Modular Design for Smart Urban WSN Applications

Marisa Gil and Beatriz Otero

Author Affiliation(s)
Computer Architecture Department, Universitat Politècnica de Catalunya
C/ Jordi Girona 1-3, Barcelona, Spain
Email: {marisa, botero}@ac.upc.edu

Abstract

Wireless sensor networks (WSN) are platforms increasingly in use for developing data-centric as well as user-centric applications. Their low-cost energy characteristics and their simplicity in programming functions, as well as their small size make them a candidate for data collection both in a wide area and in spaces difficult to access. Despite their numerous advantages, some important constraints must be addressed in order to take advantage of these specific devices. Among these constraints, the wide variability in hardware and software components makes programmability a critical point for obtaining the desired performance while maintaining a suitable productivity time-frame.

In this paper, we propose a modular design of the higher-level WSN application algorithm, mainly based on communication and data storage. The work is based on a real smart-city project, using a new device, the Zolertia Z1 and the Contiki OS.

1. Introduction

Sensor networks are one of the possible platforms for the fourth paradigm for scientific exploration: data intensive science. Current technology in wireless protocols as well as research in technology have made it possible to develop pervasive and ubiquitous platforms all around the world, in very different geographical locations and conditions [1-2].

In hospitals, neighborhoods, forests, remote islands or deep canyons, an extended platform consisting of hundreds of small devices—“motes”—can be deployed to provide data about the environment, the people, etc. Sensors, timers or other devices connected through the network are able to provide this data.

Despite their numerous advantages, some important constraints must be addressed in order to take advantage of this specific hardware. Some of these constraints are due to the platform itself, the three most prominent of which are as follows: the small memory capacity, the necessity for low power consumption, and the communication protocols [3-4]. Communication is one of the most important actions. First, the location of the devices and the distance between them, as well as the communication protocol may be crucial in data collection. It is very important how a communication path from the data source to the final destination is established without losing messages [5]. A wide range of protocols from very simple to quite complex and powerful are offered, but not all of them are accessible in all the platforms.

Other no less important constraints are related with programmability concerns, based on the software and firmware, libraries, APIs and other such elements.

This wide variability in components and sensor programmability also makes learning in this field difficult, with the subsequent loss of opportunities for developing business and making computer science and engineering work visible [6].

In this paper, we propose a modular composition of the main parts of a WSN application capable of adapting to any low-level software or hardware modification needed in the future. Although this is not addressed in this paper, this modularity also facilitates the study and comprehension of the whole application and prevents errors and energy consumption. Our goal is to obtain a kind of “plug-in layer” where applications are able to run regardless of the specific storage characteristics, for example, or the kind of data type a sensor controller is collecting.

The application used is NoiseMap, a smart-city project for real-time monitoring of environmental sound and for making maps of noise pollution in cities [7].

In the following section we introduce our working platform: Z1, the Zolertia hardware modules and Contiki, which is the OS we use to develop WSN applications. Section 3 explains the NoiseMap application algorithm and the implementation proposed to collect data in order to generate the noise information needed. Section 4 presents our proposed methodology and shows how it works in the current NoiseMap application. Section 5 concludes with some ongoing interesting deployment that can benefit from our results. Finally, the bibliography and references are presented.
2. The Platform

The experiments and deployments of this work are based on an initial collaboration conducted jointly with the technological enterprise Zolertia, which provided professional expertise and advice on the appropriate devices. In fact, the application is based on the prize-winning design ZoundTracker and the hardware artifacts are the Zolertia flagship Z1 wireless sensor motes [8]. The OS chosen to develop the application is Contiki [9], an open source project. In the following subsections, the main characteristics of the project are described.

2.1 Z1 Mote

The Z1 node (see Fig. 1) is a device based on the well-known Tmote by Crossbow [10], but with some interesting improvements:
- 16 MHz clock speed (vs. 8MHz)
- Built-in clock factory calibration
- 92KB flash (vs. 48KB)
- Lower Power consumption (Half)
- 10KB RAM (vs. 8KB)
- 2MB external flash memory (vs. 1MB)
- Digital 3-axis accelerometer.

![Figure 1. Wireless sensor node - Zolertia Z1](image)

2.2 Contiki Overview

Contiki [9] is an event-driven open source OS, designed for microcontrollers with small amounts of memory. It works successfully in WSN, but also in more powerful platforms such as laptops. Contiki provides the necessary tools to deal with processes (to perform tasks) and files (to store information). Task activation and asynchronous communications are conducted through events. Node inter-communication is possible due to the network interface which uses the RIME protocol. RIME is the low power network stack, which provides a hierarchical set of wireless network protocols, from a simple anonymous broadcaster to a mesh network routing.

Events are the means of communicating processes with other processes or sensors with processes. There are several types of events in Contiki:
- External events (events generated by sensors).
- Internal events (events posted from a process to another process).
- Timer events (event generated when a timer expires).

Contiki provides the use of several types of storage devices in resource-constrained systems through various file systems that make use of full or part of the Contiki File System (CFS) interface.

CFS interface defines an abstract API for reading and writing. This API is modeled under the POSIX file API, and is slightly simplified. Below are some of these file system calls:
- `int cfs_open(const char *n, int f)`
- `void cfs_close(int f)`
- `int cfs_read(int f, void *b, unsigned int l)`
- `int cfs_write(int f, const void *b, unsigned int l)`

The full CFS interface functionality is offered by both the CFS-POSIX, which offers the POSIX file API provided by the host operating system, and the Coffee file system, which is mainly aimed at sensor devices and is equipped with flash memories or EEPROM. The other file systems –CFS-EPPROM, CFS-RAM and CFS-XMEM- are restricted to working with a single file only. In our work we focus on the full functionality file systems.

3. The NoiseMap Application

NoiseMap is an application for measuring the approximate noise level of a specific area in a city. An experimental infrastructure has been developed in the Campus Nord area of the UPC BarcelonaTECH to test its viability, as well as its cooperation among other networks in charge of other applications. In this work, we focus on the application itself as an example of high-level WSN programmability.

The platform consists of a set of wireless multi-radio mesh routers and a number of sensor nodes in the same order, which are capable of measuring the noise level in dB by means of an external analog device like a microphone. The main work of the nodes consists in periodically sampling the noise level of the area. When a certain number of samples are collected, they are sent to the base station node, which is responsible for collecting all

---

1 ZoundTracker was designed by Zolertia, for which it won the Barcelona SmartCity International Award in 2010.

2 By “high-level” WSN programmability we refer to the algorithm for solving the problem itself, without taking in account the specific platform or other dependent issues.
samples from all nodes in the network in order to correlate these data later (see Fig. 2). In this way, it is possible to generate noise maps for the scanned area and to identify where the sources of noise are located. It is also possible to perform an analysis over time to observe the noise level evolution over a day, a week, etc. In conclusion, the platform allows a complete analysis of the evolution of the noise level of an area, thereby providing sufficient information to make further decisions.

In the following subsections, the algorithm and the specific design decisions taken are described.

3.1. Algorithm

The NoiseMap application has two types of wireless nodes, depending on the job they are assigned: most nodes have a microphone sensor and are designed to collect environmental noise information; after a while this information is sent to the base station node. These microphone nodes are referred to as MNs. The second type of node is the base station (referred to as BS): there is only one BS and its main job is to collect data sent by MNs. Hence, it also needs to ensure that MNs remain active and operative. To this end, a “hello” packet is exchanged periodically when there is a delay in data reception.

As there are multiple MNs (there may be hundreds) collecting and sending data through the network, it would be advisable to disseminate data periodically and in small quantities, thereby making it easier to detect reception errors and enable data to be re-sent.

The algorithm that each Microphone Node (MN) executes is the following:

- Every N minute(s) the MN takes a measure and stores it until a maximum of K measures.
- Periodically, an N*K-measure packet is sent to the BS.
- When the packet sent is ACK by the BS, the related information is discarded.

And the Base Station (BS) executes the following algorithm:

- On receiving a “hello” packet, it adds the MN to the node table.
- Timestamps of the node table are updated periodically.
- If an MN has not sent a packet in a while, a poll packet is sent (data request).
- For each data packet received from an MN, BS sends an ACK packet.

Figure 3 shows a basic state diagram with all events for both kinds of wireless nodes.

3.2. Implementation and Design Decisions

Based on the hardware characteristics and the application specific deployment, as well as requirements established by the NoiseMap initial design, the following values are determined according to the type of node.

Communication Protocol

The base station (BS) node is in charge of building the mesh with all the microphone node (MN) participants. On receiving a “hello” packet from a new MN, the BS adds the node to a node table, thus keeping control of all the current MN in the mesh. The node table information is updated with a timestamp every 1s. We establish a period of P=12m to send a poll packet for data requesting.

MN nodes periodically send new sensor data to the BS. Each MN node collects groups of K samples and when a group is completed, which is every N*K minutes, a message is sent to the BS. These data packets travel through the network to the BS, but in this process some packets are lost due to many factors: environmental interference, electromagnetic noise, congestion level of the network, BS availability, etc. This makes it necessary to develop fault-tolerant applications, for example by using message-passing communication based on acknowledgement: this strategy resends the packets that have not been confirmed after a maximum number of trials, because in that case the packet is assumed to be lost or the data has not arrived correctly.

Figure 2. Overview application with locations of sensor nodes.

Figure 3. Basic state diagram of MN and BS Node.
Based on the above, the NoiseMap message-passing communication passes through three different phases (see Fig. 4) that consist of node recognition by the rest of the network, acknowledgement of data by the receiver node and, in the case of the BS node, requesting a node that has been silent for a while:

1. **Initial Handshake**: When a node starts to work and initialize the network module, it proceeds with a mechanism that announces the presence of the node. The mote sends a HELLO_BS or HELLO_MN message depending whether the node is the BS or a MN. A BS node that receives a HELLO_MN message confirms the packet reception by sending a HELLO_ACK message. MNs that receive a HELLO_BS message send an acknowledgment HELLO_MN message. In this way, the BS node is able to identify all MNs on the network and all the MNs identify the BS node in order to send data packets.

2. **Data Send**: When an MN wants to send a data packet to the BS, it sends a DATA type packet. The BS sends a DATA_ACK packet to confirm the correct reception of the data, and the sender node continues sending more packets.

3. **Poll Request**: If the BS does not receive a data packet from a specific MN for a while, it sends a POLL packet to this MN, thereby forcing the sending of data in the same way as described in case 2. Should the MN fail to respond, the BS removes it from the node table.

Figure 4. Network protocol communication phases during the application life.

### Storage Management

Once an MN node sends a data packet, the data should be stored in the MN until an ACK message arrives from the BS. In the meantime, new data packets are being sent and are waiting for the corresponding ACK messages in a FIFO list (see Fig. 5). When an ACK message is received, the first packet is deleted and memory space is freed.

Figure 5. Buffer to allocate data into the MN until acknowledge.

If these ACK messages are not received in a specific time, a memory overflow could be produced and part—perhaps all—of the stored data may be lost. In the worst case, the MN node may be permanently disabled due to its simple memory management, and the whole application may fail (these kinds of meshes could be made virtually inaccessible, so the node cannot be replaced). To prevent the MN from running out of memory, a maximum number of data groups must be defined to fit in memory. Furthermore, should this maximum number be reached, the storage of new data is prioritized and the oldest data is lost.

The basic storage unit in WSN is the memory, although current nodes provide a certain amount of flash memory as well as some type of API interface to manage space through the file-system abstraction. Data organization in files is a way of virtualizing and maintaining the implementation independent from the design decision, should we decide to change the libraries or hardware platform in the future, or to use a hybrid combination of different libraries or hardware nodes. Furthermore, the use of files makes it easy to allocate and free up space and also to maintain the associated information for each delivery: a file contains K measures, and a packet sent consists of one file.

In our case, the use of Contiki OS enables us to work with files, although this is not the case when using TinyOS or some other WSN software. At this point, the implementation decisions are highly dependent on the platform possibilities as well as the resource amount, so that any change, either in the application dimensions (number of samples to collect, time between messages, etc) or in the underlying platform, may imply modifications in the code and increase the possibility of errors.

This situation provides the motivation to develop a methodology for splitting the application

---

3. We do not consider unordered ACK arrivals, since timing considerations are not an issue in this application. Neither is it a goal or concern to be considered in our proposal.

4. As the application is collecting data constantly and over a long period of time, the loss of some data does not compromise the application behavior, although it is important to reduce the loss rate to a minimum.
code itself into two levels: a higher level, with the algorithm, and a lower level more related with implementation and specific platform concerns. This is a well-known and useful methodology used for the system level, and also for applications in more complex and powerful environments, using libraries and components.

4. Our Proposal: a Modular-based Application Design

As explained above, WSN applications are event-based and usually built as a collection of functions performing a few simple tasks: collecting sensor data, storing data temporarily in the node, sending/receiving data to or from other nodes in the mesh or the base station. When the application evolves, functionalities increase and new, possibly more complex, features are added; hardware changes are sometimes also involved. The fact that it is event-driven is also an important factor: the same event can lead to different actions depending on the state of the application. This implies modifications over all the application; in some cases this increased design may even affect the resource management consistency and compromise the application performance and behavior.

In each of the tasks performed by the program, we may distinguish the application algorithm itself (in our case the noise mapping, under the requirements established by the client; for example, the time interval to collect data sensor and the amount of measures to pack), and the code belonging to the runtime and the specific protocol as well as hardware libraries API (in our case Contiki OS and Zolertia Z1).

As regards the modular design for low-level software components, we also propose that a module decomposition in the high-level software part - the application itself - should be in charge of a basic functionality (see Fig. 6). With this modular design, and by defining function calls to interact among modules, it is possible to work with virtual functionalities while maintaining applications independent from the specific hardware or protocol characteristics.

The addition of functionalities implies adding new modules, and the modification of those already existing implies changes in the module part, maintaining the same application algorithm without dealing with changes in hardware.

The following subsections describe the methodology used and how it is applied to the three modules that form the basis of our proposed design, as well as the interface for communicating with the applications: Sensor, Communication and Storage modules.

4.1. Methodology to Make the Application Algorithm Independent from the Platform

The basic algorithm proposed for the NoiseMap application is designed to run in a specific platform, i.e. Contiki and Zolertia Z1 motes (see Section 3). The steps to separate the application behavior itself from the platform-dependent implementation are described below:

1. Clearly define the code position to collect data and send data: these positions only depend on the algorithm itself (for example, read sensor data every N minutes, or send a packet every K samples).
2. Establish the packet composition (i.e. how many measurements it has) and if some type of resending or acknowledgement policy is required: this also refers to the algorithm design for the data structure to be used (i.e. buffers).
3. Define what “collect data” means: this part will be dependent on the sensor and storage platform support. This is the dependent part for introducing the data from sensor and writing data into the correspondent storage media.
4. Define what “send data” means: this part will be dependent on the storage and network protocol used by the application depending on the platform support. This is the dependent part for sending data to the base station node and whether or not to maintain a buffer to resend data in case of network errors.

It is necessary to point out note that the dependent functionalities must work not only for our application (in this case NoiseMap), but for all other applications that “collect data” and “send data”, so a generic API interface should be defined, as shown in Fig. 7.
We focus on the composition of the three main modules, and also the most general in these frameworks: Communication, Sensor and Storage. However, other functionalities can be added and managed; for example, dividing storage into file and memory and greater accuracy in the space management (map and free functions); or on the other hand, integrating storage with communication in a single module to hide data acquisition through the network, etc.

The mechanism is for parameterizing general functions in specific functions through a table, thereby being able to go to the concrete functionality, as shown below in the corresponding subsections.

4.2. Sensor Module

Sensors have very specific information and behavior depending on whether they are analog or discrete values. The way to read sensor information is entirely dependent on the sensor itself and its driver (analog/digital data, data format, etc): to read temperature data, position data or noise data, the application should be modified completely.

The main goal of our module design for sensors is to make the application independent of the functionalities and hiding the specific sensors it uses to collect data by offering general functionalities and hiding the specific characteristics bound to a determined sensor, thereby making it easier to add new sensors without changing the algorithm. The scheme we use is the same as that used for the device driver in Unix/Linux-like systems.

The several types of sensors are provided in a structure (shown in Fig. 8), and the API between the module and the application provide the following general functions:

- **void initSensors()**: initialize every sensor that can be used.
- **SensorData getSensorData(SensorType type)**: return sensor data specified by “type” parameter.

A different sensor data is defined for each sensor used by the application, and more sensors with their basic access operations may be added to the type structure if needed (see Fig. 9).

4.3. Communication Module

The network is the main vehicle for communication in WSN (for communicating both data and the node state), and the use that the application makes of this component is critical for performance (frequency in sending messages, data types, packet size, etc.).

As in other systems, in Contiki OS there are several protocols and functionalities that may be used, ranging from very simple to very sophisticated, some of which are under development. Each protocol has different packet format and message sending parameters and synchronization characteristics (for example, including checksum, etc).

```c
typedef struct {
    SensorType type;
    union {
        #ifdef NOISE_SENS
            NoiseData noise;
        #endif
        #ifdef TEMP_SENS
            TempData temp;
        #endif
        #ifdef BATTERY_SENS
            BatteryData battery;
        #endif
        #ifdef ACCEL_SENS
            AccelData accel;
        #endif
        #ifdef HUMIDITY_SENS
            HumidityData humidity;
        #endif
    } data;
} SensorData;
```

**Figure 8.** C Union structure for representing the various sensor types that the application can use.

It is important to be able to tune the application to the best protocol and parameter values without modifying the application scheme and, if possible, without any change in the code application. Furthermore, sometimes more than one protocol is in use, depending on the node to be communicated; for example, when using a specific WSN protocol between the mesh nodes, and a more powerful and complex protocol such as IPv6 to communicate with the base laptop or other machines or nodes.

The Contiki RIME network protocol, for example, is very simple but needs a more complete specification than other more powerful protocols such as IPv6, where some fields are not necessary.

The main design goal for our communication module is the ability to change functionalities without affecting the application. In other situations, it would be interesting when doing research to test the various network protocols while preserving transparency to the application.

Furthermore, applications are aware of the protocol they are using, precisely because of the specific features (multihop, broadcasting, implicit packet ordering, etc) provided.

The proposed Communication module is in charge of managing the specific protocol(s) in use.
and hides the internals to the application, which communicates with the network by using the basic send/receive functions. The module provides the specific protocol functions corresponding to:

- unsigned short computeChecksum(Packet *my_packet)
- void mountPacket(Packet *my_packet, unsigned char *packet)
- Packet unmountPacket(unsigned char *my_array)

The underlying communication protocol used may be: RIME, RPL or IPv6, for example. When using the Communication module, the NoiseMap application is responsible for initializing it, stopping it, sending a packet (indicating the destination address, the packet type, the data, the data size and whether or not the packet is a broadcast packet) or receiving a packet. Other tasks (compute checksum, resend mechanism, what protocol must be used...) are managed by the Communication module in a transparent way. Figure 10 shows the general structure for the communication protocols RIME and IPv6.

**Figure 10.** Data structure maintained for the communication module with the two defined protocols.

### 4.4. Storage Module

The storage module hides the current specific storage management used by the node as well as its specific characteristics, such as the amount of free space. Given this separation between the main application and data storage functionality, this module allows data to be stored transparently.

In addition, this module makes it possible to divide the memory space of a node into several independent domains for storing data. While this is not the case for application under study, it would be useful when working with complex hierarchical multilevel networks [11].

The module provides the API functions corresponding to:

- void initStoreManagement(StDescriptor *stdesc, char prefix, int maxSampleNum, int maxLong): initializes the specific descriptor, the storage media, the sample amount in a chunk, buffer longitude and other information.
- int write(StDescriptor *stdesc, const void *data, int size): stores in the current chunk ‘size’ bytes starting from the beginning of ‘data’ buffer.
- int read(StDescriptor *stdesc, const void *data, int size): stores into ‘data’ buffer ‘size’ bytes starting from the read offset of the file.
- void updateReadStorage (StDescriptor *stdesc): deletes the current read chunk and prepares the next to be read.
- void updateWriteStorage (StDescriptor *stdesc): closes the current write chunk and prepares the next to be written. The Storage module automatically performs this action when the maximum samples are reached.

The data structure has the following fields:

```c
typedef struct {
    int typeStorage;
    int sizeSample;
    int longPacket;
    int longBuffer;
    int *firstACKwait;
    int *current;
    …
    *open (params);
    *write (params);
    *read (params);
    *close (params);
    buffer
} StDescriptor;
```

In our case, when working with the ContikiFS, the Storage module uses the data structure shown in Fig. 11.

**Figure 11.** Collecting Data using a buffer and file storage

### Testing the Proposal: Modify the Application to Collect Data with Different Storage Support

If the first and last nodes send the temperature every T minutes, we modify the algorithm that each Microphone Node (MN) executes as follows:

- Every T minute(s) the MN takes a measure and sends it to the BS.
- No ACK by the BS is needed, so the data is discarded as soon it is sent.
In this case, data is allocated in memory and it is not necessary to keep a buffer, so the Storage manager (stdescTem) for Temperature data is initialized and the specific write function code is shown in Fig. 12.

![Figure 12](image)

Figure 12. Information values when data is collected in memory.

5. Conclusions and Future Work

WSN applications are increasing in a variety of areas, one of the most notable being smart cities, where the mesh of deployed sensors can collect data from various sensors -humidity, light, microphone, etc.- that can change dynamically.

The kind of problems to be solved, based on events, use algorithms simpler than those in other types of programs (collecting data within a range of values, sending a message when a threshold value from the sensor is reached, etc.), and the hardware can easily be made up by adding sensors or deploying more artifacts.

However, reliance on a state to perform an action or not, together with the variability in both the underlying hardware and software, as well as the lack of standards in the protocols and interfaces, all adds complexity to the applications and makes it difficult to update and to change components and sensors, thereby causing programmability to become cumbersome and time-consuming.

In this paper we present a methodology to separate the specific application algorithm from libraries and hardware, and also to separate the different components in order to facilitate both programming and scalability. Modularity is a well-known methodology in low-level software in the embedded world; by applying this technic at application level we make programmability easy and clear. The proof of concept is conducted with a real application - a noise map - while the module functionalities proposed are the Storage, Communication and Sensor, with several examples.

Furthermore, when designing the framework for a pervasive application; that is, the location and amount of devices to be used, as well as network protocols and communication paths, it is important to evaluate and measure the energy and performance impact. While it is not addressed in this paper, future work will be focused on better policies and strategies with the purpose of enabling the element (sensor type, network protocol, etc.) in runtime to be selected, as well as inclusion of energy-aware algorithms.

6. Acknowledgment

This work has been supported by the European Commission through the HIPEAC-3 Network of Excellence (FP7/ICT-217068), the Spanish Ministry of Education (TIN2012-34557), and the Generalitat de Catalunya (2009-SGR-980).

7. References