Towards a Versatile Problem Diagnosis Infrastructure for Large Wireless Sensor Networks*

Konrad Iwanicki†‡ and Maarten van Steen†

†Vrije Universiteit, Amsterdam, The Netherlands
‡Development Laboratories (DevLab), Eindhoven, The Netherlands
{iwanicki, steen}@few.vu.nl
http://www.few.vu.nl/~iwanicki/lupa-web/

Abstract. In this position paper, we address the issue of durable maintenance of a wireless sensor network, which will be crucial if the vision of large, long-lived sensornets is to become reality. Durable maintenance requires tools for diagnosing and fixing occurring problems, which can range from internode connectivity losses, to time synchronization problems, to software bugs. While there are solutions for fixing problems, an appropriate diagnostic infrastructure is essentially still lacking. We argue that diagnosing a sensornet application requires the ability to dynamically and temporarily extend the application on a selected group of nodes with virtually any functionality. We motivate this claim based on deployment experiences to date and propose a highly nonintrusive solution to dynamically extending a running application on a resource-constrained sensor node.

"During the spring of 2004, 80 mica2dot sensor network nodes were placed into two 60 meter tall redwood trees in Sonoma, California. [. . .] One month later, initial examination of the gathered data showed that the nodes in one tree had been entirely unable to contact the base station. Of the 33 remaining nodes, 15% returned no data. Of the 80 deployed nodes, 65% returned no data at all, from the very beginning. [. . .] One week into the Sonoma deployment, another 15% of the nodes died [. . .] and no records exist of the events that may have caused this failure. [. . .]"

G. Tolle and D. Culler [2].

"[. . .] In 2004, [Dutch] researchers [. . .] teamed up in an ambitious project to use 150 wireless sensor nodes in a three-month pilot deployment on precision agriculture. [. . .] Out of 97 nodes running for [only] three weeks generating 1 message per 10 minutes, we received only 5874 messages, which amounts to 2%. [. . .]"

K. Langendoen et al. [3].

1 Introduction

Wireless embedded sensing provides real-time, on-the-spot surveillance of the physical environment. The progress in miniaturization, energy conservation, radio technology, and algorithm design, enables building larger and longer-lived wireless sensor networks

* This paper is based on an earlier technical report [1], available online.

(WSNs) which run increasingly complex applications. Their potential has been widely recognized and we are now seeing evidence that embedded sensing in the large can provide new scientific tools, maximize productivity, limit expenses, improve security, and enable disaster containment [4].

However, if such WSNs of hundreds or even thousands of nodes operating over months, if not years, are to become reality, it is time we started paying much more attention to their deployment and subsequent durable maintenance. Both these aspects require the ability to first diagnose, and then fix occurring problems. There are stable solutions for fixing problems, once detected, but an infrastructure for inspecting and diagnosing large, operational WSNs is essentially still lacking. In other words, what we need is a *network debugging system*.

One challenge that needs to be faced is the difficulty of predicting during application development, which features may need to be examined for diagnosing yet unknown, post-deployment problems. First, applications for sensor nodes are no longer simple "sense-and-send." Currently, they often exceed thousands of lines of code and can provide SQL support [5], decentralized topographic mapping [6], or primitive visual object recognition [7]. Second, the exposure of the system to the physical environment and complex internode interactions, together often result in unanticipated situations, which are the main cause of post-deployment problems [2,3,8,9,10]. Third, because there are still so many unexplored issues of WSNs, or issues unique to a particular environment, many occurring problems have not yet been fully studied [8,11].

Another challenge is the infeasibility of performing exhaustive and/or on-the-spot diagnosis. Limited accessibility to nodes combined with the scale of a system often rule out direct inspection of individual nodes. Hardware and resource constraints of sensor nodes preclude intensive debugging over large regions. Moreover, for critical applications, diagnosis of a problem in one part of the network should not disrupt the correctly functioning parts.

In this paper, we take the position that *diagnosing a WSN application requires the ability to dynamically and temporarily extend the application on a selected group of nodes with virtually any functionality*. Being able to dynamically extend the functionality of a running application provides the means to react to unanticipated situations, which is crucial considering all the deployment experiences to date. Combining this with temporal and spatial locality ensures scalability and resource conservation. In particular, there is no need for a node to perform extra functions when the application is behaving apparently correctly. A major issue that needs to be addressed is that extending a running application for diagnostic purposes should be as unintrusive as possible. As we discuss in this paper, this additional requirement calls for special solutions.

We further motivate our position based on existing work in Sect. 2. Section 3 explains our proposal, while Sect. 4 discusses the architectural support it requires. In Sect. 5, we conclude and present our future research agenda.

2 Motivation

Maintaining any large network is difficult. It is even more demanding in the case of a network composed of tiny, cheap, resource-constrained devices embedded in and interacting with the surrounding environment. Deployments to date were literally plagued by a number of rarely foreseen technical problems including, amongst others, losses of connectivity with a number of nodes, duty cycle desynchronization, sensor decalibration, too rapid power consumption, and software bugs in seemingly mature and stable code [2,3,8,9,10]. Drawing from that experience, we predict that, like corporate or campus networks, large WSNs will ultimately be run by specialized personnel rather than proverbial "herbologists."[1] Equipping such personnel with the ability to deal with unanticipated situations without exhaustive and/or on-the-spot inspections imposes a number of requirements on a diagnostic infrastructure.

To begin with, the infrastructure should enable remote observation of crucial application metrics, that is, a network administrator operating from a remote base station should be able to inspect selected metrics of the application run by particular nodes.

Because of the complexity of current WSN applications and resource limitations of sensor nodes, it is infeasible to define all application variables as observable metrics. Due to the lack of prior knowledge, inherent to novel deployments, and the unpredictability of the majority of problems, a programmer simply cannot foresee which of the variables may need monitoring. Consequently, the infrastructure should enable on-demand access to arbitrary application variables. Additionally, some situations require the ability to trace the changes of a variable's state in a particular part of code. For instance, unstable connectivity can present an unexpected combination of inputs that triggers bugs in untested control paths of the application-level routing code [8].

Diagnosing certain problems (e.g., related to time synchronization or sensor calibration) often requires local coordination and collaboration of nodes in reading metric values. For example, although a clock synchronization algorithm was embedded in the application, a few nodes in the Sonoma deployment died due to time synchronization issues and resulting energy depletion [2]. To diagnose such a problem, one needs to compare clocks of neighboring nodes. Because of huge multi-hop routing delays, the comparison should be performed locally by the neighboring nodes (e.g., using Cristian's timestamp exchange [12]). Otherwise, it is intractable to determine whether the time difference is authentic or due to the multi-hop delays incurred by the network.

The resource constraints of nodes combined with possible network sizes prevent each node from constantly reporting the values of selected metrics. Taking a single snapshot of a thousand-node multi-hop network may involve hundreds of thousands of messages. Such periodic traffic could drain the energy of the nodes and seriously disturb regular application communication. Instead, by exploiting spatial and temporal characteristics of problems, the network debugging infrastructure should minimize intrusiveness. It should be possible to run diagnostic code only as long as it takes to pinpoint a problem and only on the nodes directly affected by this problem.

Finally, communication in WSNs consumes several orders of magnitude more energy than computation. Therefore, if possible, the processing involved in problem diagnosis should be performed by the nodes rather than at a base station. This way, the expensive reporting of raw metric values to a base station is replaced with much cheaper computation and local communication.

[1] At least, until we fully master (if ever) the technology of self-diagnosing and self-healing systems. The real-world experience to date, however, evidences that in the case of large WSNs we are still quite far away from this goal.

Existing problem diagnosis infrastructures for WSNs [2,8,13] proved invaluable for deploying and maintaining relatively small, real-world networks running simple, well-studied applications. However, they were not designed for rapidly emerging systems, characterized by the increasing complexity and scale, and often operating in unfamiliar settings. Although all solutions enable remote monitoring of application metrics, they require the programmer to foresee the important ones prior to the deployment [2,8], or put constraints on how the metric values are obtained [2,13]. Thus, many unanticipated problems cannot be diagnosed. Moreover, metric values are collected per node, and then processed at a base station [2,8,13]. This lack of support for in-network processing and dynamic collaboration of nodes limits scalability and restricts applicability to certain problems. Consequently, as large WSNs pose many challenges unencountered before, an appropriate problem diagnosis infrastructure is necessary.

In contrast to the aforementioned software-based solutions, a recent alternative approach to deploying and maintaining WSNs [11] involves specialized hardware. It requires a second support network to be deployed together with the network running the actual application. The advantage of this approach is that the support network provides a separate reliable wireless backbone for the transport of debug and control information from and to the nodes running the application. The major disadvantage is the cost. The solution we propose in this paper is orthogonal to using a support network. In particular, the support network can be used for transporting application extensions to and retrieving results from the nodes.

3 Principal Operation

Diagnosing a problem involves a human administrator. We assume that he operates from a (possibly mobile or remote) base station capable of communicating with the sensor network. The base station runs the client-side software for interaction with the node-side software of our infrastructure. In other words, the administrator uses the base station to post diagnostic code to the nodes, and to retrieve and analyze information gathered by this code.

Problem detection is often application specific. For example, in a typical data-gathering application, the fact that nodes in some part of the network do not return data implies a problem in that part. In contrast, detecting a problem in a system for reactive tasking (i.e., based on their readings, sensor nodes trigger nearby actuators) looks completely different. For this reason and due to space limitations, here we focus only on those aspects of our solution that are application independent. However, we recommend the application to be designed in a way that facilitates problem detection. For instance, a simple, lightweight generic problem detection mechanism can be implemented by defining a few basic "health" metrics of a node and using multi-resolution aggregation on those metrics [14,15]. Abnormal aggregate metric values indicate a problem, which is then diagnosed using the proposed infrastructure. Alternatively, the infrastructure itself can be used to periodically install primitive diagnostic extensions, which can detect if there are any problems. In either case, remote diagnosis of a problem requires some nodes not to be harmed by the problem, so that they can diagnose other nodes.

We illustrate the basic idea behind our solution with the example use-case from Fig. 1. Consider an application, like TinyDB [5], that gathers sensor data at the base station.

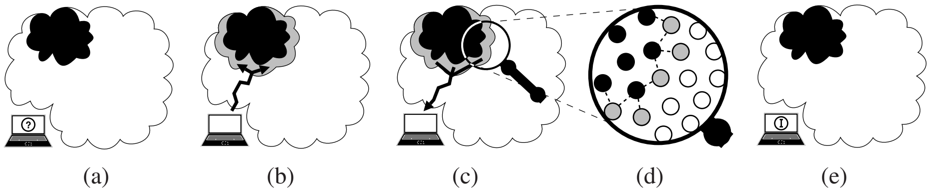

Fig. 1. An example illustrating the operations involved in a problem diagnosis using our approach.

The nodes are organized in a tree that is used for routing. On their way up the tree (to the base station), the data can be aggregated based on accompanying timestamps.

At some point the administrator notices that data from the nodes in the black area are missing [Fig. 1(a)]. This may indicate, for instance, that the nodes in this area ran out of batteries or that they lost connectivity with the rest of the network. To diagnose the problem, the administrator first installs an application extension on the nodes at the perimeter of the black region — marked with gray [Fig. 1(b)]. The task of the extension is to check whether the black region is connected to the rest of the network [Fig. 1(d)]. This can be accomplished by examining the neighbor table maintained by the application on the nodes at the perimeter or by snooping incoming packets at the perimeter nodes and analyzing their sender. All nodes running the extension aggregate a single connectivity result (YES or NO) and route it to the base station [Fig. 1(c)]. When the administrator has gathered requested information, the extension is removed from the application [Fig. 1(e)].

Problem diagnosis is performed by gradually narrowing the set of possible causes. Extending the application on a group of nodes can be viewed as “zooming into this group,” and the extension itself as “a magnifying glass.” Multiple magnifying glasses can be used to progressively “zoom into the problem.” The increasing level of detail of subsequently used glasses (in terms of the amount of data analyzed and reported) and the decreasing range (in terms of the number of nodes affected by a glass) correspond to pinpointing the cause of the problem. For instance, if the black region was not connected, a further on-the-site inspection would be the only option. However, if it turned out that the black region was connected, the administrator could install another extension, but this time in the black region itself, since the nodes there were obviously running. The new extension would get the routing subtree used by the application. If the subtree was correct, then the problem was likely caused by its root node, and thus, an even more specific extension could be installed on that node and its neighbors. This extension could check if the root node did not have time synchronization problems, such that it rejected the data from its subtree as stale. The extension could even force resynchronization or turn off the faulty node if the problem persisted. Using a sequence of application extensions allows for precisely diagnosing occurring problems, and in some cases (e.g., time desynchronization, sensor decalibration, software bugs), eliminates the need for on-the-site inspections.

The proposed approach explicitly deals with the unpredictability of problems. By extending the application in an arbitrary place in the code with virtually arbitrary diagnostic functionality, we can pinpoint and react to the problems that were unanticipated by the application developers, even if the application is extremely complex. We are also able to temporarily coordinate a group of nodes to perform collaborative debugging. The examples include the aforementioned clock synchronization problem, or local isolation of nodes reporting erroneous sensor readings.

Furthermore, our solution addresses the scale and resource limitations of large WSNs, and minimizes intrusiveness. First, the diagnosis is performed in reaction to a problem, only for the time necessary to determine the cause, and only on the nodes directly affected by the problem. Second, diagnostic extensions can perform local inspection and aggregation of results. In the example from Fig. 1, the nodes in the perimeter do not individually report their neighbor tables to the base station. Instead, they determine the connectivity on their own, collaboratively aggregate the value locally, and report a single YES/NO answer. This way resource-critical processing and communication are limited to the vicinity of the problem rather than stretched over the whole network. In extreme cases, the extension can alter the application (e.g., reconfigure it) to fix the problem, without even resorting to the base station.

None of the aforementioned diagnostic infrastructures for WSNs [2,8,11,13] addresses all these issues. The issues are, however, crucial for large-scale systems.

The idea of monitoring an application's health using dynamic extensions was explored previously in the context of wired networks [16]. That work proposed on-demand delegation of diagnostic code to a monitored server, to minimize the volume of data transferred between the server and a monitoring host. However, because of a completely different setting, our approach is distinctly novel. Not only does it introduce internode collaboration as a technique for problem diagnosis in distributed systems, but also encourages employing the characteristic properties of WSNs for the benefit of debugging. In the example from Fig. 1, the spatial dependencies between nodes enable the nodes in the perimeter of the black region to collaboratively determine whether that region is connected to the network. Likewise, the broadcast nature of the wireless medium allows a node to snoop all packets sent by its neighbors. Furthermore, because of the hardware constraints of sensor nodes, the implementation of our infrastructure must face challenges not encountered in traditional systems, as explained in the next section.

4 Infrastructure Support

An infrastructure for problem diagnosis according to the presented approach requires resolving a number of issues: delivering a diagnostic extension to selected nodes; ensuring authenticity and integrity of the delivered code; extending the application running on a node in a nonintrusive way; or securely patching a bug in the application. Most of these issues already have stable solutions. In particular, for delivering an extension to the selected nodes, one of the existing routing and multicasting protocols [17] or mobile-agent middleware [18] can be employed. Code authenticity and integrity can be ensured at a minimal energy cost incurred by the nodes using a finite-stream signing technique, adopted for WSNs [19]. To patch a bug in the application, a secure high-throughput network reprogramming protocol [20,19] can be employed.

Here, we concentrate on a core aspect — the architectural support for extending an application running on a node. This is extremely challenging because, despite the hardware constraints of a sensor node and application complexity, the extension installation

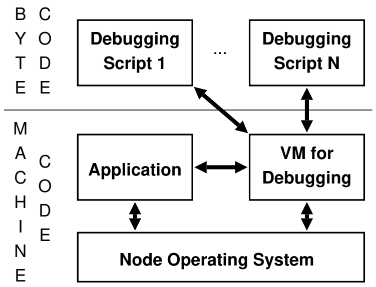

Fig. 2. Architecture overview.

should be as nonintrusive as possible while giving the extension full access to the application state in the face of concurrency. In particular, it is infeasible for the hardware and the OS to provide traditional mechanisms, like `ptrace`. We also do not want to modify the compilers, which is not always possible, or redesign the operating system, which would preclude using an already immense existing contributed code-base.

Due to space limitations, in the rest of the paper, we assume a network of homogeneous nodes, each running the same application. Although we use TinyOS as our base platform, most of the solutions presented in this section are generally applicable.

4.1 Extending the Application

To extend the application at runtime, the node OS should support dynamic code linking, which is usually not the case due to hardware constraints. A node's MCU executes a single static binary combining the application and OS. The binary is executed directly from EEPROM to minimize the amount of expensive RAM and to ensure preservation across node reboots. Reprogramming EEPROM, however, drains a lot of power and is difficult, time consuming, and error-prone, especially if the application is constantly running.[2] Moreover, the number of times EEPROM can be written is always limited. Finally, many architectures do not support position-independent code, so dynamically installing an extension could require patching addresses in the whole binary.

For these reasons, we propose extensions based on interpreted bytecode scripts. Apart from the application and OS, a node's software contains a virtual machine (VM) for interpreting diagnostic scripts (see Fig. 2). Since the scripts are executed from RAM, installation and uninstallation is easy and inexpensive. By using an application-specific VM with a custom instruction set the execution overhead can be minimized [21,22]. Moreover, a bytecode script is usually much smaller than its machine-code counterpart (tens *versus* hundreds of bytes), which allows for delivery through the network and minimizes the RAM footprint to a few hundred bytes [21,22,23].

A script is installed as a breakpoint handler in the application. When the execution of the application reaches a breakpoint the control is transfered to the VM which interprets the script associated with this breakpoint (see Fig. 4). Interrupting the application

[2] Since the OS and the application are merged into one address space, during erroneous reprogramming we may damage not only the application, but also the internal OS code.

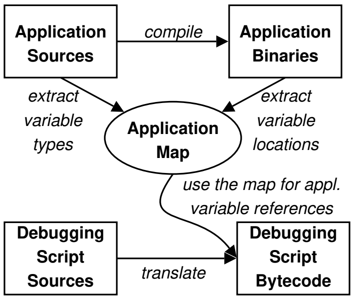

Fig. 3. Application mapping.

on a breakpoint can be implemented either in the hardware or in the software. The hardware implementation involves no changes to the application and no runtime overhead, but not every architecture supports it. The software solution requires modifying the application binary (at runtime or compile time) and imposes some runtime overhead. Preliminary experiments for our implementation, however, indicate that the overhead per compile-time breakpoint is very small (6-7 MCU cycles). Thus, we believe this is a viable solution for the architectures lacking the necessary hardware support.

4.2 Accessing the Application State

The diagnostic scripts must be able to access the state of the application. For instance, a script for checking the connectivity of a network region may need to examine neighbor tables maintained by the application. In general, internally the application uses a number of variables declared by the programmer. To read (and write) those variables, the script needs their precise mapping to the node's RAM. Problems arise because the application is a machine-code binary, generated by an arbitrary (third-party) compiler with arbitrary optimizations.

To cope with this, we concentrate on mapping globally accessible variables (i.e., heap and static variables). Embedded applications store the critical state (e.g., message buffers, caches, neighbor tables, timer counters) statically, and many OSes, including TinyOS, enforce this approach. Heap-allocated variables can also be mapped as the heap is declared statically. Only stack variables are not mapped, because this would require the knowledge of the compiler's internals and would preclude applying many crucial code optimizations.

The application map is generated at compile time (see Fig. 3). From the application sources we extract variable type information (e.g., the types of a structure's fields). From the symbol table in the application binary we extract variable locations in RAM. The combined information constitutes the application map. The map is used to replace symbolic references to application variables in a diagnostic script with memory access instructions. This is performed at the base station, during the translation of the diagnostic script from the scripting language to the bytecode to be interpreted by the VM. Thus, an application map, once generated, is utilized for translating many scripts, as long as

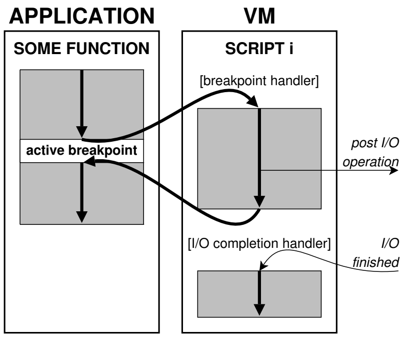

Fig. 4. Invoking breakpoints.

the application is not modified. Moreover, such an approach also enables accessing internal OS structures, offering superb debugging capabilities.

Our implementation of generating application maps is nonintrusive with respect to the original compiler of the application. It bears some similarities to Marionette [13], which uses maps for remote procedure calls.

To cope with possible future memory protection of a sensor node and a separation between the application and OS address spaces, the VM can be a library linked to the application. This way it can freely access the application's address space.

4.3 Handling Concurrency (TinyOS-specific)

An application for TinyOS is composed of tasks triggered by events.[3] Once started, a task runs to completion, that is, it cannot be preempted by any other task. This concurrency model simplifies synchronization and allows for using the same stack for all tasks. It also enables a breakpoint handler to be run in the context of the task that triggered it (see Fig. 4). Upon completion, the handler returns the control to the application, which resumes from the instruction following the breakpoint — again, in the context of the same task. Such a synchronous, nonpreemptable handler execution inherently ensures mutual exclusion of operations on application variables.

A breakpoint handler, however, can perform long I/O operations (e.g., sending a message or logging a value to the flash memory). To avoid blocking the whole application, I/O operations are asynchronous. A breakpoint handler posts an I/O operation (see Fig. 4), which is executed later, in the context of a different task. When the operation has finished, the I/O completion handler of the diagnostic script is invoked. This way the diagnostic script can perform more processing outside the breakpoint handler. Such an asynchronous I/O model with completion handlers is the same as the one used by TinyOS, which reduces the learning curve of our solution and simplifies implementation.

The concept of handlers is further generalized to other event handlers (e.g., for receiving a message sent by a diagnostic script on a neighbor node). Therefore, in fact,

[3] Concurrency handling may need to be implemented differently if some other OS is used.

a diagnostic script consists not only of the breakpoint handlers, but also of a number of other handlers, each corresponding to a well-specified event.

5 Work Status

Considering the research results on mobile VM scripts for WSNs and our preliminary experience with an ongoing implementation, we suspect that the overhead and memory footprint incurred due to the interpreted scripts will be outweighed by the benefits provided by the dynamic diagnostic extensions when deploying our solution on a large scale. It is also possible to further reduce the delivery overhead by caching the scripts in the flash memory of the nodes, and then only sending activation/deactivation commands. In the future, such an approach would allow us to investigate to what extent the network can autonomously help with diagnosing occurring problems. The latter goal constitutes a particularly exciting research agenda, but requires substantially more insight into the nature of such problems.

Our project received much attention from a Dutch consortium planning to build and deploy a very large WSN (see `http://www.devlab.nl/myrianed/`). The established collaboration will enable us to validate whether nonintrusive interpreted dynamic extensions and collaborative problem diagnosis are the tools for industry to maintain future ubiquitous embedded systems.

Acknowledgments

Discussions with S. Sivasubramanian, M. Szymaniak, E. Ogston, and H. Bos helped to refine this paper. The authors would also like to thank anonymous reviewers for their perceptive comments.

References

1. Iwanicki, K., van Steen, M.: Sensor network bugs under the magnifying glass. Technical Report IR-CS-033, Vrije Universiteit, Amsterdam, the Netherlands (2006) Available at: `http://www.few.vu.nl/~iwanicki/`.
2. Tolle, G., Culler, D.: Design of an application-cooperative management system for wireless sensor networks. In: Proc. 2nd EWSN, Istanbul, Turkey (2005)
3. Langendoen, K., Baggio, A., Visser, O.: Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In: Proc. 20th IPDPS, Rhodes Island, Greece (2006)
4. Culler, D., Hong, W., eds.: Wireless Sensor Networks. In: Communications of the ACM. Volume 47. (2004)
5. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: TinyDB: An acquisitional query processing system for sensor networks. ACM Transactions on Database Systems (TODS) **30**(1) (2005) 122–173
6. Hellerstein, J.M., Hong, W., Madden, S., Stanek, K.: Beyond average: Toward sophisticated sensing with queries. In: Proc. 2nd IPSN, Palo Alto, CA, USA (2003)
7. Rahimi, M.H., Baer, R., Iroezi, O.I., García, J.C., Warrior, J., Estrin, D., Srivastava, M.B.: Cyclops: In situ image sensing and interpretation in wireless sensor networks. In: Proc. 3rd SenSys, San Diego, CA, USA (2005) 192–204

8. Ramanathan, N., Kohler, E., Estrin, D.: Towards a debugging system for sensor networks. International Journal of Network Management **15**(4) (2005) 223–234
9. Krishnamurthy, L., Adler, R., Buonadonna, P., Chhabra, J., Flanigan, M., Kushalnagar, N., Nachman, L., Yarvis, M.: Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea. In: Proc. 3rd SenSys, San Diego, CA, USA (2005) 64–75
10. Glaser, S.D.: Some real-world applications of wireless sensor networks. In: Proc. 11th SPIE Symposium on Smart Structures and Materials, San Diego, CA, USA (2004)
11. Dyer, M., Beutel, J., Kalt, T., Oehen, P., Thiele, L., Martin, K., Blum, P.: Deployment support network. In: Proc. the 4th EWSN, Delft, The Netherlands (2007) 195–211
12. Cristian, F.: Probabilistic clock synchronization. Distributed Computing **3** (1989) 146–158
13. 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: Proc. 5th IPSN, Nashville, TN, USA (2006) 416–423
14. Ganesan, D., Estrin, D., Heidemann, J.: DIMENSIONS: Why do we need a new data handling architecture for sensor networks? ACM SIGCOMM Computer Communication Review **33**(1) (2003) 143–148
15. Iwanicki, K., van Steen, M.: The PL-Gossip algorithm. Technical Report IR-CS-034, Vrije Universiteit, Amsterdam, the Netherlands (2007) Available at: `http://www.few.vu.nl/~iwanicki/`.
16. Goldszmidt, G., Yemini, Y.: Distributed management by delegation. In: Proc. 15th ICDCS, Vancouver, Canada (1995) 333–340
17. Akkaya, K., Younis, M.: A survey on routing protocols for wireless sensor networks. Ad Hoc Networks **3**(3) (2005) 325–349
18. Fok, C.L., Roman, G.C., Lu, C.: Rapid development and flexible deployment of adaptive wireless sensor network applications. In: Proc. 25th ICDCS, Columbus, OH, USA (2005) 653–662
19. Dutta, P., Hui, J., Chu, D., Culler, D.: Securing the Deluge network programming system. In: Proc. 5th IPSN, Nashville, TN, USA (2006)
20. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proc. 2nd SenSys, Baltimore, MD, USA (2004) 81–94
21. Levis, P., Gay, D., Culler, D.: Active sensor networks. In: Proc. 2nd NSDI, Boston, MA, USA (2005)
22. Koshy, J., Pandey, R.: VM⋆: Synthesizing scalable runtime environments for sensor networks. In: Proc. 3rd SenSys, San Diego, CA, USA (2005) 243–254
23. Levis, P., Culler, D.: Maté: A tiny virtual machine for sensor networks. In: Proc. 10th ASPLOS, San Jose, CA, USA (2002) 85–95