Implementation and Validation of a New Protocol Stack Architecture for Embedded Systems

Danilo Blasi, Luca Mainetti, Luigi Patrono, and Maria Laura Stefanizzi

Original scientific paper

***Abstract*: The worldwide spreading of Internet, in combination with the development of new low power and low cost embedded devices, has enabled the so-called Internet of Things vision. Wireless Sensor Networks represent an invaluable resource for realizing such scenario, inside which new and innovative applications could be developed. However, the low availability of resources and the reduced processing capacity of the target embedded platforms make the development of the next-generation applications very challenging. This paper proposes an innovative system architecture, called STarch, able to simplify the development of new applications and protocols for resource-constrained objects. It is meant to follow the software engineering principles and to support a wide range of applications, making both the programming easier and the code portable over multiple hardware platforms. STarch simplifies the network configuration process, through the use of an automatic mechanism based on the XML language and it runs properly on different operating systems, including FreeRTOS and Contiki. The feasibility of the proposed architecture has been proved by using a test bed approach, while an extensive performance analysis have been carried out in order to demonstrate its effectiveness in terms of memory requirements and processing delays.**

***Index terms*: Wireless Sensor Networks, System Architecture, Contiki, Protocol Design, Performance Evaluation.**

I. INTRODUCTION

The next-generation Internet aims to assert the concept of Internet of Things (IoT) [1], according to which the everyday objects that surround us will become proactive actors of the global Internet, with the capability of generating and consuming information. In such vision, Internet is no longer seen as a tool for linking people to services, but as a means to allow the realization of the new Machine-to-Machine (M2M) paradigm. Wireless Sensor Networks (WSNs) can facilitate this evolution process, thanks to their ability to self-configure and self-organize. Moreover, the recent progress in embedded systems has enabled the development of low-cost, low-power, multifunctional sensor nodes, characterized by ad hoc communication. Such nodes are able to capture important data from the surrounding environment (e.g., humidity, pressure, temperature) and transmit them to a processing centre for a proper utilization. This main feature makes WSNs suitable for the development of a wide range of applications, such as building automation, surveillance, military operations, healthcare, and logistics. However, sensor nodes are typically battery powered devices with very limited resources and, therefore, the development of next generation applications has to be tailored to meet the resource constraints of these devices [2, 3, 4, 5]. Furthermore, the wide range of sensor nodes available on the market are characterized by very different hardware features, especially in terms of computation capabilities, communication range and power consumption. A protocol optimized for a specific device may not work properly on a different hardware platform. All these issues make the design of new applications and network protocols for WSNs very challenging.

A key element for the design of new applications for WSNs is the development of the protocol stack. Typically, developers design and configure a protocol stack according to standard models (e.g., the ISO/OSI model) [6, 7]. Such a stack is characterized by a tight coupling between protocols at adjacent layers of the stack. Each of them has to interface directly and exclusively with the lower layer and with the higher layer to request and provide communication services. Moreover, such stack is based on isolation among different layers, so that each protocol cannot share own control information with the rest of the stack without violating such a constraint. Consequently, a stack built by following the classical approach should be used as a whole and its porting to different platforms might require significant effort. This is even more true for those proprietary solutions, often not disclosed, which are highly optimized for very specific applications and platforms, and merge protocols as much as possible.

An alternative approach for the development of a protocol stack for embedded systems is the cross-layer design [8, 9, 10, 11]. It basically consists in making protocols at different layers able to share information and to collaborate with each other, in order to reduce considerably the waste of resources due to useless redundancy. However, this approach may lead to produce "spaghetti" code, which increases the coupling among different protocols, making the code poorly portable.

A possible solution to the previous problems is the definition of a new programming approach based on the use of a software core able to (i) ease the development of a protocol in a modular way, (ii) enable the communication among protocols through an intermediary and without direct coupling,

Manuscript received July 10, 2013, revise September 30, 2013.

L. Mainetti, L. Patrono, and M.L. Stefanizzi are with the Department of Innovation Engineering at University of Salento, Lecce, Italy (email: {luca.mainetti, luigi.patrono, laura.stefanizzi}@unisalento.it).

D. Blasi is with the Advanced System Technology/Ultra Low Power Radio and Network at STMicroelectronics, Lecce, Italy (email: danilo.blasi@st.com).

(iii) support different cross-layer optimizations, thanks to an inherent programming structure that allows to easily find out and avoid redundancy, and (iv) enable the immediate porting of single protocols and of entire stacks among different and heterogeneous hardware platforms through a protocol abstraction layer.

At present, only a few architectures with the same optimization purposes have been introduced in the literature. One of the most performing solutions is the Information DRiven Architecture (IDRA) [12, 13]. It is expressly designed to support next generation applications on resource constrained networked objects. IDRA presents useful optimizations at an architectural level that include support for cross-protocol interactions, energy efficiency, quality-of-service (QoS), mobility and heterogeneous network. However, it does not provide a hardware abstraction layer able to simplify the porting of the implemented network protocols among different HW platforms.

This paper presents a new system architecture, named STarch, which meets all the requirements above mentioned. It is meant to follow the principles of SW engineering and to support different and heterogeneous applications, making both the programming easier and the code portable over multiple hardware platforms. More in detail, STarch allows indirect and dynamic interaction among protocols at any communication layer, together with the use of common data spaces for coordination and information sharing. It follows the principle of the "blackboard model", where several network protocol agents read and write on a common information base in order to share knowledge and cooperate in support of network setup and management. STarch also simplifies the network configuration process through the use of an automatic mechanism based on the XML language [14]. It is able to run properly on different operating systems (OSs), including FreeRTOS and the Contiki [15]. The use of the Contiki's simulation/emulation environment (i.e., the Cooja network simulator and the MSPsim device emulator) further simplifies the cumbersome and time-consuming job of developing and debugging applications for WSNs. In order to demonstrate the actual portability of the proposed approach a simple routing protocol has been implemented and validated by using four embedded devices. Furthermore, an extensive simulation performance analysis has been carried out to demonstrate the effectiveness of STarch in terms of memory requirements and processing delays.

The rest of the paper is organized as follows. Section II reports some recent related works. Section III provides a description of the proposed software architecture STarch. Implementation details about porting issues of STarch on the Contiki OS and the test environment are summarized in Section IV, while in Section V, simulation and experimental results are discussed. Conclusions are drawn in Section VI.

II. Related Works

This section summarizes the most important research studies related to the design of new solutions able to simplify the development of network protocols presented in the literature.

In [16], Dunkles et al. present the Chameleon architecture, which is part of the Contiki OS and aims to simplify the development of new protocol solutions by separating the protocol logic from implementation details related to the packet creation. Specifically, authors introduce the concept of "packet attributes" as an abstract representation of all information usually contained in the packet headers. Packet attributes, together with application data, are transformed into packets by some specific header transformation modules, which know all details about the headers management. However, such modules are implemented in the higher layers of the network stack, above the MAC protocol, and therefore, the MAC header does not take any advantage from their use.

In [17], Finne et al. present Chi, a full-system configuration architecture, which aims to improve network performance by separating protocol logic and system configuration. This separation allows to customize the configuration without changing the inner logic of a protocol. The main component of Chi is a blackboard that holds the system configuration along with the relevant part of the system state. It provides also an abstraction of all shared variables, accessed through an independent module in each sensor node. Besides providing a programming interface for accessing such variables, the blackboard has also a notification process for subscribers of value modifications.

The marshalling/unmarshalling problem (i.e., network-byte-order and host-byte-order conversions) is addressed in [18]. The authors present a solution able to mask this problem to the developer, who can access to the packet structure just as to a data structure in the memory. This data structure matches the packet layout, and accessing it is pretty much like accessing to a regular type in the C language. The compiler assures that this type has the same representation on all platforms and generates any proper conversion code. This solution simplifies the development of new applications and protocols, and the adaptation of existing ones, in order to make them interoperable on different hardware platforms.

The WASP (Wirelessly Accessible Sensor Populations) European IST Project [19] aimed to develop an integrated model for implementing applications using wireless sensor networks. One of the main outcomes of this project was the definition of the so called WASP Postmaster, based on the concept of communication decoupling among protocols through the use of letters, which are mainly associated to packets and timers. The WASP architecture provides also an abstraction level to the reference hardware/software platform, thus permitting an easy portability of the code on multiple platforms.

Let us observe that the works summarized above do not present a complete architecture able to support the development of next-generation applications but they only introduce optimizations that aim to solve individual problems. A more complete system, called IDRA architecture, is proposed in [12, 13].It simplifies the development of new network protocols by delegating common operation to the system. The system is responsible for queue provisioning,

packet generation and packet interactions. More in detail, in IDRA, protocols are not tasked with header creation or manipulation but they make use of packet attributes to add information to the packet. As a result, the protocol logic and packet representation are decoupled. Furthermore, this architecture uses a shared buffer to store outgoing and incoming packets, so as to limit the total number of multiple packets in different layers. The IDRA architecture also provides mechanisms for supporting advanced network requirements for next generation sensor application. In particular, it supports energy efficiency requirements by reducing the number of packet transmissions, enforces QoS by managing different packet priorities and uses shared neighbour table to support network mobility.

STarch differs from all the presented architectures for two main aspects: (i) it is easily portable among different platforms and system architectures, because it adopts a platform abstraction layer; (ii) it simplifies the network configuration process through the use of an automatic mechanism based on the XML language.

III. System Design

STarch is a new framework for network protocol design and programming. It replaces the traditional paradigm of direct/coupled communication among application layer (APP), network layer (NWK), Hardware Abstraction Layer (HAL), and among protocols of a stack, by using a broker able to guarantee an indirect/decoupled communication. This broker is called Network Layer Manager (NLM) and it acts as a central coordinator. It provides the developer with a set of Application Programming Interfaces (APIs) able to simplify the process of implementing and validating a new application or protocol solution. Specifically, the NLM module manages the common aspects of protocol design and implementation, like packet handling, timer scheduling, inter-layer communication and coordination, and adaptation to the host platform. Details of these services are hidden to the developer, who can focus mainly on the definition of the protocol's behaviour, rather than on its inner logic. Figure 1 shows the STarch block diagram, while more information about the most important features of the proposed architecture are given in the following.

A. Modularity

STarch allows interaction among different protocols without that they have knowledge of identities of each other. In STarch, each protocol is represented by a so-called *Network Entity* (NENT),which identifies a category of protocols with similar functionalities (e.g., reliable/unreliable transport, address-/data-centric routing, tracking, time synchronization, service discovery, etc.). Protocols of the same class can replace each other without affecting the overall behaviour of the stack. This simple but useful feature allows increasing the modularity of the architecture to a very large extent.

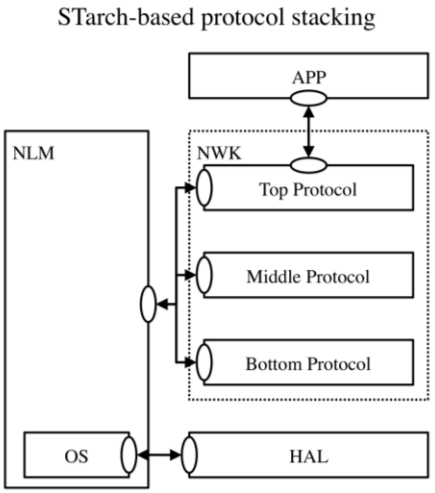


Fig. 1. STarch protocol stack architecture

Before starting their activities, protocols must register themselves to the NLMby calling a specific API, declaring their class identifier and the callback point (i.e., entry point) through which they will receive messages from other entities. The NLM registers such an information and replies to protocols by sending them, through the associated callbacks, an *acknowledgement* (ACK) letter without attachments. Once received the ACK letter, each protocol is allowed to start running. This feature could be useful to perform the replacement of single protocols or of the entire stack at run-time. A device management logic might decide to replace a running protocol with a new one still not active. In this case, the former must deregister itself by calling another NLM API. The NLM first acknowledges the protocol's request, secondly removes the information associated to the old protocol. The new protocol must register itself to the NLM with the already mentioned procedure.

B. Cross-layer communication

STarch entities communicate with each other in a cooperative way through a *mail exchange service*. STarch uses a *letter* as data and information carrier, and the NLM can be considered a postmaster that sorts and dispatches the stack's internal mail. Every entity can write letters directed to any other entity, which will later receive, read and handle their contents. This communication strategy allows to route data and control information among protocols in a very arbitrary way or, in other words, to change easily the protocol execution order at run-time, based on the actual network management needs. Let us observe that STarch does not charge a specific module with this task. Every entity, in principle, may decide to change the destination of its letters. The "path" followed by letters and the way it might change at run-time is out of the scope of this work. The described way of exchanging information among NENTs makes the cross-layer communication *quite anonymous*, since letters writers and readers do not need to know exactly the identity of the

protocol which requests or provides a service, but they only need of an abstract view of it.

A delivery priority level is assigned to each letter. Before being processed, letters are temporarily stored inside some First In First Out (FIFO) queues, called *mailboxes* (MBOXs), arranged according to their delivery priorities. The NLM module manages the mailboxes, sorts the letters and dispatches them, giving precedence to letters that are more time critical than others. Let us observe that in such a way STarch architecture is able to provide support for specific QoS requirements.

Each letter may also have an attachment. Timers are usually attached to letters. In this case, the writer and the reader likely correspond to the same entity. Anyway, the developer can define new letter types, with specific attachments. This is a flexible and simple way to allow protocols to share complex object by means of the mail exchange mechanism. More in detail, the letter types are declared at compile-time and currently they include: acknowledgement, packet, timer and user, the latter meant for any other purpose. However, this list can be extended based on actual needs. Of course, source and destination entities must agree on the kind of letters they want to exchange, according to the "communication protocol" existing (or to be defined) between the two entities. All the entities belonging to the same class must be able to prepare and interpret the letter types defined for each of them. The letter type is an attribute of the letter and is used by the letter's producer/consumer to interpret the letter's attachment. The letter is a carrier of many types of information (i.e., attachments). It is received by an entity through its registered callback. Once received the letter, the entity can read the letter's type attribute and handle the letter accordingly. To clarify the letter concept the sequence diagram of creation, delivery and use of a packet letter is shown in Figure 2.

C. Platform virtualization

STarch provides an abstract view of the actual hardware/software platform to applications and network protocols, in order to make the code extremely portable on heterogeneous devices. As shown in Figure 3, it brings a

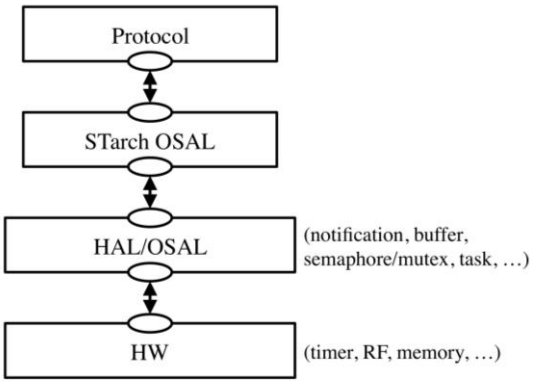


Fig. 3. Virtualization of the software and hardware platform

STarch Operating System Abstraction Layer (OSAL). This layer interacts with the HAL or the OS of the host device and hides their features to the developer, enabling, in such a way, a platform-independent programming. Consequently, a protocol stack designed and developed on the STarch OSAL is straightforwardly portable above different software/hardware platforms. The porting effort is reduced to establish functional links between corresponding APIs of the STarch abstraction layer and of the actual underlying operating system.

D. Dynamic stack configuration

Besides the mailboxes, the NLM provides three shared memory areas to guarantee asynchronous communication among the network entities. One of them is called *Configuration Information Base* (CIB) and it is mainly intended to store protocol stack configuration settings specified by the user in terms of attribute-value pairs. This centralization of the configuration management allows to significantly reduce the code redundancy. It also reduces the memory occupation and allows to propagate a configuration change to multiple protocols as soon as they request for an update. Furthermore, separating configuration from protocol logic enables consistent dynamic reconfiguration, without

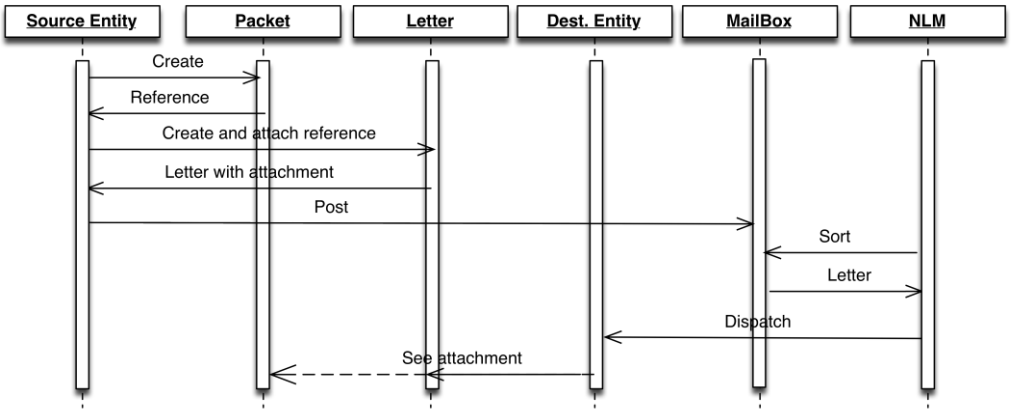


Fig. 2. Sequence diagram of creation, delivery and use of a packet letter

changing the protocol implementation. This is possible if, for instance, a commissioning network entity is invoked at run-time to change some CIB parameter settings, and later requests for a reset of the system. In such a case, the already registered network entities should update their configuration status by reading the latest values of the parameters of interest from the CIB.

Every network entity can access to information contained in the CIB area according to its privileges. In particular, every CIB parameter can be written by a single network entity, said the "owner" of the parameter itself, and read by a restricted set of entities, based on a fixed security policy. Let us observe that every CIB parameter is associated with a data type, which is identified by a code that is known to all the entities. Based on this code, every entity can read the right format of a CIB parameter's content/value. If a parameter is not present, a proper error code is returned to the querying entity.

E. Network management

Following the specifications of standard models (e.g., the ISO/OSI model), protocols cannot share with each other information collected from the network, unless a number of well-defined SAP primitives (e.g., request/confirm) are available for this purpose. Addresses, routes, distances, positions, statistics concerning other nodes in the network are examples of information collected by different protocols separately, often with high degree of redundancy, and waste of dedicated resources (e.g., memory, processing, bandwidth). In STarch, all network entities share with each other information obtained from other nodes in the network, through the use of a repository called *Network Information Base* (NIB). It is actually a table, whose entries are associated to different network members.

Entities can insert, delete and query such table by means of a powerful *NIB Query Language* (NQL), very similar to the SQL language. The NLM must check whether a network entity is allowed to execute a query or not, based on its privileges. Such an admission control is needed to protect NIB information against security attacks. More in detail, the NQL is based on query statements (i.e., *select*, *insert*, *delete*) and clauses (i.e. *what*, *where*, *order_by*). The statements are used to retrieve, add o remove information to/from the NIB. As previously introduced, the NIB is seen as a list of records (i.e., rows of a table), each composed by a list of attributes (i.e., columns of a table). The statements rule the access to entire NIB records, while the clauses are used to filter attributes and records, and to sort records based on specific attributes. The *select* statement includes all the clauses and returns a set of records, which is a view of the NIB, filtering out all the NIB attributes and records that are of no interest for the querying entity. The *insert* statement allows adding a new record to the NIB by specifying only a subset of attributes and making the others assume a default value. Finally, the *delete* statement allows removing a record from the NIB by specifying a matching condition.

F. Simplifying network protocols

As previously mentioned, the STarch architecture simplifies the design of a network protocol taking care of common operations, such as packet creation and handling.

More in detail, the NLM provides a mechanism for optimized packing that makes use of "*packet attributes*", which are an abstract representation of information contained in the packet headers. Network entities may read and/or write attributes of their interest. Moreover, the NLM takes care of constructing outgoing packets and of parsing incoming ones. Every packet attribute is characterized by: (i) a unique name used as identifier, (ii) a type associated to the host memory representation of the attribute value, (iii) a bit size of the portion of the host representation that will be actually packed, (iv) a bit offset for the alignment of the attribute in the packet, and (v) a "More" flag indicating whether the attribute is grouped with the following one in the packet to form a single-byte or a multi-byte field.

A packet may optionally contain special attributes used to transfer among different protocol layers additional information about the packet itself, such as Received Signal Strength Indication (RSSI), Link Quality Indication (LQI), timestamp(s), statistics (e.g., number of re-transmissions), transmitter and receiver addresses.

Furthermore, the NLM provides some shared buffers, called *Packet Pool* (PKP) that allow to queue more than one outgoing or incoming packet. Such buffers are organized in different queues, characterized by a different priority level and served according a priority model (i.e., the oldest packet in the transmission/reception queue with the highest priority is served first).

G. Support for heterogeneous networks

In heterogeneous environments, neighbouring devices might use different communication technologies. To limit the dependence of network protocols on a specific radio technology, in STarch the radio management is entirely delegated to a network entity, called Radio Manager (RMNG), which is implemented by a MAC protocol or a wrapper of a radio driver.

The RMNG entity was designed to decouple the higher layer protocols from underlying radio technologies. For example, a routing protocol could be used with different radio technologies, without having direct knowledge of which radio technology is currently used. The high layer protocol simply uses network connections established by the actual link layer protocol. Furthermore, the RMNG, if extended with the needed logic, could switch between different network interfaces, provided that the host device is an hardware gateway with multiple network interfaces installed. The choice of the right network device to use may be based on opportunistic needs (e.g., use of the shortest link, use of the largest bandwidth) or dictated by traffic flow (e.g., from wireless sensor device to Internet).As a result, multiple

services can reside on the same node, each with one or more associated communication interfaces.

H. Support for the Internet Standard IPv6

The STarch architecture provides an adaptation layer for the Internet standard IPv6 [20], which is backed by a network entity that redirects IPv6 packet letters to the underlying radio manager and vice versa and, at the same time, allows bridging IPv6 networks with other STarch-based proprietary networks.

More in general, when an application wishes to send data or to be notified of incoming data through the STarch architecture, it has to use a Network API (NAPI) that is delegated to an entity, which will make the link between the application and the network stack. Indeed, this is a transport entity that allows communication among multiple applications' ports and can operate in two modes:

- *Transmission mode*: it gets the message from the source application and splits it in multiple fragments if the message is too long with respect to the Maximum Transmission Unit (MTU) of the used radio interface; then, each fragment is inserted into a packet that is, in its turn, enveloped into a letter to be later sent to the NLM for dispatching;
- *Reception mode*: the entity receives a letter for a destination application, then extracts the packet from it and, in particular, the message fragment; such a fragment is reassembled with the other fragments received previously and next from the same source application, in order to restore the complete message that is finally passed to the destination application's callback.

Note that similar entities, which would adapt the STarch architecture to work with any existing standard or legacy solutions, can be easily developed.

I. Support to network configuration

Configuring a network may become very complex, especially if the number of parameters used by applications, protocols and system is high. To cope with this problem, the STarch framework can be configured through an automatic process that involves the use of the XML language. Specifically, the developer has to create a system configuration file named `config.ini`,encoded in XML according to a formalized and well-defined data dictionary. This configuration file can be written in two ways: via a text editor (i.e., the programmer manually enters all the necessary tags for the correct configuration of the file) or via a Graphical User Interface (GUI). This GUI is implemented as a plugin for the Eclipse Integrated Development Environment (IDE) [21] and it aims to make the creation of the configuration file easier and less time-consuming.

Once installed, the plug in allows the configuration of all the information contained in the `config.ini` file without worrying about the physical structure of the file itself (i.e., the correct utilization of the XML tags according to the data dictionary). In particular, the plugin provides a wizard that assists the programmer in choosing the appropriate parameters without taking care of the constraints to be respected for editing, since these constraints are calculated by the application and provided to the user through graphics widgets.

IV. Implementation Details and Test Environment

In this section, details about the porting of the STarch architecture on the Contiki OS are discussed, and, afterward, the platform used to validate the proposed architecture and the test settings are described.

A. Porting to the Contiki OS

As previously declared, STarch provides its own abstraction layer on top of any platform-dependent operating system, guaranteeing that STarch-based code can be easily ported to different hardware and software platforms. When a module needs to invoke a low level system service it simply calls the associated STarch API. Therefore, porting STarch to the Contiki OS has required only few and well-defined steps.

Contiki is a popular open-source operating system targeted to small microcontroller architectures. As shown in the right side of Figure 4, Contiki is characterized by a communication stack organized in several layers, in which both protocol solutions and radio transceiver features can be easily configured. The lowest layer of the stack is the *NETSTACK_CONF_FRAMER*. It is in charge of the data packet format conversion before the transmission over the physical channel. The upper layer is the *NETSTACK_CONF_RADIO*. It directly manages the wireless transceivers features through the appropriate device driver. These two first levels can be considered the PHY layer of the ISO/OSI model. The third layer of the Contiki stack is the *NETSTACK_CONF_RDC*, which cannot be directly mapped to the ISO/OSI model. It is just below the MAC layer, identified as *NETSTACK_CONF_MAC*, and it is in charge of managing the radio duty cycling to provide energy saving capabilities. The last layer of the stack is the *NETSTACK_CONF_NETWORK* providing the functionality of the network layer of the ISO/OSI model.

Considering the above described communication stack architecture, the STarch porting has been realized by acting on the *NETSTACK_CONF_RADIO* and the*NETSTACK_CONF_RDC* layers as shown in Figure 4.More in detail, in order to ensure a proper management of the packets coming from the network, a simple RDC driver, able to interrupt the packets flows in the Contiki OS and send them toward the STarchOSAL, has been implemented. On the contrary, the STarch abstraction layer can directly interface with the RADIO driver to send a packet toward the network. The use of the NETSTACK_CONF_RADIO macro, defined in the Contiki OS, allows the abstraction from the actual radio driver.

In order to integrate STarch into the Contiki system a new Makefile (i.e., Makefile.STARCH), including all files needed to make STarch properly run, has been developed. To not modify substantially the structure of the Contiki OS, the new Makefile has been stored in the project's directory and included in the project's Makefile. Furthermore, to guarantee the correct working of the STarch architecture with all platforms supported by the Contiki with the minimum effort, all definitions related to STarch have been included in a new *project-conf.h* file, which can be easily included in new projects.

Another important aspect that has required particular attention during the porting activity was the management of the processes' execution flow. STarch can be run in both single-task and multi-task operating modes. The choice of the proper mode to be used depends on the features of the operating system which STarch is executed over, since it could not support the multiprogramming mode or could support it only partially. Let us observe that Contiki supports *multi-threading* by implementing it as a library that can be optionally linked with programs that explicitly require it. This feature is important whenever a lengthy computation is performed, given that the event-driven kernel will monopolize the CPU and will make the system unresponsive to external events. However, the thread scheduling has to be planned and designed by the programmer who decides its logic based on timer expiration or on the execution of specific conditions. This means that all threads must be declared in advance and used together within the same process structure in order to define a scheduling logic.

The discussed issues suggested us to consider the single-task operating mode, to assure a proper working of STarch on the Contiki OS.

Finally, the last step of the described porting activity has involved the implementation of the APIs provided by the STarch abstraction layer, which allows interfacing with the underlying operating system. These APIs are essentially related to packets transmission and reception, time and timers management, memory management and the terminal activity.

C. Test settings and data collection scenario

As previously introduced, the performances of the STarch architecture have been evaluated by means of extensive simulation campaigns while an experimental campaign has demonstrated the actual usability of the proposed architecture on real embedded devices.

All simulations and tests have been performed considering a simple Link State Routing (LSR) protocol, based on the following rules:

- Each node sends periodically and in case of connectivity changes an advertisement packet, containing its address, a sequence number, the list of the last discovered neighbours, and the number of hops traversed by the packet itself (initially reset).
- When a node receives an advertisement packet, it checks whether a copy of the packet, recognized by the same sequence number, has been previously received, and, in such a case, it discards the packet. Otherwise, the node creates or updates an entry in the local Link State Table (LST), containing the addresses of the node that originally produced the message and of its actual neighbours. Finally, the node increments the counter of the hops traversed by the packet and rebroadcasts it to inform other nodes of the included information.
- Once completed the LST initialization phase above, when a node needs to send a data packet to a target destination, it first establishes the shortest path in number of hops towards the target based on the information stored in the LST, then sends the packet to the next hop along the path found. The packet will be relayed hop-by-hop until it is received by the destination, which will reply with an acknowledgement packet.

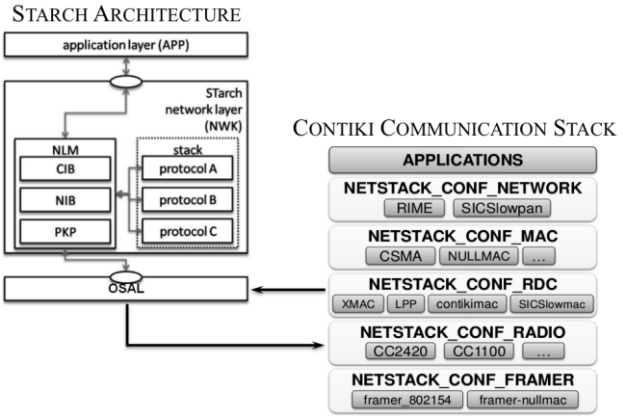


Fig. 4. STarch porting to the Contiki OS.

The experimental campaign (called STM3210_TEST in the rest of the paper) has been carried out by using four STM3210-EVAL Evaluation boards [22] of STMicroelectronics, equipped with the SPIRIT1 Sub 1-GHz transceiver [23] (Figure 5). The selected board is a complete development platform for STMicroelectronic's ARM Cortex-M3 core-based STM32F103ZET6 microcontroller, while SPIRIT1 chip is a very low-power RF transceiver, intended for RF wireless applications in the Sub 1-GHz band and air data-rate programmable from 1 to 500 Kbps. The FreeRTOS operating system was adopted. The test has been performed in an indoor environment, positioning the four nodes at the corners of a square with arbitrary side and numbering them according to the clockwise direction. The maximum length of a communication link is equal to the square side. According to the implemented application, node 3 reads a text and sends the read words to node 1, which writes them on a terminal. After sending a word and before sending the next one, node 3 waits for an acknowledgement from node 1. This process is repeated until the transmission of the whole text is completed and then it restarts endlessly.

The simulation campaigns (called WISMOTE_SIM in the rest of the paper) have been carried out by using Cooja, the Contiki network simulator. Cooja integrates MSPsim, a tool that can emulate motes based on the MSP430 microcontroller. Cooja/MSPsim provides cycle-accurate simulation of the individual devices, as well as bit-level accurate simulation of

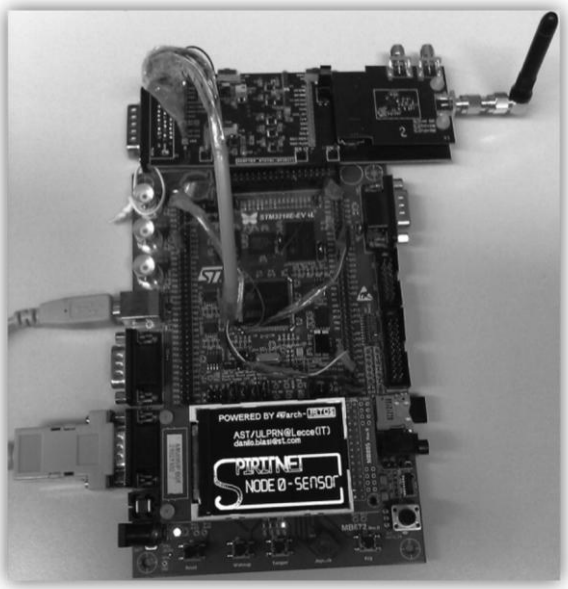


Fig. 5. STM3210-EVAL Evaluation board equipped with the SPIRIT1 Sub 1-GHztransceiver.

their radio transceivers. As a consequence, Cooja/MSPsim allows running the exact same binaries in the simulator as on actual hardware. Specifically, we considered the WiSMote platform [24], which is equipped with an MSP430F5 microcontroller having 16 kB of RAM and 256 kB of flash memory. The board integrates the CC2520 transceiver, a 2.4 GHz wireless transceiver compliant with the IEEE 802.15.4 standard. The same application developed for the test bed has been used, while a simple network of two nodes has been evaluated. In this case, node 1 (called sensor in the rest of the paper) replies to node 2 (called sink) sending back, together with the acknowledgements, also the text received. This simple network allows to evaluate the STarch performance minimizing the influence of the routing protocol. To better appreciate the effectiveness of the proposed architecture, an ideal channel, characterized by a packet error rate equal to zero, has been simulated. Furthermore, to analyse the system behavior with different levels of network load, five data rates have been considered during the simulations: 1 packet per second (high load), 1 packet every 15 seconds, 1 packet every 30 second (medium load), 1 packet every 45 seconds, and 1 packet per minute (a typical data rate used in sensor networks [25]). For each packet rate, three different values of message length, which match to packets sent using 1 fragment, 2 fragments, or 3 fragments, were considered. The main simulation parameters are reported in Table I, while the results of the performed analysis are discussed in the next section.

During the simulation campaigns, the following metrics have been evaluated: (i) average fragment transmission delay (i.e., the time interval between the creation of a letter, containing the application message, and its transmission to the radio driver); (ii) average fragment reception delay (i.e., the time interval between the reception of a packet by the radio driver and the end of its management by the recipient protocol); (iii) average routing protocol delay (i.e., the time interval between the delivery of a letter to the routing protocol and the end of its processing); (iv) average reaction time to a timer expiration (i.e., the time interval between the notification of the expiration of a timer in the Contiki OS and the time instant when the corresponding timer letter is processed from STarch); (v) memory footprint (i.e., the memory footprint of the whole architecture and of each components). The last metric has been measured considering the following memory sections: .text, including the functions of a program; .data, which contains initialized global variables; .bss, which contains uninitialized global variables; and .rodata, which contains the constant global variables.

TABLE I
SIMULATION PARAMETERS

Parameter	Value
Network Topology	Chain
Number of nodes	2
Data Rate	1 packet per second 1 packet every 15 seconds 1 packet every 30 seconds 1 packet every 45 seconds 1 packet per minute
Number of fragments	1, 2, 3

Finally, we observe that all simulations were carried out by using the independent replications method and all results are characterized by a 95% confidence interval with a 5% maximum relative error.

V. RESULTS

In this section, main results obtained by both simulations and experimental campaigns are separately reported.

A. System validation

Figure 6 shows the messages generated by nodes 3 and 1 during the STM3210_TEST.For each node, the first set of messages is related to advertisement packets exchanged to build the LSR table. Next, the node starts printing application and routing messages associated with data packets and acknowledgement packets, according to the node role (i.e., data source, forwarder, or data destination).The test bed has demonstrated the actual functionality of the architecture and the portability of the STarch-based protocol as a function of devices characteristics (e.g., clock speed, memory) and of the operating systems.

B. Performance analysis

The results of the WISMOTE_SIM tests are reported and discussed in this sub-section.

Figure 7 shows the average delay for the transmission of a packet or, in case of fragmentation, of a fragment, versus the

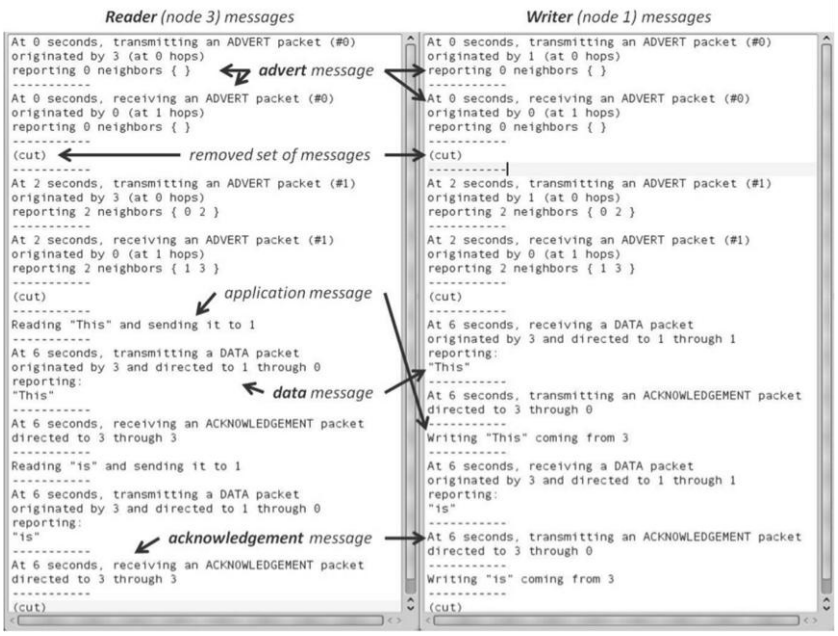
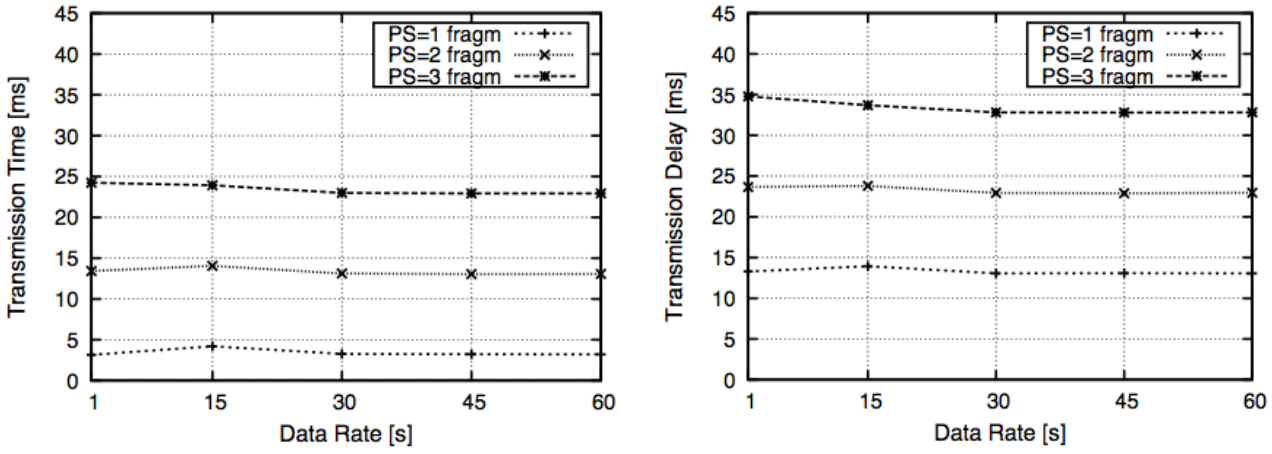


Fig. 6. Output messages collected for nodes 1 and 3 during the STM3210_TEST carried out by using four STM3210-EVAL Evaluation boards running FreeRTOS.

data rate for the sink and the sensor nodes. All measured values are expressed in ms, while the used data rates are reported in seconds, indicating the elapsed time between two consecutive packets. The three used packet sizes are labelled as PS with the indication of the number of fragments needed to transmit a packet. Let us observe that all curves exhibit a constant trend, which indicates that in all configurations the measured delays are not affected by the data rate. On the contrary, the transmission delay significantly increases when the packet size increases. This behaviour is mainly due to queuing delays of different fragments. Indeed, a message fragmentation causes the creation of two or three letters that are immediately dispatched to the routing protocol to be handled. Obviously, the protocol, but in general the whole architecture, can manage only a letter at a time, and the queuing delay of different fragments significantly affects the average transmission delay. Moreover, in the sink node (Figure 7.a), the protocol cannot transmit a queued fragment until it has received an acknowledgment from the sensor, referred to the previous transmission. As consequence, each fragment suffers from a different transmission delay. Although the curves related to the two nodes show a similar trend, however, the transmission of a fragment on the sensor node requires a longer execution time than that measured on the sink node. This result is due to the more complex activity of the routing protocol on the sensor, which has to manage the acknowledgements and, before transmitting, has to wait the reception of a fragment from the sink.

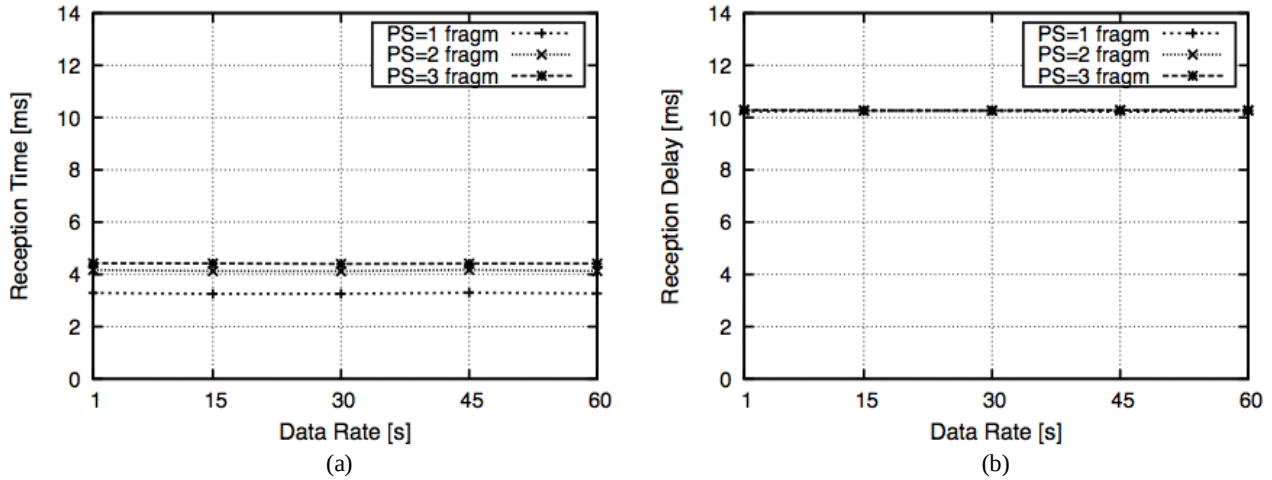
The results of the average reception delay are reported in Figure 8. Also in this case, the measured delays are not influenced by the data rate, since all curves are characterized by a constant trend. Moreover, Figure 8.a shows that the sink reception delay significantly increases for higher values of packet size. When a packet is received, a letter containing such information is immediately sent to the NAPI. At the same time, if a packet is composed of more than one fragment, a letter is sent to the radio manager, in order to require the transmission of the first queued fragment. In such a case, the reception delay increases. A similar behaviour is not shown by the sensor node (Figure 8.b) because it uses a different procedure for packet transmission. Regardless of the number of fragments that compose the message, after receiving a fragment, the sensor node always transmits an acknowledgment containing the application payload. Furthermore, let us observe that the sensor node delays are significantly greater than those of the sink. Upon receipt of a fragment, in fact, the sensor node first requires the packet transmission, sending a letter to the radio manager, and then manages the received fragment. This produces a lengthening of the packet delivery time.

Figure9 shows the protocol delay analysis in the sink and the sensor nodes. Let us observe that all kinds of letters managed by the routing protocol have been considered in this analysis. Regarding the sink node (Figure 9.a), the protocol delay slightly increases for higher data rates. In the analysed scenario, in fact, the sink node periodically checks the status of all routes, and when the timer associated with this check expires a proper letter is created. Consequently, when the data rate decreases also the number of application letters decreases, on the contrary, the number of letters due to a timers expiration remains constant. However, timer letters require less processing time than data letters. Furthermore, when the packet size increases also the protocol delay increases, since a great number of data letters is produced. On the contrary, in the sensor node (Figure 9.b), the protocol delay is not influenced by the data rate and the packet size considered, because this node does not manage timer letters. These results, therefore, represent a more reliable estimate of the protocol average execution times.



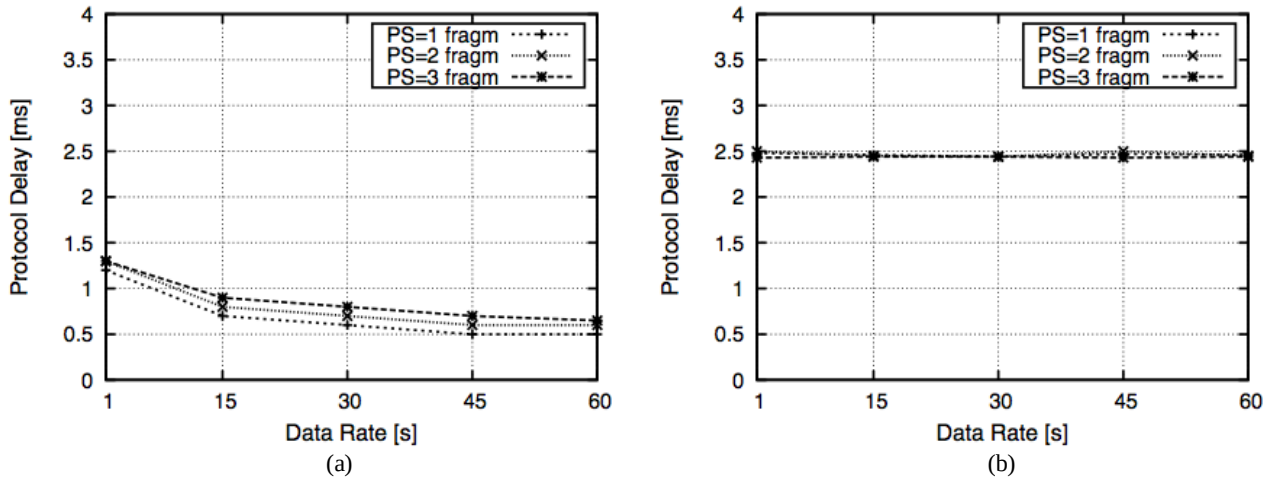
(a) (b)

Fig. 7. Average delays for the transmission of a fragment versus the data rate during the WISMOTE_TEST for: (a) the sink node, and (b) the sensor node.



(a) (b)

Fig. 8. Average delays for the reception of a fragment versus the data rate during the WISMOTE_TEST for: (a) the sink node, and (b) the sensor node.



(a) (b)

Fig. 9. Protocol delays versus the data rate during the WISMOTE_TEST for: (a) the sink node, and (b) the sensor node.

Although the implemented protocol and the considered scenario significantly affect the obtained results, however, the performed analysis allowed us to approximately estimate the delay introduced by the Starch architecture. Specifically, considering the sink node, a packet size equal to 1 fragment, and the maximum data rate (i.e., 1 s), in order to reduce as much as possible the influence of the routing protocol, the STarch delay can be evaluated using the following equation:

$$STarch\ Delay = \frac{(Delay_{TX}+Delay_{RX})-2*Delay_{PT}}{2} \quad (1)$$

where $Delay_{TX}$ represents the average transmission delay, $Delay_{RX}$ represents the average reception delay, and $Delay_{PT}$ is the average delay introduced by the routing protocol. This value is approximately equal to 1971,665 μs. The discussed tests confirmed that acceptable values for delay-tolerant applications, such as those developed for WSNs, are guaranteed.

Another interesting performance metric is the reaction time of the STarch architecture to the expiration of a timer. This analysis involves only the sink node, since, as previously described, the sensor node does not manage timers. The curves in Figure 10clearly show that the measured delays are not affected by the data rate and the packet size. It is important to observe that the average reaction time to timer expiration is about 540µs, a very low value for a WSN.

The memory footprint of the different architectural components is shown in Table II. The whole architecture requires about 16kB ROM and 8kB RAM memory, under the memory limit of most sensor nodes. Furthermore, this larger initial memory cost is compensated by the small size of the STarch network protocol, which only requires about 7 kB of memory. The memory requirements of the described components are limited. This shows that these techniques can be used in most typical sensor networks.

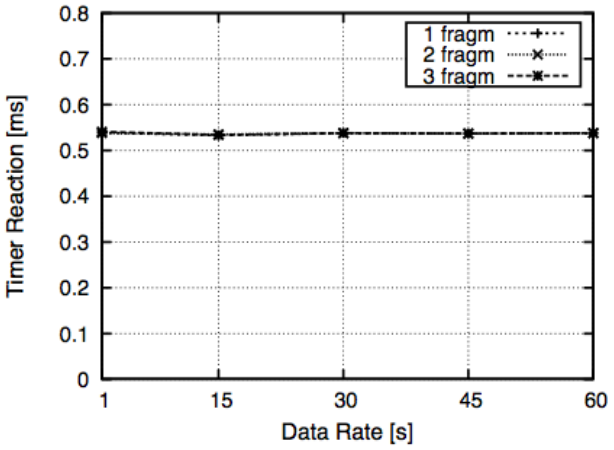


Fig. 10. Reaction time of the STarch architecture to the expiration of a timer during the WISMOTE_TEST.

VI. CONCLUSIONS

One of the most exciting challenges of the modern digital communication technology concerns the realization of the Internet of Things vision, inside which innovative applications could be developed. The most used networking protocols are application-specific, platform-dependent, and are stacked according to standard models; however, such protocol stacks are very often characterized by high degrees of coupling and redundancy. Alternative design approaches can be found in literature and, basically, aim to remove redundancy among protocols as much as possible isolating common information and functionalities.

This paper has presented and evaluated STarch, a novel integrated protocol architecture that shares goals of the aforementioned approaches and, in addition, decouples protocols from each other, from applications, and from the

TABLE II
MEMORY FOOTPRINT OF THE DIFFERENT ARCHITECTURAL COMPONENTS IN BYTES

Components	.text	.rodata	.data	.bss
NAPI	1988	0	0	2768
NLM	366	0	1	6
NIB	5832	0	0	0
CIB	498	0	0	0
PKP	3752	0	0	28
MAILBOX	988	0	0	22
NENT	192	0	0	6
TIMER	620	0	0	8
RADIO MANAGER	630	0	0	0
STARCH OSAL	696	0	1	20
CONFIG	0	0	858	4288
TOOL	498	0	0	0
PROTOCOL	7528	0	0	100
STARCH	**24066**			
	ROM		**RAM**	
	16060		8006	
PROTOCOL	**7628**			
	ROM		**RAM**	
	7528		100	

underlying SW/HW platform. STarch-based protocols are easily portable, maintainable, and configurable. Particularly, configuration is assisted by a powerful Eclipse-based GUI and applies to both protocols and whole host system.

The actual feasibility of the proposed architecture has been proved by using a test bed approach, while an extensive simulation campaign has demonstrated its effectiveness. Specifically, the analysis has shown that STarch is able to introduce only low delays and to quickly react to a timer's expiration. Moreover, it was demonstrated that the memory overhead of each components is small enough to be implemented on resource constrained embedded devices.

A performance evaluation by using real devices, such as STMicroelectronics' or other chip vendors' products, and the design of important improvements, including integration of different communication technologies (e.g., WSN and RFID) represent a natural evolution of this work.

ACKNOWLEDGEMENTS

This work has been partly financed by the BAITAH Project (Methodology and Instruments of Building Automation and Information Technology for pervasive models of treatment and Aids for domestic Healthcare), funded by the Italian National Operating Program (PON) 2007/2013, Information and Communication Technology (ICT) sector, with contract number PON01_00980. Furthermore, the authors thank Dr. Mirko Pizzaleo and Dr. Andrea Leo, who collaborated with the IDA Lab group of the University of Salento, Lecce, ITALY, for their support in the performance evaluation and the Eclipse GUI plug-in implementation, respectively.

REFERENCES

[1] L. Mainetti, L. Patrono, A. Vilei, Evolution of wireless sensor networks towards the Internet of Things: A survey, in: Software, Telecommunications and Computer Networks (SoftCOM), 2011.

[2] D. Alessandrelli, L. Patrono, G. Pellerano, M. Petracca, M. L. Stefanizzi, Implementation and validation of an energy-efficient MAC scheduler for WSNs by a test bed approach, in: Software, Telecommunications and Computer Networks (SoftCOM), 2012.

[3] D. Alessandrelli, L. Mainetti, L. Patrono, G. Pellerano, M. Petracca, M.L. Stefanizzi, "Performance *Evaluation of an Energy-Efficient MAC Scheduler by using a Test Bed Approach",* Journal of Communication Software and Systems, vol 9, no 1, 2013.

[4] L. Catarinucci, S. Guglielmi, L. Mainetti, V. Mighali, L. Patrono, M. L. Stefanizzi, and L. Tarricone, "An Energy-Efficient MAC Scheduler based on a Switched-Beam Antenna for Wireless Sensor Networks", Journal of Communication Software and Systems, Vol. 9, No. 2, June 2013, pp. 117-127

[5] L. Catarinucci, S. Guglielmi, L. Patrono, and L. Tarricone, "Switched-beam antenna for wireless sensor network nodes," Progress In Electromagnetics Research C, V ol. 39, 193-207, 2013.

[6] A. Dunkels, Full tcp/ip for 8-bit architectures, in: Proc. of the First International Conference on Mobile Systems, Applications, and Services (MobiSys), 2003.

[7] ZigBee Alliance, ZigBee Specification Document 053474r17.

[8] L. van Hoesel, T. Nieberg, J. Wu, P. Havinga, Prolonging the lifetime of wireless sensor networks by cross-layer interaction, in: Wireless Communications, IEEE, pp. 78–86, 2004.

[9] M. Sichitiu, Cross-layer scheduling for power efficiency in wireless sensor networks, in: Proc. of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM), 2004.

[10] A. Beach, M. Gartrell, S. Panichsakul, L. Chen, C. Ching, R. Han, X-layer: An experimental implementation of a cross-layer network protocol stack for wireless sensor networks, Department of Computer Science, University of Colorado at Boulder, Technical Report CU-CS-1051-08, 2008.

[11] S. Rotolo, D. Blasi, V. Cacace, L. Casone, "From WLANs to ad hoc networks, a new challenge in wireless communications: Peculiarities, issues, and opportunities", in M. Ilyas (Ed.), Handbook of Wireless Local Area Networks: Applications, Technology, Security and Standards, CRC Press, 2005

[12] E. De Poorter, E. Troubleyn, I. Moerman, P. Demeester, IDRA: A flexible system architecture for next generation wireless sensor networks, in: Journal of Wireless Networks, vol. 17, no. 6, pp. 1423-1440, 2011.

[13] E. De Poorter, I. Moerman, P. Demeester, Enabling direct connectivity between heterogeneous objects in the internet of things through a network-service-oriented architecture, in: EURASIP Journal on Wireless Communications and Networking, 2011.

[14] Extensible Markup Language (XML) 1.1 Specification, http://www.w3.org/TR/xml11/.

[15] A. Dunkels, B. Gronvall, and T. Voigt, Contiki a lightweight and flexible operating system for tiny networked sensors, in: Proc. of First IEEE Workshop on Embedded Networked Sensors, Tampa, 2004.

[16] A. Dunkels, F. Osterlind, Z. He, An adaptive communication architecture for wireless sensor networks, in: Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys), 2007.

[17] N. Finne, J. Eriksson, N. Tsiftes, A. Dunkels, T. Voigt, Improving sensornet performance by separating system configuration from system logic, in: Proceedings of the 7th European conference on Wireless Sensor Networks, (EWSN), pp. 194-209, 2010.

[18] K. K. Chang, D. Gay, Language support for interoperable messaging in sensor networks, in: Proc. of the 2005 workshop on Software and Compilers for Embedded Systems, 2005.

[19] The Wirelessly Accessible Sensor Populations (WASP) project, http://www.wasp-project.org.

[20] Internet Protocol Version 6 (IPv6) Specification, http://ipv6.net/RFC/rfc-2460-internet-protocol-version-6-ipv6-specification.html.

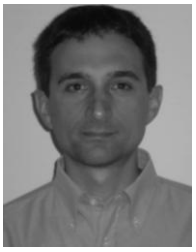
[21] The Eclipse Foundation, www.eclipse.org/.

[22] STM3210-EVAL Evaluation Board User Manual, http://www.st.com/internet/com/TECHNICAL_RESOURCES

/TECHNICAL_LITERATURE/USER_MANUAL/CD00178166.pdf
[23] SPIRIT1 Sub 1GHz transceiver User Manual, http://www.st.com/internet/com/TECHNICAL_RESOURCES/TECHNICAL_LITERATURE/DATASHEET/DM00047607.pdf
[24] Wismote node: http://wismote.org/doku.php
[25] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol.In Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys), Berkeley, CA, USA, 2009.

Danilo Blasi graduated summa cum laude in Computer Engineering at the University of Lecce in November 2001. Since January 2002, he has been working in STMicroelectronics (Lecce site) for the Advanced System Technology (AST) Research & Innovation group. His main activities include research and development of protocols and software architectures for wireless and wired digital communications. His research interests are in the areas of Wireless Sensor Networks, Low Power Radio and Networks, Auto Meter Reading and Embedded Systems. He is author of scientific papers about Cross-Layer Protocol Design, Routing and Medium Access Control (MAC) in Wireless Sensor Networks, Synchronization in Low Duty Cycle Networks.

Luca Mainetti is an associate professor of software engineering and computer graphics at the University of Salento. His research interests include web design methodologies, notations and tools, services oriented architectures and IoT applications, and collaborative computer graphics. He is a scientific coordinator of the GSA Lab - Graphics and Software Architectures Lab and IDA Lab - IDentification Automation Lab at the Department of Innovation Engineering, University of Salento. He is the Rector's delegate at the ICT.



Luigi Patrono received his MS in Computer Engineering from University of Lecce, Lecce, Italy, in 1999 and PhD in Innovative Materials and Technologies for Satellite Networks from ISUFI-University of Lecce, Lecce, Italy, in 2003. He is an Assistant Professor of Network Design at the University of Salento, Lecce, Italy. His research interests include RFID, EPCglobal, Internet of Things, Wireless Sensor Networks, and design and performance evaluation of protocols. He is Organizer Chair of the international Symposium on RFID Technologies and Internet of Things within the IEEE SoftCOM conference. He is author of about 70 scientific papers published on international journals and conferences and four chapters of books with international diffusion.

Maria Laura Stefanizzi graduated cum laude in Computer Engineering at University of Salento (Italy) in April 2012. Since January 2009 she collaborates with IDA Lab - IDentification Automation Laboratory at the Department of Innovation Engineering, University of Salento. Since July 2013 she is a PhD student at University of Salento. Her activity is focused on the design and validation through test beds on real devices of innovative applications and protocols aimed to reduce power consumption in Wireless Sensor Networks. She is also involved in the study of new solutions for the integration of RFID and WSN technologies.