Design and Implementation of a Framework for Efficient and Programmable Sensor Networks

Abstract – **Wireless ad hoc sensor networks have emerged as one of the key growth areas for wireless networking and computing technologies. So far these networks/systems have been designed with static and custom architectures for specific tasks, thus providing inflexible operation and interaction capabilities. Our vision is to create sensor networks that are open to multiple transient users with dynamic needs. Working towards this vision, we propose a framework to define and support 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. The replication/migration of such scripts in several sensor nodes allows the dynamic deployment of distributed algorithms into the network. Our framework, SensorWare, defines, creates, dynamically deploys, and supports such scripts. Our implementation of SensorWare occupies less than 180Kbytes of code memory and thus easily fits into several sensor node platforms. Extensive delay measurements on our iPAQ-based prototype sensor node platform reveal the small overhead of SensorWare to the algorithms (less than 0.3msec in most high-level operations). In return the programmer of the sensor network receives compactness of code, abstraction services for all of the node's modules, and in-built multi-user support. SensorWare with its features apart from making dynamic programming possible it also makes it easy and efficient without restricting the expressiveness of the algorithms.**

I. 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 been mostly concerned with exploring applications such as target tracking and distributed estimation, investigating new routing and access control protocols, proposing new energy-saving algorithmic techniques for these systems, and developing hardware prototypes of sensor nodes.

Little concern has been given on 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) 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. However, it is widely accepted that WASNs will have long-deployment cycles and serve multiple transient users with dynamic needs. These two features clearly point in the direction of dynamic WASN programming.

What kind of dynamic programmability do we want for WASNs? 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) 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 a distributed database system (e.g., [1]). Multiple users can inject database-like queries to be autonomously distributed into the network. The WASN is viewed as a distributed database and the query's task is to retrieve the needed information by finding the right nodes and possibly aggregate the data as they are routed back to the user. This approach ignores though the fact that information is not always resident in nodes but sometimes has to be retrieved by *custom collaboration* among a changing set of nodes (e.g., target tracking). Thus even though the database model is programming the network in the desirable way, it is not expressive enough to implement any distributed algorithm.

The other approach to WASN programmability that is used by our framework, and is gaining momentum lately,

is the "active sensor" approach. This term was used in [19], to describe a family of frameworks that 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 are reacting 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 installing a virtual machine or a high-level script interpreter at each node. For example, single instructions of the scripts (or bytecodes) can send packets, or read data from the sensing device. Moreover, the scripts (or bytecodes) are made mobile through special instructions, so nodes can autonomously task their peers.

The difficulty in designing an active sensor framework is 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. Our proposal of such a framework, called SensorWare, employs lightweight and mobile control scripts that are autonomously populated in sensor nodes after a triggering user injection. The sensor node abstraction was made in such a way so that multi-user accessibility is given to all of the node's modules (e.g., radio, sensing devices) while also creating other services (e.g., real-time timers). Considerable attention was given to the portability and expandability of the framework by allowing the definition of new modules. By choosing the right level of abstraction the scripts are compacted to 10s-100s of bytes. For the non-trivial application examined in section V.A, the SensorWare script is smaller than the code of other frameworks with comparable capabilities in algorithm expressiveness (e.g. other active sensors scripts, binary images).

Our implementation and porting of SensorWare in several sensor node platforms shows that the size of the framework is small enough (<180Kbytes) to fit in most current sensor node designs. Moreover, extensive measurements in our prototype iPAQ-based sensor node platform reveal the delay and energy overheads of SensorWare. Every SensorWare script command has a delay less than 0.3msec showing the limits of real-time operation. Note that the script commands have a high-level of abstraction (i.e., each command performs multiple low-level operations). Experiments with both compiled and interpreted versions of the scripts are conducted in order to explore the energy trade-off space between different representations of an algorithm.

Section II discusses in depth the nature of WASNs, approaches to WASN programmability, and the general idea of our approach. Section III presents related work. Section IV presents SensorWare's architecture. Section V illustrates how is SensorWare ported to a platform and explains a moderately large script solving a real problem. Section VI presents our current implementation and the measurements we acquired through it. Finally, section VII concludes the paper.

II. MOTIVATION AND BACKGROUND

A. *Wireless Ad hoc Sensor Networks*

Figure 1 shows an example of a WASN, highlighting its main characteristics. An ad hoc network of miniature, resource-limited, static, wireless, sensor nodes is being used to monitor a dynamic physical environment. The use of low power communication and the need for diversity in sensing necessitates a multi-hop, distributed architecture [22]. The computation capabilities at the nodes can be leveraged for event detection via data fusion and collaborative signal processing among nearby nodes, so that higher bandwidth raw sensor data does not need to be sent to the users. Typically a user queries the network (consider the term "query" in the broad sense, not just database query), the query triggers some reaction from the network, and as the result of this reaction 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 or a distributed estimation algorithm. Furthermore, there are multiple users who are transiently connected to the network, each having different needs in requested information. The WASN is there to accommodate all or most of their needs.

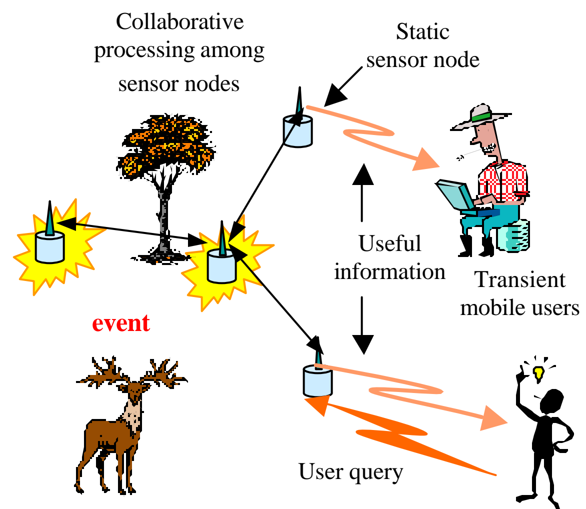


Figure 1: Wireless Ad-hoc Sensor Network

These systems are quite different from traditional networks. First, they have severe energy, computation, storage, and bandwidth constraints. Second, their overall usage scenario and the implications that this brings to the traffic and the interaction with the users 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 David Tennenhouse [27]. We also adopt the term "proactive" throughout the paper 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. Given the nature of SNs, one can coarsely define two classes of problems in their design. First, the application-specific problem: How does one find the most efficient distributed algorithm for a particular problem? Second, the generic 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?

For the first class of problems (i.e., finding efficient algorithms for particular applications), there are many research efforts in a variety of application problems (e.g., target tracking, sensor reading aggregation). In this paper we will not expand into any particular application problem. We only note, that in general, localized distributed algorithms (i.e., distributed algorithms that act locally, using only local information) are particularly efficient in most WASN problems as they achieve small and well-distributed energy consumption, thus prolonging the network lifetime.

The second class of problems (i.e., what is the right framework to express and dynamically deploy distributed algorithms for WASNs) is the focus in this paper. We describe our proposal of such a framework, called SensorWare. SensorWare provides a language model powerful enough to express the most efficient distributed algorithms while at the same time hiding unnecessary low-level details from the application programmer and providing a way to share the resources of a node among many applications and users that might concurrently use the WASN. The language model is developed after examining what are the properties of efficient algorithms for SN (e.g., localized distributed algorithms), and in conjunction with developing our own applications on real sensor networks [3].

Equally important is the role of SensorWare in the dynamic deployment of the distributed algorithms into the network. As sensor nodes are memory-constrained, they cannot store every possible application in their local memory. Thus, a way of dynamically deploying a new application is needed. Usually this means that a distributed algorithm has to be incorporated in several sensor nodes, which in turn means that these sensor nodes have to be dynamically programmed. A user-friendly and energy-efficient way of programming the nodes keeps the user out-of-the-loop most of the time by allowing sensor nodes to program their peers. By doing so, the user does not have to worry about the specifics of the distributed algorithm (because the information on how the algorithm unfolds lies within the algorithm), and the nodes save communication energy (because they interact with their immediate neighbors and not with the user node through multi-hop routes). The programming model of SensorWare is designed in such a way, so as to facilitate the user-friendly and energy-efficient dynamic deployment of an algorithm. The user "injects" the query/program into the network, and the query *autonomously* unfolds the distributed algorithm into the nodes that should be affected.

B. Approaches to WASN programmability

As discussed in the introduction, one of the approaches currently under investigation is a distributed database model. A good example of this approach is the work done at Cornell [1]. A similar scheme called DataSpace focusing on location addressing has also been developed in Rutgers [14]. 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. Although this approach takes into account the distributed nature of the system and works well in several scenarios, it does not take into account the proactive nature of the system. The user is the central place of control and most data flows to/from the user. This property can prove inefficient in applications such as target tracking, where it is better for nodes to form clusters around the target, collaboratively compute the target's location and just send the location information back to the user. Clearly a more flexible way of programming the sensor network is needed to enable this kind of behavior.

C. SensorWare

The SensorWare architecture is based on a scriptable lightweight run-time environment, optimized for sensor nodes that have limited energy and memory. This environment securely hosts one or more simple, compact, and platform-independent sensor-node control scripts. The sensing, communication, and signal-processing resources

of a node are exposed to the control scripts that orchestrate the dataflow to assemble custom protocol and signal processing stacks. SensorWare has to also promote the creation of distributed proactive algorithms based on the scripting language described above. For this reason the scripts are made mobile using special language commands and directives. A script can 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.

A usage scenario can be like the following: A user sends a query to the sensor network. The query is a script, a state machine in its simplest form, which is injected to one or more sensor nodes. The script will describe among other things how it is going to populate itself to other nodes. The process of population can continue depending on events and the current state. For example as the events of interest are moving to a different area, the scripts can move along with them, possibly trying to predict their next move. The populated scripts will collaborate among themselves in order to extract the information needed by the user, and when this information is acquired it is sent back to the user.

III. 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 [2] for a comprehensive comparison of SensorWare with mobile agent platforms, as well as with an active networking framework called PLAN [10].

An active sensor framework for WASNs is currently being developed in Berkeley under the name Maté. Maté [19] is a tiny virtual machine build on top of TinyOS [12]. TinyOS is an operating system, designed specifically for the Berkeley-designed family of sensor nodes, generically named "motes" [11][12]. Maté's goal is to make a WASN made of motes dynamically programmable in an efficient manner. This includes the capability to dynamically instruct a mote to execute any program, and expressing this program in a concise way. They achieve this by building a virtual machine (VM) for the motes. The virtual machine 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. Indeed, Maté shares the same goals as SensorWare as well as the same basic principles to achieve these goals. 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 128Kbytes of program memory and 4Kbytes of RAM. An older version called rene2 has 16Kbytes of program memory and 1Kbyte 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 (all instructions are 1 byte) is adopted which is reminiscent of a low-level assembly language or the byte code of the Java VM. This kind of model makes programming of even medium-sized tasks difficult. Furthermore, due to the ultra-compact instruction set, many 1-byte instructions are needed to express a medium complexity algorithm, which in turn leads to large programs, compared to a higher-level, more abstracted scripting language. The size of programs is important since the code is transmitted/received using the radios of the nodes spending energy for every transmitted/received bit. Second, the behavior of a program when radio packets are received is rather rigid. A handler to process such events is essentially stateless in Maté. Thus, if a new pattern of packet processing is needed, a new handler has to be transferred through the network. 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.

SensorWare cannot fit in the restricted memory of a mote. SensorWare targets richer platforms that we believe are going to be the mainstream in sensor node design in the immediate future. Such platforms (e.g., [24]) include a 1Mbyte of program memory and 128Kbytes of RAM. Having the luxury of more memory, SensorWare supports easy programming with a high-level scripting language, as well as concurrent multi-tasking of a node so that multiple applications can concurrently execute in a WASN. The programming model and properties of SensorWare are extensively discussed in section IV.

Particularly instructive is to study the relationship between SensorWare's mobile scripting approach and the mobile code approach in Penn State's Reactive Sensor Network [23] (RSN) project under DARPA's SenseIT program [25]. RSN's focus is on providing an architecture whereby sensor nodes can: (i) download executables and DLLs, identified by URLs, from repositories or their cache, (ii) execute the program at the local node using input data which itself may be remotely located and identified by a URL, and (iii) write the data to a possibly remote URL. The RSN model is in essence Java's applet

model generalized to arbitrary executables and data, and combined with a lookup service. The focus of RSN is quite different from SensorWare. Differences include: (i) RSN provides a general lookup and download service, (ii) RSN does not seek to provide a scripting environment with an associated sensor node resource model for use by scripts, and (iii) RSN's notion of mobility is download oriented, as opposed to SensorWare's approach of a script which can autonomously spawn scripts to remote nodes. RSN views sensor nodes as network switches with dynamically adaptable protocols, trying to directly map the motivation and methods of classical active networks into sensor networks. Unfortunately such an approach does not address the basic problems of sensor networks. Although one might be able to construct some distributed applications using the above scheme, by no means the creation and diffusion of distributed proactive applications into the network is supported by its architecture.

Finally, extremely relevant is the work that is being conducted in University of Delaware by Jaikaeo et al. [16] called SQTL (Sensor Querying and Tasking Language). Having the same goals as our research, but starting from a different point (database-like queries), the researchers end up with the same basic solution as SensorWare, namely a tasking language for sensor networks. To lively demonstrate the relevance to our work we are quoting an excerpt from [16]. *"We model a sensor network as a set of collaborating nodes that carry out querying and tasking programmed in SQTL. A frontend node injects a message, that encapsulates an SQTL program, into a sensor node and starts a diffusion computation. A sensor node may diffuse the encapsulated SQTL program to other nodes as dictated by its logic and collaboratively perform the specified querying or tasking activity."*

SQTL fits in a more general architecture for sensor networks called SINA (Sensor Information Networking Architecture) [26]. SINA uses both SQL-like queries as well as SQTL programs. Some of its main features include: 1) hierarchical clustering, 2) attribute-based naming, 3) a spreadsheet paradigm for organizing sensor data in the nodes. SQL-like queries use these three features to execute simple querying and monitoring tasks. When a more advanced operation is needed though, SQTL plays the essential role by programming (or "tasking" as the researchers from Delaware call it) the sensor nodes and allowing proactive population of the program. In SINA, SQTL is used as an enhancement of simple SQL-like queries. The framework is there mainly to support the queries not the mobile scripts. As a consequence, SQTL scripts do not have all the provisions that SensorWare scripts have. The most important of them are: 1) Rich sensor-node-related APIs (e.g. for networking, sensing). 2) Diverse rules for mobility. A SQTL script can only specify the nodes to be populated. SensorWare first checks if the script is already in the remote node and offers a multitude of possibilities depending on how many instances of the script are already running in the remote node. 3) Code modularity in order to share functionality and avoid redundant code transfers 4) Support for multi-user scripts. 5) Resource management in the presence of multiple scripts running in the node.

IV. Architecture

First, we show SensorWare's place inside the overall sensor node's architecture (Figure 2). The architecture of a sensor node can be viewed in layers. The lower layers are the raw hardware and the hardware abstraction layer (i.e., the device drivers). An operating system (OS) is 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 around the network. Static applications and services coexist with mobile scripts. They can use some of the functionality of SensorWare as well as standard functions and services of the OS. These applications can be solutions to generic sensor node problems (e.g., location discovery), and can be distributed but not mobile. They will be part of the node's firmware.

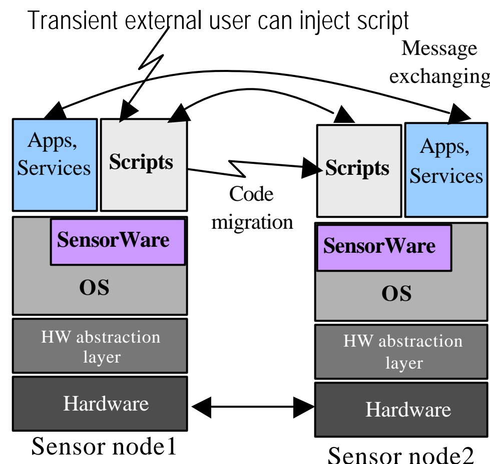


Figure 2: The general sensor node architecture

Two things comprise SensorWare: 1) the language, and 2) the supporting run-time environment. The next two subsections describe each of the parts in detail. A third subsection discusses issues of portability and expandability, and presents the final SensorWare code structure.

A. The language

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

First, a scripting language needs proper functions/commands to be defined and implemented in order to use them as building blocks (i.e., these will be the basic commands of the scripts). Each of these commands will abstract a specific task of the sensor node, such as communication with other nodes, or acquisition of sensing data. These commands can also introduce needed functionality like moving a script to another node or filtering the sensing data through a filter implemented in native code. Second, a scripting language needs constructs in order to tie these building blocks together in control scripts. Some examples include: constructs for flow control, like 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 they combine several of the basic building blocks to make actual control scripts.

Figure 3 illustrates the different parts of the SensorWare language. Several of the basic commands/functions are grouped in theme-related APIs. We use the term API in a generic fashion, to denote a collection of theme-related functions that provide a programming interface to a resource or a service. As the figure hints, there is a question on what happens when we are dealing with different sensor node platforms that may support different/additional kinds of modules. Do we allow the set of APIs to be expandable? If so, who has the authority to name and define new commands? We will return to this topic with a solution in subsection C.

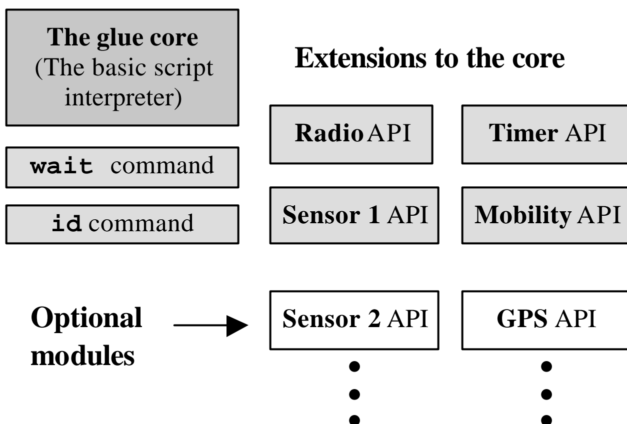


Figure 3: The language parts in SensorWare

As a glue core we can use the core from one of the scripting languages that are freely available, so we are not burdened with the task of building and verifying a core. One such scripting language, that is well suited for SensorWare's purposes, is Tcl [20], offering great modularity and portability. Thus, the Tcl core is used as the glue core in the SensorWare language. All the basic commands, such as `wait`, or the ones included in the APIs, are defined as new Tcl commands using the standard method that Tcl provides for that purpose.

The set of APIs is basically a way of easily exporting services and shared resources to the scripts. For example, the Timer API defines and sets/resets real time timers, while the Mobility API provides the basic functions to the scripts so they can transfer themselves around the network.

A.1 The general programing model

As discussed earlier, according to the proactive distributed model the scripts will look mostly like state machines that are influenced by external events. Such events include network messages from peers, sensing data, and expiration of timers. 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 (light) processing and possibly create some new events or/and alter the current state. Figure 4 illustrates SensorWare's programming model with an example.

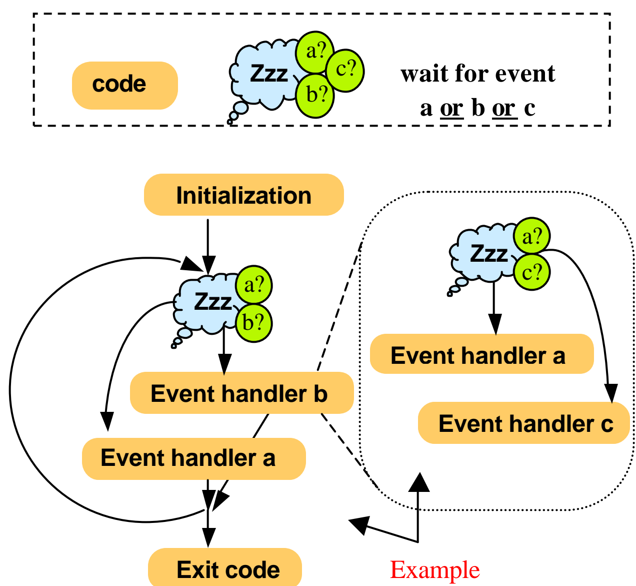


Figure 4: The programming model

The behavior described above is achieved through the `wait` command. Using this command, the programmer can define all the events that the script is waiting upon, at a given time. Examples of events that a script can wait upon are: i) reception of a message of a given format, ii) traversal of a threshold for a given sensing device reading, iii) filling of a buffer with sensing data of a given sampling rate, iv) expiration of several timers. 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.

B. The run-time environment

As important are the scripts in the SensorWare platform, equally important is the run-time environment that supports them. Figure 5 illustrates the basic tasks performed by the environment. We separate tasks into fixed and platform-specific. The fixed tasks are always included in a SensorWare implementation, while the platform-specific depend on the existence of specific modules and services in the node platform. Again, the problem of expandability and portability appears. Do we allow any developer to *arbitrarily* define and create any tasks, according to the specific needs of each platform? Subsection C addresses this question.

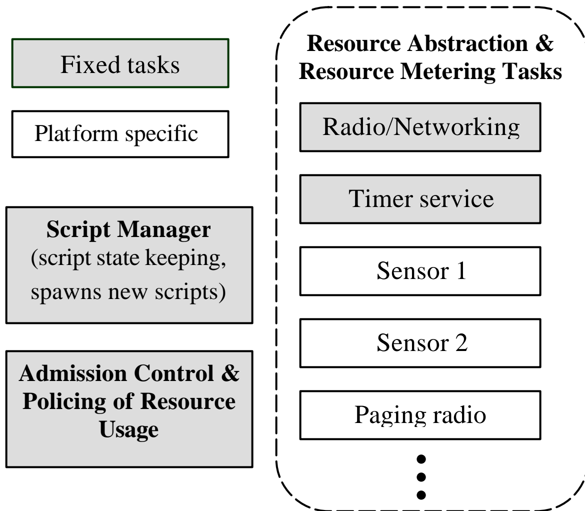
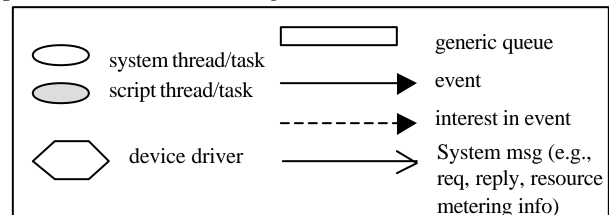


Figure 5: Tasks in the SensorWare run-time environment

The Script Manager is the task that accepts all requests for the spawning of new scripts. It forwards the request to the Admission Control task and upon receiving a positive reply, it *initiates a new thread/task running a script interpreter for the new script*. The Script Manager also keeps any script-related state such as script-data for as long as the script is active. Possible attacks, such as snooping or spoofing, are banned by the strict security model. The script manager also keeps a script-code cache in order to reduce code transmissions over the wireless channel. The Admission Control and Policing of Resource Usage task, as the name reveals, takes all the script admission decisions, makes sure that the scripts stay under their resource contract, and most importantly checks the overall energy consumption. If the overall consumption exhibits alarming characteristics (e.g., the current rate cannot support all scripts to completion) the task selectively terminates some scripts according to certain SensorWare policies. Resource management and the security model are not discussed in this paper. The interested reader can refer to [2].

The run-time environment also includes "Resource Abstraction and Resource Metering" tasks (sometimes referred to as "Resources Handling" tasks for brevity). Each task supports the commands of the corresponding APIs and manages a specific resource. There are two fixed tasks in this category since every platform is assumed to have at least one radio and a timer service. The “Radio” task manages the radio: i) it accepts requests from the scripts about the format of network messages that they expect, i) it accepts all network messages and dispenses them to the appropriate scripts according to their needs, and finally iii) measures the radio utilization for each script, a quantity that is needed by the “Admission Control & Policing of Resource Usage” task. The second fixed task, the "Timer service", accepts the various requests for timers by all the scripts and manages to service them using a real-time timer the embedded system provides. In essence the task provides many virtual timers relying on one timer provided by the system. According to platform capabilities a specific porting of SensorWare may run additional tasks For instance, a “Sensor Abstraction” task manages a sensing device. It accepts all requests for sensor data from all the scripts and decides on the optimal way to control the sensing device (e.g., setting the A/D sampling rate). It also measures the sensing device utilization for each script. Figure 6 depicts an abstracted view of SensorWare's run-time environment for an example platform with one sensing device.



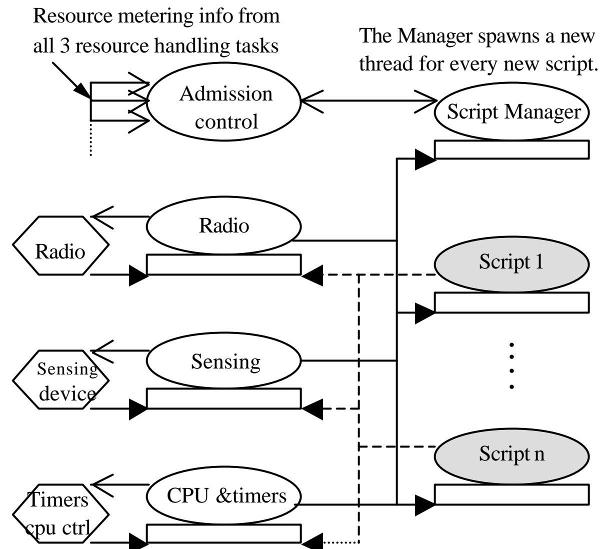


Figure 6: Abstracted view of SensorWare's run-time environment for an example platform

Most of the threads running are coupled with a generic queue. Each thread "pends" on its corresponding queue, until it receives a message in the queue. When a message

arrives it is promptly processed. Then the next message will be fetched, or if the queue is empty, the thread "pends" again on the queue. A queue associated with a script thread is receiving events (e.g., reception of network messages, sensing data, or expiration of timers). A queue associated with one of the resource handling tasks, receives events of one type (from the specific device driver that is connected to), as well as messages that declare interest in this event type. For instance, the Sensing resource-handling task is receiving sensing data from the device driver and interests on sensing data from the scripts. The Script Manager queue receives messages from the network that wish to spawn a new script. There are also system messages that are exchanged between the system threads (like the ones that provide the Admission Control thread with resource metering information, or the ones that control the device drivers).

C. *Portability and expandability of SensorWare*

In the previous subsections the problem of platform variability was revealed. Here we will present a solution for SensorWare's code structure. There are two kinds of platform variability: 1) capabilities variability (i.e. having different modules, such as sensing devices, GPS), 2) HW/SW variability (i.e. although the capabilities are the same we have different OS and/or specifics of hardware devices). We will explore solutions for each kind in two different subsections.

C.1 Capabilities variability

Different platforms may have different capabilities. For instance, imagine that one platform A has a radio and a magnetometer, while another platform B has two radios (a normal and a paging one) and a camera. How will we abstract the two platforms with the same framework? Since SensorWare's building blocks are the interface to the abstracted modules/services, we can allow an expandable API. Further, most modules/services will need a supporting task (as described in subsection B), so we can allow the definition and addition of arbitrary tasks in SensorWare's run-time environment. This kind of solution would create severe problems in the manageability of the code by different developers. SensorWare advocates a more modular and well-structured solution. SensorWare declares, defines, and support virtual devices (an idea triggered by Linux's virtual devices). Any module or service is represented as a virtual device. For example a radio, a sensing device, the timer service, a location discovery protocol are all view 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`, `createEventID`, and `disposeEventID`. `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 some parameters of the device, or if the device is an actuator perform an action). `CreateEventID` describes a specific event that this device can produce and gives this event a name/ID. The name can be used subsequently from the `wait` command to wait on this specific event. `DisposeEventID` just disposes that name. Additionally, if a device can produce events, a task is needed to accept create/disposeEventID 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 in a custom fashion by the developer. This is where the expandability stems from, while at the same time keeping a structured form.

C.2 HW/SW variability

Even though two platforms may have the same capabilities (i.e., the same modules/services), they may rely on different hardware and/or operating system. In order to facilitate the porting process it is desirable to clearly separate the OS and HW-specific code from the fixed code and the capabilities-definition code. To achieve this we need to identify the dependencies of the code to the OS and the hardware and create abstracted wrapper functions. The wrapper functions are actually defined in separate sections of the code (i.e., different .c files) so that the developer can easily identify the points of change for a porting procedure.

From the OS we need support to create and initiate threads/tasks, and support to define, post, and pend into mailboxes/queues. Thus we create wrapper functions for these operations. We also need low-level functions to access the hardware, thus we create wrapper functions around them (these functions will depend on the specific capabilities the platform supports). Figure 7 illustrates SensorWare's code structure.

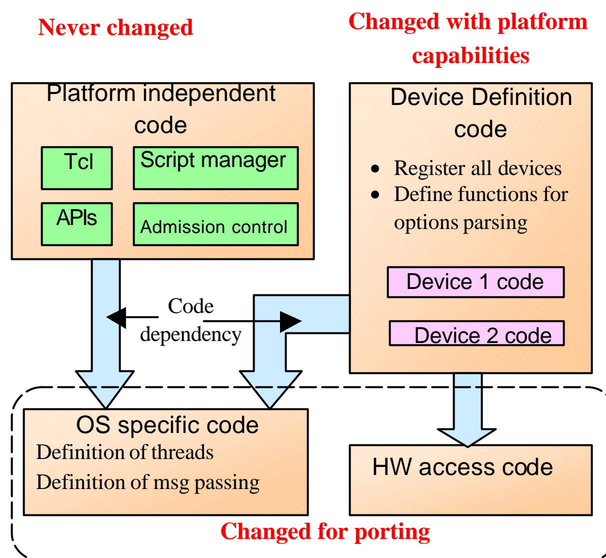


Figure 7:SensorWare code structure

V. Code Examples

In order to make SensorWare more concrete, we will present code examples and porting details in the next two subsections. The first one involves the creation of a specific application using the SensorWare script language. The second example, present details on how to port SensorWare in a specific platform. More specifically, we will show how to define new devices and how to connect the framework with the existing OS and hardware.

A. *Script example*

In this subsection we will present the code for the snapshot aggregation application with multiple (static) users support. The specific problem that we are solving is to find the global maximum among current sensor node readings and report it back to the user. Furthermore, multiple users may request this maximum while the algorithm is running (i.e., time to populate the script into the network, collect and aggregate data towards the user). The users are accommodated with the minimum traffic, without the need to launch a different application/script for each user. Finding the minimum, average, or any other aggregation function, among different kinds of sensor node readings or state, can be easily achieved by trivial modification in our script. More on aggregation applications in general can be found in [3] and [4].

Before proceeding with the script code, it is beneficial to describe the internal workings of two Sensorware commands, namely "`replicate`" and "`wait`". `Replicate` (possibly) transfers the script that it was called from, to other node(s). It does not blindly pack and transmit the code and state of the script like all other active sensor approaches currently do. `Replicate` first starts with a transmission of "intention to replicate" message, carrying the name of the script and the issuing user. If the same script already exits in the other node(s) `replicate`, according to options, may choose not to transfer the code, may choose to initiate a second script of the same type in the node, or if the script has multi-user support, send an "add user" message. By default, `replicate` will send the "intention to replicate" message to avoid unnecessary code transfers, and will spawn a second script only if the requesting user is different by the existing one. Furthermore, it is assumed by default that the parent of the script (i.e., the node that spawned the script to the current node) already has the code for the script, thus does not need an "intention to replicate" message. The arguments of the `replicate` command are:

replicate [-[f] [d] [p] [m] [rc] [rs] [ru]] [node_list]
[ ] means optional
f : forced replicate, no "intention to replicate" message sent
d: duplication of script at remote node irrespective of user
p: parent not assumed to have script in memory
m: script supports multi-users. Do not spawn new script in remote node, instead send "add user" message to existing script
rc: return nodes that code was transferred
rs: return nodes that spawned new script
ru: return nodes "add user" message was sent
by default option rsru is in effect.
node_list: nodes to replicate. Leaving this field empty implies a broadcast to neighbors. Parent is excluded unless p is chosen.

It is also useful 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 scrip code, the wait command sets the following predefined variables:

event_name : the name of the occurred event. It indicates the device that caused the event and the type of the event
event_data: data returned by the event
If the event is a packet reception the following are defined and set: ***msg_sender***, ***msg_body***

Listing 1 shows the actual SensorWare script. SensorWare commands and reserved words are in boldface. Variable names are in italics. Reserved variable name are in boldface and italics. Basic Tcl knowledge is needed to follow the script, although we do explain most of the code step by step. The example is sufficient to illustrate the programming style and the use of some of the most important commands, while solving a real problem.

```
set need_reply_from [ replicate -m]
set maxvalue [ query sensor value ]
if {$need_reply_from == ""} { send $parent $maxtemp; exit }
else { set return_reply_to $parent }
set first_time 1
while {1} {
      wait anyRadioPck  // "anyRadioPck" is a predefined eventID
      if { $msg_body ==add_user } {
            if { $first_time == 1 } {
                        send $parent $msg_body
                        set first_time 0
            }
            set return_reply_to "$return_reply_to $msg_sender"
      }else {
            set maxvalue [expr {($maxvalue< $msg_body) ? $maxvalue
: $ msg_body }]
            set n [lsearch  $need_reply_from $ msg_sender]
            set need_reply_from [lreplace $need_reply_from $n $n]
      }
      foreach node $return_reply_to{
            if { ($need_reply_from=="")||($need_reply_from==$node)} {
                        send $node $maxvalue
                        set n [lsearch  $return_reply_to $node]
                  set return_reply_to [lreplace $return_reply_to $n $n]
            }
      }
      if {$return_reply_to==""} {exit}
}
```

Listing 1: Multi-user aggregation code

The specific script keeps two important variables at each node: a list of nodes that replies are needed from, and

a list of nodes that replies are due. The first command tries to replicate the script to all the neighbors (except the parent), declaring that this is a multi-user script. The nodes that the script was spawned or an "add user" message was sent are returned and added to the need_reply_from variable. The second command reads the current value from the sensing device and sets the maxvalue variable with it. If there are no nodes to return a reply the script sends the maxvalue to the parent node and exits. Otherwise the parent node is added to the list return_reply_to and the big loop begins. Each time a packet is received we check if it is a data reply or an "add user" message and modify our lists and maxvalue accordingly. To graphically see how this algorithm works, refer to [4].

The script is its raw form is 882 bytes. If reserved words and variable names are compressed, the script becomes **277** bytes. If furthermore, we compress this intermediate form with gzip, we end up with **209** bytes. This is a compact description for this non-trivial algorithm. An equivalent SQTL script has a size in the order of 1000 bytes (based on the simpler algorithm of aggregation for a single user and without replication checking). Building the same algorithm in Maté was proven impossible due to its limited heap and stack sizes. There was not enough space to hold the need_reply_from and return_reply_to lists. Even with a larger memory space though, Maté's stack based architecture and lack of higher-level services results in code of many instructions even for simple tasks. As stated earlier, Maté's restrictions are a design choice, coming from the desire to support the restrictive underlying platform. Finally, C code is written for this algorithm, with external references to SensorWare functions. The compiled native code has a size of **764** bytes (without including the size of SensorWare functions called from within the native code).

B. *Porting SensorWare to a platform*

In this subsection we will present some of the issues while porting SensorWare to a platform. We consider our iPAQ-based prototype as the testbed. A full description of the platform can be found in section VI.A. Here it is sufficient to know that the node has one radio and one sensing device, and that the underlying OS is Linux.

First, we should add the proper capabilities to SensorWare by creating a virtual device for the sensing device (the radio has a virtual device by default). This means name and register the device by calling the function:

```
create_device(char* name, int (*query)(), int (*act)(), void* (*createEventID)(), int (*disposeEventID)(), void* (*task)() )
```

As it can be seen by the declaration of the create_device function we need to define the four functions to parse the arguments of the four standard interface commands, plus a function to be executed by the thread/ task of the device. Not going any further into the definition of these functions, we are sufficed to say that they are very similar to the radio device functions.

The next step is to define the OS-specific code. More precisely, have the ability to create threads and use mailboxes/queues. For the definition and creation of threads we use the pthreads (i.e., posix threads) provided by Linux. Even though mailboxes are available in Linux, we chose to construct our own structures using semaphores. Finally, the hardware-specific code is directly provided by the Linux's device drivers.

VI. IMPLEMENTATION

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? How much slower and consequently how much more energy-consuming is SensorWare compared to native code approaches? These questions are answered in the following subsections. We begin by a description of the implementation platform.

A. *Platform description*

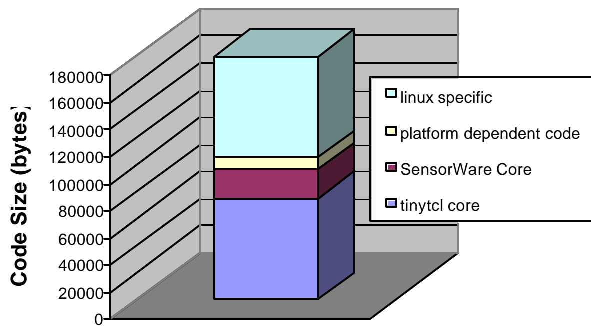
The prototype platform used in the implementation and evaluation of SensorWare was built around the iPAQ 3670 [15]. The iPAQ has an Intel StrongARM 1110 rev 8 32 bit RISC processor, running at 206Mhz. The flash memory size is 16Mbytes and the RAM memory size 64Mbytes. The OS installed is a familiar v0.5 Linux StrongARM port [9], kernel version 2.4.18-rmk3. The compiler used, is the gcc cross-compiler. A wavelan card [28] is used as the radio device and a Honeywell HMR-2300 Magnetometer [13] as the sensing device.

Figure 8: The implementation platform

SensorWare is also ported into the Rockwell WINS nodes [24] that also have a StrongARM processor, but only 1Mbyte of flash memory. Both eCos [6] and microC/OS-II [18] were used as operating systems for these nodes.

B. Memory size measurements

The first question to answer is how much size does the whole framework occupy. Figure 9 shows that the total size is 179Kbytes and it is consisted of 74Kbytes of Linux specific code (e.g., kernel, libraries), 74Kbytes of a stripped down Tcl core called tinyTcl, 22Kbytes of SensorWare code and 8Kbytes of platform dependent code (i.e., functions to access the hardware). The bottom part of the figure shows the breakdown of the SensorWare core part into smaller parts.



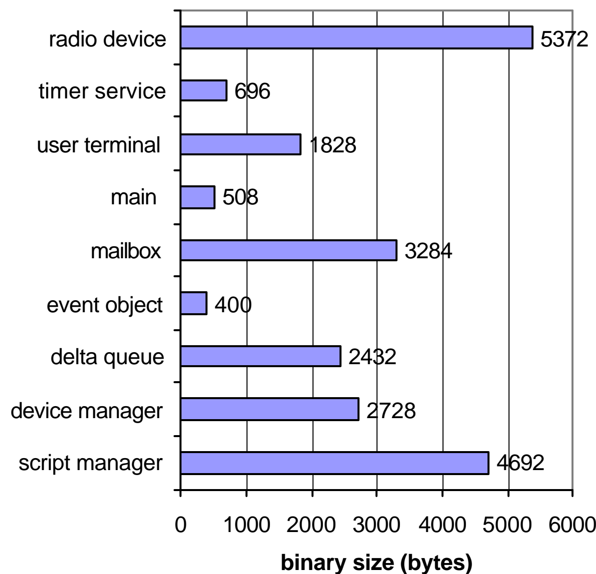


Figure 9: Code size breakdown

C. 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) and derived an average and standard deviation for the delay. Most commands exhibited negligible variance. All the commands, except the ones that used the radio and the one that spawned a 50byte script, have an execution time less than 0.3msec.

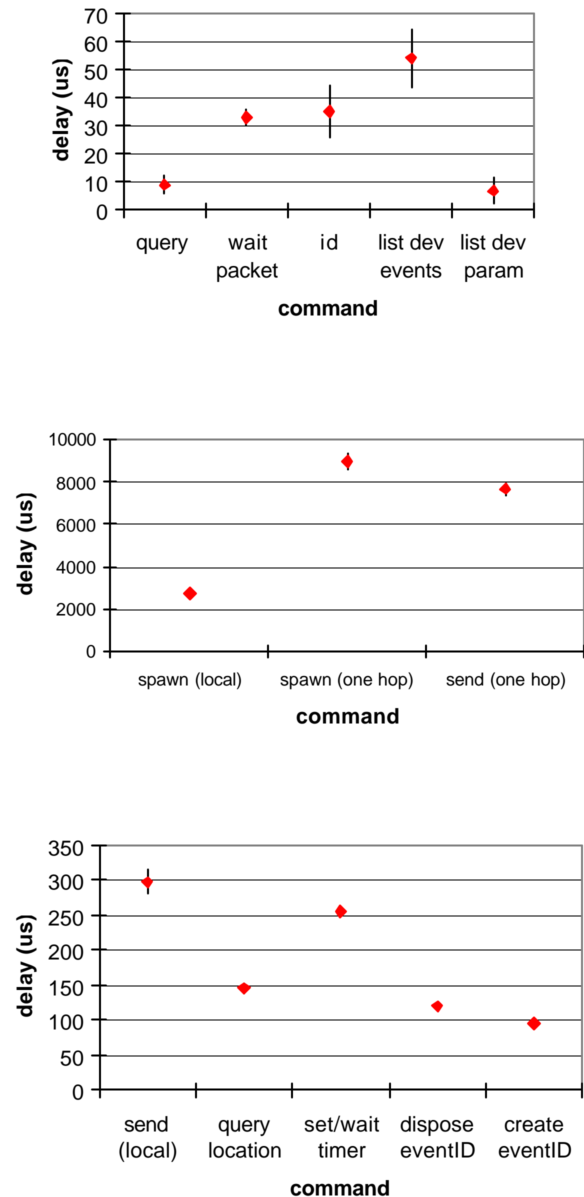


Figure 10: Execution times of SensorWare commands

The top graph of the figure 10, shows commands with less than 0.06msec delay. The last two commands that return some part of the device's state are internal to SensorWare and not exported for script use. The middle

graph shows the most time consuming commands. The first one spawns a 50 byte script locally. The other two commands use the radio to spawn a script in a neighboring node and send a message in a neighboring node. The delay for achieve these two operations is dominated by the radio transmission time. Note that the send command and some operation modes of the spawn command, do not wait for the whole operation to finish, instead they return as soon as they hand off the task to the radio device. In the graph, the total operation time is shown. The bottom graph of figure 10, shows yet another set of delays. Of particular interest is the set/wait timer delay. For this instance, we measure the delay to set a zero-valued timer and wait for its expiration. In essence we are measuring the overhead of real-time measurements in scripts. The overhead is 0.25msec with very small variation. Thus, we can internally subtract this number each time a script sets a timer, in order to measure the true desired time.

In order to acquire all delay measurements we used 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 1μsec.

D. Energy measurements - related tradeoffs

Finally, we are interested in knowing the energy overhead from the interpreted nature of SensorWare. For that purpose we compare the interpreted version of the script presented in section V.A., with a compiled native code version of the same algorithm. The native version uses the services that SensorWare provides by directly calling the appropriate functions. Since most of the work inside a script is done by the SensorWare commands and services (which are implemented in native code) we do not expect a significant change when we resort to fully native code. Indeed, we measured an 8% speedup of the native code compared to the interpreted code. We acquired this number by measuring the total execution times of both codes, and *excluding* time periods when the code was accessing the radio, or was waiting for events to occur. Essentially, the time we measured, was the non-idle CPU time. This time is linearly coupled with the energy spent on the CPU, assuming that we have a mechanism to shut down the CPU during idle time. Thus a reduction of 8% in the non-idle time, directly translates to a reduction of 8% in CPU-energy spent.

As we already mentioned in section V.A, the script has a final compressed size of 209 bytes, while the native code has a size of 764 bytes. So even if the native version executes faster (and potentially consumes less energy, by allowing to shut down the CPU during idle time), there is an energy overhead related to its transmission. The wavelan radio in typical operation would spent 0.47mJ to transmit the script, and 1.10mJ to transmit the native code (including the MAC overhead). Thus, the energy difference between the two transmissions is 0.63mJ. The typical power for the StrongARM is 230mW, so 0.63mJ are spent in 2.7msec. From these numbers we deduce that if the native code uses StrongARM for 2.7msec less than the interpreted code then its initial transmission energy overhead is balanced. For the particular algorithm that we tested, 8% speedup is translated into 1.2msec gain in absolute numbers. So for the particular algorithm transmitting and executing native code is not beneficial overall. For applications with heavier computation workload it might be desirable, from an energy viewpoint, to transmit and execute native code. Note though that we would sacrifice the portability of the code in several platforms, and most importantly we would sacrifice the code safety offered by the scripts (refer to [2] for more information on scripts code safety). Furthermore, most sensor node platforms have a much slower radio than wavelan, which in turn means that they spend more energy to transmit the same amount of bytes, changing the tradeoff points. In conclusion, for most sensor node platforms, one would have to have a very computation-intensive algorithm to prefer the native code over SensorWare scripts.

VII. CONCLUSIONS

In this paper we argue that the development of a framework based on a scripting abstraction where the scripts are mobile, will help bring many desired properties in sensor networks. It will make the sensor networks programmable and open to external users and systems, keeping at the same time the efficiency that distributed proactive algorithms have. We explain the framework's architecture and present code examples. Through our implementation we are able to measure the time and energy overheads that we are paying for programmability and explore some part of the solution space for sensor node run-time environment abstractions.

VIII. REFERENCES

[1] P. Bonnet, J. Gehrke, and P. Seshadri, " Querying the Physical World", IEEE Personal Communications, October 2000.

[2] Withheld to preserve authors anonymity.

[3] Withheld to preserve authors anonymity.

[4] Withheld to preserve authors anonymity.

[5] L. Clare, G. Pottie, J.R. Agre, "Self-Organizing Distributed Sensor Networks", Proceedings of SPIE conference on Unattended Ground Sensor Technologies and Applications, pp. 229-237, April 1999.

[6] eCos: Embedded Configurable Operating System, http://sources.redhat.com/ecos/

[7] D.Estrin, R.Govindan, J.Heidemann (Editors), "Embedding the Internet", Communications of the ACM. Vol. 43, no 5, pp. 38-41, May 2000.

[8] D. Estrin, R. Govindan, J. Heidemann, S. Kumar, "Next Century Challenges: Scalable Coordination in Sensor Networks", ACM Mobicom Conference, Seattle, WA, August 1999.

[9] Familiar Project, "http://familiar.handhelds.org".

[10] M. Hicks, P. Kakkar, J. Moore, C. Gunter and S. Nettles, "PLAN: A Packet Language for Active Networks", Proceedings of the International Conference on Functional Programming (ICFP '98), 1998.

[11] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization", Intel Research IRB-TR-02-00N, 2002.

[12] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System Architecture Directions for Networked Sensors", Proceedings of ASPLOS-IX, November 2000 Cambridge, MA, USA.

[13] Honeywell HMR-2300 Magnetometer, http://www.ssec.honeywell.com.

[14] T. Imielinski and S Goel, "DataSpace: Querying and monitoring deeply networked collections in physical space", IEEE Personal Communications, Oct. 2000.

[15] iPAQ 3670, http://thenew.hp.com/.

[16] C. Jaikaeo, C. Srisathapornphat, and C. Shen, "Querying and Tasking of Sensor Networks", SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V), Orlando, Florida, April 26-27, 2000.

[17] D. Kotz, R. Gray, "Mobile Agents and the Future of the Internet", in ACM Operating Systems Review, 33(3), 1999.

[18] J. Labrosse, " MicroC/OS-II: The Real Time Kernel", CMP Books, November 1998.

[19] P. Levis, D. Culler, "Maté: A Tiny Virtual Machine for Sensor Networks." Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X), October 5-9 2002.

[20] J. K. Ousterhout, "Scripting: higher level programming for the 21st Century", Computer, vol.31, (no.3), IEEE Comput. Soc, March 1998. p.23-30.

[21] J. K. Ousterhout, "Tcl and the Tk toolkit", Addison-Wesley, 1994.

[22] G.J. Pottie and W.J. Kaiser, "Wireless Integrated Network Sensors", Communications of the ACM. Vol. 43, no 5. May 2000.

[23] Reactive Sensor Networks, http://strange.arl.psu.edu/ RSN/

[24] Rockwell WINS nodes, http://wins.rsc.rockwell.com/

[25] SenseIT program, http://www.darpa.mil/ito/research/sensit/index.html

[26] C. Srisathapornphat, C. Jaikaeo, and C. Shen, "Sensor Information Networking Architecture", International Workshop on Pervasive Computing (IWPC'00), Toronto, Canada, August 21-24, 2000.

[27] D. Tennenhouse, "Proactive Computing", Communications of the ACM. Vol. 43, no 5, pp.43-50, May 2000.

[28] Wavelan card, "http://www.orinocowireless.com"

APPENDIX

A. The SensorWare Language

SensorWare supports Tcl syntax and the following 41 Tcl commands: append, array, break, case, catch, concat, continue, error, eval, expr, for, foreach, format, global, if, incr, info, join, lappend, lindex, linsert, list, llength, lrange, lreplace, lsearch, lsort, proc, regexp, regsub, rename, return, scan, set, split, string, trace, unset, uplevel, upvar, while.

There are 11 other commands defined by SensorWare that essentially abstract the node's run-time environment. They are:

spawn [-[f] [d] [p] [m] [rc] [rs] [ru]] [<node_list>] <code> [<variable_list>]

replicate [-[f] [d] [p] [m] [rc] [rs] [ru]] [<node_list>] [<variables_list>]

migrate [-[f] [d] [p] [m] [rc] [rs] [ru]] [<node_list>] [<variables_list>]

send [<node_id>][:<script_name>] <message>

setTimer <timer_name> <value>

disposeTimer <timer_name>

query <device_name> [var_arg...]

act <device_name> [var_arg...]

createEventID <device_name> <eventID> [var_arg...]

disposeEventID <device_name> <eventID>

wait <event_name>...

<u>Legend:</u> [ ] indicates optional, < > indicates a variable (either a Tcl variable or an SensorWare variable such as an eventID or a timer name), the suffix "_list" in variable names indicates that the variable is a list (i.e., zero or more elements). The symbol "var_arg ..." indicates variable arguments. The modifier "..." indicates a list of arguments of the preceding argument type.

There are 6 reserved Tcl variable names. These are: parent, neighbors, event_name, event_data, msg_sender, msg_body.

There are 7 reserved words used as arguments in some commands. By reserving words for commonly used features we compact the scripts further. These are: anyRadioPck, anyTimer, add_user, sensor, value, radio, timer.