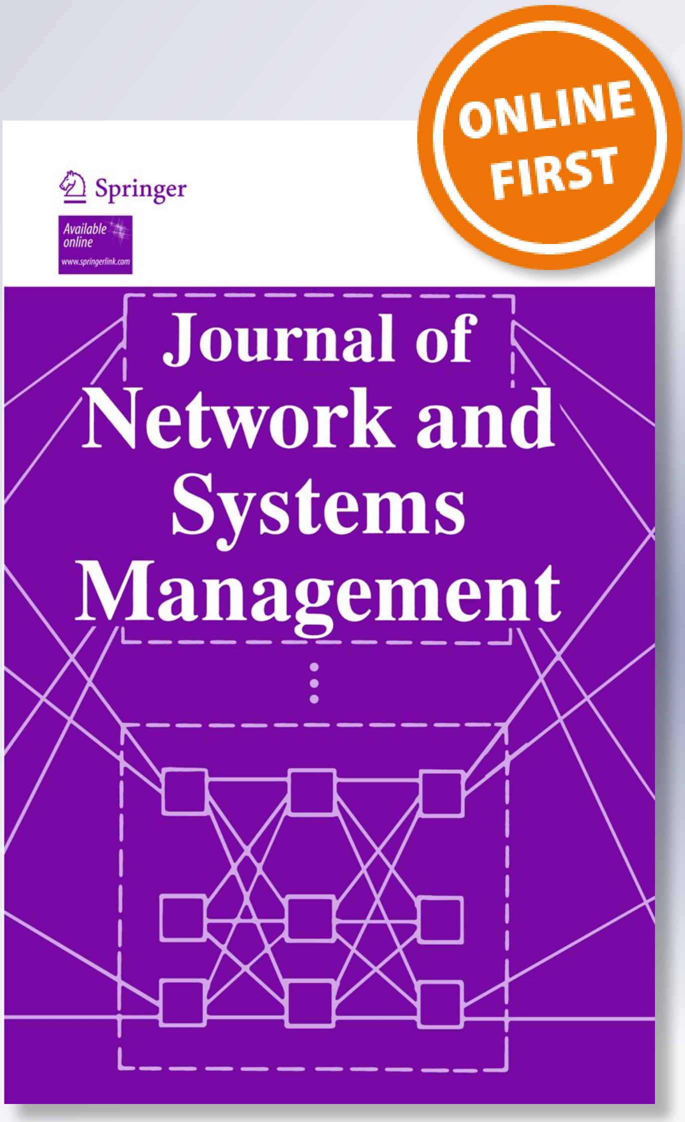
Diagnostic Tools for Wireless Sensor Networks: A Comparative Survey

André Rodrigues, Tiago Camilo, Jorge Sá Silva & Fernando Boavida

Journal of Network and Systems Management

ISSN 1064-7570

J Netw Syst Manage
DOI 10.1007/s10922-012-9240-6



J Netw Syst Manage
DOI 10.1007/s10922-012-9240-6

Diagnostic Tools for Wireless Sensor Networks: A Comparative Survey

André Rodrigues · Tiago Camilo · Jorge Sá Silva · Fernando Boavida

Received: 7 June 2011 / Revised: 19 May 2012 / Accepted: 22 May 2012

Abstract The availability of tools to diagnose Wireless Sensor Network (WSN) failures is a key success factor for this type of networks as already demonstrated by several long-running deployments. By nature, WSNs are resource-constrained, fragile, complex to analyse, and failure-prone. Naturally, with the growing number of installations, it is becoming fundamental to efficiently diagnose failures as soon as possible, in order to deal with the underlying causes. In accordance with this, from 2005 onwards, the offer of diagnostic tools has been increasing, as the other base technologies (e.g. networking, operating system, localisation, synchronisation) become reasonably stable. The purpose of this survey is to provide an overview of existing post-deployment WSN diagnostic tools, by briefly presenting their functionality, architecture and constraints, in order to enable a basic understating of each tool. The survey also includes a multi-dimensional comparative analysis of the various tools, based on a proposed classification scheme and evaluation criteria, as well as an identification of the main open research issues. Although the number of diagnostic tools is high and considerable work has been done in this area, we conclude that there are still several challenges concerning post-deployment WSN

A. Rodrigues (✉)
Centro de Informática e Sistemas, Universidade de Coimbra, Pólo 2, 3030-290 Coimbra, Portugal
e-mail: arod@dei.uc.pt

T. Camilo
Departamento de Engenharia de Computadores, Instituto Politécnico da Guarda, Av^{a} Dr. Francisco Sá Carneiro, no. 50, 6300-559 Guarda, Portugal
e-mail: tandre@dei.uc.pt

J. S. Silva · F. Boavida
Departamento de Engenharia Informática, Universidade de Coimbra, Pólo 2, 3030-290 Coimbra, Portugal
e-mail: sasilva@dei.uc.pt

F. Boavida
e-mail: boavida@uc.pt

Published online: 06 June 2012

diagnostic tools, regarding scope, flexibility, generality, mobility and security. Moreover, there is a need for mature, native diagnostic-oriented functionality in WSN platforms and operating systems.

Keywords Sensor networks · Sensor diagnosing · Testing and debugging · Sensor failure · Embedded debugging

1 Introduction

As opposed to what happens in lab environments, real-life deployment of Wireless Sensor Networks (WSN) brings out a whole range of problems that are difficult to anticipate as a result of harsh environmental conditions, mechanical aggression, communications interference, sensing interference, increased energy consumption, and unexpected node mobility, among many other causes, either natural or malicious.

Communication problems due to link failures, packet corruption or network congestion are common in wireless networks and tend to manifest themselves more frequently in fragile, resource-constrained, wireless multi-hop networks such as WSNs. Although, in most cases, these problems can be mitigated by adequate design and implementation, in the case of WSNs this may not be feasible. In addition, design, implementation, and operation errors can become apparent in real world operations and their effects can be further amplified by the use of incorrect/incomplete models of the phenomenon whose behaviour is being monitored (for instance, incorrect mobility models, inadequate maximum number of objects to track, etc.).

Thus, as failures are unavoidable, it is important to diagnose them as quickly and effectively as possible in order to deal with the underlying causes. In case the faults are restricted to individual nodes, an execution log plus the internal state of the faulty nodes can be enough. On the other hand, WSNs are distributed systems made of cooperating nodes and, as a result, node faults can impact the behaviour of the whole network, affecting services such as distributed sensing, communications, localisation, and timing, or even have an impact on the global WSN energy consumption. Consequently, in most cases, in order to diagnose a fault it is necessary to analyse combined information from multiple nodes.

Evidence gathered from long running deployments [1] has shown the importance of post-deployment WSNs diagnostic tools, due to the very fragile, complex and failure-prone nature of this type of network. Nevertheless, diagnosing failures in deployed WSNs is a difficult task because it implies analysing distributed behaviour and, at the same time, using as little sensor node resources as possible to acquire the relevant information (e.g., traffic traces, code execution, state, or events).

In later years, the research community has addressed this issue and several ideas have been put forward. Some of these ideas have further been used in tools that were made available to the community. This is a fast evolving field that is expected to continue to be an important research area, not only because the visibility, applicability, and deployment rate of WSNs is growing at a considerable rate, but also because the utility of the existing diagnostic tools is still limited. In spite of this, several relevant contributions are available and, thus, it is fundamental to compile, systematically analyse and describe them, so that the research community can better assess the state-of-the-art and, based on it, develop the next generation of WSN diagnostic tools.

The scope of this survey is on diagnostic tools for WSNs as a whole, which can then be used in post deployment scenarios. This can be considered part of network management [2], more specifically related to performance and fault management. Nevertheless, in this survey we will not specifically address tools for diagnosing several very narrow classes of problems, such as individual sensor faults, hardware malfunction, or security-related anomalies. Examples of such tools are:

- faulty sensors—a recent survey [3] details common sensor faults in deployed WSNs, reviews the fault detection literature, and presents a set of features useful for detecting and diagnosing faults; [4] surveys the outlier detection techniques applicable to WSNs;
- hardware malfunction—in [5] a low-energy software-based self-test technique to enable a post-deployment node hardware test of CPU, RAM, flash and radio (i.e. transmitted power, receiver sensitivity and adjacent/alternate channel rejection) on an MSP430/CC2420 platform is described; in [6] a self-diagnosis mechanism based on accelerometer readings is proposed in order to assess node damage probability from external impact;
- security-related anomalies—these are surveyed in [7], where the types of security attacks and detection techniques are identified and summarised.

Prior to WSN deployment, as part of the application development process, several tools are useful in helping to create applications that behave according to their respective requirements and specifications. Although these types of tools fall outside the scope of this survey, it is useful to briefly mention some of them:

- Simulators (e.g. TOSSIM [8], COOJA [9], both specifically developed for WSN simulation) are valuable tools for debugging and scalability assessment, saving costs and allowing for great flexibility. Nevertheless, they are as good as the models they support. Even carefully constructed models have limitations, as it is extremely hard, if not unfeasible, to capture the frailty and sensitivity of WSN nodes, especially when they are subject to real-life environmental conditions.
- Validation tools like KleeNet [10] target any WSN application bugs that result from complex interactions of multiple nodes, non deterministic events, and unpredictable inputs, via high-coverage testing supported on an application execution of symbolic input; T-Check [11] aims at discovering errors in WSN applications running on TinyOS (simulated in TOSSIM) by using random walks and explicit state model checking.
- Hardware debuggers and emulators (e.g. AVR JTAG ICE [12], MSP430 FET [13]) target the debugging of programs running on real hardware. By requiring additional hardware, they are not practical to use on the field and they do not scale. Also, they do not allow for easily understanding distributed behaviour.

- Software emulators (e.g., MSPSIM [14], an instruction level emulator for MSP430 microprocessors that also supports complete platforms) do not have the cost and deployment impact of hardware emulators, but are also limited in their capabilities to reproduce WSN field behaviour, even when integrated in environments such as the ones presented in the next bullet.
- Integrated environments like EmStar [15], a tool to simulate/emulate heterogeneous systems with support for integration and interoperability between motes and micro servers, which offers a unified set of tools to capture, view, and analyze real-time debugging information from simulations, emulations, and deployments; Avrora [16], a simulation, emulation, and analysis framework for an Atmel/Mica2 platform; or COOJA/MSPSIM [17] that offers a heterogeneous simulator/emulator environment that also enables interoperability testing and power profiling.
- Test-beds enable developers to run WSN applications in real hardware and are the closest to reality as one can get in a lab. Most are private, but there are publicly available ones that enable WSN application evaluation by allowing the uploading of executable code and retrieval of results (e.g. MoteLab [18], 190 nodes, Harvard University; Kansei [19], 210 nodes, Ohio State University; Indryia [20], 127 nodes, University of Singapore; and TWIST [21], 204 nodes, TUBerlin).

Given the specific scope of this survey—post-deployment WSN diagnostic tools—two things should be highlighted: on one side, this text is not intended to replace existing surveys on WSNs and, on the other side, to the best of the authors' knowledge, it is the first systematic and comprehensive approach to post-deployment WSN diagnostic tools in the literature. Therefore, it contributes to a better understanding of the underlying key ideas, to a better knowledge of concrete tools, their applicability and limitations, as well as to a clear view on the main trends and research issues in the area. This is accomplished by identifying the types of problems that justify the work in this area (Sect. 1), presenting a tool classification proposal and defining a set of criteria that can be used to evaluate and compare the various existing approaches (Sect. 2), describing a representative set of existing tools, encompassing requirements, functionality, architectural choices, available performance data, and a list of pros and cons (Sect. 3), presenting a comparative assessment of each of the presented tools based on the developed criteria (Sect. 4) and, last but not least, identifying open research challenges (Sect. 5). Conclusions and directions for further work are presented in Sect. 6.

2 Classification and Evaluation Criteria

Tools that assist in the diagnosis of WSN failures are important assets. Before we proceed to review a representative set of the available tools for debugging deployed WSNs, it is essential to classify them, so that this classification can serve as a framework for their subsequent evaluation.

There are several classification schemes in the literature, normally proposed in papers presenting a specific tool. Consequently, none of these schemes is comprehensive and/or general enough, their scope being limited to a specific area addressed by a relatively small number of tools. In [22], debugging solutions are categorized into source-level debugging, query-level debugging, and decision-tree debugging, but this classification has become out-dated with the introduction of new machine learning techniques. Ref. [23] groups tools into active monitoring or passive (sniffer-based) monitoring, but this classification is too general. Ref. [24] classifies tools into watchers (making using of external hardware), loggers, and interactive ones, based on how each node contributes to the diagnosing process. It is a useful approach, but it does not prevent some border problems as some tools operate in several of these modes. Ref. [25] groups tools into automatic debugging, network visibility, and pre-deployment coverage analysis, which, although it is a possible approach, it is not comprehensive enough.

Looking at the various classification schemes, it is apparent that there is increasing difficulty in grouping tools based on any single criteria. This is due to the fact that the scope of the available tools is becoming larger, as they support multiple operating modes, automation levels, network visibility and resource usage levels, among many other features.

Having this in mind, we propose a multi-dimensional classification and evaluation framework, based on three major vectors, according to Fig. 1: architectural aspects, functional aspects and dynamic aspects. These are explained in the following sub-sections.

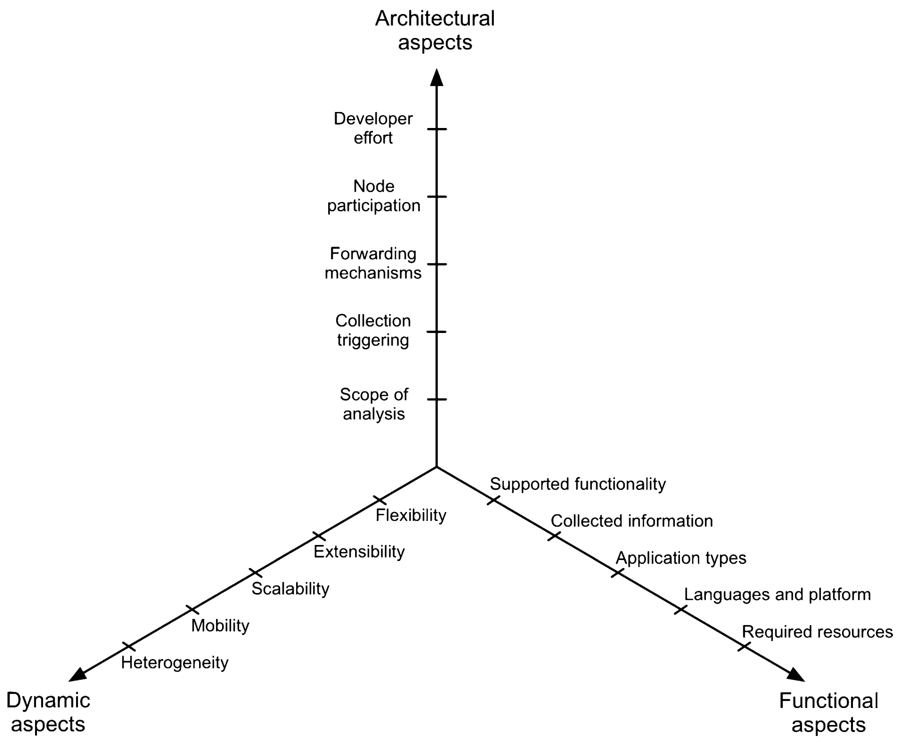


Fig. 1 Classification and evaluation dimensions

2.1 Architectural Aspects

Architectural options determine fundamental aspects of the tools' behaviour, functionality and usability. In what concerns architecture, the following aspects have been considered: scope of analysis, collection triggering, forwarding mechanisms, node participation and developer effort.

2.1.1 Scope of Analysis

Several strategies can be used in terms of scope of analysis. One possibility is to try and diagnose problems looking at the WSN as a whole, by combining information from multiples nodes. Typically, this is done at a management unit (**MU**), usually connected to the sink, either by using simple rules based on collected information or more sophisticated machine learning techniques to help diagnose the problems. Another possibility is to rely on cooperation between nodes to diagnose problems in a distributed way, possibly with some kind of traffic reduction by aggregation of information at some nodes. Finally, another possibility is to monitor and debug individual nodes' behaviour, without considering information from others nodes. In addition, diagnosing can either be performed online or offline (post-mortem).

2.1.2 Collection Triggering

Forwarding information to a management unit may imply high energy costs. In order to minimize these costs there are two basic strategies that, in some cases, can be combined: query-based—the user asks for specific information at run-time; or event reporting—nodes report events in an unsolicited manner although, in some cases, the user can configure the level and periodicity of the reporting. Reported events can be diverse, ranging from low-level events (e.g., a code trace) to high-level events (e.g., the result of a predicate evaluation). Flash-based logging is typically used as a temporary solution before applying one of the previously mentioned mechanisms. Nevertheless, it can also be used to save information that will be manually collected at a later point in time.

2.1.3 Forwarding Mechanisms

The communication channel impacts WSN resource consumption and, ultimately, its applications' behaviour. In what concerns forwarding of information, some strategies do not interfere with the global WSN by using dedicated sniffer nodes, while others require a secondary communications channel on the monitored node. The latter option leads to lower reliability and lower costs than the former, but has a higher impact on energy consumption. In spite of this, it does not consume main application bandwidth. Nevertheless, the most used approach is to use the node's main application communication channel, as this does not require any extra hardware. The price to pay is the impact of the monitoring traffic on the node resources. In what concerns the protocol stack, some approaches use the target application network protocols, others work on top of MAC, and others use their own complete stack, and are thus lighter and/or more robust to failures of the main application stack.

2.1.4 Node Participation

There are several approaches to WSN debugging in what concerns the nodes' contribution to the process. They can be classified into:

- Node passive participation—all the monitoring is done by sniffer nodes, either networked or stand-alone. This provides non-interference with the monitored WSN at the expense of additional hardware, leads to a good understanding of communication problems, but does not allow the detection of failures that do not affect the transmitted data.
- Node logging—the application code is instrumented to log the desired information. Specific techniques differ, depending on the developer effort, on the consumed resources, on the intended robustness, and on the impact on the WSN application. This approach has the advantage of giving access to the run-time node status. The drawback is added resource consumption and the dependence on the WSN operating condition.
- Node management (state modification, function calls, code debugging, code updating)—in relation to the previous approach, this approach adds the options to change state and to execute several operations on the node as, for instance, running debugging code or updating it. This leads to greater management flexibility, at the expense of more resources, primarily ROM and RAM, but also bandwidth and energy, depending on the usage pattern.

2.1.5 Developer's Effort

In what concerns the developer's effort, three categories have been considered: (1) Light—there is no need for code instrumentation and the management station application is almost plug and play; (2) Medium—simple code instrumentation is required and the management station application may require some configuration; (3) High—complex code instrumentation is required and/or the management station application requires the writing of scripts.

2.2 Functional Aspects

Functional aspects basically determine what can be done with the tool and with what resources. The following aspects have been considered: supported functionality, collected information, application types, languages and platforms, and required resources.

2.2.1 Supported Functionality

Each tool offers a set of functions. In this topic we identify the main functionality and give information on the technique supporting it.

2.2.2 Collected Information

The collected information is, in general, of a diverse nature, and can be restricted by the way each node contributes to the management process. For instance, tools that do not operate inside the node do not have access to the node status, such as the energy level or memory usage.

2.2.3 Applications Types

The WSN application type has implications on the applicability of the tools. For instance, tools that only rely on traffic information are dependent on the traffic pattern and are limited by the scope of the collected information. Monitoring applications frequently generate traffic, and this can be used to detect anomalies by resorting to techniques that rely on traffic observation. Event driven applications can be running for long periods of time without generating traffic, until some event occurs. Nevertheless, diagnosing networks by just observing their traffic is not enough because, for example, this does not allow the ability to easily distinguishing a "no news" situation from a "malfunctioning node" situation. Tools for query-driven applications depend on the application query pattern. If enough traffic is generated, the same techniques that are applicable to monitoring applications can be used. Otherwise, techniques that actively generate reports should be employed.

2.2.4 Languages and Platforms

This topic identifies what are the tool's supported node programming languages (e.g. nesC, C), operating systems (e.g. TinyOS, Contiki, MantisOS, LiteOS, freeRTOS), and platforms (e.g. MicaZ, TelosB, Iris).

2.2.5 Required Resources

Different tools can have different requirements in terms of the target sensor node platform (e.g. on ROM/RAM, energy, bandwidth, CPU), management station hardware, and other hardware (e.g. sniffer nodes).

2.3 Dynamic Aspects

A set of features determine the adaptability and applicability of a tool to different conditions. These were grouped under the 'dynamic aspects' category and include flexibility, extensibility, scalability, mobility support and heterogeneity.

Flexibility determines the ability to access state and/or execute code that was not anticipated before the WSN deployment. This is important because sometimes it is not easy to deploy additional debugging functionally on a running WSN.

Extensibility is the feature that enables adding more capabilities to the tool's basic capability set. This not only enables coping with more types of problems but also impacts the number of tools in the developer toolbox.

Scalability relates to the ability of the tool to support a bigger and more complex WSN (i.e. more nodes, more traffic) and at the same time, not requiring proportionally more resources, including human resources to analyse their outputs.

Mobility support provides the ability to deal with node mobility, which is an essential feature in some WSNs.

Heterogeneity relates to the ability to support scenarios where there are nodes with diverse hardware or software. This can range from minor software version differences to differences in hardware.

3 Tool Description

Before presenting a comparative assessment of the various WSN diagnostic tools according to the classification and evaluation criteria described in Sect. 2, in this section we provide an overview of each tool's functionality, requirements, architectural choices, available performance data, positive aspects and drawbacks.

When deciding in what order to present and describe the tools, the option was to group the tools in the following way:

- Traffic-based tools—these tools mostly operate on network traffic and try to diagnose a failure by using data analysis techniques;
- Node state tools—these tools use on-node components that collect and output each node's information, or enable interactive node debugging, in order to help the user analyse a failure;
- Global state tools—these tools combine information from multiple nodes to discover and/or diagnose failures, usually by identifying an inconsistent or invalid global state.

Inside each group we used the ascending publication year to help assess the tool development context and evolution trends. Table 1 identifies the described tools and the groups they belong to. Tools marked with a bold X are described in the Global States Tools section. Note that this grouping is for the purpose of presentation only. In addition, the table identifies the main purpose of each of the tools.

Before proceeding to the description of the tools, it should be noted that there are tools/techniques related to the surveyed ones that will not be presented and evaluated because these are tools for improving safe operations and not for diagnosing. Examples of such tools are Safe TinyOS [26] and Interface Contracts [27]. The former enforces type and memory safety at run time, supports safety break reports, and protects TinyOS applications from array and pointer errors, detecting them before they corrupt the memory. The latter checks proper interface use in TinyOS applications by allowing to state pre/post conditions on component interfaces use and reporting on their violations.

Table 1 Tool grouping

Tool name	Traffic-based	Node state	Global state	Main purpose
Sympathy [28]	X			Failure detection (e.g., node crash, node reboot, no neighbours, no route, bad path to node) and root cause identification and localization
SNIF [29, 30]	X			Detection of node problems (e.g. unexpected node reboot), link problems (e.g. isolated node), path problems (e.g. orphaned node) and global problems (e.g. partitioned network)
SNTS [31]	X			Automatic failure analysis with focus on environment-related problems, namely node connectivity, communication errors, and sensing range
LiveNet [23]	X			Providing information on the dynamics of a WSN (enables node traffic rate analysis, hotspot identification, network topology discovery, and routing paths determination)
Nucleos [33]		X		Enabling access to each node's internal state in order to support monitoring of running applications. Can operate in a query-based mode or in a node-generated event mode
EnviroLog [34]		X		Distributed tool, executing in WSN nodes and allowing to record and replay asynchronous events, thus providing repeatability of experimental testing
Marionette [36]		X		Enabling limited debugging of WSN applications, using a management workstation. The tool has the capability to call functions and read/write variables on a remote WSN node
Clairvoyant [37]		X		Source-level debugger for WSNs that supports post-deployment node debugging without requiring application source code modification
NodeMD [38]		X		Managing WSN nodes in the presence of software faults (i.e. stack overflow, deadlock, livelock, and application specific faults)
L-SNMS [40]		X		Management tool that supports node configuration, status querying, event reporting, and remote reprogramming
LIS [24]		X		Framework designed to enable the monitoring of WSN applications execution, mainly providing execution trace generation and state logging
Memento [42]			X	WSN failure detection and symptoms alerting service using user provided failure detection functions
DT [35]	X	X	**X**	Debugging of WSN applications in a flexible and efficient way. The tool features a declarative programming language with which multiple debugging techniques can be implemented
Dustminer [44]	X		**X**	Detection and identification of failure root causes that result from complex interactions between different components (e.g., MAC performance problem). Based on machine-learning techniques
MEGS [45]			X	Definition and further evaluation of assertions and predicates, based on the collected WSN state, to pinpoint the localization of anomalous behaviour (e.g. routing cycle detection)

Table 1 continued

Tool name	Traffic-based	Node state	Global state	Main purpose
Wringer [22, 46]			X	Debugging, supported on predicates specification and monitoring and the definition of actions that should be taken when predicates are met
MDB [47]			X	Debugger for the MacroLab programming system that targets the debugging of logical, configuration, and synchronization errors
PDA [49, 50]	X		**X**	Failure detection (e.g. functional- and timing-related bugs) and hinting to their causes. Enables to formulating assertions in order to verify hypotheses about distributed WSN state

3.1 Traffic Based Tools

3.1.1 Sympathy

The tool aims to provide failure detection and root cause identification and localization. The main idea behind Sympathy [28] is that a properly working monitoring WSN application should produce a continuous flow of data. When the tool detects a node generating less traffic than is expected, a failure diagnostic is triggered.

The tool's architecture is based on a node component that answers sink queries and also transmits metrics, and a set of sink components responsible for collecting metrics, detecting failures, identifying their root causes, localizing their sources, and notifying the user.

Metrics can be calculated at the sink from application traffic, from traffic captured by a sniffer at the sink, or from the sink application itself. Optionally active metrics can also be generated at the Sympathy node component, thus extending the use of Sympathy beyond its main monitoring applications target.

Three groups of metrics are available: connectivity metrics that collect each node's routing table and neighbour list, as well as the sink neighbour list; flow metrics that collect transmitted and received packet information for each application component at each node and at the sink, in addition to the last timestamp at the sink; and node metrics that include node uptime, node counters, and good and bad packet counters.

Using the collected metrics, Sympathy tries to identify the failure cause (e.g., node crash, node reboot, no neighbours, no route, bad path to node), and to pinpoint its source with the help of a decision tree, based on some heuristics and lessons learnt from previous deployments. The tool distinguishes between three possible localized sources for a node failure: self (the problem is in the node), path (the problem is a failure in the path from the node to the sink), and sink (the problem is in the sink).

Sympathy also distinguishes between primary failures (localized on the node itself) and secondary failures. Primary failures are highlighted since they cannot be traced to any other cause and can potentially be the cause of other failures.

Evaluation results support the stated claim that Sympathy accurately detects failures with low network overhead and low latency. There is a trade-off between detection latency (time between failure injection and user notification by the system) and accuracy (number of primary failures identification). The impact on detection latency of running a different routing protocol, varying the traffic pattern, or having multiple simultaneous failures is quite small.

In what concerns Sympathy's overhead:

- the impact of increasing the active metric period (reducing the traffic) on the detection latency depends on the failure type;
- the tool takes 47 bytes of RAM and 1,558 bytes of ROM; adding the ping-about command increases the RAM usage to 97 bytes and the program memory to 1900 bytes;
- the system is extendable with more metrics.

Sympathy's main advantages include the support for single-hop or multi-hop tree-based routing, no need for modifications to the application code because it collects the information from TinyOS (with the exception of traffic information collection on the individual application components), the use of the same channel as the node's main application and, last but not least, a small impact on the target node flash and RAM.

On the other hand, Sympathy was not developed for event-driven applications, although it can support them with some resource waste, by explicitly querying a node or by using active metrics. In addition, the support of new, user-defined metrics requires node reprogramming.

3.1.2 SNIF

The main idea behind Sensor Network Inspection Framework (SNIF) [29, 30] is to deploy a monitoring WSN in the same scenario as the monitored network. Each node will have two radios, one to monitor the traffic and another one to forward the information to a base station. In this way there will be no impact on the functioning of the monitored network. According to the authors, with this technique it is possible to detect node problems (e.g. unexpected node reboot), link problems (e.g. isolated node), path problems (e.g. orphaned node) and global problems (e.g. partitioned network).

The system architecture is composed of four major blocks: radio listening, packet decoding, data stream processing, and root cause analysis. These components require the specification of several physical layer parameters, a packet description, an operator graph for supporting data stream problem detection, and a decision tree for primary problem identification.

The tool's major advantage is the non-interference with the monitored WSN, as it has no impact on the application code, it does not use sensor network resources, inspection is separated from the application, and the tool deployment and removal does not affect the functioning of the monitored network.

The disadvantages include the need for resource duplication—a new, separate network is needed—and the fact that it does not detect problems that do not have impact on the communications.

3.1.3 SNTS

Sensor Network Troubleshooting Suite (SNTS) [31] performs automated failure diagnoses in a WSN. The main idea is to collect traffic in a non-invasive way, classify segments of this traffic as normal or abnormal using developer-provided knowledge, and subsequently use machine-learning techniques to build a rule-based model. The model is expressed as a set of conditions over metrics derived from the captured traffic, and should correctly classify a high traffic percentage.

The tool's architecture has two main components: data collection and data analysis.

The collection component uses debugging nodes that passively collect the traffic information and save it on local memory. At the end of the experiment these nodes forward the information to a workstation for subsequent data analysis.

The data analysis component extracts the relevant information from the merged logs received from the nodes, and further splits these data into segments. Each segment will be classified as normal or anomalous using the user-provided decision function. For each traffic group, metrics are generated and used for training a machine-learning algorithm, so that it can infer the rules for segment classification. The rules for classifying segments as anomalous should provide clues to help the developer understand the failure causes.

As an example on how to use SNTS, the authors described its application to the identification of an occasional duplication ID bug on EnviroTrack [32]. SNTS led to the conclusion that the problem derived from design assumptions that did not apply in the field (namely, 100 % sensory coverage and groups with more than one member), and also helped in tuning the algorithm.

The following are the main SNTS advantages: it enables automatic failure analysis with a focus on environment-related problems, namely node connectivity, communication errors, and sensing range that impact distributed component interaction; it does not depend on the monitored WSN node software, the only requirements being a compatible sniffer radio, a packet description, and a decision function; data collection and analysis components are independent from the monitored WSN application and the tool is applicable to various scenarios; deployment, use, and removal of the tool does not impact the monitored network.

The main disadvantages are the following: being an offline tool, with the aim of avoiding interference with the WSN, it cannot be used as a continuous monitoring solution; as it is sniffer-based, it requires the existence of traffic, which can be difficult to obtain in query or event-based applications; as it requires sniffer nodes, it leads to resource duplication; the tool was not designed to detect failures that do not impact traffic in such a way that enables its classification into normal or anomalous.

3.1.4 LiveNet

The main objective of LiveNet [23] is to provide information on the dynamics of a deployed WSN. Specifically, it enables node traffic rate analysis, hotspot identification, network topology discovery, and routing paths determination.

The tool uses sniffer nodes that are co-deployed with the monitored network. These nodes record packet traces observed on the radio channel. Traces from all the nodes are then collected and merged into a single trace that can be further analyzed. The merging process is done offline, on a dedicated machine, due to its complexity and high processing and storage requirements.

LiveNet enables the following types of analysis on the merged packet traces:

- Coverage—used to estimate the coverage of the LiveNet sniffers.
- Traffic rate and hotspot identification—allows to measure the traffic generated by each node and to identify the nodes that are in the top list of packet origins or destinations.
- Topology—used to discover the network topology.
- Routing path inference—a path inference algorithm was developed to determine the most probable path between any two nodes at any given point in time.

The tool validation was done in an indoor testbed with 188 TmoteSky nodes, half of them used as sniffers and the other half constituting the monitored network. It was also done in an outdoor scenario simulating a bus accident where sensors read victims' vital signs data that were subsequently forwarded, through the WSN, to a command post.

In the indoor testbed the measured results were the following:

- Sniffer reception rate—maximum 100 packets per second—100 % reception ratio.
- Merge performance—between 301 s (for 2 traces) and 2,859 s (for 25 traces). For a small number of traces to merge, the merging could be done in real-time (each trace represents 1,000 s of packet data).
- Merge accuracy—0.14 % duplicate and 0.005 % out-of-order packets were found in the merged trace.
- Coverage—90 % coverage was achieved with 27 sniffers. The coverage ratio was topology-dependent.
- Topology reconstruction (in a scenario where 10 nodes forwarded data along a spanning tree to a sink)—16 % of the links were discovered using unicast packets. By using the neighbour table beacons, it was possible to discover 93 % of the links.
- Path inference—in a scenario where 2 nodes communicated using a predefined set of paths that changed every 300 s (used as ground truth), the algorithm correctly identified these paths.

The disaster scenario evaluation provided information on how the tool behaved in the field. Interesting results included: detecting a potential routing protocol bug that manifested itself as a storm of corrupted routing maintenance packets; coverage results were impacted by having mobile nodes; network topology was chaotic because of node mobility/reboots; the repeaters were identified as hotspots; the path inference algorithm identified most of the paths; it was possible to identify the reasons (i.e. packet loss, node failures/reboots, query timeout) for having a 20 % average data yield during the experiment.

The ability to analyse routing, congestion, and topology problems is among the main advantages of this tool. In addition, deployment and removal of the tool does not impact the monitored network. The main disadvantages of the tool are the following: the tool was not designed to pinpoint problems in real-time; it only works on packets received by the TinyOS Active Messages Layer; it requires extra hardware (sniffer nodes); and, last but not least, the tool was not designed to detect problems that do not have impact on network traffic.

3.2 Node State Tools

3.2.1 Nucleos

The basic idea behind Nucleos [33] is to enable wireless access to each node's internal state (TinyOS nesC components data structures) in order to support monitoring of running applications. Nucleos can operate in a query-based mode or in a node-generated event mode.

The Nucleos network architecture was designed to support two different traffic patterns, namely collection of WSN health data, and dissemination of management commands and queries. The query and event subsystems can optionally utilize the main application network component, thus saving memory at the expense of reduced robustness.

In order to use the query functionality, the application developer has to choose which attributes (e.g. variables) to export. These will be the ones that can be queried when the system is running. For each of these attributes, the developer has to write a piece of code that will read and provide the attribute value to the Nucleos runtime query system that will further send it to the base station as the result of a specific query.

The aim of the event logging functionality is to give the developer the means to insert debugging calls in the application code. When the developer wants to retrieve the event log he/she should send a remote playback command to the node set he/she wants to read from. The system supports additional features, such as controlling the remote playback traffic flow, enabling real-time retrieval, or retrieving messages from specific components.

Based on the communication, query, and event logging subsystems, a set of management components were developed to support node identification, remote sleep/wakeup, physical parameters access, task queue statistics tracking, reboot cause identification and statistics, an instrumented radio stack that exposes a rich set of queryable counters, and node binding mechanisms to limit the propagation of commands and data.

Information on the ROM usage is not available in the literature, and details on RAM consumption depend on the chosen configuration. For instance, support for 4 active queries, separate buffers for a basic identification scheme and the input part of the event system, several instrumented components, the full Nucleos components, and the null application, account for 1,281 bytes.

The instrumented network stack, in conjunction with the radio duty-cycle control, can provide real-time data to estimate energy consumption and assist in failure prediction. In addition, the scheduler/sensors drivers' instrumentation allows an estimation of CPU and sensor energy consumption. Moreover, persistent data logging can be valuable for post-mortem analysis. Finally, multiple-root data collection, in conjunction with epidemic dissemination, can help constructing a reliable network at deployment time.

Nucleos' main advantages are the following: it can be used with any WSN application type; it is highly flexible in what concerns query and event functionality, providing a set of useful management functions; communications do not depend on the WSN application network protocols, although the developer has the option to use them, thus trading ROM/RAM for reliability; it uses the same communication hardware as the WSN main application.

Nucleos' main disadvantages include: no support for combining and analysing information concerning multiple attributes and/or nodes; the developer has to state at compile time what are the exported attributes and also write their respective support functions; it requires instrumenting the source code with the logging calls; to support detection of new kinds of failures complete node reprogramming is required.

3.2.2 EnviroLog

EnviroLog [34] is a distributed tool that executes in WSN nodes and allows recording and replaying asynchronous events, thus providing repeatability of experimental testing. The tool also enables to record state. The motivation behind EnviroLog was to build a tool for simplifying system evaluation, tuning, and comparability, when working with real deployment data.

The system design is based on the assumption that application data is transferred from producer modules to consumer modules via function calls only, as is the case of TinyOS, the tool's host operating system. In this way, event recording is done by logging all the corresponding function calls in the local memory. When replaying, the consumer modules receive the previously logged data. Record and replay are distributed operations under the control of the developer that activates/deactivates them by sending commands from a workstation-based application.

EnviroLog's architecture is based on three main blocks:

- Preprocessor—supports the specification of the asynchronous events that the developer wants to record/replay, and the variables he/she wants to record/output; this is simply done by introducing special comment instructions (called annotations) in the nesC application code.
- Record and replay—receives control commands (i.e. record, replay, retrieve) from the developer via the stage controller; provides the record function that is introduced in the code by the pre-processor to record events and variables in memory; during the replay mode, reads the memory and issues the events in their original time sequence.
- Stage Controller—enables the communication between the developer on a workstation and the record and replay modules inside the WSN nodes. The available commands enable recording, replaying, and retrieving variables. These

commands can be started, paused, or stopped. In case of retrieval operations, it is possible to specify the node/variable the developer wants. Replay can be done at different speeds.

EnviroLog was validated in Mica2 and XSM hardware platforms, using TinyOS 1.x. A set of benchmarks using standard TinyOS applications was done with the aim of determining the maximum recording period, showing that it supported a 90-min recording with a 10 Hz sampling rate, and 100 % success rate. The overhead impact on the application runtime behaviour was found to be minimal (0.4 ms delay), and replay accuracy was found to have a small average difference from the ground truth (less than 1 ms). According to [35], the required resources are 15,160 bytes of flash memory and 809 bytes of RAM when using EnviroLog to record/replay a read sensor event in the OscilloscopeRF application.

The effectiveness of EnviroLog was demonstrated by using it in conjunction with a surveillance system called Vigilnet, in a 37-node WSN. The record and replay operations were shown to be effective when capturing and reproducing the trajectories and velocities of the targets (people and vehicles). The replay functionality enabled the optimisation of the tracking algorithm without requiring the repetition of the experiments. The retrieval functionality was used to determine the nodes' energy status, in order to decide on node maintenance. By changing the replaying velocity, the algorithm was tested while detecting targets at different speeds.

By enabling the capture of asynchronous events in real scenarios and replay them when needed, EnviroLog eases system evaluation, tuning and comparability. Low or high-level events, at any application layer, can trigger event recording. EnviroLog is easy to use, does not depend on the type of application (although it targets event-based applications), and does not require extra hardware.

The main EnviroLog disadvantages are the following: changing the record/replay configuration requires application code instrumentation, which implies having access to the nodes. In addition, it is specific to TinyOS 1.x on Mica2, or compatible motes.

3.2.3 Marionette

Marionette [36] enables limited debugging of TinyOS-based WSN applications, using a management workstation.

The tool has the capability to call functions and read/write variables on a remote WSN node. To use this functionality, all the developer has to do is to "tag" the functions that will be remotely callable in the application source code, and deploy the application on the WSN. One of the provided diagnosing applications is a remote terminal, through which the user has access to the objects that constitute the running application and the possibility to execute remote code, change variables, and display application information.

More sophisticated tasks such as traffic monitoring, routing algorithm stress testing, and application fine-tuning, can be done by writing small scripts in Python that will be executed at the management station. In these scripts, remote node functions and variables are accessible using almost the same nesC object syntax. Individual nodes or the entire network can be addressed.

As a development tool, Marionette enables the developer to decide on how much of the application will stay on the nodes and how much of it will stay on the management station side. Moreover, this decision can be changed over the development cycle. This allows, for instance, for better testing of the algorithms in an initial development stage and, at a latter stage, to achieve an adequate balance between efficiency (communications' costs) and visibility (access to state) by moving more functionally to the node part of the application.

Marionette's functionality is implemented on a simplified embedded remote procedure call architecture that consumes less than 4 KB of ROM and 153 bytes of RAM. This includes all the supported mechanisms and a minimal set of function calls. Variable read/write functionality does not increase these numbers, although additional tags to enable more functions to be remotely called will increase the used memory in an amount of 100 ROM bytes per remotely accessible function.

Marionette's simplified remote procedure call server uses native data types for network data transmission, native functions definition as interface definitions, does not support threading or queuing, and is added to the target application code. Also added to the target application code is a component that provides the ability to read/write in any memory address. Two communication modes are available, namely local communication and multi-hop, using Drip for query dissemination and Drain to retrieve the answers.

In order to evaluate Marionette, two different perspectives were selected. The first one was to analyze the impact on the application code sizes of several applications. The second approach was to show how simple Python scripts enabled the creation of new functionality for a WSN without requiring node reprogramming. In the first case, evidence was presented supporting the claim that Marionette leads to smaller code size than the one that would be needed to develop code to access the variables/functions, although these savings are not clearly documented.

The main Marionette advantages include: ease of access to node variables, as this does not require developer source code modification; simple remote code execution, just requiring the addition of a "@rpc" tag to the interfaces/functions one wants to access; easy application development, allowing the developer to freely decide about the application architecture; script-based applications can use information from various nodes; it does not depend on the type of WSN application; functionality can be added to an already deployed WSN application by scripting on the management station side, using remote access capabilities; multi-hop or single-hop in-band communications; availability for TinyOS 2.x.

The main Marionette limitations are related to the heavy use of communications, leading to an increase in energy consumption, impact on the network algorithm performance, and latency and packet loss effects on the application. In addition, Marionette is not a full debugging solution, as it does not support breakpoints, watches and traces. Moreover, variable read/write is only done in a task context, and the set of the functions that can be remotely called is fixed at compile time.

3.2.4 Clairvoyant

Clairvoyant [37] is a source-level debugger for WSNs that supports post-deployment node debugging without requiring application source code modification, something made possible by the use of dynamic binary code instrumentation.

A developer can perform node debugging by issuing appropriate commands through a terminal window, in Gnu Debugger style. These commands are sent using the main application communication channel. Besides supporting the usual debug commands like *break*, *step*, *watch*, and *backtrace*, and more specific commands that enable reading RAM/flash memory and analysing/call interrupts, it also supports network-wide commands (e.g., *gstop*, *gcontinue*, *gdetach*, *greset* and *gbreak*) in order to simplify the issue of sending multiple single commands and, at the same time, providing a way to synchronise nodes. The system also supports command logging, by allowing the definition of points in the code where variables can be logged (if some condition is verified) to the LEDs, RAM, flash, or radio.

The system architecture is based on a host side component and a component running in the target node.

The host side component supports several debugging units. Each unit reuses Gnu Debugger (GBD) as much as possible for the basic commands, supporting specific commands on a component called Clairvoyant DeBugger (CDB). The developer communicates with the debugger via a terminal interface, much like GDB. Visualization components can be used in order to graphically display the debug information that is forwarded from the sensor network.

Node side components are based on a command interpreter and multiple command executors. They are compiled and loaded in the boot sector of the nodes. After the boot process, this code loads the application and transfers control to it, but it also interfaces with the radio stack in order to detect user debug commands and to communicate.

The current version of Clairvoyant works on a Mica2 (TinyOS) platform and requires 32 KB of flash and 1 KB of RAM. The following summarizes the performance evaluation results:

- Execution speed is impacted only when executing the *watch* and *cond* debug commands. Log commands have a diverse impact, with LEDs and RAM being the ones with less impact.
- The impact on radio communication seems to be acceptable if the user refrains from using too many network-wide commands.
- Dynamic binary code instrumentation puts pressure on the nodes' life due to the fact that there are limits to the number of times the flash can be reprogrammed. Commands like *watch* and *step* can, if not carefully used, have great impact on the flash life.
- Clock and Timer consistency—clock consistency is preserved for most of the debugger commands; for fast logging commands, clock consistency is traded by reduced delay in the execution of these commands.

In [37], the authors describe how the debugger was used in identifying stack overflow and deadlock situations that occurred in a previous version of TinyOS 1.x code.

The main advantages of Clairvoyant include the availability of powerful debugging commands that are easy to learn and use and do not require application source code modification, the use of the same channel as the nodes' application for debug communication, and a minimal impact on node performance.

Clairvoyant's main disadvantages are the 32 KB program memory requirement that limits its applicability, the negative impact on the nodes hardware lifetime if there is heavy use of the *cond* and *watch* commands, and the requirement for the application to use a low-level stack similar to Clairvoyant and not turn off/sleep the radio in a way that disables the debugging option.

3.2.5 NodeMD

This tool [38] aims at managing WSN nodes in the presence of software faults. In order to accomplish its goal, the tool implements mechanisms to detect faults before they disable the system; logs predefined events to help in remote debugging; informs the user about faults; and provides a remote debugging mode after putting the system in a stable state.

The system was developed to be used in conjunction with the multithreading Mantis operating system. This guided the authors to focus on stack overflow, deadlock, livelock, and application specific faults. The system architecture comprises three main subsystems—fault detection, fault notification, and fault diagnosis—succinctly described below.

3.2.5.1 Fault Detection Monitoring the system's health can be accomplished in various ways depending on the fault type.

Stack overflow detection is done by automatically instrumenting the source code with code that checks stack overflow at the entry point of the application and operating system functions.

Deadlock and livelock detection make use of a thread checkpoint mechanism. The basic idea is that in WSN applications the developer has an estimate of a thread execution period. This information is then provided in the application code and the checkpoint mechanism periodically checks each thread condition considering this estimate plus some tolerance.

Application specific faults are supported by the inclusion of an assertion mechanism that the developer can use in order to enable the system to test specific conditions.

3.2.5.2 Fault Notification The system keeps an execution trace of specific system events such as context switches, procedure call and return, hardware interrupts, thread blocks and unblocks, software timer set and unset, thread sleep/wakeup behaviour, and threads creation and exiting. Custom application events can also be defined in order to provide the manager an image of the node execution history.

When a fault is detected the tool freezes the system in order to preserve the state, and only reinitializes the specific software components required to support the notification and diagnosis mechanisms. After that, the system administrator is notified, error cause and event trace are sent, and the system enters a standby mode waiting for the administrator's instructions.

3.2.5.3 Fault Diagnosis In debug mode, the administrator can send commands to obtain additional details (such as system parameters not previously sent, memory dump), fine-tuning the monitoring parameters and restarting the node with the objective of replicating the error and obtaining more information.

The tool's effectiveness was validated in experiments that created stack overflow, deadlock, and livelock conditions. The authors also identified a legacy bug in Mantis by analysing NodeMD-generated traces. In what concerns the impact on ROM and RAM requirements (MicaZ/Mantis platform), the system was evaluated using two different applications, namely blink_led, and FireWxNet [39]. The latter is a relatively complex application that monitors weather conditions in wildfire-prone scenarios. The impact of adding NodeMD to the above applications was an addition of 3,556 and 4,266 bytes of ROM for blink_led and FireWxNet, respectively. There was also an increase of 302 and 292 bytes in RAM usage by blink_led and FireWxNet, respectively.

The log operations are claimed to take between 43 and 79 cycles, values that the authors think are unlikely to change the program execution order; the same applies to the stack overflow detection code. The authors also claim that the majority of WSN applications have abundance of free CPU cycles, so that frequent checking procedures will not affect the overall application timing. Due to the low impact on normal program execution and memory use, the authors claim the tool should not have a substantial impact on battery life. Nevertheless, debugging sessions add on energy expenditure, depending on their duration and communications use.

The following stand out as main advantages of the tool: automation of stack overflow detection, simple mechanism to enable livelock/deadlock detection, independence from the type of applications, good debugging capabilities, and in-band communications.

The main disadvantages/limitations of the tool are its orientation to software faults only, thread-based deadlock and livelock detection only, the lack of detailed information on energy and CPU impact, and the fact that the tool is only supported on Mantis.

3.2.6 L-SNMS

Lightweight Sensor Network Management System (L-SNMS) [40] was developed as a management tool to support node configuration, status querying, event reporting, and remote reprogramming.

L-SNMS is organized around three main components, namely: a remote procedure call mechanism based on Marionette [36] that enables to call configuration functions on the node and to get/set status parameters; a specifically-developed run-time event report mechanism; and a node reprogramming component supported on Deluge [41].

The run-time configurable event report mechanism is based on a node component that offers the developer two interfaces: EventReport and EventConfig. The EventReport interface offers a command that enables the developer to report an event (namely, a self-describing string with variable references, like in a *printf*) and assign a type and priority to it. The EventConfig interface enables run-time changes to event priorities via the remote procedure call mechanism.

In addition to node event reporting and node reprogramming functionality, L-SNMS major difference with respect to Marionette is that, while Marionette offers a scripting language environment to enable interaction with the sensor network, L-SNMS offers a JAVA-based graphical management application that simplifies management work. The management application tools include topology display, real-time event display and configuration, remote parameter query and changing, remote procedure call, remote node reprogramming, and a sink-based sniffer that records communication between the nodes.

The authors state that L-SNMS was tested in a real field experiment to monitor volcano activity, where it was used to monitor/control the network status and to enable geologists to adjust the network focus on certain data types or specifically interesting volcano spots. ROM and RAM requirements for L-SNMS, in the volcano scenario, were 2,924 and 588 bytes, respectively, using MicaZ motes and TinyOS 1.x.

L-SNMS's main advantages are: the graphical management application; a simplified mechanism to enable remote function call (just requiring a "@rpc" tag at the interfaces/functions); no need to produce code to access functions or variables; it does not depend on the type of applications; its very good debugging capability; the support for multi-hop or single-hop in-band communications.

On the other side, L-SNMS's disadvantages are the following: by heavily relying on wireless communications, more traffic is generated to support remote access to variables and functions, with an impact on energy consumption and on network algorithm performance; although event reporting is tuneable in terms of priority and event type, it is not optimized; variable read/write operations can only be performed in task context; the developer has, at compile time, to state what functions will be remotely callable.

3.2.7 LIS

Log Instrumentation Specification (LIS) [24] is a framework designed to enable the monitoring of WSN's applications execution, mainly providing execution trace generation and state logging.

The LIS architecture relies on a PC-based instrumentation engine that modifies the WSN application source code in order to include logging statements, and on a runtime logging module located at the nodes.

In order to use LIS, a developer has to produce an LIS script using a declarative language, describing the logging mechanisms to deploy and their location in the WSN application source code. After that, an instrumentation engine parses the source code and the LIS script and, as a result, produces application source code instrumented with LIS logging function calls. Also added to the code that executes on the node is an LIS support runtime library that supports the node logging function calls and the log data storage and retrieving functionality.

During runtime, execution traces and state are saved on local memory and can also be sent to the sink node using either the Collection Tree Protocol (CTP) or the TinyOS AMSend interface. When the packets arrive at the sink node they can be parsed using a generic LIS parser and the LIS script information to produce meaningful information.

The authors claim the LIS language can be used directly or as an intermediate language supporting reusable high-level tasks definitions. The three uses of high-level tasks are:

- *Region of Interest (ROI) Call Trace Monitoring* The developer can specify a ROI (e.g. one TinyOS subsystem) and the system generates the corresponding LIS script that will create a log of the function calls inside the subsystem;
- *System Health Monitoring* An LIS script that collects information on the WSN health status can be built; e.g. such a script was created to help diagnose CTP problems by collecting protocol execution traces and state;
- *Problem solving* Using LIS to do some high-level debugging by writing small scripts that help pinpoint problems. Even if the problem is not precisely identified, narrowing the region will help when doing source code debugging.

The system was evaluated on TinyOS 2.1 (MicaZ and TelosB) and detailed statistics on program and RAM memory usage were provided. In the case of the MicaZ node, using the CTP protocol, the system consumed 1,798 bytes of program memory and 394 bytes of RAM. This amount did not include 14 bytes for each logging call. Each logging function call consumed CPU cycles and introduced latency. This should be taken into account when logging in time-critical systems. Nevertheless, the authors claim the system enabled them to diagnose timing problems in the CC2420 radio stack.

LIS' advantages are the following: flexible and powerful logging supported by easy-to-develop LIS scripts; it can be used to build more sophisticated analysis systems; developers are not required to modify the application source code; LIS communication uses the same channel as the node main application; a small impact on target applications (small latency and light use of flash and RAM memory); portability, as it is not restricted to applications written in nesC language (it supports any C language based application because it is applied after nesC to C pre-processing is done). The disadvantages include the need for scripting, and requiring node reprogramming whenever new logging functionality is needed.

3.3 Global State Tools

3.3.1 Memento

The goal of Memento [42] is to provide an efficient WSN failure detection and symptoms alerting service.

Network health status is represented as a collection of status bitmaps. Each status bitmap describes the state of all WSN nodes in respect to a health condition (e.g. node being alive, low energy, congestion, etc.).

Memento is architected around two main components: a distributed failure detector and an efficient communication protocol to deliver state summaries.

At each node, a specific code is running in order to individually monitor the node's health. If a health condition change is detected, the local copy of the status bitmap for that condition will change in the bit position corresponding to the node.

The used control delivery mechanism is based on the aggregation, at each parent node, of the child local status bitmaps. The aggregation operator is just a simple bitwise OR. In this way, each status bitmap sent in the direction of the sink summarizes the status of the sub-tree starting at and including the node. This mechanism is repeated periodically. By using caching mechanisms at parent nodes that enable child nodes to refrain from transmitting whenever no health condition has changed, the aggregation mechanism is reported to achieve an 80–90 % bandwidth reduction when compared to standard data collection methods.

The system was implemented on a Mica2/TinyOS platform and tested in a network with 55 nodes. The application protocol was the one used by Memento for reporting the state. In [42] it is reported that the system consumes 400 bytes of RAM.

Memento's advantages include no restrictions on the application type, use of multi-hop tree-based routing with node traffic aggregation and optimization, use of the same communication channel as the nodes' main application, and packet-loss-resilient node failure detection.

As main disadvantages, Memento requires the development of the node failure detection functions needed for health condition diagnosis, meaning that detection of new kinds of failures require node reprogramming. No information is available on the impact of detection functions and of the aggregation mechanism on program memory usage.

3.3.2 DT

Declarative Tracepoints (DT) [35] was designed to support the debugging of WSN applications in a flexible and efficient way. The tool features a declarative programming language with which multiple debugging techniques can be implemented. For example, the tool can be programmed to offer the core functionality of StackGuard [43], Sympathy, EnviroLog, and NodeMD. In addition, DT supports node loading/unloading of debugging engines, each one supporting a debugging technique, enabling the developer to select, at runtime, the debugging technique to use.

DT's architecture includes workstation components to generate each debugging engine (called a tracepoint engine), in order to interact with the nodes and to interpret the traces they generate. On the node side, each tracepoint engine is responsible for supporting a debugging technique.

In order to use DT, a developer has to write a script using the DT-specific TraceSQL declarative language, with the objective of problem debugging. The script can contain configuration statements, variable declarations, use built-in functions and at least a tracepoint declaration. It is targeted at investigating a possible cause of the problem. A tracepoint declaration specifies the place on the application code where to introduce the tracepoint, the action it should accomplish, and an optional condition predicate that enables the conditional execution of the tracepoint action.

A Python-based compiler receives the DT script and the WSN application source code as input and generates a tracepoint engine in C language that will be further compiled and sent to a node to be loaded. The traces generated by the tracepoints are collected, parsed by a trace interpreter at the workstation, and finally outputted for further analysis.

The efficiency of DT was measured according to the following perspectives: time to discover a bug, effort to code a debug technique, flash and RAM consumption, and CPU consumption.

Time to discover a bug The authors evaluated the tool effectiveness for detecting two known bugs in an outdated version of the LiteOS operating system. By applying DT to this version, they concluded that DT was able to detect the bugs in a very short period, ranging from a few hours to 1 day. They also identified new bugs.

Effort to code a debug technique Implementing EnviroLog and NodeMD major functionalities, using DT, enabled the authors to compare the original versions with the new ones. Recreating these tools in DT required more work than using the original versions, due to the fact that each tool's functionality had to be expressed as a small program; nevertheless, the authors claim it is a small price to pay for having a flexible tool.

Flash and RAM consumption Comparing EnviroLog and NodeMD original versions with the same functionality in DT is difficult because, on one side, the paper′s results show DT as more efficient but, on the other side, LiteOS is a multithreading operating system supporting dynamic loading, flash storage and logging mechanism (something quite different from plain TinyOS) and all this functionality is used by DT. The maximum values reported were a consumption of 3 % flash and 10 % RAM on a LiteOS/MicaZ platform. The external flash consumption, when used to save logs, depends on the amount of logging activity.

CPU consumption This is related to the number of tracepoints and their associated actions. For tracepoints with no actions or read/write memory actions, the CPU consumption will be no more than 12 % (in the evaluated scenario with 15 tracepoints). When using file operations on the tracepoint actions (requiring read/write in external flash) the overhead can scale up to 70 %.

DT's main advantages are: it does not depend on the type of application; it does not require application source code modification; it provides flexible development of debugging engines in order to support various debugging techniques; the debugging engines can be loaded/unloaded during runtime; it does not require extra hardware.

DT's main disadvantages are: it requires scripting; porting DT to platforms other than LiteOS/MicaZ may be complex as DT requires multithreading, runtime binary code loading, a flash based file system, and several console commands that facilitate the access to data on the node.

3.3.3 Dustminer

Dustminer [44] is aimed at detecting failures that result from complex interactions between different components. These failures include the ones that do not manifest themselves in any single component and, thus, would not be detected by analysing their individual state. The approach taken by the tool is to record events in log sessions and, in a second phase, to single out sequences of events that could expose the failure reasons, using machine-learning techniques.

Dustminer uses a modular architecture based on three independent layers, namely collection front-ends, data processing middleware, and data analysis back-ends. This flexibility allows addressing various applications and platforms by enabling the developer to use the most adequate combination of collection mechanisms (e.g., packet sniffers, node internal logging, simulation based logging) and data analysis back-ends (e.g., one or more data analysis algorithms, a visualization tool) in order to analyze the failures and identify their root causes.

In what concerns the flexibility to choose the front-end, there are implicit tradeoffs between visibility (inside node logging) and non-interference (passive sniffers), and also between reality (real hardware) and cost (simulators).

The front-end generated log files are sent to the data processing middleware, where the first step is to convert them into a generic data format with the help of a user-supplied format template. The next step is to parse this raw data and extract the meaningful events, using developer-supplied application-specific event descriptions. At this point, each log has been transformed into a sequence of events that must be labelled as "bad" or "good" through a user-provided software module that decides on which label to assign. The final operations carried on by the data processing middleware are converting the labelled sequences of events into the formats required by the data analysis tools used in the data analysis back-end.

The data analysis back-end can use any of several algorithms to identify the causes of failures. In [44] the authors adapted a frequently used discriminative pattern mining algorithm to extracting the sequences of events that led to failures.

The system was evaluated by testing its effectiveness in identifying a LiteOS kernel bug and in discovering a performance degradation problem in a new MAC protocol for TinyOS. In both cases, the tool helped to discover the failure root causes.

Dustminer's support for debugging the MAC problem introduced an overhead of 14,670 bytes of program memory (including TinyOS BlockRead and BlockWrite functions) and 830 bytes of RAM. Each log function added between one line of code and $n + 1$ lines of code (for an event with n parameters). The time required for the analysis was $\sim$6 min in a 2.53 GHz/512 MB RAM machine. The maximum impact of the logging mechanism on the MAC throughput was under 2 %.

The main advantages of Dustminer are its ability to work on various platforms and application types using appropriate collection mechanisms and data analysis algorithms, the ability to trade-off between visibility/non-interference and reality/cost, and the fact that it can be used to debug an already deployed WSN.

On the other hand, the main disadvantages are: being an offline tool, it cannot be used as a monitor; the inability to pinpoint the bugs' location in the application code; it requires the developer to write the module that labels logged data as "bad" or "good".

3.3.4 MEGS

The authors of Monitored External Global State (MEGS) [45] claim that the tool can re-create the global state of a WSN using the debug output sent via a side channel to a workstation by the WSN nodes.

The tool's architecture is based on a node component that outputs the value of variables and information on the reached assertion points. All this information is processed in a workstation in order to generate global performance metrics and to detect invalid WSN state.

In order to support the debug generation, the developer has to instrument the nodes with code that sends the value of specific variables, at least when they change values, via a side channel. These variables are chosen according to the problem being diagnosed.

The tool enables the definition of assertions and predicates (written in JAVA) that are evaluated at the workstation against the re-created WSN state, in order to pinpoint the localization of the anomalous behaviour. For each variable being observed in a node, there is a mirror variable in the workstation.

MEGS can use the collected values to create graphics of aggregate values over time, e.g. to know the average hop count to the sink over time. If the user defines a predicate over the global state, he/she can know when it is true and for how long. Finally, the system allows the definition of assertions that use variables from various nodes and are triggered at defined positions in the application code.

The tool was tested with TOSSIM and also in a testbed. The test case selected to evaluate the tool was routing cycle detection on the TinyOS 1.x MintRoute algorithm.

MEGS' main advantage is the minimal impact on the main application, due to the fact that it uses side-channel communication. The downside of this low impact is radio hardware duplication or restrained mobility if using serial cables between nodes and the workstation. An additional negative aspect is the need for node source code instrumentation and for writing JAVA code in order to support predicate/ assertion evaluation. The authors additionally identify the impact of node synchronization mechanism limitations.

3.3.5 Wringer

Wringer [22, 46] is a lightweight, TinyOS 2-based, rapid-prototyping framework for WSNs that focuses on "predicate-oriented debugging". The system enables the specification and monitoring of predicates (i.e., conditions based on a local or distributed system state) and the definition of actions that should be taken when the predicates are met.

Wringer is based on a lightweight Scheme programming language interpreter located at each node, working as a separate application with its own memory, code management and, at least, requiring access to the MAC layer. If the application already uses a routing protocol and a data dissemination protocol, it will be used. Otherwise, this functionality can be defined in Wringer.

The tool is organized around three building blocks, that are used to support the predicate evaluation functionality: primitives, which can be native or dynamically loaded, and are triggered by certain events; variables, which maintain the nodes' state and may be used by primitives (by parsing application code, Wringer knows how to access the variables in runtime); and triggers, which may be fired on variable value changes, packet reception, timeouts, or function calls.

Node predicate evaluation begins when a trigger fires. The condition verifier is then invoked in order to evaluate the Boolean expression representing the predicate. If the condition is met, the data processor will perform some local processing and/or packet transmission.

Wringer is a proposal that offers local and network-wide debugging facilities supported on a predicate paradigm. In the network-wide case, it is not clear from [46] how the system supports distributed predicates or the "collaborative logging" service that enables a node to replicate other node's state.

Wringer's advantages include not requiring application code modification, using the same communication channel as the WSN application, and ease of specification of predicates describing network state conditions that can be deployed during WSN operation. The main disadvantages are related to the fact that it is an unfinished tool with an unknown impact on node resources, node synchronization and message loss, and lack of a predicate library to help in identifying classes of problems.

3.3.6 MDB

Macrodebugging (MDB) [47] is a post-mortem debugger for the MacroLab programming system [48]. MacroLab provides a Matlab-based vector syntax language where all the node information is presented to the user in vector form and the user can use the standard set of Matlab vector operations to develop a macro-program.

Macro-programs are programs that are made using high-level programming abstractions. The need for MDB derives from the fact that although available macro-programming tools simplify program creation, they do not make debugging easier as one still has to debug at the node level. MDB targets the debugging of logical, configuration and synchronization errors.

The MacroLab system compiles a macro-program written in its high-level language down to micro-programs in the nesC language that will be further compiled and run at individual nodes. System state recreation in MDB is supported by logging data traces to node memory. Node logging functionality is automatically added by the MacroLab compiler. After program execution, the logs can be retrieved using TinyOS CTP in order to enable further offline debugging.

When a macro-program is running, probably not all the nodes are executing the same code at the same time. To cope with this, MDB offers two distinct ways of visualizing the WSN state: logical views that allow the user to see a distributed state at a specific macro-program operation; and temporal views that allow the user to view the distributed state at a specific point in time. These views are the very basis of the system, supporting common debugging commands such as *breakpoint*, *step*, *continue* (in the logical view interface), as well as *jump* and *step* (in the temporal view interface).

In addition to state views, MDB also provides historical search and hypothetical changes functionality. These do not require code deployment, as they operate on previously collected system state.

The historical search allows the user to specify complex queries over the distributed system state history. This can, for instance, be used to investigate the occurrence of bugs or to analyze the distributed system state data over time.

Hypothetical changes simulate the impact of program modifications that can be useful, for instance, to verify if a code modification can correct a specific bug. Hypothetical changes can also enable testing the impact of adding processing and data synchronization primitives to a macro-program.

Logging data has an impact on energy consumption. However, removing the logging code from production systems can open the door to heisenbugs. The found solution was to use MDB Lite, which does not log state but preserves memory use pattern and program timing.

The tool's evaluation—carried out for common WSN applications such as Surge, OTA, and Acoustic Monitoring—proved that logging data is more efficient than logging events. This was done by comparing the number of interrupts generated per 100 data states written to flash. The result was that the number of data state update interrupts was lower than logging event interrupts, the difference being between 14–32 times less interrupts.

In what concerns resource consumption, the maximum RAM requirement was 304 bytes (OTA). The TelosB 1 MB flash enabled logging a maximum of 9 h (Surge) and a minimum of 1 h (OTA). The impact of the logging code on CPU execution time was less than 0.5 %. Energy consumption increased by 30 % when using MDB with OTA in a low-power-listening 1-s interval configuration. When MDB Lite was used in the same configuration, this overhead was 0.9 %. Finally, the authors do not provide statistics on MDB's impact on program size.

As main advantages of MDB, the authors refer its ability to perform debugging at a very high-level with no need for code instrumentation nor constraints on the application type. It also does not require extra hardware. On the other hand, the disadvantages include the fact that it is a post mortem debugger, it is restricted to applications written in MacroLab, and it lacks information on flash memory overhead.

3.3.7 PDA

The goal of Passive Distributed Assertions (PDA) [49, 50] is to aid in failure detection and in providing hints on their causes. PDA enables one to formulate assertions in a declarative language in order to verify hypotheses about the distributed WSN state.

Assertions are evaluated at a backend, using state information previously outputted by the nodes. To support the output of node state information, at each node there is a runtime module that outputs assertions information and attribute values periodically or when there is any change.

In order to use PDA, the developer has to insert the distributed assertions to be evaluated in the application source code. An assertion takes the form of a Boolean expression over node attributes (e.g. local node variables or attributes in other nodes) that is evaluated when the program flow reaches it. If the assertion fails, the developer is notified and has access to the assertion source code location and to the corresponding attribute values.

In order to collect the information outputted by the nodes, the tool supports several strategies: in-band sending of information, logging at each node, logging at sniffer nodes, using a wireless sniffer network, using a wireless testbed, and having a wired testbed. Coming from the former to the latter, there is a decrease in interference with the sensor network application, at the expense of an increase in the complexity of the collection infrastructure. Some of these strategies are more appropriate to be used during development and others are more appropriate for field use.

Two important aspects addressed by the tool are trace synchronization (implemented in a way that avoids the WSN nodes to be synchronized) and dealing with inaccurate traces (allowing assertions to be in various stages and have a final result of success, failure or unknown). Further analysis on the assertion results can be done by inspecting the values of the attributes and their states at evaluation time.

The tool was tested in a scenario where it was used during the complete life cycle of a specially developed tracking application. The application ran on a WSN with 8 BTnodes equipped with a CC1000 radio. Other 2 BTnodes formed a sniffer network using a Bluetooth radio to forward the information to the backend. The authors concluded that the periodic attribute update messages and two specific assertions accounted for the majority of the outputted traffic, which averaged 3.6 bytes per second.

Concerning PDA's accuracy, around 10 % of the assertions were not decidable due to the loss of attribute update messages. This, in turn, related to the used trace collection mechanism. As a consequence of this level of accuracy, the authors identified the possibility of missing transient or sporadic problems.

The authors claim that PDA can be useful in identifying functional- and timing-related bugs. PDA's main advantages are its ability to perform failure detection and cause identification with the possibility to trade between visibility and required infrastructure (the balance can be changed during application development), and its independence from the application type.

On the other hand, the tool has some drawbacks and limitations: it requires the developer to insert assertions into the application code and to state what attributes should be outputted; failure detection is critically dependent on the correctness and completeness of the assertions that largely depend on the skill of the developer; node reprogramming is required to change and/or update assertions; the tool does not allow the analysis of system state evolution over time. In addition, PDA scalability issues are related to the number of nodes on the network (impact on the sniffer network), the update frequency of assertions and attributes (impact on the sniffer network and on node resources), as well as the number of nodes involved in an assertion (the number of non-decidable assertions increases with the number of nodes involved). Finally, the impact on node resources is not documented, being bound between pure sniffer solutions and solutions exclusively based on node resources.

4 Comparative Assessment

In order to endow the reader with an overall view of the WSN diagnostic tools presented in Sect. 3, in this section a tabular comparison is presented, using the three major classification vectors proposed in Sect. 2: architectural aspects, functional aspects, and dynamic aspects. In the tool comparison tables we maintained the ordering used in the tool summaries: traffic-based tools, node state tools, and global state tools. In the tables the tools groups are differentiated by the light gray background. The information contained in the tables is derived from the tool description papers referred to in Sect. 3. The section ends with some overall considerations on the approach taken by some of the more representative tools and on their scope.

4.1 Architectural Aspects

Table 2 presents the comparative assessment of the various tools from the points of view of the following architectural aspects: scope of analysis, collection triggering, forwarding mechanisms, node participation and developer effort.

4.2 Functional Aspects

Table 3 presents the comparative assessment of the various tools, from the functional aspects' points of view, namely, supported functionality, collected information, application types, languages, OSs and platforms, and required resources. In addition, a column was added, providing information on the tools' availability. Whenever there is no information on a certain characteristic, NI is used in the table.

4.3 Dynamic Aspects

Table 4 presents the comparative assessment of the various tools, from the dynamic aspects points of view, namely flexibility, extensibility, scalability, mobility and heterogeneity. Whenever there is no information on a certain characteristic, NI is used in the table. If the characteristic is not applicable, NA is used.

Concerning scalability, the tools were marked with "NO(1)" if the used diagnostic mechanisms primarily target the internal debugging of individual nodes. These tools can be used on a network-wide basis (e.g., as is the case of Marionette and Nucleos in the Trio experiment) but this requires considerable effort in order to

Table 2 Tool comparison according to the various architectural aspects

Tool name	Analysis location & schedule	Information collection triggering	Information forwarding mechanisms	Individual node participation	Required developer effort
Sympathy	Management Unit (MU)	Event reporting Query-based Also MU sniffer	Node main application channel (single or multi-hop tree based routing)	Node logging	Medium
SNIF	MU, not connected to the WSN	Decision done outside the monitored WSN	Dedicated sniffer nodes with two radios	Passive participation	Light
SNTS	MU, not connected to the WSN, post mortem analysis	Decision done outside the monitored WSN	Dedicated sniffer nodes, test results sent to MU	Passive participation	Light
LiveNet	MU, not connected to the WSN, post mortem analysis	Decision done outside the monitored WSN	Dedicated sniffer nodes listen on the radio and send by wire	Passive participation	Light
Nucleos	Individual node	Query-based Event reporting	Main node channel, using tool stack or application stack	Node logging (also sleep/wakeup control	High
EnviroLog	Individual node	Flash-based logging	Main node channel, using tool stack	Node management	Light
Marionette	Individual node (scripts to analyse node information can run at the MU)	Query-based	Main node channel, using tool stack or application stack	Node management	Medium
Clairvoyant	Individual node	Query-based Event reporting Flash logging	Node main application channel	Node management	Light
NodeMD	Individual node	Query Based Event Reporting	Node main application channel	Node management	Medium
L-SNMS	Individual node (also MU sniffer)	Query-based Event Reporting	Main node channel, using tool stack or application stack	Node management	Medium
LIS	Individual node	*Flash*-based logging Event reporting	Main node channel using AMSend or CTP	Node logging	Medium

Table 2 continued

Tool name	Analysis location & schedule	Information collection triggering	Information forwarding mechanisms	Individual node participation	Required developer effort
Memento	Individual node analysis (and neighbour data aggregation)	Event reporting	Main node channel using application routing protocol	Node logging	High
DT	MU, post mortem analysis	Flash-based logging	Node main application channel, after experiment conclusion	Node management	Medium
Dustminer	MU, post mortem analysis	Flash-based event logging or Event reporting	Node main application channel after experiment conclusion, or node secondary channel during the experiment	Node logging	Medium
MEGS	MU	Event reporting	Node secondary channel	Node logging	High
Wringer	Node cooperation	Event reporting	Main node channel (possible reuse of application stack routing/dissemination)	Node management	Medium
MDB	MU, post mortem analysis	Flash-based logging	Node main application channel, after program execution	Node logging	Light
PDA	MU	Event reporting	All the options are available	Node logging	Medium

Table 3 Tool comparison according to the various functional aspects

Tool name	Supported functionality	Collected information	Application types	Languages, OSs, platforms	Required resources	Availability
Sympathy	Decision-tree-based failure detection, identification and localization	Network-related System uptime	Monitoring	nesC, C TinyOS 1.x Mica2	Node (1558B ROM, 47B RAM)	YES
SNIF	Decision-tree-based failure detection and identification	Traffic	Monitoring	All All CC1000	Other (Sniffer network)	YES
SNTS	Offline machine-learning-based failure analysis	Traffic	Monitoring	All All NI	Other (sniffer nodes)	NO
LiveNet	Network dynamics analysis	Traffic	Monitoring	nesC, C TinyOS TelosB, MicaZ	PC (2.4 GHz, 1 GB RAM) Other (sniffers)	YES
Nucleos	Monitoring (also node sleep/wakeup)	State Code traces	All	nesC, C TinyOS 1.x	Node (NI ROM, 1281B RAM)	YES
EnviroLog	Record/replay asynchronous events and state	State Events (flash recorded)	All	nesC, C TinyOS 1.x Mica2, XSM	Node (15160B ROM, 809B RAM)	NO
Marionette	Debugging (some node configuration also supported)	Variables Remote functions call	All	nesC, C TinyOS 1.x, 2.x TelosB,MicaZ	Node TelosB (1 KB ROM, 100B RAM)	YES
Clairvoyant	Source level debugger, also logging	Code traces Variables	All	nesC TinyOS 1.x Mica2, TelosB	Node(32 KB ROM, 1KBRAM)	YES

Table 3 continued

Tool name	Supported functionality	Collected information	Application types	Languages, OSs, platforms	Required resources	Availability
NodeMD	Debugging SW faults (using fault detectors)	Event trace System param. Memory dump	All	C Mantis Mica2, MicaZ	Node MicaZ—blink—3556B ROM,302B RAM	NO
L-SNMS	Debugging, Configuration, Programming	Variables Remote function call Traffic	All	nesC, C TinyOS 1.x MicaZ	Node (2924B ROM, 588B RAM)	NO
LIS	Logging framework	Code traces Variables	All	nesC, C TinyOS 2.1 Mica2/Z,TelosB	Node MicaZ (1798B ROM, 394B RAM)	YES
Memento	Failure detection and alerting -distributed detector	Developer-defined health conditions	All	nesC, C TinyOS Mica2	Node (NI ROM, 400B RAM)	NO
DT	Debugging (can emulate diverse functions)	Traces (events and state)	All	C LiteOS MicaZ	Node (4006B ROM, 436B RAM)	NO
Dustminer	Offline machine-learning-based failure analysis	Events with state information	All	nesC, C TinyOS 2, LiteOS MicaZ	Node MicaZ/T2 (14670B ROM, 830B RAM) PC(2.53 GHz–512 MB)	NO
MEGS	Anomaly detection (uses assertions/ predicates)	Assertions Triggers State	All	nesC, C TinyOS TOSSIM	Node (NI) Other (extra radio)	NO

Table 3 continued

Tool name	Supported functionality	Collected information	Application types	Languages, OSs, platforms	Required resources	Availability
Wringer	Predicate-oriented debugging	All node variables	All	nesC TinyOS 2 MicaZ,TelosB	Node (NI)	NO
MDB	Debugging (offline macro debugger)	Data trace	All	MacroLab TinyOS TelosB	Node (TelosB—OTA, RAM 304B)	NO
PDA	Monitoring WSN health condition (assertions over WSN state)	Node state Assertion reached	All	NI (nesC?,C?) NI (TinyOS?) BTNode	BTnode (NI) Other (BTNode sniffers)	NO

Table 4 Tool comparison according to the various dynamic aspects

Tool name	Flexibility	Extensibility	Scalability	Mobility	Heterogeneity
Sympathy	NO	NI	YES	NO	NO
SNIF	NA	YES	YES	NI	YES(2)
SNTS	NA	YES	YES	YES(2)	YES(2)
LiveNet	NA	NI	YES	YES(2)	YES(2)
Nucleos	NO	YES	NO(1)	YES(1)	NO
EnviroLog	NO	NI	YES	YES	NO
Marionette	YES	YES	NO(1)	YES(1)	YES(1)
Clairvoyant	YES	YES	NO(1)	YES(1)	NO
NodeMD	YES	NI	NO(1)	YES(1)	NO
L-SNMS	YES	NI	NO(1)	YES(1)	YES(1)
LIS	NO	YES	YES	YES(1)	NO
Memento	NO	YES	YES	NI	NO
DT	YES	YES	YES	YES(2)	NO
Dustminer	NO	YES	YES	YES(2)	YES(2)
MEGS	NO	YES	YES	NO	NO
Wringer	YES	YES	NI	YES(1)	NO
MDB	NA	YES	YES	NI	NO
PDA	NO	YES	YES	YES(1,2)	YES(2)

analyse the results and derive a global perspective. In the case of LIS, the tool supports the generation of scripts that can be used for network-wide diagnostics.

Concerning mobility, tools marked as “YES(1)” can be used in mobile node scenarios as they have the option of using the WSN main application communication protocol for their communication needs. Tools marked as “YES(2)” operate in sniffer mode and any mobility support is conditioned by the sniffers radio range.

Concerning heterogeneity, tools marked as “YES(1)” have the ability to retrieve an XML file describing the component structures used on specific nodes. The tools marked with “YES(2)” can work on a sniffer mode and, in this way, work with a variety of nodes as long as their internal aspects are not visible to the outside.

4.4 Overall Comments

The majority of the presented tools were developed for TinyOS 1.x. and were not ported to TinyOS 2.x. Tools for other WSN operating systems are uncommon, which is not surprising considering that TinyOS was the cornerstone in supporting research in WSNs.

Tools that are based on specifically developed components that run on the nodes are inherently dependent on the platform/OS. The approach followed by LIS—C code instrumentation—was a significant step in increasing the tools’ generality by not requiring programs to be written in nesC. Also tools that decouple collecting from analysing, such as PDA and Dustminer, potentially increase generality.

Marionette and DT cover more use cases by supporting scripting languages. Sniffer-based tools are less dependent on the platform/OS, but the failure space they can diagnose is limited by the information that can be extracted from traffic. In addition, they require more equipment and deployment work.

Mobility is not explicitly covered in the documentation of most of the tools. One can assume that in the cases where the monitoring data is forwarded with the support of the main applications network protocols, the tools will accommodate the node mobility patterns.

In the following sub-sections, some comments specific to the tools in each of the groups used for presentation—traffic-based tools, node state tools, and global state tools—are provided.

4.4.1 Traffic-Based Tools

Most traffic-based tools use machine-learning algorithms for analyzing packet traces, in order to support automatic network-related failure detection and diagnosis. Due to their traffic-based nature, they are essentially applicable to monitoring applications.

Sniffer-based tools can support higher platform diversity and do not have an impact on WSN node resources. It can be said that these tools trade non-interference and independence from the monitored WSN for the extra hardware required by sniffer mechanisms. In addition, as these tools do not interfere with the nodes, they can easily be used on already deployed WSNs.

Traffic-based tools have good scalability potential. Regarding mobility, it is conditioned by the sniffer nodes coverage. These tools usually have good support for heterogeneity, as long as the various platforms under study do not impact the packet traces. By not requiring node code modification, there is not much work on the developer side in order to configure these tools.

4.4.2 Node State Tools

Node state tools use on-node components that collect and output per node information, such as state, code or event traces. Some of these tools also enable interactive node debugging, in order to help the user analyse a failure. The majority of node state tools can support some level of node management functionality. By supporting their functionality on on-node components, they can be used in all types of applications, although they usually target TinyOS-based applications, the exception being NodeMD.

Node resource consumption is a concern that implies using as little ROM/RAM, processing, and communications as possible. ROM/RAM usage by the tools in this group is quite different from tools like Clairvoyant that requires 32 KB ROM (restricting its use to platforms with such resources), or, on the opposite side, Marionette that requires a minimum of 1 KB ROM.

When debugging time-sensitive code sections, special care should be taken. This is the case of Clairvoyant and LIS that try to minimise the impact of their own code latency in the debugging of time-critical code sections.

Node state tools that support interactive node debugging are quite flexible, typically enabling access to state information and code execution not anticipated before the WSN deployment. In general, these tools allow adding more capabilities to the basic set. Scalability is normally limited, mostly because they do not support automated debugging. Mobility is typically supported or dependent on the used information forwarding mechanisms. Roughly half the tools can use the WSN application protocols, and the other half makes use of their own protocol stack. The used approach is determined by the trade-off between robustness and ROM/RAM resources. Most of the tools do not cope with platform diversity.

The effort required to prepare the tools in this group is, in general, medium. The majority of the tools do not require the writing of node code, the exceptions being Nucleos and partly NodeMD. Nevertheless, some of them require the user to insert simple, special comments in the source code. LIS and Marionette offer server side scripting capabilities.

4.4.3 Global State Tools

Global state tools combine information from multiple nodes in order to discover and diagnose failures, using a variety of techniques that try to identify an inconsistent or invalid global state. For instance, PDA, Wringer and MEGS use assertions or predicates over the global state. DT and Dustminer operate off-line over event/state traces, and MDB operates off-line over data traces. These tools have no restrictions on the application type, but mostly only support TinyOS. DT supports LiteOS, and Dustminer supports TinyOS and LiteOS. With the exceptions of DT and Dustminer, there is not much information on ROM/RAM usage.

In general, these tools are extensible and scalable, mobility support is dependent on the tool configuration or on the sniffer's range, and they have limited heterogeneity support. None of the presented tools in this group is publicly available for download.

Most global state tools do the analyses on a separate workstation. In the cases of DT, Dustminer, and MDB the analysis is done off-line due to the necessary CPU resources. In these cases, traces are saved in the nodes' flash and collected after the test finishes. As in the case of node state tools, the required preparation effort to use the tools is, in general, not high.

5 Research Challenges

The overall view of the WSN diagnostic tools presented in Sect. 3 and their comparative assessment provided in Sect. 4 clearly point to the fact that considerable effort has been done in order to develop diagnostic tools that are as easy-to-use and as general as possible. Nevertheless, the carried out analysis also points to the fact that although some of the more recent tools go in that direction, there is still a long way to go.

In general, the use of the presented tools outside the environments for which they were developed has been residual or even null due to several reasons. These include lack of maturity, not being part of the most common operating systems and/or software packages, or in several cases, simply not being available at all.

Nevertheless, the most important reasons have to do with the need for additional research addressing several key aspects of any effective post-deployment diagnostic tool. This section identifies these key research challenges, again using the three major classification vectors proposed in Sect. 2, namely, architectural aspects, functional aspects and dynamic aspects.

5.1 Architectural Aspects

With the exception of traffic-based tools, most WSN post-deployment diagnostic tools require a significant effort in terms of both configuration and use. Even in the cases for which the configuration effort is not high—such as in the cases of Clairvoyant and MDB—the user has to perform a step-by-step diagnosis.

Thus, in what concerns architectural aspects, the challenge is to conceive and develop tools for which configuration and use are simplified. One possible way forward would be to include scripting languages in the tools and/or provide sets of scripts to perform a variety of diagnostic actions, in line with what is already done in the case of the DT, although without its limitations in terms of operating systems and offline operation.

5.2 Functional Aspects

Most tools have been developed for the TinyOS operating system and many of those do not work with version 2 of TinyOS. Tools for other operating systems are scarce. Some examples are NodeMD (for MantisOS) and DT (for LiteOS). The only tool supporting two operating systems is Dustminer.

Portability of most tools is hampered by the fact that they use specific programming languages, such as nesC. The approach followed by LIS—i.e., C code instrumentation—is a good step towards portability.

Another important portability-enabling feature would be the careful structuring of the various software components that constitute them, in order to ease the separation of system-dependent modules from generic modules. This is probably the most important research challenge when functional aspects are concerned.

5.3 Dynamic Aspects

Several research challenges exist regarding the various dynamic aspects under consideration. One of them has to do with scalability. With the increasing use of WSNs for home automation and commercial or industrial buildings management, this will lead to an increase in the number of WSNs to be managed by a single organisation. Thus, scalability must be regarded not only in terms of the number of supported nodes in a single WSN, but also in terms of the number of different WSNs that have to be monitored and/or diagnosed. The available diagnostic tools only address the former problem, not the latter. This embodies several limitations, as it requires a manager to use several monitoring interfaces (one for each WSN), instead of providing a single view of the various WSNs. Moreover, an integrated view has the potential to ease the diagnosis of problems that manifest themselves across various WSNs.

Another important research issue is the support for mobility, as many application scenarios require it. When a tool uses the same protocol stack as the main WSN application, then the mobility scenarios supported by the application are also supported by the tool. However, for tools that are sniffer-based or tools with their own protocol stack, mobility support is not automatic. In this context, it is apparent that mobility support is far from being stabilised in the existing tools and many research issues are still open.

Many WSN applications require dealing with a diversity of sensors, sensor data and configuration parameters, requiring different processing, storage and communication capabilities. For instance, in the case of an intrusion detection WSN application, some nodes with video acquisition and processing capabilities may be cooperating with movement detection nodes. In cases like these, it is important that the tool is able to deal with node heterogeneity, a characteristic still absent in many tools. This ability is also important in order to make the tool more resilient to node replacement (e.g., due to node malfunction) when this replacement involves components not exactly the same as the replaced ones, and to allow for interoperation of equipment from different vendors. In this respect, standardisation efforts—such as the ones developed in the context of the IETF's 6LowPAN working group—lead to the definition of Management Information Bases (MIB) and the use of the Simple Network Management Protocol (SNMP) in WSNs, and are key enablers for research and development activity on tools' heterogeneity support.

6 Conclusion

In this paper, a representative set of tools that enable post-deployment failure diagnosis in Wireless Sensor Networks has been surveyed, with the aim of providing the reader with a broad view of the state-of-the-art and of helping to understand open challenges and possible research directions.

The individual description of each tool covered its main operation principle, underlying paradigm, architecture, main functionality, validation and assessment, main advantages and main disadvantages. In addition, a comparative assessment was presented, considering three major evaluation dimensions: architectural aspects, functional aspects and dynamic aspects.

Although functionality is a key aspect when choosing a diagnostic tool, the developer effort required to use it—comprising its configuration and also the debugging work—as well as node resource consumption, namely ROM/RAM, processing, communications, and energy, are decisive factors. In this line, the more recent tools try to minimize node code instrumentation efforts, reduce the analysis work, increase the tools' applicability and, at the same time, minimize resource usage.

Many challenges still remain concerning post-deployment WSN diagnostic tools. The maturity of ambient intelligence will inevitably lead to an increase in the number of networked embedded systems, as well as their heterogeneity, mobility needs, and size. Monitoring and debugging tools will have to cope with this trend and, additionally, be able to extend their scope to several geographically distant networks at the same time. Another important challenge is security, an aspect that is not covered by existing tools. On one side, the functionality of several of the existing tools would be highly affected if encryption of the monitored data were used. On the other, it is essential to guarantee the security of the monitored as well as the monitoring information.

The availability of WSN failure diagnostic tools is a key aspect in progressing from lab WSNs to real-world, deployed WSNs. Currently, there is no standard tool in this area, and the majority of the tools reviewed in this paper are not sufficiently mature, stable, or well supported, being mostly research projects. As a result, many of them are unavailable to the community. This is mainly due to the fact that WSNs are a relatively new area, mostly driven by academia. Nevertheless, there is clearly an on-going effort to build developer-friendly, easy-to-use, light-impact, non-intrusive, general tools. A key factor to the success of this effort will be the native deployment of appropriate functionality in WSN platforms and operating systems, although this is a process that will necessarily take considerable time.

References

1. Dutta, P., Hui, J., Jeong, J., Kim, S., Sharp, C., Taneja, J., Tolle, G., Whitehouse, K., Culler, D.: Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In: Proceedings of the IPSN/SPOTS (2006). doi:10.1109/IPSN.2006.243880
2. Lee, W.L., Datta, A., Cardell-Oliver, R.: Network management in wireless sensor networks. In: Denko, M.K., Yang, L.T. (eds.) Handbook on Mobile Ad Hoc and Pervasive Communications, 1st edn. CRC Press, Boca Raton (2012). ISBN: 1439846162 (to appear)
3. Ni, K., Ramanathan, N., Chehade, M., Balzano, L., Nair, S., Zahedi, S., Kohler, E., Pottie, G., Hansen, M., Srivastava, M.: Sensor network data fault types. TOSN **5**(3), 1–29 (2009)
4. Zhang, Y., Meratnia, N., Havinga, P.: Outlier Detection Techniques for Wireless Sensor Networks: A Survey. Technical Report TR-CTIT-08-59, Centre for Telematics and Information Technology University of Twente, Enschede (2007). ISSN 1381-3625
5. Zhang, R., Zilic, Z., Radecka, K.: Energy-efficient software-based self-test for wireless sensor network nodes. In: Proceedings of the VTS (2006). doi:10.1109/VTS.2006.32
6. Harte, S., Rahman, A., Razeeb, K.: Fault tolerance in sensor networks using self-diagnosing sensor nodes. In: Proceedings of in IEE Seminar on Intelligent Building Environments (2005). doi:10.1049/ic:20050211
7. Rajasegarar, S., Leckie, C., Palansiwami, M.: Anomaly detection in wireless sensor networks. IEEE Wirel. Commun. **15**(4), 34–40 (2008)
8. Levis, P., Lee, N., Welsh, M., Culler, D.: TOSSIM: accurate and scalable simulation of entire tiny OS applications. In: Proceedings of the SenSys (2003). doi:10.1145/958491.958506
9. Osterlind, F., Dunkels, A., Eriksson, J., Finne, N., Voigt, T.: Cross-level sensor network simulation with COOJA. In: Proceedings of the IEEE Conference on Local Computer Networks (2006). doi:10.1109/LCN.2006.322172
10. Sasnauskas, R., Landsiedel, O., Alizai, M., Weise, C., Kowalewski, S., Wehrle, K.: KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In: Proceedings of the IPSN (2010). doi:10.1145/1791212.1791235
11. Li, P., Regehr, J.: T-Check: bug finding for sensor networks. In: Proceedings of the IPSN (2010). doi:10.1145/1791212.1791234

12. AVR JTAG ICE: User guide. http://www.atmel.com/dyn/resources/prod_documents/doc2475.pdf (2001). Accessed 16 December 2010
13. MSP430 Hardware Tools User's Guide. http://focus.ti.com/lit/ug/slau278e/slau278e.pdf (2010). Accessed 16 December 2010
14. Eriksson, J., Dunkels, A., Finne, N., Österlind, F., Voigt, T.: MSPSim—an extensible simulator for MSP430-equipped sensor boards. In: Proceedings of the EWSN (2007) (poster/demo session)
15. Girod, L., Stathopoulos, T., Ramanathan, N., Elson, J., Estrin, D., Osterweil, E., Schoellhammer, T.: A system for simulation, emulation, and deployment of heterogeneous sensor networks. In: Proceedings of the SenSys (2004). doi:10.1145/1031495.1031519
16. Titzer, B., Lee, D., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: Proceedings of the IPSN (2005). doi:10.1109/IPSN.2005.1440978
17. Eriksson, J., Österlind, F., Finne, N., Tsiftes, N., Dunkels, A., Voigt, T., Sauter, R., Marrón, P.: COOJA/MSPSim: interoperability testing for wireless sensor networks. In: Proceedings of the SIMUTools (2009). doi:10.4108/ICST.SIMUTOOLS2009.5637
18. Werner-Allen, G., Swieskowski, P., Welsh, M.: MoteLab: a wireless sensor network testbed. In: Proceedings of the IPSN (2005). doi:10.1109/IPSN.2005.1440979
19. Ertin, E., Arora, A., Ramnath, R., Naik, V., Bapat, S., Kulathumani, V., Sridharan, M., Zhang, H., Cao, H., Nesterenko, M.: Kansei: a testbed for sensing at scale. In: Proceedings of the IPSN (2006). doi:10.1145/1127777.1127838
20. Doddavenkatappa, M., Chan, M., Ananda, L.: Indriya: a low cost, 3D wireless sensor network testbed. http://indriya.comp.nus.edu.sg/motelab/html/testbed.pdf (2010). Accessed 16 December 2010
21. Handziski, V., Köpke, A., Willig, A., Wolisz, A.: TWIST: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In: Proceedings of the REALMAN (2006). doi: 10.1145/1132983.1132995
22. Tavakoli, A.: Wringer: A Debugging and Monitoring Framework for Wireless Sensor Networks. Doctoral Colloquium. SenSys (2007)
23. Chen, B., Peterson, G., Mainland, G., Welsh, M.: LiveNet: using passive monitoring to reconstruct sensor network dynamics. In: Proceedings of the DCOSS (2008). doi:10.1007/978-3-540-69170-9_6
24. 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)
25. Sundaram, V., Eugster, P., Zhang, X.: Lightweight tracing for wireless sensor networks debugging. In: Proceedings of the MidSens (2009). doi:10.1145/1658192.1658195
26. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for TinyOS. In: Proceedings of the SenSys (2007). doi:10.1145/1322263.1322283
27. Archer, W., Levis, P., Regehr, J.: Interface contracts for TinyOS. In: Proceedings of the IPSN (2007). doi:10.1145/1236360.1236382
28. Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., Estrin, D.: Sympathy for the sensor network debugger. In: Proceedings of the SenSys (2005). doi:10.1145/1098918.1098946
29. Ringwald, M., Römer, K., Vitaletti, A.: SNIF: Sensor Network Inspection Framework. Technical report no. 535, Department of Computer Science, ETH Zurich (2006)
30. Ringwald, M., Romer, K.: SNIF: a comprehensive tool for passive inspection of sensor networks. http://www.vs.inf.ethz.ch/publ/papers/mringwal-snif-kuvs.pdf (2007). Accessed 16 December 2010
31. Khan, M., Luo, L., Huang, C., Abdelzaher, T.: SNTS: sensor network troubleshooting suite. In: Proceedings of the DCOSS (2007). doi:10.1007/978-3-540-73090-3_10
32. Abdelzaher, T., Blum, B., Cao, Q., Chen, Y., Evans, D., George, J., George, S., Gu, L., He, T., Krishnamurthy, S., Luo, L., Son, S., Stankovic, J., Stoleru, R., Wood, A.: EnviroTrack: towards an environmental computing paradigm for distributed sensor networks. In: Proceedings of the ICDCS (2004). doi:10.1109/ICDCS.2004.1281625
33. Tolle, G., Culler, D.: Design of an application-cooperative management system for wireless sensor networks. In: Proceedings of the EWSN (2005). doi:10.1109/EWSN.2005.1462004
34. Luo, L., He, T., Zhou, G., Gu, L., Abdelzaher, T., Stankovic, J.: Achieving repeatability of asynchronous events in wireless sensor networks with Envirolog. In: Proceedings of the INFOCOM (2006). doi: 10.1109/INFOCOM.2006.114
35. Cao, Q., Abdelzaher, T., Stankovic, J., Whitehouse, K., Luo, L.: Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In: Proceedings of the SenSys (2008). doi:10.1145/1460412.1460422

36. Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., Culler, D.: Marionette: Using RPC for interactive development and debugging of wireless, embedded networks. In: Proceedings of the IPSN/SPOTS (2006). doi:10.1109/IPSN.2006.243881
37. Yang, J., Soffa, M., Selavo, L., Whitehouse, K.: Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In: Proceeding of the SenSys (2007). doi:10.1145/1322263.1322282
38. Krunic, V., Trumpler, E., Han, R.: NodeMD: diagnosing node-level faults in remote wireless sensor systems. In: Proceedings of the MobiSys (2007). doi:10.1145/1247660.1247669
39. Hartung, C., Seielstad, C., Holbrook, S., Han, R.: FireWxNet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In: Proceedings of the MobiSys (2006). doi:10.1145/1134680.1134685
40. Yuan, F., Song, W., Peterson, N., Peng, Y., Wang, L., Shirazi, B., LaHusen, R.: A lightweight sensor network management system design. In: Proceeding of the PerCom (2008). doi:10.1109/PERCOM.2008.36
41. Hui, J., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proceedings of the SenSys (2004). doi:10.1145/1031495.1031506
42. Rost, S., Balakrishnan, H.: Memento: a health monitoring system for wireless sensor networks. In: Proceedings of the SECON (2006). doi:10.1109/SAHCN.2006.288514
43. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., Hinton, H.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Conference, pp. 63–78 (1998)
44. Khan, M., Le, H., Ahmadi, H., Abdelzaher, T., Han, J.: Dustminer: troubleshooting interactive complexity bugs in sensor networks. In: Proceedings of the SenSys (2008). doi:10.1145/1460412.1460423
45. Lodder, M., Halkes, G.P., Langendoen, K.G.: A global-state perspective on sensor network debugging. In: Proceeding of the HotEmNets (2008). ISBN: 978-1-60558-209-2
46. Tavakoli, A., Culler, D., Shenker, S.: The case for predicate-oriented debugging of sensornets. In: Proceedings of the HotEmNetS (2008)
47. Sookoor, T., Hnat, T., Hooimeijer, P., Weimer, W., Whitehouse, K.: Macrodebugging: global views of distributed program execution. In: Proceedings of the SenSys (2009). doi:10.1145/1644038.1644053
48. Hnat, T., Sookoor, T., Hooimeijer, P., Weimer, W., Whitehouse, K.: MacroLab: A vector-based macroprogramming framework for cyber-physical systems. In: Proceedings of the SenSys (2008). doi:10.1145/1460412.1460435
49. Römer, K.: Passive distributed assertions for sensor networks. In: Proceedings of the DCOSS (2008)
50. Römer, K., Ma, J.: PDA: Passive distributed assertions for sensor networks. In: Proceedings of the IPSN (2009). ISBN: 978-1-4244-5108-1

Author Biographies

André Rodrigues has a B.Sc. in Informatics Engineering from the University of Coimbra and an M.Sc. in Finance from ISCTE Business School. Currently, he is a Ph.D. student in Informatics Engineering at the University of Coimbra, Portugal, and works as a teacher at the Polytechnic Institute of Coimbra, giving classes on networking. His main research interest is Wireless Sensor Networks, more specifically platforms, debugging and deployment. He is the author of several papers in conferences and magazines in the area of Wireless Sensor Networks. He is a researcher at the Laboratory of Communications and Telematics, Department of Informatics Engineering, University of Coimbra.

Tiago Camilo has more than 8 years experience in telecommunication networks and he is currently a Solution Architect for Telecommunication Networks Software in Nokia Siemens Networks. He received his Ph.D. in Informatics Engineering in 2008 from the Faculty of Science and Technology of the University of Coimbra. He is a professor at the Polytechnic Institute of Guarda, Portugal, where he teaches courses in the area of Wireless Sensor Networks. He is an active member of the Portuguese IPv6 Task-Force. He participated in European projects such as E-NET, E-NEXT, and CONTENT. His publications include one book chapter, and around 30 papers in international refereed journals and conferences.

Jorge Sá Silva received his Ph.D. in Informatics Engineering in 2001 from the University of Coimbra, where is a professor at the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra and a senior researcher for the Laboratory of Communication and Telematics at the Centre of Informatics Engineering of the University of Coimbra, Portugal. His main research interests are mobility, IPv6, network protocols and Wireless Sensor Networks. He has been serving as a reviewer and has published at top conferences and journals in his expertise areas. His publications include 2 book chapters and over 70 papers in refereed national and international conferences and magazines. He participated in European initiatives and projects such as FP5 E-NET, FP6 NoE E-NEXT, FP6 IP EuQoS, FP6 IP WEIRD and FP7 Ginseng (as Portuguese Leader). He actively participated in the organization of several international conferences and workshops (e.g. he was the Workshop Chair of IFIP Networking 2006, Publicity Chair of EWSN 2009, General Co-Chair of EWSN 2010), and he was also involved in program committees of national and international conferences. He is a member of IEEE, and he is a licensed Professional Engineer. His homepage is at http://www.dei.uc.pt/~sasilva.

Fernando Boavida received his Ph.D. in Informatics Engineering in 1990, and he currently is a full professor for the Department of Informatics Engineering (DEI) of the Faculty of Sciences and Technology of the University of Coimbra. He is the founder of the Laboratory of Communications and Telematics (LCT) of DEI and the Strategic Director for Communications and Information Technology of the University of Coimbra. His main research interests are mobility, content networks, quality of service, and Wireless Sensor Networks. His publications include 14 books, 9 book chapters, 42 papers for national conferences and journals, and 130 papers in international refereed journals and conference proceedings. He was the chairman of the Program Committee of QofIS'2001, IDMS-PROMS'2002, NETWORKING 2006, WWIC 2007, FMN 2008 and EWSN 2010 international conferences/workshops. He has been involved in numerous program committees of major international conferences, including INFOCOM 2006 and 2007. He participated in European initiatives/projects such as RARE (Réseaux IP Européenes), EWOS (European Workshop for Open Systems), COST263 (Quality of Future Internet Services), and several FP5, FP6 and FP7 European projects, including E-NET, E-NEXT, EuQoS, WEIRD, OpenNet, CONTENT, MICIE and GINSENG. He is a senior member of IEEE and a licensed Professional Engineer. He is a member of the editorial advisory board of the Computer Communications journal. His homepage is at http://staff.uc.pt/boavida.