The 18th Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC'07)

DYNAMIC SENSOR NETWORK REPROGRAMMING USING SENSORSCHEME

Leon Evers
University of Twente
Enschede, the Netherlands

Paul Havinga
University of Twente
Enschede, the Netherlands

Jan Kuper
University of Twente
Enschede, the Netherlands

ABSTRACT

Building wireless sensor network applications is a challenging task, and it has become apparent that it is crucial for many sensor networks to be able to load or update the application after deployment. Since communication is a scarce resource and costly in terms of energy, it is important to minimize code size when reprogramming WSNs in the field. This paper introduces SensorScheme as a novel interpreted WSN platform for dynamically loading sensor network applications. It is based on the semantics of the Scheme language and is equipped with high-level programming facilities such as higher-order functions, garbage collection, communication by automatic marshalling of data items, and co-routines to implement blocking I/O operations. SensorScheme makes efficient use of the little available memory in WSN nodes, uses a very compact program notation during wireless transmission, and provides a safe interpreted execution environment, preventing malfunctioning programs from crashing the device. We illustrate the use of SensorScheme and evaluate its code compactness and energy-efficiency.

I INTRODUCTION

As the field of wireless sensor networks matures, application complexity increases, and the need arises for more powerful programming methods. Specifying the program after deployment and changing it during operation has become necessary, since the precise application requirements and processing methods are often not fully known until a sensor network is actually deployed. A range of possible solutions have been proposed [11, 2, 9, 10, 3, 8] allowing sensor nodes to be reprogrammed in the field. Since network bandwidth is a precious commodity, most of these put strong focus on reducing program size. One such approach is to use high level programming abstractions that yield compact programs. TinySQL queries, for example, are very compact 'programs' specifying how sensor data should be aggregated in-network towards the collection root. Likewise, in the Maté WSN virtual machine platform programs are transported into the network as small capsules of instructions specifying high-level operations.

While effective at reducing application size, these approaches also reduce expressiveness. TinySQL queries all follow a very rigid shape, allowing only aggregation of data along a tree. Maté programs must be truly tiny, consisting of only a few hundred bytes of code, and just a dozen or so application data values. In the case of Maté this limitation of expressiveness is primarily caused by the severe resource constraints of WSN hardware, and the significant portion of it taken by the implementation of the VM runtime. Other platforms based on program interpretation, such as the Sun SPOTs [14] and SensorWare [2] require larger platforms than the current mote-class WSN hardware.

More effective use of a WSN node will therefore involve keeping the (in-field loadable) program compact while providing enough expressiveness to specify a variety of WSN applications. Program specifications must be both compact and expressive, and resources – especially memory – must be allocated for optimal use by the application.

One way to minimize the size of programs to be transmitted across the network is to refrain from translating the source text into a more verbose format such as (virtual) machine instructions, but to directly use the abstract syntax of the program text. The choice of semantics expressed in the abstract syntax can further influence compactness: by using high a level language, many operations can be expressed in a few powerful constructs, thereby improving the program's compactness.

The next major design consideration is memory allocation. By the nature of dynamically loaded programs, the sizes of data structures and program code are not known at compile time of the runtime environment, and must be allocated dynamically upon loading and execution of the program. For memory-scarce WSN hardware, dynamic allocation generally is not feasible, because of the risk of fragmentation. Allocating only fixed-sized structures can better guarantee memory availability for long-running WSN applications.

This paper presents *SensorScheme*, a WSN platform for dynamic program loading and execution (in Section II). SensorScheme is based on the semantics of the Scheme programming language and is designed to meet the demands of sensor network applications. It puts strong focus on efficient code transport by separating the format while transmitted from the in-memory code storage while executing. Furthermore, it enables fragmentation-free dynamic memory allocation, and automatic memory management to ensure memory safety. SensorScheme also puts strong focus on writing communication-centric applications by automatic serialization of data items, and enable blocking I/O calls through the use of continuations.

To illustrate the power and capabilities of SensorScheme, we present an application scenario in section III, and show how SensorScheme can be used to build efficient and compact WSN programs as a result of its high-level nature and powerful abstractions.

Interpretation of programs, rather than execution in the CPU's native instruction set, significantly increases computation time. Because the CPU cannot spend that time in sleep mode, this causes an increase in energy use. We use the aforementioned example application to evaluate in section IV the impact of program interpretation on the total energy use of a WSN node.

We finish this paper with a conclusion in section V.

1-4244-1144-0/07/$25.00

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

prim ::= cons | car | cdr | set-car! | set-cdr! | ...
       | null? | pair? | symbol? | number? | ...
       | + | - | * | / | < | = | > | ...
       | eval | apply | call/cc | ...
       | call-at-time | bcast | sensor | ...
```

Figure 1: A grammar for SensorScheme

II SensorScheme

SensorScheme is a novel platform for WSN's enabling (re)programming of the network after deployment using the wireless link. SensorScheme is designed for use on WSN hardware platforms like the Telos [12] motes, taking into account their resource restrictions.

The SensorScheme platform uses execution semantics of the programming language Scheme, hence its name. It is not an implementation of the Scheme language, however, and creating SensorScheme programs does not require the use of the Scheme language or syntax. The code examples in the following sections do use Scheme syntax.

A 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, each containing 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.

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.

B Program representation and execution semantics

SensorScheme programs take the shape of a specially formatted linked list of memory cells, containing the abstract syntax tree (AST) of the program. Figure 1 lists a grammar of the SensorScheme syntax tree (written in Scheme bracket notation). The operational semantics of these rules is as in regular Scheme.

The first rule, *exp*, 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).

The set of defined primitives, some of which are given by the second rule, *prim*, includes most of the common Scheme primitives, and includes (line by line): cons-cell manipulation, type predicates, arithmetic, flow-control, and I/O.

For a description of the Scheme execution semantics, we refer the reader to [5].

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

   (define-handler (gossip a b c)
(b)  ; react to the gossip message
     ; variable src is bound to ID of sender
     ...)
```

Figure 2: Example code snippets showing the use of timer and communication events

C Task scheduling

WSN nodes have an inherently reactive or event-based nature. This is 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, each starting only after the previous one has ended[1].

SensorScheme is designed to run on event-based WSN operating systems like TinyOS [7] or Contiki [4]. SensorScheme defines its own scheduling mechanism on top of the OS. When an event occurs, a SensorScheme task is scheduled. These tasks are handled in FIFO order. The kinds of events that can occur in SensorScheme are 1) firing a timer, 2) reception of a network message 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 2(a) the `time-loop` function repeatedly schedules itself at 5 second intervals to broadcast a message.

D Communication

SensorScheme messages consist 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

[1] With the exception for interrupt handlers or other high-priority tasks, which can interrupt running tasks.

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 2 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 2 (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 the receiving node *evaluates* the code using `eval` which results in installing (through calls to `define`) and executing the code.

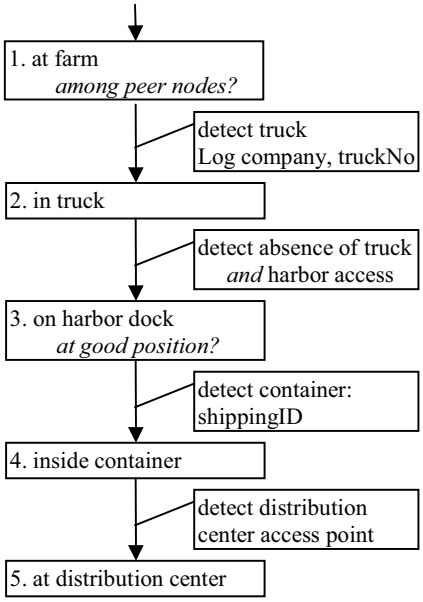

Figure 3: State diagram of the transportation process

III Example Application

The technology of Wireless Sensor Networks can provide great benefit to the supply chain management industry, using WSN nodes attached to returnable transport items, such as crates, rolling containers, pallets and shipping containers. In this example application we identify a role of sensor networks for logistics processes, and describe how (parts of) this role might be fulfilled with the use of SensorScheme.

A Scenario

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 sensor node. Early in the morning, these pallets travel in trucks from the farm 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.

Our bananas pass through a number of stages during their transport from farm to distribution center, as figure 3 shows. During the transportation process – which we'll call a *journey* – the device on each pallet checks for adherence to the transportation plan. The devices signal an alarm whenever the transport is not carried out correctly, or within the given time constraints.

At the farm, before loading, a pallet-attached device will verify whether it is positioned correctly: when it is near other pallets that are to be loaded into the same truck. It does this by comparing its destination and contents with peer nodes on other pallets nearby. Next, nodes are loaded into a truck, which they can detect by 'hearing' another device, placed inside the truck. While in the truck, pallet nodes have to detect being taken out of the truck, which can be concluded from absence of the truck, and presence of the wireless infrastructure (access point) of the harbor loading dock. If the right dock is not detected an alarm needs to be signaled.

After unloading on the dock, the pallets are loaded inside shipping containers and transported all the way to the distribution center. Again the devices verify whether they are positioned correctly to be reloaded into shipping containers before loading starts. After loading, the nodes can detect arrival at the distribution center again by detecting the distribution center's access points.

When errors are detected during the journey, the devices signal alarms. Different methods of raising the alert can be used, depending on the transport stage. For example, while pallets are outside the truck, waiting to be loaded, a beeping sound and blinking lights attract the attention of workers that can correct the problem. But when inside the truck, the alert should be notified to the driver in the truck cabin instead.

B Programming

The presented scenario is somewhat simplified, but one can imagine more complex transport routes and requirements. Possibly, WSN nodes on pallets might also make use of their sensors, for example to monitor the temperature for cooled transports, or detect shocks using acceleration sensors to monitor the risk of damage to fragile goods. Furthermore, with the support of local infrastructure, devices can be programmed to connect home through a internet connection to report their status.

To accommodate the most varied transport requirements we assume that all involved WSN devices are programmed upon loading (of the pallet) with a short program that executes the various monitoring and verification tasks. Enabling full programmability obviously opens some security risks. Usually, pallets are owned and managed by a pool organization. Only if users (transporters) will not be able to 'break' the devices (ie.

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

Figure 4: Example program source code

modify their software operation) can this scenario be a realistic one.

C Implementation

To illustrate the use of SensorScheme, we will now discuss an example implementation of a part of this scenario. The example shows how SensorScheme enables easy construction of communication protocols and blocking call creation, especially useful for communication-oriented WSN applications.

Recall that while pallets with bananas are at the farm waiting to be loaded into trucks, they will check with each other to verify correct placement. Pallets are placed correctly if their destination and content matches that of their peers. The SensorScheme code presented in figure 4 defines procedure `peer-dict`. This procedure takes a key and timeout value as parameters, and requests from each neighbor their value with that key in their own dictionary `global-dict`. The procedure returns after *timeout* seconds with the associated values of all its neighbors.

The SensorScheme code in figure 4 contains a number of procedure references defined in the Scheme standard [1] or one of the *srfi*'s [13], and we will use them without further mention of their operation.

SensorScheme provides continuations, that can be used to implement a light-weight concurrency mechanism. It allows an arbitrary number of simultaneous outstanding blocking I/O operations, without using more memory than strictly needed to contain application state. We will not discuss the semantics of continuations and the `call/cc` primitive here; for a thorough description of continuations we refer the reader to [6].

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 `waiting-reqs` dictionary (line 5-6). The `call/cc` invocation on line 7 creates a continuation, used to return to the function's caller after the timeout. At line 9 a timer is set up to signal the end of the timeout. Finally, a call to exit (line 13) aborts the current task, allowing other events to be processed while `peer-dict` is blocked.

	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 1: Code sizes of example program

The message broadcast at line 4 is handled by the `peer-dict-hdl` handler at all receiving nodes (lines 16-18). 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 22-25), it looks up the request ID in the `waiting-reqs` dictionary, and extends the value list with the value just received (line 25).

When after *timeout* seconds the timer expires (line 10-12), 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.

The absence of error checking code is intentional and illustrates one of the consequences of the use of SensorScheme. For example, in the `peer-dict-hdl` handler, if the requested key entry does not occur in the dictionary, assoc returns `#f` (false), and taking the *cdr* of `#f` results in an error, which immediately aborts the handler, without sending any message. This is the expected behavior and can be achieved without any explicit error detection or handling code.

IV Evaluation

SensorScheme is implemented on a WSN platform, and using the code example in figure 4 we now evaluate some performance aspects of this implementation. Our hardware platform is based on an MSP430 micro-controller, containing 10 KB of RAM, and a 50 Kbps radio transceiver. The global memory pool contains 9400 cells, which take up 4 bytes each.

A Code size

We first consider the size of the program code. Besides the code shown in figure 4, some standard library functions such as `assoc` are also made available on the nodes. Table 1 shows their sizes. Compared to the size in memory, the compact network encoding used reduces it to less than a quarter during transmission across the network. Besides this program, another 1975 cells are available for additional program code and for use during program execution, by the call stack, variables, scheduling and timer queues and application data.

		#	ms	mJ	%
Program execution	(# cycles)	1,245 K	208	1.27	6.9%
OS comm. TX	(# msgs)	14.4	41.4	0.25	1.3%
RX	(# msgs)	119	88.9	0.54	2.9%
Radio TX / RX	(air time)		455	16.45	89%
Total energy				18.52	

Table 2: Results of example program evaluation

B Runtime performance and energy use

We have measured the impact of evaluation overhead on total computation time and the energy used by execution of SensorScheme programs. For computation time measurements we used a processor emulator in a simulated network of 20 nodes, each periodically calling the `peer-dict` function of figure 4. All energy calculations are based on the data sheets of the hardware components of our implementation platform.

Table 2 presents some results of the running time per invocation of the `peer-dict` function. For each period, the SensorScheme code takes only 208 ms to execute. Using a period duration of 10 seconds, this would be just two percent of CPU time spent. Next, the energy spent on additional computation by the operating system and network stack. These numbers are based on estimations, since it was not possible to measure this using the simulated network. The last line shows the air time and energy used for transmitting and receiving the radio packets.

The last column shows the relative energy cost of computation and communication. It shows that most energy is used by the radio during communication (89 %), while program interpretation takes just 6.9 % 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 would 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 less than 7 percent.

V Conclusion

The ability of loading and updating applications after deployment is an important requirement to make wireless sensor networks practically usable. This paper presents SensorScheme as a platform for dynamically reprogramming wireless sensor networks. By design, SensorScheme is focused on minimizing the cost of wireless transport of program code to nodes in the network, and optimally using the scarce working memory available to WSN nodes. Based on the semantics of the programming language Scheme, it brings the benefits of high level languages, like garbage collection, concurrency, and automatic encoding and decoding of messages, together resulting in even smaller program sizes.

As a result of its interpreted program execution, SensorScheme also provides security against faulty or intentionally harmful programs, an important requirement for WSN applications in logistical processes, as our example application scenario shows.

The paper discusses an example application, showing that SensorScheme programs are extremely compact. Using the application example, we have evaluated the SensorScheme implementation and shown that the interpreted nature of SensorScheme programs has only marginal effect on total energy use of a node.

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. A. Iv, D. P. Friedman, E. Kohlbecker, J. G. L. Steele, D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised report on the algorithmic language scheme. *Higher Order Symbol. Comput.*, 11(1):7–105, 1998.

[2] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 187–200, New York, NY, USA, 2003. ACM Press.

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

[4] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[5] R. K. Dybvig. *The Scheme Programming Language*. The MIT Press, 2003.

[6] D. Ferguson and D. Deugo. Call with current continuation patterns. In *8th Conference on Pattern Languages of Programs*, Sept. 2001.

[7] 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*, pages 93–104, 2000.

[8] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

[9] P. Levis, D. Gay, and D. Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical Report CSD-04-1343, UC Berkeley, Aug 2004.

[10] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[11] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM Press.

[12] J. Polastre, R. Szewczyk, and D. E. Culler. Telos: enabling ultra-low power wireless research. In *IPSN*, pages 364–369, 2005.

[13] SRFI. Scheme requests for implementation. `http://srfi.schemers.org/`.

[14] Sun SPOTs. `http://www.sunspotworld.com/`.