HiNRG Technical Report: 22-09-2008
The Latte Programming Language

Răzvan Musăloiu-E.
razvanm@cs.jhu.edu

1 Introduction

Several attempts have been made to construct high-level languages for implementing TinyOS-based applications. Some of these attempts have taken the form of general purpose languages (TinyScript and Mottle in Maté [3]), some were designed for a specific application domain (TinyDB [4], Regiment [6], WaveScope [5], etc), while others explored specific programming paradigms (DSN [1]). The method used to run code written in each language also varies from virtual machines (Matté), to interpreters (SensorScheme [2]), to binary code implemented in nesC (DSN, Regiment, WaveScope).

The Latte programming language is another attempt, with similar goals. Specifically, Latte makes several contributions:

- It brings to sensor networks JavaScript, a popular scripting language used in many other application domains;
- It allows high-level emulation inside JavaScript enabled web browsers;
- Latte programs compile to binary code via a source-to-source translation into C.

The benefit of the first two contributions is that now sensornet applications can be rapidly prototyped without the need for compiling them and running them either natively on a mote or in a simulator such as TOSSIM. The fact that an application written in Latte compiles to C, however, means it can be interfaced with TOSThreads and ultimately run on a mote.

2 The TinyOS Object

The entry point for a Latte program is through a function called `main()`. From within `main()`, Latte programs interface with a TOSThreads kernel using a global object called TinyOS. Table 1 presents some of the most common service abstractions available through this object.

3 Language Constructs

Just as in JavaScript, users declare variables using the `var` keyword and declare arrays using the `new` operator. The typing of these variables as well as the typing of return values

Objects	Functions
TinyOS	NODE_ID
TinyOS.Leds	set(value), get()
TinyOS.Led[*n*]	on(), off(), toggle()
TinyOS.Radio	ALL, start(), stop()
TinyOS.Radio.AM[*am*]	send(addr, msg, len), receive([timeout])
TinyOS.Serial	start(), stop()
TinyOS.Serial.AM[*am*]	send(addr, msg, len)
TinyOS.Sensor[*n*]	read()
TinyOS.Block[*vol*]	getSize(), erase(), crc(addr, len) read(addr, len), write(addr, obj)
TinyOS.Log[*vol*]	append(obj), seek(cookie), read()
TinyOS.Time	now()
TinyOS.Thread	create(function[, description]) sleep(interval)
TinyOS.Collection	send(), receive(), setRoot()
TinyOS.Random	get16(), get32()

Table 1: The objects available in Latte through the TinyOS object. The parameters in square brackets are optional.

to functions are inferred rather than explicitly declared. Such type inference is possible because the prototypes of all external functions are known at compile time.

The equivalent of C structures can be defined in Latte by calling a built-in function we wrote called `Struct`. The only parameter to this function is an associative array whose keys contain the names of the fields in the structure. If a structure's element's type must be precisely defined (e.g., when defining a packet structure to be sent over the serial link or the radio), predefined values (`uint16_t`, `uint32_t`, etc.) can be used in dummy assignments to indicate the desired type. To declare a field without setting its type, the `undefined` type can be used.

Because Latte is directly translated to C, we've removed all JavaScript features related to run-time compilation.[1] Due to limited usefulness in this application domain we also chose not to implement some built-in JavaScript objects like `Date` and `Number`.

4 Latte Emulator

Because Latte is a subset of JavaScript, Latte programs are also valid JavaScript programs. This makes it possible to turn any JavaScript enabled browser into a development tool

[1] A similar thing is done in ECMAScript Compact Profile, a JavaScript standard designed for resource-constrained devices.

capable of emulating code runnable on a mote. To make this possible we implemented `TinyOS.js`, a JavaScript library implementing all of the objects listed in Table 1.

To implement this emulator we had to overcome two hurdles: the fact that JavaScript does not support multi-threading, and the limitation of only being able to emulate a single mote. We make it possible to support non-preemptive multithreading through the use of a combination of a special JavaScript keyword[2], `yield`, and the `setTimeout()` function. We enabled the emulation of multiple motes by adding a special `sim` function to Latte which contains calls that instruct the emulator how to respond to send and receive commands. In the example given in Appendix A the `sim` function indicates that a `Request` packet with AM ID 0 will be received 1000 ms after calling the `receive` function (line 28). This example presents a complete environmental monitoring application written in Latte. It demonstrates many of the Latte's feature including the use of the TinyOS object, structures, arrays and blocking calls.

Experience has shown us that compiling a Latte application into code based on the TOSThreads API is much easier than compiling it for use in standard TinyOS.

We previously implemented a version of Latte that compiled code runnable in standard TinyOS, but it made the language much harder to use and much less elegant. Additionally, when compiling this version of Latte for dynamic linking and loading, the callbacks associated with making TinyOS split-phase calls increased the number of entry points that the loader need to handle as well as complicated the unloading process significantly. Latte's ability to compile down to standard C for easy integration with TOSThreads and its TinyLD loader has implications beyond Latte itself; it opens the door for the development of other high-level languages providing similar features.

[2]Only available as of JavaScript 1.7.

A Environmental Application in Latte

```
SENSOR_NUM = 3;
SAMPLE_PERIOD = 2048;
DATA_SIZE = 90;

RadioData = TinyOS.Radio.AM[0];
LogData = TinyOS.Log[0];

readings = new Array(SENSOR_NUM + 2);

Request = Struct({'offset': uint32_t,
                  'len': uint32_t,
                  'delay': uint16_t});

Packet = Struct({'offset': uint32_t,
                 'nextOffset': uint32_t,
                 'data': new Array(DATA_SIZE)});
packet = new Packet();
busy = false;

function main()
{
  var len;
  request = new Request();
  TinyOS.Thread.create(sample, "sample");
  yield TinyOS.Radio.start();

  while (1) {
    request = yield RadioData.receive();
    yield LogData.seek(request.offset);
    for (len = 0; request.len; request.len -= len) {
      len = min(DATA_SIZE, request.len);
      packet.data = yield LogData.read(packet.data,
                                       request.size);
      yield RadioData.send(0xFFFF, packet, len);
      yield TinyOS.Thread.sleep(request.delay);
    }
  }
}

function sample()
{
  var i;
  while (1) {
    for (i = 0; i < SENSOR_NUM; i++) {
      readings[i] = yield TinyOS.Sensor[i].read();
    }

    yield LogData.append(readings);
    yield TinyOS.Thread.sleep(SAMPLE_PERIOD);
  }
}

function sim()
{
  TinyOS.Sim.addMessage(0, 1000,
                 new Request(0, 5*DATA_SIZE, 10));
}
```

References

[1] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems (SenSys)*, 2007.

[2] L. Evers, P. Havinga, J. Kuper, M. Lijding, and N. Meratnia. SensorScheme: Supply Chain Management Automation using Wireless Sensor Networks. In *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2007)*, Sept. 2007.

[3] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the Second USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, April 2005.

[4] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[5] R. Newton, L. Girod, M. Craig, G. Morrisett, and S. Madden. Design and evaluation of a compiler for embedded stream programs. In *LCTES '08: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools*, June 2008.

[6] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN)*, 2007.