SensorScheme: Supply Chain Management Automation using Wireless Sensor Networks

L. Evers P.J.M. Havinga J. Kuper M.E.M. Lijding
N. Meratnia
University of Twente
Drienerlolaan 5
7522 NB Enschede, the Netherlands
{l.evers, p.j.m.havinga, j.kuper, m.e.m.lijding, n.meratnia}@utwente.nl

Abstract

The supply chain management business can benefit greatly from automation, as recent developments with RFID technology shows. The use of Wireless Sensor Network technology promises to bring the next leap in efficiency and quality of service. However, current WSN system software does not yet provide the required functionality, flexibility and safety. This paper discusses a scenario showing how WSN technology can benefit supply chain management, and presents SensorScheme, a platform for realizing the scenario. SensorScheme is a general purpose WSN platform, providing a safe execution environment for dynamically loaded programs. It uses high level programming primitives like marshalled communication, automatic memory management, and multiprocessing facilities. SensorScheme makes efficient use of the little available memory present in WSN nodes, to allow larger and more complex programs than the state of the art. We present a SensorScheme implementation and provide experimental results to show its compactness, speed of operation and energy efficiency.

1. Introduction

Supply chain management is a complex business, involving many parties and high volumes of goods. Not surprisingly, manual handling of transported goods leads to human errors accounting for significant loss in revenue. Automation can improve supply chain visibility, efficiency and yield higher turnovers. The recent application of RFID technology is already making a great impact on the retail supply chain, according to a recent study ([13]).

Other recent developments in low power wireless communication has spawned a new technology: Wireless Sensor Networks, consisting of small computing devices equipped with tiny sensors and wireless communication capabilities. Different from RFID, these devices are battery operated, and can communicate with any other device nearby, sense their environment, and continuously reason upon the perceived state of the world around them.

Wireless sensor networks hold a great promise for 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 actively monitor the transportation process, and verify proper handling conditions of goods like temperature for fresh foods. Furthermore, these devices can detect damage due to sudden shocks, or opening of containers and other forms of contract breach. This results in significant quality of service improvements and greater efficiency which in turn lead to lower transport cost.

Although current wireless sensor network hardware platforms are suitable as Active Transport Tracking Devices, the state of the art in WSN system software is lacking the right set of features.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. It presents a safe execution environment, in which malfunctioning programs cannot crash the device, and is equipped with high-level programming facilities such as garbage collection, communication by automatic marshalling of data items, and co-routines to implement multiple threads of control and enable blocking I/O calls. Besides tracking logistical processes, SensorScheme can find good use in many other, more 'traditional' WSN applications.

The rest of the paper is organized as follows: Section 2 presents an 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).

1-4244-0826-1/07$20.00 © 2007 IEEE

2 Scenario

The technology of Wireless Sensor Networks can provide great benefit to the supply chain management industry, when used as active transport tracking devices (ATTDs) attached to returnable transport items (RTIs), such as crates, rolling containers, pallets and shipping containers. To illustrate the use of how ATTDs 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. Early in the morning, these pallets travel in trucks (owned by the Banana Transportation Company, or BTC) 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. 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.

During the transportation process from the farm to the distribution center – which we'll call a *journey* – a number of things are monitored:

1. Temperature sensors on the ATTDs measure the ambient temperature at 1 minute intervals, and store the measured temperature in the device's log file. If the temperature exceeds the allowed range, the device will signal an alarm, so measures can be taken immediately.

2. Each pallet's tracking device communicates with others around it to verify whether any of those is transporting coffee beans or other harmful products. When a pallet is loaded into a container, the device also requests whether the container can find other containers carrying harmful products within a certain distance (of say 10 meters). If coffee beans are found nearby, the tracking device stores this in its log file and signals an alarm.

3. During the entire journey, the ATTD on each pallet checks for adherence to the transportation plan. At every stage of the journey, it verifies whether it is loaded into the right truck, and unloaded at the correct warehouse, as is depicted in figure 1. Every transition into a new stage of the journey is logged, and the devices signal an alarm whenever the transport is not carried out correctly, or within the given time constraints.

Our bananas pass through a number of stages during their transport from farm to distribution center, as figure 1 shows. At every stage, a different method of verification will have to be carried out, depending on the local circumstances.

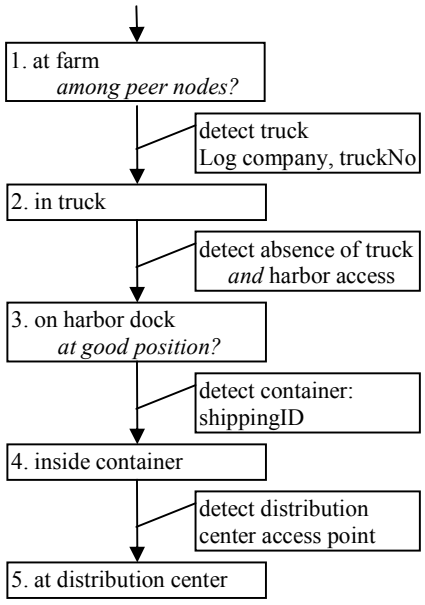

Figure 1. State diagram of the transportation process

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. When in the truck, each pallet device requests from the truck device the company and truck IDs and records these into the log file (along with the current time). Pallet devices are programmed with the information that they will be transported with any of the trucks of the BTC company before 6 am, and will signal an alarm if either condition is not met.

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. If the right dock is not detected, or it takes too long before the pallets arrive, an alarm needs to be signaled.

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. Unlike trucks, access points can be out of direct radio connectivity, only connected through multiple hops, and will need different, multi-hop communication protocols to communicate with.

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. Being the home base for this transport, the Rotterdam distribution center access point uses proprietary wireless protocols, to allow access only to the supermarket chain's owned goods. Again this requires different communication protocols to perform the journey's verification.

When errors are detected during the journey, the ATTDs signal alarms. Different methods of raising the alert are required, depending on the transport stage. 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. At the start of each new stage, the proper alert procedure is selected to be used by any of the sources of error that can occur in the device, including the temperature and coffee proximity processes.

2.1. Programming ATTDs

Now that we've outlined the functionality ATTDs should demonstrate, the question arises of how to program the devices to achieve this behavior. Simple implementations might involve a fixed program embedded in the devices, to be supplemented with a few parameters, like the content, destination, transport carrier details. These would be programmed in when the shipment is sent off. For a number of limited transportation scenarios this might suffice, but this will certainly not satisfy the worldwide supply chain management business in all its diversity. Maximum flexibility is achieved by maximum freedom, ie. to specify the entire journey verification program, and install it in each node before the start of the journey.

This flexibility opens up the possibility for many other verification methods for ATTDs. For example, opening of container doors can be detected from a sudden change in ambient temperature. Alternatively, using acceleration sensors, sudden shocks can be detected and logged, to find which transporter is responsible for damaged fragile goods like consumer electronics. Furthermore, with the support of local infrastructure, ATTDs can be programmed to connect home through a internet connection to report their status.

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

From the scenario described above we can distill a number of requirements a possible implementation must meet. We will discuss these and review to what extent the current state of the art has addressed these issues. As an implementation platform for active transport tracking devices we assume stock WSN hardware, such as the mica motes [9].

At the start of each journey, each ATTD needs to be reprogrammed for the upcoming journey, using the wireless interface. Several wireless 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 [8] for sensor networks includes XNP [3] and Deluge [10] 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 [10], [14]). This can be improved somewhat by using one of various compression and differential algorithms (RSYNC [11], MOAP [17], FlexCup [14]).

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 [4], SOS [7]), 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 hardware 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 [18]). SensorWare [2] is another platform based on interpretation and sandboxing for WSN's, using the TCL language. However, both of these require more resource-rich platforms than the ones we consider for our application scenario.

Maté / Bombilla [12] is a virtual machine designed specifically for memory-constrained WSN devices. Un-

fortunately, 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. An implementation of our scenario does, however, require memory allocation by the loaded program, and dynamic loading of procedures and custom communication protocols, as demonstrated in section 5. These severe complexity restrictions exclude our application scenario from being implemented on top of MatéFurthermore, Maté's concurrency model does not support multiple independent tasks running in parallel.

4. SensorScheme

We propose SensorScheme as a novel interpreted platform for WSN's that can be used to implement our application scenario. SensorScheme is designed for use on WSN hardware platforms, taking into account their resource restrictions. Most importantly, memory – especially RAM, is in short supply, and no mechanisms like virtual memory exist to extend memory use beyond what is physically available. Furthermore, wireless communication consumes the majority of energy compared to computation, and should be minimized as much as possible.

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. Instead, a more familiar C-inspired syntax will be available to users of the system. The code examples in the following sections do use Scheme syntax, as the 'assembly language' of the SensorScheme runtime engine.

4.1. 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.

4.2. 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 2 lists a grammar of the SensorScheme syntax tree (written in Scheme bracket notation). The operational semantics of these rules is as in regular Scheme.

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 2. A grammar for SensorScheme

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.

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

4.3. 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, starting after the previous one has ended.[1]

SensorScheme is designed to run on event-based WSN operating systems like TinyOS [8] 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 and 3) hardware events originating from sensors.

Timer events perform a computation scheduled at a predetermined moment in time. SensorScheme provides

[1] 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
     ; variable src is bound to ID of sender
     ...)
```

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

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.

4.4. 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 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 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 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

Using SensorScheme, ATTDs can be programmed to perform the tasks discussed in section 2. To illustrate this we now discuss a SensorScheme 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 contains a number of procedure references defined in the Scheme standard [1] or one of the *srfi*'s [16], and we will use them without further mention of their operation.

Figure 4 shows a SensorScheme implementation of procedure called `peer-verify`. The procedure accepts an association list[2] 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 2-7).

[2] 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.

```
 1 (define (peer-verify alist)
 2   (if (not (every (lambda (kv)
 3                     (every (lambda (v)
 4                              (eq? v (cdr kv)))
 5                            (peer-dict 5 (car kv))))
 6                   alist))
 7       (alerter 'itinerary-error 'peer-verify)))
 8
 9 ; requests the value of given keys from all neighbors
10 (define (peer-dict timeout key)
11   (let ((reqid (rand)))
12     (bcast (list 'peer-dict-hdl reqid key))
13     (set! waiting-reqs (cons (cons reqid ())
14                              waiting-reqs))
15     (call/cc
16      (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 4. Example program source code

Most of the actual work is done in procedure `peer-dict` (lines 10-21). 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 [6].

Function `peer-dict` sends a request to all neighbors (line 12) containing a unique request ID (created at line 11) and the requested key, and stores the request ID in the `waiting-reqs` dictionary (line 13-14). The `call/cc` invocation on line 15 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 12 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, no reply message should be sent. This can be achieved without any explicit error detection or handling code: Assoc returns `#f` (false), and taking the *cdr* of `#f` results in an error, which immediately aborts the handler, without sending any message.

6. Evaluation

We have implemented SensorScheme on a sensor network hardware platform and used it to analyze memory use, interpretation overhead and energy cost while running the example application of figure 4.

6.1. Implementation

We have built an initial SensorScheme implementation 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 transceiver chip, communicating at 50 Kbps.

The implementation has not received much attention towards speed optimizations, leaving significant room for improvement on the runtime statistics presented later in this section.

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 collector, similar to the original Deutsch-Schorr-Waite algorithm [15].

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.

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 4, some standard library functions are also made available

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

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

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.

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

6.3. Runtime performance and energy use

We have measured 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 will measure 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-verify` function of figure 4. This represents a real-world situation, since only one itinerary verification would be taking place at any given time.[3] All energy calculations are based on the data sheets of the hardware components of our implementation platform.

Table 3 (a) lists some results of the running time per invocation of the `peer-verify` function. For each such

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

(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 3. Time and energy use of example program evaluation

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 3 (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 3 (c) shows the relative cost of each of those. It shows that most energy is used by the radio power dur-

ing 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.

7. Conclusion and future directions

This paper identifies supply chain management as another application area of wireless sensor networks. We have outlined the possibilities of *Active Transport Tracking Devices* that can greatly enhance efficiency and quality of service for the transport and logistics industry. Furthermore, we have discussed a typical use case for ATTDs and analyzed how this might be implemented using current WSN technology. Current WSN system software has high resource requirements, or provides too little functionality to satisfy the application requirements, mainly due to memory-starved WSN platforms.

SensorScheme is our proposed platform for implementing active transport tracking functionality on stock WSN hardware. Based on the semantics of the programming language Scheme, it brings the benefits of high level languages, like garbage collection, concurrency, and automatic marshalling of messages, together resulting in even smaller program sizes. Still, the SensorScheme implementation is small and efficient. By making better use of the little available memory, SensorScheme is able to provide a wider range of functionality. SensorScheme causes only marginal additional energy use and no significant delays due to program interpretation and garbage collection.

Still, research challenges remain. Additional security, like tamper-proof operation and secrecy of log data should also find their way into the SensorScheme platform. Furthermore, 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.

Acknowledgement

This work was partly funded by the European 6th Framework Programme as part of the projects CoBIs (IST-2-004270) and e-SENSE (IST-4-027227).

References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. AdamsIv, D. P. Friedman, E. Kohlbecker, J. G. L. Steele, D. H.Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. B. andC. 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 programmablesensor 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 networkedsensors. 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] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems,applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[8] 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.

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

[10] 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*, pages 81–94. ACM Press, 2004.

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

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

[13] LogicaCMG. Making waves: Rfid adoption in returnable packaging, 2004.

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

[15] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

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

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

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

455