Available online at www.sciencedirect.com
ScienceDirect
Pervasive and Mobile Computing 3 (2007) 386–412
pervasive and mobile computing
www.elsevier.com/locate/pmc

SensorWare: Programming sensor networks beyond code update and querying

Athanassios Boulis[a,*], Chih-Chieh Han[b], Roy Shea[b], Mani B. Srivastava[b]

[a] *National ICT Australia, Networks and Pervasive Computing program, Australia*
[b] *Networked and Embedded Systems Laboratory (NESL), EE Department UCLA, United States*

Received 31 October 2006; received in revised form 7 April 2007; accepted 14 April 2007
Available online 5 May 2007

Abstract

Wireless ad hoc sensor networks have been largely designed with static and custom architectures for specific tasks, thus providing inflexible operation and interaction capabilities. Efforts to make sensor networks dynamically programmable stumble upon the problems of algorithmic expressiveness, compactness of transferred code, efficiency of executed code, and ease of programming. In short, the problem is the choice of abstraction for the sensor node run-time environment. Our framework, called SensorWare, defines and supports lightweight and mobile control scripts that allow the computation, communication, and sensing resources at the sensor nodes to be efficiently harnessed in an application-specific fashion, through the use of abstraction services. A key feature is that the run-time abstraction can change by dynamically defining new services. Furthermore, by making the scripts autonomously mobile we enable the deployment of the algorithm to be tied to its execution, a feature that reduces the code transferred, compared to conventional code deployment and update approaches. The implementation of SensorWare on an XScale-based prototype sensor node platform occupies less than 240 KB of code memory. The implementation is used to measure the delay and memory overheads, but more importantly, quantitatively highlight the trade-offs involved in run-time abstraction versatility.

Keywords: C.2.7.c Sensor networks; D.1.8 Distributed programming

* Corresponding author.
E-mail addresses: athanassios.boulis@nicta.com.au (A. Boulis), simonhan@ee.ucla.edu (C.-C. Han), roy@ee.ucla.edu (R. Shea), mbs@ee.ucla.edu (M.B. Srivastava).

1574-1192/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.pmcj.2007.04.007

1. Introduction

Wireless ad hoc sensor networks (WASNs) have drawn a lot of attention in recent years from a diverse set of research communities. Researchers have mostly been concerned with exploring applications scenarios, investigating new routing and access control protocols, proposing new energy-saving algorithmic techniques, and developing hardware prototypes of sensor nodes.

Little concern has been given to how to actually program the WASN. Most of the time, it is assumed that the proposed algorithms are hard-coded into the memory of each node. In some platforms the application developer can use a node-level OS (e.g. TinyOS [15]) to create the application, which has the advantages of modularity, multi-tasking, and a hardware abstraction layer. Nevertheless the developer still has to create a single executable image to be downloaded manually into each node. The difficulty in algorithmic deployment is one of the reasons that deter WASN researchers to cross the simulation boundary and move to real platform implementations (the other main reason is difficulty in the physical deployment of nodes at an interesting environment). Furthermore, it is widely accepted that commercially-deployed WASNs will have long-deployment cycles and serve multiple transient users with dynamic needs. These features clearly point in the direction of dynamic WASN programming.

What kind of dynamic programmability should WASNs provide? Having a few algorithms hard-coded into each node but tunable through the transmission of parameters, is not flexible enough for the wide variety of possible WASN applications. Having the ability to download executable images into the nodes is not feasible because most of the nodes will be physically unreachable or reachable at a very high cost. Having the ability to use the network in order to transfer the executable images to each and every node is energy inefficient (because of the high communication costs and limited node energy[1]) and cannot allow the sharing of the WASN by multiple users. What we ideally want is to be able to dynamically program the WASN as *a whole, an aggregate*, not just as a mere collection of individual nodes. This means that a user, connected to the network at any point, will be able to inject instructions into the network to perform a given (possibly distributed) task. The instructions will task individual nodes according to user needs, network state, and physical phenomena, *without any intervention from the user*, other than the initial injection. Furthermore, since we want multiple users to use the WASN concurrently, several resources/services of the sensor node should be abstracted and made sharable by many users/applications.

One approach of programming the WASN as an aggregate, is viewing it as a distributed database system (e.g., [28]). Multiple users can inject database-like queries to be autonomously distributed into the network. The query's task is to retrieve the needed information by finding the right nodes and possibly aggregate the data as they

[1] A recent research effort by Reijers and Langendoen [31] has devised a scheme that sends only incremental changes of the executable image through the radio, saving considerable energy. Nevertheless, one still needs a central entity to control all tasks of the nodes (thus, no multi-user support), and all nodes run the same code (i.e., there is no notion of code distribution area). Furthermore, there is no notion of abstraction of the run-time environment, resulting in fairly sizable code transfers even when the scheme is applied.

are routed back to the user. The database approach follows a declarative model, i.e., the user states what he wants and the system provides it. The user does not describe *the way* (the algorithm) the information is to be extracted, thus the underlying system uses predefined ways (e.g., tree formation to find the sources and aggregation of data through the reverse path). Thus, even though the database approach is programming the network in the desirable way, it is not expressive enough to implement arbitrary distributed algorithms.

A different approach to WASN programmability that is used by our framework, is the "active sensor" approach. This term was used in [25], to describe a family of frameworks that take an imperative approach and try to task sensor nodes in a custom fashion, much like active networking frameworks task data network nodes. The difference is that while active networking tasks react only to reception of data packets, active sensor tasks need to react to many types of events, such as network events, sensing events, and timeouts. Active sensor frameworks abstract the run-time environment of the sensor node by providing useful native services, and oftentimes by installing a virtual machine or a high-level script interpreter at each node. Moreover, pieces of code (native, scripts, or bytecodes) are made mobile, so nodes can be tasked using the network.

The difficulty in designing an active sensor framework is deciding how to properly define the abstraction of the run-time environment so that one achieves compactness of code, sharability of resources for multi-user support, portability in many platforms, while at the same time keeping a low overhead in delays and energy. The innovation of our framework, called SensorWare, is that it enables a versatile abstraction of the sensor node by dynamically installing new services. Furthermore, SensorWare targets not-so-memory-constrained nodes, allowing for a rich set of services, multi-user support, resource management and a high-level scripting language to facilitate programming. Finally, giving the ability to the mobile code to move autonomously, makes the algorithm's deployment and execution concurrent. This means that the code is transferred selectively only to the needed nodes and not to all the nodes belonging to a statically predefined area.

Section 2 discusses the nature of WASNs, approaches to WASN programmability, and the general idea of our approach. Section 3 presents SensorWare's architecture. Section 4 illustrates SensorWare's capabilities and versatility by presenting three versions of a geographical routing application. Section 5 presents our current implementation and the measurements we acquired through it. One set of important measurements shows the trade-offs between the three versions of the application presented in Section 4, making the argument for the necessity of versatile abstraction more tangible. Section 6 presents related work. Finally, Section 7 concludes the paper.

2. Motivation and background

2.1. Wireless ad hoc sensor networks

A WASN is an ad hoc network of resource-limited, static, wireless, sensor nodes that is being used to monitor dynamic physical processes. Typically a user queries the network (consider the term "query" in a broad sense, not just database query), the query triggers some reaction from the network, and as the result the user receives the information needed. The reaction to the query can vary from a simple return of a sensor value, to a complex

unfolding of a distributed algorithm among some or all of the sensor nodes, such as a collaborative signal processing algorithm, a distributed estimation algorithm, or an actuator control algorithm. Furthermore, there are multiple users who are transiently connected to the network, each having different needs in requested information.

These systems are quite different from traditional networks. First, they have severe energy, computation, storage, and bandwidth constraints. Second, their overall usage scenario is quite different from traditional networks. There is not a mere exchange of data between users and nodes. The user will rarely be interested in the readings of one or two specific nodes. The user will be interested in some parameters of a dynamic physical process. To efficiently achieve this, the nodes have to form an application-specific distributed system to provide the user with the answer. Moreover, the nodes that are involved in the process of providing the user with information are constantly changing as the physical phenomenon is changing. Therefore the user interacts with the system as a whole. The WASN is not there to connect different parties together as in the traditional networking sense but to provide information services to users.

As a consequence, efficiently designed WASNs operate in a fashion where a node's actions are affected largely by physical stimuli detected by the node itself or nearby nodes. Frequent long trips to the user are undesirable because they are time and energy consuming. This decentralized (i.e. not all traffic flows to/from user), autonomous (i.e., user out-of-the-loop most of the time) way of operating, is called "proactive computing" (as opposed to interactive) by Tennenhouse [34]. We also adopt the term "proactive" to denote an autonomous and non-interactive nature.

Efficiently designed WASNs are application-specific distributed systems that require a different distributed proactive algorithm as an efficient solution to each different application problem. From this remark arises the generic deployment problem: How does one dynamically deploy different algorithms into the network, what is the programming model that will implement these algorithms, and what general support does one need from the framework?

2.2. Approaches to WASN programmability

As mentioned in the introduction, a popular approach to dynamic WASN programmability views the WASN as a distributed database. The data exist in the network and have to be found, probably processed in predefined ways (e.g., aggregated) and delivered to the user. Heidemann et al. [12] closely follow this model without explicitly employing traditional database terms and mechanisms. They focus on a data-driven low-level naming scheme based on attributes. A query describes the data it is looking for and directed diffusion [18] is used as the underlying routing protocol. The data can be processed with predefined filters as they are routed back to the user. Other systems, such as Cougar [1], focus more on transferring the SQL semantics of traditional databases to the distributed setting of WASNs. In this case, the naming system developed in [12] is replaced by an SQL equivalent. Each node is equipped with a fixed database query resolver. As queries arrive to a node, the local resolver decides on the best, distributed plan to execute the query and distributes the query to the appropriate nodes. A more recent system that follows the database model is TinyDB [28] developed in Berkeley. Focusing on aggregate queries (e.g.,

min, max, avg) TinyDB provides special optimizations (e.g., exploitation of the shared medium, hypothesis testing).

The strong point of the database approach is that it offers an intuitive way to extract information from a WASN while hiding the complications of *embedded and distributed* programming. The model's limitation though is that there are only pre-defined ways to process the data, which implies that only certain types of applications (e.g., aggregation applications) are addressed in the most efficient way by the database model. If a new way to process and react to the data is needed by application X, this can only be done at the user node (assuming that the human-controlled user node is easily upgradable). Consequently, the algorithmic pattern to address application X under the database model will be an iteration of the generalized steps: (1) partially processed data arriving to the user node, (2) data undergoing custom processing and (3) based on the result a new database query is issued. In most cases, this is not the structure of the most efficient algorithm to solve an application problem.[2]

On the other hand, we have active sensor frameworks (instances of which are discussed in the "Related Work" section) that seek to remedy the limited flexibility problem of the database model at the expense of increased responsibility for the programmer. In this case, the distribution and operation of the algorithm is explicitly designed by the programmer (i.e., it is not a task of the system), allowing for any distributed algorithm to be executed in the sensor network. To get a clear view of the type of applications/algorithms that the active sensor model is superior to the database model refer to [8]. Flexibility is not the end of the road for these frameworks though. They have to strive for efficiency too. Efficiency means small code transfers and small overheads in the executed code, which in turn translates into choosing the "proper" abstraction for the run-time environment of the sensor node. If the abstraction is too low-level then the code might be efficient but it will also be large. If the abstraction comprises many high-level services, then apart from the memory required on the node to host such services, the executed code might not be as efficient as some services will not be exactly tuned to the application's needs. Current active sensor frameworks do not address the abstraction problem. They adopt some level of abstraction assuming it is the right one for all applications. In this paper we will show how our framework deals with the abstraction problem and the consequences of a poorly chosen abstraction. Another important issue for active sensor frameworks is the distribution of the algorithm. Some frameworks (especially those that adopt the transfer of native code) separate the distribution from the execution of the algorithm. That is, the code has to be deployed a priori to the whole network, or at best to a statically predefined area. With some applications this means that code is unnecessarily transferred to more nodes, simply because there is no way to know the right nodes before the algorithm actually executes. An example of such a case is offered in the "Code examples" and "Measurements" sections.

[2] In an effort to deal with this problem TinyDB researchers have added the capability of dynamically upgradable aggregation functions, and special language constructs to install event triggers. With these upgrades TinyDB is essentially moving towards the realm of active sensor frameworks. Even though it is not clarified how the upgrades are made, there is no notion of abstracting the run-time environment (so presumably this is done in native code). Furthermore even with these modifications, the user has only partial control on the algorithm's distribution and execution, thus the efficiency–flexibility problem (i.e., executing the right distributed algorithm) is not eliminated.

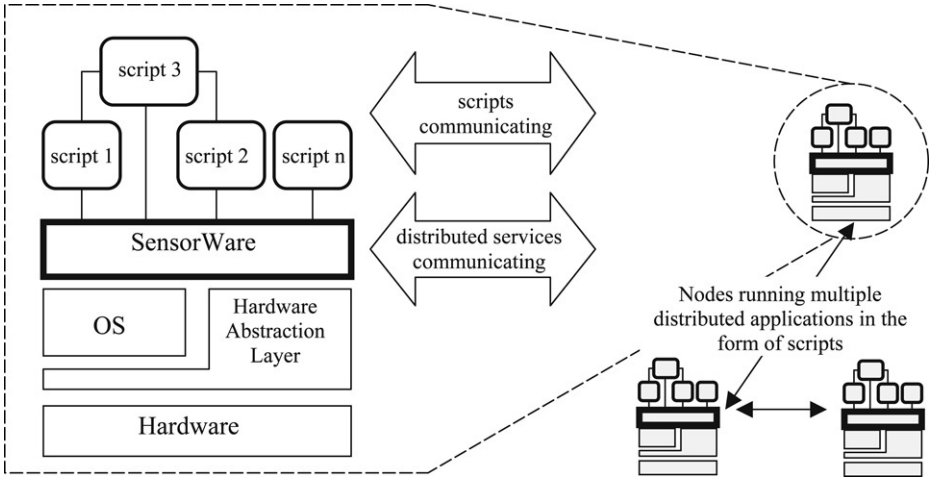

Fig. 1. The general sensor node architecture.

2.3. SensorWare

Our system, SensorWare, adopts an active sensor approach to allow any distributed algorithm to be executed in the network. A distributed algorithm can be viewed as a set of collaborating programs executing in a corresponding (often time-varying) set of nodes. In SensorWare these programs are sensor-node control scripts. The sensing, communication, and computing resources of a node are exposed to the control scripts through abstraction services. These services share the resources of a node among many applications and users that might concurrently use the WASN. One key feature of SensorWare is that it allows the dynamic instantiation of new services (coded as control scripts) enabling a versatile abstraction of the sensor node run-time environment. A SensorWare script can autonomously replicate or migrate its code and data to other nodes, directly affecting their behavior. The replication or migration of a script will be called "population" in the paper. The user "injects" the query/program into the network, and the query *autonomously* unfolds the distributed algorithm only into the nodes that should be affected. For most applications these nodes can only be known at run-time, thus the deployment of the algorithm has to be tied with the execution of the algorithm. With the features of versatile abstraction and autonomously mobile code a programmer can choose the implementation that best fits the application's needs. For instance, the programmer can design a self-sufficient, self-deployed script, which is based only on native services, or a custom pre-deployed service and a script that uses this service. Examples of such design trade-offs are described in the "Code examples" and "Measurements" sections.

3. Architecture

Fig. 1 shows SensorWare's place inside a layered sensor node's architecture. The lower layers are the raw hardware and the hardware abstraction layer (i.e., the device drivers). An

operating system (OS) exists on top of the lower layers. The OS provides all the standard functions and services of a multi-threaded environment that are needed by the layers above it. The SensorWare layer for instance, uses those functions and services offered by the OS to provide the run-time environment for the control scripts. The control scripts rely completely on the SensorWare layer while populating through the network. Control scripts use the native services that SensorWare provides as well as services provided by other scripts to construct distributed applications. Scripts will (usually) communicate with scripts of the same kind (i.e., having the same code) at remote nodes.

Two things comprise SensorWare: (1) the language, and (2) the supporting run-time environment. The next two subsections describe each of these two parts. A third subsection discusses the issues of addressing and routing in SensorWare.

3.1. The language

As discussed earlier, the basic idea is to make the nodes programmable through mobile control scripts. In this subsection, the basic parts that comprise the language are described as well as the programming model that emerges from the parts.

First, a scripting language needs commands to act as the basic building blocks of the scripts. These commands are essentially the interface to the abstraction services offered by SensorWare. Simple examples include: timer services, acquisition of sensing data, mobility of scripts, a location discovery protocol. Second, a scripting language needs constructs in order to tie these building blocks together in control scripts. Some examples include: constructs for flow control, such as loops and conditional statements, constructs for variable handling and constructs for expression evaluation. We call all these constructs the "glue core" of the language. As a glue core one can use the core from one of the scripting languages that are freely available. One such scripting language, that is well suited for SensorWare's purposes, is Tcl [29], offering great modularity and portability. Thus, the Tcl core is used as the glue core in the SensorWare language. All the basic commands are defined as new Tcl commands using the standard method that Tcl provides for that purpose.

As SensorWare's design progressed over time, the set of commands changed considerably. We started with some basic commands for script mobility along with some commands for timer, network, and sensing abstraction, and kept adding commands as necessary. This tactic brought up an important issue: How do we handle the language evolution? An equivalent face of the problem is: How do we handle platform variability? Imagine two node platforms with different modules (e.g., one has a GPS device, the other has a second radio), how do we handle both platforms with the same framework? Do we allow SensorWare to be arbitrarily expanded? As a solution to this issue, SensorWare declares, defines, and supports the creation of virtual devices (an idea triggered by Linux's virtual devices). *All abstraction services are represented as virtual devices*. There is a *fixed interface* for all devices. More specifically there are four commands that are used to communicate with the device. They are: query, act, interest, and dispose. Query asks for a piece of information from the device and expects an immediate reply. Act instructs the device to perform an action (e.g., modify parameters of the device, or if the device is an actuator perform an action). Interest describes a specific event that the device can produce

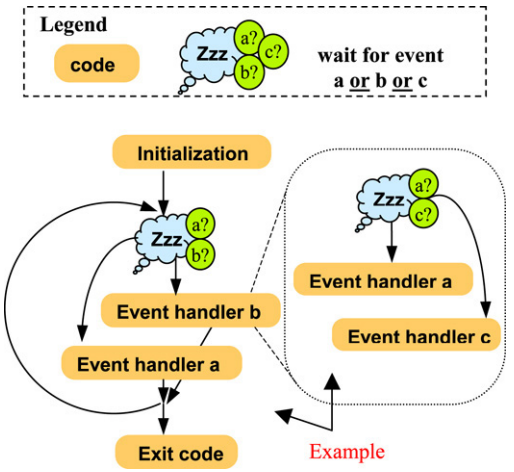

Fig. 2. The programming model.

and gives this event a name/ID. The name can be used subsequently by the wait command to wait on this specific event. Dispose just disposes an event name. Additionally, if a device can produce events, a task is needed to accept interest and dispose commands and react to wait commands that are waiting on the device's events. The task definition, and the parsing of the arguments of the four commands are defined by the developer. This is where the expandability stems from, while at the same time keeping a structured form.

3.1.1. The general programing model

An intuitive description of a sensor node task (a part of a distributed application) has the form of a state machine that is influenced by external events. This is also the form of SensorWare scripts. The programming model that is adopted is equivalent to the following: An event is described, and it is tied with the definition of an event handler. The event handler, according to the current state, will do some processing and possibly create some new events or/and alter the current state. Fig. 2 illustrates SensorWare's programming model with an example.

The behavior described above is achieved through the wait command. Using this command, the programmer can define the events that the script is waiting upon, at a given time. An interesting question to ask at this point is: what kind of events are the scripts waiting on? Fig. 3 presents the answer. By making scripts waiting on high-level events, results in more efficient code (since most of the work is done by the natively written services, and the script is invoked only when a less-frequent high-level event happens) plus the programmer takes full advantage of the offered services. Examples of events that a script can describe to a device and wait upon are: (i) reception of a message, (ii) threshold traversal for a sensing device, (iii) filling of a buffer with sensing data of a given sampling rate, (iv) expiration of several timers, (v) return of the node's location within some error bounds.

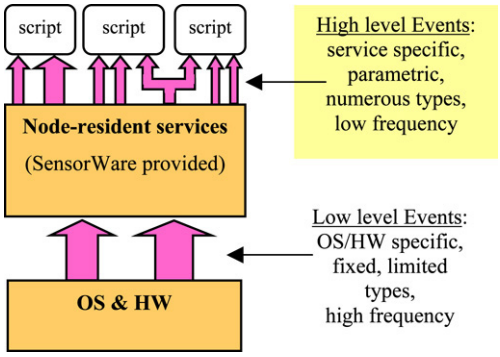

Fig. 3. Type of events processed by scripts.

When *one* of the events declared in the wait command occurs, the command terminates, returning the event that caused the termination. The code after the wait command processes the return value and invokes the code that implements the proper event handler. After the execution of the event handler, the script moves to a new wait command, or more usually it loops around and waits for events from the same wait command. Although the scripts are defining behavior at the node level, SensorWare is not a node-level programming language. It can be better viewed as an event-based language since the behaviors are not tied to specific nodes but rather to possible events that depend on the physical phenomena and the WASN state.

3.2. The run-time environment

Equally important to the programming model is the run-time environment that supports the scripts. The structure of this environment evolved along with the language. Initially, it included a few basic tasks, such as the script manager, radio and timer abstraction tasks, which are universally useful in all sensor node platforms, but the problem of expandability quickly arose: Would we allow arbitrary definition of tasks according to the platform? How would different tasks interact? The solution came again through the notion of devices. Devices allow a structured run-time environment. As mentioned briefly in the language subsection, devices that can create events (the majority of devices) have to define an accompanying task to process and provide these device-related events to the scripts. When a script uses the interest command of a device, it names and registers an event description of this device. When wait is called on this event name, the device starts waiting on the appropriate lower-level events, process these events, and starts "building" the higher level event. For instance, waiting on an event description of the sort "fill a buffer of 10 temperature samples taken at 1 kHz", makes the temperature abstraction device to take individual temperature samples until the buffer is filled. When this happens the script's wait command returns with the high-level event. Devices can also define other internal tasks to perform certain functions (e.g., a time sync service has to run the proper distributed protocol).

SensorWare has a well-defined, structured way of providing services (i.e., by registering a device name, defining functions for the four basic device-interaction commands, and

defining any device tasks). So far, we have presented the definition of native services (i.e. definition of devices) as static; that is, a task to be done at SensorWare's compile time. There is nothing though that prevents one from making the device definition and registration a dynamic procedure. Indeed, SensorWare uses scripts to dynamically define and register new devices. The script device has to define four specially named procedures to implement the four basic device-interaction commands (i.e., query, act, interest, dispose) and register the device name using the special command device. The script's main body (i.e., the main wait loop) implements the device tasks. By allowing the dynamic instantiation of new devices (i.e., services) *SensorWare can change the abstraction of the run-time environment offered to subsequent scripts*. We will show how this feature can improve efficiency (measured in reduced communication, thus energy consumption) in Sections 5 and 6.

Finally, there are a few non-device tasks, such as the Script Manager. The Script Manager accepts all requests for the spawning of new scripts and keeps any script-related state. Passing data (state) along with the script code is achieved through the commands carry and parameter. Carry (executing before a script spawns) passes a list of values to the script, while parameter (executing from within the script) assigns variable names to these values. These variable names can then be used as externally defined parameters within the script. If the script replicates or migrates, it automatically carries with it the most recent values of its parameters.

For more information on SensorWare's run-time issues, and general features, such as code safety, resource management, and script compression the reader can refer to [5,7] and [3].

3.3. Addressing and routing

So far, the problems of script addressing and routing were left open. Addresses in SensorWare have the generic format: [nodes_specification mailbox]. Nodes_specification as the name suggests is a description of nodes. The resolution of this description is essentially an act of routing. In its simplest form, the specification could just be a node ID of a neighbor (no need to do any routing, just transmit to that particular neighbor). A more advanced specification would be a remote node ID along with the routing protocol used to reach it. An even more advanced specification is a set of attribute-described nodes (e.g., nodes belonging to a geographical region in the form [point, radius], along with a geographical protocol to interpret the attributes and perform the routing). Mailbox is a tuple needed to describe a specific instance of a script. It has the string form of: userID.codeID.appID. UserID is just the user's id that spawned the script in question. CodeID is an id describing the script. AppID is an id denoting the application (i.e. collection of scripts) that the particular instance of a script belongs to. It is also used to distinguish instances of the same script (same codeID) running under the same node and under the same user (but for different applications).

As hinted by the preceding paragraphs addressing and routing are interwoven. The format of addressing is fixed but allowing for multiple routing protocols. Routing can take multiple facets in a WASN (e.g. directed diffusion, geographical routing, energy aware unicast, multicast to members of a cluster, etc.). All these examples can be used by different

applications or even by the same application according to circumstances. Furthermore, many applications can use their own custom-made routing, or more frequently, no routing at all, as they are restricted to purely 1-hop local interaction. Thus, SensorWare needs to provide a way to easily export the functionality of multiple routing protocols to the scripts and allow the easy insertion of new routing protocols at SensorWare compile-time or even run-time. One clear way is to define routing protocols as devices in SensorWare. Nodes_specification becomes (<device_name>, <attributes>). Calling a SensorWare command that specifies some nodes (e.g. the send, spawn, or replicate commands) involves a routing device. The command call triggers a special function of the routing device, handling the routing part of the command. Thus, apart from the basic four device functions a routing device should define a fifth function to implement the needed routing functionality. Furthermore, a task is needed to handle packets received from the radio and use the routing protocol the device implements.

4. Code examples

In order to give a more concrete description of SensorWare and provide more insight into the feature of versatile abstraction, we will present three versions of an application. Although SensorWare has been used to implement algorithms for practical problems (e.g., tracking a target and actuating a camera, distributed estimation in periodic aggregation [4]), these are not suitable for showcasing SensorWare's capabilities in the limits of a technical paper. Therefore we carefully chose a geographical routing task to act as our application for illustrative purposes. The application is the following: We wish to spawn a certain script in a geographic area, defined by a point in space and a radius. The script will do its job (in our case it will just send a message to the node's debugging device) and then send a message back to the user. The application's total execution time is the time from the user's initial request (i.e., the invocation of an initial script) till the time all the messages from the spawned scripts return to the user. The three versions of this geographical routing application are the following: (1) Implementation based on a native service providing point (not area) geographical routing, (2) Implementation *not* based on any native services (apart from the device neighbor, which provides a list of immediate neighbors and trivial routing to neighbors), and (3) Implementation that uses the code in the second version to deploy a new service (in the form a script device) and then uses this new service. It is interesting to see the delay and energy trade-offs of these three versions. In this section we present and explain the code of the different implementations. In the next section we present the measurements for each of the versions taken on a real platform.

4.1. Geo-routing using a natively implemented service

We begin with the listing of the script we wish to spawn in a geographical area. Since we are using this script only to illustrate the abilities of our application, it remains fairly simple, and thus it makes a proper first example. All scripts start with a tcl comment line (i.e., a line beginning with the character '#'), that begins with the word "code_id" and is followed by a number. This line is used by the spawn command (a command used to spawn a new script) to assign a code id to the script. None of the comment lines are transferred

Listing 1
The script we wish to populate in an area

```
#code_id 65 populated_script
# to be used with gpsr test

parameter user_x user_y

debug "POPULATED SCRIPT: I am here"
send gpsr $user_x $user_y" [id -u].63.[id -a]" "nodeid [id -n]"
```

with the code around the network. The first SensorWare command of this specific script is parameter, which gives the names of the two script parameters (i.e., variables that are externally set).

The debug command simply outputs a message to the debugging interface of the script. Finally, send uses the native geographic routing service and the known user position to route back a message to the user. Notice that send is using the device gpsr (the device implements the well-known protocol Greedy Perimeter Stateless Routing [21]) along with two parameters (i.e., the x and y coordinates of the destination point) to give a < nodes_specification> as discussed in the previous section. The message is delivered to script with id 63 (we will see shortly which is this script). The id command, used inside send, returns different identifying values, such as node_id, user_id, code_id, and app_id, depending on the command's arguments.

As mentioned already, the device gpsr routes to a single target point and not an area, as our application requirement dictates. Nevertheless, we could use gpsr to provide routing to the center of the area we wish to route and then use another script to flood an area of radius r around this point. This is the functionality of the script given in Listing 2. The parameters of this script are: the values that define the spawn area, the script-to-be-spawned (an interesting feature of SensorWare is that scripts can be treated as any other variable), and the user coordinates since the script-to-be-spawned needs them (as we saw in Listing 1). The script in Listing 2 defines a procedure to calculate the squared Euclidean distance between two points and uses it to determine whether or not it should flood.

Listing 3 presents the code that uses the gpsr device and the flood script to achieve the application's goal. This is the script with code id 63 (i.e., the script that the replies are routed back to). This script needs to be spawned only in the user node. Its main function is to use the gpsr device to route the flooding script to the center of the target area. Then it simply waits for replies from the spawned scripts.

This is a good point to reveal some of the details of the wait command. Wait returns when an event named in the command's arguments occurs. In order to expedite processing of the event by the subsequent script code, the wait command sets the following predefined variables: (1) "event_name" containing the name of the occurred event, and (2) "event_body" containing data returned by the event (e.g. the data of a network packet). There are also some common event names, such as "packet", which are predefined in order to make the code clearer and smaller.

Finally we present the invocation script in Listing 4. This is not a SensorWare script in the sense that it does not have a code id and does not roam around the network. It simply

Listing 2
The script that floods an area of radius r around a target point

```
#code_id 64 radius_flood2
# to be used with gpsr test

parameter target_loc_x target_loc_y radius\
            populated_script user_x user_y

proc dist {x1 y1 x2 y2} {

          set diff_x [expr "$x1 - $x2"]
          set diff_y [expr "$y1 - $y2"]
          return [expr "($diff_x * $diff_x) + ($diff_y * $diff_y)"]
}

set my_loc [query location]
set my_loc_x [lindex $my_loc 0]
set my_loc_y [lindex $my_loc 1]
set dist_squared [dist $my_loc_x $my_loc_y\
              $target_loc_x $target_loc_y]

# ********************************************************************
# We check if we are inside the area that must be flooded.
# if yes, we replicate in ALL neighbors and spawn the
# populated_script
# ********************************************************************

if {($radius <0) || ([expr $dist_squared-$radius*$radius] <= 0)}
{
debug "flooding into neighborhood"
          replicate neighbor 0
          carry $user_x $user_y
          spawn self $populated_script
}
```

loads all the relevant SensorWare scripts and sets the parameters to the desired values. The user simply executes an invocation script at the user node and the distributed application is deployed.

4.2. Geo-routing with no native service support

Imagine that the gpsr device is not included in the package of native services offered by SensorWare. This, for example, might be true if we decide to reduce the footprint of SensorWare by reducing some of the implemented devices. Or simply imagine that a programmer, finding the service of gpsr inadequate (since it only routes to a point), wants to implement the application from scratch. This is the case of the second version of the geo-routing application. The first change is noted to the populated script. Since there is no gpsr device to route replies back to the user, we have to use another service. This is the updated send command:

```
send self "[id -u].73.[id -a]" "nodeid [id -n]"
```

Listing 3
The script that routes to an area

```
#code_id 63 gpsr test
parameter target_loc_x target_loc_y radius\
           populated_script radius_flood_code
set my_loc [query location]

carry $target_loc_x $target_loc_y\
          $radius $populated_script\
          [lindex $my_loc 0] [lindex $my_loc 1]
spawn gpsr $target_loc_x $target_loc_y\
          $radius_flood_code

while {1} {
    wait packet
    debug "Arrived at user: $event_body"
}
```

Listing 4
The invocation script

```
# This is an invoke script.
# Just source it in the terminal to start the app.
# Populates script in a geographical area.
# Uses native service GPSR

set target_loc_x 45
set target_loc_y 37
set radius 8
load populated_script\
        /home/boulis/scripts/geo/GPSR/populated_script2.txt
load radius_flood\
        /home/boulis/scripts/geo/GPSR/radius_flooding2.txt
load gpsr_test\
        /home/boulis/scripts/geo/GPSR/gpsr_test.txt

carry $target_loc_x $target_loc_y $radius\
         $populated_script $radius_flood

spawn self $gpsr_test
```

The message "nodeid [id –n]" is now routed to another script in the *same* node. The recipient script is a user relay script that keeps the next node towards the user (see Listing 5).

The new flooding script is given in Listing 6. Its code is very similar to the corresponding script of the first version with the addition that now it has to spawn the relay script too. This includes setting the parameters of the relay script. The single parameter of the relay script is set differently depending on whether the script was created through self-replication or another script spawned it. The variable "parent_node" is a predefined

Listing 5
The user relay script

```
#code_id 73 relay
parameter relay_node
while {1} {

            wait packet

            if {$relay_node == [id -n]} {
                debug "ARRIVED USER: $event_body"
            } else {
                send neighbor $relay_node\
                    [id -u].[id -f].[id -a] $event_body
                debug $event_body
            }
            if {[lindex $event_body 0] == "Error:"}
              {debug "EXIT relay"; exit}
```

Listing 6
New flooding script

```
#code_id 74 radius_flood
# uses relay

parameter target_loc_x target_loc_y radius\
            populated_script relay_code relay_node
...the same proc and location acquisition code
as before....

if {($radius <0) || ([expr$dist_squared-$radius*$radius] <= 0)}
{
debug "flooding into neighborhood"
            replicate neighbor 0
# we have to set correctly the parameter to relay_code
            if {$parent_node == [id -n]} {
            # this script was spawned through
            # geo_routing_flooding
                  carry $relay_node
            } else {
            # this script was spawned through self-replication
                  carry $parent_node
            }
            spawn self $relay_code
            spawn self $populated_script
}
```

variable containing the node ID of the node that hosted the script that spawned the currently executing script.

Listing 7 is the script that achieves the geographical routing with the help of the flooding script. The comments in the code offer an explanation of the script's basic functions.

It is interesting to see how the script in Listing 7 gets the location information from the neighboring nodes. From Listing 7 we can see that a script stored in the variable "ask_loc_code" is used. Listing 8 is its single line code.

The invocation script is similar to the corresponding script of the first version, thus it is not listed here. It is interesting to note that although we did not use any high-level services we were able to implement our geographical routing application, paying the price of larger scripts. Note also that the relay script and ask_loc script could be omitted since there are native SensorWare services that cover them (the service/routing protocol named "user" essentially implements the relay script natively and a feature in the location service can return the location of neighbors) but abiding to the no-high-level-services rule we chose not to use them.

4.3. Geo-routing using script device

Suppose that we have many requests for routing to several geographical areas. In the previous case we developed code that met the needs of a geographical routing request. It would be desirable if other scripts could use this code as a resident service. We can achieve such a dynamic instantiation through the use of the script devices. This is the third version of our implementation. We first use the code from the second version to define a new script device called "GEO_script". This modification keeps most of the previous code and simply rearranges it in a form of a device description (i.e., defining the special device functions and the device task) and registers the new device under the name GEO_script. The device script is pre-deployed throughout the network, much like the code update programming schemes do. Then other scripts can use this new device. Listing 9 shows such a script, using the newly instantiated script device.

The third version of our implementation can be beneficial if we have many requests for geographical routing to an area. In the next section we quantify this argument.

5. Measurements

Some active sensor frameworks choose to evaluate their performance by showing their expressiveness. They create a distributed algorithm for a particular application and compare it against a more centralized approach (usually a distributed database approach). We believe that the energy savings from such comparisons are evident for *any* active sensor framework and do not add value to the investigation and evaluation of the framework. To evaluate SensorWare we chose to implement it and measure the overheads we are paying for dynamic programmability. How much memory do SensorWare and its components occupy? How much delay is introduced by various SensorWare operations? These questions are answered in Section 5.2. Furthermore, we use our implementation platform to quantify the trade-offs between the different run-time abstractions, showing that there is no single best solution. This way we also show the necessity of versatile abstraction that SensorWare offers. We begin with a description of the implementation platform.

5.1. Platform description

The prototype platform used in the implementation and evaluation of SensorWare was built around the iPAQ 3950 [19]. The iPAQ has an Intel Xscale 32-bit processor, running

Listing 7
The script that routes to an area

```
#code_id 70 geo_route
# uses radius_flood relay ask_loc

parameter target_loc_x target_loc_y radius\
          populated_script radius_flood_code\
          relay_code ask_loc_code
...the usual dist proc and location acquisition code....

if {($radius <0) || ([expr $dist_squared-$radius*$radius] <= 0)}
{
        carry $target_loc_x $target_loc_y $radius\
            $populated_script $relay_code\
        $parent_node
        spawn self $radius_flood_code

        exit
}
# ********************************************************************
# Query each neighbor to find their location. We broadcast
# this script since most nodes will have more than one
# neighbor. Note that we clear the queue before we start
# (by waiting on nothing)
# ********************************************************************
wait
spawn neighbor 0 $ask_loc_code
# ********************************************************************
# We wait for some time to hear back from the neighbors and
# choose the best candidate to proceed
# (node closest to target point)
# ********************************************************************
interest timer t 2000
while {1} {
          wait packet t

            if {$event_name == "t"} {break }
            set tmp_x [lindex $event_body 0]
            set tmp_y [lindex $event_body 1]
            set neigh_dist_squared [dist $tmp_x $tmp_y\
                    $target_loc_x $target_loc_y]

            if {$neigh_dist_squared <$dist_squared} {
              set dist_squared $neigh_dist_squared
              set best_node $msg_src_node
            }
}
# ********************************************************************
# We need to set up a relay agent to get packets back to the user.
# ********************************************************************
```

(*continued on next page*)

Listing 7 (*continued*)

```
carry $parent_node
spawn self $relay_code

# ********************************************************************
# Check if a better node was actually found. If yes proceed
# with the replicate (and exit), otherwise send an error
# message
# to the relay agent (and exit)
# ********************************************************************
if {$best_node == 0} {
            debug "no easy route to target"
            send self "[id -u].73.[id -a]"\
                   "Error: no route, stop at node $my_addr"
} else {
            replicate neighbor $best_node
}
```

Listing 8
The script that retrieves the location information

```
#code_id 71 ask_loc
send neighbor $parent_node [id -u].70.[id -a] [query location]
```

Listing 9
Routing using the script device

```
#code_id 63 test GEO_script service
parameter target_loc_x target_loc_y\
            radius populated_script

spawn GEO_script $target_loc_x $target_loc_y\
          $radius $populated_script
while {1} {
  wait packet
  debug "Arrived at user: $event_body"
}
```

at variable speeds (we executed all versions of the geo-routing application at 100, 200, and 400 MHz). The flash memory size is 16 MB and the RAM memory size 64 MB. The OS installed is a familiar v0.5 Linux StrongARM port [11], kernel version 2.4.18-rmk3. The compiler used is the gcc cross-compiler. A wavelan card [35] is used as the radio device and a Honeywell HMR-2300 Magnetometer [16] as the sensing device. Fig. 4 shows one node of the platform on the left side, and the network used in our experiments on the right side. The notebook in the corner of the network is the user node. The arrows show the allowed communication channels. SensorWare is also ported into the Rockwell WINS nodes [32] that have a StrongARM processor, but only 1 MB of flash memory. Both eCos [10] and

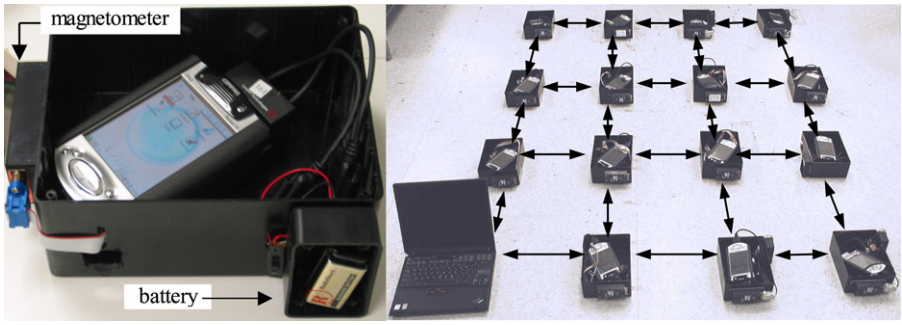

Fig. 4. The implementation platform.

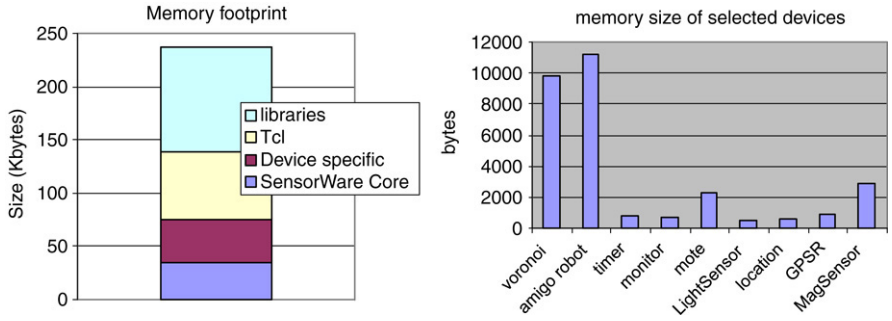

Fig. 5. Code size breakdown.

microC/OS-II [24] were used as operating systems for these nodes. SensorWare is also operational in Stargate, Sensoria and PASTA nodes.

5.2. *Memory size measurements*

The first question to answer is how much memory does the whole framework occupy. Fig. 5 shows that the total memory size is 237 KB and it is consisted of 97 KB of Linux specific code (e.g., libraries), 64 KB of a stripped down Tcl core called tinyTcl, 35 KB of SensorWare core code and 40 KB of device specific code.

The right part of the figure shows the code size of selected devices. Some of the devices are platform specific. For example, we will not include the "amigo robot" device if our sensor node does not have a robot attached to it.

5.3. *Delay measurements*

The next question to answer is how long do different basic commands need to execute. We measured each command individually 100 times under the same basic conditions (only one script executing, processor speed at 200 MHz) and derived an average and standard deviation for the delay. Most commands exhibited negligible variance. All the commands, except send and spawn (sending a 17 B message and spawning a 59 B script respectively), have an execution time less than 0.14 ms.

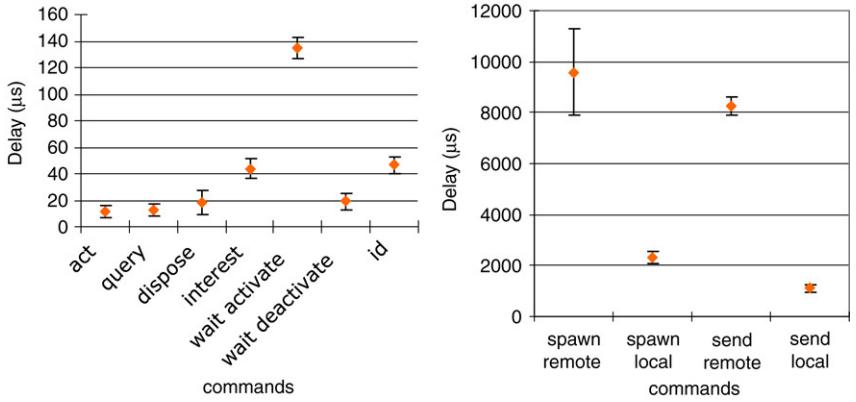

Fig. 6. Execution times of SensorWare commands.

Of particular interest is the "wait activate" delay. This delay refers to the time it takes since a new event name is entered as an argument in the wait command, till the time the wait command is ready to intercept this event. For instance, the overhead of setting a zero-valued timer, and wait for its expiration, is 0.18 ms (i.e., the overhead of interest and wait activate). Note that this overhead has small variation, a fact that facilitates real-time programming. In other words, the programmer (or the system) can internally subtract this number each time a script sets a timer, in order to measure the true desired time. The right part of Fig. 6 shows the delays of spawn and send, for both local and remote execution. For the remote execution the radio delay is not included. The local operation is much faster since the messages (data or code) do not have to go through the whole communication stack (i.e., routing, compression, fragmentation). All delay measurements were acquired using the gettimeofday() system call. This function is based on the timer count register found in the StrongARM processor. The accuracy of this method is measured to be 1 μs.

5.4. Trade-offs between different run-time environment abstractions

In this section we quantitatively evaluate the three versions of the geographical routing application described in Section 5. For each of the three versions we measured: (1) the total transmitted bytes per node, and (2) the total execution time (as defined in the beginning of Section 5) for different processor speeds. From the measurements at different speeds we were able to derive the breakdown of the total execution time in communication and computation components.

Fig. 7 shows the total transmitted bytes per node in our grid network setup (depicted in Fig. 5). The x and y axis represent the x and y coordinates of each node. The user node is at position (1,1). The different colors of the bars were used to make the figure more legible. Major remarks: The version using GPSR transmits fewer bytes than the pure script version, overall. Their difference though is not great, since GPSR supports only point routing, thus the populated script has to be encapsulated in a flooding script as we described in Section 5. Using the script device yields the best results (bottom right corner of the figure) but requires the large initial overhead of deploying the device (bottom left corner). Concerning

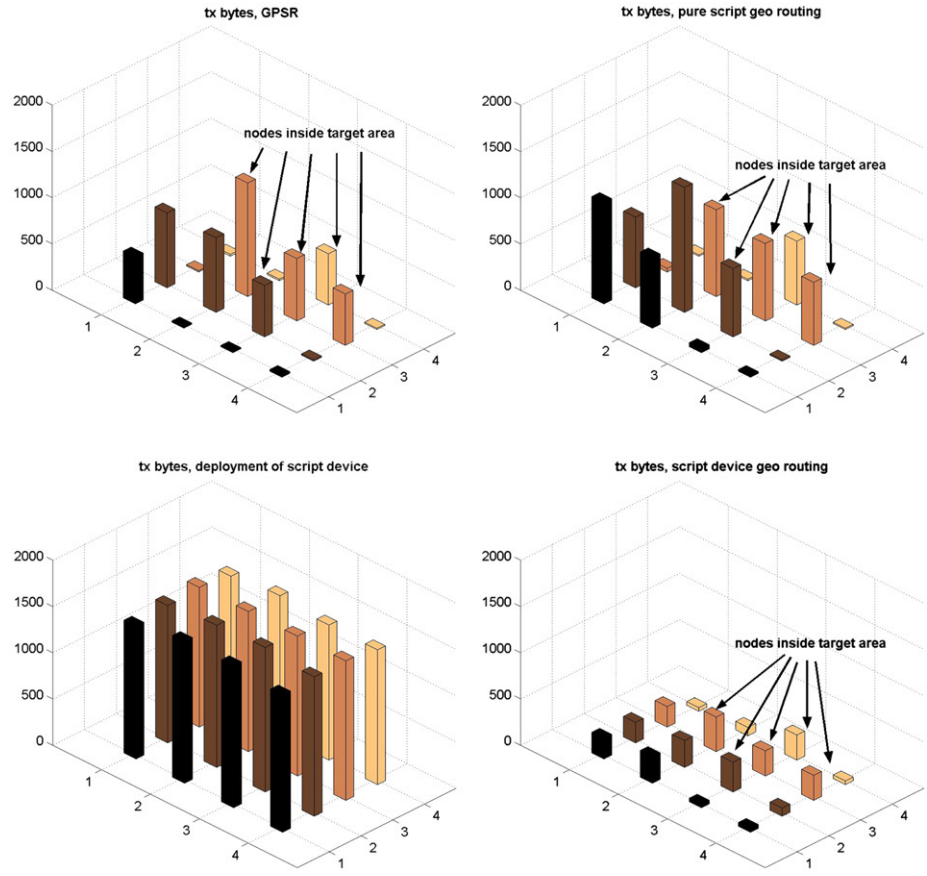

Fig. 7. Transmitted bytes per node for the 3 application versions.

delay measurements, we measured the total execution time (i.e., from invocation of initial script till replies from all the populated scripts arrived back to the user node) for all three versions.

We wanted to provide more information than just plain mean delay numbers (over 10 experiments) so we also calculated the breakdown of the total delay into communication and computation parts. In order to derive the breakdown, we measured the delay in three different processor speeds. Knowing the relative speedup each speed setting offers (through measurement of purely computational scripts), we could construct an over-determined system of equations (2 unknowns: communication part, computation part, and 3 equations: corresponding to the 3 different measurements for 3 different speeds) and using least-square minimization compute the unknowns for each version of the application.

Fig. 8 presents the final results revealing that the GPSR and script device versions are considerably faster that the pure script version and the deployment of the script device.

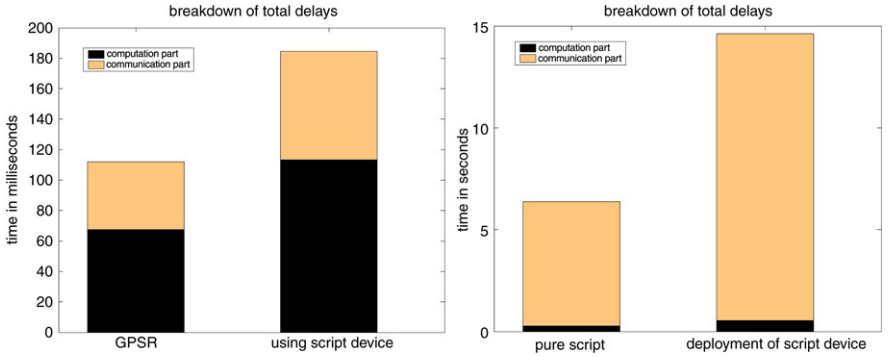

Fig. 8. Total delay breakdown for the 3 application versions.

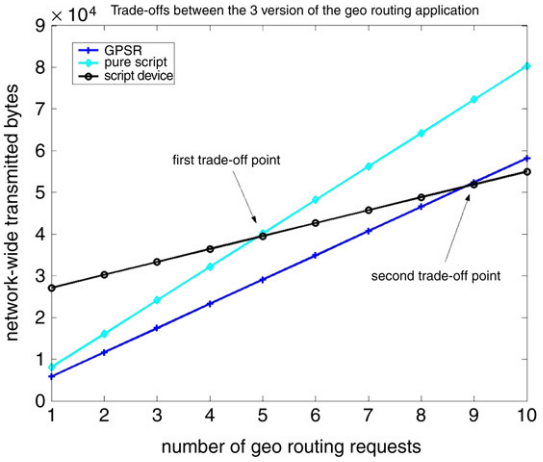

Fig. 9. Trade-offs for the 3 application versions.

Furthermore, the bulk of the delay in the last two cases is attributed to the communication part.

Finally, we wish to examine trade-offs between the three different versions. The trade-off quantity we check is total transmitted bytes, a major contributor to energy consumption. More specifically, how do the total transmitted bytes for each version increase as the number of routing requests increase? Fig. 9 presents a concise description of the trade-offs. As expected, GPSR always performs better than the pure script approach. That is, if we have the GPSR service in SensorWare we should use it. Do not forget though that this service might not be available due to memory restrictions (i.e., we choose not include this and other high-level services to decrease the SensorWare footprint). Examining the performance of the script device reveals the transmitted bytes related trade-offs. The script device implementation with its high initial overhead is not beneficial for a few geographical routing requests. But eventually the script device performs better than the pure script

version (for 5 requests or more) and even better than the GPSR version (for 9 requests or more).

Note that the role of autonomously mobile code (as opposed to globally pre-deployed code or pre-deployed in a statically defined area) is instrumental in achieving these trade-offs. Without this ability, versions 1 and 2 of the application would transmit more bytes network-wide, and would result in the extinction of at least the first trade-off point (since the pure script and script device implementations would be identical).

6. Related work

SensorWare falls under the family of active sensor frameworks. Its closest relatives in the traditional networks realm are Mobile Agent frameworks. Other active networking frameworks exhibit similarities, such as the scripting abstraction. In this section we only consider work that tries to make WASNs programmable using active sensor concepts. Therefore, general mobile-agent and active-network platforms are not discussed, nor any distributed database systems for WASNs are presented. The interested reader can refer to [6] for a comprehensive comparison of SensorWare with mobile agent platforms, as well as with an active networking framework called PLAN [13].

An active sensor framework for WASNs was developed in Berkeley under the name Maté. Maté [25] is a tiny virtual machine build on top of TinyOS [15]. TinyOS is an operating system, designed specifically for the Berkeley-designed family of sensor nodes, generically named "motes" [14,15]. Maté is based on a virtual machine (VM) for the motes. The VM supports a very simple, assembly-like language, to be used for all needs of mote-tasking. Programs (called capsules) written on the VM language can be injected to any node and perform a task. Furthermore the capsules have the ability to self-transfer themselves by using special language commands. This model seems extremely like our own in SensorWare. Differences appear though when one looks thoroughly into each platform's implementation.

Maté, like its substrate TinyOS, was built with a specific platform in mind: the extremely resource-limited mote. The main restriction for the developer of mote-targeted frameworks (such as an OS or a VM) is memory. The newest version of a mote called mica offers 128 KB of program memory and 4 KB of RAM. An older version called rene2 has 16 KB of program memory and 1 KB of RAM. Maté, with an ingenious architecture, supports both platforms. Being so memory constrained, Maté has to sacrifice some features that would make programming easier and more efficient. First, a stack-based architecture with an ultra-compact instruction set is adopted which is reminiscent of a low-level assembly language or the byte code of the Java VM. If non-trivial tasks are considered, this low-level run-time abstraction usually results in large code transfers. Second, the event handlers are essentially stateless in Maté, forcing their frequent updates for non-trivial tasks. This imposes an overhead in energy consumption and execution time. Third, because there is only one context (i.e., handler) per event (e.g., clock tick, reception of packet) multiple applications cannot run concurrently in one mote.

Another interesting piece of work is the Pushpin [26] programming framework. This is essentially the realization of the "paintable computer" [9] programming model developed

in MIT. Computation is performed in nodes (called "pushpins") that are more resource-constrained even than motes. There is no abstraction layer. Pieces of native code, called process fragments (pfrags), are transferred and executed in the pushpins. The operating system is responsible for the execution of pfrags, for provision of shared memory space for inter-pfrag communication, and for a limited number of system calls (e.g., for migration). This is a desirable solution for very resource-limited nodes but the cost is absence of fairness and resource management, limited code safety, and a poor set of services to help the programmer.

SensorWare cannot fit in the restricted memory of a mote or a pushpin. SensorWare targets richer platforms that we believe are going to be the mainstream in sensor node design in the immediate future. Platforms such as Rockwell's WINS node [32], Crossbow's/Intel's Stargate node [33], and ISI's PASTA node [30], have 1 MB–32 MB of Flash memory and 128 KB–64 MB of RAM. These nodes are already used in applications such as hierarchical target tracking [23] and mobile infrastructures for WASNs [20]. Having the luxury of more memory, SensorWare supports easy programming with a high-level scripting language and versatile run-time abstraction, as well as multi-tasking of a node so that multiple applications can concurrently execute in a WASN.

Another recent research effort is Impala from Princeton [27]. Impala is a non-VM middleware for application update on sensor networks. Applications are transferred in native code and linked dynamically in the target node without affecting existing applications. A node can host multiple pieces of code to perform a single task and a state machine is used to toggle between them. Applications are built as five event handlers responding to five fixed types of events. There is no way for an application to define its distribution, thus distribution of code is global. Impala, with its native code format and global network deployment is suitable for long-lived network-wide protocols. For instance, a widely-used routing protocol could be updated with Impala. On the other hand, short-lived neighborhood-oriented distributed applications would benefit from SensorWare's approach.

Researchers from Rutgers have developed a mobile code platform for embedded systems called Smart Messages (SM) [2]. Based on SM the same researchers suggest a programmable sensor network framework, called Spatial Programming (SP) [17]. Smart Messages are entities that carry code, data, and execution state. The code is written in Java, supporting a few extra commands relevant to the SM environment. The SM environment also supports the abstraction of tags. Tags are essentially SM-persistent storage and are used as universal names. To create Spatial Programming, the SM platform is augmented with a way to refer to nodes by spatial and arbitrary content properties of the node. Essentially SP is more a resource-based routing scheme than a programming model. The examples developed by the researchers to illustrate their framework present centralized applications (executing only at one node) that access resources remotely, much like RPC calls. These examples do not answer crucial questions: How is the code distributed in the network? How is collaborative operation between agents facilitated by the programming model?

Finally, Kumar [22] aspire to generalize and facilitate the data fusion process (termed "aggregation" by other researchers) by providing a framework called DFuse. The framework consists of an API to define arbitrary fusion processing and an algorithm for automatic fusion point placement and relocation. The API allows the fusion application to be specified as a directed dataflow graph along with the definition of the fusion

functions. The API hides many programming details (common to fusion applications) such as buffer management, time-stamping, and exception mechanism for error control. Furthermore, using the automatic placement algorithm considerably eases the deployment of such an application. The algorithm decides where the fusion points should be placed and periodically re-evaluates the placement. DFuse seems successful because it restricts itself to a certain type of applications without making overstatements on its general application. Restrictions on the dataflow graphs (i.e., the sources and sinks of the fusion computation should be fixed) limit the type of applications that can benefit from DFuse but the framework presents an interesting combination of the database and active sensor models. The arbitrary definition of fusion algorithms brings an element of the imperative active sensor model, while the definition of dataflow graphs and the automatic placement of fusion points bring an element of the declarative database model.

7. Conclusions

In this paper we present our framework for dynamic and efficient WASN programmability. Since one face of efficiency is the distributed algorithm chosen to tackle a particular problem, we wanted to offer rich algorithm expressiveness. Therefore, we opted for an active sensor model. Within active sensor frameworks the efficiency comes from the choice of abstraction and the ability of autonomous mobile code. Since there is not a single right choice of abstraction (it highly depends on the application's characteristics and usage) we adopt a scheme that can alter the run-time abstraction by dynamically installing new services. We explain the framework's architecture and present code examples. Through our implementation we are able to measure the time and memory overheads that we are paying for programmability and most importantly highlight the trade-offs that the choice of abstraction involves. Indeed, we have shown quantitatively the trade-offs between three versions of a geographical routing application as the application's usage increases. SensorWare makes WASN platforms open to transient users with dynamic needs. This fact, apart from giving an important flexibility advantage to deployed systems, greatly facilitates researchers to cross the simulation boundary and start testing their algorithms/protocols in real platforms.

Acknowledgements

This work was done while the first author was a Ph.D. student at UCLA. The work was partially supported by DARPA SensIT program, by ONR AINS program, and by the NSF-funded Center for Embedded Networked Sensing. Early versions of this work appear at [5] and [6]. SensorWare code is freely available through the sourceforge.net web site under the project name "sensorware".

References

[1] P. Bonnet, J. Gehrke, P. Seshadri, Querying the physical world, IEEE Personal Communications (2000).
[2] C. Borcea, D. Iyer, P. Kang, A. Saxena, L. Iftode, Cooperative computing for distributed embedded systems, in: Proceedings of the 22nd International Conference on Distributed Computing Systems, ICDCS, July 2002.

[3] A. Boulis, Programmable sensor networks: Framework and applications, Doctoral Dissertation, UCLA EE Department, 2003.
[4] A. Boulis, S. Ganeriwal, M.B. Srivastava, Aggregation in sensor networks: a energy-accuracy trade-off, in: Sensor Network Protocols and Applications, Elsevier Science Ad Hoc Networks 1 (2–3) (2003) 317–331. (special issue).
[5] A. Boulis, C.C. Han, M.B. Srivastava, Design and implementation of a framework for efficient and programmable sensor networks, in: The First International Conference on Mobile Systems, Applications, and Services, MobiSys, 2003.
[6] A. Boulis, M.B. Srivastava, A framework for efficient and programmable sensor networks, in: Proceedings of OpenArch 2002, New York, NY, June 2002.
[7] A. Boulis, M.B. Srivastava, Node-level energy management for sensor networks in the presence of multiple applications, in: The first IEEE Annual Conference on Pervasive Computing and Communications, PerCom 2003, Dallas-Fort Worth, TX, 23–26 March, 2003.
[8] A. Boulis, Programming Sensor Networks using Mobile Agents, in: The 6th International Conference on Mobile Data Management, MDM'05, Ayia Napa, Cyprus, 9–13 May, 2005.
[9] W. Butera, Programming a Paintable Computer, Doctoral Dissertation, MIT Media Laboratory, 2002.
[10] eCos: Embedded Configurable Operating System. http://sources.redhat.com/ecos/.
[11] Familiar Project. http://familiar.handhelds.org.
[12] J. Heidemann, F. Silva, C. Intanagonwiwat, R. Govidan, D. Estrin, D. Ganesan, Building efficient wireless sensor networks with low-level naming, in: Proceedings of Symposium of Operating Systems Principles, October 2001.
[13] M. Hicks, P. Kakkar, J. Moore, C. Gunter, S. Nettles, PLAN: A packet language for active networks, in: Proceedings of the International Conference on Functional Programming, ICFP '98, 1998.
[14] J. Hill, D. Culler, A wireless embedded sensor architecture for system-level optimization, Intel Research IRB-TR-02-00N, 2002.
[15] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, System architecture directions for networked sensors, in: Proceedings of ASPLOS-IX, Cambridge, MA, USA, November 2000.
[16] Honeywell HMR-2300 Magnetometer. http://www.ssec.honeywell.com.
[17] L. Iftode, C. Borcea, A. Kochut, C. Intanagonwiwat, U. Kremer, Programming computers embedded in the physical world, in: Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems, FTDCS 2003, May 2003.
[18] C. Intanagonwiwat, R. Govindan, D. Estrin, Directed diffusion: A scalable and robust communication paradigm for sensor networks, MobiCOM '00, Boston, MA, August 2000.
[19] iPAQ 3950. http://thenew.hp.com/.
[20] A. Kansal, A. Somasundara, D. Jea, M.B. Srivastava, D. Estrin, Intelligent Fluid Infrastructure for Embedded Networks, in: The 2nd International Conference on Mobile Systems Applications and Services, MobiSys, 2004 (in press).
[21] B. Karp, H.T. Kung, GPSR: Greedy perimeter stateless routing for wireless networks, in: Proc. of MobiCom, August 2000, pp. 243–254.
[22] R. Kumar, DFuse: A framework for distributed data fusion, in: Proc of ACM SenSys 2003, Los Angeles, CA, 2003.
[23] R. Kumar, V. Tsiatsis, M.B. Srivastava, Computation hierarchy for in-network processing, in: Proc. of the 2nd ACM Conf. on Wireless Sensor Networks and Applications, WSNA '03, San Diego, CA, USA, 2003, pp. 68–77.
[24] J. Labrosse, MicroC/OS-II: The Real Time Kernel, CMP Books, November 1998.
[25] P. Levis, D. Culler, Maté: A tiny virtual machine for sensor networks, in: Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X, 5–9 October 2002.
[26] J. Lifton, D. Seetharam, M. Broxton, J.A. Paradiso, Pushpin computing system overview: A platform for distributed, embedded, ubiquitous sensor networks, in: Proc. of the 1st International Conf. on Pervasive Computing, 26–28 August 2002.
[27] T. Liu, M. Martonosi, Impala: A middleware system for managing autonomic, parallel sensor systems, in: Proceedings of the ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming, PPoPP'03.

[28] S.R. Madden, R. Szewczyk, M.J. Franklin, D. Culler, Supporting aggregate queries over ad-hoc wireless sensor networks, in: Workshop on Mobile Computing and Systems Applications, 2002.
[29] J.K. Ousterhout, Tcl and the Tk toolkit, Addison-Wesley, 1994.
[30] PASTA node. http://pasta.east.isi.edu/.
[31] N. Reijers, K. Langendoen, Efficient code distribution in wireless sensor networks, in: Second ACM Int. Workshop on Wireless Sensor Networks and Applications, WSNA '03, San Diego, CA, USA, September 2003.
[32] Rockwell WINS node. http://wins.rsc.rockwell.com/.
[33] Stargate node. http://www.xbow.com/Products/productsdetails.aspx?sid=85.
[34] D. Tennenhouse, Proactive computing, Communications of the ACM 43 (5) (2000) 43–50.
[35] Wavelan card. http://www.orinocowireless.com.

Athanassios Boulis is a researcher in the Network and Pervasive Computing (NPC) research program. He joined NICTA in September 2004 after receiving a Ph.D. from the EE dept. at University of California, Los Angeles (UCLA), doing research in Wireless Sensor Networks. His research interests include programming models for networked embedded systems as well as distributed algorithm methodologies that promote robustness and expose energy/fidelity/delay trade-offs for such systems.