Flexible Sensor Network Reprogramming for Logistics

Leon Evers Paul Havinga Jan Kuper
University of Twente
Drienerlolaan 5
7522 NB Enschede, the Netherlands
Email: {l.evers,p.j.m.havinga,j.kuper}@utwente.nl

***Abstract*—Besides the currently realized applications, Wireless Sensor Networks can be put to use in logistics processes. However, current WSN software platforms cannot provide the flexibility and safety needed. This paper presents SensorScheme, a runtime environment based on semantics of the Scheme programming language, used to realize a logistics scenario. SensorScheme is a general purpose WSN platform, providing dynamic reprogramming, memory safety (sandboxing), blocking I/O, marshalled communication and compact code transport. We illustrate the use of our platform and provide experimental results that show its speed of operation and energy efficiency.**

I. Introduction

In the foreseeable future wireless sensor networks can make a great impact in the supply chain management business. WSN nodes attached to crates, roll containers, pallets, and shipping containers can be programmed to actively monitor the transportation process.

This paper describes a WSN enabled logistics application scenario in section II, that verifies proper handling conditions of goods like temperature for fresh foods and actively monitors every transported item to significantly reduce delivery delays and loss or theft of goods, which cause a significant loss of revenue.

Programming these devices requires a level of flexibility and security beyond what is currently offered by WSN system software. In this paper we present a platform called *SensorScheme* that is able to deliver on the requirements posed by active tracking logistics scenarios. SensorScheme is an interpreter to execute dynamically loaded application code for WSN platforms based on the Scheme programming language.

Besides tracking logistical processes, SensorScheme is also beneficial to many other, more 'traditional' WSN applications. SensorScheme bears many similarities with the Maté [1] virtual machine platform in terms of functionality. But due to its different design, SensorScheme can provide a wider range of capabilities, and allow richer applications to be executed on it. Section III describes the design of SensorScheme. Section IV discusses application implementation issues using the logistics scenario as an example. We then evaluate SensorScheme's performance in section V, and conclude in section VI.

1-4244-1455-5/07/$25.00

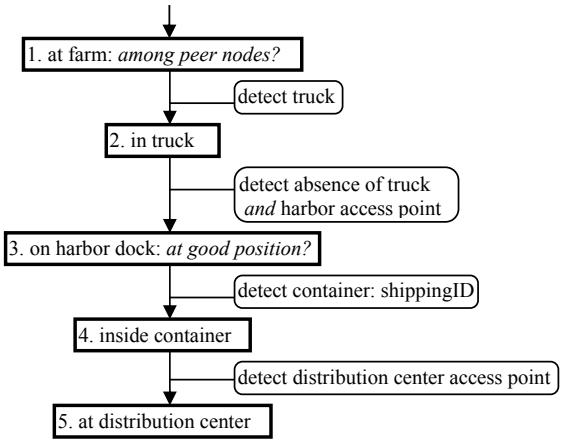

Figure 1. State diagram of the transportation process

II. Scenario

This section discusses a small transportation scenario to illustrate how WSN nodes can be used in logistics. Consider a shipment of bananas as it travels from the farm near Rio de Janeiro, Brazil to a supermarket distribution center in Rotterdam. The bananas are packed in boxes stacked onto pallets, each equipped with a tracking device. From the farm, these pallets travel in trucks to a loading dock at the harbor, where they are loaded into shipping containers that carry them all the way to the supermarket chain's distribution center.

Figure 1 shows a state diagram of the stages and transitions that these pallets will go through during the transportation process from the farm to the distribution center.

While a pallet is waiting at the farm to be loaded into the truck it tries to verify whether it is positioned correctly, near other pallets that are to be loaded into the same truck. It does this by comparing its destination and contents with (the majority of) peer nodes on other pallets nearby. When a pallet is not positioned correctly or no peer nodes are found, it should raise an alert.

Next, the pallets are loaded into the truck transporting them to the harbor. Nodes can detect being loaded by 'hearing' a device inside the truck, at which point they'll make the transition to stage 2. While in the truck, pallet nodes have

to detect being taken out of the truck, which is concluded from absence of the truck, and presence of the wireless infrastructure (access point) of the harbor loading dock.

When unloaded on the dock, the devices again verify whether they are positioned correctly to be reloaded into shipping containers. When placed incorrectly, it can directly send an alert message to the dock infrastructure that will inform workers to correct it.

For the last stage of the transport, the pallets are loaded into containers. These can be recognized by a matching shipping ID programmed into each container. Finally, when the container arrives in the distribution center, pallet devices sense the distribution center access point and make the state transition.

A. Application requirements

Upon every new transport, each device is reprogrammed with a small executable program that in effect tracks the bananas as they move through the logistics process. A number of code reprogramming mechanisms exist for wireless sensor network platforms, such as XNP [2] and Deluge [3] for the TinyOS platform [4]. These replace the entire program image (typically are a few tens of kilobytes in size) as a whole, taking time in the order of minutes, which would take too long for our scenario and cost significant amounts of energy.

Safety must also be considered: transporters may not be able to 'break' the devices (ie. modify their software operation). Therefore, an interpreter or virtual machine is a more suitable approach for our scenario, acting as a *sand box* to shield off the hardware. Misbehaving or buggy applications thus cannot crash a device or damage its critical functions. Furthermore, only the application code needs to be transported to the devices, which significantly reduces the size of transported code.

Maté [1] is a virtual machine designed specifically for memory-constrained WSN devices. Unfortunately, Maté can contain only truly tiny applications and exclude our application scenario from being implemented on top of Maté because of the lack of VM-implemented procedure libraries, container data types, and limited communication capabilities.

III. SensorScheme

SensorScheme is a novel interpreted platform for WSN's used to implement our application scenario. It uses execution semantics of the programming language Scheme, hence its name.

A. Program representation

In SensorScheme, program fragments take the shape of a specially formatted linked list of memory cells. Figure 2 summarizes the SensorScheme grammar. The operational semantics is as in regular Scheme. The grammar describes the set of legal SensorScheme expressions. Its first three constructs represent SensorScheme's lambda-calculus core: variable reference, application and lambda abstraction. The next four constructs are the special forms needed to make a minimally complete Scheme implementation: global variable definition, variable assignment, conditional evaluation, and literal quotation. Then primitive procedure invocation, and the last four rules represent constant reference (numbers, true, false, empty list).

```
exp ::= sym
      | (exp exp ...)
      | (lambda (sym ...) exp)
      | (define sym exp)
      | (set! sym exp)
      | (if exp exp exp)
      | (quote exp)
      | (prim exp ...)
      | num | #t | #f | ()
```

Figure 2. A grammar for SensorScheme

Using the SensorScheme program representation and execution model, programs are represented as data structures that can be operated on. One of the operations that can be performed on these programs-as-data structures is to execute or *evaluate* them, using the `eval` primitive. SensorScheme relies on this principle for loading new programs at runtime: When a node receives a program-as-data from the wireless network interface, it will invoke the `eval` primitive on it, which executes the contents of the program. This program then calls `define` to add new global procedures and event handlers.

B. Memory

SensorScheme is designed specifically for the small memory size of WSN platforms. All memory is allocated from a single pool of small equally-sized cells. These cells correspond to Scheme cons-cells, and each contains two data members which can be a reference to any other value, such as another cons-cell, a number, booleans (`#t`, `#f`) or the empty list (`()`). Cells can be combined to form lists, trees, association lists, and so on. Garbage collection is used to reclaim unused cells in the memory pool.

The global memory pool stores application data as well as program code and interpreter state like the call stack, local and global variable bindings and scheduling queues. Garbage collection reclaims unused cells in the memory pool.

C. Task Scheduling

WSN nodes have an inherently event-based nature, reflected in today's WSN operating systems. Program execution is organized in a number of short-running tasks, which can be scheduled to execute in response to some event. In general, tasks run until completion, starting after the previous one has ended. SensorScheme is designed to run on these sensor network operating systems, and is implemented as a single operating system task. The 'OS-level' SensorScheme task defines its own scheduling mechanism. When an event occurs, a SensorScheme task is scheduled. These tasks are handled in FIFO order. The kinds of events that can occur in

a.
```
(define (time-loop)
  (call-at-time (+ (now) 5) time-loop)
  (bcast (list 'gossip 1 2 3)))
```

b.
```
define-handler (gossip a b c)
  ; react to the gossip message
  ...)
```

Figure 3. Example code snippets showing the use of timer and communication events

SensorScheme are 1) reception of a network message and 2) firing a timer, and 3) hardware events originating from sensors.

Timer events perform a computation scheduled at a predetermined moment in time. SensorScheme provides a primitive procedure `call-at-time` that takes as parameters the scheduled time and the computation as a zero-argument function. At the scheduled time, the computation is executed as an event handler.

Use of timer events is best illustrated by an example. In the code sample in figure 3(a) the `time-loop` function repeatedly schedules itself at 5 second intervals to broadcast a message.

D. Communication

Wireless network communication is one of the crucial components to WSN platforms. In SensorScheme communication is designed to be compact and simple.

All SensorScheme data is contained in memory cells of a small set of data types, tagged with a type code. When sending, devices encode the message (a data structure) into a linear representation. Upon reception the receiver can decode the message into (a copy of) the same data structure from the linear representation.

A SensorScheme message consists of a header symbol and a number of data items. The message header refers to the global function that will handle the message, and the data items in the message act as parameters to the handler function. The primitive procedure `bcast` simply sends a message to all nodes within transmission range. It accepts a single parameter: a list containing the message content. See figure 3 for a code sample containing `bcast`. The `bcast` primitive encodes the message content in linear form into one or more physical packets, depending on the size of the message content.

Receivers of this message decode the content of each packet into the corresponding data items. Then the message handler denoted by the header symbol is looked up and scheduled to run as an event handler. The code sample of figure 3 (which is loaded at all nodes in a WSN) shows how communication takes place. Nodes broadcast a message containing header `gossip` and three data items, the values 1, 2 and 3. Receiving nodes schedule procedure `gossip`, which takes the source ID of the sending node as an implicit parameter bound to `src`, and bind the three data items of the message to *a*, *b*, and *c*.

Communication of SensorScheme application code is straightforward: the data structure describing the code can be packed inside a SensorScheme message, and on reception 'eval'-ed to load and execute. There is a primitive procedure

```
 1 ; requests the value of given keys from all neighbors
 2 (define (peer-dict timeout key)
 3   (let ((reqid (rand)))
 4     (bcast (list 'peer-dict-hdl reqid key))
 5     (set! reqs (cons (cons reqid ()) reqs))
 6     (call/cc (lambda (k)
 7       (call-at-time (+ (now) timeout)
 8                     (lambda ()
 9                       (k (cdr (assoc-and-remove!
10                                reqid reqs)))))
11                (exit)))))
12
13 ; handler invoked at neighbors
14 (define-handler (peer-dict-hdl reqid key)
15   (bcast (list 'peer-dict-rpl src reqid
16                (cdr (assoc key global-dict)))))
17
18 ; handler receiving values from neighbors
19 ; called at requesting node
20 (define-handler (peer-dict-rpl dst reqid val)
21   (when (= dst id)
22     (let ((req (assoc reqid reqs)))
23       (set-cdr! req (cons val (cdr req))))))
```

Figure 4. peer-dict source code

called `eval-handler`, that performs only that, making it possible to bootstrap an 'empty' SensorScheme node. The eval-handler primitive is defined as:

```
(define-handler (eval-handler sexpr)
  (eval sexpr))
```

and can be used in the following way:

```
(bcast (list 'eval-handler
             '(define sqr (lambda (x) (* x x)))))
```

The SensorScheme communication interface poses no restrictions on the size of a message, and the message contents can not be assumed to fit inside a single packet used by the physical network interface, and multiple packets must be used. We will not discuss the details of encoding and packing of these messages and correct unpacking on the receiver in this paper due to space constraints.

IV. Discussion

We show by example how SensorScheme can serve as an implementation platform for our logistics scenario, with an implementation of one of the states of figure 1. The example shows construction of communication protocols and blocking call creation, especially useful for communication-oriented WSN applications.

Figure 4 shows a SensorScheme implementation of the `peer-dict` procedure. It takes a key and timeout value as parameters, and communicates with all direct neighbors to find their dictionary entries of given keys. The procedure blocks the calling task, returning only after *timeout* seconds.

Function `peer-dict` sends a request to all neighbors (line 4) containing a unique request ID (created at line 3) and the requested key, and stores the request ID in the `reqs` dictionary (line 5). The `call/cc` invocation on line 16 creates a continuation, used to return to the function's caller after the timeout. At line 7 a timer is set up to signal the end of the timeout. Finally, a call to exit (line 11) aborts the current task, allowing other events to be processed while `peer-dict` is blocked.

Code size	program	library	all	
Source code	963	1032	1991	chars
Net-encoded	176	186	362	bytes
In memory	181	194	375	cells
Available			1975	cells

Table I
CODE SIZES OF EXAMPLE PROGRAM

program time	208	ms	
program energy	1.27	mJ	7%
OS time	130	ms	
OS energy	0.80	mJ	4%
# messages TX	14.4	msgs	
# messages RX	119.6	msgs	
air time	455	ms	
radio energy	16.45	mJ	89%
Total energy used	18.52	mJ	100 %

Table II
EXECUTION STATISTICS

The message broadcast at line 4 is handled by the `peer-dict-hdl` handler at all receiving nodes (lines 14-16). These nodes simply reply with a `peer-dict-rpl` message containing the senders' ID, the original request ID and their global dictionary value associated with the key.

Upon reception of `peer-dict-rpl` messages at the requesting device (lines 20-23), it looks up the request ID in the `reqs` dictionary, and extends the value list with the value just received (line 23).

When after *timeout* seconds the timer expires (line 8-10), the request ID is once more looked up, and removed from the dictionary. Then, with a call to the continuation bound to variable *k*, procedure `peer-dict` is returned, with the value list created in subsequent invocations of `peer-dict-rpl` as return value.

V. EVALUATION

A. Code size and memory use

Before we will discuss the performed evaluations, we first consider the size of the program code, shown in table I. To enable running the program presented in figure 4, some standard library functions are also made available on the nodes, like `every` and `assoc`. Table I shows that the library code is just slightly larger than the application itself. Compared to the source code, the compact network encoding used reduces it to less than a fifth during transmission across the network. In memory, the program code size is larger, since it is contained in memory cells, and consumes a total of 1500 (375×4) bytes. That leaves another 1975 cells available for additional program code and for use during program execution, by the call stack, global and local variables, scheduling and timer queues and application data.

Especially during transmission of program code, SensorScheme produces a very compact representation that enables fast and energy-efficient reprogramming.

Energy use is the crucial performance factor for battery-operated devices, so we have measured the energy used by execution of SensorScheme programs. We use a processor emulator in a simulated network of 20 nodes, each periodically running the `peer-dict` function of figure 4. This represents a real-world situation, since only one itinerary verification would be taking place at any given time. All energy calculations are based on the data sheets of the hardware components of our implementation platform.

Table II lists the running time per invocation of the `peer-dict` function. For each such a period, the SensorScheme code takes only 208 ms execution time. With a period duration of 10 seconds (the minimum with twice a timeout of 5 seconds) this is just two percent of CPU time spent.

Communication takes a significant fraction of the total energy use on WSN nodes. Table II shows the number of messages sent and received per period, and the energy spent on additional computation by the OS, and the energy use of the radio during sending and receiving. (Before sending, the radio needs to power up taking an additional 3 ms, included in the air time.)

Finally, taking these three sources of energy use together, shows that most energy is used by the radio power during communication (89 %), while computation time takes only 7 % of the total energy spent. We have not taken into account other sources of energy use like MAC protocol overhead (idle listening) and sensor readouts, which only reduce the fraction of energy used by program interpretation. In conclusion, the effect of interpretation overhead on the total energy budget is minimal, accounting for no more than 7 percent.

VI. CONCLUSION

We discussed a logistics application example requiring frequent change of on-device applications, which are possibly insecure, and require compact program representations. Virtual machines have typically been used to meet similar requirements. For wireless sensor networks, existing solutions have high resource requirements, or provide too little functionality to satisfy the application requirements, mainly due to memory-starved WSN platforms. By making better use of the little available memory, SensorScheme is able to provide a wider range of functionality. Despite their interpreted nature, SensorScheme programs cause only marginal additional energy use and no significant delays due to program interpretation.

REFERENCES

[1] P. Levis, D. Gay, and D. Culler, "Bridging the gap: Programming sensor networks with application specificvirtual machines," UC Berkeley, Tech. Rep. CSD-04-1343, Aug 2004. [Online]. Available: citeseer.ist.psu.edu/levis04bridging.html

[2] Crossbow Technology, "Mote in-network programming user reference," Crossbow Technology, Inc., 2003, http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf.

[3] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programmingat scale," in *Proceedings of the 2nd international conference on Embedded networked sensorsystems*. ACM Press, 2004, pp. 81–94.

[4] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E.Culler, and K. S. J. Pister, "System architecture directions for networked sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104. [Online]. Available: citeseer.ist.psu.edu/382595.html