A Simulation Tool For System Services In Ad-Hoc Wireless Sensor Networks

S.N.I. Mount, R.M. Newman and E.I. Gaura

Cogent Computing, Coventry University, UK
{s.mount, r.m.newman, e.gaura}@coventry.ac.uk

ABSTRACT

Whilst the physical design and instrumentation of ad-hoc wireless sensor networks (WSNs) has recently received a lot of attention, it is often assumed that the system software for such devices will simply copy the design of software for larger, fixed, context-ignorant computers. We believe that this lack of experimentation is not conducive to developing optimal architectures and services for WSNs.

In this paper we present SenSor: a simulation tool which facilitates experimentation with novel software architectures, by enabling a top-down approach to software design. This is in contrast with other simulation tools which offer a way to validate code for a particular hardware platform. We describe the top-down design method which SenSor facilitates with examples.

Keywords: wireless sensor networks, top-down design, software engineering

1 INTRODUCTION

Whilst the physical design and instrumentation of ad-hoc wireless sensor networks (WSNs) has recently received a lot of attention, it is often assumed that the system software for such devices will simply copy the design of software for larger, fixed, context-ignorant computers. Perhaps this is partly because we have yet to see a widespread proliferation of tools with which engineers can simulate the effects of deploying system software on ad-hoc, wireless networks. In this paper, we address this situation by taking an analogy from computer aided design.

Electronic computer aided design typically follows a linear progression from concept to architectural design through logical and physical design to layout and fabrication. We believe that the design of systems software for ad-hoc WSNs shares some of the constraints of hardware design. For example the finished product is a black box, the complexity of the system scales exponentially with new components and (unlike with other software) mistakes can be expensive to correct.

To ameliorate the difficulty of designing software for WSNs, we have built *SenSor*, a simulator with which software engineers can build algorithmic simulations of systems components. SenSor is sufficiently flexible that one, many or all systems software components and applications can be prototyped in the same simulation - and these can be run in any order. This allows an important separation of concerns during the development process. For example, an engineer might build a fault detection mechanism by first of all assuming a fully-connected network with a fixed topology. Later in the development of the fault detection algorithms, these constraints will need to be relaxed and the engineer can experiment with running the software on different types of network with different topology management strategies.

We believe that SenSor is distinct from other network simulators, which are hardware-specific, typically assume a layered network protocol model and often include an industry standard protocol stack. Issues in WSNs are then explored within these constraints. Instead, SenSor assumes nothing about the structure in which system services may be built, which leaves the user free to explore new models of system structure, away from the traditional OSI partition set.

1.1 Nomenclature

The word “sensor” is used in this paper with a variety of different spellings, each of which have a distinct meaning. Below is a brief glossary:

sensor A computational element with access to sense data from it’s environment.

probe That part of a sensor which generates sense data from the environment.

SenSor The SenSor simulation tool.

Sensor A Python class, within the SenSor tool, which represents a sensor.

1.2 This Paper

The remainder of this paper is organised as follows: Section 2 describes the SenSor workbench in detail. Section 3 gives an overview of the top-down design method that SenSor facilitates. Section 4 gives a small example of a simulation. Section 5 describes related work on simulations for wireless sensor networks and Section 6 concludes.

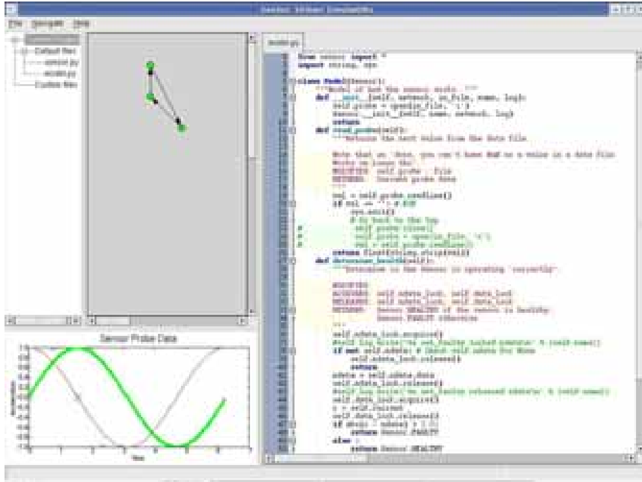

Figure 1: The SenSor Workbench

2 THE WORKBENCH

The SenSor workbench is composed of two parts. A graphical interface (Figure 1 is used to edit and animate simulations. The interface provides an API to it's graph and chart panels. This enables simulations to directly manipulate their visual representation. Sensor nodes are represented by circles and arcs between the circles are used to represent communications or topological structures, etc. Each node can register a change of state by changing the colour of its representative circle. For simulations with large numbers of nodes, colour is perhaps the easiest way for a user to track sensor behaviour.

Behind the user interface lies a set of Python classes which are subclassed to create each simulation. These superclasses are designed to represent the operating system (OS) of a sensor and are able to schedule and run prioritised tasks and provide communication between motes. These two tasks are the core of any OS for WSNs and are a subset of the capabilities of a production OS for a desktop environment. Since it is intended that users can experiment with new OS architectures, this minimalist approach means that other OS facilities can be specifically designed for individual simulations.

2.1 SenSor Architecture

To understand how SenSor is intended to be used it is important to appreciate the role of each of the base classes. Below we describe the contents of each class, what it is intended to represent and any important details about extending it.

Sensor objects represent the OS of a sensor and may be thought of as a blank piece of memory. Each sensor runs in it's own thread and subclasses need to lock and unlock data structures which they define. Each Sensor object has a Scheduler running a Task list, where each Task also has a reference to it's Sensor (so it can read/write to that Sensor's memory and call `send()` and `recv()` in order to communicate).
The `recv()` method which accepts a Packet object and places it on a thread-safe queue called `recv_q`. A `send()` method is provided which removes a Packet object from a thread-safe queue called `send_q` and "sends" that message either directly to another Sensor or via the `send()` method in the Ether class. `send()` and `recv()` may be overridden.

Scheduler objects represent the OS scheduler of each Sensor object. Each Scheduler has a Task list which it cycles through `n` times (where `n` is an integer argument to the constructor of Scheduler), executing a user-defined methods in each Task object.

Task objects define a particular job that a sensor may be scheduled to perform. Useful tasks may include obtaining probe data from a file or socket, interacting with the message queues in the Sensor class, managing internal data structures, etc. It is intended that Tasks may only interact with one another via shared data structures within Sensor classes.
The scheduler module provides extra classes to represent the priority of a task within the task queue (as with Unix systems, the lower the integer value, the higher the priority) and how often a task should be run. This means that a Task may be scheduled to run only once, or every `n` times the scheduler reaches it in the Task queue.

Packet objects define the format for packets of data being passed between Sensors. This format may be very high level (if the simulation is not immediately concerned with communication issues) or much closer to a real implementation.

Ether objects control the routing of messages between Sensor objects. They may be used to facilitate broadcasting, or to route each individual message in a simulation.

Simulation objects are the entry point for each simulation. They should contain an instantiation of the Ether and Sensor classes.

3 A TOP-DOWN DESIGN CYCLE

Perhaps the most important claim that we make for SenSor is that it enables the user to separate the various concerns of each simulation. Separation of concerns is inherently a subjective matter, and the examples given

here and (more fully) in Section 4 are intended to support our claim.

An example mentioned earlier involves building a fault detection mechanism by first of all assuming a fully-connected network with a fixed topology. Later in the design cycle these constraints will need to be relaxed and the user can experiment with running the software on different types of network and with different topology management strategies.

A simulation of a communication protocol, on the other hand, might be written as if each mote had no sensors attached and generate random numbers in place of real data. In this case, subclasses of Packet may need to be a realistic model of real packets used in a production network and would probably hold only a single binary value. In the previous example of a fault management system, packet design was irrelevant and Packet objects might have a high-level representation, perhaps containing string data and other high level structures.

The design cycle of a simulation strongly depends on the purpose of the algorithms being simulated. Even so, we have a coarse design strategy which we advocate. Here, algorithms are first designed formally, in an appropriate specification language (such as the π-calculus [4]). A formal analysis of the algorithms (perhaps its stability or complexity) can be undertaken. Next the algorithm is animated in SenSor, in a simulation which only realistically models the algorithm itself and it's necessary data structures. In a sequence of simulations various aspect of the algorithm might be tested. For example, in a fault detection simulation, random data can be introduced to each probe at random times. Next the simulation can be gradually refined until all relevant data structures are represented. In the fault detection example, the user might begin with an Ether object which explicitly routes all packets through the network. Later refinements might include a topology discovery and management module, wherein nodes would perform network routing themselves and the Ether object might be absent or only responsible for broadcasting Packets. When realistic simulations have been experimented with, production code might be written for a particular hardware platform. However, it may still be useful to add in another level of refinement before deployment, where a hardware-specific simulator is used to gather quantitative data about the scalability or robustness of a particular application. There are few simulators which offer a sufficiently high fidelity model of network behaviour to gather realistic results, although TOSSIM is one [2].

4 EXAMPLE SIMULATION

A full refinement from specification to is too large to describe in this paper. Instead we present a high-level simulation of a fault management protocol, described in [1] which has already been specified in the π-calculus

```
class ProbeReaderFromFile(ProbeReader):
    def read_probe(self):
        ln = self.probe.readline()
        if ln == "": # EOF
            sys.exit()
        val = float(string.strip(ln))
        self.sensor.probe_q.put(val)
        return
```

Figure 2: Probe reader class.

```
class FaultDetector(Task):
    def is_faulty(self, d1, d2, d3):
        if ( ... ):
            return 1
        return 0
    def fault_detect(self):
        if self.is_faulty(probe_q.get(),
                          probe_q.get(),
                          probe_q.get()):
            self.sensor.is_faulty = 1
        else:
            self.sensor.is_faulty = 0
        return
```

Figure 3: Fault detection class.

[4]. Here, each node in the network is assumed to have a nearest neighbour, whose presence has been determined by a topology management system which is outside the concern of this simulation. Each node obtains a reading from one of it's probes and places it in a queue called `probe_q`. Code for this task is in Figure 2. Note that the `ProbeReaderFromFile` class inherits from `ProbeReader`. Other subclasses of `ProbeReader` might obtain data from a socket, via a web connection, etc. Although we have omitted some details (such as constructors), the class is very short and deals only with the specific task at hand.

Fault detection is carried out by another subclass of `Task`, shown in Figure 3. Here, a separate method is used to determine, based on three readings[1] from the nearest neighbour and the sensors own readings, whether the probe is faulty. This information is used to set a flag in the `Sensor` object, which determines whether probe data is sent on to other probes, or whether data from the nearest neighbour is sent instead. Note that we have missed out a few details, for example, we haven't dealt with situations where `probe_q` holds less than three items, and we haven't saved the most recent value in the queue in case it needs to be sent out. Figure 3 gives an outline of the task.

Most importantly the `is_faulty` method is kept separate from the main fault detection algorithm. This is

[1] Three readings are used as the system described in [1] requires three readings for a neural net to detect faults.

```
class Reader:
    def __init__(self, data, time):
        self.data = data
        self.time = time
class ProbeReaderWithTS(ProbeReader):
    def read_probe(self):
        ln = self.probe.readline()
        if ln == "": # EOF
            sys.exit()
        val = float(string.strip(ln))
obj = Reading(self.val,
                        self.sensor.clock)
        self.sensor.probe_q.put(obj)
        return
```

Figure 4: Probe reader class with timestamps.

deliberate, as this method will be iteratively refined by subclassing `FaultDetector`. In initial simulations of the algorithm a naive method can be written (say, averaging the values from the neighbour and examining a threshold difference). Later the method can be refined to the one described in [1], where a neural network is used to detect faults.

Once we have a working simulation, we may wish to test the robustness of the algorithm against certain problems such as clock drift. In this initial simulation clock drift is irrelevant, as we are holding data in queues and we only have three tasks in the scheduler (the probe reader, the fault detector and a task to send data to the nearest neighbour). However, as the simulation is refined, more tasks will be added and the `probe_q` queue may be quite large by the time the fault detector examines it and it may need to be flushed. On the other hand, we may decide to only keep the latest three values from the nearest neighbour. In either case we may want to know whether or not it is useful to ensure that the probes current reading was taken at a time very close to the readings from the nearest neighbour and whether clock drift might affect this.

In this case, we need to alter the `ProbeReaderFromFile` class to time-stamp each reading and subclass the `Sensor` class, adding in a method for clock drift. In this high-level simulation, we can represent timestamped data in an object such as `Reader` in Figure 4 where `time` could be either a built in Python type or a real. A new subclass of `Sensor` need only override the `clock` method.

5 RELATED WORK

TOSSIM [2] and other lower-level simulation tools (e.g. [8]) all address a slightly different problem: how to gather quantitative data about an algorithm or protocol. These tools can be used to discover bugs in algorithms and implementations as well as investigating the scalability of network interactions. However, they are usually hardware-dependent and intended to be used late in the design cycle.

Work on animating algorithms [6], [5] is close to ours, but is mainly used in education. These tools are used to illustrate algorithms from the literature, not to allow students to experiment with new algorithms.

Other sorts of workbench (e.g. the Concurrency Workbench [3] and the Mobility Workbench [7]) allow for the step-though of theoretical representations of systems. These systems are useful in verifying high-level specifications but cannot be used in refinement.

6 CONCLUSIONS

We have described SenSor, a simulation tool for wireless sensor networks which facilitates the top-down, iterative design of systems software for WSNs. We claim that simulations in SenSor can be rapidly prototyped and animated and that the design of SenSor allows a useful separation of concerns within the design cycle.

REFERENCES

[1] E. Gaura and R. M. Newman. Microsensors, arrays and automatic diagnosis of sensor faults. In *IEEE International Conference on Advanced Intelligent Mechatronic (AIM2003)*, pages 360–366, Kobe, Japan, 2003.

[2] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137. ACM Press, 2003.

[3] F. Moller and P. Stevens. Edinburgh Concurrency Workbench user manual (version 7.1). Available from http://homepages.inf.ed.ac.uk/perdita/cwb/.

[4] R. M. Newman and E. Gaura. Using very large arrays of intelligent sensors. In *IEEE International Conference on Advanced Intelligent Mechatronic (AIM2003)*, pages 356–359, Kobe, Japan, 2003.

[5] J. Stasko. Animating algorithms with XTANGO. *SIGACT News*, 23(2):67–71, 1992.

[6] E. Sutinen, J. Tarhio, and T. Teräsvirta. Easy algorithm animation on the web. *Multimedia Tools Appl.*, 19(2):179–194, 2003.

[7] B. Victor and F. Moller. The Mobility Workbench — a tool for the π-calculus. In D. Dill, editor, *CAV'94: Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 428–440. Springer-Verlag, 1994.

[8] S. Y. Wang, C. L. Chou, C. H. Huang, C. C. Hwang, Z. M. Yang, C. C. Chiou, and C. C. Lin. The design and implementation of the NCTUns 1.0 network simulator. *Computer Networks*, 42(2):175–197, 2003.