Multi-level Software Reconfiguration for Sensor Networks

Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, Mani Srivastava
University of California at Los Angeles
420 Westwood Plaza
Los Angeles, California, USA
{rahulb, simonhan, ram, ilias, mbs}@ee.ucla.edu

ABSTRACT

In-situ reconfiguration of software is indispensable in embedded networked sensing systems. It is required for re-tasking a deployed network, fixing bugs, introducing new features and tuning the system parameters to the operating environment. We present a system that supports software reconfiguration in embedded sensor networks at multiple levels. The system architecture is based on an operating system consisting of a fixed tiny static kernel and binary modules that can be dynamically inserted, updated or removed. On top of the operating system is a command interpreter, implemented as a dynamically extensible virtual machine, that can execute high-level scripts written in portable byte code. Any binary module dynamically inserted into the operating systems can register custom extensions in the virtual machine interpreter, thus allowing the high-level scripts executed by the virtual machine to efficiently access services exported by a module, such as tuning module parameters. Together these system mechanisms permit the flexibility of selecting the most appropriate level of reconfiguration. In addition to detailing the system architecture and the design choices, the paper presents a systematic analysis of flexibility versus cost tradeoffs provided by these mechanisms.

Categories and Subject Descriptors: C.3 [Special - Purpose and Application-Based Systems]: Real-time and embedded systems

General Terms: Performance, Design, Reliability

Keywords: Virtual Machine, Sensor Networks, Reprogramming, Reconfiguration, Multi-tasking

1. INTRODUCTION

Embedded in buildings [18], forests [4], machinery [5] etc. as a part of the infrastructure, the sensor networks are used in uncontrollable environments, whereby users often do not have exact information about the sensing data or the network characteristics. So, they must be able to reprogram or *reconfigure* the sensor network after deployment [9]. In addition, they are increasingly being seen as a *shared* resource with multiple concurrent users. Hence, they accommodate a variety of services over their lifetime. This emphasizes the ability to *retask* and *re-use* the deployment on demand. However, it is not possible to have precise information about all future operation scenarios of a sensor network before deployment. Hence, the design should be *flexible* enough to accommodate the new requirements, while simultaneously respecting the severe resource constraints imposed by typical sensor nodes.

Current research in sensor networks has focused on achieving a single point in design space. Various reconfiguration mechanisms proposed earlier make a trade-off in flexibility vs. update cost as shown in figure 1. Update cost is defined as the network energy required to retask or reconfigure the sensor network. *Full image binary upgrades* in TinyOS [11] provide maximum flexibility by allowing arbitrary changes to the functionality, but incur unacceptable update cost. *Modular binary upgrades* in systems like SOS [10] provide almost similar flexibility as the full image upgrade but at a significantly lower cost. *Virtual machines* provide a more cost efficient way to update application level functionality of the system. However, the virtual machine scripts are severely restricted in the flexibility of updates. *Parameter tuning* frameworks provide the least expensive and least flexible way to change software. A more detailed discussion of these mechanisms is deferred till section 2.

We believe that choosing a fixed design point as above, restricts the applicability of the system to a few scenarios. Changing applications often require the underlying run-time system to dynamically transition from one operating point to the other with least overhead. Application Specific Virtual Machine (ASVM) [16], based on the Maté [15] virtual machine framework for TinyOS, proposes an architecture for building domain-specific run-times. To the best of authors' knowledge, it is the first attempt at providing two design choices in the same system combined with an ease of use. But, given the high update cost in TinyOS, it restricts frequent re-use of same sensor network deployment for another domain (unless of-course, the user has precise information about the future before initial deployment, which we argue, is often not possible). Systems providing modular binary upgrades, like SOS, overcome this limitation, but incur a higher update cost for simple application-level modifications like parameter reconfiguration. Porting parameter tuning frameworks to the above systems will solve only a part of the problem as they loose out on the benefits provided by a virtual machine.

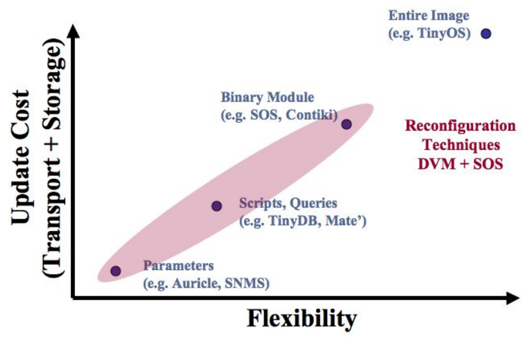


Figure 1: Software Reconfiguration in Sensor Networks

In this paper, we present a system that supports *flexible* software reconfiguration of sensor networks. It integrates the three design alternatives (excluding full binary upgrades) discussed above, into one complete system. The architecture consists of SOS kernel [10], that allows incremental upgrades to the system by dynamically linking binary modules at run-time. A virtual machine (VM), implemented on top of SOS, executes application scripts in response to some events. The design, partly inspired by the ASVM and Maté framework, enables dynamic extensibility of the virtual machine at run-time. Essentially, it allows the users to define the abstraction boundary between the native code and virtual machine script *on-the-fly*. Shifting the boundary helps trade-off update cost for execution efficiency and flexibility. Hence, the name - *Dynamically extensible Virtual Machine* (DVM). Our primary intellectual contribution is the design, implementation, and evaluation of the DVM architecture that is based on the design principles discussed above. The DVM source code is a part of the SOS distribution and can be downloaded from [3].

Three unique features of DVM make it different from existing architectures.

- The SOS binary modules can expose an interface to the virtual machine. Henceforth, we will refer to such modules as *scriptable modules*. DVM leverages the power of efficient modular upgrades in SOS to allow these modules to register custom extensions to the virtual machine at run-time. Therefore the set of functionality available to the virtual machine and the scripts is dynamic. This helps a sensor network deployment to be *re-used* for different domains in a cost efficient manner.

- The application scripts can interact with the scriptable binary modules running natively on SOS. These scriptable modules can expose their configurable parameters to the virtual machine through a published interface. Thus, the scripts provide a very concise and flexible way to *reconfigure* module parameters according to the operating environment.

- Besides allowing users to install and update the scripts, it enables them to modify, at run-time, the set of events to which the VM responds. This is crucial for *retasking* or *multi-tasking* the system as different applications often react to a different set of events.

The complete virtual machine architecture consists of two parts. (i) *Core VM* framework, which runs on the sensor nodes and is responsible for executing scripts. (ii) A high-level *language*, which is used to write application scripts, and is compiled to the virtual machine instruction set. This paper describes only the core VM architecture. The language design is left as future work, but given due consideration while choosing the VM instruction set. It is assumed that, for now, the application scripts are written manually in the VM instruction set, but later, they will be generated by the language compiler.

The outline of the paper is as follows. The related work in the area of software reconfiguration is described in section 2. An architecture to support multi-level reconfiguration contains two core components - SOS operating system and the Dynamic Virtual Machine. Section 2 also provides a brief background about the SOS operating system and its features that are crucial to the design of DVM. The architecture of the dynamic virtual machine is described in section 3. It also discusses its unique features and the various design issues that arise due to its dynamic nature. Section 4 evaluates the performance overhead of the system for micro-benchmarks and analyzes the trade-offs in cost vs. flexibility provided by the DVM through the case study of an outlier detection application. We conclude in section 5.

2. RELATED WORK

There have been many proposals for implementing software reconfiguration in resource constrained sensor networks. They can be classified into four major categories as discussed in the previous section (figure 1). We present related works in every category.

Full Image Upgrades: TinyOS [11] is a popular operating system for sensor nodes that generates a monolithic binary image of the entire application. Deluge [12] is a networked bootloader and dissemination protocol that performs full image upgrades of TinyOS applications. The high update cost is primarily due to the large size of the monolithic binary image (30-40 KB) that needs to be transferred reliably to the entire network. Installing the full image also disrupts the ongoing applications on the nodes, resulting in a loss of work and resources that has already been spent in processing the previous data. Optimizations to the full image upgrades include sending a diff of the new image into the network. Such approaches have been proposed in [20] and [13]. The main drawback of such approaches is the that the complex algorithm to patch the diff images needs to execute on the resource constrained nodes.

Modular Binary Upgrades: Systems that support modular binary upgrades comprise mainly of a run-time loader and linker. The loader is responsible for tracking the storage of the binary modules in the program memory and allocating appropriate resources for them to execute. The linker is responsible for resolving any references made by the modules to the kernel or other modules in the system.

SOS [10] and Contiki [6] operating systems allow modular binary upgrades at run-time. The modules are installed seamlessly into the system and don't cause process-disruption as mentioned above. The diff based optimizations (suggested for full image upgrades) are also applicable to the modular binary upgrades.

SOS: The architecture of SOS consists of a thin kernel that is statically installed on all the nodes in the network. The rest of the system and application components are implemented as modules. Modules are binary software components that can be dynamically installed on a node at run-time. The SOS kernel provides support for loading and unloading modules at run-time, besides a rich set of services such as dynamic memory allocation, software timers, sensor manager and high-level I/O interface.

Inter-module communication in SOS can be synchronous or asynchronous. Asynchronous communication between the modules occurs through message passing. Modules post messages to other modules which are queued by the SOS kernel and dispatched to the destination module. The module is implemented as a message handler that processes the received messages. Synchronous communication between the modules occurs through a process of dynamic linking. Every module indicates the set of functions that it subscribes from other modules and the set of functions that is provides to the rest of the system. The dynamic linker tracks down and links all the publish subscribe pairs in the system during the load time of a module. These functions are referred to as *dynamic functions* in the latter sections.

Virtual Machines: Maté [15] was the first virtual machine architecture proposed for the resource constrained sensor devices. It had very limited flexibility and minimal support for concurrency. The Application Specific Virtual Machines (ASVM) [16] proposed by Levis et. all addressed the limitations of Maté. It provides a customizable and extensible abstraction boundary that is fixed during the compilation of the virtual machine. This enables the generation of very concise and powerful scripts for implementing application logic. DVM significantly improves the capabilities of ASVM by allowing the users to change this abstraction boundary at run-time.

Agilla [7] is a mobile agent architecture for sensor nodes. The primary focus of Agilla is to provide primitives for efficient code propagation and exploring programming models based upon mobile agents. VM* [14] is a system that interprets JAVA bytecodes on the sensor nodes. The sensor network application is written in JAVA and compiled to class representation. VM* tools compact the class representation and automatically synthesize a virtual machine that natively implements some of the system classes. We were unable to evaluate our system against the VM* as its source code was unavailable for distribution.

Parameter Updates and Query Frameworks: SNMS [22] is a framework for updating parameters of TinyOS components written in NesC [8] programming language. SNMS has very limited flexibility but it also has a very low update cost.

TinyDB [19] is a SQL-like query framework for gathering data from sensor networks. The TinyDB framework allows re-tasking of the software by moving around points of data aggregation in the network. VanGo [9], a high rate data collection system for sensor networks, aims to narrow down the fundamental gap between the data collection and data transmitting capabilities of sensor nodes. It uses flexible transcoding to enable *online* calibration of data processing algorithms so as to reduce the amount of data transmission without loosing precision.

3. DYNAMIC VIRTUAL MACHINE

In this section, we first describe the design of various components which make up the DVM. Emphasis is given to the specific design choices which enable unique features of the DVM system. We briefly discuss various consistency issues which arise due to the dynamic nature of DVM. Next section elaborates on the mechanisms which help the system achieve its goals of *flexible reconfiguration*. The term *reconfiguration* is used here in the broader sense of enabling parameter tuning, retasking and multi-tasking of the sensor network deployment.

The outlier detection (OD) application is used as an example to explain the functionality and various features of the system. Figure 2 shows the application dataflow graph. It samples the light sensor at a sample rate `P`. `a(n)` is the current sensor reading, `a[0:S]` is the buffer consisting of `S` such readings, and `r[0:S]` are the ratings calculated by the distance based Outlier Detection module for each reading. Distance between pairs of readings is calculated using

$$D[i,j] = |a[i] - a[j]| \tag{1}$$

The readings classified as *outliers* are not included while calculating simple average in *Action*. Implementation of the application using DVM will consist of two scripts - (i) *init script*, to initialize the application parameters (`P`, `S`, <`f`, `T`>) and the timer; and (ii) *timer handler*, to be executed once at each timer expiration. The outlier detection block is also referred to as the *core OD* block later in this text.

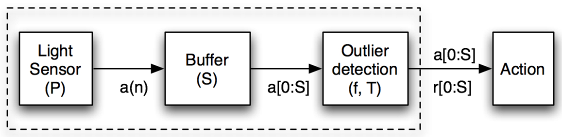


Figure 2: Outlier Detection (OD): Dataflow

3.1 DVM Architecture

The design of Dynamic Virtual Machine (DVM) is shown in figure 3. The concurrency manager in the DVM is functionally equivalent to its counterpart in the ASVM. It is responsible for sharing of resources while ensuring race-free execution of the application scripts. The scheduler and the capsule store develop on the basic ideas, borrowed from ASVM, to support dynamic addition of new instructions and reduced memory (RAM) usage, respectively. However, the dynamic nature of the DVM requires additional components such as Event Manager, Basic Library, Extension Libraries and the Resource Manager. Event Manager is responsible for handling various events generated in the system, while the Resource Manager performs simple admission and installation control for scripts. The two libraries implement the various instructions recognized by the DVM. All these components are a part of the SOS kernel and are not dynamically loadable, except the Extension libraries.

The DVM defines two major abstractions: *handlers* and *operations* [16]. Handlers are code routines that run in response to events. Operations are the units of execution functionality [1].

[1]The terms 'operations' and 'instructions' are used interchangeably in this text.

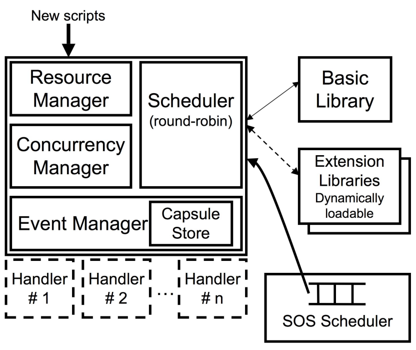


Figure 3: DVM Architecture

DVM requires reliable delivery of scripts, and binary modules implementing the Extension library, for consistent operation across the network. A reliable code dissemination protocol has been implemented, by modifying the Trickle protocol [17], for both SOS and DVM. This paper does not discuss reliable distribution further as it is orthogonal to the issue considered here. The following sections describe some components of DVM highlighting their contribution towards its dynamic features and related issues.

3.1.1 Resource Manager

Memory is one of the most severely constrained resources on a typical sensor node. It gains more significance in systems like SOS where it is allocated dynamically at run-time. The Resource Manager is required to deal with varying levels of memory usage to ensure minimal operating conditions for the DVM scripts. It performs two important functions: *Admission* control and *installation* control for scripts.

All scripts operate on their individual stacks and manipulate local and global variables. They also need space to store their state which includes their ID, program counter and resources held or required, besides various other variables. Typical sensor nodes like the Berkeley Mica motes [2] based on AVR micro-controller [1] have limited memory (4 KB RAM). Thus, it is not possible to *admit* each and every script into the system. Moreover, scripts can specify dependency amongst themselves in the script headers. Allowing incompatible scripts to execute concurrently can lead to corruption of data and/or incorrect results. The Resource Manager performs admission control by ensuring there is enough memory for execution of all dependent scripts. It guarantees memory for one script to allow the users to perform corrective action like evicting other scripts to free up resources used by them.

Once admitted, the scripts need to be *installed* into the system so that they can be invoked for handling appropriate events. This procedure should cause least disruption of the currently executing scripts to save work that has already gone into executing them. Hence, the script is installed only if there is no other script being executed at that time, and no script is marked *ready to run* at the scheduler. In case the above check fails, the resource manager queues the script for installation later. The memory overhead imposed by this approach is dependent on the length of the *wait* queue in the resource manager, which is configurable at compile time. It is roughly three bytes per queue element. Meanwhile, the entry into scheduler's *run* queue is blocked. The event handlers invoked during this period are enqueued in the concurrency manager until the installation is complete. As a result, the response time of the system increases for certain events.

3.1.2 Event Manager

An event in DVM is specified by the tuple $<$ *Module Identity, Message Type*$>$. This maps to asynchronous messages in SOS described in section 2. The event information is carried along with the script in the form of meta-data and is extracted by the capsule store at the time of initialization. Capsule store is mainly responsible for storing the scripts.

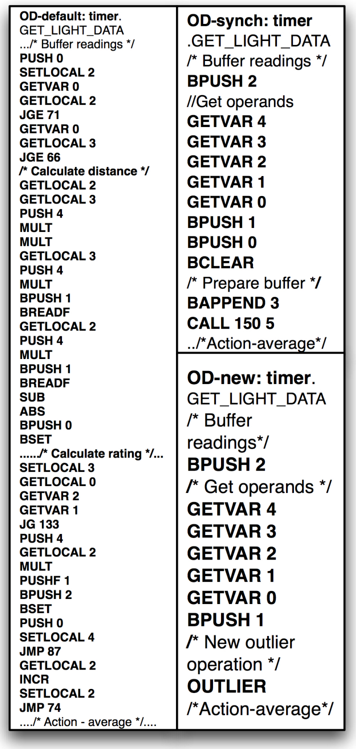
As stated in section 1, the ability to change the set of recognized events at run-time is crucial to *flexible retasking* of the sensor network. The event manager is responsible for binding the handler script to its corresponding event at run-time. This architecture improves the flexibility of the DVM to incorporate new events. An event can invoke multiple scripts. The reverse is also true - a script can have multiple event definitions in its header, and can be installed to handle multiple events. Each instance requires separate memory for script state, stack and local variables. Thus, it may not be always possible to admit all the instances of the script. The policy decision - on whether to install none or as many as possible instances - is indicated to the resource manager through the use of a flag in script header.

The script invocation at an event is a two step process. First, the Concurrency Manager analyzes a script and tries to obtain locks on all the resources required by it. On success, the script state is submitted to the scheduler for execution, or marked as *waiting for resources.*

3.1.3 Operation Libraries

The operation libraries are responsible for implementing the DVM instruction set. Each library is an SOS module that provides a function called *execute*, which decodes all the instructions and calls respective functions. It can be categorized into a Basic or an Extension library depending on type of operations implemented by it. There is only one Basic library, and it implements operations directly supported by the DVM. The choice of the operations supported by the Basic library define the programming language for the DVM. This is a crucial design decision as it directly impacts the flexibility and the efficiency of the scripts. It includes arithmetic operations, stack operations, buffer and variable access operations, which constitute the majority of any application script. In addition to these, the operations to enable script-module interaction are included in the basic library. The commonly used SOS kernel services such as timers, sensor management and radio messaging are also available as basic operations to the DVM scripts. All these operations are required for meaningful execution of scripts irrespective of application requirements.

The use of only basic library operations, excluding script module interaction, pushes the abstraction boundary towards low level operations. This sacrifices execution efficiency and flexibility for lower update cost. The version of OD application implemented in the basic instruction set is referred to as *OD-default.* Figure 4 shows the structure of the timer handler scripts for OD-default and other versions



```
OD-default: timer.
GET_LIGHT_DATA
.../* Buffer readings */
PUSH 0
SETLOCAL 2
GETVAR 0
GETLOCAL 2
JGE 71
GETVAR 0
GETLOCAL 3
JGE 66
/* Calculate distance */
GETLOCAL 2
GETLOCAL 3
PUSH 4
MULT
MULT
GETLOCAL 3
PUSH 4
MULT
BPUSH 1
BREADF
GETLOCAL 2
PUSH 4
MULT
BPUSH 1
BREADF
SUB
ABS
BPUSH 0
BSET
....../* Calculate rating */...
SETLOCAL 3
GETLOCAL 0
GETVAR 2
GETVAR 1
JG 133
PUSH 4
GETLOCAL 2
MULT
PUSHF 1
BPUSH 2
BSET
PUSH 0
SETLOCAL 4
JMP 87
GETLOCAL 2
INCR
SETLOCAL 2
JMP 74
..../* Action - average */....
```

```
OD-synch: timer
.GET_LIGHT_DATA
/* Buffer readings */
BPUSH 2
//Get operands
GETVAR 4
GETVAR 3
GETVAR 2
GETVAR 1
GETVAR 0
BPUSH 1
BPUSH 0
BCLEAR
/* Prepare buffer */
BAPPEND 3
CALL 150 5
../*Action-average*/
```

```
OD-new: timer.
GET_LIGHT_DATA
/* Buffer
readings*/
BPUSH 2
/* Get operands */
GETVAR 4
GETVAR 3
GETVAR 2
GETVAR 1
GETVAR 0
BPUSH 1
/* New outlier
operation */
OUTLIER
/*Action-average*/
```

Figure 4: OD: Timer handler script consists of 3 parts - buffering sensor data, outlier detection computation and action on processed data.

of the application discussed in section 3.2. The text in bold shows a portion of the outlier detection calculations, which occupy 112 bytes in OD-default.

Extension library modules permit language extensions by providing additional application specific instructions. These libraries are loadable at run-time, and thus allow the users to dynamically shift the abstraction boundary towards higher level operations. This results in higher execution efficiency at the cost of higher update energy. Flexibility is not sacrificed as the basic library resides permanently in the system. The Extension libraries publish *execute* as a dynamic function, which is subscribed to by the scheduler. The scheduler calls *execute* in the appropriate library for every instruction. More details on this mechanism are given in section 3.2.2.

3.1.4 Scheduler

The Scheduler supports addition of custom DVM functionality at run-time. It is responsible for fetching an opcode from the capsule store and partially decoding it to determine the implementation library. Hence, it must have information about all the Extension libraries present in the system. A script depending on a library, which is not yet installed in the system, is not executed untill the required library is received and installed at the node. Thus, whenever a library is added or removed from the system, it is required to inform the scheduler so that it can take appropriate action.

An addition or removal of library may cause inconsistency in the system due to corruption of data and/or production of incorrect results due to a change in implementation of certain instructions. Thus, all the script states and shared variables are reset in order to maintain consistency in such a case. This is referred to as the *VM-reboot*. The disruption caused is not considered significant as the libraries are not expected to be updated frequently.

Rest of the scheduler is similar to Maté. It uses a simple round robin scheduler and employs VM-reboot on script failure during execution to maintain consistency of shared data amongst the scripts.

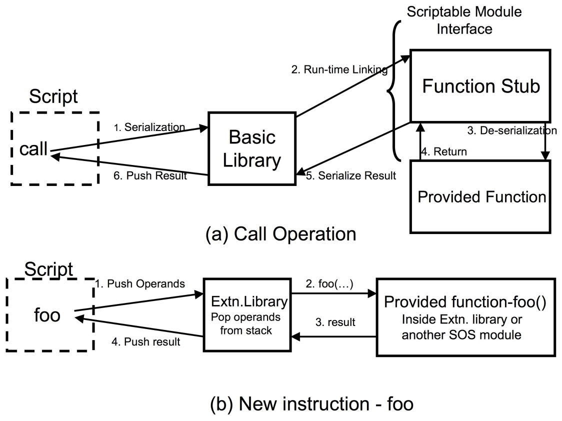


Figure 5: Script-module interaction: `call` operation and a new instruction.

3.2 Features

DVM allows customization of the abstraction boundary between the virtual machine script and native code at runtime. In this section, we first describe the script-module interaction enabling mechanism which provides the basis for above customization. It is supported in the Basic library and is mainly used for parameter reconfiguration and supporting custom application-specific functionality. Another benefit of this mechanism is that it allows the scripts to access complex application specific functionality, which otherwise could have been very difficult to express in basic operations.

Next, we describe a mechanism to register this functionality as a regular DVM instruction. The difference between these two features is subtle and is aptly shown in the figure 5. The latter sacrifices update cost to provide higher flexibility and marginally lower execution cost than the former. Finally, we describe the modifications to the DVM event set in the light of script-module interactions. The focus here is on *multi-tasking* as opposed to *retasking* described in section 3.1.2.

3.2.1 Script-module interaction

The binary modules in the system expose an interface to the DVM. These modules can be added and removed from the system at any time. DVM provides synchronous and asynchronous communication interfaces for interaction between the scripts and the dynamic modules.

The `call` operation invokes a synchronous function call in the destination module. Essentially, it is replacing a part of the script operations with native C code. It pushes the abstraction boundary towards higher level operations, allowing the application to reduce interpretation overhead at the cost of higher update energy. For instance, the piece of script responsible for computing outliers in our application, can be replaced by a single `call` into an SOS module implementing OD, which returns the same result (figure 4). The function in OD module takes the data buffer, size and parameters (`f`, `T`) as arguments. This is referred to as the *OD-synch* application. It results in 30x improvement in execution cost for performing outlier detection (push operands + serialize + core OD, table 3(a)) at the cost of up to 3.5 times higher update energy as compared to OD-default (section 4).

The synchronous functions provided by a module are identified through the tuple < *Module Identity, Function Identity*>. This tuple is present in the bytecode of the script, which implies that it is a constant known at compile time. In our example, the core OD function can be identified by <150,5>. The concurrency manager checks for the presence of scriptable modules accessed by a script before submitting it for execution. This prevents run-time failures due to absence of scriptable modules. Though, the script can be written to handle such errors, we argue against it to keep the script simple and concise.

The parameters to the synchronous function call are serialized and stored in a buffer that is pushed onto the operand stack. The function invoked in the destination module is actually a stub that de-serializes the parameters and invokes the actual function within the module. The return value of the function is again serialized by the stub and returned back to the virtual machine in the form of a buffer. This is shown in figure 5. The prototype of the stub code is fixed (figure 6). It is required to make the operation flexible enough to interact with any SOS module loaded at run-time. Hence, the presence of this stub is necessary to support dynamic *reconfiguration* and *retasking* with the use of this mechanism. Currently, the stub needs to be manually written in the module, but we are working on its automatic generation. However, the requirement to include the stub at compile time often restricts flexibility as the user is expected to know about its exact usage scenario. This goes against our basic assumption of incomplete usage knowledge before deployment. Section 3.2.2 removes this restriction, albeit at a higher update cost.

A typical use of the `call` mechanism is to get or set the application parameters exposed by an SOS module. Setting the parameters helps in tuning the application to its environment, or changing its control flow depending on the design of the module and the parameters exposed by it. The `call` operation can also be used in run-time debugging where a script can periodically retrieve the current state of the module and send it to the base station for analysis. This usage is well-suited to the `call` mechanism as it is reasonable to expect the user to provide the ability to access these parameters at run-time by compiling in the stub before deployment.

```
void stub(uint8_t fnID, DvmBuffer *argBuf, uint8_t size, DvmBuffer *resBuf) {
  switch(fnID) {
     case 5: {
          ArgBufType *args = (ArgBufType *)argBuf->data;   // de-serialize arguments
          outlier_detect(args->readings, args->size, args->param, resBuf);
          break;
     }
     default: break;
  }
}
```

Figure 6: OD-synch Application: Stub code in OD module (id=150) for interaction via `call`

Serialization of parameters imposes an argument dependent overhead on script size and execution. For every `call` operation, the dynamic virtual machine links to the appropriate function in the destination module through a run-time linking mechanism, called *subscribe*, provided in SOS. Though, *subscribe* is expensive in terms of execution, serialization is generally the major constituent of the total overhead for functions which require parameters. The serialization and subscription overhead thus introduced, can outweigh the execution benefits gained by using native code instead of the DVM instructions. We observed this for simple applications like calculating exponentially weighted moving average (EWMA). To retain the flexibility of the `call` operation, we can not do away with *subscribe* for increasing efficiency. However, the subscription result can be cached and reused as a simple optimization. The cache size and refresh policy still need some discussion and evaluation, and hence, are left as future work.

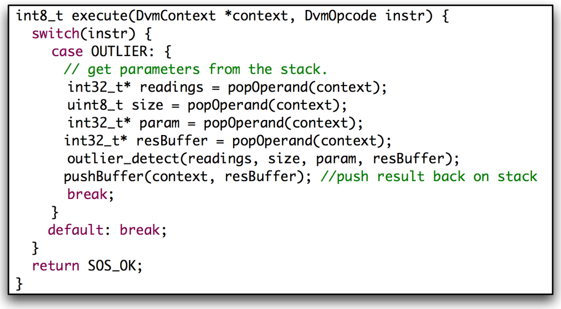
The `post` operation allows the scripts to interact asynchronously with the SOS modules. It provides same level of flexibility as the synchronous approach described above, but with 1.1x higher execution overhead and decreased reliability. This mechanism uses message dispatching in SOS, which has a higher overhead in SOS as compared to the dynamic function call. It is also based on best effort semantics. It could fail due to numerous reasons such as insufficient memory or the absence of destination module. This implies that the `post` operation might fail despite the module being present in the system.

Operation `GET_LIGHT_DATA` in figure 4 is an example of a split-phase interaction between the script and the module providing an API to access sensor readings. It is initiated by a synchronous function call from the script to the sensor module, and is completed when the script receives the data asynchronously [15]. It is not associated with the subscription overhead of `call` instruction as the initial synchronous call is made to a fixed function known at compile time. In future, we plan to extend it to include any arbitrary module providing such a service.

3.2.2 Addition/Modification of new instructions at run-time

DVM allows extension of its default instruction set by addition of Extension library modules that are implemented in native code. It provides a mechanism to convert an interface, exposed by a module, to a regular instruction recognized by the DVM. Essentially, it removes the need to serialize or de-serialize the function parameters by accessing these parameters directly from the stack. Hence, it also removes the *stub-compile-in* restriction discussed in the previous section. In the modified *OD-new* application shown in figure 4, the `CALL` operation is replaced by `OUTLIER` and the operations to prepare the argument buffer are discarded.

Extension library, containing the OUTLIER instruction, publishes its *execute* function upon installation (figure 7). The scheduler subscribes to this function when the library is added to DVM, and stores the pointer for future use. As a result, the overhead of *subscribe* is amortized over multiple executions of the instruction. This amortization, along with the removal of parameter serialization from the script, improves the execution efficiency by a factor of 2.6 (serialize + core OD, table 3(a)), with only a small memory overhead of 2 bytes per Extension library.



```
int8_t execute(DvmContext *context, DvmOpcode instr) {
  switch(instr) {
    case OUTLIER: {
      // get parameters from the stack.
      int32_t* readings = popOperand(context);
      uint8_t size = popOperand(context);
      int32_t* param = popOperand(context);
     int32_t* resBuffer = popOperand(context);
      outlier_detect(readings, size, param, resBuffer);
     pushBuffer(context, resBuffer); //push result back on stack
      break;
    }
    default: break;
  }
  return SOS_OK;
}
```

Figure 7: Extension library: execute() function. Modified from original to enable clear understanding.

In the *execute* function, OUTLIER instruction is essentially a dynamic function call into the module implementing its functionality. Hence, to execute the instruction, scheduler calls *execute*, which further calls the function in the OD module. This is referred to as the *indirect instruction* in latter parts of this paper. It implicitly assumes that the module is present in the system at the time of execution. Thus, it is less reliable than the simple script-module interaction described in previous section, as there are no run-time checks that span across two or more levels of indirection.

In this particular example, it is also possible to implement the core outlier detection function completely inside the Extension Library as it is a stateless service provided to the DVM. This approach is later referred to as the *direct instruction*. It is more reliable than the previous one as it doesn't have that extra level of indirection, and is marginally more efficient in execution time (section 4). But, the indirect approach provides more flexibility as it removes the need of a stub in the module providing the script-accessible function. Thus, modules can be made scriptable at run-time without the need to inject re-compiled modules with appropriate stubs. This promotes re-use of dynamic functions within the DVM-SOS system.

Hence, this mechanism is more flexible and results in better execution efficiency than the `call` operation. However, an overhead of this approach is the energy required to transmit and receive the Extension Library at run-time. Section 4 also shows that an Extension Library is bigger in size as compared to the module providing the OUTLIER function. This is because of the constant overhead imposed to enable interaction with the scheduler and access operands from the stack. Thus, it is encouraged to group instructions into as few libraries as possible to reduce the significance of this overhead.

3.2.3 Run-time modification of events

The previous two sections described script-module interaction where the interaction was initiated by the script. DVM provides another mechanism where it allows SOS modules to invoke the script(s) and pass data to them. This is achieved through the creation of new events with higher semantic values. For instance, in figure 2, an SOS module implementing the part inside the dotted box, can generate a DVM event whenever the outlier detection is complete. A number of *actions* can be performed on the resulting data, like calculating simple average, broadcasting the ratings etc. for the application in consideration. This provides extreme flexibility in using the same (partially) processed data for a variety of application requirements emerging at run-time. Thus, it promotes *multi-tasking* through the use of multiple scripts to handle the same event.

This, in a way, pushes the abstraction boundary towards higher level operations, hence restricting update flexibility for increased execution efficiency. It results in 1.6x faster execution times as compared to both OD-synch and OD-new applications. However, the update cost in this approach varies depending on the kind of update required. An update to the event action will be more energy efficient as compared to previous approaches, while an update to parameters of the event flow graph may be as expensive as an update to OD-sos, i.e. if an appropriate interface has not been exposed for controlling the application parameters. This modification is referred to as the *OD-event* application in later sections.

4. EVALUATION

The previous section described the design and implementation of DVM. It also discussed various features which help achieve *flexible reconfiguration* of sensor networks. In this section, we first try to analyze the DVM overhead over native code execution in SOS. We, then evaluate the trade-off between execution efficiency and update cost provided by the proposed system through the case study of outlier detection (OD) application. The DVM system and OD application were implemented and tested on the Mica2 motes [2], which are based on the AVR AtMega 128L micro-controller (7.3 Mhz, 4 KB RAM, 128 KB internal flash) [1]. Evaluation results presented in this paper were obtained using the Avrora simulator [21].

4.1 Micro-benchmarks

Table 1 measures the overheads imposed by DVM in code size, memory and interpretation over the SOS kernel. The DVM requires 334 bytes of memory for its basic operation, including 8 shared variables and 4 shared data buffers of 32 bytes each. The execution of a script in DVM requires interpretation and concurrency management. Concurrency management in DVM is similar to the corresponding one in ASVM.

Code size	SOS Kernel	DVM
	38 KB	13 KB
Memory	Shared data	Core DVM
	156 B	178 B
Interpretation	Opcode fetch	404
	Opcode decode	66
(CPU cycles)	Stack- Push	42
	Stack- Pop	37

Table 1: DVM Overheads

The interpretation overhead measures the cost to fetch an opcode, decode it, and perform stack operations. The corresponding costs for SOS are almost negligible as these operations are directly supported by the RISC-based micro-controller. Thus, it can be expected that for even a simple operation like increment-by-1, the overhead will be approximately 550 CPU cycles. Then, some of the DVM operations, esp. math related, are overloaded to constrain the number of opcodes. This increases the overhead even further because of embedded type-checking.

It is important to note that the high cost of fetching an opcode from the capsule store is the result of an architecture-specific optimization. The AVR 128L micro-controller has very less data memory (RAM) as compared to program memory (flash). We chose to store the script bytecode on the flash to free up RAM from the burden of storing the script, whose size can vary from tens to hundreds of bytes. Thus, we sacrificed execution efficiency for significantly lower memory consumption.

These results help in explaining the huge cost difference between native code and DVM operations as seen in the next section.

4.2 Application Case Study : Outlier Detection

Experimental Setup : The motes are initially assumed to be installed with the base SOS kernel including the DVM core. The application modules and scripts are sent over the radio at run-time. Energy required to install them is highly dependent on the routing protocol and network density. For our experiments using the modified Trickle protocol and a grid placement with uniform density, it was found that the energy required to install them was directly proportional to the size of modules and scripts. Hence, code size provides a good measure of the initial setup and update cost. The execution time for each version was calculated as the time taken to fill up the buffer with `S` sensor readings, perform outlier detection, and remove the *outliers* from the simple average calculation. The application was executed for 10 minutes and same parameters ($S = 4$, $P = 2sec$, $f = 0.40$, $T = 100.00$) and data set were used for all the experiments.

Results : Table 2(a) compares the various versions of the OD application for their execution cost, code size and memory requirements. It lists the applications in decreasing proportion of their functionality implemented in native code. The execution times observed follow an increasing trend as expected from the micro-benchmarks. The memory requirements reported in the table is the overhead over the SOS kernel and core DVM usage. In all DVM related versions, the majority of memory overhead is due to space required for the script state.

It can be observed that the size of SOS modules is significantly higher than the corresponding scripts. This is due to two main reasons. First, the DVM primitives are at a higher abstraction level than the primitives available to the native code. For example, the scripts invoke the timer service through a `SETTIMER` opcode while the module access it through the `ker_timer_start` SOS system call. The native code that corresponds to the invocation of the SOS system call is larger than the DVM opcode. Second, the modules implement their own event management while the DVM performs the event management for the scripts [2].

Each version of the application has different execution costs for each component (figure 2) as shown in table 3(a). The differences mainly arise due to implementation in native code vs. virtual machine script. Buffer readings refer to the execution cost for buffering the light sensor readings, while next column gives the cost to push arguments, to core OD function, on the respective stack. Serialization costs in OD-event refer to serializing the result before passing it to the action script. For OD-default, the cost to push operands is merged into the core OD value as it does not call any function explicitly. It stands out in core-OD costs due to implementation in basic script operations. Marginal differences in OD-new vs. OD-event / OD-sos are due to dynamic function call overheads.

Table 2: Outlier Detection Application Comparisons

(a) Execution cost, code size, memory

Application	**Code Size** (bytes)				**Execution Cost**	**Memory**
	SOS Module		**DVM Script**			
	OD	Library	Init	Timer	(CPU Cycles)	(bytes)
OD-sos	1304	-	-	-	4,933	15
OD-event	1142	-	4	60	93,829	153
OD-new (direct)	-	1356	22	85	153,151	145
OD-new (indirect)	684	882	22	85	153,274	147
OD-synch	684	-	22	90	158,310	138
OD(default)	-	-	22	211	531,557	138

(b) Initial code size of updates

Application	**Update Component** (initial size)			
	Application Parameters	OD function	Parameters to OD function	Action
OD-sos	1304	1304	1304	1304
OD-event	1142	1142	1142	60
OD-new (direct)	22	1356	1441	85
OD-new (indirect)	22	684	1651	85
OD-synch	22	684	774	90
OD(default)	22	211	211	211

Table 3(b) lists the breakup of core OD costs for OD-new and OD-synch versions. It shows that OD-synch incurs a little higher cost mainly due to subscription overhead per invocation. The cost to fetch operands for the `call` operation in OD-synch is constant and independent of the number of arguments. It fetches < `module identity`, `function identity` > from the flash, and pops two operands from the stack for the argument buffer to the OD( ) function and result. On the other hand, operand fetch cost for OD-new versions depends on the number of arguments to the OD( ) function. It may lead one to believe that for higher number or arguments, the cost for OD-new versions may increase beyond that of OD-synch. However, the total cost for OD-synch will still dominate due to high serialization overhead as shown in table 3(a). Other miscellaneous costs are due to opcode decode in the libraries and dynamic function call overheads. The indirect version has the highest miscellaneous overhead as expected due to an extra dynamic function call on top of *execute( )*.

Analysis : Typical sensor nodes are mainly constrained by limited memory and energy availability. Since, table 2(a) shows that all OD versions have similar memory consumption, with the exception of OD-sos, the design choice should be based on energy consumption of the network. The total energy consumption per node is the sum of network and execution energy. In reconfiguring or retasking the sensor nodes, network energy to distribute the new updates often has a significant impact on design decisions. Thus, table 2(b) brings out the trade-off between execution efficiency and update cost to help us analyze the available design alternatives. It broadly classifies various reconfiguration options

[2]We are using the terms event and message interchangeably

Table 3: Outlier Detection: Distribution of execution costs

(a) Breakup of total execution costs

Application	Buffer readings	Push operands	Serialize	Core OD	Action
OD-sos	1544	209	-	2353	827
OD-event	1544	209	1197	2353	88,526
OD-new (direct)	58,328	4206	-	2923	87,694
OD-new (indirect)	58,328	4206	-	3046	87,694
OD-synch	58,328	4206	3719	3959	87,694
OD-default	58,328	-	-	385,535	87,694

(b) Breakup of core OD costs

Application	Fetch operands	Subscribe + Stub	OD(...)	Other Misc.
OD-new (direct)	583	-	2137	203
OD-new (indirect)	583	-	2137	326
OD-synch	886	813	2137	123

as updates to: (i) Application parameters, (ii) OD function, (iii) Number and/or type of arguments to OD, and (iv) Action to be taken on the (partially) processed data. We consider the effect of these updates independently of each other to keep the analysis simple.

Application Parameters : An update to application parameters consists of updating either all or few of `P`, `S`, or <`f`, `T`>. OD-default, OD-synch and OD-new versions have lower update costs as it is required to change only the init script. The rest require bigger SOS modules to be sent out, i.e. if they do not provide script-accessible functions to change their parameters. In this case, and all the cases discussed below, it is difficult to pick out a clear winner because the total energy consumption of the nodes is highly dependent on the frequency of the updates, which we claim is usage dependent and thus, difficult to know accurately before deployment. For instance, OD-sos will have lower energy consumption than others if the updates are spaced out well enough so as to make execution energy dominate the total energy consumption.

A simple calculation of this *optimal frequency* for all the options is possible for applications with a fixed sample rate [3]. But, for purely event-driven applications, or applications with a dynamic sample rate, it becomes difficult to determine this parameter before deployment. The only solution in these cases is to monitor live data after deployment, and then make a smooth transition to the desired operating point. The ability to make this transition at run-time is the main contribution of our system. An important question still remains - *how do we make the initial choice?*. We subsequently try to answer this question below. Our answers are guided by an important observation - *initial deployments are often very dynamic and susceptible to frequent changes; later, as the applications get tuned to the environment, updates become rarer.* Hence, *flexibility* provided by the update also becomes an important concern while choosing the initial operating point.

OD function : A modification to the core OD function, such as a change in equation 1, will require an SOS module to be replaced for all versions except OD-default. Here, it is difficult to pick an option just by observing, as the transitions in update cost are not as drastic as in the previous case. Intuitively, OD-synch and OD-new (indirect) seem to be a safe option as they have medium update and execution cost as compared to others. According to the authors, OD-new (indirect) is a better choice to start here as it provides more flexibility and better execution efficiency than both the synch and default option.

Parameters to OD function : The third update scenario describes the worst case for above, where number and type of arguments to the OD function also change. This necessitates replacement of the existing extension library with a newer version providing an appropriate interface to the function in OD-new case. Making similar arguments as the last case, we believe that OD-synch makes a good initial choice for a dynamic deployment.

Action : Last update scenario considers addition, modification or removal of the script responsible for processing data received from the SOS module. This efficient multitasking is the sole benefit of OD-event over OD-sos. OD-event is also a clear choice above other DVM-involved options as it is incurs lesser execution cost.

It is important to note here that a combined requirement of more than one scenarios, which will mostly be the case, will complicate the analysis tremendously. While calculating energy consumption for even fixed-rate applications, we will need more information about the usage scenario than before. For instance, we will need to know the *expected* relative frequency of each update and assign weights to them accordingly. This goes against our central hypothesis of incomplete information before deployment.

The results shown in this section indicate that OD-new (indirect) achieves a mid-point in the execution efficiency - update cost spectrum. Our analyses from section 3.2 also show that it is more flexible than the OD-synch, new (direct) and default approaches. Thus, it is recommended to start the deployment by operating at this point in the design space, unless there are compelling reasons, like extreme memory constraints, for not doing so. Later, as usage scenario becomes clear, users have the option to move to farther ends of the design spectrum depending on long-term usage trends.

5. CONCLUSION

The emerging operating scenarios for sensor networks require the ability to support unforeseeable applications and multiple users. It is often not possible to know the exact usage scenario before deployment. Thus, *flexible* techniques to *reconfigure* the software after deployment have become a prerequisite for sensor networks. The proposed system provides several alternatives to update the network, each with a unique cost versus flexibility trade-off. Its architecture comprises of the SOS kernel and a Dynamically Extensible Virtual Machine implemented on top of it. The main features of the system include script-module interaction, language set extension and the dynamic set of events recognized by the DVM. It is observed that it is often difficult to de-

[3]We get a simple linear system of equations for each option, which can be solved easily by several computational tools. To calculate the network energy, the equations also require a network constant that depends on the topology and routing protocol as mentioned earlier. Execution cost from experimental results can be used to calculate execution energy.

terministically choose an initial optimal operating point for a network prior to its deployment. The script-module interaction, along with the language set extension provide a reasonable starting point as they achieve mid-points in the flexibility vs. cost tradeoff.

Future Work : The ability to manage memory dynamically in SOS provides us with lots of interesting optimizations to explore. Currently, allocation of memory for the script state is done only at script admission time. This memory can be de-allocated only by explicit eviction of a script by the user. An interesting future direction will be to implement a light-weight garbage collector which automatically reclaims unused script memory for use by other applications on the system.

DVM allocates a stack of fixed size for each instance of a script irrespective of its actual requirement. This is found to be very restrictive for simple scripts like the init script in our example. A conservative solution to this problem is to calculate the worst case stack requirement while compiling the script, and allocate space accordingly at script admission. We believe that this approach will be highly restrictive on the low memory sensor nodes. Hence, we plan to use dynamic memory service provided by SOS to *reallocate* more stack space for the scripts if they exhaust their pre-allocated space. The reallocation will be limited by a maximum allowed stack space to prevent buggy or malicious scripts from harming the co-existing applications. The initial and maximum limits on the stack space will be determined empirically.

As stated in previous sections, work is already going on to develop a high level language and a compiler for the DVM framework. An automatic stub code generator is also being developed so that it is convenient to make a module scriptable at compile time.

6. ACKNOWLEDGMENTS

This paper is based upon work supported by NSF award # 0306408, ONR award # N00014-06-1-0253, and by the Center for Embedded Networked Sensing. Any opinions, findings and conclusions or recommendations expressed in this papers are those of the author(s) and do not necessarily reflect the views of the NSF, ONR, or CENS.

7. REFERENCES

[1] Avr ATMEGA micro-controller. http://www.atmel.com/products/avr/.

[2] Mica2 motes (UC Berkeley/Crossbow). http://www.xbow.com/.

[3] SOS-1.x project. http://nesl.ee.ucla.edu/projects/sos-1.x.

[4] Neon: Addressing the nation's environmental challenges. http://www.neoninc.org/, November 2003.

[5] R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, L. Krishnamurthy, N. Kushalnagar, L. Nachman, and M. Yarvis. Design and deployment of industrial sensor networks: Experiences from the north sea and a semiconductor plant. In *Third ACM Conference on Embedded Networked Sensor Systems (SenSys)*, November 2-4, 2005.

[6] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE workshop on Embedded Networked Sensors*, 2004.

[7] C. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. Technical report, Washington University, Department of Computer Science and Engineering, St. Louis, 2004.

[8] D. Gay, P. Levis, R. von Behren, and M. Welsh. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation*, 2003.

[9] B. Greenstein, A. Pesterev, C. Mar, E. Kohler, J. Judy, S. Farshchi, and D. Estrin. Collecting high-rate data over low-rate sensor network radios. Technical report, CENS Technical Report 55, UCLA, 2005.

[10] C. C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *Third International Conference on Mobile Systems, Applications, And Services (Mobisys)*, 2005.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the ninth internation conference on Architectural support for programming languages and operating systems*, 2000.

[12] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the second internation conference on Embedded Networked Sensor Systems*, 2004.

[13] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Proceedings of the First IEEE Communications Society Conference on Sensor and Ad-Hoc Communications and Networks (SECON)*, 2004.

[14] J. Koshy and R. Pandey. Vm*: Synthesizing scalable runtime environments for sensor networks. In *Proceedings of the third international conference on Embedded Networked Sensor Systems*, 2005.

[15] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of the tenth international conference on Architectural support for programming languages and operating systems*, 2002.

[16] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the second international conference on Networked Systems Design and Implementation (NSDI)*, 2005.

[17] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code maintenance and propagation in wireless sensor networks. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[18] J. P. Lynch, S. Seth, and D. M. Tilbury. Feasibility of real-time distributed structural control upon a wireless sensor network. In *42nd Allerton Conference on Communication, Control, and Computing*, September 2004.

[19] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *In proceedings of SIGMOD*, 2003.

[20] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM internation conference on Wireless sensor networks and applications*, 2003.

[21] B. L. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Fourth International Conference on Information Processing in Sensor Networks (IPSN)*, 2005.

[22] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.