Running Real-World Software on Simulated Wireless Sensor Nodes

Georg Wittenburg
wittenbu@inf.fu-berlin.de

Jochen Schiller
schiller@inf.fu-berlin.de

Department of Mathematics and Computer Science
Freie Universität Berlin
Takustr. 9, 14195 Berlin, Germany

ABSTRACT

In the domain of wireless sensor networks, simulation is the predominant way of evaluating new algorithms. One commonly found drawback of simulation tools is that they provide a programming environment that does not match the one present on real-world platforms.

In this paper, we address this problem by presenting the steps required to port an existing software stack to a popular network simulator. This novel procedure allows for applications to run both in the simulated environment and on real-world sensor nodes without any changes to the source code. We verify our results by means of performance analyses of several simulated applications.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*wireless communication*; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*discrete event*; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*

Keywords

Wireless Sensor Networks, Simulation, ScatterWeb, ns-2

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are formed by a potentially large quantity of sensor nodes that are deployed in the field to cooperatively perform a specific task. They achieve this goal by organizing themselves into a wireless adhoc network. Individual sensor nodes are embedded devices equipped with a variety of sensors to take task-specific measurements of their environment. The stated goal of research is to reduce the physical dimensions of the sensor nodes as much as possible, and hence the efficient use of limited resources, such as processing power, memory and energy, is of paramount importance. Due to this limitation, simply porting existing operating systems and applications to sensor nodes is not feasible and research projects usually develop their own minimalistic software stack.

Software development for WSNs is inherently difficult because of the highly embedded nature of the hardware on one side, and the highly distributed nature of the algorithms used on the other side. To make things worse, having to deploy a large number of sensor nodes in order to evaluate a new algorithm is impractical, if not completely infeasible in scenarios that include, for instance, manually updating the firmware of all devices. Just as in other areas of network research, simulation or emulation are used to work around these problems. In the field of WSNs, simulation tools offer the additional benefit of direct control over both the physical communication medium and the phenomena in the environment that are registered by the sensors. It is hence easy to analyze the exact behavior of one or many algorithms in a variety of reproducible scenarios. [2]

Unfortunately, most simulation tools require algorithms to be adapted or even reimplemented. This puts an extra burden on the developer, who is primarily interested in developing software that runs on real-world devices and uses simulation merely as an intermediary step. In this paper, we address this problem by describing how an existing WSN software stack can easily be ported to the widely-used `ns-2` network simulator [6]. Applications written as part of this software stack can run both on the real hardware platform as well as on the simulator with only minimal one-time adaptations to their source code. The only prerequisite for our approach to be applicable is that the existing software stack must contain a set of C API functions that control access to the actual hardware. Given the fact that C is the de-facto standard programming language for embedded systems with limited resources and that encapsulating hardware resources with C functions is common practise, we expect the majority of existing platforms to meet this prerequisite.

We illustrate our approach by the example of the ScatterWeb platform [8] developed at Freie Universität Berlin. ScatterWeb Embedded Sensor Boards (ESB) are built based on the Texas Instruments MSP430 microcontroller and the TR1001 radio transceiver which operates at 868 MHz with data rates up to 19.2 kbit/s. They include 2 KB RAM and 8 KB EEPROM of memory and sensors to sample temperature, luminosity, noise, and vibration.

REALWSN'06, June 19, 2006, Uppsala, Sweden.

2. RELATED WORK

While several approaches to using the same code base for both simulation and deployment have been proposed, most of them are closer to the field of emulation: They run the unmodified software components on dedicated systems and transparently inject the effects of mobility, radio interference and changing link quality. Emulators may interface with the software components at various layers, e.g. transport [7], network [5], data link [11], and physical [3]. All emulation-based approaches share the difficulty that sufficient processing power needs to be available in order to execute the software under scrutiny in real-time. While this issue can be mitigated by either distributing the emulation over several hosts or adding a layer of virtualization and executing multiple instances of the software on the same host, the problem of scalability remains.

Two noteworthy emulation-based approaches are VMNet [10] and Avrora [9]. By integrating an emulator for the Atmega 128 microcontroller (as used in the MICA2 sensor motes), they not only emulate the networking components of sensor nodes, but additionally the actual hardware itself. This architecture is similar to our approach in that the hardware platform is part of the simulation. The difference is that we simulate the sensor nodes at the abstraction level of a software API rather than at instruction set architecture (ISA) level. While both VMNet and Avrora avoid the effort of having to reimplement the simulated software API, they are less flexible than our approach because they require an emulator of the target CPU to be available. Further, it depends on the particular use case of the simulation tool whether the higher granularity offered by full machine emulation is worth the increase in simulation overhead.

An approach more similar to ours is taken with TOSSIM [4]. It is a simulator specific to applications written in the nesC programming language for the TinyOS operating system. As such, it is rather a TinyOS simulator that includes a network model, while our approach models only hardware-specific parts of the software stack and delegates the simulation of networking aspects entirely to `ns-2`. Our proposed way of integrating an existing software stack with `ns-2` is thus complementary to TOSSIM because it allows for TOSSIM to make use of the existing simulation capabilities available in `ns-2` rather than relying on its own custom network model.

3. PROBLEM STATEMENT

The goal of this project is to run ScatterWeb applications with as little modifications as possible on `ns-2` while retaining the semantics of the real ScatterWeb sensor nodes. This is to be achieved by constructing a layer of glue code between the ScatterWeb application and `ns-2` that implements the C API of the ScatterWeb firmware as shown in Figure 1.

There are several constraints that make the implementation of the glue code layer challenging:

- The core of `ns-2` and most of its components are implemented in C++, while ScatterWeb applications are written in C, against a C API, and otherwise closely tied to the actual hardware, e.g. by using platform specific data types.
- We need to integrate the header files of the real ScatterWeb firmware because they define data types and constants which are used by the applications.

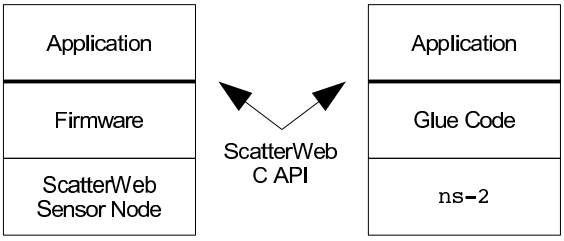

Figure 1: Conceptual sketch of running ScatterWeb applications on `ns-2` by reimplementing the ScatterWeb C API.

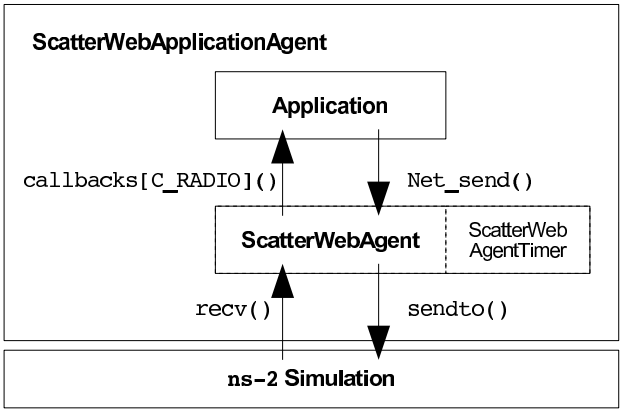

Figure 2: Interactions between the application, the glue code layer and `ns-2` at runtime.

- All components of `ns-2` are statically linked into the `ns` binary at compilation time. We need to link the ScatterWeb application object code into the binary, possibly dealing with clashing symbols.
- ScatterWeb applications are written to run on a dedicated embedded processor. When simulated by `ns-2`, several instances of an application find themselves running as part of the same process. Hence the global variables need to be replicated for each running instance.

While trying to meet these constraints, it is important not to forget that modifications of ScatterWeb applications are to be avoided whenever possible. We cannot expect application developers to port their code back and forth between the real and the simulated ScatterWeb platform.

Neither a simple compatibility layer on top of `ns-2` that implements the C API of the ScatterWeb firmware nor automatic conversion of the application code from C to C++ meet our functional requirements. The crucial problem indeed is to provide separate memory regions for the global variables of each application, i.e. to convert the application code into real object-oriented code.

4. IMPLEMENTATION DETAILS

We have implemented the layer of glue code between `ns-2` and the ScatterWeb application as a set of three C++ classes. The `ScatterWebAgent` class reimplements the ScatterWeb firmware API in C++ and takes care of the interaction with `ns-2`. The `ScatterWebApplicationAgent` inherits this functionality from `ScatterWebAgent` and additionally pulls in

the C code of the application that is to be run on the simulated sensor node. In other words, `ScatterWebAgent` takes care of the functionality of the firmware, while `ScatterWebApplicationAgent` constructs the logical unit of firmware and application that is implicitly created when deploying firmware and application object code on one sensor node. To be more precise, there must be one specialized `ScatterWebApplicationAgent` per specific application, i.e. one needs to slightly adapt `ScatterWebApplicationAgent` for each application. Being derived from the `MessagePassingAgent` superclass of `ns-2`, a `ScatterWebAgent` is integrated as source / sink into the simulated `ns-2` network stack.

Figure 2 illustrates the interaction between `ns-2`, `ScatterWebAgent`, `ScatterWebApplicationAgent` and the application at runtime. For the simple example of receiving and sending a packet, the diagram illustrates how `ScatterWebAgent` translates a call to its `recv()` (which is part of the interface of a `ns-2 MessagePassingAgent`) to the appropriate callback of the ScatterWeb application. When sending a packet, the application calls the `Net_send()` method of the `ScatterWebAgent` object, which is in turn translated into the `sendto()` function of the `ns-2 MessagePassingAgent`.

The diagram also shows a third C++ class, `ScatterWebAgentTimer`, which interacts closely with `ScatterWebAgent`. Its purpose is to schedule periodic events for the simulated firmware, such as the interrupt-driven timer tick on the real sensor nodes.

4.1 Linking C Code into ns-2

The most difficult part of our approach is the `Net_send()` method: While implemented as method of the `ScatterWebAgent` class, it must look just like a C style function from the point of view of the application code. Further, as there may be multiple `ScatterWebAgent` objects at runtime, the application needs to call the method on the correct object.

To achieve this goal, we use the C preprocessor (`cpp`): Listing 1 shows a shortened version of the `ScatterWebApplicationAgent` class definition. In lines 3 to 9, it first defines its public interface consisting of the constructor and the `Process_init()` method, which conceptually belongs to the application. The crucial lines are however lines 15 to 18. In line 15, we include a list of `defines` which map the C API functions as required by the application to the methods implemented by `ScatterWebAgent`. For example, the conversion of the `Net_send()` function is shown in Listing 2.

There are similar `defines` for all 113 functions of the ScatterWeb firmware API. As the API is stable, this list of `defines` only needs to be created once and can be reused for all applications. Note however that the names of the API functions must be unique strings in the application code.

In line 16 of Listing 1, we add another `define` for the `Process_init()` function of the application. This cannot be done as part of the `defines` for the other functions because we need to know the name of application wrapper class, which in this case is "`ScatterWebApplicationAgent`". Invocation of `Process_init()` will happen from the super class via polymorphism. Finally in lines 17 and 18, we `include` the C code of the application into the C++ wrapper class.

Admittedly, lines 15 to 18 use language features in unusual ways. However, the import code is concentrated in exactly those four lines and is independent from the code of the application. In this light, we think it is an acceptable solution for the problem at hand.

Listing 1: Definition of the `ScatterWebApplicationAgent` wrapper class.

```
1  #include "ScatterWebAgent.h"
2
3  namespace ScatterWeb {
4    class ScatterWebApplicationAgent : public
         ScatterWebAgent {
5    public:
6      ScatterWebApplicationAgent();
7      void Process_init();
8    };
9  }
10
11 [...]
12
13 ScatterWebApplicationAgent::
       ScatterWebApplicationAgent() : ScatterWebAgent()
       {}
14
15 #include "ScatterWebFirmwareWrapper.h"
16 #define Process_init ScatterWebApplicationAgent::
       Process_init
17 #include "ScatterWeb.Event.c"
18 #include "ScatterWeb.Process.c"
```

Listing 2: Excerpt from `ScatterWebFirmwareWrapper.h` mapping a C function to a C++ method.

```
[...]
#define Net_send ScatterWebAgent::instance->Net_send
[...]
```

Listing 3: Glue code for sending a packet.

```
1  bool ScatterWebAgent::Net_send(packet_t* packet) {
2    packet->num = txNum++;
3
4    int packet_size = sizeof(packet_t) + packet->
         data_length;
5    PacketData* data = new PacketData(packet_size);
6    char* buffer = (char*) malloc(packet_size);
7    memcpy(buffer, packet, sizeof(packet_t));
8    if(packet->data_length > 0)
9      memcpy(buffer + sizeof(packet_t), packet->data,
           packet->data_length);
10   memcpy(data->data(), buffer, packet_size);
11   free(buffer);
12
13   ns_addr_t dst;
14   dst.addr_ = -1;
15   dst.port_ = message_port;
16   sendto(packet_size, data, 0, dst);
17
18   return True;
19 }
```

Listing 4: Glue code for receiving a packet.

```
1  void ScatterWebAgent::recv(Packet* pkt, Handler* h) {
2          instance = this;
3          updateInternalTimers();
4
5          memcpy(&rxPacket, pkt->accessdata(), sizeof(
               packet_t));
6          rxState = RXSTATE_FULL;
7          Packet::free(pkt);
8
9          runModule |= MF_RADIO_RX;
10         loop();
11 }
```

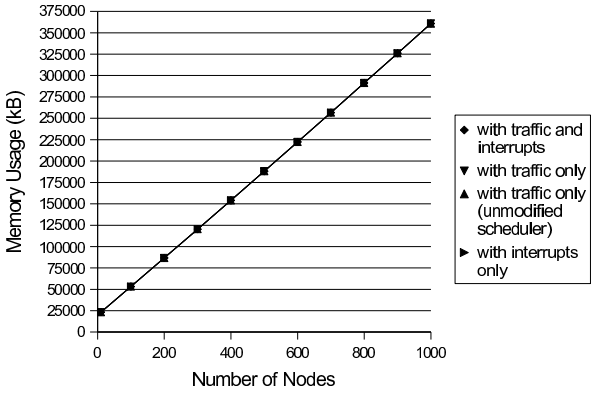

Figure 3: Number of nodes vs. memory usage.

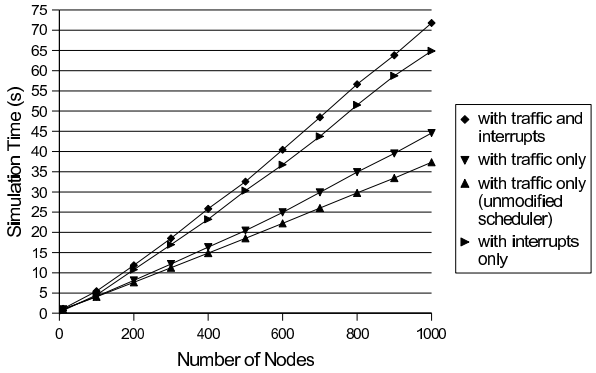

Figure 4: Number of nodes vs. simulation time with constant average node degree.

4.2 Connecting the Network Stack

Now that we have established the means of interaction between the `ns-2` simulator and the ScatterWeb application, we will describe the details of this interaction taking place: A shortened version of the `Net_send()` method of the `ScatterWebAgent` class is given in Listing 3. This method is called by the ScatterWeb application or the simulated firmware in order to send a packet over the network. After calculating the sequence number of the ScatterWeb packet in line 2, the method allocates a `ns-2` packet and its payload object in lines 4 and 5. The ScatterWeb packet is then copied into the the payload of the `ns-2` packet in lines 6 to 11, before lines 13 to 16 hand the packet down to the core of the `ns-2` simulation.

Listing 4 shows the `recv()` method of the `ScatterWebAgent` class, which is invoked by the `ns-2` simulation when a packet has been received by the lower networking layers of the simulated sensor node. In line 2, the `instance` variable is set to point to the current object. `instance` is a static field of the `ScatterWebAgent` class, that gives the C code of the application a reference to the object it is part of. Therefore `instance` needs to be adjusted whenever the `ns-2` simulation transfers control to a `ScatterWebAgent` object. Line 3 adjust the internal timers of the sensor node to the current simulation time. In lines 5 to 8, the newly received packet is made available to the ScatterWeb application. In lines 6 and 9, the state of the sensor node is updated to just having received a full packet. Finally in line 10, the main loop of the ScatterWeb firmware is executed once, just as if an interrupt had occurred on the real sensor node.

4.3 Further Adjustments

With the ability to send and receive packets, the crucial functionality of the glue code layer is operational. However, there are a few issues that still need to be considered:

- On the ScatterWeb platform, the hardware provides timer interrupts that allow for periodic tasks to be scheduled. The overhead resulting from a direct simulation of these interrupts is prohibitively expensive. Analogous to [9], we only simulate those timer interrupts that are relevant to the simulated application.

- As shown in line 3 of Listing 1, C++ namespaces are used to differentiate duplicate type definitions.

- In the ScatterWeb firmware headers, we used `ifdefs` to redefine certain data types for language compatibility.

With these minor fixes, the layer of glue code between ScatterWeb and `ns-2` is complete.

5. EVALUATION

Aside from accuracy which we only discuss qualitatively in this paper, the critical parameters in the performance of any simulation tool are memory usage and simulation time. In order to illustrate the applicability of our approach we must thus verify that the cost incurred by running a large number of ScatterWeb applications as part of the `ns-2` simulator is acceptable.

We have chosen the following three test cases to evaluate how the simulation behaves under different load conditions:

Traffic: Every 0.1 simulated seconds a random sensor node broadcasts a PING packet to which all nodes in radio range reply with a PONG packet. This test is used to see how the simulation behaves under network load.

Interrupts: A timer interrupt is fired every simulated millisecond on each sensor node, and every second a simulated LED is toggled. This test evaluates how the simulation behaves under system load on the individual sensor nodes.

Traffic and Interrupts: This test combines the two previous tests and checks whether there are unforeseen interactions between the two types of load.

All simulations run for 10 simulated seconds. In order to isolate the effects of increased communication between the sensor nodes, the average node degree, i.e. the average number of nodes affected by a simulated broadcast packet, was kept at 10 nodes. Where applicable, test have been run with network sizes of both 100 and 1,000 nodes. All data points are the averages of measurements taken on an idle Pentium 4 desktop PC running at 1.8 GHz with 1 GB of RAM.

Especially in the cases of big networks with high interrupt load, i.e. those cases that generate a large number of events in the simulator, we noticed scalability issues in the calendar queue scheduler [1] used by `ns-2`: In about 30% of the randomized test runs, the simulation time was by one order of magnitude longer than expected. We solved the

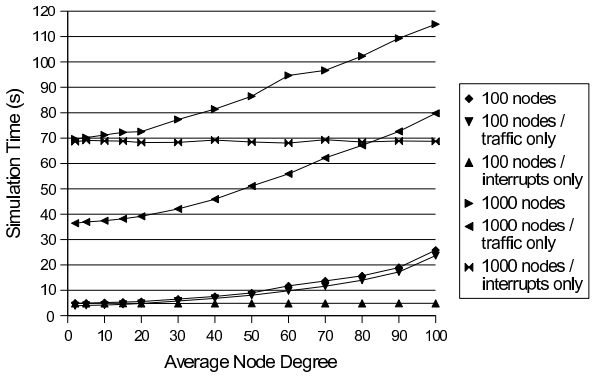

Figure 5: Average node degree vs. simulation time.

problem by initializing the calendar queue with the number of buckets set to the number of nodes in the simulation and the bucket width set to the reciprocal of the number of nodes. While working flawlessly under high event loads, we found that performance was adversely affected under very low loads. We therefore resorted to the unmodified `ns-2` scheduler for the test case with only 1,000 simulated nodes without interrupt load.

Figure 3 shows that the memory usage is independent of the test case and even of the scheduler. It increases proportionally with the number of simulated sensor nodes. Each additional sensor node adds about 345 kB to the simulation. Hence the simulation always ran completely in RAM, swap memory was not used.

Figure 4 depicts the total simulation time given different network sizes but with a constant average node degree, thus emphasizing the cost of node-local processing. We note that for all test cases the time required to complete the simulation increases proportionally with the size of the network. Even for a large sensor network with 1,000 nodes with traffic and interrupts the simulation time is merely 72 seconds.

Finally, in Figure 5 we illustrate how the simulation behaves under different average node degrees, thereby concentrating on the cost of communication between nodes. As expected, the time is constant for those test cases that do not involve network traffic. For the other test cases, we observe that simulation time increases polynomially with the average node degree. More interesting is, however, the fact that simulation time is well below two minutes for all average node degrees that are of interest for realistic WSN simulations.

6. CONCLUSION

There are numerous advantages of running unmodified ScatterWeb applications on the `ns-2` network simulator: Compared to the work required for deploying a large number of sensor nodes in order to test an application, the work of setting up a simulation is trivial. It is even possible to simulate very large sensor networks, for which it would otherwise be impossible to procure enough real sensor nodes. On a smaller scale, even the development of small applications is faster because the development cycle of implementing, compiling and testing does not involve the time consuming act of flashing the binary images onto the sensor nodes. Perhaps even more importantly, debugging of large distributed applications, such as routing or load balancing algorithms, is supported by tracing events in the network and the availability of standard debugging tools.

In our evaluation, we have established that ScatterWeb simulations scale well enough to allow for realistic WSNs comprising up to 1,000 sensor nodes to be conducted on standard PC hardware. The qualitative simulation results match our expectations and have already been utilized to improve previously existing applications, such as the implementation of the directed diffusion paradigm on ScatterWeb. Several other research projects use or have expressed interest in using our approach. We are currently working towards improving the quantitative accuracy of the simulation to correspond to the system parameters of the ScatterWeb platform, especially concerning radio model, energy consumption and execution time. The long term goal is to construct a hybrid network in which both simulated and real sensor nodes will be able to communicate with each other.

7. REFERENCES

[1] R. Brown. Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem. *Commun. ACM*, 31(10):1220–1227, Oct. 1988.

[2] J. Heidemann, K. Mills, and S. Kumar. Expanding Confidence in Network Simulation. *IEEE Network Magazine*, 15(5):58–63, September/October 2001.

[3] G. Judd and P. Steenkiste. Using Emulation to Understand and Improve Wireless Networks and Applications. In *Proceedings of NSDI 2005*, 2005.

[4] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[5] P. Mahadevan, A. Rodriguez, D. Becker, and A. Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. *ACM SIGMOBILE Mobile Computing and Communications Review (MC2R)*, Apr. 2006.

[6] The Network Simulator – ns-2. `http://www.isi.edu/nsnam/ns/`.

[7] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, Jan. 1997.

[8] ScatterWeb Homepage. Computer Systems & Telematics Working Group, Freie Universität Berlin, `http://scatterweb.mi.fu-berlin.de`.

[9] B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable Sensor Network Simulation with Precise Timing. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05)*, pages 477–482, 2005.

[10] H. Wu, Q. Luo, P. Zheng, B. He, and L. M. Ni. Accurate Emulation of Wireless Sensor Networks. In *Proceedings of Network and Parallel Computing, IFIP International Conference, NPC 2004*, pages 576–583, Wuhan, China, Oct. 2004.

[11] J. Zhou. TWINE: A Hybrid Emulation Testbed for Wireless Networks and Applications. In *IEEE INFOCOM 2006*, Barcelona, Spain, Apr. 2006.