2009 12th Euromicro Conference on Digital System Design / Architectures, Methods and Tools

A framework for compile-time and run-time management of non-functional aspects in WSNs nodes

Carlo Brandolese and William Fornaciari
Politecnico di Milano - DEI
Piazza Leonardo da Vinci, 32, 20133 Milano, Italy
carlo.brandolese@polimi.it, william.fornaciari@polimi.it

***Abstract*—The quality of realistic complex wireless sensor networks requires several non-functional aspects to be accounted for starting from the early phases of the application development. The most relevant aspects that need to be considered for optimization trade-offs are for sure computational efficiency (in a wide sense) and network lifetime. At lower level these aspects are measured as power/energy consumption and execution time. These are though not the only non-functional aspects to be considered: code size, memory requirement, security, reliability and other properties play often an important role. Accounting for and managing all these aspects explicitly and in an ad-hoc manner for each and every application deployed on a WSN is a time consuming and complex task. This paper proposes a portable, flexible and extendable framework for the description and management of non-functional aspects in the wireless sensor network context.**

I. Introduction

During the last decade, Wireless Sensor Network (WSN) architectures achieved a maturity sufficient to cope with realistic application scenarios [4]. The parallel advances in sensing technologies, such as Micro Electro-Mechanical Systems (MEMS), provide system designers and software developers with a rich set of opportunities to realize wireless, low power, multi-functional sensor nodes ranging from harsh environment monitoring to consumer electronics applications [3].

To better exploit such a potential, a significant research effort has been dedicated to optimize the communication protocols in order to leverage energy requirements while ensuring the capability to mange a large number of sensor nodes [9] [8]. At the beginning [2] the focus of the research in WSNs has been more on conserving power with respect to the traditional approach stressing the Quality of Service (QoS) levels. Currently, other research topics that are becoming particularly active relate to the deployment of the sensors [12] [11], the simulation of entire WSNs behavior, the high-level modeling of the WSN-based applications [16] [15] and the structuring of the application and system software running on the nodes [13] [14] to improve mainly reliability, availability and lifetime [10].

Many design and simulation environments for WSNs have been presented in literature. A list—far from being exhaustive—of the more mature and publicly available results comprises TOSSIM, NS-2, Avrora, J-Sim, SENSE, OMNeT++, VisualSense, SensorSim, EmStar, OPNET, ATEMU, Ptolemy. Most of these environments allow the designer to simulate at the network-level the behavior of the application, considering the presence of wireless communication channels. In general, they do not provide a full support to the development of software under the variability of the design constraints.

Since most of the WSN applications foresee a strong embedding of the architecture within the environment, the nodes architectures as well as the software is going to be very heterogeneous, with the result of preventing the possibility to produce in volume the hardware and to make available design and development environments with a maturity comparable to that of the EDA/embedded market segments. To overcome such a barrier to the widespread diffusion of WSNs, valuable standardization effort—especially under the IEEE 802-15.4 umbrella [6] [5]—has been put in place. Another important initiative has been the creation of the WINA Alliance to stimulate the adoption of wireless standards and the cooperation among the industry of this sectors [7].

Although the path of the industries towards designing new sensors and integrating them in wireless communicating nodes seems to be entered in a positive feedback loop, designing the application as a whole is still mostly an *handcrafted* work. In fact, the factors influencing the design of a WSN are a multitude, including fault-tolerance, cost, operating environment and sensing requirements, network topology, hardware constraints, transmission media and energy efficiency. A good survey of such problems and of the state of the art in the related research fields is reported in [2] [3].

In this paper we address the problem of managing the impact non-functional requirements to the structure of the software under the typical severe power constraint of a wireless sensing application. The goal of providing tools and methodologies to automate (partially, at least) the generation of a lightweight adaptive software, is in between many of the abovementioned topics, and for this reason it is not sometimes considered as part of a WSN standard design flow, with the result of simply increasing the final optimization/tuning effort of the designer.

978-0-7695-3782-5/09 $25.00 © 2009 IEEE
DOI 10.1109/DSD.2009.241

This problem is not new to the community of researchers working in the distributed software arena, but many of the existing or proposed solutions require such heavy software infrastrucures (in terms of energy, memory and computational power) to prevent their adoption in most of the battery-operated architectures and applications.

In this paper we present a complete overview of the approach we are proposing. The description begins with a modeling of the application in term of configurable components, whose behavior and characteristics can be tuned statically or dynamically, according to the degree of flexibility imposed by the application itself. Based on this application model, a framework for the development of WSN applications is presented. The main goal is to provide the designer with an abstraction for the management of the non-functional aspects of an application. Such framework takes into account all the aspects of the design and software structuring, including *data structures*, *libraries* and *toolhain* for generating the code fulfilling the static and dynamic application requirements.

The practical usage of the proposed methodology and framework, is described in two sections. A simple example describes how the software needs to be structured to allow the static and dynamic configuration of the node's software behavior. Moreover, experimental data coming from the porting of the framework to real operating systems of wireless node [18],[17] reveals that the overhead of the management framework can be made negligible compared to the benefits emerging even with the very simple application scenarios reported.

A section reporting a summary of the achievements of the work we presented, together with an outline of the future research directions, will close the paper.

II. Formal application model

The basic assumption necessary to allow non-functional aspects management at software level is that the application (including the underlying operating system) is – to some extent – statically and dynamically configurable. This general idea can be restated in the following terms.

1) The application is a collection of interacting and independent *components*, some of which are subject to management and are the only ones that we consider. The application can thus be described as $\mathcal{A} = \{c_0, c_1, \ldots, c_n\}$.
2) Each component provides different implementations with different non-functional characteristics but performing exactly the same functionality. We call these implementations *modes*. A component is thus described as a set of implementations, that is $c_i = \{c_{i,0}, c_{i,1}, \ldots, c_{i,m_i-1}\}$.
3) Each mode of each component is characterized in terms of a predefined set of non-functional metrics, whose specific nature has no impact on the structure of the model. The value of the k-th metric for component mode $c_{i,j}$ is indicated in the following as $M[c_{i,j}, k]$. It is worth noting that the model and the framework supports both *static* metrics, i.e. metrics computed once at compile-time and constant throughout execution, and *dynamic* metrics whose value may change at tun-time. The footprint of a component is a typical static metric while its execution time is a dynamic one.
4) One or more non-functional managers are developed and integrated with the application. A manager is responsible for changing the mode in use for each component based on the status of the node and of the environment and using the metric information of the different modes of each component.

Assuming that these requirements are satisfied, then a *static configuration* of the application is defined by means of an $n \times m_{max}$ *usage matrix* $\mathbf{U}$ such that $\mathbf{U}(i,j) = 1$ if mode j of component i is included (i.e. compiled and linked) in the application and $\mathbf{U}(i,j) = 0$ otherwise. Since different components may have a different number of modes, the matrix $\mathbf{U}$ is larger than needed because m_{max} is the maximum number of modes provided for a certain component.

Choices concerning the static configuration can be supported by general multi-objective optimization algorithms using values of the static metrics and average values of the dynamic ones.

Once the application has been configured, only a subset of the modes of each component are available at run-time. Without loss of generality we can assume that for a generic component c_i only the first q_i modes have been selected. In this case the dynamic description of a component can be expressed as: $c_i = \{c_{i,0}, c_{i,1}, \ldots, c_{i,q_i-1}\}$ where $q_i \leq m_i$. The *dynamic configuration* of the application is described as an n-element *call vector* $\mathbf{C}[\mathbf{i}]$ indicating in every moment the mode to be used for each component c_i. In particular $C[i] = s$ indicates that, at the current time, mode s of component c_i, that is $c_{i,s}$ will be used.

The dynamic management of an application requires defining some *cost functions*, based on the values of the metrics of the different components, and minimizing it. Both *local* (.i.e. at the level of a single component) and *global* (i.e. at the level of the whole application) optimizations may be performed at run-time. A typical form of the local cost function may be, for example:

$$M[c_{i,j}] = \sum_{\forall k} \alpha_k \cdot M[c_{i,j}, k] \tag{1}$$

where the summation extends over all possible values of k, that is over all available metrics. A local optimum for component c_i is calculated by finding a mode with index $jopt$ such that:

$$M[c_{i,jopt}] = \min_{\forall j} M[c_{i,j}] \tag{2}$$

Global optimization, on the other hand, can be formulated either a the combination of independently optimized local cost functions or as a global multi-objective problem. Regardless of it mathematical formulation, the result of the global optimization indicates the best mode $jopt_i$ to be used for each component c_i, that is one specific assignment of the C vector:

$$C_{opt} = [jopt_0, jopt_1, \ldots, jopt_n] \tag{3}$$

It is worth noting that locally optimizing a set of components usually requires much less computational power (and thus execution time and power consumption) that globally optimizing it and for this reason a global optimization algorithm can hardly be implemented on node. A possible trade-off consists in classifying the expected operating conditions, or *states*, $\mathcal{S} = \{S_0, S_1, \ldots, \S_t\}$ of a node and precalculating at compile time the optimal configuration for each of these operating points. This approach results in a set of configurations $\mathcal{C} = \{C_0, C_1, \ldots, C_t\}$. According to this approach dynamic optimization consists in determining the current operating condition and consequently apply the corresponding configuration.

III. Framework

The goal of the proposed framework is to provide an abstraction for the management of the non-functional aspects of an application deployed on a wireless sensor network. Such a framework is composed of *data structures*, *libraries* and *tools*. The core data structures are an implementation of the mathematical model discussed in the previous section. Libraries implement basic functionalities of the abstraction layer that are neither application-specific nor system-dependent. The toolchain, on the other hand, supports the configuration and code-generation phases and integrates seamlessly into typical development flows. The main issues considered in the design and development of the framework are the following:

1) The framework implements both *static configuration* (i.e. compile-time) and *dynamic management* (i.e. run-time) mechanisms.
2) The libraries and the code generated by the toolchain have been designed to be easily portable across different node platforms, i.e. different hardware and different operating systems.
3) The libraries and the generated code have been structured to have minimal impact on the target application, both in terms of code and data sizes and in terms of execution time overheads.

The following sections provide further details concerning the constituents of the proposed infrastructure.

A. Data structures

According to the model proposed in this paper and described in the previous section, each component must be characterized by a complete description of each operating mode. The implementation of the framework requires that each component, say `foo`, is described by three files: `foo.h` and `foo.c` store the declarations and the implementations of the different modes, respectively, while file `foo.xml` collects all the non-functional information about he component in the form of an XML tree. It is worth noting that XML is a very convenient language to describe and organize the non-functional data we are considering but, on the other hand, it is a very verbose language that requires rather complex parsers to be read.

The framework thus uses XML descriptions at compile-time only. When the static configuration is completed, the required portions of the XML descriptions of the selected component are translated into highly optimized static C data structures by means of code generation. While the details of XML descriptions are omitted here for the sake of conciseness, it is useful to shortly consider the organization of the data structure that is used at run-time.

Each component is described by both a numeric identified and by a name in the form of a digest. This second identification mechanism is necessary to support dynamic loading both entire components and single modes (modes also, in fact, are described by an identifier and a digest). Furthermore the structure contains an indication of how many modes are available for the function and which one is currently in use.

Access to the functions implementing the top level component and its different modes is made through pointers. It is worth noting that while mode pointers are inherently necessary to implement mode management, the component pointer is only needed to support dynamic loading.

To avoid dynamic memory allocation and, most importantly, indirect memory access through pointers and linked lists, all the data structures are statically declared and their sizes are thus fixed. In particular the maximum number of modes that component may have and the number (and type) of metrics to be used must be decided and set during static configuration, prior to code generation and compilation.

B. Libraries and architecture

To achieve portability the non-functional aspects API are implemented almost completely in pure C and make a very limited use of the underlying operating system resources. An important role in achieving this goal is played by the combination of static data structures and code generation at compile time. Figure 1 shows the typical hardware/software architecture of a node, in which the non-functional API lays between the topmost operating system layer and the application. At the same level of the application there are one or more non-functional managers that are not, in fact, part of the framework but rather custom components.

The non-functional API layer is composed of two different groups of functions: the metric & mode management functions and managers support functions. The former group provides an interface for accessing the component descriptions (modes and metrics) while the second is used to perform all common operations involving the manager. The interaction among the

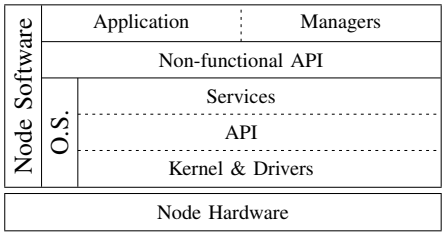

Fig. 1. General node architecture

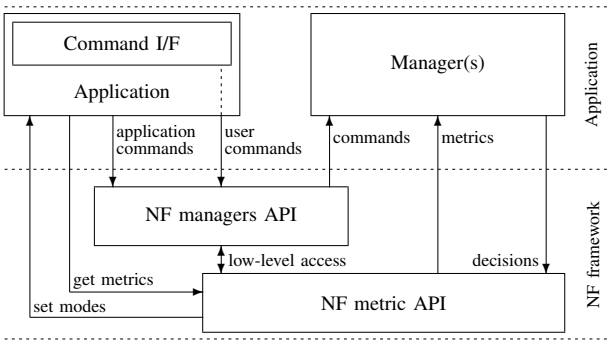

Fig. 2. Interactions through layers

application, the managers and these two groups of programming interfaces is depicted in Figure 2.

While executing some function may change their non-functional behavior, leading to *metric updates*, saved in the descriptors through the NF metrics API. When invoked – either periodically or implicitly – a manager accesses the saved metrics and makes decisions on the modes to be used, which are communicated to the application, again through the NF metrics API. The NF manager API have two different goals: on one hand they implement the set-up and configuration procedures for the managers and, on the other hand they implement a communication channel from the application (either operating autonomously or under the control of a remote user) to the manager

Figure 2 shows the interactions between the non-functional layer and the application layer. As already mentioned, component management can assume one of the following forms, leading to different modes of interactions between the blocks depicted in this figure.

- *Explicit management.* The mode to be used for each component is fixed explicitly at compile-time and cannot change during execution. The same component can be invoked in different modes from different points in the application. Explicit management does not require a run-time manager. It is worth noting that this type of management can always coexist with the remaining.
- *Implicit management.* The manager is invoked prior to each call to one of the components subject to management to decide about the mode to use for that call.
- *Periodic management.* The manager is invoked periodically to decide about the best modes to use for all the components under its control for the next period of time. This type of management can further be classified as autonomous or non-autonomous. In the first case all decisions are taken autonomously by the manager without any influence from other nodes or from a supervisor, while in the second case the node must have an interface for exchanging commands and information with other nodes and a manager can thus be controlled remotely.

It is worth noting that, in principle, implicit and periodic management may coexist and influence different sets of functions.

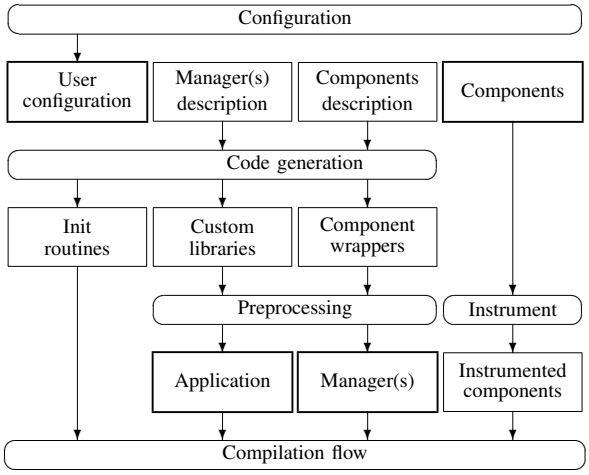

Fig. 3. Non-functional infrastructure design flow

The current implementation, though, disallows such a mixed approach, primarily for performance reasons.

C. Toolchain

The toolchain necessary to implement the whole infrastructure must accomplish several different tasks. It must, in particular, support the application designer in the static configuration phase, generate the application specific and system dependent portions of the libraries and facilitate the development of new and ad-hoc non-functional aspect managers.

All these different tasks have been designed with two goals in mind. On one hand, and primarily, the toolchain should minimize the impact of the infrastructure itself on the final application and, on the other hand, should be flexible and easy to integrate in the customary software development flow and toolchain. The design flow, schematically described in Figure 3 provides the following functionality.

1) *Configuration.* A graphical interface and the underlying tools allow the developer to choose the components (both at the application level and at the operating system level, if available) to be included in the application. At the present stage o development static configuration is completely driven by user decision but further developments are being studied to include a design exploration phase to support such decisions. The output ("User Configuration", in the figure) of this first phase is the list of components to be used, the list of modes to be included for each component, a set of application-level parameter that define general properties of the infrastructure (i.e. the accuracy to be used to represent metric values, the support for dynamic loading and management of new components, the number and type of managers to be used at run time, and so on) and the list and configuration parameters of each manager.
2) *Code generation.* To limit the complexity of the infrastructure several choices defined in the configuration phase result in ah-hoc generated code rather than pa-

rameters of a more general—and thus less efficient—programming interface. The main problem solved by this approach is the need of providing flexibility while avoiding the usage of complex, dynamic data structures. Also critical is the implementation of the portion of the API that is responsible for reading and writing the metric values for each mode and each component. A general and flexible solution would in fact require complex arithmetic operations to manage and combine values with accuracies ranging from 4 to 32 bit, as needed. The output of the code generation phase is a set of C source files implementing the customized function for metric access and a set of macro definitions used to conditionally compile the relevant portions of each component and to assign unique numerical identifiers to components to optimize data structure accesses ("Init routines" and "Custom libraries"). For each component included in the application that will be subject to dynamic management (either periodic or implicit), a wrapper ("Component wrappers") is also generated to limit the amount of modifications that are needed to adapt a preexisting component to the non-functional management mechanism.

3) *Preprocessing.* During this phase, the standard C preprocessor is used to specialize—thanks to the generated macros— the user code according to the configuration settings. The input to this phase are the generated macros and the original code of the application ("Application") and the managers ("Manager(s)"), while the output is the code resulting from the macro expansion. This new code constitute the actual implementation of the application and the managers.
4) *Instrumentation.* The source code of each component is fed to the instrumentation toolset that implements the dynamic metrics update mechanism by means of library function calls injected into the original code. Output of this phase is the final components' code.

The preprocessed application and manager source code and the source code of the components are eventually input to a traditional compilation toolchain.

IV. A SIMPLE EXAMPLE

This section presents a simple example that has the main goal of describing how the non-functional management mechanism can be used to build a new application or to adapt an existing one. The description of the steps that follow assumes that the programmer has already developed three different implementations of each of the two component and has also implemented a manager. Under this assumptions, the steps to follow are described below.

A. Components adaptation

Consider first a component called `foo` for which three implementations called `foo_a`, `foo_b` and `foo_c` are available. The first step consist in assembling the three implementation in a single file called `foo.c` and structured as:

```
#if NF_MODE_USE(foo,a) > 0
int foo_a( ... ) { ... }
#endif
#if NF_MODE_USE(foo,b) > 0
int foo_b( ... ) { ... }
#endif
#if NF_MODE_USE(foo,c) > 0
int foo_c( ... ) { ... }
#endif
```

The second step consists in building the description of the component by means of an XML file whose structure is:

```
<component name="foo" ...>
  <prototype> ... </prototype>
  <mode id="1" function="foo_a"
    <metric id="2" value="12" />
    <metric id="6" value="370" />
    ...
  </mode>
  <mode id="2" function="foo_b"
  </mode>
  ...
</component>
```

In this description there are three sections: the component information, provided in the form of attributes of the `component` tag, the `prototype` section that describes the interface of the component, that is the signature of the function implementing it (note that all modes must have the same prototype), and one or more `mode` sections collecting information about the different modes (mainly the name of the function implementing it and its metric characterization).

The application uses these components to implement the desired functionality and we suppose that its code completely contained in the body of the `main_thread()` function. This function invokes the two components by referring to them using their name, i.e. `foo()` and `bar()`. Note that neither of these function are defined in the component implementation.

To introduce the management mechanism for the two component it is sufficient to modify its original code by wrapping function calls within a macro, as in:

```
int main_thread() {
  NF_MANAGE(foo)();
  while( ... ) {
    NF_MANAGE(foo)();
    NF_MANAGE(bar)();
  }
}
```

The last step require for implementing non-functional management consists in configuring the application. In this example we assume that the manager has already been implemented and needs only to be referenced in the configuration file. A configuration file is structured as an XML description of the relevant parameter for the application. Omitting some important details that are more implementation specific and do not change the fundamental aspects, we can consider as an example the following configuration file.

```
<configuration name="myapp" ...>
  <prototype> ... </prototype>
  <component name="foo" usemodes="2"
             management="static" ...\>
  <component name="bar" usemodes="1,3"
             management="periodic" ...\>
  <manager name="mymgr" period="10" ...\>
  ...
</component>
```

The output of the toolchain are mainly three files¿ The first is `myapp_init.c` and contains the definition and initialization of the data structures for run-time management of the selected components (`bar` only, in the example) and the creation of a thread periodically executing the manager called `mymgr`. The second file is the wrapper that allows calling the different implementations of the component `bar`:

```
static int nf_fptr_id = -1;
void bar( void ) {
  void (*fptr)(void);
  if( nf_fptr_id == -1 )
    nf_fptr_id = nf_get_fidx( bar );
  fptr = nf_get_mode_pointer( nf_fptr_id );
  (fptr)();
}
```

Since functions can be loaded at run-time it is not possible to statically associate a numeric identifier to the different components and use it as an index into the array of component-description data structures. For this reason, the first time the wrapper is invoked, it looks up the pointer of the function into the array (`nf_get_fidx()`) and saves its index in a static variable. Based on this index the wrapper accesses the data structures to determine the current mode and to retrieve the corresponding function pointer (`fptr`). Finally, the wrapper calls the desired mode. It is worth noting that since the management policy selected is periodic, there are no explicit calls to the manager. Finally, the preprocessing phase, through a sequence of expansions of the macros generated from the XML configuration, produces the customized version of the main application thread function, that is:

```
int main_thread() {
  foo_b();
  while( ... ) {
    foo_b();
    bar();
  }
}
```

It can be noticed that the uses of component `foo`, being configured statically, have been replaced by explicit calls of the selected mode, that is `foo_b`. On the other hand, component `bar` is managed periodically, thus it is called through the wrapper function `bar()` generated on purpose.

V. Experimental results

A. Portability and size

The proposed framework has been completely implemented and has been ported over several operating systems. The libraries have been originally designed for Linux then ported onto μcLinux, MantisOS, FreeRTOS, and a custom operating system called WASP-OS, developed within the IST Project WASP. The porting activity required only very small modifications of the code (less than 1% of the LOC) in the function for manager thread creation. The support for dynamic loading of components is not available in all the considered operating systems: an ad-hoc dynamic loader is being developed for MantisOS only.

The footprint of the non-functional library, in its most complete form is on average 5.7KB bytes and can be suitably downsized (excluding some ancillary features) to approximately 3.5KB. A version of the library that does not support dynamic component loading has also been developed and its footprint is 1.0KB. To these footprints, we must add the size of the wrappers that need to be created for dynamically managed components. This size is almost constant and is approximately 400 bytes per component with dynamic loading enabled and less 250 bytes when disabled.

Finally, the size of the data structures needed for component description and management is proportional to the number of components and modes available and can be approximated by the following relation:

$$D_{size} = n \cdot (18 + m_{max} \cdot (12 + \sum_{i=0}^{v} -1 \cdot p_i)) \qquad (4)$$

where n is the number of components, m_{max} the maximum number of allowed modes per component, v the number of metrics and p_i the precision used to represent metric number i and can vary between 4 bit (i.e. 0.5 bytes) and 4 bytes. Reasonable complexity applications have shown an overall data structure size ranging from 0.5 KB to less than 2.0 KB.

B. Simulation environment

As stated earlier, several complex metrics can be adopted to measure the "performance" of the nodes of a system. Power and energy consumption are for sure the most relevant in a wide range of applications. Of course, optimizing power consumption alone will lead to the identification of reasonable implementations at compile-time and the need of a more general, multi-objective non-functional aspects management framework would significantly be reduced.

Also in such a simplified scenario, explicit coding is required to account, for example, of the changing environmental conditions e to properly react, using the most appropriate component mode for one or more tasks. The availability of ready-made, general-purpose managers significantly reduces the development effort and improves the application quality by reducing hard-to-detect programming errors.

To quantitatively show the impact of the proposed framework on typical scenarios, we have performed a number of simulation experiments. Using power consumption and execution time figures reported in [1] and combining them with other qualitative measures we have constructed a multi-linear cost function of the form given by Equation (1).

The simulator operates in the following way. First of all, it requires some system parameters such as the running time

(T), an indication of how quickly is the external environment is changing (τ_s), the period of the manager (τ_m), the number of components(n), the number of modes (m_{max}) and a metric characterization of each component. To exclude all functional aspects from the simulation – such as those necessary to actually implement the modes and the manager – we assume that at every moment in time one of the modes of each component is the "best" candidate for execution, that is, has the lowest cost. Let $c_{i,jopt}$ be that mode. We then describe the relevant properties of each component c_i by the triple:

$$\langle p_i, B_i, b_i \rangle \tag{5}$$

where p_i is the probability that the function implementing component c_i is called at the current time, and is a parameter used to simulate realistic execution profiles; B_i is the cost of the "best" function mode, i.e. the cost of the most appropriate mode for component c_i at the current time and with the current environmental conditions. This cost is given by

$$B_i = M[c_{i,jopt}] \tag{6}$$

Finally, the parameter R_i is the average cost associated to the "remaining" modes of c_i, i.e. all the modes except $c_{i,jopt}$.

As an example, assuming that component c_0 has four modes, the triple $\langle 1.2, 2.2, 6.5 \rangle$ has the following meaning. Component c_0 will be approximately invoked 1.2 times out of 100 simulation loops; whenever the manager will choose the most suited mode for the current environmental conditions, the cost of its execution will be 2.2 (in arbitrary adimensional units), while if the manager will not make the correct decision (for example simply because it is not invoked at that moment) the cost of the function – corresponding to the average cost of the three remaining modes – will be 6.5.

C. Scenarios

This section collects the results obtained with the aforementioned simulator with different components sets and different system configurations.

1) Local vs. remote data analysis: In this first simple scenario we suppose that only one component c_0 is under the control of the manager and it has four modes corresponding to different trade-offs between local computation (low power consumption but limited accuracy) and remote computation (high power consumption due to more data being transmitted but high accuracy thanks to computation on a remote workstation). The simulator has been set-up in 15 different configurations (3 invocation frequencies p_0, times five system time constants τ_s). Table I summarizes the configurations. Furthermore, we assume that the cost of both the implicit and periodic managers is 1, i.e. comparable with that of the best component mode.

The following results have been obtained. In case we use no management and select the mode to be use at compile time, we only have one probability out of four (i.e. the available function modes) that the selected function fits the environmental conditions every time it is called. This value, though probably pessimistic, is used as a reference and its relative overall average cost is conventionally set to 1.0.

TABLE I
CONFIGURATION FOR THE FIRST SCENARIO

PARAMETER		VALUE
T_{max}	Simulation time	10^5
τ_s	System time constant	$\{101, 505, 1515, 5150, 50217\}$
τ_m	Manager period	10^3
F_{max}	Functions	1
M_{max}	Function modes	4
p_0	Call frequency of f_0	$\{1\%, 31\%81\%\}$
g_0	Good cost of f_0	1
b_0	Bad cost of f_0	20

TABLE II
RESULTS FOR THE FIRST SCENARIO

τ_m/τ_s	$p_0 = 1\%$	$p_0 = 31\%$	$p_0 = 81\%$
9.909	0.976	0.976	0.968
2.100	0.487	0.458	0.456
0.667	0.289	0.281	0.277
0.194	0.246	0.239	0.239
0.020	0.192	0.207	0.193

When using the implicit manager, we are sure that the good mode will be used at every invocation. The resulting overall cost does not depend on the dynamic of the environment is almost constant and equal to 0.129, regardless of the function call frequency. Using an implicit manager, thus, improves the cost function by a factor of approximately 8.

The last case concerns the periodic manager. Table II summarizes the results obtained varying the function probability and the system time constant (expressed as the ratio τ_m/τ_s. In this case also the dependence on the function call frequency is limited.

It is worth noting that when the environment changes very slowly, i.e. when $\tau_m/\tau_s = 0.02$, the periodic manager guesses the right function to use with a high probability (approximately 96%) and thus the relative costs (0.192, 0.207 and 0.193) tend to mimic the situation in which the implicit manager is used. This last manager, in fact, always determines the best function to be used.

2) Architecture without sleep modes: The second scenario simulated describes a system in which seven components are managed and the architecture of the node does not allow deep-sleep modes. In this case the difference in the cost of the good and bad cases only depends on the algorithmic choices and is typically limited. The functions for this experiment are characterized by the parameters summarized in Table III.

The simulations have been carried out with five different τ_m/τ_s ratios, and Table IV summarizes the results for the three types of management. Values are relative to the costs without management.

The result indicates clearly that when several components with limited spreads between the best and worst case costs are managed, a periodic solution is more suitable than an implicit

TABLE III
CONFIGURATION FOR THE SECOND SCENARIO

COMPONENT	DESCRIPTION	COMPONENT	DESCRIPTION
f_0:	$\langle 25, 1, 20\rangle$	f_1:	$\langle 34, 10, 11\rangle$
f_2:	$\langle 46, 15, 18\rangle$	f_3:	$\langle 35, 55, 95\rangle$
f_4:	$\langle 34, 45, 55\rangle$	f_5:	$\langle 35, 35, 36\rangle$
f_6:	$\langle 35, 8, 12\rangle$		

TABLE IV
RESULTS FOR THE SECONDS SCENARIO

	MANAGEMENT		
τ_m/τ_s	NONE	IMPLICIT	PERIODIC
9.909	1.000	0.957	1.007
2.680	1.000	0.957	0.920
0.174	1.000	0.957	0.785
0.056	1.000	0.957	0.777
0.023	1.000	0.957	0.774

TABLE V
RESULTS FOR THE THIRD SCENARIO

	MANAGEMENT		
τ_m/τ_s	NONE	IMPLICIT	PERIODIC
9.909	1.000	0.209	1.017
2.680	1.000	0.209	0.739
0.174	1.000	0.209	0.311
0.056	1.000	0.209	0.283
0.023	1.000	0.209	0.276

one. This is especially true when the environment changes slowly with respect to the manager invocation period, which is often the case.

3) Architecture with sleep modes: When sleep modes are available (both for the microprocessor core and for the critical devices such as the radio) and easily controllable via software, the differences between the costs of the good cases and those of the bad ones become more significant. We have considered again seven functions that on average show a good/bad cost ratio of approximately 1/10.

Since switching between sleep modes takes time and energy, we have considered a scenario in which the functions exploiting the power modes have low frequencies, around 5%.

The results obtained are summarized in Table V and show that in this case periodic management is appropriate only when the environment changes very slowly.

VI. CONCLUDING REMARKS

The paper proposes a framework to develop a non-functional aspects management mechanism for wireless sensor network applications.

The framework has been designed and implemented to satisfy several requirements. In particular the proposed infrastructure has proved to have small impact on the application itself – in terms of code footprint, memory requirements and computational overhead -, has shown a very good portability over five popular operating systems, and, finally has proved that the overall "quality" of the application significantly benefits from dynamic non-functional aspects management.

While on one hand optimizations are being studied to further reduce the impact of the infrastructure on the application, more experiments are being designed and carried out to replace simulation results with actual data measurement.

Though the current framework is at a very early stage of development, validation and testing, the results obtained in simulation and some preliminary measurement suggest that the propose approach is viable.

ACKNOWLEDGMENTS

This work has been partially supported by the IST WASP [18] and ARTDECO [17] projects.

REFERENCES

[1] M. Carsana, "Characterization of the power consumption of a wireless sensor network node," *MSc Thesis*, Politecnico di Milano, Italy, 2008.
[2] I.F. Akyildiz, S. Weilian, Y. Sankarasubramaniam and E. Cayirci, "A survey on sensor networks," *IEEE Comm. Mag.*, vol. 40, n. 8, pp. 102–114, Aug. 2002.
[3] I.F. Akyildiz, S. Weilian, Y. Sankarasubramaniam and E. Cayirci, "Wirelss Sensor Networks: A Survey Revisited," *Computer Networks (Elsevier) Journal*. 2005.
[4] MoteIV Nodes, `www.moteiv.com`,`www.xbow.com`
[5] Zigbee, `www.zigbee.com`
[6] IEEE 802.15.4, "Wireless Medium Access Contgrol (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANS)," Oct. 2003.
[7] Wireless Industrial Networking Allianca (WINA), `www.wina.org`
[8] E. Shih et al., "Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks," *ACM Mobicom 2001*, pp. 272–286, Rome, Italy, Jul. 2001.
[9] M. Sichitiu and C. Veerarittiphan, "Simple, Accurate Time Synchronization for Wireless Sensor Networks," *In Proc. of Wireless Communications and Networking,* (WCNC'03), Vol. 2, pp. 16–20, Mar. 2003.
[10] A. Sinha, A. Chandrakasan, "Dynamic Power Management in Wireless Sensor Networks," *IEEE Design and Test of Computers,* Mar. 2001.
[11] S. Dhillon, K. Chakrabarty and S. Iyngar, "Sensor placement for grid coverage under imprecise detections," *In Proc. of Int. Conf. on Information Fusion,* pp. 1581–1587, July 2002.
[12] A. Howard et al., "An incremental self-deployment algorithm for mobile sensor networks," *Autonomous Robots – Special Issue on Intelligent Embedded Systems,* Vol. 13(2), 2002, pp. 113–126.
[13] S. Madden, M. Franklin, J. Hellerstain and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems,* Vol. 30(1), 2005, pp. 122–173.
[14] K. Aberer, M. Hauswirth and A. Salehi, "A middleware for fast and flexible sensor deployment," *In Proc. of Int. Conf. on VLDB,* pp. 1199–1202, 2005.
[15] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis and S. Shenker, "The design and implementation of a declarative sensor network system," *Proc. of SenSys'07,* Sydney, Australia, Nov. 2007.
[16] M. Zoumboulakis, G. Roussos, "Escalation: Complex Event Detection in Wireless Sensor Networks," *LNCS – Smart Sensing and Context,* Vol. 4793/2007, Oct. 2007.
[17] ARTDECO, "Adaptive Infrastructure for Decentralized Organiza- tions," *Min. of National Research,* 2006–2009, `artdeco.elet.polimi.it`.
[18] WASP, "Wirelessly Accessible Sensor Populations," IST project, `www.wasp-project.org`.