Energy Aware Software Evolution for Wireless Sensor Networks

Danny Hughes*, Eduardo Cañete*, Wilfried Daniels*, Gowri Sankar R*, James Meneghello*, Nelson Matthys*, Jef Maerien*, Sam Michiels*, Christophe Huygens*, Wouter Joosen*, Maarten Wijnants§, Wim Lamotte§, Erik Hulsmans†, Bart Lannoo‡ and Ingrid Moerman‡

.

*iMinds-DistriNet, KU Leuven, Heverlee, B-3001, Belgium.
Email: {firstname.lastname}@cs.kuleuven.be
†OneAccess, Heverlee, B-3001, Belgium.
Email: Erik.Hulsmans@oneaccess-net.com
§iMinds-EDM, Hasselt University, Diepenbeek, B-3590, Belgium.
Email: {firstname.lastname}@cs.kuleuven.be
‡iMinds-IBCN, Ghent University, Gent, Belgium, B-9000, Belgium.
Email: {firstname.lastname}@intec.ugent.be

***Abstract*—Wireless Sensor Networks (WSNs) are subject to high levels of dynamism arising from changing environmental conditions and application requirements. Reconfiguration allows software functionality to be optimized for current environmental conditions and supports software evolution to meet variable application requirements. Contemporary software modularization approaches for WSNs allow for software evolution at various granularities; from monolithic re-flashing of OS and application functionality, through replacement of complete applications, to the reconfiguration of individual software components. As the nodes that compose a WSN must typically operate for long periods on a single battery charge, estimating the energy cost of software evolution is critical. This paper contributes a generic model for calculating the energy cost of the reconfiguration in WSN. We have embedded this model in the LooCI middleware, resulting in the first energy aware reconfigurable component model for sensor networks. We evaluate our approach using two real-world WSN applications and find that (i.) our model accurately predicts the energy cost of reconfiguration and (ii.) component-based reconfiguration has a high initial cost, but provides energy savings during software evolution.**

I. Introduction

Wireless Sensor Networks (WSNs) are composed of embedded computers, or 'motes', equipped with low power radios and sensors that are capable of detecting phenomena in the physical world. Motes must typically execute for long periods on limited batteries, making energy conservation a critical issue. Software reconfiguration is also a critical issue for WSN due to two factors. Firstly, due to the high cost of deploying a WSN, sensor networks are increasingly required to support multiple applications throughout their lifespan [1], [2]. Secondly, the resource constraints of WSN necessitate optimal configuration of software to suit environmental conditions. As environmental conditions change, reconfiguration is required to maintain optimal operation. As WSNs are often deployed at scale in inaccessible or dangerous locations such as flood plains [3], remote reconfiguration is required.

Software evolution approaches for WSN can be categorized by their granularity. Monolithic approaches allow for reconfiguration through replacement of the entire software image running on each mote, including both OS and application functionality. This approach is exemplified by TinyOS [4]. Application-based approaches, such as Contiki [5] and Squawk [6] separate OS functionality from application functionality, allowing for the replacement of complete application images. Component-based approaches such as OpenCOM [7], Figaro [8] and LooCI [9] allow for the replacement of individual components within an application at runtime.

While research from the field of WSN has resulted in a variety of software evolution approaches, current techniques do not quantify the energy cost of reconfiguration, which makes it difficult for application developers to reason over reconfiguration options. Furthermore, the relative costs of each reconfiguration approach have not yet been evaluated in real world WSN scenarios.

The first contribution of this paper is a generic model for calculating the energy cost of software evolution in WSN. We validate this model using two real world industrial case studies. Our results show that, our energy model has an average accuracy of 98% for embedded motes [10] and 89% for more capable Java-based mote platforms [11]. The second contribution of this paper is the incorporation of this energy model into the LooCI middleware [9], resulting in the first energy aware, runtime reconfigurable component model. The final contribution of this paper is an evaluation of the energy cost of component-based and application-based reconfiguration in two real-world applications. Our results indicate that component-based development consumes more energy during initial configuration but offers distinct advantages for *software evolution*.

The remainder of this paper is structured as follows: Section II provides background on reconfiguration and software evolution in WSNs. Section III describes the case-study scenarios. Section IV describes a generic model for calculating the energy cost of the reconfiguration in WSNs. Section V models the LooCI middleware running on two heterogeneous mote platforms. Section VI evaluates an implementation of the energy model for LooCI. Section VII discusses related research. Finally, Section VIII concludes and discusses directions for future work.

II. Background

The need for reconfiguration of software functionality has been evident since the inception of the field of WSN research. For example, in the pioneering Great Duck Island experiment [12], researchers used a WSN to monitor the nesting behavior of birds. The application used a heterogeneous architecture wherein battery powered motes were deployed to monitor nests, smart phones were used for on-site data gathering and a PC-class gateway provided a satellite uplink. In the early stages, low-power radios were used for local communication within sensor patches and 802.11b was used to relay data to the WSN gateway. This architecture was later *reconfigured* to use low-power links for all communication. The authors highlighted the need for a range of reconfiguration techniques ranging from re-parameterization to *evolution* of deployed software functionality.

The SICS factory surveillance system described in [13] used a WSN to monitor conditions in a factory complex. The application used a homogeneous architecture composed of 25 battery powered motes and was implemented as a set of Contiki modules [5]. The authors highlight the need for autonomic reconfiguration as well as the need for *evolution* of deployed application functionality to meet changing application requirements.

The GridStix flood monitoring system [3] was deployed on two rivers in the UK to provide early warning of floods. The system used a heterogeneous architecture composed of Embedded Linux boards powered by batteries and solar panels to monitor conditions on a 1KM stretch of river. Ad-hoc 802.11b and Bluetooth were used to implement spanning-tree data collection protocols, which relayed data to a single GSM uplink. All application functionality was implemented using the OpenCOM [7] component model. In [3] the authors demonstrate that component-based reconfiguration can be used to optimize application behavior to meet changing environmental conditions.

The Cambridge badger monitoring experiment [14] used a WSN to monitor the behavior of badgers in a nature reserve. The application used a heterogeneous architecture with RFID tags deployed on the collars of badgers, static RFID detection stations and battery-powered motes that monitored the local microclimate. Application functionality was implemented as a set of Contiki modules [5]. After deployment, software was subject to *evolution* based upon input from domain experts and to accommodate changes in the hardware platform.

Considering the example applications discussed above, it can be seen that the need for software evolution is inherent in WSN applications, from the pioneering early experiments [12] to contemporary WSNs [14]. Software evolution serves two general purposes: (i.) to evolve application functionality to meet changing requirements and (ii.) to optimize application functionality to suit changing environmental conditions.

A. Requirements for WSN Reconfiguration

Reconfiguration in WSN imposes some specific requirements in terms of energy awareness, remote reconfiguration and granularity of reconfiguration:

Energy Awareness: In all of the scenarios discussed above, some or all of the motes must last for long periods on a single battery charge, and thus have a finite energy budget. *Requirement 1: software evolution approaches must accurately predict the energy cost of reconfiguration actions.*

Remote Reconfiguration: As WSNs are typically deployed at large scale, the manual reconfiguration of thousands of nodes is generally too costly to be feasible. Furthermore, WSNs are frequently deployed in inaccessible or dangerous locations. This necessitates support for remote (re)configuration of software functionality. *Requirement 2: it should be possible to enact reconfiguration remotely.*

Granularity of Reconfiguration: The reconfigurations discussed in Section II, call for not only monolithic re-flashing of application functionality [12], but also fine-grained evolution and tailoring of the components that compose an application [3]. *Requirement 3: support is required for component-based reconfiguration of application functionality.*

III. Case Studies

We introduce two case-study applications: smart parking and waste bin tampering. These case-study applications were provided by OneAccess, an international company with research and development facilities located in the Flanders region of Belgium. OneAccess have developed a common hardware/software architecture that is used to support both case studies. This architecture features a common network environment and four tiers of functionality:

1) **Network environment:** IPv6 is supported end-to-end using the Routing Protocol for low-power and Lossy networks (RPL) [15] and all motes use industry standard IEEE 802.15.4 radios.
2) **Sensor tier:** based on a custom embedded mote platform that offers a 16MHz ARM Cortex CPU, 16KB RAM and 128KB flash memory. *Sensor motes* monitor their local environment, execute simple analysis algorithms on sensor data and send the results to the routing tier. To conserve resources, the sensor tier does not participate in multi-hop routing, which is provided by the *routing tier*. Sensors motes are powered by two AA batteries with an average initial charge of 2400mAh. This battery pack must last for a minimum of 5 years without replacement to ensure economic viability.
3) **Router tier:** based on the same embedded mote platform as the sensor tier, *router motes* offer dedicated multi-hop routing functionality, relaying data between

the sensor tier and the gateway tier . The routers form a tree-like topology with the gateway as root. The routers are permanently powered from the electricity grid.

4) **Gateway tier:** is based on a more powerful Alix embedded PC platform, which offers a 500MHz AMD Geode CPU, 256MB RAM, 802.15.4 and 802.11g networking. The gateway runs embedded Linux and Java SE. The gateway is powered directly from the electricity grid. The gateway serves as bridge between the WSN running RPL over 802.15.4 and community wireless networks running standard IP over 802.11g.
5) **Back-end tier:** is comprised of powerful servers on high-speed connections, which gather data from all sensors and expose processed results to users via a web interface.

In order to maximize return on investment, OneAccess use a common routing, gateway and back-end infrastructure to support multiple sensing applications. The *sensor mote* tier is then extended as needed by specific sensing applications. Section III-A describes a 'smart parking' application and III-B describes a 'waste bin tampering' application.

A. Smart Parking Application

The smart parking application makes parking more efficient in crowded city centers, by using motes to monitor free parking spaces and communicating this information to drivers via their smart phones.

Sensor motes are embedded in a durable casing known as a 'speeddisk', which is commonly deployed on roads to slow down traffic. The speed disk is securely attached to the concrete at street level and is capable of withstanding the pressure of cars driving over it. Each mote is equipped with a magnetometer, which measures the local magnetic field on three axes and an Infra-Red (IR) distance sensor. The magnetometer is polled once per second and a simple algorithm is applied to detect whether a car has arrived or left. The battery level of the mote is polled once per minute. Where a state transition is detected (i.e. a car has arrived or left), an update message containing a boolean value representing parking space availability and the current battery level is forwarded to the routing tier and from there to the gateway. The back end system then publishes updated parking space availability.

B. Waste Bin Tampering Application

The waste bin tampering application monitors when public waste bins are opened and closed in order to detect tampering (the bins should only be opened by staff). Where tampering is detected, cleaning crews are dispatched to the waste bin to fix the problem.

For this application, sensor motes are extended with a magnet activated reed switch which detects when the bin is opened or closed. When a hardware interrupt is generated by the reed switch, the time that the bin was opened is stored in flash memory. This log of open/close times is sent to the gateway every 24 hours.

As with the smart parking system, the waste bin tampering system has been extensively simulated and tested at small scale at the OneAccess site. However, further reconfiguration is expected to be necessary when the system is deployed.

IV. Energy Aware Reconfiguration Model

In Section IV.A, we first present a generic model for calculating the energy consumption of remote software reconfiguration. In Section IV.B we then introduce a methodology for parameterizing this model based upon the energy characteristics of specific mote platforms.

A. Generic Modeling Approach

The energy cost of a reconfiguration can be broken down into the cost of inspecting a configuration using *introspection*, the cost of *deploying* new functionality and the cost of *configuring* the deployed functionality. The total energy consumption of any reconfiguration can therefore be defined as:

$$E_R = E_D + E_I + E_C. \tag{1}$$

Where E_D is the energy consumption of the component deployment, E_I is the energy consumption of introspection calls and E_C is the energy consumption of configuration calls. We expect that the energy consumed during the deployment of new software functionality will have a positive linear relationship to the size of the software that is being deployed. E_D may thus be calculated using the following linear regression equation:

$$E_D = \sum_{i=1}^{n}((CS_i \times \beta_1) + \beta_0). \tag{2}$$

Where β_0 is the minimum cost of a deployment operation, β_1 defines the relationship between component size and additional energy consumption, CS_i refers to the size of the component and n denotes the number of deployments. The energy consumption of introspection (E_I) and control (E_C) operations are given by:

$$E_I = \sum_{i=1}^{n} E_i. \tag{3}$$

$$E_C = \sum_{i=1}^{n} E_i. \tag{4}$$

Where n refers to the number of introspection or control commands and E_i denotes the energy consumption of i introspection or control commands.

B. Energy Measurement Methodology

This section introduces a methodology that may be used to obtain fine-grained energy calibration data for any mote platform and reconfiguration API. The only requirements are the availability of a digital output pin on the test mote platform and the ability to instrument the code under test to signal when energy measurement should start and stop.

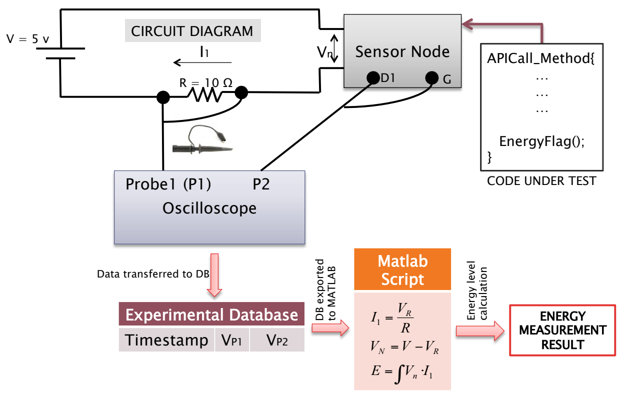


Fig. 1. Energy Monitoring Methodology

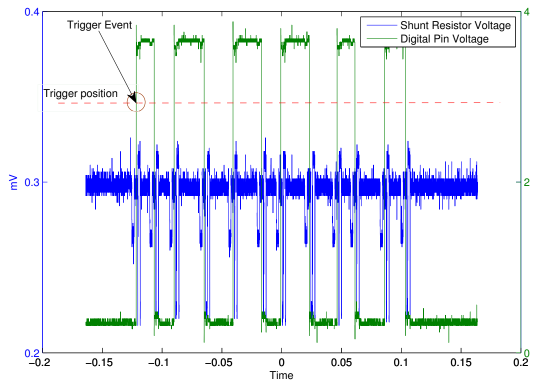


Fig. 2. Example DSO Experiment Output

1) Hardware Instrumentation: The proposed methodology is designed to measure individual API calls, which may execute over short time-scales. Contemporary mote platforms offer from tens [10] to hundreds [11] of MIPS, requiring high-speed energy monitoring. To meet these time constraints, a Digital Storage Oscilloscope (DSO) is required. We use a common mid-range DSO with a maximum sampling frequency of 500 MSa/s [16], which is fast enough to provide clock-tick accurate power measurements for motes of up to 250 MIPS, which is faster than either of our test platforms. The power consumption of the mote under test is measured using Ohm's law based upon the voltage drop over a shunt resistor placed in series with the power supply of the mote, as shown in the circuit diagram provided in Figure 1. In our experiments we use a high precision 10Ω resistor with a maximum relative error of 0.1%. Due to the short time-scale over which software operations occur and the limited memory of DSOs, it is necessary to automatically trigger the oscilloscope to start monitoring when a software operation is called and stop monitoring when it returns. To achieve this, we provide a simple API to support software instrumentation, as described in the following section.

2) Code Instrumentation: The software developer is supported with a simple code instrumentation scheme that allows methods to be monitored using the hardware setup described in the previous section. For remotely initiated operations, each method must be executed multiple times, as the total energy consumed in enacting an API call also includes the energy consumed by the radio in receiving the command, which happens before the API call itself. To instrument a method, the EnergyFlag() method call is added to the end of the function. This method flips the correct digital output pin from low to high or vice versa. The operation is then called multiple times in quick succession. This allows the complete operation, including radio transmissions, to be identified by only considering the interval in between two pin flips. The signal captured by the DSO is shown in Figure 2.

3) Data Analysis: As described previously, each experiment is composed of several executions of the code block being tested. Once an experiment has been successfully run, data from the DSO is exported to an experimental database. Each entry in the database contains: (i.) a time-stamp, (ii.) the voltage drop over the shunt resistor and (iii.) the digital output signal state. Together this information is sufficient to perform a fine-grained trace of power consumption.

Figure 2 shows a graphical representation of a single set of experimental results. The blue signal represents the voltage drop of the shunt resistor used to calculate the current of the circuit, and the green line is the voltage value of the digital pin. The red line represents the 'trigger position', i.e. the voltage value of the digital pin which causes energy measurement to start. The transitions of the green signal (see Figure 2) represent the entirety of a single API call execution.

To calculate the energy consumption of a concrete API call, the database entries are analyzed by a Matlab script as follows:

1) The shunt resistor signal is split into many parts as indicated by the transitions of the digital output signal. In the case of the Figure 2, the blue signal is split into 6 parts as the EnergyFlag() is called 12 times. This gives the $V_{sr}(t)$ of the shunt resistor for each API call.
2) Once $V_{sr}(t)$ is calculated, Ohm's Law is applied to calculate the current draw: $I_c(t) = \frac{V_{sr}(t)}{R_{sr}}$.
3) The input voltage to the node n is calculated using: $V_n = V_{ps} - V_{sr}$, where V_{ps} is a constant voltage supplied by a external power supply.
4) The power consumption of the node is then calculated using: $P(t) = I_c(t) \times V_n$.
5) Finally, the energy consumption of a concrete API call is calculated as $E_i = \int_a^b P(t)\,\mathrm{d}t$.

To ensure the statistical validity of the experimental results, we applied simple statistical tests. We obtained 10 energy consumption readings for each API call. We then computed the mean and the standard deviation for each API call. We also calculated the confidence interval for the mean to identify the range of energy cost for each API call, with a confidence level of 95%. For the calculation of confidence interval, we used the small sample confidence interval methodology based

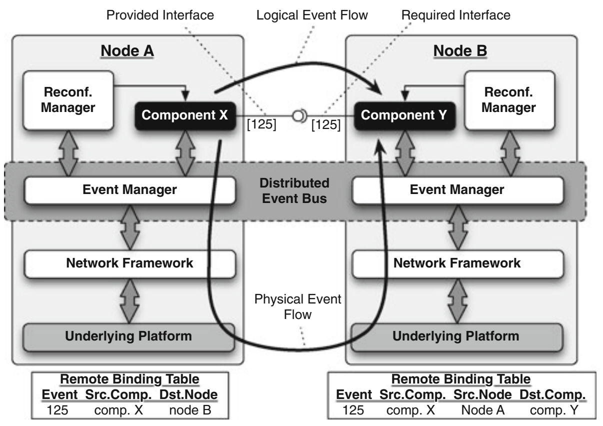


Fig. 3. Communicating LooCI Execution Environments

on t-distribution [17].

V. Modeling The LooCI Reconfiguration API on Heterogeneous Mote Platforms

We now use the generic energy model and energy measurement methodology to create a specific energy model for the LooCI middleware running on the Sun SPOT [11] and the AVR Raven [10].

A. The LooCI Middleware

We model the reconfiguration API of the Loosely-coupled Component Infrastructure (LooCI) [9], a component-based middleware for sensor networks that was developed by our group. LooCI is comprised of an execution environment, component model and event-based binding model. The architecture of the LooCI execution environment is shown in Figure 3. LooCI provides a good test platform as it runs on heterogeneous hardware/software stacks and thus affords the opportunity to demonstrate that our energy modeling approach is *generic* (i.e. not tied to a specific hardware platform, operating system or programming language).

LooCI currently supports three *underlying platforms*: Contiki [5], Squawk [6] and OSGi [18]. The LooCI *event manager* implements an Event Bus to which all components are connected. The *reconfiguration manager* maintains references to all local components and enacts incoming deployment, configuration and introspection commands via the event bus. *Deployment commands* allow for the insertion and removal of software components. *Introspection commands* allow for the discovery of what components are present on a node, their interfaces, state and bindings. *Configuration commands* allow for the activation, deactivation and binding of components. All commands may be enacted remotely at runtime. The LooCI reconfiguration API is shown in Listing 1.

Listing 1. The core LooCI API

Deployment
```
CompID    deploy(ComponentFile, NodeID)
Boolean   removeComponent(CompID, NodeID)
```
Configuration
```
Boolean   deactivate(CompID, NodeID)
Boolean   activate(CompoID, NodeID)
Boolean   wireLocal(EventType, SourceCompID,
                 DestCompID, NodeID)
Boolean   wireFrom(EventType, SrcCompID, SrcNodeID,
                 DestCompID, DestNodeID)
Boolean   wireTo(EventType, SrcCompID, SrcNodeID,
                 DestNodeID)
```
Introspection
```
CompID[]  getComponentIDs(NodeID)
String    getComponentType(NodeID, CompID)
State     getComponentState(NodeID, CompID)
Event[]   getInterfaces(NodeID, CompID)
Event[]   getReceptacles(NodeID, CompID)
```

B. Mote Platforms

We have modeled the energy cost of the LooCI reconfiguration API on two mote platforms: the AVR Raven [10] and Sun SPOT [11]:

The **AVR Raven** offers a 20MHz ATmega1284PV MCU, 16KB RAM, 128KB flash and AT86RF230 IEEE 802.15.4 radio [10]. The Raven motes run the Contiki OS [5] and the LooCI middleware [9]. All software is implemented in C.

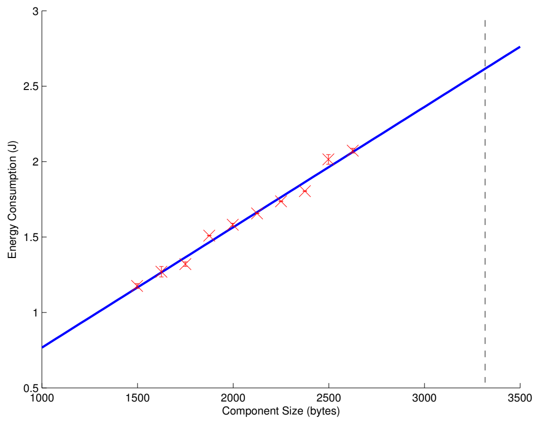
The **Sun SPOT** offers a 180MHz ARM920T MCU, 512KB RAM, 4MB flash and CC2420 IEEE 802.15.4 radio [11]. The SPOT motes run the Squawk OS and JVM [6] and the LooCI middleware [9]. All software is implemented in Java.

As can be seen from the specifications above, these motes are heterogeneous in terms of hardware resources, operating systems and languages and are thus appropriate to demonstrate the *generic* nature of our approach.

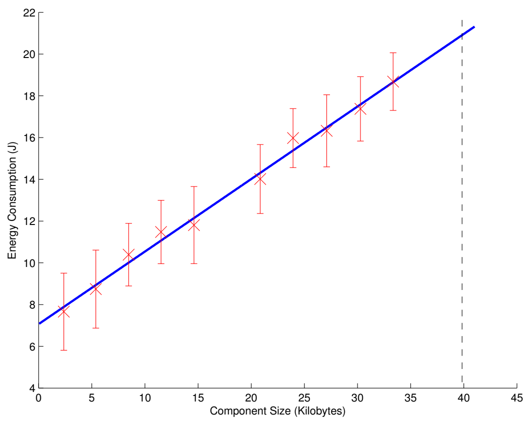
C. Energy Models

The LooCI reconfiguration API is grouped into *introspection* commands, which support inspection of component configurations, *deployment* commands which support over-the-air installation and removal of components and *configuration* commands which allow components to be bound together, activated and deactivated. Any reconfiguration action in LooCI is thus composed of a set of *deployment* and *configuration* operations, while *introspection* commands allow component configurations to be validated before and after reconfiguration.

As expected, *deployment* operations consume orders of magnitude more energy than *introspection* or *configuration* commands due to the transmission of component functionality. This is shown in Figure 4, wherein the red cross shows the average energy cost of the component deployment and the red line indicates the 95% confidence interval. In the current version of LooCI, the maximum component size on the AVR Raven is 3.2KB, while the maximum component size on the Sun SPOT is 40KB.

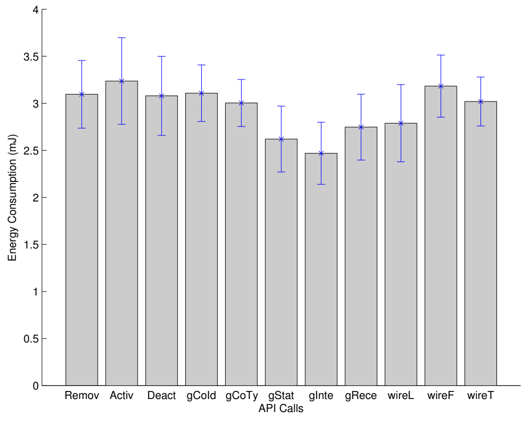


(a) Energy Cost of Software Deployment on AVR Raven

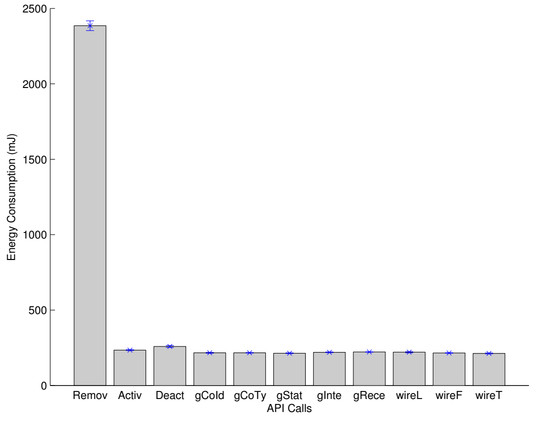


(b) Energy Cost of Software Deployment on Sun SPOT

Fig. 4. Energy Cost of Software Deployment on AVR Raven and Sun SPOT



(a) Energy Cost of LooCI Reconfiguration API on AVR Raven



(b) Energy Cost of LooCI Reconfiguration API on Sun SPOT

Fig. 5. Energy Cost of Non-Deployment API Calls in LooCI

Figure 5 shows the energy cost of non-deployment remote API calls for Raven and Sun SPOT hardware platforms. The average energy cost of *non-deployment* API calls follows a similar trend on both of our experimental platforms, except that the energy cost of removing a component consumes a large amount of energy on the Sun SPOT due to the energy used when accessing flash memory. The average energy consumption is 3mJ for the AVR Raven and 225mJ for the Sun SPOT, thus Raven API calls consume 1.33% of the energy consumed by the SPOT API calls. We have represented the 95% confidence interval with blue bars for each API call.

Figure 4 shows that, there is a positive relationship between the energy cost of the deployment and the component size. A linear regression equation was computed from our sample data and is shown by the blue line in Figure 4. Our linear energy model for the component deployment captures the relationship between the energy cost of the deployment and the component size. This model can therefore be used to obtain the energy cost of any component deployment for Raven and Sun SPOT hardware platforms.

VI. Implementation and Evaluation

All reconfiguration in LooCI is enacted via a manager component which runs on the network gateway. The manager accepts and executes simple scripts of reconfiguration API calls. We extended this script interpreter to provide energy cost estimates for all reconfiguration scripts using the model presented in the previous section. This provides developers with a simple mechanism to assess the energy costs of reconfiguration actions before they are enacted. In the following section we explore the accuracy of the model in predicting the energy cost of archetypal reconfiguration scripts for our case-study applications. All experiments report the energy cost of reconfiguration for a single mote in a one-hop network. As the

	Raven		Sun SPOT
	Data Size (bytes)	ELF Size (bytes)	Suite Size (bytes)
Smart Parking			
Manager	154	1068	2404
Sensor	338	1408	2256
Filter	202	1140	2117
Battery	318	1368	2229
Total	1012	4984	9006
Waste Bin Tampering			
Manager	452	1068	2443
Sensor	316	1388	2061
Battery	318	1368	2228
Total	1086	3824	6732

TABLE I
SIZE OF COMPONENT-BASED APPLICATION COMPOSITION

	Raven		Sun SPOT
	Data Size (bytes)	ELF Size (bytes)	Suite Size (bytes)
Smart Parking	436	1682	6289
Waste Bin Tampering	472	1720	6048

TABLE II
SIZE OF THE SINGLE APPLICATION IMPLEMENTATION

router tier is powered, we do not model energy consumption in this tier and only the sensor motes are relevant.

A. Case-study Evaluation

To evaluate our energy model, two versions of the smart parking and waste bin tampering applications were developed. The first version was developed as a single executable application, while the second version was developed as a composition of components. A series of reconfiguration scripts were then run in order to compare the energy predictions of the model against the energy readings recorded by the oscilloscope. All experiments were conducted 10 times.

Tables I and II show that, as expected, single-unit applications are smaller than component-based applications due to the overhead of component meta-data and their encapsulation in an executable object format prior to deployment. On Contiki components are encapsulated in the Executable and Linkable Format (ELF) [19], while on Squawk components are encapsulated in a custom suite format [6]. Table IV shows that a component-based implementation of the case-study applications incurs a size overhead for deployable components of between 122% and 196% for the Raven and between 11% and 43% for the SPOT. The higher overhead on Contiki is primarily due to inefficiencies in the ELF object format. Switching from ELF to Compact ELF would reduce overhead by 50% [19], however, this is outside the scope of this work.

Experiment 1 - Initial Configuration: The first experiment we performed was initial deployment and configuration of each application implementation. In the case of the single-unit application, no configuration is necessary, and therefore all energy cost is due to the deployment of functionality. In the case of the component-based application, components had to be deployed, wired and activated. The results of our energy evaluation are presented in Tables III and IV, which shows that (i.) our energy model is on average 98.57% accurate on the AVR Raven and 87.3% accurate on the Sun SPOT and (ii.) the initial component-based configuration incurs an energy overhead of 178.63% on the AVR Raven and 176.94% on the Sun SPOT. The greater error in predicting the energy consumption of the SPOT is attributed to the non-deterministic behavior of the Squawk JVM.

Experiment 2 - Reconfiguration: The second experiment tackles a problem which emerged for OneAccess during small-scale testing of their system. The magnetometer sensor was proving hard to calibrate and so the application had to be reconfigured to also use the IR sensor to detect when cars arrived or left. In the case of a component-based application, this change necessitates the deployment, wiring and activation of a new IR sensor component, while in the case of the single application, it requires wholesale redeployment of the application image. The results of our energy evaluation are presented in Tables III and IV, which show that (i.) our energy model is 96.33% accurate on the AVR Raven and 93.08% accurate on the Sun SPOT and (ii.) component-based reconfiguration results in energy savings of between 18.98% and 32.52% compared to a single application implementation for the underlying platform.

In summary, it can be seen that our generic energy modelling approach is accurate for application-based and component-based applications running on heterogeneous OS and hardware platforms. Furthermore, our analysis reveals that while component-based reconfiguration has a significantly higher initial cost than application-based reconfiguration, incremental software updates are significantly more efficient, which over the life-time of the sensor network is likely to compensate for the initial configuration overhead.

Using our energy model to estimate the cost of monolithic re-flashing of application and OS functionality, as used in TinyOS [4], we find that the cost of configuring and reconfiguring our case-study applications is over 40 Joules for the Raven and 340 Joules for the SPOT. This is more than an order of magnitude more energy than is required by either application-based or component-based (re)configuration.

VII. RELATED WORK

Energy consumption is one of the most important considerations WSN developers face. This issue has led to the development of processes, methodologies and simulation software designed to estimate energy consumption in WSN. Work in this area can be categorized into three groups: analytical, software-based and experimental methods.

	Raven			Sun SPOT		
	Estimation (mJ)	Real Consumption (mJ)	Accuracy (%)	Estimation (mJ)	Real Consumption (mJ)	Accuracy (%)
Smart Parking Initial Configuration	3885	3887	99.94	32876	29851	90.79
Bin Tampering Initial Configuration	3387	3484	97.21	24594	20612	83.81
Smart Parking Reconfiguration	1089	1124	96.33	7780	7260	93.08
		Average	97.83		**Average**	89.23

TABLE III
EVALUATION OF ENERGY MODEL ACCURACY FOR INITIAL CONFIGURATION AND RECONFIGURATION

	Effect of Using Components on Size			Effect of Using Components on Energy		
	Comp-based (b)	App-based (b)	Change (%)	Comp-based (mJ)	App-based (mJ)	Change (%)
Raven						
Smart Parking	4984	1682	+196.31	3887	1311	+196.49
Waste Bin Tampering	3824	1720	+122.32	3484	1336	+160.77
Smart Parking Reconfiguration	1404	1922	-26.95	1124	1594	-29.49
SunSPOT						
Smart Parking	9006	6289	+43.20	29851	9302	+220.90
Waste Bin Tampering	6732	6048	+11.30	20612	8847	+132.98
Smart Parking Reconfiguration	2084	6409	-32.52	7260	8961	-18.98

TABLE IV
IMPACT OF COMPONENT-BASED RECONFIGURATION ON APPLICATION SIZE AND ENERGY CONSUMPTION

A. *Analytical Methods*

Analytical models provide designers with the possibility of evaluating the lifetime of their WSN applications in a fast and platform-independent way. In [20], the authors propose a probabilistic lifetime energy model based upon the relationship between the lifetime of a single mote and the whole sensor network. A different approach is presented in [21] where a state-based battery model is proposed. This model has been implemented within the resource constraints of a sensor node and provides accurate battery life estimates. A key problem of these methods is that they only take into account energy consumption related to packet transmission and reception. These approaches therefore provide poor estimates when the energy consumption of other devices such as a CPU or sensors is significant. For example our results show that API calls that access the flash memory on the Sun SPOT consume significant amounts of energy.

B. *Software-based methods*

In [22], authors have identified and quantified the different factors which cause deviations between the software and hardware techniques. Furthermore, the have proven that software techniques are a feasible way of estimating energy consumption (estimation error is $1.13\% \pm 1.15\%$), particularly evaluating protocols where the CPU is used intensively. PowerTOSSIM [23] is a simulation environment for sensor networks based on the TOSSIM simulator [24]. This system is able to track the activity of each hardware component on the simulated motes. PowerTOSSIM generates a data file which is combined with a platform-specific energy model in order to calculate the energy consumption of each component. A similar approach is proposed in [25], wherein the authors present a mechanism to calculate how long each of the node's hardware components have been active. Once these times are known, the energy consumption of a mote can be estimated. Software methods are suitable for measuring energy consumption when large simulations are required. However, they do not allow for the accurate measurement of the energy consumption of a node over short periods of time. In contrast to our approach, software based methods do not provide explicit support for modelling new hardware platforms.

C. *Experimental methods*

In [26], the authors propose the use of large capacitors to estimate the lifetime of a sensor node. By replacing the battery of the nodes with such a capacitor and measuring the voltage before and after a set of operations, it is possible to accurately discern energy consumption due to the predictable discharge profile of capacitors. A more commonly-used methodology for measuring energy consumption is based on the use of an oscilloscope, an operational circuit connected to the target node and a program executing on a PC to analyze the data obtained from the oscilloscope [27], [28]. We build upon this energy monitoring approach, while adding software instrumentation to automate the testing process and providing a model that accurately predicts the energy consumption of software operations.

VIII. CONCLUSIONS AND FUTURE WORK

This paper has highlighted the critical problem of predicting the energy cost of reconfiguration operations for WSN middleware. We address this problem through two contributions: (i.)

a generic energy model that predicts the energy consumption of any reconfiguration API call and (ii.) the first energy-aware runtime reconfigurable component model. Evaluation shows that our energy model accurately predicts the energy consumed by reconfiguration actions on two heterogeneous mote platforms: the Sun SPOT [11] and the AVR Raven [10]. Furthermore, our analysis shows that while the component-based approach uses more energy during initial configuration, it is more efficient for software evolution.

Our future work will focus on two complementary research tracks. In the short term, we will address two key limitations of our current work by (i.) extending our model to predict the energy cost of reconfiguration for 3rd party middleware and (ii.) extending our approach to consider multi-hop network environments. In the longer term, we will focus on extending and automating our code instrumentation and testing approach, with the goal of realizing a reusable energy testbed.

We have taken care to ensure that the results presented in this paper are reproducible. All software tools, circuit diagrams, data sets and software used in this paper are available online at: http://people.cs.kuleuven.be/~wilfried.daniels/energy.

Acknowledgments

This research is partially supported by the Research Fund, KU Leuven, the IWT and iMinds (a research institute founded by the Flemish government). The research is conducted in the context of the iMinds-GreenWeCan, IBBT-STADiUM and IWT-SBO-SymbioNets projects.

References

[1] I. Leontiadis, C. Efstratiou, C. Mascolo, and J. Crowcroft, "Senshare: transforming sensor networks into multi-application sensing infrastructures," in *Proceedings of the 9th European conference on Wireless Sensor Networks*, ser. EWSN'12, 2012, pp. 65–81.

[2] C. Huygens, D. Hughes, B. Lagaisse, and W. Joosen, "Streamlining development for networked embedded systems using multiple paradigms," *IEEE Software*, vol. 27, no. 5, pp. 45–52, September 2010.

[3] D. Hughes, P. Greenwood, G. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, and K. Beven, "An experiment with reflective middleware to support grid-based flood monitoring," *Concurr. Comput. : Pract. Exper.*, vol. 20, no. 11, pp. 1303–1316, Aug. 2008.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Not.*, vol. 35, no. 11, pp. 93–104, Nov. 2000.

[5] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*. IEEE, 2004, pp. 455–462.

[6] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White, "Java on the bare metal of wireless sensor devices: the squawk java virtual machine," in *Proceedings of the 2nd international conference on Virtual execution environments*. ACM, 2006, pp. 78–88.

[7] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1:1–1:42, Mar. 2008.

[8] L. Mottola, G. P. Picco, and A. A. Sheikh, "Figaro: fine-grained software reconfiguration for wireless sensor networks," in *Proceedings of the 5th European conference on Wireless sensor networks*, ser. EWSN'08, 2008, pp. 286–304.

[9] D. Hughes, K. Thoelen, J. Maerien, N. Matthys, J. D. Cid, W. Horre, C. Huygens, S. Michiels, and W. Joosen, "Looci: a loosely-coupled component infrastructure for networked embedded systems," in *Proceedings of the 11th IEEE International Symposium on Network Computing and Applications*, ser. NCA '12. IEEE, 2012, pp. 236–243.

[10] "AVR RZ Raven Datesheet," 2012. [Online]. Available: http://www.atmel.com/Images/doc8117.pdf

[11] "Sun SPOT - Theory of Operation," 2012. [Online]. Available: http://www.sunspotworld.com/docs/Yellow/SunSPOT-TheoryOfOperation.pdf

[12] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, ser. WSNA '02. New York, NY, USA: ACM, 2002, pp. 88–97.

[13] N. Finne, J. Eriksson, A. Dunkels, and T. Voigt, "Experiences from two sensor network deployments self-monitoring and self-configuration keys to success," in *Wired/Wireless Internet Communications*, ser. Lecture Notes in Computer Science, J. Harju, G. Heijenk, P. Langendrfer, and V. Siris, Eds. Springer Berlin Heidelberg, 2008, vol. 5031, pp. 189–200.

[14] V. Dyo, S. A. Ellwood, D. W. Macdonald, A. Markham, C. Mascolo, B. Pásztor, S. Scellato, N. Trigoni, R. Wohlers, and K. Yousef, "Evolution and sustainability of a wildlife monitoring sensor network," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '10. New York, NY, USA: ACM, 2010, pp. 127–140.

[15] "RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks," 2012. [Online]. Available: http://tools.ietf.org/pdf/rfc6550.pdf

[16] "ADS1000 Series Digital Storage Oscilloscope," 2012. [Online]. Available: http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Tools/ADS1000-User-Manual.pdf

[17] D. Wackerly, W. Mendenhall, and R. L. Scheaffer, *Mathematical Statistics with Applications*, 7th ed. Duxbury Press, 2007.

[18] "OSGi Core Release 5," 2012. [Online]. Available: http://www.osgi.org/download/r5/osgi.core-5.0.0.pdf

[19] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 15–28.

[20] K. Sha and W. Shi, "Modeling the lifetime of wireless sensor networks," *Sensor Letters*, vol. 3, no. 2, pp. 126–135, 2005.

[21] J. Rahmé and K. Al Agha, "A state-based battery model for nodes' lifetime estimation in wireless sensor networks," in *Proceedings of the tenth ACM international symposium on Mobile ad hoc networking and computing*. ACM, 2009, pp. 337–338.

[22] P. Hurni, B. Nyffenegger, T. Braun, and A. Hergenroeder, "On the accuracy of software-based energy estimation techniques," in *Proceedings of the 8th European conference on Wireless sensor networks*, ser. EWSN'11, 2011, pp. 49–64.

[23] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh, "Simulating the power consumption of large-scale sensor network applications," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 188–200.

[24] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: accurate and scalable simulation of entire tinyos applications," in *Proceedings of the 1st international conference on Embedded networked sensor systems*, ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 126–137.

[25] A. Dunkels, F. Osterlind, N. Tsiftes, and Z. He, "Software-based on-line energy estimation for sensor nodes," in *Proceedings of the 4th workshop on Embedded networked sensors*, ser. EmNets '07. New York, NY, USA: ACM, 2007, pp. 28–32.

[26] H. Ritter, J. Schiller, T. Voigt, A. Dunkels, and J. Alonso, "Experimental evaluation of lifetime bounds for wireless sensor networks," in *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, jan.-2 feb. 2005, pp. 25 – 32.

[27] C.-C. Chang, D. J. Nagel, and S. Muftic, "Assessment of energy consumption in wireless sensor networks : A case study for security algorithms," in *Proceedings of IEEE International Workshop on Wireless and Sensor Networks Security (IEEE WSNS 2007)*. IEEE, 2007, pp. 1–6.

[28] K. Shinghal, A. Noor, N. Srivastava, and R. Singh, "Power measurements of wireless sensor networks node," *IJCES*, 2011.