TinyNET: A Tiny Network framework for TinyOS

Angelo P. Castellani, Paolo Casari and Michele Zorzi

Department of Information Engineering, University of Padova
Via G. Gradenigo, 6/B, 35131 Padova, Italy
Email: {castellani,casarip,zorzi}@dei.unipd.it

Abstract—In this paper we present TinyNET, a modular framework that allows development and quick integration of protocols and applications for Wireless Sensor Networks in TinyOS. As a sample experience of software development using TinyNET, we employ an environmental monitoring application. We organize our network tested in a converge-casting topology, where the sink gathers the data collected by all sensors. Routing toward the sink is achieved based on a hop-count based algorithm. Our framework also integrates support for the 6LowPAN standard, which becomes therefore integrated with other network components and allows, e.g., to query single sensors within the network using ping messages. Thanks to TinyNET, these messages will make transparent use of the underlying network protocols. The main advantages yielded by TinyNET are found in the modularity of network components, in specific code that transparently manages the interfaces among different modules and translates standard TinyOS interfaces into new TinyNET ones, and in the readily available infrastructure to multiplex different applications over the same network stack. This allows a global vision of the system as well as the chance to focus on the design of separate components.

I. INTRODUCTION

Wireless Sensor Networks (WSNs) have emerged as a promising paradigm for a number of smart 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, e.g., 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 [1].

From a developer’s point of view, it is very convenient to create new software based on the reuse of as many program components as possible, taken from both open-source and proprietary repositories. However, as noted in [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 services. From these macro-blocks, it becomes difficult to distinguish which components provide a given set of functionalities. While this usually bears greater efficiency at execution time and a slightly more contained 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 enabling more flexibility and giving freedom and control to the programmer as to the packets injected into the channel, this does not make any specific networking stack available to the programmer, 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 in case he decides in favor of a layered one, he has to build a custom solution that encompasses all required protocols (e.g., channel access control, 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 be multiplexed over the network interface in a completely custom fashion.

In this paper, we move a first step towards the resolution of 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).

II. 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 [4] face the challenges of 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 [5] 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
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, e.g., [4], [5] could be integrated seamlessly as
part of the TinyNET framework. This also applies to the MAC
components of [2]. Finally, it is worth noting that the Contiki
operating system [14] also implements an adaptive networking
architecture for WSNs through the Chameleon/Rime stack [9].
However, as most applications developed to date have been
programmed in TinyOS, TinyNET presents considerable ad-
vantages, 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, see Section IV.

III. TINYNET

A. Motivation

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 compo-
nents, whose interfaces are linked using a highly optimized C
dialect (nesC [15]). This paradigm has proven to be effective
when building a system with shared, highly reusable compo-
nents, and helps reduce the final binary image size.

The communication abstraction employed in TinyOS is
the Active Message (AM) model [16]. 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 Active
Message 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 indi-
cators (TX power, RSSI, 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, bearing hardly reusable parts
and usually incorporating 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 represents a drawback to open contributions
of network protocols and applications. TinyNET is a network
framework designed to help fill the gap between TinyOS itself
and any kind of networked system built on top of it.

B. 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

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 [6]. 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 SP architecture proposed in [7] 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 [8]. Chameleon [9] 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 [2] is slightly different: the authors
propose a MAC Layer Architecture (MLA), which aims at
subdividing usual MAC-layer functionalities into atomic op-
erations, 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 [2]
is also followed in [10], where the authors propose COPRA,
a communication processing architecture 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 [11].

SensorStack [12] 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 in-
formation 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) [13] 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 ar-
chitecture works at a lower level. We focused more on the
reusability of any software block, rather than of specializing
the architecture to support a certain network task or applica-
tion. In this light, the most similar approaches are [2], [10]:
however, there are also some differences, in that [2] focuses
on MAC protocols, while [10] requires specific protocol en-
gines 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

III. TINYNET

A. Motivation

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 compo-
nents, whose interfaces are linked using a highly optimized C
dialect (nesC [15]). This paradigm has proven to be effective
when building a system with shared, highly reusable compo-
nents, and helps reduce the final binary image size.

The communication abstraction employed in TinyOS is
the Active Message (AM) model [16]. 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 Active
Message 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 indi-
cators (TX power, RSSI, 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, bearing hardly reusable parts
and usually incorporating 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 represents a drawback to open contributions
of network protocols and applications. TinyNET is a network
framework designed to help fill the gap between TinyOS itself
and any kind of networked system built on top of it.

B. 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

The SP architecture proposed in [7] 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 [8]. Chameleon [9] 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 [2] is slightly different: the authors
propose a MAC Layer Architecture (MLA), which aims at
subdividing usual MAC-layer functionalities into atomic op-
erations, 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 [2]
is also followed in [10], where the authors propose COPRA,
a communication processing architecture 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 [11].

SensorStack [12] 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 in-
formation 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) [13] 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 ar-
chitecture works at a lower level. We focused more on the
reusability of any software block, rather than of specializing
the architecture to support a certain network task or applica-
tion. In this light, the most similar approaches are [2], [10]:
however, there are also some differences, in that [2] focuses
on MAC protocols, while [10] requires specific protocol en-
gines 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

operations as radio state control and channel selection; in addition, the control of the radio subsystem has been centralized, thus providing the ability to intercept any control request. In turn, this common entry point facilitates the development of resolution techniques for concurrences 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 which contains any modules that require full access to every packet 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, transparently to applications. This layer also supports ordered access of protocols to transmitted/received packets, and provides full control over the packet itself, e.g., the variable 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.

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 to 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’s, and has been renamed RX, in order to support later modifications.

C. Inter-component interfaces

Four different kinds of interfaces are required in TinyNET. Application layer interfaces—Transmission and reception interfaces are required by applications to 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 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.

1 In the current implementation, the number of network modules has been fixed to three for simplicity.
2 The MAC module manages channel access independently of actual MAC/routing protocols. It is used to implement, e.g., ALOHA vs. CSMA.
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:

```c
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.

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

Analogously, 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 any error codes faced during packet processing. Additionally a persistent per-packet opt variable is provided for internal module use. A ProcessRXPacket interface is provided, analogously to the aforementioned ProcessTXPacket.

Apart from 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.

```c
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 if a packet should be further relayed over a multihop 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.³

```c
interface TXSchedule {
    command error_t schedule(
        uint16_t dst,
        uint8_t_t id,
    error_t_t error );
    event txring_buffentry_t* doTX(
        uint8_t_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 can eventually take place, the MAC module fires a doTX event, which will return the pointer to the TX buffer to be transmitted. Upon transmission end, a TXdone event is fired to return the result of the operation.

³This is required to support reservation-based slotted access protocols: unslotted protocols may allow access right away or according to specific rules.

### 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:

```c
interface HardPacket {
    command uint8_t getPower( message_t* p_msg );
    command void setPower( message_t* p_msg, uint8_t power );
    command uint8_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:

```c
interface RadioChannel {
    command error_t set( uint8_t channel );
    event void setDone( 
        uint8_t channel,
        error_t error );
    command uint8_t get();
}
```

The RadioChannel.set command allows to set the operating radio channel of the 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 any former TinyOS network-enabled application to TinyNET, by instantiating TinyNET components instead of standard TinyOS components.

### D. Technical description

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

The sys directory contains all the components implementing the actual framework. As shown in the wiring scheme reported in Fig. 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, 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 RXXRingC, 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 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.

IV. PROOF-OF-CONCEPT SCENARIO

Our first experience with the framework focused on simple tests to measure overhead and basic functionalities. The BlinkToRadio application has been ported to TinyNET as a reusable application module using native TX/RX interfaces. Hence, 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 in comparison to a plain BlinkToRadio binary. 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 depends neither on which nor on how many applications are loaded, and is also independent of how many network modules have been wired to the framework. The RAM occupancy overhead depends on 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 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 application, networking and communication 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 TinyNET allows easy and straightforward implementation of these modules, compared to TinyOS, rather than on collecting performance metrics related to the system itself.

A. 6LowPAN/IPv6 stack

Currently TinyOS already has a 6LowPAN implementation in lib/net/6lowpan, which is directly linked to the radio subsystem. The implemented features and exposed interfaces fit our system needs, and have been ported to TinyNET.

In TinyNET, 6LowPAN sits on top of the framework, behaving as a standard application. In this way, 6LowPAN can make straightforward use of any available link-layer network protocol (e.g., routing, 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 wiring to the 6LowPAN subsystem.

B. Routing network module

A simple routing protocol based on hop count (HC) descent has been implemented: basically, 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 [17], 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 hop count (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 [17] 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 by 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.

C. Environmental monitoring and querying application

The application built upon the described system performs environmental monitoring and supports individual node queries. Nodes periodically report sensor readings to the sink, and the sink itself can query any specific sensor at any time. The 6LoWPAN support allows to query any node from any host on the Internet; also, periodic sensor readings can be routed to the 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).

V. 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 development of TinyOS applications, as it endows TinyOS developers with the 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 the modularity of the developed system, every component can be swapped with no further adaptations; additionally, 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 network modules, yet retaining 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 already developed applications.

ACKNOWLEDGMENT

The authors would like to thank Prof. Michele Rossi for his valuable help and useful suggestions.

REFERENCES