A Lightweight Middleware Platform for Distributed Computing on Wireless Sensor Networks

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

In Proceedings of the 2d International Workshop on Body Area Sensor Networks (BASNet-2014)

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

A Lightweight Middleware Platform for Distributed Computing on Wireless Sensor 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

Abstract

The peculiar features of Wireless Sensor Networks (WSNs) suggest to exploit the distributed computing paradigm to perform complex tasks in a collaborative manner, in order to overcome the constraints related to sensor nodes limited capabilities. In this context, we describe a lightweight middleware platform to support the development of distributed applications on WSNs. The platform provides just a minimal general-purpose software layer, while the application components, including communication and processing algorithms, as well as the exchanged data, are described symbolically, with neither preformed syntax nor strict distinction between data and code. Our approach allows for interactive development of applications on each node, and requires no cross-compilation, a common practice that makes the development of WSN applications rigid and time-consuming. This way, tasks and behavior of each node can be modified at runtime, even after the network deployment, by sending the node executable code.

1 Introduction

Wireless Sensor Networks (WSNs) are composed of tiny embedded devices connected in a network that are able to collect some useful measurements from the physical environment. The existing works in literature confirm the enormous attention to this technology, which lends itself to be useful in various application scenarios [1, 2]. A distributed application running on a WSN consists of a set of cooperating, interacting nodes. Each node performs small computations and exchanges information with the others, contributing to the accomplishment of the application's goals [3, 4]. Due to the absence of appropriate high-level abstractions to simplify programming WSNs, application development is still challenging. The middleware for sensor networks is responsible of supporting programmers during the application development phases, by reducing the complexity related to the underlying hardware level knowledge and providing adequate system abstractions [5]. In this work we introduce a middleware to facilitate the development of distributed applications for WSNs by combining the practical simplicity of programming with a high degree of versatility. We only provide a minimal software layer allowing for the construction of all the application requirements –e.g. in terms of communication protocols– above it. Moreover, using a symbolic description, code and data are also treated equally, and the application code results readable and understandable. In this preliminary work, we introduce a novel middleware approach in which the nodes are able to directly exchange executable code. Indeed, this mechanism is suitable for implementing adaptive real-time behavior of remote nodes, since the possibility of exchanging executable code does not force nodes to have a default set of predefined abilities. The network may instead acquire new capabilities at runtime. A relatively similar strategy was described for Active Networks programming [6], but it is unsuitable for resource constrained devices as it requires a thick separation layer between hardware and applications. In Section 2 we describe in detail our current work, together with the development methodology and the primitives for code exchange and execution. Section 3 discusses a sample distributed application relying on our middleware, showing how, through the exchange of executable code, a synchronization mechanism between the nodes can be triggered and accomplished. Finally, Section 4 reports our conclusions.

2 The Adopted Approach

In this section, we describe in detail the key requirements guiding the implementation of our middleware layer and the solutions adopted to satisfy them. The middleware platform we are developing shows characteristics that, despite its simplicity, make it different from other middleware implementations existing in literature [7]. The keystone of our approach consists in the use of Forth as a software development methodology [8]. Forth is a language that combines the advantages of several languages with pure assembly [9]. By using Forth, it is not necessary to define a rigid layered architecture, but a cross-layer design approach is simply realized without any strict separation between high and low levels, and by creating just a minimal software layer between the underlying hardware and the application level. Forth is a highly scalable language because adding new words to the language is simple, which makes it easy to extend the middleware platform. The Forth approach allows maintaining a high degree of expressivity, since no syntax elements other than a sequence of word names are required to describe a computation. Therefore, a Forth program can be read similarly to a natural language description of the task it performs. Forth can be thus considered a "meta-application language", lending itself well to the creation of problem-oriented languages.

Conventionally, applications for WSNs are based on traditional operating systems [10] which are usually written in compiled languages and thus any code change involves recompilation before code execution. Using a Forth environment, on the contrary, subsequent code changes can be made in a simple and interactive manner, with considerable time saving. Consequently, the programmer may spend less time in the application development because the feedback is instantaneous and the testing phase takes place simultaneously with code drafting.

The environment we adopted for our middleware is the AmForth interpreter [11]. AmForth is mostly written in Forth, with only a few words written in assembly, and permits to program AVR microcontroller based devices in an interactive way through a serial terminal. AmForth inherits from Forth the programming model based on indirect threading and uses an inner interpreter for processing the code. This 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. The interactive environment is instead provided by the text interpreter, a line based command interpreter. Each word composing a string typed at the terminal is stored in a system buffer where each word is processed. If the word is found in the dictionary, it is executed, otherwise the system tries to convert it to a number. To implement our middleware we ported AmForth to the IrisMote platform, defining words to control the onboard radio and sensors.

2.1 Primitives for Code Exchange and Execution

In a distributed scenario, an adequate communication paradigm reveals itself of fundamental importance, since it represents the mechanism regulating the interaction among the individual components of the network. Our system is based on the interrupt events, generated by hardware or software events. Interrupts are managed by the inner interpreter that is responsible of the interrupt handling at a lower level and, then, of switching to the Forth word dealing with the interrupt. This mechanism ensures that all the interrupts are handled immediately when an event occurs, before resuming the execution of the word previously running. Based on this operating principle, the middleware provides the abstraction mechanisms to handle transmission of code among nodes, and the possibility to inject code through the shell. We implemented this abstraction mechanism in Forth, by defining the pair of primitives `tell:`*`code`*`:tell` and `received`, as shown in Figure 1.

- `tell:`*`code`*`:tell` literally "tell a node to do something". This word creates a packet with the specified destination address and containing the executable code as payload. In detail, the word `tell:` parses what follows until the last `:tell` marker is encountered, and uses `send` to send a frame via radio. Due to the symbolic nature of the language, data and code are treated equally. It is worth noticing that this primitive allows sending the code to the nodes listening to the channel, by using the word `bcst` representing the broadcast MAC address. Therefore, more than two entities can be involved simultaneously in the communication process. Furthermore, this primitive lends itself well to a "recursive" use, e.g. "*to tell a node to tell another node to do something*". Correct transmission is signaled by an interrupt. Dynamically computed values may be inserted in the packet by using syntactic placeholders, such as $\sim$ for the single cell Forth value, that are substituted at runtime with the content of the top of the stack using a hexadecimal representation. Inside a word definition, the couple `[tell:]`*`code`*`[:tell]` must be used. This slightly complicates the syntax with respect to a single word couple having both interpreted and compiled behaviors, but avoids insidious bugs that may arise with Forth state-smart word implementations, traditionally adopted in such cases [12].

- `received` Once an interrupt indicating the frame reception is generated, the frame content is injected into the system input buffer where it is processed by `interpret`.

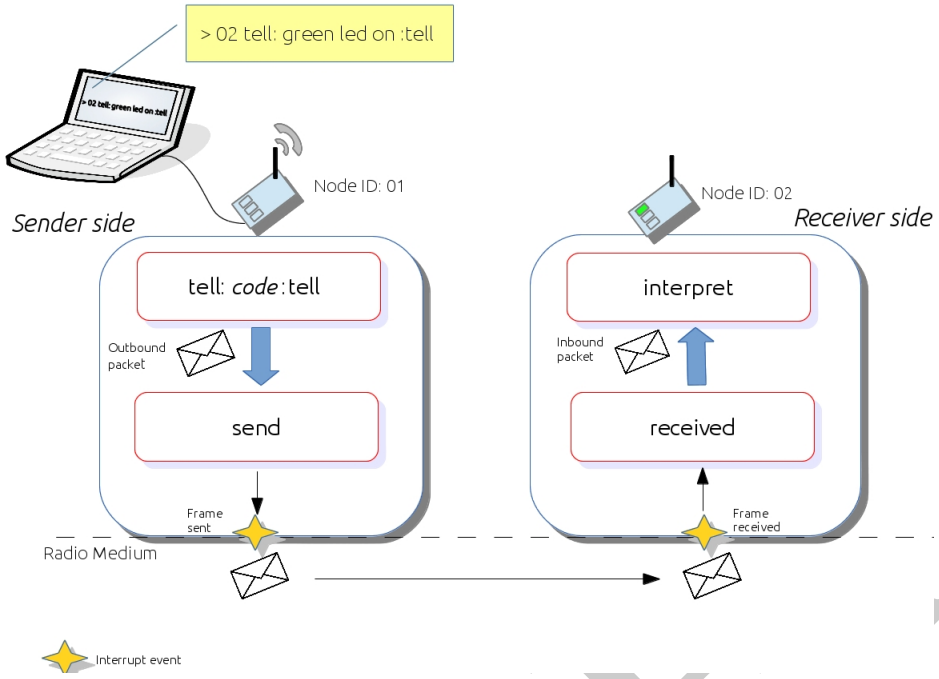

Figure 1: To tell the node with ID 02 to turn its green LED on the sender uses the syntactic construct `tell:code:tell`. This creates a default packet the destination address and the code to be executed are then copied to. Internally, this primitive executes `send` to transmit the packet over the radio channel. Correct transmission is then indicated by an interrupt. On the receiver side, packet detection is signaled by an interrupt and handled by the primitive `received` that injects the packet content into the system input buffer where it is processed by `interpret`.

Figure 2 shows an example of a code and data exchange between two nodes using these primitives. In this example the node with ID 01 tells the node with ID 02 to perform the sum between 2 and 3, and then to reply with the value on top of the stack. Even though the reply message consists only in a literal value, it is interpretable Forth code and it is simply executed by node 01 leaving 5 on top of its stack.

3 Case Study: Time Synchronization

In order to explain how the code migration mechanism can expand the middleware functionalities, we describe a time synchronization protocol implementation. This is a basic service for a middleware as several applications running on sensor nodes require node clocks being synchronized to function properly. This is the case of link layer protocols [13] as well as application layer ones [14, 15]. Even when nodes are turned on at the same time, their clocks would drift differently for several reasons, such as local temperature changes and tolerances. Hence, proper synchronization is required to eliminate the inconsistencies existing among the clocks. In our case study, we adopt a synchronization protocol requiring a hierarchical topology, known as Timing-sync Protocol for Sensor Networks (TPSN) [16]. This sample application relying on our middleware platform shows how, through the exchange of executable code, a synchronization process can be triggered and accomplished. According to the TPSN protocol, the synchronization process takes place in two phases. The first phase creates a hierarchical structure in the network, by assigning a level to each node. Just one node, the root, has level 0, while its neighbors take level 1. Neighbors of level 1 nodes take level 2, and so on. Here we show our implementation of the second phase that performs synchronization. The root node sends a `time_sync` packet to all of its neighbors. Each neighbor waits a random time and sends a `synchronization_pulse` packet to the root containing its level and its local time of the packet reception. The root replies with an `acknowledgement` packet containing its level, the time received from the neighbor, its local time of arrival and its time of sending. The node thus estimates the drift and the propagation delay and it adjusts its clock according to root's clock. This process propagates through the network from nodes of level 1 to nodes of level 2, and so forth. For simplicity, we assume that all the nodes belong to level 1. To implement this protocol, we have to extend the nodes' set of capabilities. In fact, it is necessary to tell the node connected to the shell to send its neighbors pieces of useful code to be stored as new word definitions in their dictionaries. In our case, we interactively send the

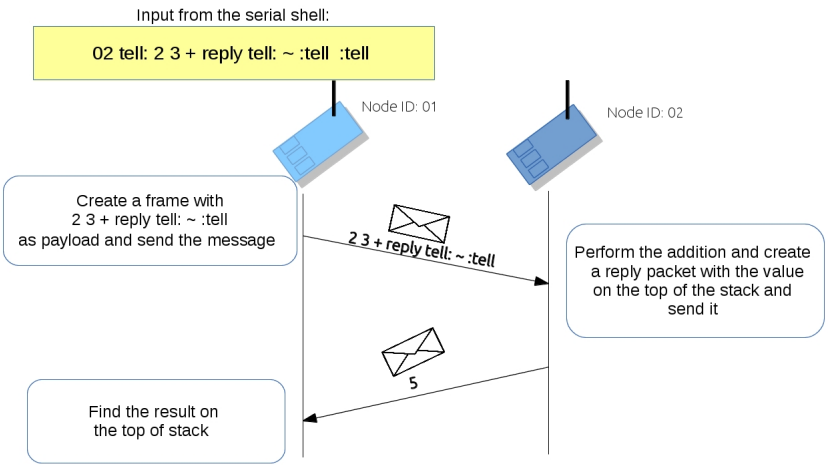

Figure 2: An example of code and data exchange between two nodes. The node with ID 01 tells the node with ID 02 to perform the sum between 2 and 3, and to reply with the value on top of the stack. Node 02 receives the code and performs the sum. The result is now on the top of Node 02 stack. Then Node 02 executes `tell: ~ :tell`, replying with a packet containing the value taken from the top of the stack. Node 01 receives the reply message and executes it. Eventually Node 01 will have 5 on top of its stack.

description of the `time_sync`, `sync_pulse` and `acknowledgement` packets as defined words, together with the code to adjust the clock. Then, we trigger a synchronization process, by sending the `time_sync` packet, as shown in Figure3

Listing 1: Code to define the synchronization primitives according to the TPSN protocol.

```
bcst tell: : clk_sync rtime @ 2diff 2dup pdly rot rot + drift + tick ! ; :tell
bcst tell: : ack tick @ rtime @ lev @ reply [tell:] ~ ~ ~ clk_sync [:tell] ; :tell
bcst tell: : sync_pulse rtime @ lev @ reply [tell:] ~ ~ ack [:tell] ; :tell
bcst tell: : time_sync 60000 random ms sync_pulse ; :tell
```

In order to explain better the synchronization code, we describe the meaning of some useful words and variables. Each word is listed together with the values it needs to have on the stack before being executed and the values left on the stack afterwards, separated by dashes. The rightmost item on either side is the item on top of the stack.

- drift ($T_2 - T_1$ $T_4 - T_3$ -- drift): $drift = \frac{(T_2-T_1)-(T_4-T_3)}{2}$;
- pdly ($T_2 - T_1$ $T_4 - T_3$ -- propagation delay): $propagation\ delay = \frac{(T_2-T_1)+(T_4-T_3)}{2}$;
- random (u_1 -- u_2): Produce a random value between 0 and u1 onto the stack;
- reply (-- addr): Leave on the stack the sender's address of the last received packet;
- 2diff (x_1 x_2 x_3 x_4 -- $x_2 - x_1$ $x_4 - x_3$): Leave on the stack the difference between the first two values on the top of stack and the difference of the last two values;
- ms (u_1 --): Wait for the time indicated as top of the stack;
- rtime: Variable representing the time of arrival of a packet;
- lev: Variable representing the node's level in the topological hierarchy;
- tick: Variable representing the node's local time.

It is interesting to note that, due to the adopted methodology and the tools used, there is a full correspondence between the code and the high-level description of the protocol, as shown in Figure 3. The symbolic description allows the code to be self-explanatory, although it lies just above the hardware layer.

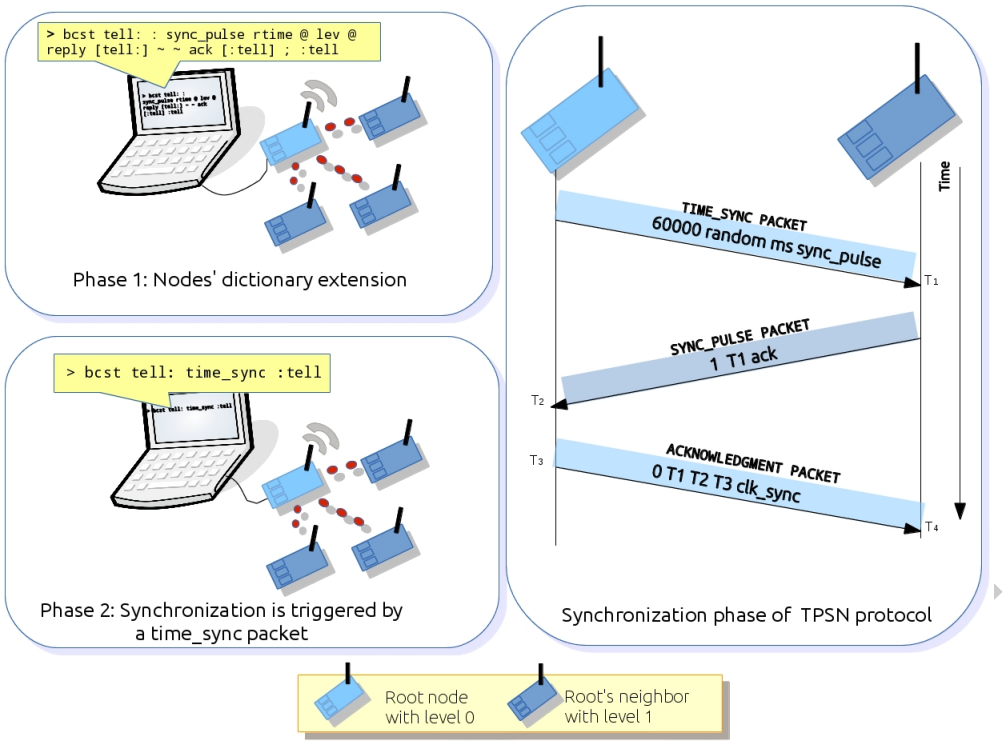

Figure 3: To make the nodes synchronized, we extend the nodes' dictionary by telling the node connected to the shell to send its neighbors new pieces of code to be stored in their dictionaries. Then, the synchronization phase is triggered by telling the node connected to the shell to send its neighbors a *time_sync packet*. T_1, T_2, T_3 and T_4. T_2, T_3 represent the time measured by the local clock of root, while T_1, T_4 represent the time measured by the local clock of the other node. The *sync_pulse packet* contains the node's level and the time of arrival T_1 while the *acknowledgement packet* contains the root's level and the time T_1, T_2 and T_3. Once the *acknowledgement_packet* has been received, the node has the same notion of time as root.

4 Conclusions

In this preliminary work, we presented a lightweight middleware platform to support the development of distributed applications on WSNs. Our middleware allows for interactive and incremental development of applications, even on deployed nodes that are only reachable through wireless links, and requires no cross-compilation, with considerable time saving. Indeed, our middleware provides a mechanism for exchanging directly data among nodes, using just a few simple primitives. As a result, tasks and behaviors of each node can be modified at runtime, without any predefined and fixed set of capabilities, and thus, increasing the adaptivity of the entire network. Typical WSN middleware services, such as authentication or data queries, could also be built exploiting this set of primitives. Future work will focus on extension and refinement of the middleware to implement basic and advanced services in order to support a broad range of distributed applications.

References

[1] Priyanka Rawat, KamalDeep Singh, Hakima Chaouchi, and JeanMarie Bonnin. Wireless sensor networks: a survey on recent developments and potential synergies. *The Journal of Supercomputing*, pages 1–48, 2013.

[2] GeorgeW. Irwin, Jeremy Colandairaj, and WilliamG. Scanlon. An Overview of Wireless Networks in Control and Monitoring. In De-Shuang Huang, Kang Li, and GeorgeWilliam Irwin, editors, *Computational Intelligence*, volume 4114 of *Lecture Notes in Computer Science*, pages 1061–1072. Springer Berlin Heidelberg, 2006.

[3] Lizhi Yang, Chuan Feng, Jerzy W. Rozenblit, and Haiyan Qiao. Adaptive tracking in distributed wireless sensor networks. In *Engineering of Computer Based Systems, 2006. ECBS 2006. 13th Annual IEEE International Symposium and Workshop on*, pages 9 –111, 2006.

[4] Chih-fan Hsin and Mingyan Liu. A Distributed Monitoring Mechanism for Wireless Sensor Networks. In *Proceedings of the 1st ACM Workshop on Wireless Security*, WiSE '02, pages 57–66, New York, NY, USA, 2002. ACM.

[5] FlviaCoimbra Delicato, PauloF. Pires, Luci Pirmez, and LuizFernandoRust Costa Carmo. A Flexible Middleware System for Wireless Sensor Networks. In Markus Endler and Douglas Schmidt, editors, *Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 474–492. Springer Berlin Heidelberg, 2003.

[6] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. Network Programming Using PLAN. In *In Workshop on Internet Programming Languages*, 1998.

[7] F. Moya, D. Villa, F. J. Villanueva, J. Barba, F. Rincn, and J. C. Lpez. Embedding Standard Distributed Object-oriented Middlewares in Wireless Sensor Networks. *Wireless Communications and Mobile Computing*, 9(3):335–345, 2009.

[8] Roger Dettmer. Go fast, go FORTH. *IEE Review*, 34(11):423–426, 1988.

[9] Douglas Lea. Using the FORTH language in real-time computer applications. *Behavior Research Methods & Instrumentation*, 14(1):29–31, 1982.

[10] 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.

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

[12] M Anton Ertl. State-smartness— Why it is Evil and How to Exorcise it. *EuroForth98*, 1998.

[13] L. Gatani, G. Lo Re, and M. Ortolani. Robust and Efficient Data Gathering for Wireless Sensor Networks. In *System Sciences, 2006. HICSS '06. Proceedings of the 39th Annual Hawaii International Conference on*, volume 9, pages 235a–235a, Jan 2006.

[14] Giuseppe Lo Re, Fabrizio Milazzo, and Marco Ortolani. Secure Random Number Generation in Wireless Sensor Networks. In *Proceedings of the 4th International Conference on Security of Information and Networks*, SIN '11, pages 175–182, New York, NY, USA, 2011. ACM.

[15] Antonio Lalomia, Giuseppe Lo Re, and Marco Ortolani. A Hybrid Framework for Soft Real-Time WSN Simulation. In *Proceedings of the 2009 13th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications*, DS-RT '09, pages 201–207, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync Protocol for Sensor Networks. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, SenSys '03, pages 138–149, New York, NY, USA, 2003. ACM.