WIRELESS COMMUNICATIONS AND MOBILE COMPUTING
Wirel. Commun. Mob. Comput. 2010; **10**:101–114
Published online in Wiley InterScience
(www.interscience.wiley.com) DOI: 10.1002/wcm.901

TinyNET–a tiny network framework for TinyOS: description, implementation, and experimentation[‡]

Angelo Paolo Castellani[*,†], Paolo Casari and Michele Zorzi

Department of Information Engineering, University of Padova, Via G. Gradenigo, 6/B, 35131 Padova, Italy

Summary

In this paper we present TinyNET, a modular framework allowing development and quick integration of protocols and applications for Wireless Sensor Networks (WSNs) in TinyOS. The motivation behind TinyNET is two-fold: on one hand it allows to adopt a divide-and-conquer approach in the development of any TinyOS application; on the other hand it provides a flexible administration of network protocols. As a sample development using TinyNET, we consider an environmental monitoring application, and test it over a floor-wide WSN testbed. Data are convergecasted toward a sink node, which gathers all data collected by the sensors. Routing toward the sink is achieved by means of a hop count (HC) based algorithm. Our framework also integrates support for the 6LowPAN standard (providing, e.g., per-sensor queries and pings). Thanks to TinyNET, s these messages will make transparent use of the underlying network protocols. Also, TinyNET transparently manages the network components and related messages, allowing different applications to share the same network stack; furthermore, it translates TinyOS interfaces so that any previously developed application can be easily ported. These features make it possible to have a global vision over any application, as well as to focus on each of its separate components. Copyright © 2009 John Wiley & Sons, Ltd.

KEY WORDS: TinyOS; modular networking framework; 6LowPAN; monitoring application

1. Introduction

Wireless Sensor Networks (WSNs) have emerged as a promising paradigm for a number of applications to be implemented in the near future. These applications are growing beyond simple data collection, localization, and information retrieval services, to incorporate increasingly complex features such as smart sensing, assisted navigation, and sensory extension. It can be foreseen that many solutions by different distributors will undergo full-fledged development and find their way to the market, e.g., see Reference [1]. From a developer's point of view, it would be very convenient to create new software for WSNs based on the reuse of as many program components as possible, taken from both open-source and proprietary

[*]Correspondence to: Angelo Castellani, Department of Information Engineering, University of Padova, Via G. Gradenigo 6/B, 35131 Padova, Italy.

[†]E-mail: castellani@dei.unipd.it

[‡]Part of this work has been presented at the 5th International Wireless Communications and Mobile Computing Conference (IWCMC), Leipzig, Germany, June 2009.

repositories. However, as noted in Reference [2], very few applications are actually built based on reusable components; in fact, the most widespread approach is to implement *ad hoc*, monolithic blocks that deliver the required functionalities. From these macro-blocks, it becomes difficult to distinguish which components provide a given set of services. While this usually bears greater efficiency at execution time and a slightly smaller memory footprint, a software block-based approach can achieve comparable efficiency and footprint while bearing the further advantage of greater modularity and broader system-level view [2].

A further desirable property of applications for WSNs would be an easily manageable network interface. WSN operating systems such as TinyOS [3] give direct access to the radio transceiver commands. While giving freedom and control to the programmer as to the packets injected into the channel, this does not make any specific networking stack available, hence there are no clear interfaces that allow to easily link protocols, e.g., according to the ISO/OSI model. In other words, the programmer is not allowed to choose whether to rely on a layered network architecture or not, and even if he decides for a layered one, he has to build a custom solution that encompasses all required protocols (e.g., channel access, routing, and application). Furthermore, the absence of a modular and easily reconfigurable framework forces to reconsider the whole structure of the software in case, e.g., additional networking capabilities need to be supported. For example, reconfiguring nodes that are currently performing environmental monitoring (erratic traffic, highly energy-efficient protocols) so that the network can support fast alarm reports (locally intensive traffic, greater tolerance for energy inefficiency) requires to insert both the new alarm application and the related MAC/routing protocols; these will then have to share the same network interface in a completely customized fashion.

In this paper, we move some steps towards a solution to these problems, by introducing TinyNET, a modular framework for TinyOS that (i) makes it easier to build applications by reusing software modules; (ii) provides any protocol and application with a layered network interface that encompasses the whole stack, while still allowing cross-layer operations and exchange of parameters; (iii) allows fast reconfiguration of applications through new protocols and functionalities, that transparently become a part of the layered network stack. Our framework operates on top of TinyOS but below the user application modules, and is completely transparent (in the sense that TinyOS module binding directives are intercepted and used to place any module within the framework).

The TinyNET framework is available for download in the Tools section of the WISE-WAI website [4].

2. Related Work

With a few exceptions, most architectures proposed for WSNs are created to comply with a particular requirement, or to support a specific protocol feature. For example, the authors in Reference [5] implement energy management in WSNs by treating energy as a fundamental design primitive. Their architecture is composed of three parts, namely a user interface for specifying an energy policy, a monitoring system to control energy usage, and a management module to enforce the energy policy. The use of expressive language to specify the energy policy enables easier user interaction.

The Tenet architecture [6] has been specifically designed to support tiered architectures, where slave (low-tier) nodes are only in charge of gathering information, whereas the complexity of system-level, computationally-intensive tasks (such as data fusion) is concentrated on high-tier master nodes, which usually own a non-volatile power supply. It is worth noting that this is in line with the Router/End Node paradigm seen, e.g., in the ZigBee standard [7]. Tenet subdivides sensing tasks into tasklets, each of which specifies the sensing operation to be carried out by low-tier motes, as coordinated by masters. Tasks are flooded to all motes upon user input.

The Sensornet Protocol (SP) architecture proposed in Reference [8] aims at providing a link layer abstraction to all protocols, by means of a shared message pool (formed of data to be transmitted in packets) and a shared neighbor table, which holds a summary of neighbor information which is made available to all protocols, instead of having each protocol maintain its own. The SP approach allows to bind the standard interfaces of the higher layers of the protocol stack to the link layer; the effectiveness of this approach is explained in Reference [9]. Chameleon [10] also targets the design of a reconfigurable architecture, that allows applications to transparently adapt to different MAC, routing, and transport protocols. The key feature of Chameleon is a universal header format which is based on packet attributes rather than bit fields.

The approach chosen in Reference [2] is slightly different; the authors propose a MAC Layer Architecture (MLA), which aims at subdividing usual MAC-layer functionalities into atomic operations, so that existing as well as new protocols can be programmed based on a large library of reusable components. Each component (either hardware-dependent or -independent) is instantiated into TinyOS; when properly connected, these software blocks allow the creation of MAC protocols that are entirely analogous to those found in the literature, and yield the same performance (e.g., throughput) while bearing only a slightly larger memory footprint. An approach similar to that in Reference [2] is also found in Reference [11], where the authors propose a Communication Processing Architecture (COPRA) based on protocol processing stages and engines, i.e., components that perform basic operations and can recursively become part of larger structures to carry out more complex tasks. A survey of other ongoing projects regarding networking abstractions in TinyOS as of a few years ago can be found in Reference [12].

SensorStack [13] is a solution to provide an abstraction of communication services to the upper layers, in order to facilitate data-centric communication. It relies on an information broker based on the publish-subscribe paradigm, and aims at providing simple interfaces and efficient use of memory to share cross-layer parameters, as well as the support for notifying complex events to related protocols. Similarly, Cross-Layer Optimization Interface (CLOI) [14] provides an interface to exchange data between protocols; this interface is also implemented in the form of data structures such as message pools and neighbor tables.

Unlike the previously cited approaches, our TinyNET architecture works at a lower level. We focused more on the reusability of any software block, rather than on specializing the architecture to support a certain network task or application. In this light, the most similar approaches are shown in References [2,11]. However, no direct comparison is possible even with References [2,11], in that Reference [2] focuses on MAC protocols, while Reference [11] requires specific protocol engines and stages to encompass network functionalities. In our case, instead, the framework's main task is to let the user promptly instantiate, switch, and connect any kind of open or proprietary TinyOS-based software, with special regard to creating multiple instances of the same components and to transparently multiplexing protocols over the same interface. In this regard, it is worth noting that most of the previous architectures such as those in References [5,6] could be integrated seamlessly as part of the TinyNET framework; this also applies to the MAC components of Reference [2]. No direct comparison is therefore possible between our approach and the previous efforts at creating architectures for TinyOS. As a final note, we recall that the Contiki operating system [15] also implements an adaptive networking architecture for WSNs through the Chameleon/Rime stack [10]. However, as most applications developed to date have been programmed in TinyOS, TinyNET presents considerable advantages, as it yields equivalently solid network architecture, modularity, and extensibility to present and future TinyOS applications. Also, while Contiki has a fixed ROM occupancy of 40 kB, TinyNET and TinyOS present a much smaller footprint, as discussed in Section 4.

3. TinyNET

TinyOS is a powerful platform to build applications for WSNs, due to its limited memory consumption and to its cross-platform support; its design is based upon tiny components, whose interfaces are linked using a highly optimized C dialect nesC [16]. This paradigm has proven to be effective when building a system with shared, highly reusable components, and helps reduce the final binary image size.

The communication abstraction employed in TinyOS is the active message (AM) model [17]. The AM header is composed of 1 byte, the AM type, identifying the user-level message handler. The rest of the packet is composed of the payload to be passed on to the handling process. The AM paradigm straightforwardly allows to share the radio interface, by binding applications to a single AM type of the AM subsystem. Applications employ available interfaces to control the radio subsystem, e.g., to power it on/off and, by means of platform specific commands, read link quality indicators such as transmit (TX) power, Received Signal Strength Indicator (RSSI) and Link Quality Indicator (LQI). Directly putting an application in control of the radio subsystem is a valid approach only if the application itself is very simple; in case a more complex system should be built, a top-down approach is preferred, which requires to design the architecture and modules of the system before developing the system itself. However, network applications are usually designed as a holistic module which is tightly integrated with TinyOS; this module is implemented using hardly reusable parts and typically incorporates platform-specific code. This is also due to the structure of TinyOS itself, whose development architecture does not encourage structured modular design. Furthermore, there is no logical network architecture available for

TinyOS, which may hinder open contributions of network protocols and applications.

TinyNET is a network framework designed to help fill these gaps, as it provides modularity and easy interconnection between TinyOS and any kind of networked system built on top of it.

3.1. Architecture Description

TinyNET exploits nesC to split any networked system into two parts: the application layer and the network layer. The *application layer* is similar to TinyOS's standard developing entry point, with the additional feature that every application module represents a single, independent process in the network system. Utility interfaces have been built to perform such operations as radio state control and channel selection; in addition, the control of the radio subsystem has been centralized, so that it can intercept any control request. In turn, this common entry point facilitates the development of resolution techniques for concurrencies in radio control (e.g., through independent locks to be imposed on the radio by those applications that require exclusive access to this resource). The *network layer* is instead a novel layer encompassing those modules that require full access to the packets received and/or transmitted by the node. The network layer is a direct development entry point for new network protocols to be inserted in the framework, transparent to applications. This layer also supports ordered access of protocols to transmitted/received packets, and provides full control over the packet itself, e.g., any module can change the field structure of the packets if required. To make the differentiation among the application and network layer easier, the layers are coded within separate files, allowing system integrators to easily combine application and network components.

The development of TinyNET has posed various design challenges, mainly in order to select a minimal yet sufficient set of inter-component interfaces, which has to be clean and practical for applications, yet powerful for network modules. Moreover, significant attention has been paid to preserving support for cross-layer interactions, in that any network module can access and process information contained in other modules (this feature is natively available in TinyOS). A hardware abstraction layer has been introduced to access specific chip features, in order to provide cross-platform support and access to low-level hardware components. Moreover, the development has been carried out on top of TinyOS in a completely independent fashion, in order to favor the porting of TinyNET to newer TinyOS versions. (For better clarity, the TinyNET code has also been placed in a different folder tree with respect to the rest of TinyOS.) As shown in Figure 1, the application packets to be transmitted are taken in charge by the network layer and are scheduled across network modules.[§] After passing through all linked network modules, the packets are sent to the MAC module to be scheduled for transmission.[‖] The reception of packets takes place following the reverse order, i.e., when the MAC module signals the reception of a new packet, this packet is processed by all receive (RX) network modules (in reverse order with respect to the transmission phase, see Figure 1) and is eventually passed on to the application corresponding to the AM type of the received message.

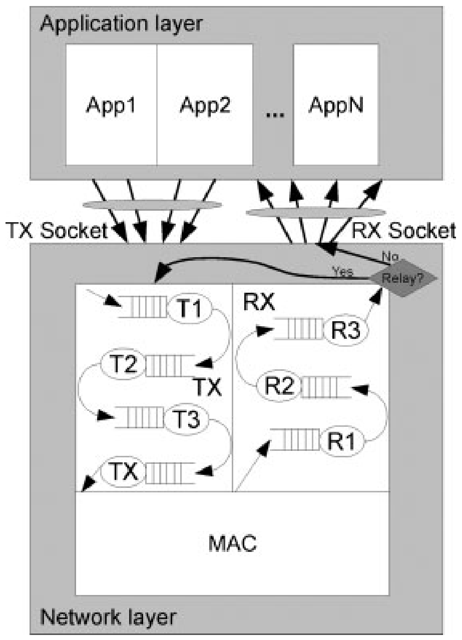

Fig. 1. A sketch of the TinyNET architecture.

3.2. Inter-Component Interfaces

Four different kinds of interfaces are required in TinyNET.

Application layer interfaces—Transmission and reception interfaces are required by applications to

[§]In the current implementation, the number of network modules has been fixed to three for simplicity.

[‖]The MAC *module* manages channel access independently of actual MAC/routing *protocols*. It is used to implement, e.g., ALOHA *vs.* CSMA.

access the framework. In general, these are an expansion of current TinyOS AMSend and Receive interfaces.

```
interface TX {
 command message_t* send(
   message_t* pkt,
   am_addr_t dst,
   uint8_t len,
   uint8_t power,
   uint8_t prio,
   bool swap );
 event void sendDone(
   message_t* pkt,
   error_t status );
}
```

The send command has been extended over the standard AMSend.send, with new per-packet TX power and scheduling priority attributes. Moreover a bool swap parameter is passed, to request a message_t* pointer to a free buffer in exchange for the message_t buffer passed for transmission. This can improve memory utilization in nodes with multiple applications that concurrently access the network subsystem. At the current stage, the Receive interface is identical to TinyOS 2.1, and has been renamed RX, in order to support later modifications.

Network layer interfaces—Network layer modules require full access to the packet, and can thus access the internal framework data structures used by scheduling components. The internal transmission buffer is defined as follows:

```
typedef struct txring_buffentry_t {
  am_addr_t src;
  message_t* pkt;
  txpkt_state_t state;
  uint8_t power;
  uint8_t prio;
  error_t error;
  uint8_t swapped;
} txring_buffentry_t;
```

where pkt is a pointer to the message_t holding the actual packet to be transmitted: swapping this pointer with a different one allows the network module to rewrite the entire packet from scratch. The variable state holds the current scheduling state of the packet, representing which network modules it has already stepped through; power is the power level at which the packet should be transmitted, prio is the scheduling priority, error stores a general purpose error code and swapped tells if the buffer space of the current packet has been swapped. Moreover, the source address of the packet is also stored in the structure (src field), to distinguish between the originator of the packet and the current relay (which is set by TinyOS). The full txring_buffentry_t* structure pointer is passed to every transmit network module which implements the ProcessTXPacket interface:

```
interface ProcessTXPacket {
 command error_t process(
   txring_buffentry_t* txbuf );
 event void processed(
   txring_buffentry_t* txbuf,
   error_t error );
}
```

Network modules handle one packet at a time: they receive the input packet through the process command, and signal back the processing completion using the processed event.

```
typedef struct rxring_buffentry {
  message_t* pkt;
  rxpkt_state_t state;
  error_t error;
  uint8_t opt;
} rxring_buffentry_t;
```

Similarly, an RX buffer structure is defined, storing the pkt message_t* pointer, the state variable of the current processing step, and an error variable holding the code of the error that occurred during packet processing. Furthermore, a persistent per-packet opt variable is provided for internal module use. A ProcessRXPacket interface is provided, analogous to the aforementioned ProcessTXPacket.

Besides the described TX/RX processing interfaces, two more specific interfaces are required to build a practical network layer: a Route interface and a TXSchedule interface.

```
 interface Route {
  command bool forward(rxring_buffentry_t* rxbuf);
  command bool isForMe(rxring_buffentry_t* rxbuf);
}
```

The `Route` interface is required to handle the delivery process of received packets. More specifically, it allows a routing module to implement custom logic to choose whether a packet should be further relayed over a multi-hop path (returning the `forward` command). Moreover, the `isForMe` command can be implemented to decide whether a packet must be delivered to the applications running on the local node.

The `TXSchedule` interface, instead, requests a packet transmission slot to the MAC module. ¶

```
interface TXSchedule {
  command error_t schedule(
    uint8_t id,
    uint16_t dst );
  event txring_buffentry_t* doTX(
    uint8_t id );
  event void TXdone(
    txring_buffentry_t* txbuf,
    error_t error );
}
```

Using this interface, the MAC module can be asked to reserve a slot for transmission of packet `id` to node `dst`. When the transmission eventually takes place, the MAC module triggers a `doTX` event, which will return the pointer to the TX buffer to be transmitted. Upon transmission end, a `TXdone` event is triggered to return the result of the operation.

Hardware abstraction interfaces—At the current stage of development only CC2420-based motes are supported, and the supplied hardware abstraction is bound to the CC2420 TinyOS implementation. When more radio chips are supported, the interface definitions and conventions will be refined. The first interface required to abstract from hardware-specific components is `HardPacket`:

```
interface HardPacket {
  command uint8_t getPower( message_t* p_msg );
  command void setPower(
    message_t* p_msg,
    uint8_t power );
  command int8_t getRssi( message_t* p_msg );
  command uint8_t getLqi( message_t* p_msg );
}
```

The per-packet TX power level, receive RSSI or LQI are extracted from the radio subsystem using this interface. As reported before, the returned values are currently interpreted as in the `CC2420Packet` module:

```
interface RadioChannel {
  command error_t set( uint8_t channel );
  event void setDone(
    uint8_t channel,
    error_t error );
  command uint radio_frequency (RF)();
}
```

The `RadioChannel.set` command allows to set the operating radio channel of the radio-frequency (RF) transceiver, according to the IEEE 802.15.4 standard; upon completion of the command, a `setDone` event is propagated. The channel currently in use can be identified by using the `get` command.

Legacy application layer interfaces—To facilitate the migration to TinyNET, a set of standard TinyOS network interfaces is provided: `AMSend`, `Receive`, `Packet`, and `AMPacket`. These interfaces are sufficient to translate former TinyOS applications to TinyNET, by instantiating TinyNET components instead of standard TinyOS components.

3.3. Technical Description

The path tree of TinyNET contains the following folders: `sys` (framework core modules); `interfaces` (interface definitions); `modules` (actual implementation of MAC, network, and application modules); `platforms` (collection of platform-specific components); `lib` (reusable components, useful to implement common network modules); `6lowpan` (porting of TinyOS's 6LowPAN implementation to TinyNET); `examples` (sample usage files demonstrating TinyNET); `install` (installation procedures and utility files).

¶This is required to support reservation-based slotted access protocols; unslotted protocols may allow access right away or according to specific rules.

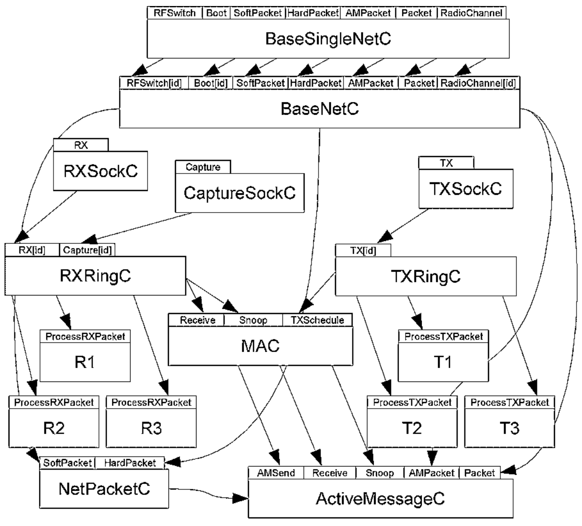

Fig. 2. Scheme of the wiring of TinyNET components.

The `sys` directory contains all the components implementing the actual framework. As shown in the wiring scheme reported in Figure 2, the `BaseSingleNetC` component is the basic module every application should instantiate to signal its own presence as part of the framework; the instantiation allows every radio-related event (power on, channel change, and radio subsystem boot) to be exposed through the offered interfaces. `BaseSingleNetC` actually instantiates `BaseNetC`, and binds it to the application with a unique `net_app_id`; the `BaseNetC` component is the network layer definition file, which is in charge of wiring all network layer components, of loading `RXRingC`, `TXRingC`, and `ActiveMessageC`, and of selecting and wiring `MacC` with the three receive and transmit modules, `R{1,2,3}` and `T{1,2,3}`, respectively.

The `RXRingC` component is in charge of passing on every packet received by `MacC` to all receive network modules, and ultimately of delivering the packet to the application, in order to re-queue it for transmission. Similarly, the `TXRingC` component is in charge of handling transmit packets to every TX network module. Furthermore, it reserves a transmission slot from the MAC module, handles the packet to that module when the slot is available, and signals back to the application when the packet has actually been transmitted. The `MacC` component has full direct access to the TinyOS radio subsystem and is in charge of every transmission and reception.

4. Proof-of-Concept Scenario

Our first experience with TinyNET focuses on a simple test aimed at measuring the framework's basic functionalities and overhead. The `BlinkToRadio` application has been ported to TinyNET as a reusable application module using native `TX`/`RX` interfaces. Therefore, `BlinkToRadio` can be loaded by simply instantiating and wiring the component in the application layer definition file. When the firmware is built using the described application and no network modules, the overhead due to the framework size can be measured by comparing the size of the binary to that of a plain `BlinkToRadio` binary, i.e., built without TinyNET. As to ROM occupancy, the use of TinyNET increased the `BlinkToRadio` size by 3.5 kB, reaching a total size of 15 kB. However, it should be noted that this overhead is fixed, and does not depend on how many applications are loaded, nor on which ones; also, it is independent of how many network modules have been wired to the framework. More specifically, it depends only on how many receive and transmit modules (see Figure 1) TinyNET is configured to handle. To make this clearer, we report in Table 3.3 the ROM footprint of the `BlinkToRadio` application. The RAM occupancy overhead, instead, depends on the scheduling queue buffer sizes as set up in configuration files, plus about 60 B of static variables allocated by the framework.

After testing TinyNET's memory footprint, we wish to experience the practical advantages yielded by usage of TinyNET, as compared to the standard TinyOS programming approach. To this end, we have built a more complex system, featuring several networking, communication, and application modules. A multi-hop environmental monitoring and querying system using 6LowPAN has been chosen in this regard, as it is complex enough to prove the advantages brought about by using the TinyNET framework. We highlight that the focus of the work was to prove that, compared to TinyOS, TinyNET allows easy and straightforward implementation of the various modules. Therefore, we

Table I. ROM occupancy for the `BlinkToRadio` application without TinyNET and with different TinyNET configurations.

Configuration	ROM Occupancy (kB)	% Increase
Without TinyNET	11.482	—
TinyNET 1 module	14.130	23
TinyNET 2 modules	14.576	27
TinyNET 3 modules	15.016	31

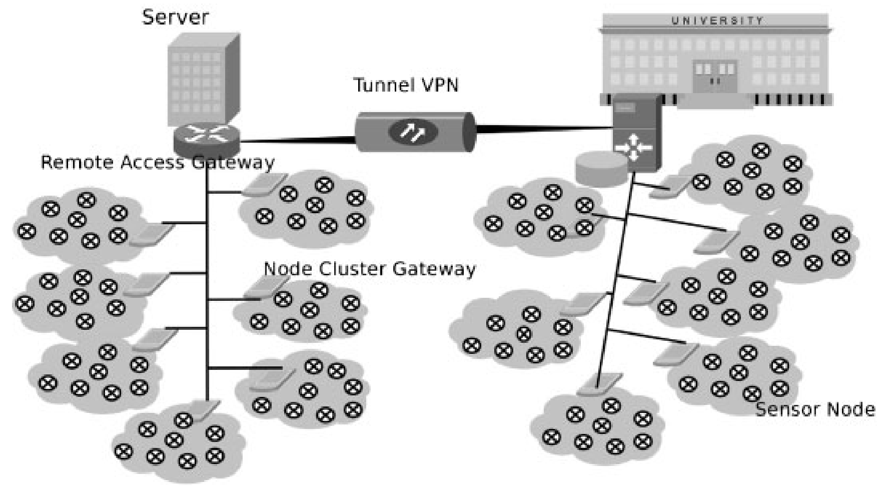

Fig. 3. Schematic architecture of the testbed deployment (see Reference [18] for more details on the WISE-WAI project).

will not be dealing with performance metrics related to the system itself.

4.1. The Testbed Setup

We start the description by introducing the testbed platform upon which we have tested the application described above. Our testbed has been designed in the context of the European SENSEI [19] and the Italian WISE-WAI [4,18] projects. The testbed serves as a flexible and reconfigurable platform to test algorithms and protocols for WSNs, and has been deployed using networking devices and solutions which allow fast communication to/from the wireless sensors nodes (e.g., for debugging or reprogramming purposes). Our network features a density of 10–20 nodes within the coverage area of any sensor; in case different (e.g., lower) densities should be needed, this number can be tuned by acting on the maximum transmit power of the nodes.

We deployed the testbed according to the hierarchical organization depicted in Figure 3, whereby all sensors are connected, *via* USB hubs, to tiny embedded computers that act as node cluster gateways (NCGs, see Figure 4). USB connections are not used for actual communication, but rather provide power supply and log data for debugging purposes. During actual operations, communications take place only through the wireless channel. For the quick deployment of applications to be executed on the nodes, USB cables can also convey new application modules. The NCGs are core elements of the network hierarchy, and interact with the nodes both in the upstream (node-to-gateway) direction, e.g., for reporting debug and log messages, and in the downstream (gateway-to-node) direction, e.g., to reprogram, reset, and power up or down the nodes. The latter functionality is provided by fully USB 2.0-compliant hubs and proves particularly useful as a sort of hard sensor reset. This is accomplished by powering off the port to which the sensor is attached. Thanks to this function, the sensors need not be manually disconnected, in case they should not respond to software reset commands. NCGs can be reached from a central server through virtual private network (VPN) connections, in order to carry out management tasks; otherwise, their presence is transparent to the

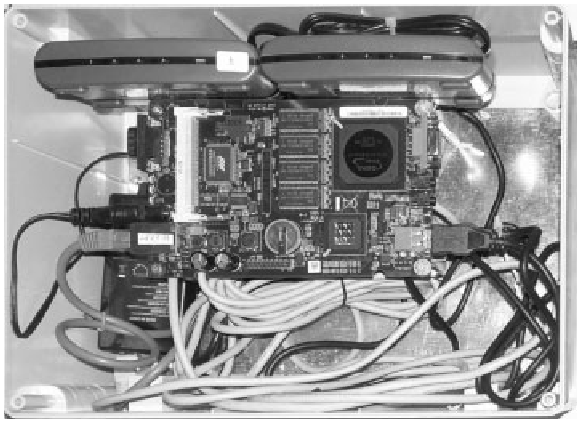

Fig. 4. An example of node cluster gateway (NCG), showing the embedded computer at the center, and the two USB 2.0-compliant hubs at the top.

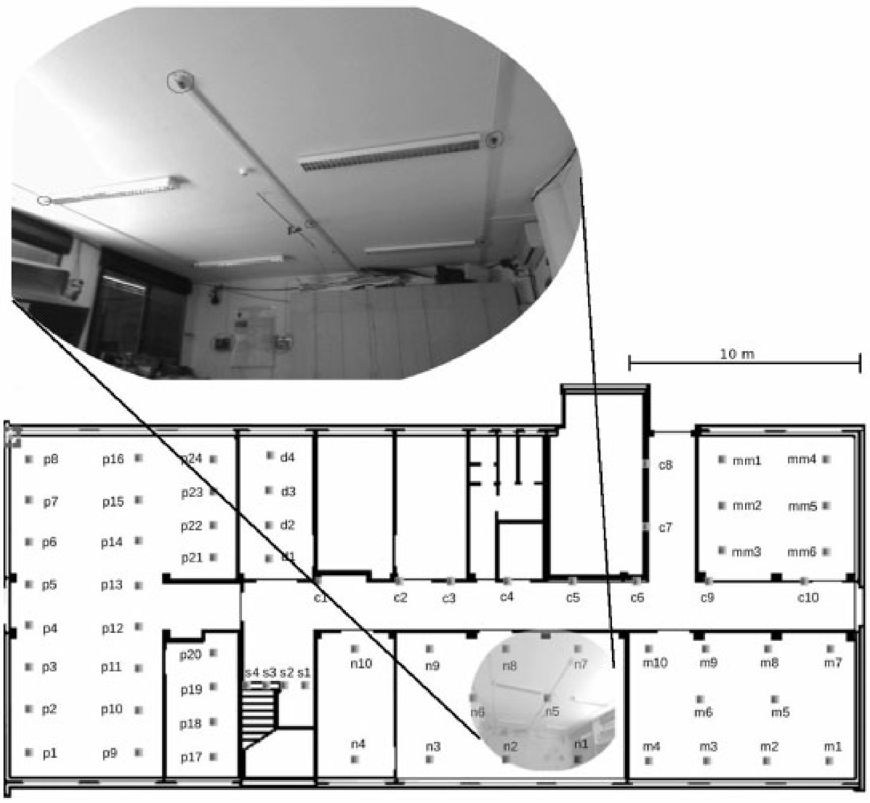

Fig. 5. Node deployment map, with a close-up on the arrangement of nodes within one of the rooms.

user, who interacts with the network as though they were directly communicating to the sensors. While command line scripts are available for this purpose, an HTTP interface has also been developed to ease node programming, as well as other management tasks (power on/off, information retrieval, and so forth).

The aforementioned architecture (server, NCGs, USB hubs, and sensors) is scalable and easy to extend; furthermore, its components can be easily replaced in case of failures. As anticipated, NCGs are key components of our network hierarchy. They are small computers of size 15 cm × 15 cm, bearing limited power supply requirements, which can be supported through the Power-over-Ethernet (PoE) standard. The wireless embedded sensors we chose for use in our testbed are the TelosB nodes [20], a widely used platform enjoying constant support and upgrades. Here we recall that the TelosB platforms are low-power ZigBee-compliant wireless nodes employing a maximum transmit power of 1 mW within the 2.4 GHz ISM band, at a maximum bit rate of 250 kbps. The nodes have been deployed within the buildings of the Department of Information Engineering at the University of Padova, Italy. Part of the testbed is shown in the map in Figure 5.

The following sections will present the components of our system architecture in more detail.

4.2. 6LowPAN/IPv6 Stack

Research on the integration between standard Internet services and WSNs, as well as the introduction of novel concepts such as the Internet of Things [21,19] bestow larger importance on protocols that allow easy connection between WSN islands and the Internet. 6LowPAN [22] plays a key role in this regard, as it is specifically designed to make any, no matter how tiny, object addressable from anywhere in the Internet through IPv6; the burden of typical IPv6 processing and header sizes, which are not optimized for the wireless channel and the limited processing power of sensor nodes, is alleviated by compressing the IPv6 header. This is achieved, e.g., by avoiding repeating patterns and useless or redundant fields, while still allowing use of the full breadth of IPv6's addressing space. Such protocols as 6LowPAN, although simple, allow packet transfers compatible with Internet communications, making WSNs become part of the world-wide network along with any other kind of connected smart object.

Currently TinyOS features a lightweight 6LowPAN implementation, which can be found in the path `lib/net/6lowpan`; this implementation provides the minimum working set of features required by IPv6 specifications: ICMPv6 Echo Request/Reply, header compression and User Datagram Protocol (UDP) socket support. The implemented features and exposed interfaces fit the requirements of our proof-of-concept application; for this reason, the 6LowPAN implementation provided along with TinyOS has been ported to TinyNET. Through the 6LowPAN component, any node can assign itself any IPv6 64-bit prefix to be added to its MAC address (`TOS_NODE_ID`); furthermore, a node can send UDP packets to any IPv6 address and listen over one or more UDP ports.

In TinyNET, 6LowPAN sits on top of the whole framework, behaving as a standard application. This way, 6LowPAN can make straightforward use of any available link layer and network protocol (e.g., routing and security). By using legacy TinyOS support interfaces, porting 6LowPAN to TinyNET has been very easy, as the only changes required involved the instantiation of some components and the setup of the proper wiring to the 6LowPAN subsystem. This is a further clue of how TinyNET straightens up development tasks when integrating objects into a more complex application.

4.3. Routing Network Module

A simple routing protocol based on hop count (HC) descent has been implemented; a node with HC equal to n always relays packets to a neighbor exhibiting HC equal to $n-1$. While this might be a suboptimal strategy [23], it is sufficiently effective and simple to serve as a proof-of-concept component. The HC information has to be renewed periodically; to this end, each node sets its own HC to an arbitrarily high value, and the sink starts a HC flooding procedure by sending an advertising packet with HC equal to 0; all nodes receiving the messages set their HC equal to 1, and choose the sink as their next hop. The procedure is recursively repeated, as every node broadcasts its HC (say n), and its neighbors set their own HC to the minimum between the current HC and the value read from the packet plus one. When the node's HC is actually updated (the packet carried a smaller value than currently held by the node), the receiver selects the packet sender as its next hop toward the sink. This is only one way to choose the next hop; other choices that lead, e.g., to some cost optimization [23] can be applied as well. In order to handle dead nodes and topology modifications, an `age` variable is associated to any chosen relay. Each time a node propagates its HC, it also increments the `age` of its relay by one. When `age` gets bigger than a preset `MAX_AGE`, the current next hop becomes outdated, and the node is required to perform a further relay choice upon reception of a HC update packet from a neighbor; in any case, the `age` of a relay is also set to zero any time a HC packet is received by that relay.

The protocol described above supports node-to-sink communication, but does not apply to sink-to-node routing, because the sink itself has no knowledge about which path to go through in order to reach the node. A simple solution to this shortcoming is to have any node, including the sink, remember which neighbor is relaying the packet sent from a specific source. By dynamically building a {relay,source} least recently used (LRU) cache table, any path can be walked in a reverse, sink-to-node direction. To accomplish the described tasks, a routing header is required, which carries information about the final destination of the packet, the chosen next hop, and the original source of the packet (which is also required in order to build the route from sink to node).

Implementing the described protocol in TinyNET requires that the network module provides three interfaces: `ProcessTXPacket`, used to build the routing header in the packets queued for transmission (in order to keep implementation simple, the routing metadata has been appended to the outgoing packet); `ProcessRXPacket`, required to extract the routing footer appended by the transmitting node; `Route`, which updates the LRU table when a packet is queued for further relaying or delivery to the application.

4.4. Environmental Monitoring and Querying Application

The application built upon the described system performs environmental monitoring and supports single node querying. The application concept is very simple; the monitoring component `ReadStoreC` periodically samples values provided by on-board sensors, and stores them into the RAM using a circular buffer of fixed size (equal to `N_PKT`). The sampling interval is fixed to `READ_INTERVAL`, and can be tuned by acting on the variable.

Asynchronous to the gatherer of sensor readings, a second component (`LocalAggregationC`) accesses the circular buffer, and stores a summary of the values on the flash memory integrated on the sensor node board. This way, the only permanent trace of past readings is kept in a compressed form, and provides useful information about reading history (which can be accessed by using proper queries), without wasting the flash memory. As a compressed representation of the readings, we chose the average value.

The networked component is named `TotalAggregationC`, and is the only module in charge of network-related operations, such as listening for incoming requests and reporting data back to the sink. The latter operation is performed periodically by all sensors, but it should be noted that the sink itself can query any specific sensor at any time, if needed. The 6LowPAN support discussed before also enables queries to be originated by any host on the Internet toward any node in the network; the converse is also true, i.e., sensor readings can be conveyed to any host on the Internet. Figure 6 shows the interfaces connecting the aggregation and data reading modules.

In order to achieve communication efficiency and scalability, the nodes progressively aggregate the sensor readings while routing packets throughout the network. As commonly done in many similar approaches [24,25], we have organized the nodes into an aggregation tree. The tree is formed in such a way that hierarchically higher nodes aggregate the readings received by the sensors from lower hierarchical levels. Hierarchical connections are devised so that nodes expected to yield correlated measurements are at the same hierarchical level and report to the same head node. Such nodes are said to form a *group*. For example, the nodes in the same room report to a node which is also inside the room, and which can decide if and how to aggregate the data coming from its children. In turn, further levels of aggregation are possible; e.g., head nodes representing groups of sensors within rooms on the same floor or wing of a building may report to another head node, which occupies a higher hierarchical position; this node receives information on a per-group basis, and can thus decide whether or not to further aggregate the readings, depending on whether the end user requires a coarse or fine data report. We note that this structure is scalable and fast to replicate. All aggregation operations applied to group readings are again delegated to the `TotalAggregationC` component.

The aforementioned application elements are connected through a custom interface `ReadData`, supporting asynchronous replies to `get` commands by means of `getDone` events. Therefore, upper layer modules propagate queries in a top-to-bottom direction whenever data is required, thereby limiting the further occupation of the sensor RAM. We recall that the averages of past readings are available on the flash memory of the node, whereas a limited amount of recent, non-averaged readings can be retrieved from the node circular buffer.

Examples of the data gathered by the application can be found in Figures 7 through 10. In particular, Figures 7 and 8, respectively, show the correlation among the time series of temperature and luminosity readings output by sensors m1 through m9, in the bottom-right room of the map in Figure 5. Figures 9 and 10, instead, show the same correlation metrics taken over the readings of sensors c1 through c5, along the corridor. From the graphs, we infer that the correlation among the luminosity levels perceived by nodes in the room (Figure 7) is very high, meaning that this metric can be aggregated into an average

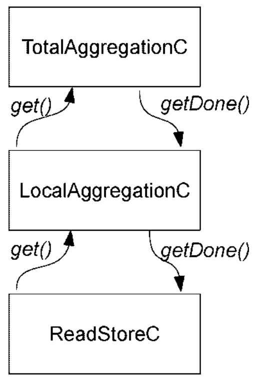

Fig. 6. Interconnection between modules of the application performing sensor reading collection and aggregation.

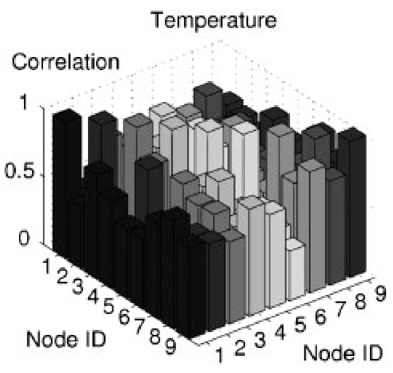

Fig. 7. Correlation among the time series of temperature readings for nodes m1 through m9 in Figure 5.

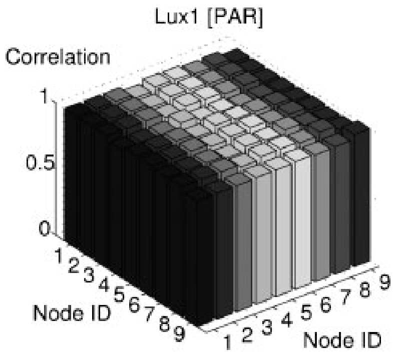

Fig. 8. Correlation among the time series of luminosity readings for nodes m1 through m9 in Figure 5.

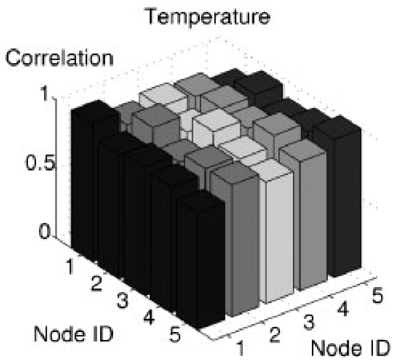

Fig. 9. Correlation among the time series of temperature readings for nodes c1 through c5 in Figure 5.

with little (if any) loss of information. The same applies to the temperature readings of nodes in the corridor (Figure 9). The reason behind the very high correlation is that the room features very uniform lighting from windows and sometimes ceiling lamps, which leads to very similar sensor readings. The same is verified for the corridor temperatures; these tend to be uniform across the corridor itself, with slightly larger deviations, which are small enough to make the temperature information amenable to be represented with an average value. The luminosity in the corridor (Figure 10) behaves differently; in this case, there are no windows, and lighting comes from rooms, doors, and side corridors as well, making the luminosity non-uniform, and more so for nodes placed farther from each other (e.g., nodes c1 and c5, which have the lowest luminosity correlation). The same applies to the temperature readings in the room (Figure 7), where this time the source of different heat levels is the equipment placed in the room itself.

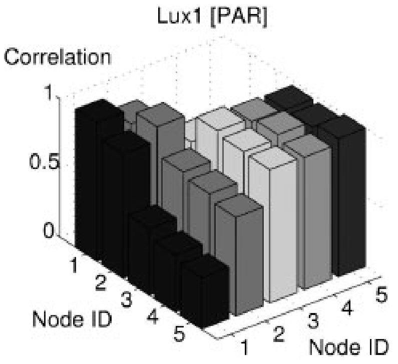

Fig. 10. Correlation among the time series of luminosity readings for nodes c1 through c5 in Figure 5.

These results are just samples of the data which can be gathered from our sensor network; however, they indeed suggest that aggregation is a viable and effective option for environmental data, making the environmental monitoring application an effective part of our proof-of-concept TinyNET application.

5. Lessons Learned and Conclusions

During the development of our system, our attention has been focused on the complexity required to develop single components and on their later integration, rather than on how to design a monolithic code encompassing the desired functionalities. We experienced that TinyNET represents a change of perspective with respect to the usual way of programming TinyOS applications, as it gives TinyOS developers a chance to follow the well-known divide-and-conquer design strategy for complex systems. In other words, TinyNET allows to subdivide the development into many simpler independent parts, each to be handled separately. Using the provided framework interfaces, the development of components was fast, allowing straightforward implementation and easy debugging; in particular, the latter is facilitated by concentrating on a single module instead of inspecting a monolithic code. Maintaining the software is also substantially easier, as each component can be easily replaced, integrated with other components, or deleted. Thanks to modularity, every component can be swapped with no further adaptations; furthermore, new features can be added on top of the already available software by creating new, separate modules, and by applying the proper wiring. TinyNET yields all these advantages with no further computational complexity and memory burden.

Future work on TinyNET includes further optimizations, including the wrapping technique of network modules, in order to provide flexibility on the number of modules, yet retaining the static component allocation and limited memory footprint (and without changing any presently available interface); we also plan to uniform the access interfaces of the 6LowPAN component to TinyNET's `TX`/`RX` interfaces, which allows simpler implementation and porting of previously developed applications.

The whole TinyNET framework is available in the 'Tools' section of the WISE-WAI project website [4].

Acknowledgement

The authors wish to thank Nicola Bressan for his key contribution to the development of the environmental monitoring application, and Claudio Lora for his help in the initial testbed deployment.

This work has been supported in part by the Cassa di Risparmio di Padova e Rovigo Foundation, Italy, under the project WISE-WAI, http://cariparo.dei.unipd.it, and by the FP7 EU project 'SENSEI, Integrating the Physical with the Digital World of the Network of the Future,' Grant Agreement Number 215923, http://www.ict-sensei.org.

References

1. CrossBow Technology. eKo Pro Series System. Available at: http://www.xbow.com/Eko/
2. Klues K, Hackmann G, Chipara O *et al*. A component-based architecture for power-efficient access control in wireless sensor networks. *Proceedings of ACM SenSys*, Sydney, Australia, 2007; 59–72.
3. TinyOS community forum. Available at: http://www.tinyos.net
4. WISE-WAI project web site. Available at: http://cariparo.dei.unipd.it
5. Jiang X, Taneja J, Ortiz J *et al*. An architecture for energy management in wireless sensor networks. *ACM SIGBED Review* 2007; **4**(3).
6. Gnawali O, Jang K, Paek J *et al*. The Tenet architecture for tiered sensor networks. *Proceedings of ACM SenSys*, Boulder, CO, 2006; 153–166.
7. The ZigBee Alliance. ZigBee Specification. Available at: http://www.zigbee.org
8. Polastre J, Hui J, Levis P *et al*. A unifying link abstraction for wireless sensor networks. *Proceedings of ACM SenSys*, San Diego, CA, 2005.
9. Culler D, Dutta P, Cheng T *et al*. Towards a sensor network architecture: lowering the waistline. *Proceedings of USENIX HotOS*, Santa Fe, NM, 2005.
10. Dunkels A, Österling F, He Z. An adaptive communication architecture for wireless sensor networks. *Proceedings of ACM SenSys*, Sidney, Australia, 2007; 335–349.
11. Karnapke R, Nolte J. COPRA—A communication processing architecture for wireless sensor networks. *Euro-Par 2006 Parallel Processing*. Springer: Berlin, 2006; 951–960.
12. Levis P, Madden S, Gay D *et al*. The emergence of networking abstractions and techniques in TinyOS. *Proceedings of USENIX NSDI*, San Francisco, CA, 2004; 1–14.
13. Kumar R, PalChaudhuri S, Ramachandran U. System support for cross-layering in sensor network stack. *Mobile Ad-hoc and Sensor Networks*. Springer: Berlin, 2006.
14. Merlin CJ, Heinzelman WB. An information-sharing architecture for wireless sensor networks. *IEEE SECON*, 2006. Demo session.
15. The Contiki Operating System. Available at: www.sics.se/contiki/.
16. Gay D, Welsh M, Levis P *et al*. The nesC language: a holistic approach to networked embedded systems. *Proceedings of ACM PLDI*, San Diego, CA, 2003.
17. Buonadonna P, Hill J, Culler D. Active message communication for tiny networked sensors 2001. Available at: http://www.tinyos.net/papers/ammote.pdf
18. Casari P, Castellani AP, Cenedese A *et al*. The "WIreless SEnsor networks for city-Wide Ambient Intelligence (WISE-WAI)" project. *MDPI Journal of Sensors* 2009; **9**(6): 4056–4082. Available at: http://www.mdpi.com/1424-8220/9/6/4056
19. The SENSEI project. Available at: http://www.ict-sensei.org
20. CrossBow Technology. The TelosB mote. Available at: http://www.xbow.com/
21. Gershenfeld N, Krikorian R, Cohen D. The Internet of things. *Scientific American* 2004; **291**(4): 76–81.
22. Mulligan G. The 6LowPAN architecture. *Proceedings of ACM ENS*, Cork, Ireland, 2007.
23. Rossi M, Rao RR, Zorzi M. Statistically assisted routing algorithms (SARA) for hop count based forwarding in wireless sensor networks. *Springer Wireless Networks Journal* 2008; **14**(1): 55–70.
24. Upadhyayula S, Gupta SKS. Spanning tree based algorithms for low latency and energy efficient data aggregation enhanced convergecast (DAC) in wireless sensor networks. *Ad Hoc Networks* 2007; **5**(5): 626–648.
25. Hong L, Hong Y, Ana L. A tree based data collection scheme for wireless sensor network. *Proceedings of IEEE ICNICONSMCL*, Morke, Mauritius, 2006.

Authors' Biographies

Angelo P. Castellani was born in Rome on 7 July 1981. He obtained his BE (Telecommunications Engineering) and ME (Telecommunications Engineering) summa cum laude, both from the University of Rome 'Sapienza' (Italy) in 2004 and 2006, respectively. During 2007 and 2008 he held a position as network consultant in a local company (People & Partners Srl). During 2008 he held a fellowship at the University of Padova (Italy), and from January 2009 he has been a Ph.D. student in Information Engineering at the University of Padova under the supervision of Michele Zorzi. His research interests include secure communications, WSN, and delay tolerant networks.

Paolo Casari was born in Ferrara, Italy, on 20 August 1980. He received both the Laurea degree (BE) in Electronics and Telecommunications Engineering (2002) and the Laurea Specialistica degree (ME) in Telecommunications Engineering (2004) summa cum laude from the University of Ferrara. From September to December 2004, he was with the same University, studying geographic protocols for WSNs. Since January 2005 he joined the School of Information Engineering at the University of Padova, Italy, where he got his Ph.D. in Information Engineering in December 2007. His main research interest is cross-layer protocol design for wireless networks through PHY/MAC/routing interactions, applied to MIMO *ad hoc* networks, WSNs and underwater acoustic networks. During the first half of 2007, he has been on leave at the Massachusetts Institute of Technology, Cambridge, MA, working on energy-efficient protocol design for underwater networks. He is currently a postdoctoral research fellow at the University of Padova, where he has been appointed Technical Project Manager for the WISE-WAI project. He is a member of the IEEE and of the ACM.

Michele Zorzi was born in Venice, Italy, on 6 December 1966. He received the Laurea Degree and the Ph.D. in Electrical Engineering from the University of Padova, Italy, in 1990 and 1994, respectively. During the Academic Year 1992/93, he was on leave at the University of California, San Diego (UCSD), attending graduate courses and doing research on multiple access in mobile radio networks. In 1993, he joined the faculty of the Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy. After spending 3 years with the Center for Wireless Communications at UCSD, in 1998 he joined the School of Engineering of the University of Ferrara, Italy, where he became a Professor in 2000. Since November 2003, he has been on the faculty at the Information Engineering Department of the University of Padova. His present research interests include performance evaluation in mobile communications systems, random access in mobile radio networks, *ad hoc* and sensor networks, energy constrained communications protocols, and broadband wireless access. Zorzi was the Editor-in-Chief of the IEEE Wireless Communication Magazine from 2003 to 2005, is currently the Editor-in-Chief of the IEEE Transcations on Communications, and serves on the Steering Committee of the IEEE Transcations on Mobile Computing, and on the Editorial Boards of the Wiley Journal of Wireless Communications and Mobile Computing and the ACM/URSI/Kluwer Journal of Wireless Networks. He was also guest editor for special issues in the IEEE Personal Communications magazine (Energy Management in Personal Communications Systems) and the IEEE Journal on Selected Areas in Communications (Multi-media Network Radios, Underwater Wireless Communication Networks). He is a Fellow of the IEEE.