Flexible Sensor Network Reprogramming for Logistics

Leon Evers Paul Havinga Jan Kuper

July 30, 2007

Abstract

Besides the currently realized applications, Wireless Sensor Networks can be put to use in logistics processes. However, doing so requires a level of flexibility and safety not provided by the current WSN software platforms. This paper discusses a logistics scenario, and presents SensorScheme, a runtime environment used to realize this scenario, based on semantics of the Scheme programming language. SensorScheme is a general purpose WSN platform, providing dynamic reprogramming, memory safety (sandboxing), blocking I/O, marshalled communication, compact code transport. It improves on the state of the art by making better use of the little available memory, thereby providing greater capability in terms of program size and complexity. We illustrate the use of our platform with some application examples, and provide experimental results to show its compactness, speed of operation and energy efficiency.

1 Introduction

Recent technological advances in low power digital RF, small scale sensors and low power silicon integrated circuits have created a new domain of computing: wireless sensor networks (WSNs) that sense their environment, and collectively compute and reason upon the perceived state of the world around them. Already, WSNs have found applications in the field of environmental monitoring [1], [2], and object tracking [3].

In the foreseeable future wireless sensor networks can also make a great impact in the supply chain management business. WSN nodes can be attached to crates, roll containers, pallets, and shipping containers to function as *Active Transport Tracking Devices* as we call them. These devices can be programmed to actively monitor the transportation process, and verify proper handling conditions of goods like temperature for fresh foods. Furthermore, they can detect damage due to sudden shocks, or opening of containers and other forms of contract breach. Actively monitoring every transported item in this way can significantly reduce delivery delays and loss or theft of goods, which cause a significant loss of revenue. Furthermore, these devices can create a complete overview of the entire logistical process, and improve efficiency and quality of service resulting in reduced safety stocks and improved handling efficiency.

Active transport tracking devices require 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.

In a nutshell, SensorScheme provides the following features:

- Interpreter based on the Scheme language;
- Communication by automatic marshalling of data items;
- Safe execution environment, in which malfunctioning programs cannot crash nodes;
- Co-routines implement multiple threads of control, enabling blocking I/O calls;

- No limits on code size, application complexity and memory use (except for available memory);
- Garbage collected memory pool, shared between interpreter state and application data.

Besides tracking logistical processes, SensorScheme is also beneficial to many other, more ‘traditional’ WSN applications. SensorScheme bears many similarities with the Maté [4] 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.

The rest of the paper is organized as follows: Section 2 gives a more thorough description of the logistics application scenario, followed by a review of the state of the art for realizing this application in section 3. Next, section 4 describes the design of SensorScheme, followed by a discussion of implementation techniques for the scenario in section 5. Then we evaluate SensorScheme’s performance in section 6, and conclude and give future directions (section 7).

2 Scenario

In the introduction we have already outlined how the use of Mote-sized [5] active transport tracking devices (ATTDs) attached to returnable transport items (RTIs), such as crates, rolling containers, pallets and shipping containers can be used to improve the effectiveness of logistical processes.

To illustrate how ATTDs might be programmed we will now discuss a small transportation 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 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. During the whole trip, the bananas need to be kept cool, between 10 and 15 degrees Celsius, and away from sources of ethylene gas, such as fresh coffee beans, that adversely influence the ripening process.

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, which we’ll call a *journey*.

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’ another device, placed inside the truck, at which point they’ll make the transition to stage 2. This device in the truck is programmed with its own itinerary, containing data about its identity, as well as the goods it is to be transporting. When in the truck, each pallet device requests from the truck device the company and truck IDs and records these into the log file.

While in the truck, pallet nodes do not have to verify anything, since no change in state will take place until they are taken out. They do have to detect being taken out of the truck, however, which can be 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 ATTDs again verify whether they are positioned correctly to be reloaded into shipping containers. The dock is equipped with advanced electronic infrastructure capable of tracking each pallet’s location, and based on this, each pallet verifies whether it is at the correct position. 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 ATTDs sense the distribution center access point and make the state transition.

At the banana farm, each ATTD is programmed with a small executable program, called an *itinerary* that in effect tracks the bananas as they move through the logistics process. The itinerary program is listed in pseudo-code in figure 2. The program consists of three task definitions that will be run in parallel. The first task in effect executes the states and transitions of the state diagram of figure 1. It has a similar structure: alternating the actions of each state and checking for transition conditions.

Before going into the details of the individual statements, it is important to note that the itinerary makes use of a global dictionary, that can be stored into and read from using resp. the `appendDict()` and `dict()` procedures. The dictionary stores data items tagged with a key that are relevant to the transportation process. As data is added to it, a transcript is made to the device's log, providing a complete record of all relevant actions and events on the current journey. Furthermore, this dictionary is also used to store information relevant to other nodes, and can thus be queried remotely, as we will see in section 5. The itinerary starts by adding general data specific to this journey to the dictionary and log. (line 2). The lowercase names are dictionary keys, and the `xxxID` names the values, to be replaced by actual ID numbers.

Upon entry of each state the itinerary task assigns an alerter function to global variable *alerter* (lines 4, 9, 16, 21). When any of the tasks detect an illegal situation, the alerter function is called to report the error in a way most appropriate to the current stage in the journey (either `BlinkAndBeepAlert`, `TruckAlert`, `AccessPointAlert` or `ContainerAlert`).

While still at the farm, (lines 5-7), as long as no truck is found within wireless connection range, the device keeps verifying matching destination and contents with (the majority of) peer nodes on other pallets nearby. If the device has unmatching content or destination it will raise an alert (currently `BlinkAndBeepAlert`). Further on, in section 5 we will more closely look at how this peer communication will take place.

Once in the truck, pallet ATTDs update their dictionary (thereby also recording the state change), and add `truckCompany` and `truckNo` data queried from the truck device's dictionary (lines 9-11).

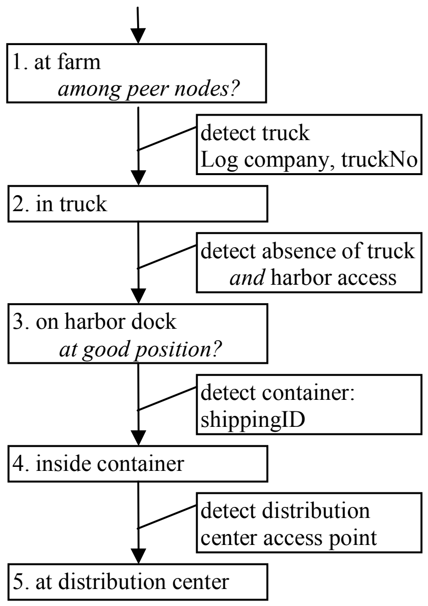

Figure 1: State diagram of the transportation process

The next state change occurs when the truck is out of range and the expected access point is found in range (lines 12, 13). Unlike trucks, access points can be out of direct radio connectivity, only connected through multiple hops, which is why a simple `inRangeNodes` call cannot be used to detect it and `AccessPointInRange` is used in stead.

The next two stages are similar in structure to the first two, but use different verification and state change detection code. At the harbor dock pallets communicate with the access point to verify their position by calling `PositionVerify (ShippingID)` instead of peer communication as used while at the farm. The Rotterdam distribution center access point uses proprietary wireless protocols, and can be detected using `ProprietaryAccessPointInRange`.

The rest of the itinerary program contains two tasks, `Logger` and `Guarder` that are occupied with a single task for the duration of the trip. The `Logger` task (lines 28-30) calls `timedLog` to monitor ambient temperature at 1 minute intervals, logs the sensed

```
task Itinerary () {
  appendDict ({ loc : MyFarmID, dest : RioHarborID,
                contents : BananasID });
  // at farm
  alerter = BlinkAndBeepAlert;
  while not any(deviceType(inRangeNodes()) == TruckType) {
    PeerVerify ({dest : dict(dest),
                 contents : dict(contents)})
  };
  alerter = TruckAlert;
  appendDict ({truckCompany, truckNo :
               TruckRequestProps([company, serial]) });
  // in truck
  until (not any (deviceType(inRangeNodes()) == TruckType)
         and AccessPointInRange(RioAccessPointID)) {
    wait;
  };
  // on harbor dock
  alerter = AccessPointAlert;
  appendDict ({ loc : RioHarborID });
  while (not any (peerDict(inRangeNodes(),
                           {shipper = ShippingID}) {
    PositionVerify (ShippingID);
  };

  // inside container
  alerter = ContainerAlert;
  appendDict ({ dest = RotterdamDistCentID });
  until (
      ProprietaryAccessPointInRange(RotterdamDistCentID)) {
    wait;
  };
  // at distribution center
  appendDict ({ loc = RotterdamDistCentID});
};

task Logger () {
  timedLog(1 minute, (senseAndAlert (TempSensor(), 10 C, 15 C))
};

task Guarder () {
  noProximity({contents : Coffee}, 10M);
};
```

Figure 2: Pseudo-code of itinerary program

values, and alerts if the temperature is outside of the allowed range, using the alerter function currently assigned to the alert variable by the itinerary task.

The `Guarder` task calls the `noProximity` function that continuously verifies whether the device can find any nearby pallets (within approx. 10 meters) that contain coffee. If found, it writes a log entry and signals an alert (again using the current alerter).

Note that the presented itinerary is somewhat simplified, omitting corner cases and handling of error conditions. Here it serves mainly as clarification of our proposed solution. We also have not mentioned how one would obtain an itinerary program. In simple cases these might be programmed by hand, but we consider it more likely to be generated automatically from a logistics management application.

The itinerary program calls a substantial number of procedures. In section 5 we will more closely discuss an implementation for one of those, `PeerVerify`. Some of these procedures, like `AccessPointInRange` are standard, and are included on the ATTDs as part of a standard library. Others, like `ProprietaryAccessPointInRange`, are specific to a particular journey, and will be programmed into the device along with the itinerary. The possibility of supplying custom procedures along with more standardized code creates a very flexible system. Since an itinerary is expressed as an application program, every time it is replaced, this effectively results in reprogramming (a part of) the wireless device's program code, while it is deployed and in operation.

Safety must also be considered. Usually, pallets are owned and managed by a pool organization. Only if users (transporters) will not be able to 'break' the devices (ie. modify their software operation) can this scenario be a realistic one. Now all that's needed is a way to express these itinerary programs that is expressive, compact, and safe.

3 Application requirements and state of the art

Dynamically replacing or updating applications is a crucial technology for the logistics scenario. Several code update mechanisms have been developed already for Wireless sensor network platforms. These work by replacing the entire program image as a whole. The TinyOS platform [6] for sensor networks includes XNP [7] and Deluge [8] as two of such technologies. Unfortunately, this approach is not suitable for our scenario for a number of reasons.

First, program images typically are a few tens of kilobytes in size, and transporting this much data takes time in the order of minutes (according to [8], [9]). This can be improved somewhat by using one of various compression and differential algorithms (RSYNC [10], MOAP [11], FlexCup [9]).

Second, these code update mechanisms aim at replacing the entire binary with a new one, including the operating system that controls task scheduling, low level hardware access, network protocols, and even the code update mechanism itself, which should not be modifiable by the ATTD users. Several WSN platforms provide runtime loadable modules (Contiki [12], SOS [13]), but these still give unrestricted access to the entire device.

A more suitable approach is the use of an interpreter or virtual machine. On the one hand, only the application code needs to be transported to the devices, which significantly reduces the size of transported code. Moreover, since code representation is a design variable, rather than a hardware characteristic, it can be engineered specifically to be compact for the kinds of programs expected to be built for it.

Additionally, the interpreter or virtual machine acts as what is usually called a ‘sand box’, shielding off the operating system from the interpreted applications. Misbehaving or buggy applications are thus prevented from modifying any state besides their own, and cannot crash a device or damage the device’s critical functions.

The most well-known and frequently used virtual machine architecture in mainstream computing is Sun’s Java Virtual Machine. It has found use as a sensor network platform already (Sun SPOTs [14]). SensorWare [15] is another platform based on interpretation and sandboxing for WSN’s. However, both of these require more resource-rich platforms than currently developed WSN’s that we consider for our application scenario.

Maté / Bombilla [4] is a virtual machine designed specifically for memory-constrained WSN devices. Unfortunately, Maté can contain only truly tiny applications. Programs are organized in *contexts* associated to event sources, containing a 128 byte instruction array that is run in response to triggered events. Each context (6 in total) has its own operand stack, set of 8 local (integer) variables and a packet buffer. These severe complexity restrictions 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. Furthermore, Matéś concurrency model does not support multiple independent tasks running in parallel. More recently, other WSN-specific VM architectures have been developed (VM* [16], DaViM [17]) that somewhat alleviate some of these issues, without solving them.

4 SensorScheme

We propose SensorScheme as a novel interpreted platform for WSN’s that can be used to implement our application scenario. Our platform uses execution semantics of the programming language Scheme, hence its name. It is, however, not an implementation of the Scheme language, and creating SensorScheme programs does not require the use of the Scheme language or syntax. For clarity, the code examples in the following sections do use Scheme syntax.

4.1 Program representation and execution semantics

The heart of the SensorScheme platform is the approach it takes on representing program code, and the execution semantics. This program representation is not a novel concept introduced by SensorScheme, but since it might be considered a somewhat unusual approach (at least in the WSN community), we will briefly describe the basics of it here. For a more indepth description of the Scheme execution semantics, we refer the reader to [18].

Program fragments take the shape of a specially formatted linked list of memory cells, called s-

```
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 3: A grammar for SensorScheme

expressions. Figure 3 summarizes the SensorScheme s-expression grammar (using Scheme syntax). 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 includes most of the common Scheme primitives, and includes (line by line): cons-cell manipulation, type predicates, arithmetic, flow-control, and I/O.

Execution of SensorScheme programs proceeds through tree-traversal of the s-expression. Operands to primitives or procedures are s-expressions themselves, which are evaluated first, before application. Scheme-defined functions (paired with local variable bindings, called *closures*, obtained as the result of lambda-abstraction) are first class objects, assignable to variables, just like any other value. Defining functions effectively amounts to defining a global variable that carries as its value a closure.

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.

4.2 Memory

The biggest limitation of WSN platforms is the very limited working memory (RAM). This is even stronger for interpreted programs, since generally, they require more memory than their compiled counterparts.

It is general practice in WSN software to store all program state in statically allocated memory and on the stack. No dynamic memory allocation from a runtime heap is used, because allocating all but the smallest blocks of memory from a small heap very quickly leads to internal fragmentation which will fail subsequent memory allocation attempts. This is clearly an undesirable situation for long-lived WSN applications.

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. Each structure uses only as much memory as is needed at the moment, and there is no inherent limit to the maximum size any structure, except the total available memory. Compared to VM architectures like Maté this allows a wider variety of applications to be executed.

4.3 Symbols

SensorScheme distinguishes only 3 data types, of which symbols are one. As is common in any member of the Lisp family of languages[1], symbols are a vital part of the SensorScheme semantic model, and are used for tagging s-expressions (in the case of special forms) and variable reference. Furthermore, symbols are a useful data type in their own right, as immutable, unique identifiers.

In SensorScheme symbols only have a numeric representation, with no actual 'name'. This allows for efficient communication and storage in memory. Furthermore, no new symbols can be created by programs, only received through communication with other devices.

In source code and in its presentation outside the network of SensorScheme devices symbols typically have names, so a network manager, connecting the SensorScheme devices to the outside world must maintain association between the numeric and textual representations of symbols. We consider the details of this network management, to be outside the scope of this paper, and will not discuss it any further.

The availability of symbols as unique identifiers is also of importance to the distributed nature of sensor networks. Symbols remain semantically meaningful across communication, and can be used as identifiers of data (or code) on remote nodes: symbols are used as protocol tags, identifying program code to be executed on arrival of the message. When used as dictionary keys, symbols can be used to refer to data stored on remote nodes, as the example in section 5 shows.

[1]in fact the focus of SensorScheme is as an execution platform instead of a source language, but – as is the case for any Lisp-like language – the two are so interrelated that it does not usually make sense to separate them. For SensorScheme, however, the program source and program execution reside on different devices, which makes the separation more meaningful.

4.4 Event-based scheduling model

WSN nodes have an inherently reactive or event-based nature. This is reflected in today's WSN operating systems like TinyOS [6] or Contiki [12]. 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.[2] 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 (implemented as a *thunk*, a parameterless procedure) is scheduled. These tasks are handled in FIFO order. The kinds of events that can occur in SensorScheme are 1) reception of a network message and 2) firing a timer, and 3) hardware events originating from sensors.

We will defer discussion of communication events to section 4.5 and continue with timer event handling now. 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 4(a) the `time-loop` function repeatedly schedules itself at 5 second intervals to broadcast a message.

4.5 Communication

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

All SensorScheme data is contained in memory cells of a small set of data types, tagged with a type

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

```
      (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 just received
        ; variable src is bound to ID of sender
        ...)
```

Figure 4: Example code snippets showing the use of timer and communication events

code. Using this *runtime type information* devices transform a data structure into a linear representation suitable for network communication. Upon reception the receiver can recreate (a copy of) the same data structure from the linear representation. This is a familiar technique known as marshalling, also used in other technologies like CORBA or Java RMI.

SensorScheme communication operates similar to TinyOS's *Active Message* paradigm. A message consists of a header symbol and a number of data items. The message header is a symbol that 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 4 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 4 (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 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)))))
```

Note that the SensorScheme communication interface poses no restrictions on the number of data items, or the size of each data item in a message. Hence, 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.

5 Discussion

We will now discuss an example implementation for one of the helper procedures referenced in the pseudocode in figure 2, and show by example how SensorScheme can serve as an implementation platform for those procedures. The example shows how SensorScheme enables easy construction of communication protocols and blocking call creation, especially useful for communication-oriented WSN applications.

The SensorScheme code presented in figure 5 contains a number of procedure references defined in the Scheme standard [19] or one of the *srfi*'s [20], and we will use them without further mention of their operation.

```
 1 ; definition of PeerVerify from figure 2
 2 ; (invoked like this:
 3 ;     PeerVerify ({dest = dict(dest), contents = dict(contents)})
 4 (define (peer-verify alist)
 5   (if (not (every (lambda (kv)
 6                     (every (lambda (v)
 7                              (eq? v (cdr kv)))
 8                            (peer-dict 5 (car kv)))) alist))
 9       (alerter 'itinerary-error 'peer-verify)))
10
11 ; requests the value of given keys from all neighbors
12 (define (peer-dict timeout key)
13   (let ((reqid (rand)))
14     (bcast (list 'peer-dict-hdl reqid key))
15     (set! waiting-reqs (cons (cons reqid ()) waiting-reqs))
16     (call/cc (lambda (k)
17                (call-at-time (+ (now) timeout)
18                              (lambda ()
19                                (k (cdr (assoc-and-remove!
20                                         reqid waiting-reqs))))))
21                (exit)))))
22
23 ; handler invoked at neighbors
24 (define-handler (peer-dict-hdl reqid key)
25   (bcast (list 'peer-dict-rpl src reqid
26                (cdr (assoc key global-dict)))))
27
28 ; handler receiving values from neighbors
29 ; called at requesting node
30 (define-handler (peer-dict-rpl dst reqid val)
31   (when (= dst id)
32     (let ((req (assoc reqid waiting-reqs)))
33       (set-cdr! req (cons val (cdr req))))))
```

Figure 5: peer-dict source code

Figure 5 shows a SensorScheme implementation of the `PeerVerify` procedure (now called `peer-verify` to match Scheme naming conventions) referred to in the pseudocode of figure 2. The procedure accepts an association list[3] of key-value pairs, and communicates with all direct neighbors to find their dictionary entries of given keys. If any of the neighbors' values are different from the given parameters, the current alerter function is called (see lines 4-8).

Most of the actual work is done in procedure `peer-dict` (lines 12-21). This is a blocking call that takes a key and timeout value as parameters, and returns after *timeout* seconds with the associated values of all its neighbors.

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 [21].

Function `peer-dict` sends a request to all neighbors (line 14) containing a unique request ID (created at line 13) and the requested key, and stores the request ID in the `waiting-reqs` dictionary (line 15). The `call/cc` invocation on line 16 creates a continuation, used to return to the function's caller after the timeout. At line 17 a timer is set up to signal the end of the timeout. Finally, a call to exit (line 21) aborts the current task, allowing other events to be processed while `peer-dict` is blocked.

The message broadcast at line 14 is handled by the `peer-dict-hdl` handler at all receiving nodes (lines 24-26). 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 30-33), it looks up the request ID in the `waiting-reqs` dictionary, and extends the value list with the value just received (line 33).

When after *timeout* seconds the timer expires (line 18-20), 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.

[3]An association list is a list of pairs or cons-cells each containing the key in the *car* and the value in the *cdr* of the cell.

SensorScheme runtime	7750	bytes Flash
– garbage collector	294	bytes
– cell allocator	122	bytes
– (un)marshaller	1640	bytes
– primitives	3728	bytes

MSP430 memory	10240	bytes RAM
OS and buffers	830	bytes
runtime state	10	bytes
memory pool (2350 cells)	9400	bytes

Table 1: Memory use of SensorScheme implementation

6 Evaluation

We have implemented SensorScheme on a sensor network hardware platform and used it to measure execution speed performance. Next, we use the example application of figure 5 to analyze memory use, interpretation overhead and and energy cost.

6.1 Implementation

SensorScheme is implemented on a wireless sensor network platform based on an MSP430 microcontroller, containing 10 KB of RAM and 48 KB program flash. The device contains a Nordic nRF905 RF transceiver chip, communicating at 50 Kbps.

Cells take up 4 bytes each, and are aligned at 4 byte addresses. The total address space of cells in RAM is addressed using only 13 bits. SensorScheme values are expressed in 15 bits, with the low 2 bits available as type tags for the four possible SensorScheme data types: symbols, short numbers, long numbers and cons cells. The other 2 bits per cons cell are used for memory allocation and garbage collection, which is a simple mark and sweep algorithm.

Table 1 shows the memory use details of the implementation. The SensorScheme runtime environment, including the primitive procedure implementations, is very small, using only 7.7 KB of program memory. Most of the 10 KB of RAM is available for the shared pool; the rest is allocated by the OS and network buffers.

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 2: Code sizes of example program

	cycles	C cyc.	slowdown
1 loop	2557	25	102
2 `(+ n n)`	837	8	105
3 `(* n n)`	944	64	14.8
4 `(rand)`	455	123	3.70
5 `(list ...)`	2377	204	11.8
6 `(func0)`	370	13	28.5
7 `(func3 1 2 3)`	1072	26	41.2
8 `(bcast ...)`	3565	267	13.4
9 `(bcast ...)` + OS	23565	20267	1.16

Table 3: Results of interpretation overhead measurements

6.2 Code size and memory use

Before we will discuss the performed evaluations, we first consider the size of the program code, shown in table 2. To enable running the program presented in figure 5, some standard library functions are also made available on the nodes, like `every` and `assoc`. Table 2 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.

6.3 Interpretation overhead

The interpreted nature of SensorScheme imposes an execution overhead in comparison to native code. To quantify this overhead we have measured the computation time of a number of simple test cases in SensorScheme, by repeatedly executing them in a tight loop, and compared it to native execution speed of the same operations programmed in C. All tests were performed using a processor emulator that accurately counts the clock cycles per instruction. Table 3 shows the results of these measurements.

The first test case measures the running time of the loop itself, representing a simple case of flow control. The next two cases perform simple (addition) and more complex (multiplication) arithmetic operations. Subsequently native procedure call, dynamic memory allocation, and function call without and with parameters.

The last two test cases measure the cost of communication. We have not been able to accurately measure the instruction cost of sending a message once it has been handed over to the OS. Instead we have measured the cost without OS overhead (case 8), and added an estimated 20000 instruction cycles spent in the OS and MAC layer (case 9). Including overhead, communication is only 16 percent slower using SensorScheme compared to native code. Applications typically used in sensor networks will contain a large fraction of communication, making the total interpretation overhead relatively low.

The other tests show similar effects. More complex operations that execute long sequences of native code, like native calls and complex arithmetic operations (cases 3 and 4) impose less overhead than simple arithmetic operations and control flow (cases 1 and 2). Furthermore the results suggest that SensorScheme's uniform memory lay-out results in comparatively cheap memory allocation (case 5) and that function application is a relatively inexpensive operation compared to other flow control (cases 6 and 7).

6.4 Runtime performance and energy use

Our second test measures memory use and the impact of evaluation overhead and garbage collection on total computation time, which are in short supply on WSN platforms. Energy use is a crucial performance factor as well, so we have measured the energy used by execution of SensorScheme programs.

(a)

	cycles	ms	mJ
Execution time and energy	1245483	208	1.27
Fraction spent in allocation	25.2%		
Fraction spent in GC	31.4%		
– # collections	6.43		
– execution time / collection	10.1	ms	
– avg. used cells	395	cells	
– max. used cells	429	cells	

(b)

Comm. energy	TX	RX	total	
`peer-dict-hdl`	2	12.6		msgs
`peer-dict-rpl`	12.4	107		msgs
OS time	41.4	88.9	130	ms
OS energy	0.25	0.54	0.80	mJ
message size	160	153		bits / msg
air time	89	365	455	ms
radio energy	2.41	14.04	16.45	mJ

(c)

Total energy used			
program execution	1.27	mJ	7%
OS communication	0.80	mJ	4%
radio TX / RX	16.45	mJ	89%
Total	18.52	mJ	

Table 4: Execution statistics

Our tests are performed using our emulator in a simulated network of 20 nodes, each periodically running the `peer-verify` function of figure 5. This represents a real-world situation, since only one itinerary verification would be taking place at any given time.[4] All energy calculations are based on the data sheets of the hardware components of our implementation platform.

Table 4 (a) lists some results of the running time per invocation of the `peer-verify` 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.

A large fraction (about 57%) of execution time is spent on memory allocation and garbage collection. This is a logical consequence of the design of SensorScheme; program data as well as stack frames are allocated from the memory pool. This quickly consumes all memory, after which a garbage collection cycle is necessary. Garbage collection itself causes application pauses of only 10 ms, an acceptable delay for most WSN applications.

At the end of every garbage collection the average number of cells used is 395, and the maximum is 429. Considering the 375 cells used to store the program, between 20 and 54 cells are taken for program data and runtime structures. Altogether, less than one fifth of the total memory is needed by this application, leaving ample space for other larger or more complex applications.

Communication takes a significant fraction of the total energy use on WSN nodes. Table 4 (b) shows the number of messages sent and received per period, and the energy spent on additional computation by the OS, based on estimations, 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, table 4 (c) shows the relative cost of each of those. It 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.

[4]Other verification tasks might also be active, each taking roughly similar execution time.

7 Conclusion and future directions

We discussed a logistics application example that requires a safe execution environment to host possibly insecure applications, which are changed frequently, 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, unhindered by the fixed, arbitrary size of any internal structure. The use of Scheme semantics brings additional benefits, like automatic memory management, concurrency, and automatic encoding and decoding of messages, together resulting in even smaller program sizes. The SensorScheme implementation is small and efficient, and shows that advanced features like continuous program update, a safe execution environment, garbage collection, and concurrency and blocking I/O are achievable targets even for mote-size WSN platforms. SensorScheme causes only marginal additional energy use and no significant delays due to program interpretation and garbage collection.

Still, research challenges remain. Dynamic memory allocation causes a degree of unpredictability, possibly causing nodes to fail at arbitrary moments when no more free memory remains. Methods to alleviate this issue will greatly increase usability of the platform. Furthermore, our application scenario relies on additional security, like tamper-proof operation and secrecy of itinerary and log data that should also find their way into the SensorScheme platform. The current implementation also leaves ample room for improvement, by storing (semi-) constant data, like program code, into the device's flash memory, or reducing the cell allocation rate to speed up computation.

References

[1] Werner-Allen, G., Welsh, M., Johnson, J., and-Jonathan Lees, M.R.: Monitoring volcanic eruptions with a wireless sensor network. Technical Report 27-04, Harvard University (2004)

[2] Mainwaring, A., Culler, D., Polastre, J., Anderson, R.S.J.: Wireless sensor networks for habitat monitoring. In: Proceedings of the 1st ACM international workshop on Wireless sensor networksand applications, ACM Press (2002) 88–97

[3] The Ohio State University NEST team: A Line in the Sand: A DARPA-NEST Field Experiment. `http://www.cse.ohio-state.edu/siefast/nest/nest\_webpage/ALineInTheSand.html` (2003)

[4] Levis, P., Gay, D., Culler, D.: Bridging the gap: Programming sensor networks with application specificvirtual machines. Technical Report CSD-04-1343, UC Berkeley (2004)

[5] Hill, J.L., Culler, D.E.: Mica: A wireless platform for deeply embedded networks. IEEE Micro **22**(6) (2002) 12–24

[6] Hill, J., Szewczyk, R., Woo, A., Hollar, S., E.Culler, D., Pister, K.S.J.: System architecture directions for networked sensors. In: Architectural Support for Programming Languages and Operating Systems. (2000) 93–104

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

[8] Hui, J.W., Culler, D.: 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) 81–94

[9] Marrón, P.J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., Rothermel, K.: Flexcup: A flexible and efficient code update mechanism for sensor networks. In: Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN2006). (2006) 212–227

[10] Jeong, J., Culler, D.: Incremental network programming for wireless sensors. In: First IEEE Comm. Soc. Conf. on Sensor and Ad Hoc Communications and Networks. (2004)

[11] Stathopoulos, T., Heidemann, J., Estrin, D.: A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing (2003)

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

[13] Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: MobiSys '05: Proceedings of the 3rd international conference on Mobile systems,applications, and services, New York, NY, USA, ACM Press (2005) 163–176

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

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

[16] Koshy, J., Pandey, R.: Vmstar: synthesizing scalable runtime environments for sensor networks. In: SenSys '05: Proceedings of the 3rd international conference on Embeddednetworked sensor systems, New York, NY, USA, ACM Press (2005) 243–254

[17] Michiels, S., Horr&#233;, W., Joosen, W., Verbaeten, P.: Davim: a dynamically adaptable virtual machine for sensor networks. In: MidSens '06: Proceedings of the international workshop on Middleware forsensor networks, New York, NY, USA, ACM Press (2006) 7–12

[18] Dybvig, R.K.: The Scheme Programming Language. The MIT Press (2003)

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

[20] SRFI: Scheme requests for implementation. (`http://srfi.schemers.org/`)

[21] Ferguson, D., Deugo, D.: Call with current continuation patterns. In: 8th Conference on Pattern Languages of Programs. (2001)