# A Global-State Perspective on Sensor Network Debugging

M. Lodder G.P. Halkes K.G. Langendoen
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
The Netherlands
mennolodder@gmail.com g.p.halkes@tudelft.nl k.g.langendoen@tudelft.nl

## Abstract

Debugging wireless sensor networks (WSN) is a notoriously hard problem. WSNs share the debugging problems of both embedded- and distributed systems. The result is that it is very hard to get an insight into the inner workings of a real-world WSN. We have designed the Monitored External Global State (MEGS) tool that recreates the global state of a WSN from debug output. With MEGS a developer can define assertions and predicates on the global state of a WSN, rather than the state of only a single node. Using MintRoute as a test case, we show that MEGS makes available information that was previously exceedingly difficult to obtain.

## Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Distributed debugging

## General Terms

Reliability

## Keywords

Asserts, simulation, testbed

## 1 Introduction

Wireless sensor networks (WSN) have characteristics of both embedded systems, and distributed systems, i.e. low visibility and distributed state. As such, WSNs inherit debugging problems from both fields. Embedded systems make it hard to use interactive debuggers. WSN nodes usually do not allow the use of debuggers at all. Since WSNs are not only embedded systems but also distributed systems, using a normal debugger in most cases will not even provide the information that the developer of the system needs to diagnose the problem. The information the developer needs resides on many nodes. Moreover, the information generally consists of state variables that change over time, which means that to get a clear picture of the WSN we need to look at consistent snapshots of the global state of the network.

Although it is usually possible to obtain a view of the global state in a simulated WSN, none of the techniques currently used for debugging real-world WSNs provide a means to easily gain access to the global state of the network due to the distribution of the data over many nodes. The first step after simulation is usually a trial run on a testbed. A testbed usually allows dumping of debug information, much like the print-style debugging that can be used on PCs. However, creating a mental picture of the operation of the network from this debug information is extremely difficult due to the large amount of information. When we consider debugging techniques for deployed networks, we find that many of them only consider the *behaviour* of the nodes, rather than the internal operation. Moreover, many current techniques for debugging deployed WSNs suffer from the so-called probe effect: inspecting the system influences the system as well because the same wireless interface is used for running the system and sending debug messages. Although the probe effect also impacts testbed measurements, the effect is usually much smaller because a side-channel is used to push out the debug output.

The main contribution of this paper is the design of the Monitored External Global State (MEGS) tool, which leverages existing debugging techniques to recreate (part of) the global state of a WSN on an external PC. Using the recreated state the developer of a WSN can gain insight into the operation of the WSN. MEGS also allows the developer to define assertions and predicates on the recreated state to easily find locations in the execution where anomalous behaviour occurred.

## 2 Related Work

Most testing of WSN software begins with simulations. These simulations come in many forms, from specialised protocol simulators like MiXiM [5] to simulators generated from the source code intended to run on the actual nodes such as TOSSIM [6] and COOJA [8]. When using these simulators, the global state of the network is already present on a single computer, although not all simulators provide easy access. Sometimes emulators like Avrora [11] are used instead of, or next to, simulators. Although emulation can reach a higher level of accuracy, they do make it harder to access the global state of the network because a node's memory is represented as an array of bytes rather than separate variables.

After simulation, the next step in developing WSN software is usually to run the software on a testbed. Testbeds like MoteLab [12] and our own PowerBench [3] normally provide a wired interface (serial port) to the nodes of the testbed. This interface can be used to provide debug information to the developer, mostly in the form of printf-style debug messages.

Once testbed experiments provide satisfactory results, the software will be deployed, sometimes in a test deployment. During the deployment phase there are still options for debugging the network. The Deployment Support Network [2] provides a testbed like environment by attaching a secondary node to each or several nodes in the network. The secondary node provides access to the serial port over a parallel wireless network.

Sympathy [9] provides a diagnosis tool which tries to determine the location of a failure by analysing the traffic coming from the network, and meta information sent in separate messages specifically for diagnosis. The Sensor Network Troubleshooting Suite (SNTS) [4] uses a parallel snooping network to analyse a WSN. Like Sympathy, SNTS only studies the external behaviour of the nodes, rather than looking at the internal state of the nodes.

Marionette [13] is a system that allows calling functions and inspecting and changing memory locations in deployed nodes through an RPC-based system. A developer can use this much like a debugger, although because it uses the wireless channel of the network it has a large impact on the operation of the network. The Clairvoyant [14] tool takes the ideas of Marionette one step further and builds a full debugger using RPC. Although the authors do their best to minimise the probe effect, the extra network traffic, frequent reflashing and break-points impact the execution of the code severly.

The approach of off-line debugging based on logs is already used in embedded systems, for example in the ZealCore debugger [15]. However, these tools inspect a single device, rather than a network of devices.

Recently, Römer et al. [10] proposed a system for distributed assertions much like MEGS, using the WSN node radio and sniffer nodes. The authors do mention the possibility of using a side channel as an option to reduce the probe effect.

From the preceding it is clear that once the step to real hardware is made, it becomes exceedingly difficult to obtain access to the global state of the network. Tools like Sympathy and SNTS only provide information on the behaviour of the network. Although Marionette provides the option to peek inside a node, it does so over the wireless network which provides very limited bandwidth and disrupts the normal network operation (the probe-effect). Only testbeds and DSN provide a side channel to relay information about the internal operations of a node to the developer. We have designed MEGS to use this side channel to gain insight into the global state of the network.

## 3 MEGS Design

Many bugs in WSNs can only be traced when taking into account the state of more than one node. However, it is important to note that to diagnose a problem, the entire memory

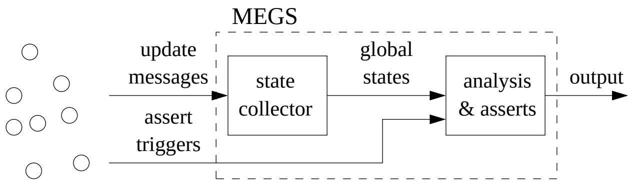

**Figure 1. Conceptual information flow for MEGS. Update messages are received from the testbed and merged into global states. Assert triggers are passed directly to the analysis engine. The global states are then analysed to generate output for the developer.**

of all nodes is not required. To detect whether a routing protocol has created a routing cycle it is sufficient to only have the routing information of all nodes. Furthermore it is usually instructive to look at the states preceding the unwanted state as well, as it can give insight into what caused the unwanted state to occur in the first place.

Given the limited hardware in WSN nodes and the distributed nature of a WSN, it is impossible to inspect the state of a node as one would do with a debugger. Therefore, we have chosen to recreate the global state of the network off-line. This means that all nodes have to provide sufficient information over the side channel to allow for the reconstruction of the (partial) global state. To put as little strain on the nodes as possible, we require that nodes at the very least send a debug message over the side channel when a variable of interest is updated. By accumulating all update messages we can faithfully recreate the global state of the network. Of course, it is possible to send more update messages. This can add to the robustness of the system, especially if the debug channel is not 100% reliable.

All the messages sent over the side channel will be processed by MEGS. For each variable of interest on the nodes, MEGS maintains an off-line state. When an update message is received, a new global state is created by modifying the previous state with the information in the update message. The new state is stamped with the time from the update message. Note that the timestamp of update messages should be generated in such a way that all messages refer to the same clock. In most testbeds the machine receiving the update messages will timestamp the data, providing a single clock. Figure 1 visualises the flow of information when using MEGS.

Once the global state of the network is available, the global state can be used for analysis. Global performance metrics can easily be extracted. For example, the average value of a specific value on all nodes can be traced and plot over time. MEGS can further track the amount of time that a predicate over the global state is true. This can provide vital performance data that is only available when combining the state of many nodes.

MEGS can also check assertions that use variables from multiple nodes. Usually, the assertions should be triggered when a certain point in the execution of a node is reached. The node should therefore send a debug message indicating that it has reached such a point. When MEGS receives such a message the assertion will be checked. The aforementioned

```
...
parent = chooseParent();
global_assert(!routingCycle());
...
```

**Figure 2. Example of a simple global assert statement in a WSN routing algorithm.**

example of a routing cycle in a routing algorithm is a good example of an undesirable state that a developer might want to detect using an assert statement. Figure 2 shows the example as pseudo code. However, this example also highlights a problem with classic assert statements: a routing cycle may be acceptable for a short amount of time when route updates propagate through the network. Therefore, MEGS should also be able to check *timed assertions*.

Timed assertions are of the form: $expression_1 \rightarrow \{\text{within } t\}\ expression_2$. The meaning associated is that once $expression_1$ becomes true, within the time indicated by $t$ $expression_2$ should be true at least once. The timed assertion is not a new concept. The UPPAAL [1] model checker already provides timed assertions, although not explicitly named as such. In the routing cycle example the timed assertion allows bounding the time a routing cycle exists.

## 4 Implementation

We have created a proof-of-concept implementation of MEGS. Our implementation covers the state collector and analysis & asserts engine. The implementation consists of a Java program that needs implementations of two classes to adapt MEGS to the specific WSN software under development. These classes implement the off-line representation of the state of a node, and the specific analysis for the WSN software. The analysis is supported by several utility functions and classes that make it easy to plot data and to create assertions and predicates.

In our current implementation the developer of the WSN software still has to do some manual work that could conceivably be done by a compiler front-end. First of all, the developer of the WSN software still has to manually send each update of each variable of interest over the side channel. A compiler front-end could be created that allows marking the variables of interest and automatically generates the code for sending the update messages at each update site in the code. Second, the compiler front-end could translate assertions in the code to assert trigger messages. Currently, the developer still has to write a statement that sends out a debug message indicating that a particular assertion needs to be checked instead of the actual assertion. The example in Figure 2 would get a statement like `debug_assert(1);` instead of the `global_assert` function call.

Besides providing support on the WSN software side, the compiler front-end can create most of the required Java classes as well. For example, the required node-representation class can be generated from the set of marked variables. Also, the assertions embedded in the WSN software can be used to generate (part of) the analysis class.

We have not implemented this automation as part of our proof-of-concept implementation. However, using tools like CIL [7], building such a compiler front-end is made relatively easy. For TinyOS one could also integrate this functionality with the existing NesC compiler.

```
import engine.*;

public class MintNode extends Node {
    // Parent value when no valid parent was found
    public final int noValidParent = 0xFFFF;

    private int parent = noValidParent;

    public MintNode(int id, SystemStatus system) {
        super(id, system);
    }

    public String toString() {
        return number + ": parent=" + getParent();
    }

    public void setParent(int parent) {
        this.parent = parent;
    }

    public int getParent() {
        return parent;
    }
}
```

**Figure 3. Implementation of the external representation of selected MintRoute state.**

MEGS does not depend on a specific testbed infrastructure. In fact, we can even use the output from simulators. We have currently implemented two interfaces: one for use with the TOSSIM simulator and one for use with our PowerBench testbed. Each of these interfaces is a class in MEGS, and by creating more classes MEGS can be readily adapted to other simulators and testbeds. The only functionality it should provide is a translation from the simulation or testbed specific output to the internal representation for update messages and assert triggers.

## 5 Test Case

In order to test and develop MEGS, we chose software that was readily available and builds up a global state. We chose the standard TinyOS 1.x routing algorithm MintRoute. MintRoute attempts to build a spanning tree over the network, rooted at the sink, to collect data from the network. To this end each node chooses a so-called parent node, to which it will forward all messages destined for the sink node. Obviously, routing cycles are undesirable as this would cause messages to continuously loop without making progress towards the sink. As our first test case we added an assertion to the parent choosing function, that will fail if a routing cycle is created.

To determine if a routing cycle has been created, we need to know the parents the different nodes in the network have chosen. This is all the state we need to test the assertion. Figure 3 shows the implementation of the node-state class for our test. It mainly consists of get and set routines for the state variable, and a function that creates a printable version of the state.

Given the node-state class, all that is required on the PC side is glue code and an implementation of the assertion. This is implemented in a second class, part of which is shown in Figure 4. What is left out from the figure is the code for printing the state of the entire network, the actual routing cycle detection algorithm and a few pieces of administration

```
import engine.*;

public class MintRouteSystem extends SystemStatus {
    ...
    public MintRouteSystem() {
        assertlist.put(1,
            "New parent has created a routing cycle");
    }

    protected void updateStatus(Node node, StatusEvent e) {
        MintNode n = (MintNode) node;
        if(e.getName().equals("PARENT")) {
            n.recordActivity(e.getTime());
            n.setParent(e.getValue());
        }
    }

    private boolean routingCycle(MintNode n) { ... }

    public boolean checkAssert(Node node, int assertNum)
                        throws AssertSkippedException {
        MintNode n = (MintNode) node;

        if (assertNum == 1)
            return !routingCycle(n);
    }
    ...
}
```

**Figure 4. Assertion and state update code for the MintRoute routing cycle assertion.**

```
Time: 218.275 Nodes: 24
...
5: parent=4
6: parent=8
7: parent=6
8: parent=6
9: parent=18
...

Assert FAILED: 8 - 218.275 : New parent has created a
    routing cycle
```

**Figure 5. Example output from MEGS, where a routing cycle has been detected, caused by node 8 choosing one of its children as its new parent.**

code, which allow for more customisation and flexibility in the checking of assertions.

Finally the MintRoute source code needs to be modified. At each point in the code where the parent may change the (possibly) new parent choice is sent over the side channel. Furthermore, whenever a new parent is chosen, the assertion needs to be triggered. This could be done simply by letting the state update code trigger the assertion every time a parent value is changed. However, for the purposes of this example the assertion is triggered by the node itself.

We tested this setup on our testbed and our experiments show that MintRoute did occasionally create a routing cycle. Figure 5 shows an example output for the failed assertion. The output shows the state of the network at the moment the assertion failed. The full state of the network is printed with each state update, to aid the developer in tracing the bug.

It should be noted though that the routing cycles did not last. As the routing information was updated, the routing cycle was resolved and normal operation continued. By using a timed assertion situations like this where unwanted be-

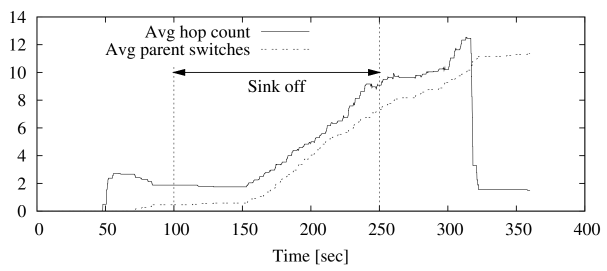

**Figure 6. Effects of a failing sink node between times 100s and 250s on average hop count to the sink node as perceived by the nodes, and the average number of parent switches.**

haviour is acceptable for limited periods of time can be expressed such that only the persistent behaviour will trigger an assertion.

Another option is to get insight in the behaviour of the software under development is the use of a global predicate. The predicate is evaluated each time part of the state is updated, and the percentage of time that it evaluated to true is printed at the end of the run. For the MintRoute test case we have used the routing cycle check to ensure that routing cycles are only present a fraction of the time.

MEGS can do more than check assertions. It can also generate data for plotting. As an example we have instrumented MintRoute to also send the hop count to the sink as perceived by the nodes in the network over the side channel. Using this information the average hop count over all nodes was calculated at each point a node provided updated information. Furthermore, we counted the number of times a node switched to another node a parent in the routing tree. Figure 6 shows a plot created from this data.

In the experiment from which Figure 6 was created the sink node was turned off at 100 seconds into the experiment until 150 seconds later it was turned on again. The figure gives information about the time that MintRoute needs to adapt to changes in the network. Figures such as this one can be helpful in getting an insight of the performance of the protocol. Of course graphs like this can also be generated from the debug logs directly, but the reusable infrastructure provided by MEGS makes it much easier.

## 6 Discussion

Of course, working with real hardware brings with it a lot of difficulties. In developing MEGS, we found that testbed hardware does not always provide the desired clock accuracy. At several points in the hardware and software stack buffering is performed. For example, the serial port to USB converters used in our and many other testbeds have an internal buffer. This results in several messages from one node getting stamped with the same time, even though there is time between the generation of the messages. When more than one node is generating messages at approximately the same time, this buffering may cause the exact ordering of the events to be lost. Although we did not encounter any problems due to this artifact in our test case, this will prove to be a problem when using MEGS for testing MAC protocols which perform actions on separate nodes simultaneously.

Our implementation of MEGS provides a rudimentary means to detect these ordering problems by matching up send and receive events from the nodes and ensuring that the send event was stamped with an earlier time than the associated receive event. This does require extra effort from the developer of the WSN software, as he has to instrument the software to generate side-channel messages on receiving and sending wireless transmissions. However, when an ordering problem is detected it is extremely difficult to reorder the messages in such a way that the correct timing can be guaranteed for all messages. The only true solution is to reduce the buffering in both hardware and software.

A related problem is that the timestamping of the messages may not be performed by a single machine. In some testbeds, including our own, different sets of nodes are connected to different debug-gathering computers. In the extreme case each node is connected to a different device, as is the case in DSN. This means that to obtain correct timestamps these computers must be synchronised. The synchronisation may not be perfect, which will cause slightly incorrect timestamps. As long as messages from different nodes are spread enough in time this is not problematic, but when dealing with time critical protocols such as MAC protocols the imperfect synchronisation may limit MEGS's usefulness.

Because MEGS can be used with any process that produces timestamped update messages, MEGS can also be used with simulators and emulators. These platforms do not suffer from the buffering and time synchronisation issues that real hardware suffers from. Although most simulators already allow access to the global state of the simulated network, there is value in using MEGS in these situations as well. Because the assertions developed for the simulation must also hold when running the software on real hardware, it is beneficial to use the same infrastructure to verify the correct working of the software in both.

As we have described in Section 4, a compiler front-end can be created to automatically generate code to send the values of interest over the side channel. It should be noted though that this is not always easy. In the MintRoute example, the parent value is not a single integer value. It is in fact found through a pointer into the routing table. Hence, the parent value is actually an expression that needs to be evaluated each time the pointer into the routing table is changed. Implementing a compiler front-end that can provide this flexibility is non-trivial.

## 7 Conclusions

We have developed MEGS, a tool which collects incremental state update messages from a WSN and recreates the global state of the network on a separate PC. MEGS leverages existing debugging tools to give developers more insight into the inner workings of their WSN software. Instead of manually looking through debug logs to build a mental image of the state of the network, developers can use the automatically recreated global state that MEGS provides. Using the recreated global state, MEGS can check assertions that involve the state of *all* nodes in the network. Besides providing assertion checking of classic assertions, MEGS also provides checking of timed assertions where detection of one state must lead to the detection of a following state within a specified amount of time. Furthermore, MEGS provides means to gain insight into the performance of WSN software by allowing evaluation of predicates over the global state and allowing easy plotting of data derived from the global state.

We have shown through a test case that MEGS can be helpful in detecting anomalous states of a WSN. Although detecting these states through standard debug messages is possible, MEGS provides an infrastructure to easily create many checks, even those that involve the state of all nodes in the network.

## 8 References

[1] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL–a tool suite for automatic verification of real-time systems. In *DIMACS/SYCON Workshop on Hybrid Systems III*, October 1996.

[2] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and P. Blum. Deployment support network - a toolkit for the development of WSNs. In *4th European conference on Wireless Sensor Networks (EWSN'07)*, pages 195–21, Delft, The Netherlands, January 2007.

[3] I. Haratcherev, G. Halkes, T. Parker, O. Visser, and K. Langendoen. Powerbench: A scalable testbed infrastructure for benchmarking power consumption. In *Int. Workshop on Sensor Network Engineering (IWSNE)*, Santorini Island, Greece, June 2008.

[4] M. Khan, L. Luo, C. Huang, and T. Abdelzaher. SNTS: Sensor network troubleshooting suite. In *3rd IEEE Int. Conf. on Distributed Computing in Sensor Systems (DCOSS'07)*, pages 142–157, 2007.

[5] A. Köpke, M. Swigulski, K. Wessel, D. Willkomm, P. Klein Haneveld, T. Parker, O. Visser, H. Lichte, and S. Valentin. Simulating wireless and mobile networks in OMNeT++: The MiXiM vision. In *1st Int. Workshop on OMNeT++*, March 2008.

[6] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *1st ACM Conf. on Embedded Networked Sensor Systems (SenSys 2003)*, pages 126–137, Los Angeles, CA, November 2003.

[7] G. C. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *11th Int. Conf. on Compilier Construction*, April 2002.

[8] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *1st IEEE Int. Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, November 2006.

[9] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *3rd ACM Conf. on Embedded Networked Sensor Systems (SenSys 2005)*, pages 255–267, San Diego, CA, November 2005.

[10] K. Römer and M. Ringwald. Increasing the visibility of sensor networks with passive distributed assertions. In *Workshop on Real-World Wireless Sensor Networks (REALWSN'08)*, April 2008.

[11] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *4th Int. Symp. on Information Processing in Sensor Networks (IPSN05)*, April 2005.

[12] G. Werner-Allen, P. Swieskowski, and M. Welsh. MoteLab: A wireless sensor network testbed. In *Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS) associated with IPSN'05*, Los Angeles, CA, April 2005.

[13] 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 *5th Int. Conf. on Information Processing in Sensor Networks (IPSN06)*, pages 416–423, April 2006.

[14] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *5th ACM Conf. on Embedded Networked Sensor Systems (SenSys 2007)*, pages 189–203, November 2007.

[15] ZealCore. `http://www.zealcore.com`.