BOSTON UNIVERSITY
Computer Science Department

CS559 - Sensor Networks Seminar

Colloquium "Programming Primitives for Wireless Sensor Networks"

presented by Matt Welsh, Harvard University
(scriber: Nahur Fonseca)

Introduction

Matt Welsh presentation had two parts. First he presented *nesC*, a programming language for networked embedded systems (aka sensor networks). This work was done with colleagues at Intel Berkeley and Berkeley University while he was a visiting researcher at Intel. Second he talked about on-going research of his group at Harvard University. They are trying to come up with programming primitives for sensor net applications. For instance, how to define spatial regions to coordinate work.

nesC Programming Language

Put simply, *nesC* is a restricted version of the C language which has hooks to ease the composition of event-driven applications. The authors of *nesC* traded the generality and power of C for more desirable features in sensor networks, such as, event-based concurrency model, whole-program analysis (to make reliable programs and aggressive optimization), and component-based architecture. Thus *nesC* is tailored for the kind of applications used in sensor networks.

nesC borrows many of its ideas from TinyOS[1]. They have implemented TinyOS in this new language. The core OS needs less than 400 bytes. To illustrate the use of *nesC*, they have also implemented Surge, a simple application of a sensor network that monitors temperature.

Next we present some main points about the features of *nesC*.

Component-Based Architecture *nesC* applications are built by assembling components.

Every component provides and uses some interfaces. For instance, a Timer component may **provide** an interface with commands to *start* and *stop* the timer, along with an event named *fired* triggered by the underlying hardware. On the other hand, it may **use** an interface Clock that is an abstraction of the hardware clock, which provides some commands to set it up, and some events like the *fired* above.

The type of component just described is a *module* in *nesC* if its code actually implements the commands. Another type of component is a *configuration*. A configuration does not implement commands, it only maps or connects its interfaces to ones which are implemented in other modules. Every application in *nesC* must have one configuration to put together all the modules it uses.

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. *System Architecture Directions for Networked Sensors*

An extract of modules and configurations from the *nesC* site[2] is worthy presenting for clarification.

```
module BlinkM {
  provides interface StdControl;
  uses interface Clock;
  uses interface Leds;
}
implementation {
  bool state; /* the state of the red LED (on or off)   */

  command result_t StdControl.start() {
    return call Clock.setRate(TOS_I1PS, TOS_S1PS);
  }

  event result_t Clock.fire() {
    state = !state;
    if (state) {
      call Leds.redOn();
    } else {
      call Leds.redOff();
    }

    return SUCCESS;
  }
  . . .
}

configuration Blink {
}
implementation {
  components Main, BlinkM, ClockC, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  BlinkM.Clock -> ClockC;
  BlinkM.Leds -> LedsC;
}
```

Here, BlinkM is a module and Blink is a configuration. We can see that BlinkM actually implements some of its commands, for example the *start* command which was defined in the interface StdControl that it provides. (The definition of the interface type StdControl is not shown.) On the other hand, the configuration of Blink only wires the interfaces of the components (or modules) it uses to their actual implementations. For instance, the Clock interface used by BlinkM module is connected to ClockC module, which is an abstraction of the clock hardware.

[2]http://berkeley.intel-research.net/dgay/nesc/

Event-based Concurrency Model

There are two ways of running code in *nesC*, using tasks or events.

Tasks are post to the scheduler, which eventually runs each task until completion. They do not preempt other tasks. Thus, memory shared between tasks is race-condition free, or concurrency-safe. In order to ensure low task execution latency, every task should be short.

The short execution time requirement on tasks prohibits the use of busy-wait to read sensors. Therefore *nesC* uses split-phase operation. In one phase, a request is made to a sensor, in the other, an event is triggered to indicate the request was completed.

Events are the second type of execution. An event is triggered by a hardware interrupt. Events can preempt tasks or other events, thus creating the possibility of concurrency errors, or race-conditions. To solve these problems in *nesC*, a programmer must either move the access of shared memory to tasks, or introduce atomic blocks. Inside an atomic block, interrupts are disabled; thus these blocks must be short.

Whole-program Analysis

At compile time, *nesC* can do data-race detection and also do aggressive in-lining to improve run-time performance, both in terms of size and CPU cycles.

At compile time, all modules of an application are put together in an unique C file.

Because *nesC* programs can not use dynamic memory, race-detection is much more easier to do. However, not all race-detection errors are detected, and also, false positives are possible.

All the code optimizations are left for the C compiler.

Critique and Discussions

These are some critique and discussions that were raised during the presentation:

- There has not been a thorough evaluation of the language. They only presented the size and speed of the programs before and after optimization. They do not compare it to other languages.
- It was said that AIDA language has similar configuration modules.
- The beauty of *nesC* is its simplicity. It creates a design environment that fits the requirements of sensor net applications. However there is nothing essentially new in this work, or anything that could not be done with a bunch of perl scripts during the development phase.

Macro-programming for Sensor Networks

This is a new topic which has been studied by Matt Welsh an this colleagues at Harvard University.

The main motivation idea is that today's sensor network applications are developed in a "node-centric" approach. The problem with this approach it is too low level. One would like to borrow some abstractions of the parallel computation model to the filed of sensor net applications.

Macro-programming primitives should expose tradeoffs relevant to these applications. For instance, one could trade *energy X correctness*. If energy is not a problem, then one can retransmit use a reliable transmission protocol for every data. On the contrary, if energy is more important, one can do a best effort and try to retransmit a lost packet only, say, 2 times.

Other primitives that were also discussed:

Abstract regions An abstract region is formed by sensors that are sharing a memory, as a result, they can aggregate their information before sending it upon a request. For instance, in an object tracking application, the abstract region moves with the object.

Work Coordination Nodes can collaborate to execute a task, for instance, finding a contour. In this case, nodes need to coordinate themselves to find at each side of an object they are, and once the contour is found, send data only for the nodes on the contour back to the sink.

Current work and Vision

Currently, they are working on spatial primitives, such as areas, object tracking etc.

They have a SensorLab (after PlanetLab). This is a network of motes, installed in the corridors of Harvard University, and attached to network points, so that code can be upload to them from the network. This testing environment is available for people interested in testing their programs on the motes. (More details by e-mail mdw@eecs.harvard.edu)

As a future/vision work, they want to create temporal primitives.

Conclusion

In the first part of the talk, Matt Welsh presented *nesC* – a programming language tailored for sensor network applications was presented. Its main features are component-based model, event-driven concurrency and whole-program analysis. The advantages of such a design are 1) it translates directly to TinyOS - an operating system that has been used in many sensor net applications; 2) its race-detection and optimization at compile time make the programs more reliable; 3) its modular design makes the creation of new applications easier, by reusing its modules (e.g. clock, sensor reading, etc.).

On the second part, it talked about on-going research at Harvard about Macro-programming primitives for sensor networks, in particular, spatial primitives, such as areas, objects, etc.