An Automated Application-Independent Approach to Anomaly Detection in Wireless Sensor Networks

André Rodrigues[1,2], Jorge Sá Silva[1], and Fernando Boavida[1]

[1] Centre of Informatics and Systems of the University of Coimbra, Coimbra, Portugal
[2] Polytechnic Institute of Coimbra, ISCAC, Coimbra, Portugal
{arod,sasilva,boavida}@dei.uc.pt

Abstract. As Wireless Sensor Networks (WSN) gain momentum in what concerns applications and deployment, monitoring is becoming crucial in order to guarantee that anomalies are promptly detected. Unfortunately, current WSN monitoring solutions have several limitations, such as being tailored for specific applications, requiring dedicated or specific hardware, consuming precious energy and/or processing resources, or relying on manual or offline intervention. In this paper we propose an approach to anomaly detection in WSNs that addresses these limitations. The approach is based on two very simple metrics, a logging tool, and a data-mining algorithm, thus leading to the following key characteristics: very low resource consumption, application independency, very good potential for multi-WSN monitoring, and automation and simplification of the detection process. The proposed approach was validated by implementation, which showed that it is quite effective in detecting several typical anomalies.

Keywords: wireless sensor networks; anomaly detection; monitoring, testing and debugging.

1 Introduction

Detecting and diagnosing problems in Wireless Sensor Networks (WSNs) is now considered essential, due to the fact that the installed basis of this type of networks is growing at a fast pace. Numerous deployments exist, either of experimental or of commercial nature.

One such example, pertaining to the research area, is a recent deployment by Intel, which equipped a few hundred homes with devices that enabled the collection of a set of parameter values (environmental, physiological and behavioral), with the objective of assessing the potential of the WSN technology in the study of aging and chronic diseases [1]. One of the conclusions of the study was the need for tools for managing research infrastructures characterized by a large number of geographically dispersed facilities, where their direct users (in this case the elderly) do not have enough technological expertise to assist in diagnosing and solving problems that inevitably occur in the installed systems.

In what concerns commercial deployments, it is expected that WSN technology will be extensively used for supporting real time monitoring of multiple installations belonging to the same or different entities. For instance, one can easily foresee its use in farms to support animal health condition monitoring. As an example, a WSN-based system could support the collection of several parameters (e.g. environmental, physiological and behavioral parameters) and, based on their values and on a set of rules, generate alarms if something abnormal happened. Such a system could easily be deployed at several facilities and, naturally, would require adequate monitoring in order to ensure its proper functioning. This is clearly an example of a scenario in which the entity that commercializes / supports the system must also have the capability to real-time monitor the installations in operation at the various customers.

Although the scenarios for the monitoring of multiple and/or large scale WSN installations are rapidly emerging, existing tools have not yet met the requirements of simple, effective, automated, distributed, and general anomaly detection. Typically, monitoring tools are application-specific, resource-consuming, complex to configure and/or use, or restricted to a single WSN.

The work presented in the current paper is an attempt to prove that it is possible to explore anomaly detection approaches that meet the above-mentioned requirements by using two simple metrics, an existing logging tool, and a data-mining algorithm. Note that the main contribution of this paper is not the presentation of a monitoring tool but, rather, the demonstration that it is possible to develop general, application-independent, lightweight WSN monitoring tools that use simple metrics.

The paper is organized as follows. Section 2 identifies the set of requirements that should be met by WSN monitoring tools. Section 3 is dedicated to the detailed presentation of the developed approach, namely in what concerns its hardware and software platforms, used metrics, logs collection and parsing, data transformation, and detection and diagnosis. The proof-of-concept implementation was subject to evaluation in two simple scenarios comprising several anomalous conditions. Evaluation results are discussed in section 4. Section 5 identifies related work, by briefly presenting and discussing a representative set of WSN monitoring tools. Section 6 provides the conclusions and guidelines for further work.

2 Requirements

This section briefly presents and discusses the requirements that a WSN monitoring tool should address. These requirements were divided into two categories: scenario-related requirements, and performance/usage requirements.

2.1 Scenario-Related Requirements

- **Scalability** – the tool should be able to scale, both in terms of the number of nodes per WSN and the number of supported WSNs. As mentioned before, scenarios comprising the monitoring of several WSNs are likely to appear.
- **Support for inter-WSN homogeneity and intra-WSN heterogeneity** – in any given WSN there is often some heterogeneity at hardware and firmware levels; nevertheless, when looking at multiple WSNs run or supported by a given

organization, one can see that they mostly run the same applications on the same platforms. Thus, monitoring tools should enable to take advantage of this inter-WSN homogeneity, and also support intra-WSN heterogeneity.
- **Support geographically dispersed WSNs** – the existence of WSNs located at several locations is a factor that must be taken into consideration in the design of a monitoring tool, as suitable communication mechanisms must be in place.
- **Support for mobile nodes** – in WSNs it is quite common to have mobile nodes; this type of nodes has no negligible impact on data collection strategies and on communications; monitoring tools should be able to cope with these nodes.

2.2 Performance/Usage Requirements

- **Support all application paradigms** – in order to not be limited by the application paradigm (i.e., schedule-driven, query-driven, event-driven) the tool must not rely on the existence of specific operation patterns.
- **Support diverse hardware platforms and operating systems (OSs)** – WSN technology is fast changing, so it is important to ensure that a tool can be used with multiple OSs and platforms in order to cope with current and future needs.
- **Minimize the use of WSN resources** – typically, WSN nodes are resource constrained; thus, monitoring tools should minimize WSN resource consumption, such as energy, processing, and memory.
- **Easy to install and to use** – the effort required for integrating a monitoring tool in a WSN or a set of WSNs should be minimized. The same applies to the effort required to use the tool.
- **Flexibility and extensibility** – it is important to support mechanisms that allow the manager to tailor the tool to its needs (at deployment and at runtime) in order to enable wider applicability.

3 Proposed Approach

Having in mind the requirements identified in the previous section, we set out to demonstrate that it is possible to construct a simple, effective, automated, and application-independent anomaly detection tool.

To this effect, we developed a proof-of-concept implementation. This section provides details on this implementation, namely, on the used hardware and software platforms and on the overall design, including metrics calculation, logs collection, logs parsing, data transformation, anomalous events detection, and anomalous events diagnosis.

3.1 Selected Platform and Operating System

The tool prototype was implemented using TinyOS [2] on a new hardware platform called Hermes [3]. This platform includes a recent MSP430, an 868 MHz band radio, an SD card reader, an accelerometer, a gyroscope, a thermometer, an heartbeat receiver, and a power management system.

Having used an event-based operation system, the metrics that will be used as the basis for anomaly detection will be collected during event procedure instances. An event procedure instance [4] is the sequence that begins with a hardware interrupt and ends with the execution of the last code associated with the initial event.

It should also be noted that, although they are essential, the collected metrics must be complemented with additional information in order to effectively pinpoint the reasons for anomalous behavior. In the case of this proof-of-concept implementation, we decided to additionally collect call traces for the application code, and also log the executed tasks.

As a final remark, it should be noted that despite the specific choice of hardware platform, operating system, and implementation, the principles that guided the tool's construction (i.e., simple metrics, application-independence, logging, and data-mining) are general and can easily be implemented using other OSs and platforms.

3.2 Metrics Calculation

When considering the issue of what metrics to use for characterizing behavior as normal or abnormal, one should take into account aspects such as impact on node resources, applicability to diverse application paradigms, descriptive power, and hardware platforms and OSs specificity. Metrics with the following characteristics should thus be avoided:

- Application dependent (e.g., using statists on specific application events);
- Requiring detailed logs (e.g., complete call traces, or vectors of instruction counters as in [4]);
- Application paradigm dependent (e.g., counters on traffic);
- Requiring dedicated hardware or OS support (e.g., dedicated energy measurement devices, OS-integrated logging mechanisms).

From the above, one can conclude that elapsed time, processing, and energy are good candidates to characterize what happens between two consecutive application level events. These metrics are general, light to compute, and do not require sophisticated support mechanisms.

In the case of the proof-of-concept implementation, the selected metrics were processing and energy. The main reason for excluding the elapsed time was that, for typical WSN applications, it does not provide much information about the used processing resources, as the majority of time between two application level events is sleep time. This also means that, for instance, an anomaly leading to an increase in active time could easily be "concealed" by a slight variation in sleep time without much impact on the elapsed time metric value.

Each metric is calculated in the following way. At boot time the metric counter is reset. When the next application event happens the metric counter value is logged and associated to the previous application event. Finally, the metric counter is reset.

Processing Metric. The first approach to calculate a processing metric was to count the microcontroller (MCU) instructions (or the MCU cycles) executed between two consecutive application level events. Unfortunately, the Hermes platform does not directly support this.

The followed approach was to count the MCU (i.e. MSP430F2618) sub-main clock (SMCLK) cycles between consecutive application level events. The SMCLK in Hermes is defined to be based on the master clock (MCLK) divided by 4. The MCU TimerA was configured to be sourced by the SMCLK, resulting that when the MCU is not sleeping the SMCLK is running and TimerA is incremented.

This is an extremely light metric as the only processing required is to read or update the counter-associated variable. For simplicity reasons, from this point onward this metric will be called MCU cycles.

Energy Metric. In iCount [5] the authors explained how a carefully selected switching regulator used to provide regulated power to a sensor node can be used to provide energy consumption measurements, almost for free. Because the regulator they selected uses pulse frequency modulation, the switching frequency is almost directly related to the load current. Their idea was to connect the output of the regulator inductor to the MCU input clock (INCLK) line that can be used to source the TimerA. In this way, each time the voltage at the inductor crosses zero in the ascending direction, TimerA is incremented as a new switching cycle was detected.

Hermes uses a Power Management System (PMS) that includes two switching regulators based in the Pulse Wide Modulation (PWM) technique. Those regulators also support pulse skipping at light loads.

Being a PWM-based part, it does not enable to directly use the iCount approach. However, at light loads, the PMS supports a burst mode where energy is provided in a burst of pulses to minimize switching losses. During this operation mode it is possible to count the pulses (using an approach similar to iCount) to obtain an estimation of the consumed energy. From now on, for simplicity reasons, this metric is called energy consumption.

3.3 Logs Collection and Parsing

Logging mechanisms are required to collect the metrics and the information on the executed application events calls and tasks from the sensor nodes, and to forward it to the management system in order to support the detection and diagnostics functionality.

The logging mechanism should be flexible and expandable, avoid application source code modifications, be easy to install and use, take advantage of main application communication capabilities, introduce small latency, be light in terms of resources usage, be easily portable to other OSs / platforms, and support node heterogeneity.

Having to decide between developing a logging system according to the previous requirements or using an already available one, the decision was to use LIS [6], as it supports most of the requirements.

In LIS, a developer has to produce a LIS script using a declarative language, describing the logging mechanisms to be deployed and their location in the WSN application source code. Then, a PC-based instrumentation engine modifies the WSN application source code, according to the LIS script, in order to include logging statements. Also added to the sensor node code are a runtime library that supports the node logging function calls, and a code module supporting log data storage and retrieving functionality.

During runtime, execution traces and state are saved on local memory and can be sent to the sink node using either wired communication (i.e., SerialActiveMessages), or wireless communication (i.e., TinyOS CTP for multi-hop, or ActiveMessages for single-hop). When the packets arrive at the sink node they are parsed using a generic LIS parser and the LIS script information to produce meaningful information.

The LIS language can be used directly or as an intermediate language supporting reusable high-level task definitions. One of these high-level tasks is Region Of Interest (ROI) call trace monitoring, where the developer specifies a ROI (e.g. one TinyOS subsystem) and the system generates the corresponding LIS script that will enable to create a log of the function calls inside that subsystem. This functionality is very useful because it enables to avoid the need for manually creating a LIS script.

In the case of the current proof-of-concept implementation, it was necessary to modify the Python scripts associated with the ROI analysis functionality, in order to enable LIS to automatically instrument the application source code with the objective of generating call traces and metrics logs.

Finally, collecting the logs is done via the "timestampedlisten" console command, provided by TinyOS, which collects the packets sent by the sensor nodes where the logging mechanism is running. The collected packets are submitted to the LIS parser, which makes use of the LIS script information to output an easy-to-read listing of the logged call traces and collected metrics.

3.4 Data Transformation

The data file produced by the LIS parser is filtered to remove incomplete application event level details that resulted from packet losses. This is necessary because if some packets are lost they can compromise the parsing of the next log entries. LIS supports a mechanism to discard incomplete log entries. However, it was necessary to enhance this mechanism, as incorrect application event information was found in the parsed logs.

After this phase, the logs are parsed in order to generate an application-level event list file with the intended data format. This new file also contains additional information required to support further diagnosis of the anomalous events. To support all these transformations, a set of Python-based scripts was developed.

Each line in the generated file has the following format:

```
<class> <m1>:<v1> <m2>:<v2> # <event> <begin> <end>
```

In this format, <class> designates the class the event belongs to, <m1>:<v1> designates a metric/value pair, <event> is the name of the application-level event, and <begin> and <end> identify, respectively, the line of the LIS parsed file where log entries related to this event begin, and the line where they end.

The data before the "#" is used by the classification algorithm used to identify anomalous events. The data after the "#" is ignored by the classification algorithm, but is used to locate, in the LIS parsed file, the log information related to the detected anomalous events.

3.5 Anomalous Events Detection and Diagnosis

The selected classification algorithm is based on a machine learning technique, called Support Vector Machines (SVM) [7], which generates a model that can be used to predict the class of an instance. In this case, an instance is an application level event instance that has metrics values as attributes, and can be classified as normal or anomalous.

In the present scenario there are, nevertheless, two issues. First, a labeled training set is not available. Second, it is expected that the anomalous application level events represent a small fraction of all application level events (as the goal is to detect sporadic problems).

Considering this scenario, the selected approach (also followed by [4]) was to use an SVM variant called one-class SVM, and to assume that the training set only contains normal events, knowing that a small percentage of them may have been misclassified as such. By defining the percentage of misclassified events in the training set, one-class SVM will create a model that places the majority of the events in normal class side of the hyper-plane, while the remaining are placed on the anomalous class side. This learned model is then used to predict the class of future received application level events. The learned model can be periodically updated, in case it is required that the anomalous event detection mechanism has some flexibility to adapt to environment / system changes.

The reasons to select this technique were the following: it does not require previously labeled data, it can work with unbalanced data sets (i.e. sets with classes not equality represented), and the existence of a well documented and easy to use code library (LIBSVM [8]). This library includes Python scripts for simplifying its use, namely scaling data sets, selecting optimized kernel function parameters, training the model, and testing the data.

The output of LIBSVM is a classification for each application level event. An analysis script was developed that uses this information to locate, in the LIS parsed file, the logged information related to the application level events classified as anomalous. This enables the responsible person to analyze them in order to identify possible reasons for their classification.

Fig. 1 summarizes the activities involved in the anomalous events detection and diagnosis functionality detailed in this section.

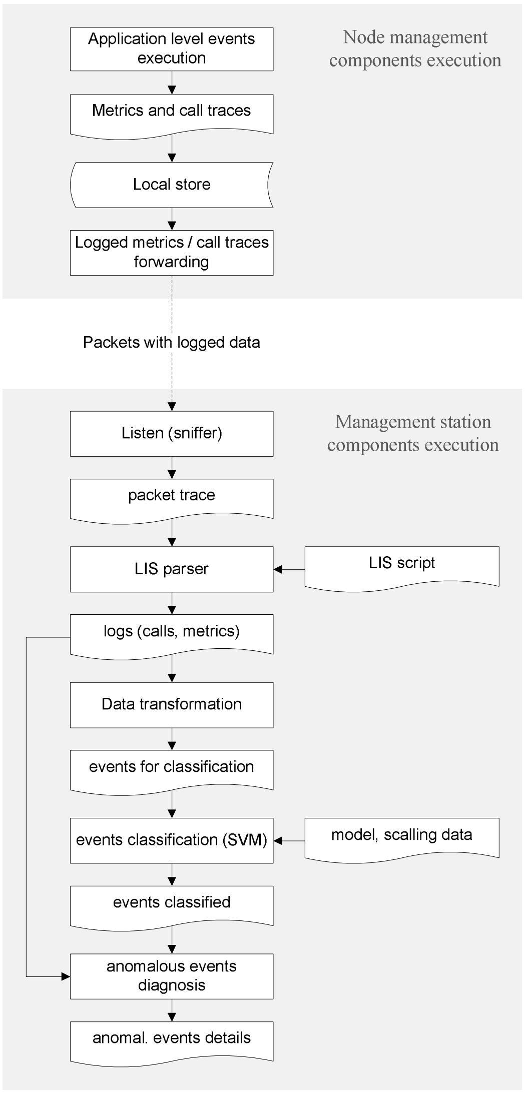
Node management components execution
Application level events execution
Metrics and call traces
Local store
Logged metrics / call traces forwarding
Packets with logged data
Management station components execution
Listen (sniffer)
packet trace
LIS parser
LIS script
logs (calls, metrics)
Data transformation
events for classification
events classification (SVM)
model, scalling data
events classified
anomalous events diagnosis
anomal. events details

Fig. 1. Detection and diagnosis fluxogram

4 Evaluation

The presented proof-of-concept implementation was subject to several tests, in order to assess the effectiveness and efficiency of the underlying concepts. This section begins by describing and presenting the results of a set of experiments carried on for evaluating the tool's detection and diagnosis capabilities. Subsequently, the tool is evaluated under the light of the initial requirements.

4.1 Experimental Results

Two sets of tests were carried out. In the first group of experiments the goal was to assess the MCU cycles metric for detecting anomalous behaviors.

The selected application was RadioCountToLeds (a standard TinyOS application where two nodes periodically broadcast packets containing a counter, and each time a node receives a packet it displays the counter's last 3 bits on the LEDs). This application was selected because it has a simple behavior and is publicly available, enabling the community to validate the paper results. The application code includes the events MilliTimer.fired, Receive.receive, and AMSend.sendDone. These are regularly triggered during normal application execution.

In the scenario used for the experiments two nodes executed RadioCountToLeds, and another one just collected the packets with the logs and send them, via USB, to the management station (Fig. 2).

The application source code was automatically transformed to include the logging mechanisms required to generate the call traces and the MCU cycles metric for the application level events, and to generate the call traces for the executed tasks.

For obtaining the training set, the RadioCountToLeds application was executed during 15 minutes under normal conditions, then the collected packet trace was parsed by LIS, transformed to remove incomplete application level events and to provide the event list in the LIBSVM format, and finally submitted to the LIBSVM script. This script scaled the data, selected the kernel function parameters (by using a grid-search approach and cross-validation), and finally outputted the learned model and the used scale data. The percentage of misclassified events in the training set was set at 1%. The events set used to train the classifier included 1486 events instances.

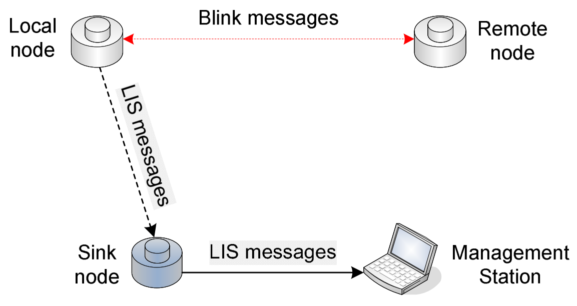

Fig. 2. Evaluation scenario

All the experiments had 5 minutes duration and the results are presented in Table 1. Before proceeding to analyse the results, two aspects should be highlighted. Firstly, it should be noted that, in the absence of problems, the percentage of events classified as normal should be around 99% (as the defined threshold for misclassified events in the training phase was 1%). Secondly, in Table 1, the 'Remarks' column presents the details found in anomalous events logs that provided clues for events classification.

Experiment #1 was designed to evaluate how the anomaly detection functionality reacted to a permanent failure, such as the remote node stopping to broadcast its counter messages. The low percentage of normal events, with more than 40% of the events being classified as anomalous, clearly points to some kind of error. This was easily diagnosed by observing the absence of Receive.receive events in the logs.

The goal of experiments #2 and #3 was to determine if a logic error (that resulted in additional execution of code) could be detected. The MilliTimer.fired event code was changed to contain a cycle that incremented a counter from 1 to 100 (or to 1000 in the case of experiment #3). This cycle was executed in 10% of the MilliTimer.fired event executions. In both cases, the anomalous events (an excess of MilliTimer.fired events with a high MCU cycles value) were detected, as indicated by a percentage of normal events below 99%.

The goal of experiment #4 was slightly more ambitious, namely to determine if another type of logic error (specifically, in this case, the counter for the broadcast message being incorrectly increment twice in 10% of the cases) could be detected. This was done by modifying the MilliTimer.fired event code. The impact on the application executing in the local node was minimal and, thus, was not detected.

Another set of experiments aimed at evaluating the efficacy of the energy consumption metric in the detection of anomalous behaviors. In these experiments the timer used to send the counter messages was increased from 1.9 s to 5 s. and the experiments duration was 10 minutes. Table 2 presents the experiments' results.

Table 1. Classification summary (MCU cycles metric)

#	Condition	Normal events %	Events (norm/total)	Remarks
1	remote node off	58.21%	163/280	no Receive.receive events
2	100 i++	98.66%	443/449	MilliTimer.fired > 21750
3	1000 i++	95.53%	406/425	MilliTimer.fired > 23600
4	extra op	99.04%	416/420	not detected

Table 2. Classification summary (energy metric)

#	Condition	Normal events %	Events (norm/total)	Remarks
5	5 remote reboots	98.77%	322/326	AMSend = 6.213.697
6	1 local reboot	96.93%	347/358	Boot.booted
7	up 1 floor	91.97%	229/249	AMSend > 6.000.000
8	gyro ON	75.36%	260/345	AMSend ~ 3.200.000

In experiment #5, one node was rebooted 5 times. By analysing the logs from the other node, it was possible to detect the existence of anomalous behaviour (as indicated by a percentage of normal events below 99%), and subsequently identify AMSend.sendDone anomalous events with high-energy values. This seemed to indicate that some messages were not sent by the rebooted node or that they got lost.

In experiment #6 the significantly lower percentage of normal events gives a hint on some anomaly. After inspecting the logs for the anomalous events, a local node reboot was detected (specifically, one of the anomalous events was a Boot.booted).

In experiment #7, one node was moved away from the other one floor up. The relatively low percentage of normal events, in conjunction with fewer events in the data set for the node that did not move, was a hint for problems. By inspecting the anomalous events logs it was possible to identify several AMSend.sendDone anomalous events with high-energy values. This suggested that packet losses had occurred, but required an analysis of the other node logs.

Experiment #8 was designed to determine if a wrong power state in a device would be detectable. Specifically, the gyroscope was not turned off at boot time in order to create an anomaly. The considerably low percentage of normal events clearly indicated that something was wrong. Also, the existence in the logs of several AMSend.sendDone anomalous events with energy values higher than 3.000.000 confirmed it. Nevertheless, because this type of problem does not impact program execution, there was no information in the logs that helped diagnosing it.

In light of the achieved results, it is clear that the simple metrics approach combined with logging analysis and data mining allowed the detection of most anomalous conditions. It should be noted that the objectives of this proof-of-concept implementation and the associated experiments were the assessment of the efficacy of the automatic anomaly detection, not the diagnosis itself.

4.2 Requirements Analysis

The goal of developing and evaluating the presented prototype implementation was, on one side, to validate the concepts on which it was based – namely, the use of simple metrics, light logging tool, and data-mining – and, on the other hand, to assess its ability to meet the identified requirements. In the previous sub-section, the tool was assessed with respect to the former. In the current sub-section, an analysis pertaining to the latter is presented.

- **Scalability** – in the current implementation, the parsing, data transformation, and classification tasks required to process a 10 min log took less than 1.5 s (1.295 s, 0.172 s, and 0.023 s, respectively) per sensor node, in an Intel Core 2 Duo 2.4 GHz computer with 3 MB of RAM. This is a low processing time. To maintain low parsing times with a high number of nodes and with more components having their function calls logged, an optimised implementation should only send the metrics at detection time, locally saving the call trace logs, for on-request later inspection.
- **Support sensor node heterogeneity** – the tool can support WSN devices with diverse hardware and software. This could be done by grouping logs, at the management system, according to their software and hardware configuration. In this way, each group of log information contains events data from sensor nodes with the same hardware / software. Each group is then analyzed individually according to the fluxogram presented in Fig. 1.
- **Support geographically dispersed WSNs** – tools implemented according to the presented principles can transparently work with monitoring data originating from multiple WSNs, provided each WSN is connected to the Internet via a gateway device that communicates with the management system.

- **Support mobile nodes** – LIS communications can use TinyOS Active Messages or CTP. For WSN applications using these protocols, the tool will support the same mobility pattern as the application.
- **Support all application paradigms** – the detection functionality is based on the occurrence of application level events on the monitored sensor nodes and uses general metrics. In this way, it supports all type of application paradigms. Nevertheless, for WSNs applications where sensor nodes sleep for long periods of time and only wakeup on rare events, this approach alone will not work. This is a common problem, not specific to the presented approach, and the usual solution is to support mechanisms, either initiated by sensor nodes or by the sink, that enable to know if a sensor node is alive.
- **Support diverse hardware platforms and OSs** – most of the detection functionality at sensor nodes is based on LIS, the exception being the metrics calculation. Currently, LIS is directly supported by Mica2/Z, TelosB, and Hermes. Using it with other OSs, like Contiki, should not be too difficult, given that LIS operates by modifying C language based applications.
- **Minimize the use of WSN resources** – MCU, RAM, and ROM consumption are minimized because LIS is a very efficient log tool and because the metrics calculation is very light. The impact in energy and bandwidth is mostly related to the number of components having its function calls logged. Only sending call traces on demand will enable further savings.
- **Be easy to install and to use** – in the case of this proof-of-concept implementation, deploying the tool on a sensor node just requires compiling the WSN application with an option stating which components should have their activity logged (in the presented evaluation experiments, these were the application and the scheduler components). Usability can only be evaluated with an integrated platform and not with a proof-of-concept prototype implementation. However the experiments did not require much analysis work.
- **Be flexible and extensible** – post deployment configuration of the logging functionality at sensor nodes was not supported in the current implementation. The implemented functionality was based on an enhanced version of the ROI mechanism provided by LIS. This work can be easily extended by using the LIS script language, more metrics, and additional classification algorithms. Most of the work would be in enhancing the parser to support the new metrics, and on developing Python scripts to support the data formats required by the new classification algorithms.

5 Related Work

Several pieces of work have addressed the problem of WSN monitoring. They are briefly mentioned in this section, with a focus on the ones that had a higher impact on the presented proof-of-concept design and implementation.

MANNA [9] was one of the first management frameworks for WSNs. In spite of being a very flexible and general architecture, it did not target the support of automatic detection and diagnosis mechanisms for the joint management of WSNs.

SWARMS [10] target the management of wide area WSNs in diverse geographic locations, providing diagnostic and programming functionalities. It was designed to be scalable, flexible, and extensible. The major concern with this architecture is the fact that it requires each sensor node to be directly connected to a computer that executes a node mate process. Moreover, there are no provisions for supporting automatic detection and diagnosis.

MARWIS [11] targets the management of a heterogeneous WSN by dividing it into homogeneous WSNs connected by a mesh network. It was designed to be scalable, flexible, and extensible. The major problem with this architecture is that supporting sensor node heterogeneity by dividing a WSN in a set of homogenous WSNs does not fit well when there is a need to manage several heterogeneous WSNs belonging to diverse organizations. Moreover, MARWIS assumes sensor nodes are executing Contiki-based applications and there are no provisions for supporting automatic detection and diagnosis.

Sentomist [4] is a tool for identifying potential transient bugs in WSN applications, which also uses SVM to identify anomalous events. Nevertheless, by using a metric based on information from the specific processor instructions executed in each event, it requires the use of the Avrora emulator, which restricts it to lab use.

Finally, there are several tools developed to help diagnose WSNs operating in the field. We have realized an extensive survey [12] that describes, analyses, and compares a representative set of them. This work guided us in the development of the current approach to WSN anomaly detection, and in identifying LIS as a good candidate to be used.

6 Conclusion

This paper proposed a simple approach to anomaly detection in wireless sensor networks, based on the use of two general metrics, a light logging strategy, and a machine learning technique. The thesis was that these underlying concepts would be enough to develop an automated, application-independent, light tool, capable of monitoring multiple WSNs. In order to assess this, a proof-of-concept implementation was developed and subject to testing. The results have shown that the proposed approach has very good potential and characteristics, being able to detect hardware and software anomalies in a very effective way, and without compromising the identified requirements, such as scalability, heterogeneity, applicability, generality and ease of use.

The presented work opens many lines for further work. First and foremost, a more extensive and broader scope evaluation should be done. In addition, other general metrics should be identified and explored. The development of a full implementation for use in existing, deployed WSNs would also very interesting, as well as the support for IPv6 (6LoWPAN) in order to increase the tool's applicability.

Acknowledgment. We would like to thank Roy Shea (from UCLA) for clarifying several aspects of the LIS operation. This work was partially financed by the iCIS project (CENTRO-07-ST24-FEDER-002003).

References

1. Hayes, T., Pavel, M., Kaye, J.: Gathering the Evidence: Supporting Large-Scale Research Deployments. Intel Technology Journal 13(3) (2009)
2. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: Tinyos: An operating system for wireless sensor networks. In: Weber, W., Rabaey, J., Aarts, E. (eds.) Ambient Intelligence. Springer (2004)
3. Rodrigues, A., Silva, M., Camilo, T., Blanco, N., Pedro, J., Martins, J., Silva, J.S., Boavida, F.: Hermes: A versatile platform for wireless embedded systems. In: Proceedings of the IEEE WoWMoM 2012. IEEE, San Francisco (2012)
4. Zhou, Y., Chen, X., Lyu, M., Liu, J.: Sentomist: Unveiling Transient Sensor Network Bugs via Symptom Mining. In: Proceedings of the IEEE ICDCS, pp. 784–794 (2010)
5. Dutta, P., Feldmeier, M., Paradiso, J., Culler, D.: Energy Metering for Free: Augmenting Switching Regulators for Real-Time Monitoring. In: Proceedings of the IPSN 2008, pp. 283–294. IEEE (2008)
6. Shea, R., Cho, Y., Srivastava, M.: LIS is More: Improved Diagnostic Logging in Sensor Networks with Log Instrumentation Specifications. TR-UCLA-NESL-200906-01 (2009)
7. Hsu, C.-W., Chang, C.-C., Lin, C.-J.: A Practical Guide to Support Vector Classification. Technical Report, Department of Computer Science, National Taiwan University (2010), `http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf`
8. Chang, C.-C., Lin, C.-J.: LIBSVM: A library for support vector machines. ACM Transactions on Intelligent Systems and Technology 2(3), article no. 27 (2011)
9. Ruiz, B., Nogueira, J., Loureiro, A.: MANNA: A management architecture for wireless sensor networks. IEEE Communications Magazine 41(2), 116–125 (2003)
10. Gruenwald, C., Hustvedt, A., Beach, A., Han, R.: SWARMS: A sensornet wide area remote management system. In: Proceedings of the TridentCom (2007)
11. Wagenknecht, G., Anwander, M., Braun, T., Staub, T., Matheka, J., Morgenthaler, S.: MARWIS: A management architecture for heterogeneous wireless sensor networks. In: Harju, J., Heijenk, G., Langendörfer, P., Siris, V.A. (eds.) WWIC 2008. LNCS, vol. 5031, pp. 177–188. Springer, Heidelberg (2008)
12. Rodrigues, A., Camilo, T., Silva, J.S., Boavida, F.: Diagnostic Tools for Wireless Sensor Networks: A Comparative Survey. Journal of Network and Systems Management 21(3), 408–452 (2013), doi:10.1007/s10922-012-9240-6