Bug-Tolerant Sensor Networks: Experiences from Real-World Applications

Marcin Brzozowski and Peter Langendoerfer

IHP, 15236 Frankfurt (Oder), Germany
{brzozowski,langendoerfer}@ihp-microelectronics.com

Abstract. Typical sensor networks include large number of motes deployed outdoors. The users expect these networks to work several months or years without maintenance. As a result, every mote must operate reliably for a long time, and it puts a high stress on both hardware and software. Therefore, programs running on motes cannot suffer from software bugs, and the developers must fix them before the deployment.

In this work, we summarize the major techniques for fixing software errors in protocols and applications for sensor networks. However, some bugs are hard to find in the lab, as they do not occur in testing conditions. Therefore, our motes include self-healing techniques, which detect and deal with software problems in the runtime. By doing so, motes keep working reliably for a long time, even when developers did not fix all bugs before the deployment. For instance, we failed to fix a few software errors in the MAC protocol, but the self-healing approach allowed motes to work several weeks outdoors.

Keywords: sensor networks, reliability, software.

1 Introduction

In recent years, our research work focused on wireless sensor networks, and included mainly communication protocols, operating systems, middleware, and security mechanisms. Further, we worked also on hardware platforms, designed our motes, and several hardware accelerators for efficient protocol processing or private/public key cryptography in sensor networks.

Apart from research activities, we deployed several sensor networks in various scenarios. For example, firefighters wore our motes in a pilot run to monitor body parameters at very high temperatures, hundreds Celsius degrees. Further, thirty of our motes managed solar power plants, and others monitored the water level in the forest.

We learned that most users expect the sensor network to work reliably for a long time. It means that software cannot include even minor software errors, as they will lead to problems sooner or later. However, finding software bugs in software for sensor networks is not trivial and requires good experience. For example, software bugs may depend on certain interactions between motes, or exist only in specific hardware configurations. Based on our experience, we know

N. Mitton et al. (Eds.): AdHocNets 2015, LNICST 155, pp. 251–262, 2015.
DOI: 10.1007/978-3-319-25067-0_20

we can fail to spot some bugs in the testing phase, and motes may suffer from them in the runtime. Therefore, our software includes self-healing instructions, which detect software bugs during the runtime and perform a reset of affected motes.

In this work, we present these two major means to deal with software bugs in sensor networks: efficient debugging and self-healing during the runtime. The latter demands more effort from developers, as they must add extra assertions in the time of implementation. However, this extra effort will pay off, especially in long-living applications of sensor networks. For example, although our software included some bugs in the communication protocols, the motes worked without serious problems for several weeks. Further, after fixing these bugs, we started the application again, but the motes suffered from other errors. However, they worked for 1.5 years without maintenance, as the self-healing code performed a reset on bug detection, once in two months on average. Without self-healing techniques, motes would have stop working after about two months in this case.

The rest of this paper is organized as follows. Section 2 gives an overview on debugging techniques and experiences with real-world applications of sensor networks. Then, in section 3 we explain both efficient debugging techniques and self-healing solutions tailored for motes. Section 4 introduces briefly our last two applications with self-healing solutions applied to motes. Then, in section 5 we present four important lessons learned from these scenarios. Finally, section 6 concludes this work.

2 Related Work

A few research works focused on debugging of software for wireless sensor networks (WSN). For example, EnviroLog [6] and another tool [10] store events in the non-volatile memory, and allow the developer to trace bugs. We included this feature in run-time assertions, and each time motes perform a reset after detecting an error, they store the error condition in the flash memory. Another debugging tool, NodeMD [5], catches software errors early enough to prevent the system is not working. After catching the error, NodeMD sends diagnostic information to developers.

In our previous work [2] we showed several techniques to efficiently debug WSN software, from hardware drivers to applications. Further, we evaluated various debugging techniques in terms of memory footprint and their impact on the execution time. This paper continues our previous work, and we show the next step of debugging: self-healing techniques in run-time applications. The need of self-healing solutions arises from problems observed in several outdoor scenarios. In the following, we show the major research works about real WSN.

In ref. [1] the authors share the experience of the entire process of WSN deployments. They noticed that the deployment is the time to face unexpected problems. One important observation made by the authors is KISS (*"Keep it Small and Simple"* or *"Keep it Simple Stupid"*), and we fully agree. Also, the authors stressed that some bugs are hard to spot before the real deployment,

because they do not occur under testing conditions. The same happened to use and was the major reason to include self-healing code in our software.

In the work [7] authors introduced lessons learned from deploying a large-scale sensor network to monitor a potato field. They confessed they neglected software testing, and the mote suffered from many problems in the runtime. They stressed the need of thorough testing of the sensor network, mainly using a testbed. We followed some ideas of the authors and put a lot of effort in offline debugging before deploying the network. We also changed our attitude towards potential problems in the deployment and assumed the worst-case - it helped us to fix most errors in the lab, and the remaining bugs were handled with self-healing code.

The "potato-field" application was started again a year later [4]. This time, the authors followed the KISS principle and make the design much simpler than before. For instance, motes include only a minimal MAC and no routing at all, meaning they sent data directly to the sink. With such a simple design, the sink gathered about 51% of sensor readings, whereas it got only 2% readings in the year before, with much complex design. These observations confirmed our approach in the early stage of the development: make the whole system simple and robust.

3 Bug-Tolerant Software

In this section, we introduce two major means to deal with software bugs: *offline bug fixing* and *self-healing*. The former includes techniques to find bugs before deployment, whereas the latter detects software errors during runtime.

3.1 Introduction

Motes, like other computer systems, may suffer from software bugs. However, as sensors networks work outdoors, performing a reset on motes cannot be easily done. Similarly, motes usually cannot be updated remotely. Therefore, developers must take great care of finding and fixing all bugs before deployment. Otherwise, the affected motes may stop working, and it cannot be easily fixed.

The authors [1] mentioned that some bugs are hard to spot before starting the application, for example, because the testing conditions differ from the real-world scenario. We fully agree on this, and therefore we claim that long-living applications for sensor networks needs more than only bug fixing. Such software must include self-healing solutions, which recover from software errors on runtime. In this way, motes can work for a long time, even when developers did not fix some bugs before deployment.

Although we find self-healing solutions important, we do not underestimate offline bug fixing. On the contrary, we consider fixing bugs before deployment the major step in building reliable software for sensor networks. However, only the combination of both, offline bug fixing and self-healing solutions, guarantee long-living applications of sensor networks.

3.2 Offline Bug Fixing

Finding bugs before deployment includes mainly testing with network simulators and also on real hardware, on motes. In the following, we introduce the major techniques we apply to find and fix software bugs in WSN protocols and applications:

1. *Cross-Platform Software*
   We implement WSN protocols, mainly MAC and routing protocols, as Cross-Platform software, which we can execute on various operating systems and in network simulators. Therefore, we can fix software errors on the PC in network simulators, before running programs on motes, leading to more efficient bug fixing. In recent years, we managed to fix major bugs only in PC simulations, and on motes we worked mainly on hardware-specific problems.

2. *Assertions*
   An assertion stops software execution when a given condition is not met. For example, a program stops when the radio is not in the RX state. With assertions developers examine the failure just after it happened, narrowing down the root cause of the problem. Further, assertions support us also in testing new program versions.

3. *Debug messages*
   With debug messages testers get run-time information while executing programs. We used such messages not only in PC simulations but also when executing software on motes. In this case, the motes sent messages to the PC, which stored them in log files. Such log files help us to find bugs in interactions between motes, for instance, in routing protocols.

4. *Testbed*
   WSN Testbed consists of several motes connected to a backbone network. Developers can program each mote remotely, and also get data from it, mainly debug messages. It is an intermediate step between PC simulations and real-world deployment.
   With our testbed, we can program all motes within a few seconds and get output from the programs running on them. We use the testbed, coupled with debug messages, mainly to find hardware-related bugs that were not found during PC simulations.

5. *Debugger*
   This tool allows to examine running software by checking variable values, setting breakpoints, etc. There are also debuggers for embedded systems, and we can easily debug program running on motes in this way. However, we applied a debugger only when other techniques failed to find bugs.

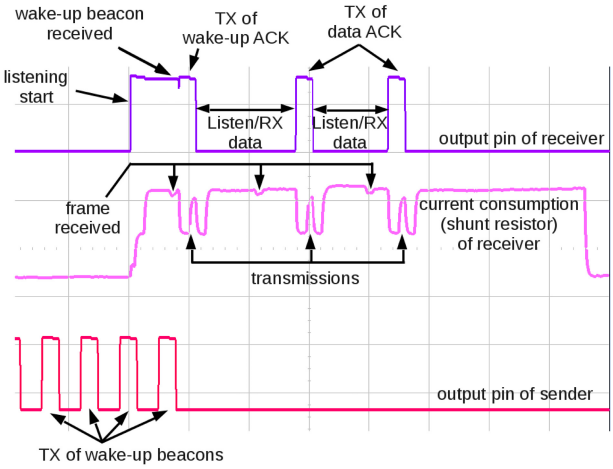


Fig. 1. Example debug session of Low Power Listening with an oscilloscope; this figure shows output pins of the receiver (top) and of the sender (bottom), and the current consumption of the receiver (middle); it allows to discover timing problems of low-level protocols or hardware drivers

6. *LED*
   To get a feedback from the running mote, mainly when implementing low-level drivers, we use a light-emitting diode (LED). For instance, to check if the mote is still running, we observed if the LED kept blinking.

7. *Scope / Logic analyzer*
   We connect General Purpose Input/Output (GPIO) pins of a micro-controller to the scope and debug low-level software problems, mainly hardware drivers. With the scope we get information how long a certain event lasted and how many times it occurred. Further, we can analyze several events at the same time. For example, we examined our MAC protocol by connecting both the sender and the receiver to the scope (see Figure 1). In this way, we were able to trace timing problems of a few milliseconds only. Without using the scope it would be extremely hard to spot this error.

8. Shunt resistor
   A shunt resistor is inserted between the power source and the node. In this way, we measure the voltage drop across it, get the time-current relation by applying the Ohm's law (see Figure 1). For example, with a shunt resistor we can estimate the exact time a message was sent or received.

To efficiently debug software for embedded system, we couple several techniques. For example, we execute programs with assertions on testbed, and check logs with debug messages. We may continue debugging process by running the debugger and setting breakpoints at specific code regions. Our previous work [2] gives more details about debugging techniques for WSN software.

```
void ps_tx_wake_up_success(addr_t rx) {
        ps_ignore_all_rx();
        if (rx != os_get_broadcast_address()) {
                CHECK_FATAL_ERROR( VAR(ps_state) != TX_ALIVE_MSG,
                        "Bad state %u in wake up\n", VAR(state));
```

Fig. 2. Assertion example. We include assertions in CHECK_FATAL_ERROR macros. If the condition is met, the mote performs the system reset in the runtime, and writes the error notice to the flash memory.
In this case, the mote resets the system if the internal state is not TX_ALIVE_MSG

3.3 Self-healing

As stated before, programs running on motes in real-world deployment can suffer from software bugs, even after thorough debugging. Therefore, WSN applications must include extra instructions that will deal with overlooked software errors, dubbed self-healing. In our applications, we included two following solutions to tackle run-time software problems:

1. *Watchdogs*
   are standard features of common micro-controllers to deal with software problems. In short, the micro-controller expects the software to clear a *watchdog flag* periodically. If software does not clear the flag, the MCU assumes that software does not work correctly, and the running program restarts.
   In our application, we clear the watchdog flag in the MAC protocol, when performing a periodic channel check.

2. *Assertions*
   We already introduced assertions previously in *offline bug fixing*. In that case, the running program stops when a condition specified in the assertion is not met (see Figure 2). Clearly, stopping the program in the outdoor application does not help at all. Therefore, when an assertion detects a run-time problem, it restarts the running program. In our opinion, it is better to start the program again instead of keeping it running in a wrong state.
   In the running system motes write the cause of assertion into the flash memory. In this way, we can trace software problems after collecting motes.
   Clearly, it is up to the developers to put reasonable assertions in the source code. We tend to put rather more assertions than too few, as a single assertion needs only about 30 bytes of memory [2], and a few extra bytes for the corresponding debug message.

In both cases, watchdogs and assertions, the mote performs the reset, and set all program variables to their initial values. However, it may lead to various problems, when the variables should keep their values after reset. For example, motes cooperate to find routes in multi-hop networks, and each mote writes partial route information locally in the routing table. In case of assertion, the

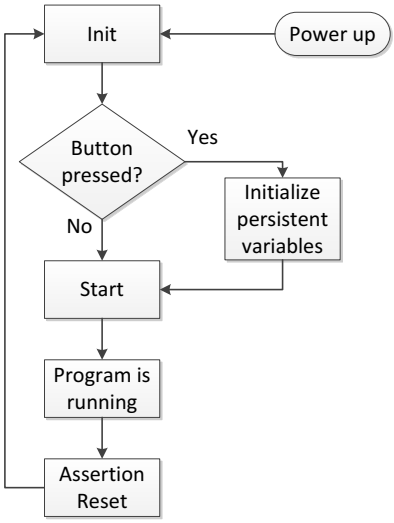


Fig. 3. With assertions motes find software problems in the run-time, perform the system reset, and sets default values to variables. However, some variables, dubbed persistent, should preserve their values on reset. To allow the mote detection of the start condition, reset or power up, we press a button when powering it up. Only in this case the persistent variables are initialized.

mote performs the reset, clears the routing table and cannot forward frames coming from neighbors. It leads to delays in packet forwarding, and to extra traffic caused by finding of missing routes. In this case, the mote should keep its routing table after performing the system reset.

To preserve values of some variable on assertion, the mote must determine whether it was powered up or restarted. Among various solutions, we selected the one based on the *"keep it simple"* principle. That is, when we want the mote to initialize all the variables, we keep the button pressed during power up. In this case, the mote detects the button is pressed, and initializes all variable (see Figure 3). Clearly, when the mote performs the reset caused by assertion, it notices the button was not pressed, and preserves the value of some selected variables.

In previous outdoor applications we learned how powerful and important assertions are. We could even run a sensor network for several weeks knowing there are still software errors in our MAC protocol.

To benefit from self-healing solutions, developers must think already during the implementation what may go wrong in run-time systems, and add appropriate assertions. However, this extra effort will surely pay off, as we experienced in previous outdoor deployments.

4 Application Background

In this section, we introduce briefly our last two real-world applications. Before starting them, we fixed bugs in WSN software in our lab, and also included self-healing techniques. In the next section, we show how well these solutions worked.

4.1 Application One: Tree Growth

During this research project, biologists examined the impact of various phenomena, such as soil quality or solar irradiance, on the tree growth. To carry out this study, they had to collect environmental data in a forest over a long period. We installed the pilot run in the forest about 100 km north of Berlin in Germany.

Our motes provided environmental data from four locations, which were located about 100 meters away from the sink. In theory, motes should send data directly to the sink without problems. However, as there were several trees between motes and the sink, they attenuated the radio signal, and the direct communication suffered from packet losses. We measured the signal strength (RSSI, Receiver Signal Strength Indication) at the sink and found out it was almost as low as the RX sensitivity level of the radio. As it caused the risk of packet losses, we put two relays between the data sources and the sink. These relays forwarded sensor readings to the sink in case there were problems with the direct connection.

After the preliminary run, we fixed some bugs and verified software with our WSN testbed again. After several weeks of in-lab testing, we deployed the sensor network outdoors. The network works already 1.5 years, and it is still operational. However, after about 6 months we replaced all motes, since they were destroyed by a lightning strike.

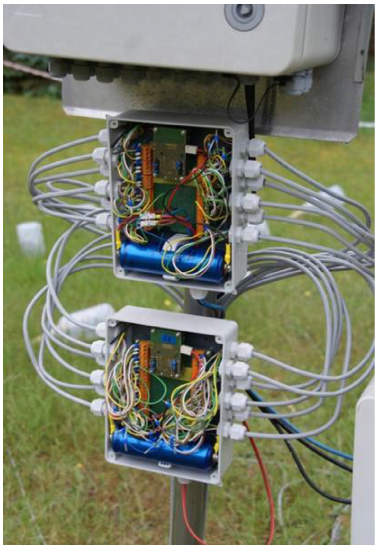
4.2 Application Two: Water Monitoring

In this project, the sensor network monitors water resources, such as reservoirs, rivers, and channels. By doing so, it provides crucial information for preventing floods, and for management of water quality and energy. The project considers water resources in the south of Spain, close to the city Malaga.

The project started in 2014, and the first year we worked only in our lab. We developed adapters needed to connect water-monitoring sensors, and also worked on communication protocols. During this time, we tested software running on motes with the testbed.

At the beginning of 2015, seven motes were deployed in the demonstrator area, in the south of Spain. Based on our previous experience with real-world deployments, we wanted to keep the network simple to avoid many problems that arise from complexity. Therefore, we deployed data sources close to the sink, and allowed direct communication between them, without relays.

Similarly to the previous application, motes in Spain sent data from sensors to the gateway, which forwarded it to the Internet server.



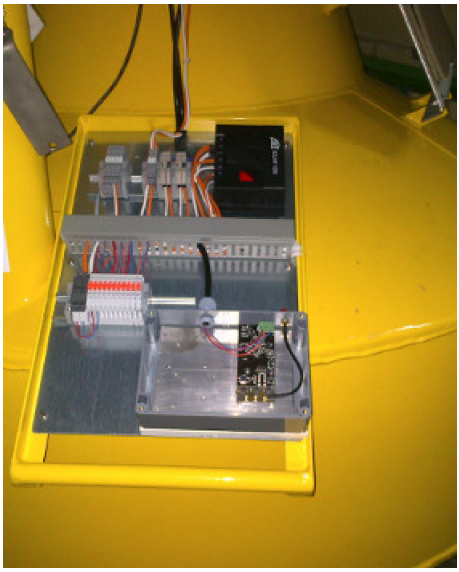


Fig. 4. Our motes used in outdoor applications. On the left, two motes with twelve sensors each for measuring the surface tension of the ground. On the right, the mote installed in a buoy, it measures the water quality in a reservoir.

4.3 Mote Architecture

In both projects we used our proprietary hardware platform for motes, named FWnode [9]. Figure 4 show motes in two above-mentioned applications.

In short, we based the FWnode on 16-bit MSP430 micro-controller, and on three transceivers (CC1101, CC2500, and CC2520), which support 868 MHz and 2.4 GHz frequency bands. In both scenarios, we used the 868 MHz frequency band, since it provides a larger communication range.

To provide long lifetimes, motes need not only a large battery but also suitable protocols that support low duty cycles. Therefore, we applied a low-duty cycle MAC protocol based on preamble sampling [3]. Further, to support multi-hop communication using relays, the motes includes also the AODV [8] routing protocol.

5 Lessons Learned

Before deploying the sensor network, we expected the motes to be the major source of problems. However, as we put a lot of effort to test software running on motes, and included self-healing solutions, the motes worked reliably. The major problems arisen from the Internet gateway. In this section, we shortly share our experience with real-world applications of sensor networks, and emphasize the need of debugging and self-healing solutions.

5.1 Keep it Simple, Or You Are in Trouble

Before starting the projects, we used mainly our schedule-based MAC protocol. It includes many solutions and workarounds to deal with clock problems, leading to high complexity. Further, we discovered some problems during PC-based simulations and fixing them would need a lot of effort. We decided to write from scratch another, much simpler MAC protocol, based on preamble sampling. We hoped that the simpler MAC would lead to smaller maintenance efforts, such as bug-fixing or adding new features. Indeed, it took us only a few weeks to write the complete MAC protocol, which was running in PC simulations and on our motes. Further, finding and fixing bugs of such simple protocol took much less time than working with the complex schedule-based MAC.

Once again we realized that following the "keep it simple" principle brings great benefits.

5.2 PC Simulations Are Essential

As stated before, we implemented the MAC protocol from scratch, and also updated several hardware drivers, mainly for the radio. Therefore, we must thoroughly test new software before deploying it outdoors. Apart from hardware drivers, we tested protocols and applications with the OMNet++ [11] network simulator, and fixed most software problems. After that, we ran tests on real hardware and fixed bugs in hardware drivers. We claim that developers of protocols for sensor networks cannot avoid testing in PC-based simulations. Otherwise, they have to test protocols directly on hardware, and finding bugs related to interaction between many motes is very inefficient and time consuming.

5.3 Self-healing Needed for Maintenance-Free Applications

During the test phase we found and fixed most software errors. However, a few bugs in the communication protocols were hard to spot, as they occurred rarely, once a day, and we could not reproduce them. The affected motes could not send and receive data, and we had to start them again. Nonetheless, we started the preliminary run outdoors, and hoped that self-healing solutions will restart motes when the problem occurs. Indeed, the assertions discovered the software problems and restarted motes several times. As a result, the sensor network worked for a month, until the battery ran out.

After this preliminary run, we fixed the software bug in the MAC protocol and started the final run. Unfortunately, there were still some bugs in software. The self-healing code performed a reset on motes once every 2 months on average. The above example shows the power of self-healing code. That is, motes without it would suffer from software bugs, leading to failures. With the self-healing code, however, the motes ran for long time without maintenance, apart from replacing batteries in case of power-hungry sensors.

5.4 Think about the Whole System, Not Only about the Sensors

We admit that we put a lot of effort to provide reliable software for sensor networks but neglected other parts of the whole application. Before the deployment we assumed that data transmission from the sensor network to our Internet server, using a cellular modem, works without problems. To deal with potential connectivity problems, we added some scripts that monitored the cellular connection. Sadly, our Internet server stopped receiving data from the sensor network a few times, and we realized the need of testing all parts of the running system, not only the sensor network. In the following, we give an example of unexpected problems with the Internet gateways.

5.4.1 GSM Modem Vanished

Once a while the cellular gateway did not send any data to our server due to connection problems. It should not happen, as there was a script running that monitored restarted the connection once a while. In this case, however, the gateway could not detect the cellular modem, as if the modem was removed from the USB slot. After resetting the gateway, it discovered the modem again and got connected to the Internet. To deal with the above mentioned problem, we added a workaround that performed a gateway reset.

5.4.2 SD Card Not Seen by the OS

Another time our server received connection logs from the gateway but the sensor readings were missing. We thought the mote working as the sink was broken. Finally, we came to the demo area and found out that the gateway could not detect the SD card used for storing data from the sink. In this case, the gateway tried to store data into the internal flash memory, but it was full. Surprisingly, after re-inserting the SD card, the gateway detected it and kept saving new data from the sink. We faced the following problem only once, but it may happen again. In this case, we will use extra USB flash drive, and store data on both SD card and USB flash drive.

6 Conclusion

In this work, we presented two major steps to deal with software bugs in WSN applications: offline debugging and self-healing code. We claim that the combination of both solutions allow long-living, reliable applications of sensor networks.

Although we put a lot of effort into software debugging, there were still a few software bugs in the running sensor network. However, as our motes included the self-healing code, they detected the error and performed a reset once in two months on average. By doing so, they worked for 1.5 years and are still operational. Further, we started the preliminary run knowing there are still some software bugs in the running programs. However, the motes detected these bugs at runtime, and continued working for several weeks. It shows the power and

importance of self-healing solutions: motes work with software bugs for a long time and do not need maintenance.

We presented briefly two last deployments of sensor networks. These applications taught us to carefully test the complete system, and not only the sensor network. That is, whereas the motes worked reliably over the whole project, the major problems arise from the Internet gateway. In the end, the sensor networks did not suffer from problems, but the users did not get data because of gateway failures.

Acknowledgment. The research leading to these results was partly funded by the European Community's FP7 Programme under grant agreement n° 619132 (project SAID).

References

1. Barrenetxea, G., Ingelrest, F., Schaefer, G., Vetterli, M.: The hitchhiker's guide to successful wireless sensor network deployments. In: Proceedings SenSys (2008)
2. Brzozowski, M., Langendoerfer, P.: Overview and benchmarks of pragmatic debugging techniques for wireless sensor networks. In: Profeedings SoftCOM (2013)
3. Brzozowski, M., Langendoerfer, P.: Multi-channel support for preamble sampling MAC protocols in sensor networks. In: Proceedings SoftCOM (2014)
4. Haneveld, P.K.: Evading murphy: A sensor network deployment in precision agriculture (2007).
`http://www.st.ewi.tudelft.nl/~koen/papers/LOFAR-agro-take2.pdf`
5. Krunic, V., Trumpler, E., Han, R.: Nodemd: Diagnosing node-level faults in remote wireless sensor systems. In: Proceedings MobiSys (2007)
6. Luo, L., Zhou, G., He, T., Gu, L., Abdelzaher, T.F., Stankovic, J.A.: Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In: Proceedings INFOCOM (2006)
7. Langendoen, K., Baggio, A., Visser, O.: Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In: Proceedings IPDPS (2006)
8. Perkins, C.E., Royer, E.M.: Ad-hoc On-demand distance vector routing. In: Proceedings WMCSA (1999)
9. Piotrowski, K., Sojka, A., Langendoerfer, P.: Body area network for first responders-a case study. In: Proceedings BodyNets (2010)
10. Sundaram, V., Eugster, P., Zhang, X.: Lightweight tracing for wireless sensor networks debugging. In: Proceedings MidSens (2009)
11. Varga, A.: The OMNeT++ discrete event simulation system. In: Proceedings ESM (2001)