Sensor Network Bugs under the Magnifying Glass

Konrad Iwanicki[*][†] and Maarten van Steen[*]
[*]*Vrije Universiteit, Amsterdam, The Netherlands*
[†]*Development Laboratories (DevLab), Eindhoven, The Netherlands*
{iwanicki, steen}@few.vu.nl

Technical Report IR-CS-033
Vrije Universiteit Amsterdam, December 2006

ABSTRACT

If the vision of large, long-lived wireless sensor networks serving the society is to become reality, deployment and durable maintenance need much more attention. Both these issues require the ability to first diagnose, and then fix occurring problems. While there are solutions for fixing problems, an appropriate diagnostic infrastructure is essentially still lacking. Our position is 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 propose a solution to this problem that is highly nonintrusive.

Faculty of Sciences
Vrije Universiteit Amsterdam
De Boelelaan 1081A
1081 HV Amsterdam
The Netherlands

vrije Universiteit amsterdam

Contents

1 Introduction

Wireless embedded sensing provides real-time, on-the-spot surveillance of the physical environment. The technology progress enables building long-lived networks of tiny, resource-constrained sensing devices, which run increasingly complex applications. To deploy and maintain such networks a diagnostic infrastructure is necessary.

The potential of wireless sensor networks (sensornets) has been widely recognized. With the progress in miniaturization, energy conservation, radio technology, and algorithm design, 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 [3].

However, if such sensornets 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 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 sensornets 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 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 lines of code and can provide SQL support [13], decentralized topographic mapping [7], or visual object recognition [14]. 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 [16, 15, 10, 5]. Third, because there are still so many unexplored issues of sensornets, or issues unique to a particular environment, many occurring problems have not yet been fully studied [15].

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.

We take the position 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*. Being able to dynamically extend the functionality of an application provides the means to react to unanticipated situations. 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 an 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 Section 2. Section 3 explains our proposal, while Section 4 discusses the architectural support it requires. In Section 5, we conclude and introduce our current implementation.

2 Motivation

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.

Because of the complexity of current sensornet 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 problems, a programmer simply cannot foresee which of the variables may need monitoring. Consequently, the infrastructure should enable on-demand access to arbitrary 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 [15].

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 [16]. 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 (e.g., using Cristian's timestamp exchange [2]). Otherwise, it is intractable to determine whether the time difference is authentic or due to the multi-hop delays.

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

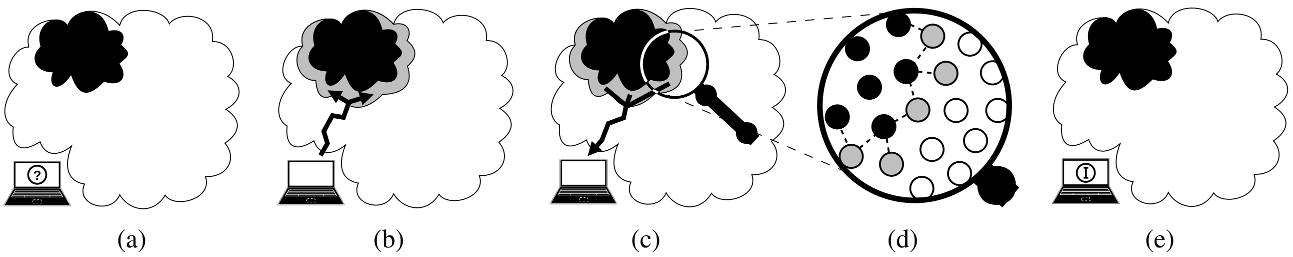

Fig. 1. An example illustrating the operations involved in a problem diagnosis using our approach.

affected by this problem.

Finally, communication in sensornets 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.

Existing problem diagnosis infrastructures for sensornets [15, 16, 17] 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 [15, 16], or put constraints on how the metric values are obtained [16, 17]. Thus, many unanticipated problems cannot be diagnosed. Moreover, metric values are collected per node, and then processed at a base station [15, 16, 17]. This lack of support for in-network processing and dynamic collaboration of nodes limits scalability and restricts applicability to certain problems. Consequently, as large sensornets pose many challenges unencountered before, an appropriate problem diagnosis infrastructure is necessary.

3 Principal Operation

We assume a network of homogeneous nodes, each running the same application.[1] The network is connected to at least one base station at which an administrator runs the software enabling interaction with the network. We explain the basic idea using the example in Fig. 1.

Consider an application, like TinyDB [13], that gathers sensor data at the base station. The nodes can be 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)]. To diagnose the problem, he 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 or by snooping incoming packets and analyzing their sender. During execution, the extension returns an aggregated connectivity result (YES or NO) 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 it turned out that the black region was connected, the administrator could install an extension in that region to 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 the subtree as stale. The extension could even force resynchronization or turn off the faulty node if the problem persisted.

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 sensornets, and minimizes intru-

[1]This assumption is due to a lack of space. Note however, that node homogeneity is necessary only from the debugging perspective.

siveness. 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.

The idea of monitoring an application's health using dynamic extensions was explored previously in the context of wired networks [6]. 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 sensornets 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: detecting a problem in the network; delivering a diagnostic extension to selected nodes; ensuring authenticity and integrity of the delivered code; extending the application running on a node in an unintrusive way; or securely patching a bug in the application. Most of these issues have already stable solutions. In particular, we believe that problem detection is application-specific, but may be facilitated with a minimalistic event-notification service. For delivering an extension to the selected nodes, one of the existing routing and multicasting protocols can be employed [1]. Code authenticity and integrity can be ensured at minimal energy cost incurred by the nodes using a finite-stream signing technique, adopted to sensornets [4]. To patch a bug in the application, a secure high-throughput network reprogramming protocol [8, 4] 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 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`. 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 CPU executes the application binary directly from EEPROM to minimize the amount of power-hungry RAM and to ensure preservation across node reboots. Reprogramming EEPROM, however, is expensive, difficult, time consuming, and error-prone, especially if the application is constantly running. 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 application 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 [12, 9]. Moreover, a bytecode script is usually much smaller than its machine-code counterpart (tens *versus* hundreds of bytes) [11, 12, 9], which allows for delivery through the network and execution from RAM.

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. 3). Interrupting the application 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. The experiments with our implementation, however, indicate that the overhead per compile-time breakpoint is very small (6-7 CPU 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

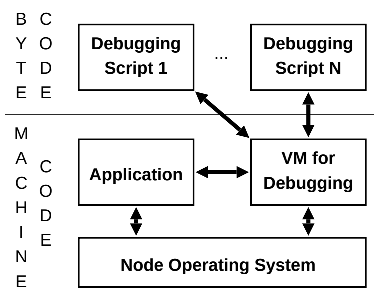

Fig. 2. Architecture overview.

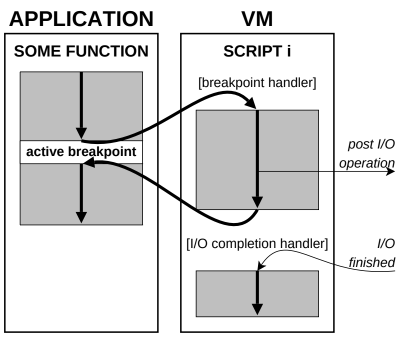

Fig. 3. Invoking breakpoints.

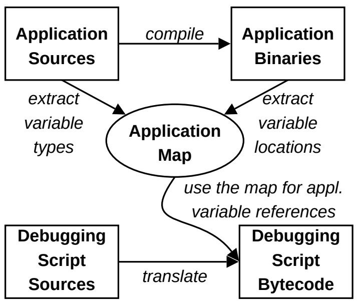

Fig. 4. Application mapping.

connectivity of a network region may 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. 4). 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 interpreted by the VM. Thus, an application map, once generated, is utilized for translating many scripts, as long as the application is unmodified.

Our implementation of generating application maps is nonintrusive with respect to the original compiler of the application. It bears some similarities to Marionette [17], which uses maps for remote procedure calls.

To cope with possible future memory protection of a sensor node, the VM is a library linked to the application, so it can freely access the application's address space.

4.3 Handling Concurrency (TinyOS)

An application for TinyOS is composed of tasks triggered by events. 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. 3). 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. 3), 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, 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

To obtain solid performance results, we have been working on the implementation of our ideas. Considering the research results on mobile VM scripts for sensornets and our preliminary experiments, we suspect that the overhead incurred due to delivery and interpretation of the scripts will be outweighed by the benefits provided by the dynamic diagnostic extensions. 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 self-diagnose occurring problems.

Our project received much attention from a Dutch consortium planning to build and deploy a 10,000-node sensornet. The collaboration enables us to validate whether non-intrusive interpreted dynamic extensions and collaborative problem diagnosis are the tools for industry to maintain future ubiquitous embedded systems.

Acknowledgments

Discussion with S. Sivasubramanian, M. Szymaniak, E. Ogston, and H. Bos refined this paper.

References

[1] K. Akkaya and M. Younis, "A survey on routing protocols for wireless sensor networks," *Ad Hoc Networks (Elsevier)*, vol. 3, no. 3, pp. 325–349, May 2005.

[2] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, pp. 146–158, 1989.

[3] D. Culler and W. Hong, Eds., *Communications of the ACM*, June 2004, vol. 47, no. 6, ch. Wireless Sensor Networks, pp. 30–57.

[4] P. Dutta, J. Hui, D. Chu, and D. Culler, "Securing the Deluge network programming system," in *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN)*, Nashville, TN, USA, April 2006.

[5] S. D. Glaser, "Some real-world applications of wireless sensor networks," in *Proceedings of the Eleventh SPIE Symposium on Smart Structures and Materials*, San Diego, CA, USA, March 2004.

[6] G. Goldszmidt and Y. Yemini, "Distributed management by delegation," in *Proceedings of the Fifteenth IEEE International Conference on Distributed Computing Systems (ICDCS)*, Vancouver, Canada, May-June 1995, pp. 333–340.

[7] J. M. Hellerstein, W. Hong, S. Madden, and K. Stanek, "Beyond average: Toward sophisticated sensing with queries," in *Proceedings of the Second International Conference on Information Processing in Sensor Networks (IPSN)*, Palo Alto, CA, USA, April 2003.

[8] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the Second ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, Baltimore, MD, USA, November 2004, pp. 81–94.

[9] J. Koshy and R. Pandey, "VM$\star$: Synthesizing scalable runtime environments for sensor networks," in *Proceedings of the Third ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, San Diego, CA, USA, November 2005, pp. 243–254.

[10] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis, "Design and deployment of industrial sensor networks: Experiences from a semiconductor plant and the north sea," in *Proceedings of the Third ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, San Diego, CA, USA, November 2005, pp. 64–75.

[11] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *Proceedings of the Tenth ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, CA, USA, October 2002, pp. 85–95.

[12] P. Levis, D. Gay, and D. Culler, "Active sensor networks," in *Proceedings of the Second USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, USA, May 2005.

[13] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 122–173, March 2005.

[14] M. H. Rahimi, R. Baer, O. I. Iroezi, J. C. García, J. Warrior, D. Estrin, and M. B. Srivastava, "Cyclops: In situ image sensing and interpretation in wireless sensor networks," in *Proceedings of the Third ACM Int. Conf. on Embedded Networked Sensor Systems (SenSys)*, San Diego, CA, USA, November 2005, pp. 192–204.

[15] N. Ramanathan, E. Kohler, and D. Estrin, "Towards a debugging system for sensor networks," *International Journal of Network Management*, vol. 15, no. 4, pp. 223–234, July 2005.

[16] G. Tolle and D. Culler, "Design of an application-cooperative management system for wireless sensor networks," in *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, Istanbul, Turkey, January 2005.

[17] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: Using RPC for interactive development and debugging of wireless embedded networks," in *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks (IPSN)*, Nashville, TN, USA, April 2006, pp. 416–423.