An Architecture for Seamless Configuration, Deployment, and Management of Wireless Sensor-actuator Networks

Edgard Neto, Rui Mendes and Luís Lopes

CRACS/INESC-TEC & Departamento de Ciência de Computadores, Faculdade de Ciências,
Universidade do Porto, Porto, Portugal

Keywords: Wireless Sensor-Actuator Network, Middleware, Web-Service.

Abstract: The goal of this work is to provide (non-specialist) users with the means to seamlessly setup and monitor a Wireless Sensor-Actuator Network (WSN) without writing any code or performing subtle hardware configurations. Towards this goal, we present an architecture that allows the seamless configuration, deployment and management of applications over WSN. We explore the fact that most deployments have a common *modus operandi*: (a) simple data readers running on the nodes periodically gather and send data to sinks, and; (b) sinks process incoming data and, accordingly, issue actuation commands to the nodes. We argue that, given the knowledge of a platform's capabilities, its sensors and actuators and their respective programming interfaces, it is possible to fully automate the process of configuring, building, and deploying an application over a WSN. Similarly, monitoring and managing the deployment can be vastly simplified by using a middleware that supports user defined tasks that process data from the nodes, divide the WSN into regions, defined by simple boolean predicates over data, and eventually issue actuation commands on regions.

1 INTRODUCTION AND MOTIVATION

Nowadays, there are many hardware platforms available that can be used to deploy a Wireless Sensor-Actuator Network (WSN), e.g., SunSPOT, Arduino, Mica, Firefly, WASP-motes. These platforms are based on nodes that: have distinct hardware characteristics, e.g., distinct combinations of sensors and actuators; use distinct communication and routing protocols, e.g., ZigBee, XBee, BlueTooth, WiFi, and; are programmed using distinct programming languages, e.g., Java, nesC, C (Akyildiz et al., 2002; Lopes et al., 2009). This heterogeneity has a negative impact on the effort required to port, configure, and deploy a given application onto distinct platforms. A typical end-user wishing to deploy such an infra-structure for personal or business use would see such challenges as daunting. This makes WSN technology less appealing to the consumer market and certainly precludes its wider dissemination. Despite this heterogeneity, we argue that most end-user applications running on WSN have similar *modus operandi*, namely: (a) periodically reading values from sensors on the nodes and sending them to a sink, each node generating a data-stream for each sensed environment variable, and; (b)

executing commands on the on-board actuators of the nodes, usually triggered by off-line processing of the aforementioned data-streams. Based on this observation, we argue that it is possible to design and implement an architecture that would allow even non-specialist end-users to buy hardware/software kits and seamlessly configure, deploy and manage a WSN, without ever coding or performing subtle hardware configurations. This philosophy of *zero-programming* WSN, we believe, would go a long way in making the technology more appealing to end-users.

The following application scenario, describing the automation of a small garden greenhouse using a WSN, may further clarify what we mean by "seamless". The greenhouse owner would buy a kit composed of a few nodes, a sink node, and a software package to be installed, say, in his home computer. The nodes have temperature and humidity sensors and two pins that, in this case, will be used to turn on and off a water sprinkler, and to open or close a nearby window for ventilation. After placing the nodes strategically in the greenhouse and connecting the pins to the switches controlling the sprinklers and vents, the user installs the software in his home computer and connects the sink node to a USB port. The software first starts a web-service, used to manage the

deployment. The user then starts a builder application that, based on input such as: the kind of platform, the readings the user is interested in and their sampling frequencies - automatically builds and deploys a data collection application onto the WSN. After this, the user can manage the deployment by running a web client, also supplied with the software, from anywhere in the Internet with access to his home computer. The client connects to the web-service and allows the user to visualize the data coming from the sensors, and to manage the WSN with tasks defined with the help of a wizard, without writing code.

It is this level of seamlessness that we aim for, and towards this goal we report, in this paper: an architecture for monitoring WSN, which we call Sensor Observation aNd Actuation aRchitecture (SONAR), and; prototype implementations for the SunSPOT and Arduino platforms.

The remainder of the paper is divided as follows. The following section presents related work, describes how SONAR fits in, and its contribution. Section 3 describes the SONAR architecture. Section 4 describes the current implementation focusing on the data layer for the SunSPOT platform. Section 5 describes current work and the conclusions.

2 RELATED WORK AND CONTRIBUTION

Several other architectures have been proposed to provide, at least in part, the kind of seamless configuration, deployment and management described above. TinySOA (Avilés-López and García-Macías, 2009) is a multi-platform service-oriented architecture for WSN that can be used to monitor data from different deployments. Global Sensor Networks (Aberer et al., 2006) introduces the concept of virtual sensor to allow users to focus on XML-based high-level descriptions of deployments to describe the applications running on a WSN platform. Its zero-programming philosophy is something that we pursue in SONAR. Sens-ation (Gross et al., 2006) is a service-oriented architecture that facilitates the development of context-aware sensor-based infrastructures. It is aimed not only at WSN infrastructures but also at ubiquitous computing platforms. IrisNet (Gibbons et al., 2003) envisions a world-wide sensor web in which users, using standard web-services, can transparently make queries on data from thousands to millions of widely distributed, heterogeneous nodes. HERA (Alonso et al., 2013) is an agent-based architecture that allows the creation of a wireless sensor network using nodes with different technologies. Corona (Khoury

et al., 2010) is a distributed query processor implemented over SunSPOT WSN. MufFIN (Valente and Martins, 2011) is a generic middleware framework that allows for managing and programming Internet of Things smart objects and to provide the resulting data-streams through publish-subscribe web-services.

Most of the systems cited above provide programming frameworks on top of which users may implement their own applications for WSN and manage the deployments using a middleware. Setting up a deployment and running applications on it requires a degree of expertise from the user that is not trivial. SONAR differs fundamentally in this respect. First, we restrict the application domain to a very simple scenario: nodes get readings from sensors at given frequencies and send them to a sink; the sink sends actuation commands to nodes based on some processing of the aforementioned data. Second, we assume that the end-user of the technology will be interested in a holistic solution that allows him to configure and build an application, deploy it to a WSN, and monitor and interact with the nodes from the Internet. Most users of WSN will not have the programming skills nor the hardware expertise to perform this sequence of operations without automation. Thus, we adhere to a zero-programming philosophy, in which, except for initial configuration information (e.g., platform, sensors of interest and reading frequencies) the building of an application is automated, based on pre-compiled modules, and its deployment is transparent to the user, which can henceforth monitor the deployment with a web client application. Part of this monitoring includes a feature that is not provided by most of the above systems and certainly not in the way SONAR offers it: the possibility of defining simple (periodic) tasks, disconnected from the client and persistent, that process the data-streams generated by the data layer and issue actuation commands on behalf of the user. We find this feature essential to allow for the disconnected management of the deployment, e.g., from a client installed in a mobile phone or tablet with only occasional network connectivity.

3 ARCHITECTURE

SONAR is a fairly typical 3-layer architecture, similar for example to TinySOA (Avilés-López and García-Macías, 2009), depicted in Figure 1. The *data layer* abstracts the WSN deployments managed by the architecture. These deployments generate data-streams that are stored in a data-store in the *processing layer*. This data can be queried and processed by the *client layer* in order to extract information on the status of

the nodes in a given WSN and, as a result, issue actuation commands for *regions* of it. A region can be thought of a subset of the nodes in a deployment, and implemented as a set of MAC addresses. All the interaction between clients and the WSN deployments is done through the processing layer. The architecture allows clients to manage the deployments in a disconnected way, through management tasks. The client may forward management tasks to the processing layer that periodically query the data-store, process the query results, and eventually issue actuation commands. These tasks run within the processing layer and are persistent, in the sense that they are automatically re-activated after a crash of the middleware. Each of these layers is composed of multiple components, as required to abstract away the details of the WSN and to make data management and processing fully generic and modular.
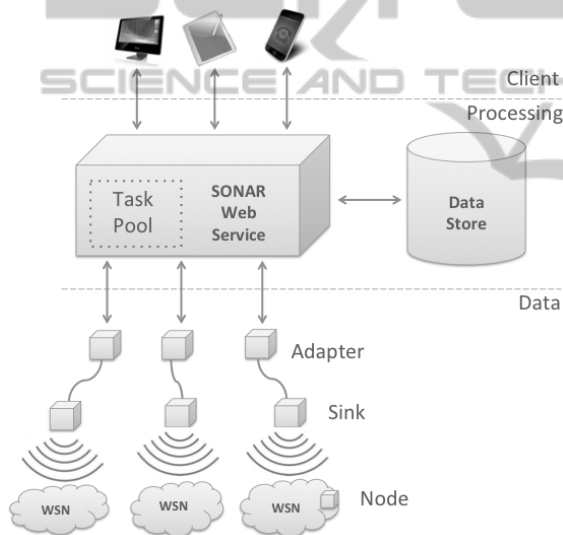

Figure 1: The SONAR architecture.

The data layer abstracts each wireless sensor-actuator platform as four components. The nodes are composed of a set of generic *readers* and an *interpreter*. The readers are simple tasks that periodically read the value of a given sensor and send it to the sink. The interpreter receives actuation commands from the sink, identifies the actuators in question and executes the commands. On the other hand, the *sink* receives the data from the nodes and forwards it to the *adapter*. The adapter is a web-service that: receives sensor data from the sink and forwards it to the processing layer, to be stored, and; receives actuation commands from the processing layer forwards them to the sink so that they are radioed to the appropriate region of the deployment. We opted to

use a web-service for the adapter so that it is possible for the sink not to be physically connected to the server running the SONAR middleware. These four components are provided for each WSN platform supported by SONAR, as pre-compiled modules and/or scripts. The interface of the adapter is given (in UML format) in Figure 2. The three methods are used to: register new deployments with the middleware - registerDeployment; forward data to the processing layer - storeData, and; receive commands from the processing layer to be forwarded to a region of the deployment - executeCommand.

Figure 2: The interface of the adapter.

The processing layer is made up of three components. The main component is a *web service* that manages the registered deployments, the data-store, and handles client requests. As we mentioned above, clients can forward management tasks to this web-service allowing for automatic and disconnected management of the WSN deployments. These tasks are executed in a *task pool* and are created and destroyed by the clients with the intervention of the web-service. The tasks periodically query the data-store, process the data and issue actuation commands to regions in the deployments, also with the intervention of the web-service. The interface for the web-service is given (in UML format) in Figure 3.

Figure 3: The interface of the SONAR web-service.

The methods in the service allow for: connecting with the data-store - mySQLConn; registering a deployment with the processing layer - registerDeployment; consulting the registered deployments and selecting one - getDeploymentList, getDeployment; storing data in the data-store - storeData; managing client

tasks - createTask, destroyTask, runTask, pauseTask, refreshTasks; finding the nodes that satisfy a given set of boolean conditions - getRegion, and; sending an actuation command to the data layer - executeCommand. The final component of the processing layer is the *data-store* that keeps information about the registered deployments and the data they are generating. The database tables are shown in Figure 4.
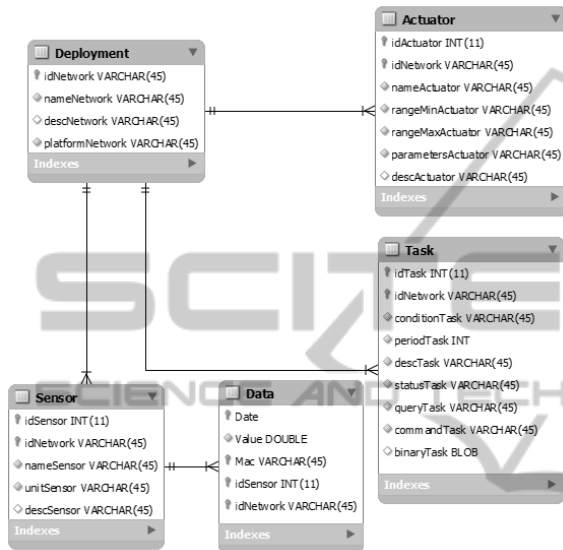

Figure 4: Database tables for the SONAR data-store.

The database tables provide a generic abstraction for a deployment. A top-level table - Deployment, keeps information about the deployment, e.g., its name, an optional description and the underlying hardware platform. It indexes three further tables that contain information about the sensors - Sensor, and actuators - Actuator, present in the nodes, and the tasks - Task, managed by the processing layer and associated with the deployment. Tasks described in the Task table may be active (running in the task pool) or inactive (on hold, waiting for a client to activate them). This table also keeps the Java byte-code associated with each task object so that, in the event of a SONAR server crash, the tasks can be restarted on recovery (field binaryTask). The actual data produced by the nodes in the deployment is stored in a single table - Data, accessed through the sensor table, that keeps the data indexed by the sensor identifier and by the time stamp. Note that we also make the simplifying assumption that all readings from sensors in the deployments are floating point values or can be converted without loss of information to this format. The components of the processing layer constitute a fully generic middleware, installed by end-users. Finally, the client layer provides web clients with a GUI that allows users to configure, build and register new deployments, visualize data, and to issue actuation commands to specific regions. These operations are mediated by the processing layer.

4 IMPLEMENTATION

In this section we describe the implementation of a first prototype for SONAR. The deployment and management of an application onto a WSN is performed in two steps. First, a *builder* application is used to configure, build and deploy an user application onto a WSN. The builder is responsible for generating the platform-specific data layer. Multiple data layers, corresponding to different deployments and platforms can be handled simultaneously by SONAR. In this prototype we provide data-layers for the SunSPOT and Arduino platforms, but we focus the following discussion on the SunSPOT implementation that uses a ZigBee mesh with simple radiogram-based communication. For the processing layer we used JAX-WS and an Apache TomCat HTTP Server for the web service, and a mySQL server to implement the data-store. As for the client layer, we provide a simple Java-based web client but it could be developed in any language that supports web services and for a variety of operating systems, namely for mobile devices, e.g., Android, iOS.

4.1 Building the Data Layer

An essential characteristic of SONAR is that the data layer for a deployment is automatically generated for the user by a Java-based builder application using simple platform and application information (e.g., hardware model, sensors to be used, sampling frequency). The descriptions of the supported platforms are stored in a XML file that is then parsed with the help of the JAXB tool and the information is used to adjust the builder's GUI to reflect the available platform functionality. The XML file includes the locations of the pre-compiled modules and scripts required for the build.

The builder application runs on the computer to which the sink device for the WSN is connected, typically to a USB port, and where the adapter web server is going to run. This must not be a full-fledged computer. Indeed, inexpensive solutions such as Gumstix modules or Raspberry Pi run Linux OS and are fully capable of running the builder, and of supporting both the sink device and the adapter web server. Once the components of the data layer: adapter, sink, and node - are built, the adapter is executed, and the code for

the nodes is deployed using whatever scripts are required by the platform, preferably using Over-The-Air (OTA) programming, a feature that, although not essential, greatly facilitates the deployment of applications. Finally, the sink application is installed and executed (Algorithm 1).

Algorithm 1: Building a platform specific data layer and deploying it.

Input: *platform* - Platform identifier
port - Hardware port for sink
config - Set of pairs (sensor,frequency)
macs - Set of mac addresses
{*libs*,*scripts*} = get_code(*platform*);
{*sink*,*node*} = build_data_layer(*libs*,*config*);
run(*adapter*,*port*);
foreach *mac* **in** *macs* \ {*sink_mac*} **do**
  node_script = get_script(*scripts*, *mac*);
  run(*node_script*,*mac*,*node*);
sink_script = get_script(*scripts*, *sink_mac*);
run(*sink_script*,*sink_mac*,*sink*);

Once the sink application starts to run it registers itself with the SONAR web server and forwards information describing the deployment, used to create an appropriate table in the data-store, and a reference for the adapter. From this point on the sink can forward data received from the nodes to the processing layer, via the adapter, and the deployment will be visible to clients that connect to the SONAR web server. The web server will also be able to send commands to nodes in the WSN, via the adapter and the sink. When a client connects to the SONAR web server and selects one of the registered deployments it receives the description of it kept in the data-store. The interface of the client is automatically adjusted to reflect the kind of information stored for that deployment, e.g., the available sensors in the node and the actuation commands it can execute, the range of the input parameters of the commands.

Data Layer Components

As described above, the data layer is divided into three components: the adapter, the sink and the node. For each platform, all sensors and actuators present in the nodes and accessible to the programmer are assigned internal integer codes and they are known as such by the application. This information is contained in XML files and passed to the builder application when the user selects the platform.

The SunSPOT nodes are programed using Java Micro Edition (Java ME) for the Squawk virtual machine (Simon et al., 2006). The node component is made up of multiple readers, as indicated by the user, and a command interpreter. The readers and the interpreter are generic pre-compiled classes. The builder simply generates a class Node that creates an instance of each object within running on a thread (Figure 5).

```
public class Node extends MIDLet {
  ...
  protected void startApp() throws ... {
    new Reader(LIGHT,5000).start();
    new Reader(TEMP,3000).start();
    new Interpreter().start();
} }
```

Figure 5: The startApp method for the MIDLet running on the nodes.

The code for the readers (Figure 6) is rather simple: a thread periodically wakes up, reads the value of the appropriate sensor, and sends it to the sink. The variable sensorID is set in the constructor (c.f. Figure 5) whereas connection and board denote, respectively, the radiogram connection used to radio the data, and an object providing access to the hardware sensors. More energy efficient data layers can be implemented using techniques such as data aggregation and/or sensor fusion, but we shall not address this point here.

```
class Reader extends Thread {
  ...
  public void run() {
    double value;
    while (true) {
      switch (sensorID) {
      case TEMP:
        value = board.getTemp().getVal();
        break;
      case LIGHT:
        value = board.getLight().getVal();
        break;
      case ACX:
        value = board.getAccelX().getVal();
        break;
      case ACY:
        value = board.getAccelY().getVal();
        break;
      case ACZ:
        value = board.getAccelZ().getVal();
        break;
      }
      conn.writeData(sensorID);
      conn.writeData(value);
      conn.sendPackage();
      Utils.sleep(period);
} } }
```

Figure 6: The code for a generic reader.

The code for the interpreter (Figure 7) is also quite simple: a thread waits for a message from the sink. When the message arrives it is unpacked and parsed in order to determine which actuator is to be activated and the corresponding parameters (e.g., which LED and which RGB combination, or which PIN and turn ON or OFF).

```
class Interpreter extends Thread {
  ...
  public void run () {
    while (true) {
      conn.receivePackage ();
      int commandCode = conn.readInt ();
      switch (commandCode) {
      case LED:
        board.setLED (conn.readInt () ,...);
        break;
      case PIN:
        board.setPIN (conn.readInt () ,...);
        break;
    } } } }
```

Figure 7: The code for a generic interpreter.

The sink receives data from the nodes and forwards it to the processing layer. It also receives actuation commands from the processing layer and forwards them to the nodes. This interaction between data- and processing layers is mediated by the adapter. In our prototype, the sink is implemented as two threads. The first thread is an instance of class DataForwarder that continuously listens for incoming messages from the nodes in the WSN, unpacks the data and forwards it to the adapter (Figure 8). The other thread is an instance of class CommandForwarder that receives pairs of the form (command,region) and radios the commands to the nodes whose MAC addresses belong to the region using unicast communication (some platforms may provide multicast communication for this purpose).

The adapter is a web service implemented in Java. It is a fully generic module and is activated in the usual way by placing the appropriate class bundle in the Apache TomCat folder installation of the computer. The implementation is quite straightforward as can be inferred from the interface (Figure 2).

The communication between the sink and the adapter is implemented through a hardware abstraction called *port* that implements a simple, low level read/write protocol on a OS serial device. This shared device must be known to the adapter when it starts. The communication protocol is platform independent and allows the code for the adapter component to be fully generic as well.

```
class DataForwarder extends Thread {
  ...
  public void run () {
    int sensorID;
    double value;
    String mac;
    while (true) {
      conn.receivePackage ();
      mac      = conn.readMAC ();
      sensorID = conn.readCode ();
      value    = conn.readValue ();
      adapter.storeData (sensorID , value ,mac);
  } } }
```

Figure 8: The code for the data forwarder class in the sink.

4.2 The Processing Layer

As we have mentioned above, the processing layer is composed of three parts: a data-store, a web service and a task pool.

The data store keeps the tables described in Figure 4 (Section 3). The mySQL server must be running before any deployment is attempted. Initially the database is empty and new tables are constructed as new deployments register themselves with the processing layer. Each sink module contains the code necessary to register the deployment with the processing layer and to build the tables in the data-base. Each deployment is uniquely identified by a key, the MAC address of the device that runs the sink module (e.g., in the SunSPOT platform this would be the MAC address of the basestation).

The web service controls the data flow in the architecture. It is implemented as a Java web service that implements the interface given in Figure 3 (Section 3). Upon initialization the web service checks for tasks marked as active in the data-base tables (Algorithm 2). For each record found, it gets the corresponding serialized object, deserializes it and adds it to the pool of (running) tasks. After this step it continues with the usual the server loop (Algorithm 2). This initialization step is important as, in case the server crashes, all the active tasks previous to the crash event, can be restarted automatically. This is of course possible because a serialized copy of each task is maintained in the data-base.

The task pool is managed through the web service interface. The tasks running on the pool can be periodic or one-shot, as specified by clients. Tasks are created by clients using the method createTask in the web service that registers the task in the appropriate table for the deployment in the data-store (Algorithm 3).

When tasks are first created they are always inactive. To activate a task the method runTask is used

---

Algorithm 2: Initializing SONAR.

init_pool();
records = db_get_active_tasks();
foreach *record* **in** *records* **do**
   bytes= get_bytes(*record*);
   task = deserialize(*bytes*);
   pool_add(*task*);
begin_server_loop();

Algorithm 3: Registering a task with SONAR.

Input: *task* - the task to be registered
task_info = get_info(*task*);
bytes = serialize(*task*);
db_store(*task_info*,*bytes*);

---

that attaches the task to one of the threads in the pool (Algorithm 4).

Algorithm 4: Running a task in SONAR.

Input: *task_info* - the task to be run
record = db_make_active(*task_info*);
bytes = get_bytes(*record*);
task = deserialize(*bytes*);
pool_add(*task*);

Typically, tasks periodically query the data store for data returned from the deployment and process it. As a result of that processing actuation commands for a region of the deployment may be issued. These are forwarded to the adapter using the executeCommand method in the web service interface. Clients may permanently remove tasks by invoking the method destroyTask with the task identifier (Algorithm 5).

Besides the task being removed from the thread pool (if it is active), the record of the task in the data-store will also be removed. Finally, a task may also be temporarily paused be a client - pauseTask, and executed again - runTask.

4.3 The Client Layer

The client layer is implemented as a Java-based GUI. When it is started it connects to the SONAR web service and requests information about all registered networks. This information is retrieved from the data-store via the web service. At this point we are not addressing privacy and/or security issues that are relevant in this context, e.g., who accesses information about a network or who can manage it.

Algorithm 5: Removing a task from SONAR.

Input: *task* - the task to be killed
pool_remove(*task*);
db_remove(*task*);

The Network Tab

The list of registered networks is provided in the "network tab" of the GUI (Figure 9). When a user selects one network from this list, the GUI is adapted to account for the different sensors and actuators supported by the network and for the current tasks associated with it. The user may then choose to visualize data, in the data tab, or manage the network, in the task tab.

ID	Name	Platform	Description
0014.4F01.0000.44C8	Greenhouse	SunSPOT	Greenhouse Temperature Control
0014.4F01.0000.7A95	Home	SunSPOT	Home Temperature and Lights

Figure 9: The network tab.

The Data Tab

In the current prototype, the "data tab" is quite simple, listing all readings that have been sent by nodes in the network. Each entry describes the sensor that reported the reading, the MAC address of the corresponding node, the reading, and the time-stamp (Figure 10). The readings are time-stamped in the sink, when the data is received from the nodes and before it is forwarded to the processing layer.

Sensor	MAC	Value	Date
Temperature	0014.4F01.0000.7EAF	17.050000000...	2013-04-02 12:08:32.0
Temperature	0014.4F01.0000.7A3E	16.350000000...	2013-04-02 12:08:31.0
Luminosity	0014.4F01.0000.7A3E	69	2013-04-02 12:08:29.0
Luminosity	0014.4F01.0000.7EAF	93	2013-04-02 12:08:29.0
Temperature	0014.4F01.0000.7EAF	17.050000000...	2013-04-02 12:08:28.0
Temperature	0014.4F01.0000.7A3E	16.150000000...	2013-04-02 12:08:27.0
Temperature	0014.4F01.0000.7EAF	16.850000000...	2013-04-02 12:08:24.0
Luminosity	0014.4F01.0000.7EAF	92	2013-04-02 12:08:23.0
Temperature	0014.4F01.0000.7A3E	16.150000000...	2013-04-02 12:08:23.0
Luminosity	0014.4F01.0000.7A3E	69	2013-04-02 12:08:22.0
Temperature	0014.4F01.0000.7A3E	16.150000000...	2013-04-02 12:08:19.0
Temperature	0014.4F01.0000.7EAF	16.850000000...	2013-04-02 12:08:19.0
Luminosity	0014.4F01.0000.7EAF	93	2013-04-02 12:08:17.0
Luminosity	0014.4F01.0000.7A3E	70	2013-04-02 12:08:16.0

Figure 10: The data visualization tab.

The Task Tab

All the management tasks associated with a network are listed in the "task tab" (Figure 11). SONAR tasks are very simple. To specify one, the user must select a

frequency associated with the task, what readings are interesting and what method is used to evaluate them (e.g., time window, average), a conditional expression on those readings, and a set of actuation commands that must be sent to all sensors for which the condition holds. Such a set of nodes forms a region. From an implementation point of view, the nodes are identified by their unique MAC addresses, so a region is just a set of MAC addresses.

Tasks are implemented internally using a small domain specific programming language. The example in Figure 11 shows one such task, for a WSN that manages a greenhouse. Every 5 minutes, it reads the last 20 minutes of temperature data, takes the average and checks whether the value is above 30 Celsius. It then sends actuation commands to all nodes with temperatures above 30 Celsius to activate the pins that switch on the sprinkler system and open the ventilation windows.
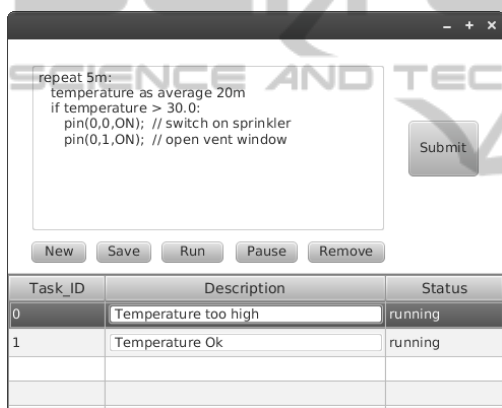

Figure 11: The task management tab.

Users do not edit tasks explicitly. Rather, a wizard is provided to guide users in the specification of tasks, without the need to write code. It is the wizard that then automatically generates the code for the tasks.

Submitted tasks are compiled by the client and an internal representation, similar to an annotated AST, is produced. Finally, a request is sent to the SONAR web server to register and start running the task, as described in Section 4.2. When the client receives an acknowledgment from this request, it adds the task to the list in the task tab.

5 ONGOING WORK AND CONCLUSIONS

Besides the SunSPOT platform, we have also implemented support for Arduino based WSN, more specifically, meshes of nodes composed of a Mega 2560

microcontroller with a XBee Series 2 module and SHT15 digital humidity and temperature sensors. We are currently developing a more powerful data visualization and processing tool for the end-user that makes use of geographical information allowing deployments to be represented over maps or satellite images. This graphical view can be used to visualize the status of each node in a mesh and to manage tasks. The goal is to make the user focus on the data and on the management of the WSN, and abstract away from low level hardware issues and raw data representations. The prototype here described shows promise in the sense that it allows the seamless building, deployment and management of WSN in both the SunSPOT and Arduino platforms. Despite this, more work on supporting more platforms and new deployments is required to assess its usability, to fully validate the approach proposed in this paper.

REFERENCES

Aberer, K., Hauswirth, M., and Salehi, A. (2006). A Middleware for Fast and Flexible Sensor Network Deployment. In *Very Large Data-Bases (VLDB'06)*, pages 1199–1202. ACM Press.

Akyildiz, I., Su, W., Sankarasubramaniam, Y., and Cayirci, E. (2002). A Survey on Sensor Networks. *IEEE Communications Magazine*, 40(8):102–114.

Alonso, R., Tapia, D., Bajo, J., and et al. (2013). Implementing a Hardware-Embedded Reactive Agents Platform Based on a Service-Oriented Architecture over Heterogeneous Wireless Sensor Networks. *Ad-Hoc Networks*, 11(1):151–166.

Avilés-López, E. and García-Macías, J. A. (2009). TinySOA: a Service-Oriented Architecture for Wireless Sensor Networks. *Service Oriented Computing and Applications*, 3(2):99–108.

Gibbons, P., Karp, B., Ke, Y., Nath, S., and Seshan, S. (2003). IrisNet: an architecture for a worldwide sensor Web. *Pervasive Computing*, 2(4):22–33.

Gross, T., Egla, T., and Marquardt, N. (2006). Sens-ation: a Service-Oriented Platform for Developing Sensor-Based Infrastructures. *International Journal of Internet Protocol Technology (IJIPT)*, 1(3):159–167.

Khoury, R., Dawborn, T., Gafurov, B., and et al. (2010). Corona: Energy-Efficient Multi-query Processing in Wireless Sensor Networks. In *Database Systems for Advanced Applications*, volume 5982 of *LNCS*, pages 416–419. Springer Berlin Heidelberg.

Lopes, L., Martins, F., and Barros, J. (2009). *Middleware for Network Eccentric and Mobile Applications*, chapter 2, pages 25–41. Springer-Verlag.

Simon, D., Cifuentes, C., Cleal, D., Daniels, J., and White, D. (2006). Java on the Bare Metal of Wireless Sensor Devices – The Squawk Java Virtual Machine. In *Virtual Execution Environments (VEE'06)*.

Valente, B. and Martins, F. (2011). A Middleware Framework for the Internet of Things. In *The Third International Conference on Advances in Future Internet (AFIN 2011)*, pages 139–144. Xpert Publishing.