high degree of expressivity proves itself crucial [7]. Moreover, the language has to be a very versatile tool to be fully exploited by the programmer, avoiding to impose rigid constraints.

For this purpose, we decided to adopt a Forth based environment as the basis of our software development methodology [8]. We begin our discussion providing a brief overview about the Forth programming principles.

2.1 Forth: Language and Principles

Forth is not just a programming language. It has particular characteristics that can be seen as principles to be followed when programming in Forth [9]. As a programming methodology, simplicity is the key requirement of the application code since understanding a problem the application target aims to solve, means to decompose it into simpler basic components easy to prototype and describe. Following this criterion, the primary element of the language is the *word*, which is comparable to a subroutine or a function in other programming languages. Unlike other programming languages, calling a subroutine can be as simple as typing its name in a terminal. Word names are in fact stored in a dictionary along with their definition in terms of executable code. The interactive environment evaluates input by looking for words in the dictionary and executing their definition. The dictionary can be seen as a linked list in which each stored word maintains a pointer to the previous defined word. Each definition contains also a pointer to the memory area containing the related code to be executed. New defined words are easily added to the dictionary. Forth is based on the explicit management of two LIFO stacks, the *data stack* and the *return stack*. Forth words use the *data stack* to pass parameters to each other. After a word has been executed, the result of its execution is pushed onto the stack and similarly, before its execution, the required parameters are pulled from the stack. Parameter passing is thus implicit. For this reasons, Forth programmers use comments in form of "stack effect" notation to describe the parameters required and left by a word onto the stack. Using this notation, each word definition includes the values it requires on the stack before being executed and the values left on the stack afterwards, separated by dashes. For example, the built-in word `+` that performs addition is defined as:

```
+ ( n1 n2 -- n1+n2 )
```

The rightmost item on either side of the two dashes is the item on top of the stack. So the word `+` affects the stack by picking two values, first `n2`, then `n1`, and leaving one, the sum `n1+n2`, on the top. The *return stack* is used to

DRAFT

hold return addresses for nested definitions or to store temporary data. Both stacks are directly controlled by the programmer, who has full control of the system.

Another peculiar feature closely related to the Forth behavior as an "expandable compiler" is the absence of a rigid predefined syntax. In many contexts, this lack turns into a powerful feature since it makes it possible to define and add any new syntactic construct considered appropriate. From a purely semantic point of view, the name of a word should be representative of the task it performs and therefore a suitable choice of the name makes the application code to coincide with its high-level description.

From the considerations above, the opportunity given to the programmer to create each time its own language tailored to the specific application purposes, rather than to the development platform, emerges clearly. Furthermore, a Forth-derived application language may act directly above the hardware layer while maintaining resemblance to natural language.

With respect to the development of high level applications, the Forth guiding principle of simplicity is thus equivalent to "component programming", based on the idea that understanding a problem implies to simplify it, by decomposing it. Therefore, the first development phase consists of creating a conceptual model of the system and analyzing the components satisfying the specifications. Eventually, a problem solution will be in the form of a sequence of words. Since words are defined in terms of previously defined words, this methodology adapt itself to a twofold approach consisting of a top-down criterion for the application design and a bottom-up approach for its coding [10]. Such decomposition grants that successive changes to the code affect just the component under scrutiny rather than the entire application.

2.2 The Interactive Environment

We developed our interactive environment using the AmForth interpreter [11] that we ported to the IRIS Mote AVR microcontroller based platform. AmForth is itself written in Forth, with the exception of a few words written in assembly, and allows to explore the AVR microcontrollers in an interactive way via a shell in a serial terminal.

AmForth inherits from Forth the programming model based on the indirect threading and uses an inner interpreter for code processing. This interpreter performs an infinite loop consisting in checking for interrupts, reading the instruction pointer pointing to the next instruction to be executed, executing it and jumping back to the interrupt check (Fig. 1).

The interactive environment is instead provided by the text interpreter. The text interpreter is an endless loop that processes the input from the current input device and looks in the dictionary for each word found in the input stream. Indeed, if the word is found in the dictionary it is executed, otherwise the system tries to convert it to a number.

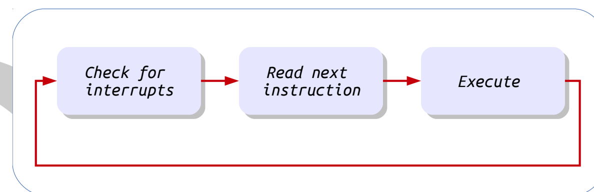

Figure 1: The AmForth interpreter performs an infinite loop consisting in checking for interrupts, reading the instruction pointer for the next instruction to be executed, executing it and jumping back to check for interrupts.

Conventionally, applications for WSANs are developed using traditional operating systems [12] written in languages that need cross-compilation. Any code change involves recompilation before code execution. On the contrary, in our Forth-based environment, subsequent code changes can be made in a simple and interactive manner with considerable time saving. Application development time can be significantly reduced since the device being programmed gives its feedback instantaneously and testing can be performed during code drafting. Our experimental setup consists of sensor nodes deployed in the environment and reachable only wirelessly, and a bridge node connected to the user computer. The user interacts with this node through the shell and can send messages to the other nodes in the network. Messages can transport code to be remotely executed, as shown in Figure 2.

2.3 The Iris Mote Platform

In this section, we provide a brief overview of the Iris Mote [13] platform that we used in our experimental setup. Indeed, our effort in porting AmForth to this platform can be viewed as an example of constructing a minimal extensible software layer to support higher layers by exploiting the interactive approach.

DRAFT

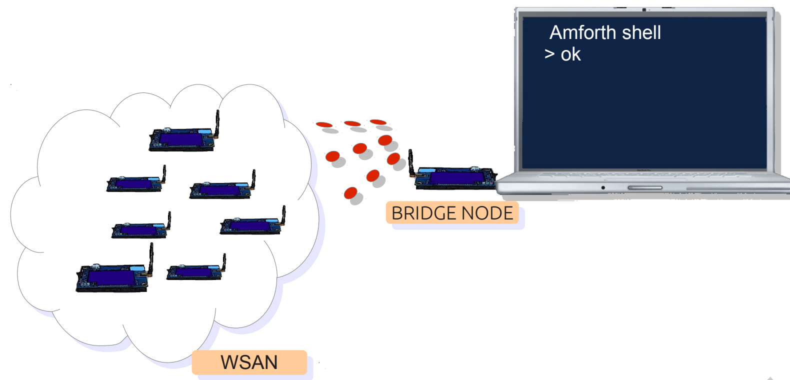

Figure 2: Our experimental setup. The user interacts with the bridge node through the shell and can send messages to the other nodes in the network.

Table 1: Summary table of words used for sensing tasks. Words are indicated together with their "stack effect"

Word (before -- after)	**Description**
temperature (-- temp_value)	Measure temperature and push the numeric value onto the stack
luminosity (-- light_value)	Measure light and push the numeric value onto the stack
mic (-- mic_value)	Measure sound level and push the numeric value onto the stack

As the vast majority of the sensor nodes used in WSANs, the Iris Mote is a device with constrained storage, processing and energy resources. This platform is based on the Atmega1281 Harvard RISC microcontroller running at about 8 MHz and provided with a 128 KB Program Flash memory, a 8 KB RAM, a 4 KB EEPROM, and a 512 KB Measurement Flash. Analog Inputs, Digital I/O, I2C, SPI and UART interfaces are exposed through the Iris Mote 51-pin expansion connector used to connect a wide range of external peripherals, such as the MTS310 sensor board used in our experimental setup.

For wireless communication Iris Mote is equipped with the AT86RF230 transceiver, a low power ZigBee 2.4 GHz radio supporting IEEE 802.15.4, 6LoWPAN, RF4CE and ISM applications. The platform also includes a DS2401 chip containing a 64 bit ROM (storing a 48 bit serial number, an 8 bit CRC, and an 8 bit Family Code) providing a unique identifier for the device that is used as MAC source address.

Porting AmForth to the Iris Mote platform required us to extend the AmForth dictionary, which is a quite large subset of the ANS94 Forth standard, in order to create an essential collection of definitions for the common functionalities usually needed by applications. To achieve this goal, we defined Iris Mote specific words as well as some more generic and platform-independent words. Some specific words are necessary to deal with the on-board radio in order to enable frame reception and transmission, and power down. Other words are not tied to specific hardware implementation and can thus easily work on different platforms. For example, we defined some words to create valid data frames according to the 802.15.4 standard or to interpret incoming messages directly in the inner interpret loop. These words –a very small word set, indeed– were all that it was needed to define the primitives for exchanging executable code among nodes with minimal dependencies on the hardware platform (Section 3).

To deal with different platforms, some words strictly related to the specific underlying hardware have a deferred implementation. These words are executed indirectly through a vector containing an execution code –briefly, *xt*– that points to the word to be actually executed. This technique, called "vectored execution" in Forth parlance, allows to change the behavior of these words at runtime according to the underlying hardware, without affecting other words that depend on them. For instance, we used deferred word to switch the interpreter input and output sources. In fact, incoming radio data interpretation requires to redirect the input from the UART, which is the default input device, to the radio module. Similarly, it can be useful –e.g. to implement the remote shell application described in Section 5– to redirect the output device, which again defaults to the UART, to the radio.

A comprehensive view of all the word we defined to build our environment for application development is provided in Figure 3. Words are grouped in word sets according to the specific purpose they serve and their interdependencies. Three layers of word sets lie above the AmForth pre-defined dictionary. The the two lower layers

DRAFT

hold the words needed for the Iris Mote implementation, as described previously, while the layer on top is discussed in detail in the following sections.

This arrangement is not rigid, with fixed abstracted boundaries between layers, but it rather results from the adoption of a "cross-layer" design in which higher-layer words use previously defined lower-layer words. For instance, words of the hardware-dependent *RF230* word set use words of the hardware-abstracted *FRAME* one, for example to set the destination node address before sending a frame.

Although most of these words directly manage hardware components, the expressivity of their implementation makes the operation of these modules quite understandable and their verification immediate. Moreover, a platform change requires to redefine just the blue word sets and to change the deferred words vectors.

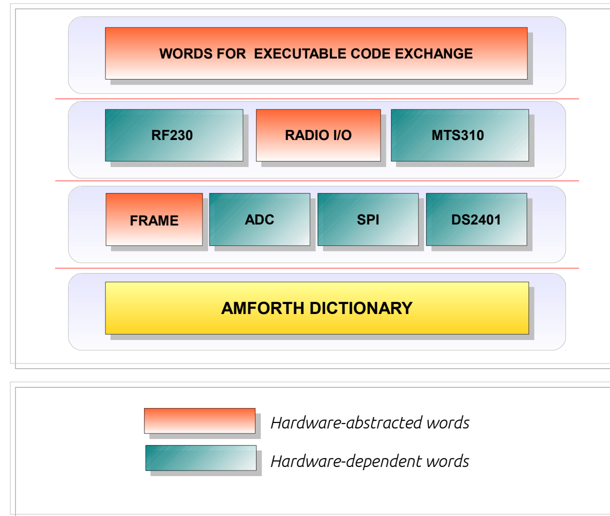

Figure 3: The collection of word sets composing our development environment, including those specific to the Iris Mote platform, defined on top of the AmForth dictionary. The total flash memory footprint of our word sets is less than 5 KB.

3 Sensing the Environment

Sensing the environment is an important functionality of WSANs. A set of sensors permits to acquire and gather significant physical measurements, such as temperature, pressure and so on. Such measurements are useful to sense the realtime environmental conditions and their changes over time. It is also important to ensure that sensor readings are available in real time and, more in general, whenever deemed necessary. To make this process simple and consistent with the methodology of development, we defined the high level Forth word `ask`, which makes a remote node measure a physical quantity using the proper sensor and send the sensory data back to the sender.

The word is used as such:

`ask` ⟨*physical quantity*⟩

Assuming to have a sensor node named `sensor1` in the network, the programmer may query this node about the temperature. This can be done by typing in the terminal connected to the bridge node the following code:

```
sensor1 ask temperature
```

where `temperature` is a word performing a local measurement that leaves on the stack the sensor reading. In detail, `ask` exploits the

⟨*destination node*⟩ `tell:` ⟨*code*⟩ `:tell`

construct, used to send executable code to a node, either sensor or actuator, through simple shell interaction. The construct execution sends a message to the destination node, with the code between `tell:` and `:tell` as payload. Messages are simply MAC frames. This choice is not a limiting factor as the implementation of more complex

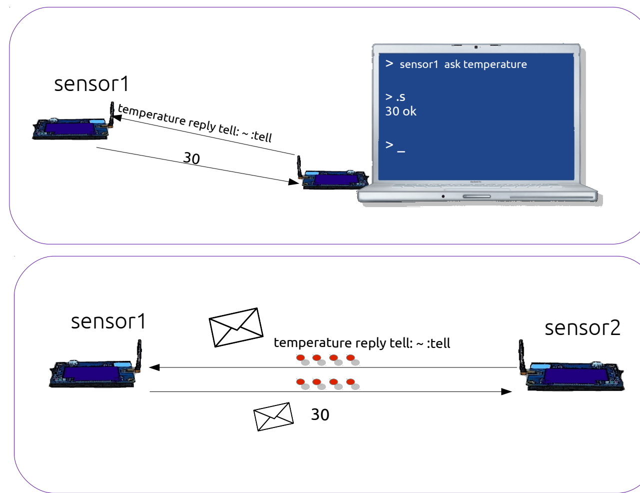

Figure 4: Interaction through the homogeneous primitive `ask`. This word creates a message containing executable code, i.e. a sequence of Forth words that are performed remotely, along with data placed on the stack. In this example, the code to be remotely executed includes the word performing the sensory measurement and the creation of a reply message containing the acquired data. Once the message has been received, `sensor1` measures the temperature. The acquired sensor reading is thus left on top of stack and inserted at runtime in the reply message to `sensor2` through the use of a syntactic placeholder. The word `reply` references the received message sender so that the `tell:` $\langle$*code*$\rangle$ `:tell` construct could answer back.

DRAFT

protocols, such as TCP/IP, is possible by simply defining new word sets exploiting the provided primitives and the executable code exchange capability of the environment.

The code to be remotely executed may contain syntactic placeholders, such as the tilde symbol ($\sim$) for a single *cell* [1] Forth value. A placeholder is substituted with the value taken from the top of the stack at the time of its evaluation. The remote execution constructs can be nested. In fact, once the message is received, `sensor1` measures the temperature and then replies with the sensor reading to the requesting node, in this case the one operated by the user, as shown in Figure 4.

The peculiarity of the remote execution primitives is that they are quite generic and can be used interchangeably by:

- a sensor node to request a measurement to another node, for example to perform distributed computing or data aggregation;
- an actuator to request the measurement to a sensor or sensors in its radio range;
- the user to request a measurement to a node.

Table 1 summarizes the words used to perform local measurements along with their meaning and the data stack effect.

4 Acting on the Environment

The presence of actuators in the network adds novel design opportunities to wireless sensor networks [14]. For instance it provides the ability to make decisions based on the data acquired by the sensors that, in turn, can be used to take appropriate actions that influence and modify the environment. Also in this case it is necessary to provide mechanisms to interact with the actuators that are both simple and expressive, and allow the user to program the actuators so that conditions could be modified in real time when acting on the environment.

Usually WSANs are highly dynamic, thus interactive programming seems to be the most appropriate development methodology.

Assuming to have an actuator node in the network named `actuator1`, the programmer may tell this node to take a specific action if certain current conditions are met. In our model this is simple and quite understandable. If the

[1] Forth value sizes are expressed as multiples of a *cell*, conventionally as large as a host processor data register. Standard Forth arithmetic operators use either single or double *cell* values.

programmer decides to change the action conditions, he can send to `actuator1` another message just by typing in the terminal:

```
actuator1 tell: sensor1 ask luminosity 30 < if light on then :tell
```

In the example, `actuator1` is told to ask `sensor1` about the luminosity value. The sensor reading is thus sent back to `actuator1` which turns on the light if the luminosity value is less than 30. If the programmer decides to change the action conditions, he sends `actuator1` another message by typing in the terminal:

```
actuator1 tell: sensor1 ask luminosity 50 < if light on else light off then :tell
```

or by making `actuator1` query another node:

```
actuator1 tell: sensor2 ask luminosity 50 < if light on else light off then :tell
```

The last two examples show how it is possible to directly interact through the shell with an actuator device just by sending it code for immediate execution. On the other hand, in order to create more complex applications involving acting devices, new words may be easily defined, for example to set a different luminosity thresholds, or to store all the possible actions as new defined words of their dictionaries.

For instance, let us suppose it is required to write the application code for an actuator device that periodically ask the luminosity value to a sensor node so that if a luminosity measurement is under the minimum required luminosity threshold, the light be turned on.

First of all, a variable indicating the threshold is defined. The user can interactively change the actuator's behavior just by sending it a message for changing its threshold value or, more drastically, by redefining the word representing the action under consideration.

At a glance, we need a word to ask the luminosity value, a word to set its threshold, and a word to perform the action triggered by luminosity exceeding the threshold. Listing 1 shows a code sample on the actuator node to achieve this goal.

Listing 1: Code of a simple application acting on the environment according to the luminosity value

```
1  : every-minute ( xt -- )
2   60000 every
3  ;
4
5  variable threshold
6
7  : switch-light ( value -- )
8   threshold @ < if light on else light off then
9  ;
10
11 : light-action ( -- flag )
12  sensor1 ask luminosity
13  switch-light
14  stop?
15 ;
16
17 rdefer action
```

In this code sample, we defined on the remote actuator a minimal word set that is composed of just three words to perform continuous monitoring of the ambient light at regular intervals. As shown in Listing 1, we defined the convenient word `every-minute` using the pre-defined word `every` that requires on the stack the execution code of the word to be executed with the time interval expressed in milliseconds as the second stack value. The word to be executed leaves on the stack a boolean flag to indicate whether or not the repeated execution should be terminated.

The word `switch-light` describes the action of switching the light to change the environmental luminosity under the specified condition.

The action to be performed by the actuator device is defined in the self-explanatory code of the `light-action` word. The actuator node asks `sensor1` about the instantaneous luminosity level and, on the basis of the acquired value, turns the light either on or off. The word `stop?` leaves on the stack the flag required by the `every` word calling conventions according to some termination rule, as reported in the stack effect of `light-action` (line 11). For the sake of completeness, a simple `stop?` definition to the control-loop end after one hour could be:

DRAFT

```
: ++@ ( addr -- value )
    dup @ 1+ tuck swap ! ;
variable elapsed-minutes
: stop? ( -- flag )
    elapsed-minutes ++@ 59 > ;
```

For convenience, we also defined the word `++@` used to increment the minute counter while leaving the updated value on the stack using the standard stack manipulation words `dup`, `tuck` and `swap`.

Since the monitoring action may be performed in different ways according to the application purposes, `action` is implemented as a deferred word and therefore its execution code can be set at runtime (line 17). For instance :

```
actuator1 tell: ' light-action is action :tell
```

sets the execution word of `action` on `actuator1` refer to the word `light-action`.

The application can be made run by typing:

```
actuator1 tell: ' action every-minute :tell
```

Then, if it is required to change the `actuator1`'s luminosity threshold, the user may type:

```
actuator1 tell: 50 threshold ! :tell
```

On the contrary, if it is necessary to change the actuator action, it may be appropriate to remotely define a new word for the desired actuator behavior, named for example `new-action`, and then type:

```
actuator1 tell: ' new-action is action :tell
```

Then `new-action` will be the word to be executed each minute in place of `light-action`.

The code to trigger the light control interactively on an actuator device may be included in a new word definition on the bridge node as indicated in Listing 2.

Listing 2: Word defined on the bridge node to initiate the light control application

```
1 : light-control ( actuator_id -- )
2   [tell:] ' action every-minute [:tell] ;
```

Therefore, once the user types

```
actuator1 light-control
```

the bridge node sends the code for executing `action` every minute to the destination actuator device. It is worth noticing the use of the `[tell:]` ⟨*code*⟩ `[:tell]` construct for sending code to a remote node which is used inside a word definition. Figure 5 depicts the interactions described above.

In the end, due to the adopted approach, it is possible to create new words and syntactic constructs for any application target and to establish adequate action policies –or changing them in real-time– even after the network deployment.

5 Case Study: A Remote Shell Application

Building applications in an interactive way reduces the development time and allows to simultaneously test the application. In this regard, the main development tool is provided by the shell. Everything typed at the terminal is instantly interpreted and executed by the node connected to the shell.

As described in Figure 2, the experimental setup includes deployed sensors and actuators nodes, and a bridge node connected to the user computer. The user interacts directly with the shell of this node as a means to send both data and code to be remotely executed.

In this section, we completely describe a simple application to demonstrate the simplicity of the proposed development methodology. The application is a telnet-like remote shell that allows to transparently interact with the shell of the bridge node as if it were the shell of a remote node, either sensor or actuator.

This application can be useful in different contexts as well as for debugging operations, to inspect the state of a remote node or to display the sensor readings as if the remote node were physically connected to the programmer

DRAFT

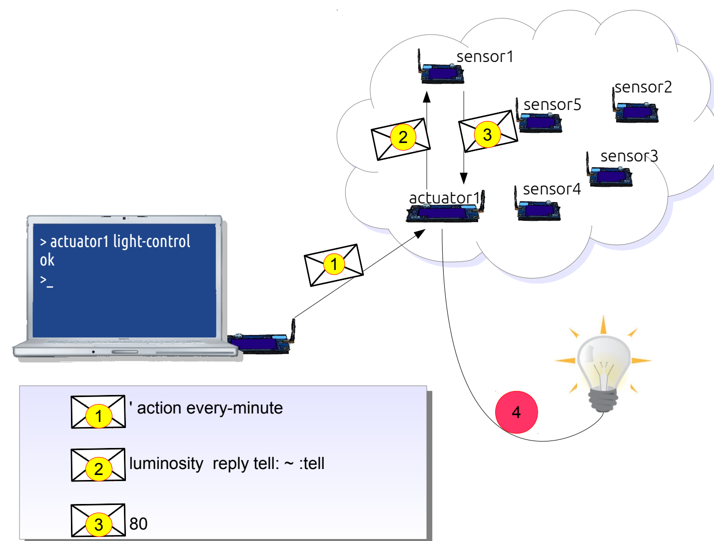

Figure 5: Sequence of interactions among nodes in the light control application. Steps 1-2 involve exchange of messages containing executable code. In step 3 a message containing a single *cell* value is sent. Actual content of each message is shown in the legend. Once the first message arrives, `actuator1` performs `action`, sending a message to `sensor1` asking for the luminosity value. Then, `sensor1` measures luminosity and sends the value back to `actuator1` that executes `switch-light`. Its threshold value is compares with 80 and the light is turned on (step 4). The sequence continues indefinitely as execution is restarted every minute from step 2.

console. During code drafting, the programmer can interactively test its operation and verify the application is working properly.

What we discuss in this section is a coding example with the main purpose to show the readability and expressiveness of the code and the simplicity of development. Nevertheless the code is fully functional, and has been extensively used in our experimentations. Indeed, with hindsight, the remote shell application can be considered as a completeness and consistency test for our approach. In fact, to implement this essential application, we defined incrementally the words we needed, testing them during the development phase. Summarizing, we defined:

- words to redirect the output to the outgoing message;
- words for substituting code at runtime;
- syntactic structures to send data and code in a symbolic way;
- words to redirect the input to the radio and inject the ingoing message into the inner interpret input buffer.

The almost complete remote shell application code is shown in Listing 3. A few additional words are omitted and their description can be found in Table 2.

Listing 3: Code for a simple remote shell application

```
1  80 constant cmd-maxlen
2  variable cmd cmd-maxlen cells allot
3  variable cmd-len
4  variable node_id
5  variable timeout
6
7  : input-send ( -- )
8      cmd cmd-len @
9      node_id @ [tell:] ∼s [:tell] ;
10
11 : rshell-task (  -- )
12     payld-reset
13     input-send
14     timeout @ wait-answer if
```

DRAFT

```
15      payld-print then ;
16
17  : user-input ( -- )
18      cmd cmd-maxlen accept ( -- len )
19      cmd-len ! ;
20
21  : close ( -- )
22      node_id @ [tell:] -radio-output
23      [:tell] quit ;
24
25  : rshell-loop ( -- )
26      begin
27       cr ." rsh>" user-input
28       close?
29       if close
30       else rshell-task
31       then
32      again ;
33
34  : on-timeout ( -- )
35      ." Connection timeout." cr ;
36
37  : welcome-msg
38      ." Welcome to the remote shell
39   application!" cr
40      ." Enter 'close' to close the
41   application" cr ;
42
43  : rshell ( id -- )
44      welcome-msg
45      2000 timeout  !
46      dup node_id !
47      [tell:] +radio-output [:tell]
48      rshell-loop  ;
```

Table 2: Summary table of additional words used in the remote shell application

Word (before -- after)	**Description**
+radio-output (--)	Redirect the output to the radio. This word is part of the Radio I/O word set
-radio-output (--)	Redirect the output to the UART. This word is part of the Radio I/O word set
radio-input? (timeout -- flag)	Check for incoming radio messages. If no message arrives before `timeout` milliseconds leave false on the stack, otherwise leave true
payld-reset (--)	Set to 0 the incoming payload length and its current pointer
payld-print (--)	Display the incoming frame content (i.e. the payload)
wait-answer (timeout -- flag)	Wait for incoming radio frame for a predefined period of time specified by the `timeout` variable. If the timeout expires without receiving any answer message, an exception handled by `on-timeout` occurs
user-input (--)	Wait for user input and store its content in the `cmd` buffer and its length in the `cmd-len` variable for further processing by `close?` and `input-send`

DRAFT

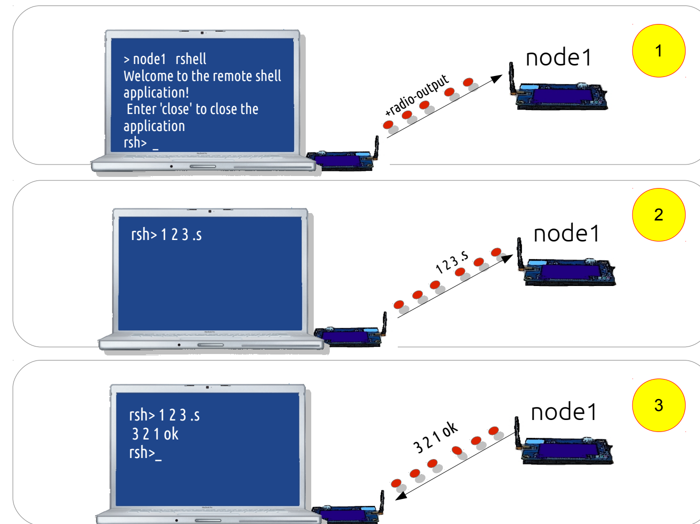

Figure 6: Key steps of the remote shell application execution. First, the welcome message is displayed together with the remote shell prompt. A message including the code to redirect the remote node output to the outgoing messages is sent to the remote destination node (1). The user pushes some numbers to the remote data stack and commands to display the stack content (2). The remote stack is displayed in the terminal (3). The interaction takes place as the remote node were physically connected with the user computer.

The core word of this telnet-like application is `rshell-loop`. It consists in an endless loop that displays the remote shell prompt (line 27) and waits for the input code from the user. If the user types `close` the application quits, otherwise the `rshell-task` word is performed. This word sends the user input to the remote node (line 13). The bridge node waits for input from the radio, i.e. incoming frames from the destination node (line 14), for at most the amount of time specified by the `timeout` variable, which defaults to 2000 ms (line 45). This is implemented as a loop with timeout. If the timeout expires without receiving an answer from the remote node, a timeout exception handled by `on-timeout` occurs, otherwise the bridge node displays the incoming frame content, i.e. its payload (line 15).

To start a remote shell session with the node named *node1*, using a telnet style convention the user types:

```
node1 rshell
```

and an initial welcome message is displayed (line 44). The word `rshell` stores *node1*'s address in the variable node_id (line 46) and sends a message to the remote node to switch its output device to the outgoing message (line 47). After that, it performs the remote shell main loop (line 48).

As a result, the user interacts with the remote node as if the remote node shell were physically connected to the user computer. Table 2 summarizes the additional words used in the the remote shell application code.

6 Conclusions

The absence of adequate abstractions makes application development on WSANs inflexible and time consuming. Conventionally, applications for WSANs are in fact developed using traditional operating systems which application code is linked with at the end of a cross-compilation process. With the aim to obviate these issues, we proposed an approach to application development based on a high-level interactive programming environment that runs on nodes and provides an essential but easy to expand set of functionalities. Using our methodology, which avoids cross-compilation phases, the programmer can count on instantaneous feedback from the system, even after its physical deployment, that enables the simultaneous verification of the application if code has been modified. The simplicity of programming a heterogeneous network is achieved by combining homogeneous primitives for sensors and actuators with symbolic processing of data and code permitted by the high-level programming language, and by enabling nodes to send executable code to other nodes. The high-level nature of our environment does not force to use a predefined and rigid syntax, but instead encourages the design of application-oriented languages through the definition of new syntactic rules. We showed how simply words can be defined that sense and act on the environment accordingly. We also described the development of a remote shell application for interacting with and

DRAFT

programming remote nodes. Future work will focus on further use of this approach in order to create complete and autonomous application working on WSANs.

References

[1] Shuqin Zhang, Zhiyong Dong, Yan Cui, and Yuejun Dong. A Middleware Platform for WSAN based Application Systems. In *Computer Sciences and Convergence Information Technology (ICCIT), 2010 5th International Conference on*, pages 981–986. IEEE, Nov 2010.

[2] A. De Paola, G. Lo Re, M. Morana, and M. Ortolani. An Intelligent System for Energy Efficiency in a Complex of Buildings. In *Sustainable Internet and ICT for Sustainability (SustainIT), 2012*, pages 1–5, Oct 2012.

[3] E. Caete, J. Chen, M. Diaz, L. Llopis, and B. Rubio. A Service-Oriented Middleware for Wireless Sensor and Actor Networks. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 575–580. IEEE, April 2009.

[4] Ian F. Akyildiz and Ismail H. Kasimoglu. Wireless Sensor and Actor Networks: Research Challenges. *Ad Hoc Networks (Elsevier)*, 2(4):351–367, 2004.

[5] Tommaso Melodia, Dario Pompili, Vehbi C. Gungor, and Ian F. Akyildiz. A Distributed Coordination Framework for Wireless Sensor and Actor Networks. In *Proceedings of the 6th ACM International Symposium on Mobile Ad Hoc Networking and Computing*, MobiHoc '05, pages 99–110, New York, NY, USA, 2005. ACM.

[6] Qiang Wang, Yaoyao Zhu, and Liang Cheng. Reprogramming Wireless Sensor Networks: Challenges and Approaches. *Network, IEEE*, 20(3):48–55, May 2006.

[7] Nelson Matthys, Sam Michiels, Wouter Joosen, Christophe Scholliers, Coen De Roover, Wouter Amerijckx, and Theo D'Hondt. Language and Middleware Support for Dynamism in Wireless Sensor and Actuator Network Applications. In *Proceedings of the 6th International Workshop on Middleware Tools, Services and Run-time Support for Networked Embedded Systems*, MidSens '11, pages 2:1–2:6, New York, NY, USA, 2011. ACM.

[8] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. History of programming languages—II. chapter The Evolution of Forth, pages 625–670. ACM, New York, NY, USA, 1996.

[9] Philip Koopman. A brief Introduction to Forth. In *Second History of Programming Languages Conference (HOPL-II), Boston MA*, 1993.

[10] BrianH. Watts. FORTH, a Software Solution to Real-time Computing Problems. *Behavior Research Methods, Instruments, & Computers*, 18(2):228–235, 1986.

[11] Amforth documentation, 2013. Available online at `http://amforth.sourceforge.net/amforth.pdf`.

[12] MuhammadOmer Farooq, Sadia Aziz, and AbdulBasit Dogar. State of the Art in Wireless Sensor Networks Operating Systems: A Survey. In Tai-hoon Kim, Young-hoon Lee, Byeong-Ho Kang, and Dominik lzak, editors, *Future Generation Information Technology*, volume 6485 of *Lecture Notes in Computer Science*, pages 616–631. Springer Berlin Heidelberg, 2010.

[13] Iris datasheet, 2013. Available online at `http://www.memsic.com/userfiles/files/Datasheets/WSN/IRIS_Datasheet.pdf`.

[14] Suet-Fei Li. Wireless Sensor Actuator Network for Light Monitoring and Control Application. In *Consumer Communications and Networking Conference, 2006. CCNC 2006. 3rd IEEE*, volume 2, pages 974–978. IEEE, 2006.

DRAFT