A Roadmap for Hardware and Software Support for Developing Energy-Efficient Sensor Networks

Christoph Weyer, Christian Renner, and Volker Turau
Hamburg University of Technology
Institute of Telematics
Schwarzenbergstraße 95
D-21073 Hamburg, Germany
{c.weyer,christian.renner,turau}@tu-harburg.de

Hannes Frey
University of Paderborn
Computer Networks Group
Pohlweg 47-49
D-33098 Paderborn, Germany
hannes.frey@uni-paderborn.de

***Abstract*—Support for developing energy-efficient applications for wireless sensor networks is still scarce. In this paper a roadmap of a combined hardware and software approach is presented. The main idea is to collect state information and trace energy consumption of an application running in a testbed of real sensor nodes.**

I. Introduction

Wireless sensor networks consist of small, micro controller driven nodes with additional sensing capabilities. Once deployed in a certain environment the network must run unattended for a long period of time. In such scenarios energy consumption is the most important system parameter, even if energy harvesting is possible. A lot of research has been conducted in this context. However, the behavior of a new protocol is often evaluated by using simulations only. Producing an executable for real sensor networks still requires a lot of additional effort. Therefore, development support for energy-efficient sensor networks is necessary.

Especially for energy-constraint scenarios it is very important to develop and evaluate the application as soon as possible on real hardware with the additional sensor technology. Running such tests on real hardware with dozens of nodes is not feasible without any additional support. The following tasks must be automated: programming of nodes, monitoring the behavior of the network during the test run, and collecting debugging information. The runtime behavior of a node is described by the energy it consumes and the program parts that are executed over time. This information is important for debugging and evaluation purposes.

In this paper a roadmap of our combined hardware and software approach is presented. The software part will be used for an automated instrumentation of existing applications, such that it provides logging information about the runtime behavior. The hardware part is supposed to measure rate and cumulative current consumption of code execution per node and provides connectivity to manage the node directly from a central management station. We will outline first design decisions on both the hardware and software part in the following sections. After that we will report the first experiences, which we gained from preliminary prototypes.

II. Software Support

In order to get meaningful information about the state of the running application, additional logging data must be provided. In [1] we proposed TinyAID, an automated instrumentation tool that supports two kinds of automated code instrumentation: call-chain logging and message logging. Figure 1 depicts the tool chain of automated code instrumentation. Given any nesC source code, the TinyOS tool chain first creates a single, plain C file by combining this code with the TinyOS components used. The automated code instrumentation intercepts the TinyOS tool chain after this point, adding an additional preprocessing step. Given a certain configuration file the instrumenter inserts additional instrumentation code, provided by a code template, into the plain C file. This step results in an instrumented C file that will then be handed back to the remaining TinyOS tool chain. Depending on the target platform, an instrumented program image or a TOSSIM library is created.

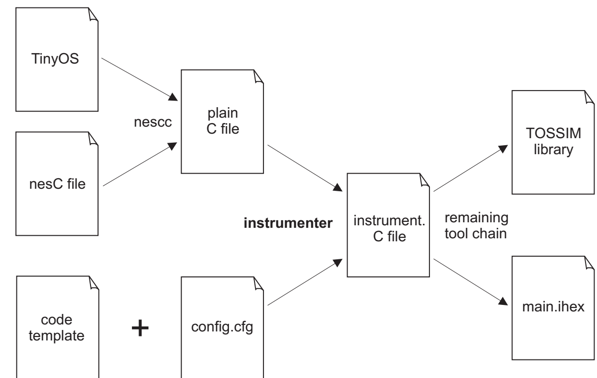

Fig. 1. Automated instrumentation by intercepting the TinyOS tool chain.

A. Call-Chain Logging

Call-chain logging is used for logging the enter and exit times of certain event handlers and functions. This is achieved via additional code that is added to these handlers and functions during the instrumentation pass. For every handler and

function, logging code is added immediately after the function entry point, at the end of the function, and immediately before each return. Since call-chain logging may result in very large data sets due to the names of event handlers and functions, the logged data only consists of unique integers. During the instrumentation pass, a separate file is created, which maps the names of event handlers and functions to a unique integer value.

The event or function to be logged is defined by the configuration file used during compile time. Every line in this file starts with either '+' or '−' to include or respectively exclude event handlers or functions that match the following expression. The inclusion or exclusion symbol is followed by `d`, `f`, or `h` to decide whether the following regular expression is applied on directory names, file names, or handler and function names. The instrumenter steps through the plain C file and checks for every encountered function or event handler entry point, if they match any of the expressions of the configuration file. For this, the list of expressions is scanned from top to bottom, until the first match is found. Depending on the inclusion and exclusion flag, this line decides, if code instrumentation is applied or not. If no entry is found, code instrumentation is not applied.

B. Message Logging

TinyAID also supports logging messages that have been created, sent, or received by the node. This is achieved by the automatic instrumentation of the appropriate TinyOS functions. Basically, additional logging code is added immediately after the entry points of the functions `AMSend.send`, `Receive.receive`, and `Packet.clear`. In order to follow the flow of a message in the network, the message header is extended by an unique message identifier. It consists of the address of the node having created the message (the message's origin) and a sequence number. Each message is tagged with this information at creation time. Here we utilize the TinyOS coding convention that for any newly created message the `Packet.clear` function has to be called.

Since message tagging is completely transparent to the application developer, any application can be monitored with this mechanism. On the event of receiving or sending a packet logging information about the current packet can be provided by the node. The level of detail of the logged data can be easily configured by applying different code templates to the instrumenter.

C. Manual Instrumentation

There are two main situations, in which manual code instrumentation may become unavoidable. These include identifying the visited states of certain state machine implementations, and secondly identifying the end points of communication protocols.

For the first aspect, a function `state(name)` is introduced. It can be added manually at any code line. The instrumenter will create a mapping from state names to automatically generated state identifiers. Again, as with call-chain logging, the additional mapping is used to keep the logged data compact. Code execution passing such function will produce additional logging data. For identifying correct delivery of messages the function `consume(msg)` is introduced, which has to be added at those code places where the message successfully reaches its semantically correct destination.

III. Hardware Support

The hardware adapter is supposed to host a single sensor node and to connect it to an Ethernet network. As sketched in Fig. 2, the adapter is built around a small micro controller with additional peripheral building blocks for adjusting available current, measuring actual and cumulative energy consumption, retrieving state information of current code execution on the sensor node, and communicating logging information towards and control information from a server controlling the experiment. The Atmel NGW100 is used for fast prototyping of the needed hardware components. The final version will be a self-developed, printed circuit board containing all necessary components. The hardware adapter and the actual sensor node are powered via power over Ethernet (PoE).

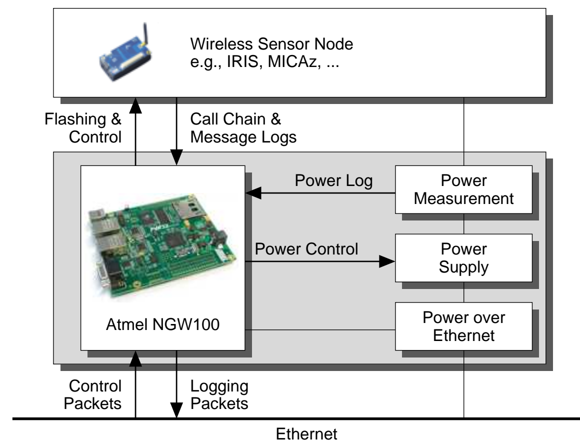

Fig. 2. An overview of the hardware adapter's building blocks.

A. Measuring and Controlling Current

The hardware adapter should enable fine-grained periodic sampling of the current drawn by a given sensor node running a certain application. The measurement should be precise enough to make energy consumption observable on the level of packet reception or function calls. An additional requirement is that our hardware adapter should be able to measure current consumption over several orders of magnitude. More precisely, a sensor node has a current consumption of a few μA, when the sensor node is running in a deep power saving mode, while the node will consume up to about 100 mA, when it is under high load with some additional active sensors. This poses the challenge that the hardware adapter should be able to measure current drawn from 10^{-6} to 10^{-1} A, i.e., five orders of magnitude.

In order to support the development of energy-aware protocols that react on available energy per node, the hardware platform should also be able to control the voltage available at the sensor node. Again, we assume 100 mA as an upper bound of current consumption.

B. Retrieving Log Data

For getting logging information out of a node without affecting its functional behavior in the network, it is important to log as less as necessary while using a most unobtrusive way to communicate such data to the outside world. We consider the usage of several I/O pins as one possibility for this communication between the sensor node and the hardware adapter.

Call-chain logging is performed by sending a single Byte via the I/O pins. Bit 8 encodes if the function is entered or left, while the remaining bits encode the function ID out of at most 127 possible ones, zero means that no data is available. Message logging requires additional information to be transmitted via the I/O pins. This includes one Byte each for the message ID, the receiver node, the message originator, and the sequence number.

For both call-chain and message logging, any additional information is determined on the hardware adapter. This includes the sensor node ID, the current time, the amount of energy consumed since the last call-chain event, and the rate of current energy consumption.

Determining the actual time requires additional effort, since we want to relate energy measurements and events on different nodes in our evaluations. We consider three possible solutions. First, the obvious way is the application of a time synchronization protocol (e.g., NTP) over the Ethernet connection. Second, depending on the deployment, all hardware adapters may be synchronized with a global clock signal over a shared control line. Third, no synchronization may be used at all. In this case, ordering of events on different nodes can be based on the fact that a receive event always happens after its corresponding send event. In other words, the latter technique may be useful if considering causal ordering of events is sufficient for the empirical study.

C. Communicating Data

Ethernet is used in order to exchange data between the hardware adapters and a central management station controlling the sensor network experiment. Information from the server to the hardware adapters includes new images to be flashed and configuration data for controlling the experiments. One example is configuration data for controlling the energy available to the sensor node. On the reverse way any generated logging information and energy consumption measurements are immediately packed into an Ethernet frame and transmitted to the management station. At this computer the information is collected and further processed.

For communicating data from the sensor node to the hardware adapter general I/O pins are utilized as described. Moreover, a single I/O pin is used for communicating between the hardware adapter and the sensor node. This flag can be used for conditionally generated log information, i.e., only if set to true, information will be logged at the sensor node and transmitted to the hardware adapter. In a future version additional ports are available at the hardware adapter for simulating sensor hardware. This is necessary when evaluating the behavior of an application that depends on specific sensor readings or events. Otherwise an automated test is not possible.

IV. Experience Report

A. Software Support

The concept of the automated instrumentation is evaluated in detail in [1]. In all cases TOSSIM is used for simulating the instrumented code. Thereby, the process of logging data is simplified by the fact that the information can be logged directly into files. The configuration of the instrumenter for TOSSIM is as follows. We have provided the modules responsible for creating, sending, and receiving packets with code templates producing trace information. An example code snippet for tracing packet reception is shown in Listing 1.

```
tossim_header_t  * header = getHeader(msg);
dbg_clear("TINYAID_PACKET_TRACING",
  "%d␣%lld␣R␣%d␣%d␣%d␣%d␣%d\n",
  sim_node(), (sim_time_t)(sim_time() * 1e-7 + 0.5),
  header->type, header->src, header->dest,
  header->origin, header->seqno);
```

Listing 1. Code template for packet reception

In order to demonstrate the useability of the automated instrumentation, the packet tracing is demonstrated by comparing three different routing protocols. The first one is TYMO, an implementation of the well-known DYMO protocol, which is included in the TinyOS 2.x code base. TYMO uses internal message types for route requests and route replies. These are sent in order to query and establish a new route, if a forwarding node does not know where to send a given message. The second routing protocol is Dynamic Source Routing (DSR), which follows a similar concept. The third protocol considered is Greedy Routing. Messages are forwarded using the geographic positions of forwarding nodes and the destination. The greedy aspect is realized by considering only neighbor nodes closer to the destination and sending the message to the neighbor closest to the destination. The output of the automated packet tracing is shown in Fig. 3.

B. Hardware Adapter

We have been experimenting with different hardware approaches for adjusting the sensor nodes available power supply and for measuring a sensor node's current consumption [2]. After evaluating all considered approaches it turned out that they are not feasible for the objectives, which we are aiming on with our planned hardware adapter. For both parts, in the following we briefly sketch the different approaches and our observations with that solution.

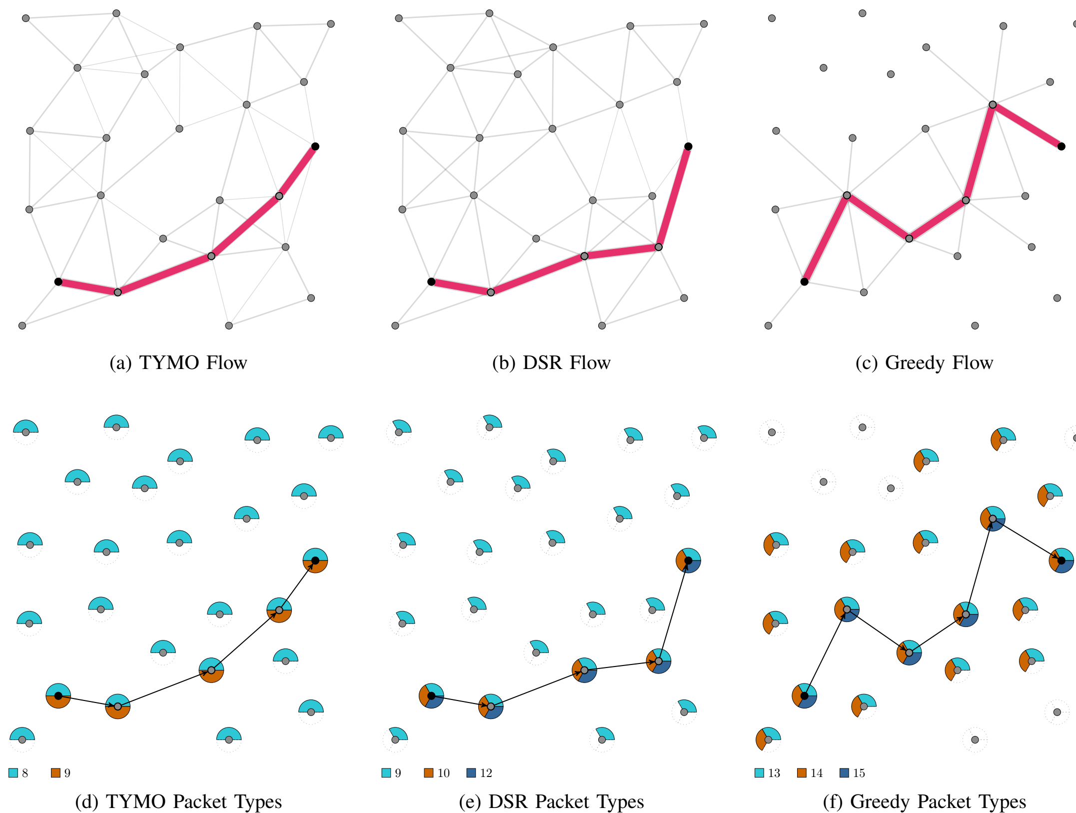

Fig. 3. Visualization of packet flows and packet type distribution

For adjusting output voltage levels an LM317 voltage regulator from National Semiconductor, which supports up to 1.5 A output current, is utilized. The voltage regulator can be adjusted by an analogous input appropriately. A digital adjustable voltage regulator is constructed by a resistor network, which is then controlled by the output lines of a binary register. As an alternative solution a DAC8831 digital analog converter from Texas Instruments is evaluated, which produces an analog voltage level according to a serial digital input. Both circuits are tested on how good they reproduce a sinus and a square wave with sampling rates between 50 Hz to 20 kHz. The results show that only the DAC8831 reproduce an acceptable shape. Unfortunately, the DAC8831 does not allow drawing enough current to support attached sensor nodes running under full load.

For measuring the current draw of the sensor node we considered different instrumentation amplifiers coupled with a shunt resistor, which is then used as an input into the logarithmic amplifier LOG104. The instrumentation amplifiers under consideration included the INA138 and the INA333. The first one was tested alone and in conjunction with an additional INA138 and a REF200 for amplifying the signal. The latter was as well tested alone and in conjunction with a cascaded amplifier INA138. In all investigated settings we were not able to measure current consumption precisely with 1 kHz over 5 orders of magnitude.

V. Conclusion

The development of energy-efficient applications requires supporting tools, especially when dealing with real hardware. We proposed a hardware and software based solution that will help during the development and first test deployments. The advantages of an automated instrumentation support over manual instrumentation are presented by a simple but effective example of message tracing. The results of a first prototype of the proposed hardware adapter have shown that it is difficult to measure the current consumption over 5 orders of magnitude at a frequency of 1 kHz. The next steps will be to solve this open issue in order to gain first experiences by using TinyAID in combination with such a hardware adapter.

References

[1] C. Weyer, C. Renner, V. Turau, and H. Frey, "TinyAID: Automated Instrumentation and Evaluation Support for TinyOS," in *Proceedings of the Second International Workshop on Sensor Network Engineering (IWSNE'09)*, Marina del Rey, CA, USA, 2009.

[2] H. Baumgart, "Entwurf eines Hardwareadapters zur Strommessung von drahtlosen Sensorknoten," Project Work, Hamburg University of Technology, Apr. 2009.