Active Sensor Networks

Philip Levis[†], David Gay[‡], and David Culler[†]
{pal,culler}@cs.berkeley.edu, david.e.gay@intel.com

[†]EECS Department
University of California, Berkeley
Berkeley, CA 94720

[‡]Intel Research Berkeley
2150 Shattuck Avenue
Berkeley, CA 94703

ABSTRACT

We propose using application specific virtual machines (ASVMs) to reprogram deployed wireless sensor networks. ASVMs provide a way for a user to define an application-specific boundary between virtual code and the VM engine. This allows programs to be very concise (tens to hundreds of bytes), making program installation fast and inexpensive. Additionally, concise programs interpret few instructions, imposing very little interpretation overhead. We evaluate ASVMs against current proposals for network programming runtimes and show that ASVMs are more energy efficient by as much as 20%. We also evaluate ASVMs against hand built TinyOS applications and show that while interpretation imposes a significant execution overhead, the low duty cycles of realistic applications make the actual cost effectively unmeasurable.

1. INTRODUCTION

Wireless sensor networks have limited resources and tight energy budgets. These constraints make in-network processing a prerequisite for scalable and long-lived applications. However, as sensor networks are embedded in uncontrolled environments, a user often does not know exactly what the sensor data will look like, and so must be able to reprogram sensor network nodes after deployment. Proposals for domain specific languages — still an area of open investigation [5, 7, 19, 21, 23, 28] — present possible programming models for writing these programs. TinySQL queries, for example, declare how nodes should aggregate data as it flows to the root of a collection tree.

This wide range of programming abstractions has led to a similarly wide range of supporting runtimes, ranging from in-network query processors [19] to native thread libraries [28] to on-node script interpreters [5]. However, each is a vertically integrated solution, making them all mutually incompatible with each other. Additionally, they all make implementation assumptions or simplifications that lead to unnecessary inefficiencies.

Rather than propose a new programming approach to in-network processing, in this paper we propose an architecture for *implementing* a programming model's underlying runtime. We extend our prior work on the Maté virtual machine (a tiny bytecode interpreter) [15], generalizing its simple VM into an architecture for building application specific virtual machines (ASVMs). Our experiences showed that Maté's harsh limitations and complex instruction set precluded supporting higher level programming. By carefully relaxing some of these restrictions and allowing a user to customize both the instruction set and execution triggering events, ASVMs can support dynamically reprogramming for a wide range of application domains.

Introducing lightweight scripting to a network makes it easy to process data at, or very close to, its source. This processing can improve network lifetime by reducing network traffic, and can improve scalability by performing local operations locally. Similar approaches have appeared before in other domains. Active disks proposed pushing computation close to storage as a way to deal with bandwidth limitations [1], active networks argued for introducing in-network processing to the Internet to aid the deployment of new network protocols [27], and active services suggested processing at IP end points [2]. Following this nomenclature, we name the process of introducing dynamic computation into a sensor network *active sensor networking*. Of the prior efforts, active networking has the most similarity, but the differing goals and constraints of the Internet and sensor networks lead to very different solutions. We defer a detailed comparison of the two until Section 6.

Pushing the boundary toward higher level operations allows application level programs to achieve very high code density, which reduces RAM requirements, interpretation overhead, and propagation cost. However, a higher boundary can sacrifice flexibility: in the most extreme case, an ASVM has a single bytecode, "run program." Rather than answer the question of where the boundary should lie — a question whose answer depends on the application domain — ASVMs provide flexibility to an application developer, who can pick the right level of abstraction based on the particulars of a deployment.

Generally, however, we have found that very dense bytecodes do not sacrifice flexibility, because ASVMs are customized for the domain of interest. RegionsVM, presented in Section 4, is an ASVM designed for vehicle tracking with extensions for regions based operations [28]; typical vehicle tracking programs are on the order of seventy bytes long, 1/200th the size of the originally proposed regions implementation. A second ASVM we have built, QueryVM, supports an SQL interface to a sensor network at 5–20% lower energy usage than the TinyDB system [19], and also allows adding new aggregation functions dynamically.

This paper has two contributions. First, it shows a way to introduce a flexible boundary between dynamic and static sensor network code, enabling active sensor networking at a lower cost than prior approaches while simultaneously gaining improvements in safety and expressiveness. Second, this paper presents solutions to several technical challenges faced by such an approach, which include extensible type support, concurrency control, and code propagation. Together, we believe these results suggest a general methodology for designing and implementing runtimes for in-network processing.

In the next section, we describe background information relevant to this work, including mote network resource constraints, operating system structure, and the first version of Maté. From these observations, we derive three ways in which Maté is insufficient, establishing them as requirements for in-network processing runtimes to be effective. In Section 3, we present ASVMs, outlining their structure and decomposition. In Section 4 we evaluate ASVMs with a series of microbenchmarks, and compare ASVM-based regions and TinySQL to their original implementations. We survey related work in Section 5, discuss the implications of these results in Section 6, and conclude in Section 7.

2. BACKGROUND

ASVMs run on the TinyOS operating system, whose programming model affects their structure and implementation. The general operating model of TinyOS networks (very low duty cycle) and network energy constraints lead to both very limited node resources and underutilization of those resources. Maté is a prior, monolithic VM we developed for one particular application domain. From these observations, we derive a set of technical challenges for a runtime system to support active sensor networking.

2.1 TinyOS/nesC

TinyOS is a popular sensor network operating system designed for mote platforms. The nesC language [6], used to implement TinyOS and its applications, provides two basic abstractions: component based programming and low overhead, event driven concurrency.

Components are the units of program composition. A component has a set of *interfaces* it *uses*, and a set of interfaces it *provides*. A programmer builds an application by connecting interface users to providers. An interface can be *parameterized*. A component with a parameterized interface has many copies of the interface, distinguished by a parameter value (essentially, an array of the interface). Parameterized interfaces support runtime dispatch between a set of components. For example, the ASVM scheduler uses a parameterized interface to issue instructions: each instruction is an instance of the interface, and the scheduler dispatches on the opcode value.

TinyOS's event-driven concurrency model does not allow blocking operations. Calls to long-lasting operations, such as sending a packet, are typically split-phase: the call to begin the operation returns immediately, and the called component signals an event to the caller on completion. nesC programming binds these callbacks statically at compile time through nesC interfaces (instead of, e.g., using function pointers passed at run-time).

2.2 Mote Networks

As motes need to be able to operate unattended for months to years, robustness and energy efficiency are their dominant system requirements. Hardware resources are very limited, to minimize energy consumption. Current TinyOS motes have a 4–8MHz microcontroller, 4–10kB of data RAM, 60–128kB of program flash memory, and a radio with application-level data transmission rates of 1–20kB/s.

Energy limitations force long term deployments to operate at a very low utilization. Even though a mote has very limited resources, in many application domains some of those resources are barely used. For example, in the 2003 Great Duck Island deployment [26], motes woke up from deep sleep every five or twenty minutes, warmed up sensors for a second, and transmitted a single data packet with readings. During the warm-up second, the CPU was essentially idle. All in all, motes were awake 0.1% of the time, and when awake used 2% of their CPU cycles and network bandwidth. Although a mote usually does very little when awake, there can also be flurries of activity, as nodes receive messages to forward from routing children or link estimation updates from neighbors.

2.3 Maté v1.0

We designed and implemented the first version of Maté in 2002, based on TinyOS 0.6 (pre-nesC) [15]. At that time, the dominant hardware platform was the rene2 (the mica was just emerging), which had 1kB of RAM, 16kB of program memory and a 10kbps software controlled radio. Maté has a predefined set of three events it executes in response to. RAM constraints limited the code for a particular event handler to 24 bytes long (a single packet). In order to support network protocol implementations in this tiny amount of space, the VM had a complex instruction set open to inventive assembly programming but problematic as a compilation target.

2.4 Requirements

Maté's hard virtual/native boundary prevents it from being able to support a range of programming models. In particular, it fails to meet three requirements:

Flexibility: The Maté VM has very concise programs, but is designed for a single application domain. To provide support for in-network processing, a runtime must be flexible enough to be customized to a wide range of application domains. Supporting a range of application domains requires two forms of customization: the execution primitives of the VM (its instruction set), and the set of events it executes in response to. For example, data collection networks need to execute in response to a request to forward a packet up a collection tree (for suppression/aggregation), while a vehicle tracking network needs to execute in response to receiving a local broadcast from a neighbor.

Concurrency: By introducing a lightweight threading model on top of event-driven TinyOS, Maté provides a greatly simplified programming interface while enabling fine-grained parallelism. Limited resources and a constrained application domain allowed Maté to address the corresponding synchronization and atomicity issues by only having a single shared variable. This restriction is not suitable for all VMs. However, forcing explicit synchronization primitives into programs increases their length and places the onus of correctness on the programmer, who may not be an expert on concurrency. Instead, the runtime should manage concurrency automatically, running handlers race-free and deadlock-free while allowing safe parallelism.

Propagation: In Maté, handlers can explicitly forward code with the `forw` and `forwo` instructions. As every handler could fit in a single packet, these instructions were just a simple broadcast. One one hand, explicit code forwarding allows user programs to control their propagation, introducing additional flexibility; on the other, it requires every program to include propagation algorithms, which can be hard to tune and easy to write incorrectly. Maté's propagation data showed how a naive propagation policy can easily saturate a network, rendering it unresponsive and wasting energy. As not all programming models can fit their programs in a single packet, a runtime needs to be able to handle larger data images (e.g., between 20 and 512 bytes), and should provide an efficient but rapid propagation service.

Our prior work on the Trickle [17] algorithm deals with one part of the propagation requirement, proposing a control algorithm to quickly yet efficiently detect when code updates are needed. The propagation results in that work assumed code could fit in a single packet and just broadcasts updates three times. This leaves the need for a protocol to send code updates for larger programs: we present our solution to this problem in Section 3.4.

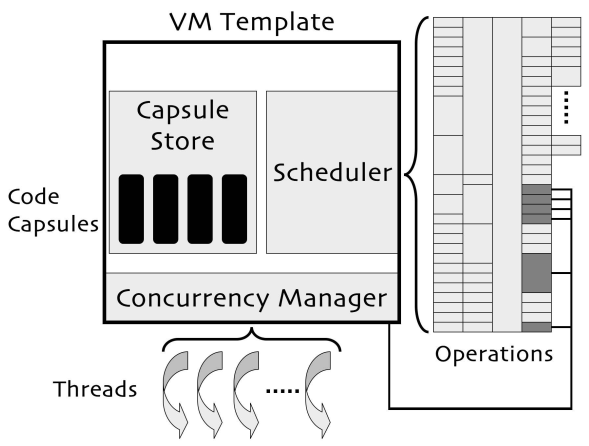

Figure 1: The ASVM architecture.

To provide useful systems support for a wide range of programming models, a runtime must meet these three requirements without imposing a large energy burden. Flexibility requires a way to build customized VMs — a VM generator — so a VM can be designed for an application domain. The next section describes our application specific virtual machine (ASVM) architecture, designed to take this next step.

3. DESIGN

Figure 1 shows the ASVM functional decomposition. ASVMs have three major abstractions: *handlers*, *operations*, and *capsules*. Handlers are code routines that run in response to system events, operations are the units of execution functionality, and capsules are the units of code propagation. ASVMs have a threaded execution model and a stack-based architecture.

The components of an ASVM can be separated into two classes: the *template*, which every ASVM includes, and *extensions*, the application-specific components that define a particular ASVM. The template includes a scheduler, concurrency manager, and capsule store. The scheduler executes runnable threads in a FIFO round-robin fashion. The concurrency manager controls what threads are runnable, ensuring race-free and deadlock-free handler execution. The capsule store manages code storage and loading, propagating code capsules and notifying the ASVM when new code arrives.

Building an ASVM involves connecting handlers and operations to the template. Each handler is for a specific system event, such as receiving a packet. When that event occurs, the handler triggers a thread to run its code. Generally, there is a one-to-one mapping between handlers and threads, but the architecture does not require this to be the case. The concurrency manager uses

```
interface Bytecode {
/* The instr parameter is necessary for primitives
   with embedded operands (the operand is instr
   - opcode). Context is the executing thread. */

  command result_t execute(uint8_t instr,
                           MateContext* context);
  command uint8_t byteLength();
}
```

Figure 2: The nesC Bytecode interface, which all operations provide.

a conservative, flow insensitive and context insensitive program analysis to provide its guarantees.

The set of operations an ASVM supports defines its instruction set. Just as in Maté, instructions that encapsulate split-phase TinyOS abstractions provide a blocking interface, suspending the executing thread until the split-phase call completes. Operations are defined by the Bytecode nesC interface, shown in Figure 2, which has two commands: `execute` and `byteLength`. The former is how a thread issues instructions, while the latter lets the scheduler correctly control the program counter. Currently, ASVMs support three languages: TinyScript, motlle, and TinySQL, which we present in Sections 4.3 and 4.4.

There are two kinds of operations: *primitives*, which are language specific, and *functions*, which are language independent. The distinction between primitives and functions is an important part of providing flexibility. An ASVM supports a particular language by including the primitives it compiles to, while a user tailors an ASVM to a particular application domain by including appropriate functions and handlers. For functions to work in any ASVM, and correspondingly any language, ASVMs need a minimal common data model. Additionally, some functions (e.g., communication) should be able to support language specific data types without knowing what they are. These issues are discussed in Section 3.1. In contrast, primitives can assume the presence of data types and can have embedded operands. For example, conditional jumps and pushing a constant onto the operand stack are primitives, while sending a packet is a function.

The rest of this section presents the ASVM data model and the three core components of the template (scheduler, concurrency manager, and capsule store). It concludes with an example of building an ASVM for region programming.

3.1 Data Model

An ASVM has a stack architecture. Each thread has an operand stack for passing data between operations. The template does not provide any program data storage beyond the operand stack, as such facilities are language specific, and correspondingly defined by primitives. The architecture defines a minimal set of standard simple operand types as 16-bit values (integers and sensor readings); this is enough for defining many useful language-independent functions.

Operation	Width	Name	Operand Bits	Description
rand	1	rand	0	Random 16-bit number
pushc6	1	pushc	6	Push a constant on stack
2jumps10	2	jumps	10	Conditional jump

Table 1: Three example operations: rand is a function, pushc6 and 2jumps10 are primitives.

However, to be useful, communication functions need more elaborate types. For example, the `bcast` function, which sends a local broadcast packet, needs to be able to send whatever data structures its calling language provides. The function takes a single parameter, the item to broadcast, which a program pushes onto the operand stack before invoking it. To support these kinds of functions, languages must provide serialization support for their data types. This allows `bcast`'s implementation to pop an operand off the stack and send a serialized representation with the underlying TinyOS `sendMsg()` command. When another ASVM receives the packet, it converts the serialized network representation back into a VM representation.

3.2 Scheduler: Execution

The core of an ASVM is a simple FIFO thread scheduler. This scheduler maintains a run queue, and interleaves execution at a very fine granularity (a few operations). The scheduler executes a thread by fetching its next bytecode from the capsule store and dispatching to the corresponding operation component through a nesC parameterized interface. The parameter is an 8-bit unsigned integer: an ASVM can support up to 256 distinct operations at its top-level dispatch. As the scheduler issues instructions through nesC interfaces, their selection and implementation is completely independent of the template and the top level instruction decode overhead is constant.

Primitives can have embedded operands, which can cause them to take up additional opcode values. For example, the `pushc6` primitive, which pushes a 6-bit constant onto the operand stack, has six bits of embedded operand and uses 64 opcode slots. Some primitives, such as jump instructions, need embedded operands longer than 8 bits. Primitives can therefore be more than one byte wide.

When the ASVM toolchain generates an instruction set, it has to know how many bits of embedded operand an operation has, if any. Similarly, when the toolchain's assembler transforms compiled assembly programs into ASVM-specific opcodes, it has to know how wide instructions are. All operations follow this naming convention:

```
[width]<name>[operand]
```

Width and operand are both numbers, while name is a string. Width denotes how many bytes wide the operation is (this corresponds to the `byteWidth` command of the Bytecode interface), while operand is how many bits of embedded operand the operation has. If an operation does not have a width field, it defaults to 1; if it does not have an operand field, it defaults to zero. Table 1 shows three example operations.

Beyond language independence, the function/primitive distinction also determines which operations can be called indirectly. Some languages have first class functions or function pointers: a program must be able to invoke them dynamically, rather than just statically through an instruction. To support this functionality, the scheduler maintains a function identifier to function mapping. This allows functions to be invoked through identifiers that have been stored in variables. For example, when the motlle language calls a function, it pushes a function ID onto the operand stack and issues the `mcall` instruction, which creates a call stack frame and invokes the function through this level of indirection.

3.3 Concurrency Manager: Parallelism

Handlers run in response to system events, and the scheduler allows multiple handler threads to run concurrently. In languages with shared variables, this can easily lead to race conditions, which are very hard to diagnose and detect in embedded devices. The common solution to provide race free execution is explicit synchronization written by the programmer. However, explicit synchronization operations increase program size and complexity: the former costs energy and RAM, the latter increases the chances that, after a month of deployment, a scientist discovers that all of the collected data is invalid and cannot be trusted. One common case where ASVMs need parallelism is network traffic, due the limited RAM available for queuing. One handler blocking on a message send should not prevent handling message receptions, as their presence on the shared wireless channel might be the reason for the delay.

The concurrency manager of the ASVM template supports race free execution through implicit synchronization based on a handler's operations. An operation component can register with the concurrency manager (at compile time, through nesC wiring) to note that it accesses a shared resource. When the ASVM installs a new capsule, the concurrency manager runs a conservative, context-insensitive and flow-insensitive analysis to determine which shared resources each handler accesses. This registration with the concurrency manager is entirely optional. If a language prefers explicit synchronization, then its operations can not declare shared resources, and the concurrency manager will not limit parallelism.

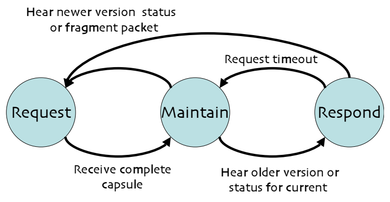
Hear newer version status or fragment packet
Request timeout
Request
Maintain
Respond
Receive complete capsule
Hear older version or status for current

Figure 3: ASVM capsule propagation state machine.

When a handler event occurs, the handler's implementation submits a run request to the concurrency manager. The concurrency manager only allows a handler to run if it can exclusively access all of the shared resources it needs. The concurrency manager enforces two-phase locking: when it starts executing, the handler's thread has to hold all of the resources it may need, but can release them as it executes. When a handler completes (executes the `halt` operation), its thread releases all of the held resources. Releases during execution are explicit operations within a program. If a thread accesses a resource it does not hold (e.g., it incorrectly released it) the VM triggers an error. Two phase locking precludes deadlocks, so handlers run both race free and deadlock free.

When new code arrives, a handler may have variables in an inconsistent state. Waiting for every handler to complete before installing a new capsule is not feasible, as the update may, for example, be to fix an infinite loop bug. Therefore, when new code arrives, the concurrency manager reboots the ASVM, resetting all variables.

The implicit assumption in this synchronization model is that handlers are short running routines that do not hold onto resources for very long. As sensor network nodes typically have very low utilization, this is generally the case. However, a handler that uses an infinite loop with a call to `sleep()`, for example, can block all other handlers indefinitely. Programming models and languages that prefer this approach can use explicit synchronization, as described above.

3.4 Capsule Store: Propagation

Field experience with current sensor networks has shown that requiring physical contact can be a cause of many node failures [25]; network programming is critical. Thus, ASVMs must provide reliable code propagation. As mentioned earlier (Section 2.4), Maté's explicit code forwarding mechanism is problematic. As demonstrated in our work on Trickle [17], the cost of propagation is very low compared to the accompanying control traffic, so selective dissemination enables few energy gains. The ASVM template's capsule store therefore follows a policy of propagating new code to every node. Rather than selective propagation, ASVMs use a policy of selective execution: everyone has the code, but only some nodes execute it.

Trickle is a suppression algorithm for detecting when nodes need code updates. The algorithm dynamically scales its suppression intervals to rapidly detect inconsistencies but sends few packets when the network is consistent. Trickle does not define how code itself propagates, as the protocol greatly depends on the size of the data item. Deluge, for example, transfers entire TinyOS binaries, and so uses a cluster formation algorithm to quickly propagate large amounts of data [11]. In the Maté virtual machine, with its single packet programs, propagation was just a simple local broadcast.

ASVM programs are between these two extremes. As they are on the order of one to twenty packets long, Deluge is too heavy-weight a protocol, and simple broadcasts are not sufficient. To propagate code, the ASVM capsule store maintains three network *trickles* (independent instances of the Trickle algorithm):

- **Version packets**, which contain the 32-bit version numbers of all installed capsules,
- **Capsule status packets**, which describe what fragments a mote needs (essentially, a bitmask), and
- **Capsule fragments**, which are pieces of a capsule.

An ASVM can be in one of three states: maintain (exchanging version packets), request (sending capsule status packets), or respond (sending fragments). Nodes start in the maintain state. Figure 3 shows the state transition diagram. The transitions prefer requesting over responding; a node will defer forwarding capsules until it thinks it is completely up to date.

Each type of packet (version, capsule status, and capsule fragment) is a separate network trickle. For example, a capsule fragment transmission can suppress other fragment transmissions, but not version packets. This allows meta-data and data exchanges to occur concurrently. Trickling fragments means that code propagates in a slow and controlled fashion, instead of as quickly as possible. This is unlikely to significantly disrupt any existing traffic, and prevents network overload. We show in Section 4.2 that because ASVM programs are small they propagate rapidly across large multi-hop networks.

3.5 Building an ASVM

Building an ASVM and scripting environment requires specifying three things: a language, functions, and handlers. Figure 4 shows the description file for RegionsVM, an ASVM that supports programming with regions [28]. We evaluate RegionsVM versus a native regions implementation in Section 4.2. The final HANDLER line specifies that this ASVM executes in response to only one event, when the ASVM boots (or reboots). ASVMs can include multiple handlers, which usually leads to multiple threads; RegionsVM, following the regions programming model of a single execution context, only includes one, which runs when the VM reboots. From this file, the toolchain generates TinyOS source code implementing the ASVM, and the Java classes its assembler uses to map assembly to ASVM opcodes.

```
<VM NAME="KNearRegions" DIR="apps/RegionsVM">

<LANGUAGE NAME="tinyscript">

<FUNCTION NAME="send">
<FUNCTION NAME="mag">
<FUNCTION NAME="cast">
<FUNCTION NAME="id">
<FUNCTION NAME="sleep">
<FUNCTION NAME="KNearCreate">
<FUNCTION NAME="KNearGetVar">
<FUNCTION NAME="KNearPutVar">
<FUNCTION NAME="KNearReduceAdd">
<FUNCTION NAME="KNearReduceMaxID">
<FUNCTION NAME="locx">
<FUNCTION NAME="locy">

<HANDLER NAME="Boot">
```

Figure 4: Minimal description file for the RegionsVM. Figure 7 contains scripts for this ASVM.

3.6 Active Sensor Networking

In order to support in-network processing, ASVMs must be capable of operating on top of a range of single-hop and multi-hop protocols. Currently, the ASVM libraries support four concrete networking abstractions through functions and handlers: single hop broadcasts, any-to-one routing, aggregated collection routing, and abstract regions [28]. Based on our experiences writing library ASVM components for these protocols — 80 to 180 nesC statements — including additional ones as stable implementations emerge should be simple and painless.

4. EVALUATION

We evaluate whether ASVMs efficiently satisfy the requirements presented in Section 2: concurrency, propagation, and flexibility. We first evaluate the three requirements through examples and microbenchmarks, then evaluate overall application level efficiency in comparison to alternative approaches.

In our microbenchmarks, cycle counts are from a mica node, which has a 4MHz 8-bit microcontroller, the ATMega103L; some members of the mica family have a similar MCU at a faster clock rate (8MHz). Words are 16 bits and a memory access takes two cycles: as it is an 8-bit architecture, moving a word (or pointer) between memory and registers takes 4 clock cycles.

4.1 Concurrency

We measured the overhead of ASVM concurrency control, using the cycle counter of a mica mote. Table 2 summarizes the results. All values are averaged over 50

Operation	Cycles	Time (μs)
Lock	32	8
Unlock	39	10
Run	1077	269
Analysis	15158	3790

Table 2: Synchronization Overhead. Lock and unlock are acquiring or releasing a shared resource. Run is moving a thread to the run queue, obtaining all of its resources. Analysis is a full handler analysis.

	Mean	Std. Dev.	Worst
Mote Retasking	20.8s	7.4s	85.8s
Network Retasking	39.5s	10.4s	85.8s
Vector Packets Sent	1.0	1.1	13
Status Packets Sent	3.2	2.4	19
Fragment Packets Sent	3.0	2.5	22
Total Packets Sent	7.3	3.6	30

Table 3: Propagation data. Mote retasking is across all motes in all experiments. Network retasking is the retasking times in all the experiments, based on the time for the last mote to reprogram. The Packets Sent are all on a per-mote basis.

samples. These measurements were on an ASVM with 24 shared resources and a 128 byte handler. Locking and unlocking resources take on the order of a few microseconds, while a full program analysis for shared resource usage takes under a millisecond, approximately the energy cost of transmitting four bits.

These operations enable the concurrency manager to provide race-free and deadlock-free handler parallelism at a very low cost. By using implicit concurrency management, an ASVM can prevent many race condition bugs while keeping programs short and simple.

4.2 Propagation

To evaluate code propagation, we deployed an ASVM on a 71 mote testbed in Soda Hall on the UC Berkeley campus. The network topology was approximately eight hops across, with four hops being the average node distance. We used the standard ASVM propagation parameters.[1] We injected a one hundred byte (four fragment) handler into a single node over a wired link. We repeated the experiment fifty times, resetting the nodes after each test to restore the trickle timers to their stable values (maximums).

[1]Status and version packets have a τ range of one second to twenty minutes, and a redundancy constant of 2. Fragments use Trickle for suppression, but operate with a fixed window size of one second repeating twice, and have a redundancy constant of 3. The request timeout was five seconds.

	Native	RegionsVM
Code (Flash)	19kB	39kB
Data (RAM)	2775B	3017B
Transmitted Program	19kB	71B

Table 4: Space utilization of native and RegionsVM regions implementations (bytes).

```
buffer packet;

bclear(packet);                  bpush3 3
                                 bclear
packet[0] = light();             light
                                 pushc6 0
                                 bpush3 3
                                 bwrite
send(packet);                    bpush3 3
                                 send
```

(a) TinyScript (b) ASVM Bytecodes

Figure 5: TinyScript function invocation on a simple sense and send loop. The operand stack passes parameters to functions. In this example, the scripting environment has mapped the variable "packet" to buffer three. The compiled program is nine bytes long.

Table 3 summarizes the results. On the average, the network reprogrammed in forty seconds, and the worst case was eighty-five seconds. To achieve this rate, each node, on the average, transmitted seven packets, a total of five hundred transmissions for a seventy node network. The worst case node transmitted thirty packets. Checking the traces, we found this mote was the last one to reprogram in a particular experiment, and seems to have suffered from bad connectivity or inopportune suppressions. It transmitted eleven version vectors and nineteen status packets, repeatedly telling nodes around it that it needed new code, but not receiving it. With the parameters we used, a node in a stable network sends at most three packets per hour.

To evaluate the effect ASVM code conciseness has on propagation efficiency, we compare the retasking cost of the native implementation proposed for regions versus the cost of retasking a system with RegionsVM. In the regions proposal, users write short nesC programs for a single, synchronous "fiber" that compile to a TinyOS binary. Reprogramming the network involves propagating this binary into the network. As regions compiles to native TinyOS code, it has all of the safety issues of not having a protection boundary.

Abstract regions is designed to run in TOSSIM, a simulator for TinyOS [16]. Several assumptions in its protocols — such as available bandwidth — prevent it from running on motes, and therefore precluded us from measuring energy costs empirically. However, by modifying a few configuration constants the two implementations share, we were able to compile them and measure RAM utilization and code size. Table 4 shows the results. The fiber's stack accounts for 512 bytes of the native runtime RAM overhead.

An ASVM doubles the size of the TinyOS image, but this is a one time cost for a wide range of regions programs. Reprogramming the native implementation requires sending a total of nineteen kilobytes: reprogramming the RegionsVM implementation requires sending a seventy byte ASVM handler, less than 0.5% of the size of the binary. Additionally, handlers run in the sandboxed virtual environment, and benefit from all of its safety guarantees. If, after many retaskings, the user decides that the particular networking abstractions an ASVM provides are not quite right, a new one can always be installed using binary reprogramming.

Deluge is the standard TinyOS system for disseminating binary images into a network [11]. Reported experimental results on a network similar to the one we used in our propagation experiments state that disseminating 11kB takes 10,000 transmissions: disseminating the 19kB of the native implementation would take approximately 18,000 transmissions. In contrast, from the data in Table 3, a RegionsVM program takes fewer than five hundred transmissions, less than 3% of the cost, while providing safety. The tradeoff is that programs are interpreted bytecodes instead of native code, imposing a CPU energy overhead. We evaluate this cost in Sections 4.5-4.6, using microbenchmarks and an application level comparison with TinyDB.

4.3 Flexibility: Languages

ASVMs currently support three languages, TinyScript, motlle and TinySQL queries. We discuss TinySQL in Section 4.4, when presenting QueryVM.

TinyScript is a bare-bones language that provides minimalist data abstractions and control structures. It is a BASIC-like imperative language with dynamic typing and a simple data buffer type. TinyScript does not have dynamic allocation, simplifying concurrency resource analysis. The resources accessed by a handler are the union of all resources accessed by its operations. TinyScript has a one to one mapping between handlers and capsules. Figure 5 contains sample TinyScript code and the corresponding assembly it compiles to.

Motlle (MOTe Language for Little Extensions) is a dynamically-typed, Scheme-inspired language with a C-like syntax. Figure 6 shows an example of heavily commented mottle code. The main practical difference with TinyScript is a much richer data model: motlle supports vectors, lists, strings and first-class functions. This allows significantly more complicated algorithms to be expressed within the ASVM, but the price is that accurate data analysis is no longer feasible on a mote. To preserve safety, motlle serializes thread execution by reporting to the concurrency manager that all handlers access the same shared resource. Motlle programs are transmitted in a single capsule which contains all handlers. Motlle-based ASVMs therefore do not support incremental changes to running programs.

```
settimer0(500);       // Epoch is 50s
mhop_set_update(100); // Update tree every 100s

// Define Timer0 handler
any timer0_handler() { // 'any' is the result type
  // 'mhop_send' sends a message up the tree
  // 'encode' encodes a message
  // 'next_epoch' advances to the next epoch
  //     (snooped value may override this)
  send(encode(vector(next_epoch(), id(), parent(),
                                    temp())));
}

// Intercept and Snoop run when a node forwards
// or overhears a message.
// Intercept can modify the message (aggregation).
// Fast-forward epoch if we're behind
any snoop_handler() heard(snoop_msg());
any intercept_handler() heard(intercept_msg());
any heard(msg) {
  // decode the first 2 bytes of msg into an integer.
  vector v = decode(msg, vector(2));

  // 'snoop_epoch' advances epoch if needed
  snoop_epoch(v[0]);
}
```

Figure 6: A motlle data collection query: return node id, routing tree parent and temperature every 50s.

4.4 Flexibility: Applications

We have built two sample ASVMs, RegionsVM and QueryVM. RegionsVM, designed for vehicle tracking, presents the abstract regions programming abstraction of MPI-like reductions over shared tuple spaces. Users write programs in TinyScript, and RegionsVM includes ASVM functions for the basic regions library; we obtained the regions source code from its authors. Figure 7 shows regions pseudocode proposed by Welsh at al. [28] next to actual TinyScript code that is functionally identical (it invokes all of the same library functions). The nesC components that present the regions library as ASVM functions are approximately 400 lines of nesC code.

QueryVM is designed for periodic data collection using the aggregated collection routing abstraction mentioned in Section 3.6. QueryVM provides a TinySQL programming interface, similar to TinyDB, presenting a sensor network as a streaming database. TinySQL's main extension to SQL is a 'sample period' at which the query is repeated. TinySQL supports both simple data collection and aggregate queries such as

SELECT AVG(temperature) INTERVAL 50s

to measure the average temperature of the network. The latter allow in-network processing to reduce the amount of traffic sent, by aggregating as nodes route data [20].

In our implementation, TinySQL compiles to motlle code for the handlers that the aggregation collection tree library provides. This has the nice property that, in ad-

```
location = get_location();
/* Get 8 nearest neighbors */
region = k_nearest_region_create(8);

while(true) {
  reading = get_sensor_reading();

  /* Store local data as shared variables */
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
    /* ID of the node with the max value */
    max_id = region.reduce(OP_MAXID, reading_key);

    /* If I am the leader node...  */
    if (max_id == my_id) {
      sum = region.reduce(OP_SUM, reading_key);
      sum_x = region.reduce(OP_SUM, reg_x_key);
      sum_y = region.reduce(OP_SUM, reg_y_key);
      centroid.x = sum_x / sum;
      centroid.y = sum_y / sum;
      send_to_basestation(centroid);
    }
  }
  sleep(periodic_delay);
}
```

(a) Regions Pseudocode

```
!!  Create nearest neighbor region
KNearCreate();

for i = 1 until 0
  reading = int(mag());

  !!  Store local data as shared variables
  KNearPutVar(0, reading);
  KNearPutVar(1, reading * LocX());
  KNearPutVar(2, reading * LocY());

  if (reading > threshold) then
    !!  ID of the node with the max value
    max_id = KNearReduceMaxID(0);

    !!  If I am the leader node
    if (max_id = my_id) then
      sum = KNearReduceAdd(0);
      sum_x = KNearReduceAdd(1);
      sum_y = KNearReduceAdd(2);
      buffer[0] = sum_x / sum;
      buffer[1] = sum_y / sum;
      send(buffer);
    end if
  end if
  sleep(periodic_delay);
next i
```

(b) TinyScript Code

Figure 7: Regions Pseudocode and Corresponding TinyScript. The pseudocode is from "Programming Sensor Networks Using Abstract Regions." The TinyScript program on the right compiles to 71 bytes of binary code.

dition to TinySQL, QueryVM also supports writing new attributes and network aggregates in motlle. In contrast, TinyDB is limited to the set of attributes and aggregates compiled into its binary.

4.5 Efficiency: Microbenchmarks

Our first evaluation of ASVM efficiency is a series of microbenchmarks of the scheduler. We compare ASVMs to Maté, a hand-tuned and monolithic implementation.

Following the methodology we used in Maté [15], we measured the bytecode interpretation overhead an ASVM imposes by writing a tight loop and counting how many times it ran in five seconds on a mica mote. The loop accessed a shared variable (which involved lock checks through the concurrency manager). An ASVM can issue just under ten thousand instructions per second on a 4MHz mica, i.e., roughly 400 cycles per instruction. The ASVM decomposition imposes a 6% overhead over a similar loop in Maté, in exchange for handler and instruction set flexibility as well as race-free, deadlock-free parallelism.

We have not optimized the interpreter for CPU efficiency. The fact that high-level operations dominate program execution [15], combined with the fact that CPUs in sensor networks are generally idle, makes this overhead acceptable, although decreasing it with future work is of course desirable. For example, a KNearReduce function in the RegionsVM sends just under forty packets, and its ASVM scripting overhead is approximately 600 CPU cycles, the energy overhead is less than 0.03%. However, a cost of 400 cycles per bytecode means that implementing complex mathematical codes in an ASVM is inefficient; if an application domain needs significant processing, it should include appropriate operations.

	None	Operation	Script
Iteration	16.0	16.4	62.2
Sort Time	-	0.4	46.2

Table 5: Execution time of three scripts, in milliseconds. None is the version that did not sort, operation is the version that used an operation, while script is the version that sorted in script code.

To obtain some insight into the tradeoff between including functions and writing operations in script code, we wrote three scripts. The first script is a loop that fills an array with sensor readings. The second script fills the array with sensor readings and sorts the array with an operation (`bufsorta`, which is an insertion sort). The third script also insertion sorts the array, but does so in TinyScript, rather than using an operation. To measure the execution time of each script, we placed it in a 5000 iteration loop and sent a UART packet at script start and end. Table 5 shows the results. Sorting the array with script code takes 115 times as long as sorting with an operation, and dominates script execution time. Interpretation is inefficient, but pushing common and expen-

```
// Initialise the operator
expdecay_make = fn (bits) vector(bits, 0);
// Update the operator (s is result from make)
expdecay_get = fn (s, val)
  // Update and return the average (s[0] is BITS)
  s[1] = s[1] - (s[1] >> s[0]) + (attr() >> s[0]);
```

Figure 8: An exponentially decaying average operator for TinySQL, in motlle.

Name	TinySQL
Simple	`SELECT id,parent,temp INTERVAL 50s`
Conditional	`SELECT id, expdecay(humidity, 3) WHERE parent > 0 INTERVAL 50s`
SpatialAvg	`SELECT AVG(temp) INTERVAL 50s`

Table 6: The three queries used to evaluate data collection implementations. TinyDB does not directly support time-based aggregates such as `expdecay`, so in TinyDB we omit the aggregate.

sive operations into native code with functions minimizes the amount of interpretation. Section 4.7 shows that this flexible boundary, combined with the very low duty cycle common to sensor networks, leads to interpretation overhead being a negligible component of energy consumption for a wide range of applications.

4.6 Efficiency: Application

QueryVM is a motlle-based ASVM designed to support the execution of TinySQL data collection queries. Our TinySQL compiler generates motlle code from queries such as those shown in Table 6; the generated code is responsible for timing, data collection message layout and how to process or aggregate data on each hop up the routing tree. The code in Figure 6 is essentially the same as that generated for the Simple query. Users can write new attributes or operators for TinySQL using snippets of motlle code. For instance, Figure 8 shows two lines of motlle code to add an exponentially-decaying average operator, which an example in Table 6 uses.

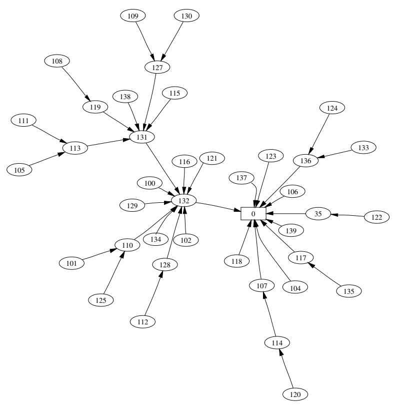

Figure 9: Tree topology used in QueryVM/TinyDB experiments. The square node is the tree root.

	Size (bytes)		Energy (mW)		Yield	
Query	TinyDB	VM	TinyDB	VM	TinyDB	VM
Simple	93	105	5.6	4.5	73%	74%
Conditional	124	167	4.2	4.0	65%	79%
SpatialAvg	62	127	3.3	3.1	46%	55%

Table 7: Query size, power consumption and yield in TinyDB and QueryVM. Yield is the percentage of expected results received.

TinySQL query results abstract the notion of time into an *epoch*. Epoch numbers are a logical time scheme that are included in query results and help support aggregation. QueryVM includes functions and handlers to support multi-hop communication, epoch handling and aggregation. QueryVM programs can use the same tree based collection layer, MintRoute [29], that TinyDB uses. QueryVM includes epoch-handling primitives to avoid replicating epoch-handling logic in every program (see usage in Figure 6). Temporal or spatial (across nodes) averaging logic can readily be expressed in motlle, but including common aggregates in QueryVM reduces program size and increases execution efficiency.

We evaluate QueryVM's efficiency by comparing its power draw to the TinyDB system on the three queries shown in Table 6. To reflect the power draw of a real deployment, we enabled low-power listening in both implementations. In low data rate networks — such as periodic data collection — low power listening can greatly improve network lifetime [22, 26]. At this level of utilization, packet length becomes an important determinant of energy consumption, so we matched the size of routing control packets between QueryVM and TinyDB. However, TinyDB's query result packets are still approximately 20 bytes longer than QueryVM's. On mica2 motes, this means that TinyDB will spend an extra 350μJ for each packet received, and 625μJ for each packet sent.

We ran the queries on a network of 40 mica2 motes spread across the ceiling of an office building. Motes had the mts400 weather board from Crossbow Technologies. Environmental changes can dynamically alter adhoc routing trees (e.g., choosing a 98% link over a 96% link), changing the forwarding pattern and greatly affecting energy consumption. These sorts of changes make experimental repeatability and fair comparisons unfeasible. Therefore, we used a static, stable tree in our experiments, to provide an even basis for comparison across the implementations. We obtained this tree by running the routing algorithm for a few hours, extracting the parent sets, then explicitly setting node parents to this topology, shown in Figure 9. Experiments run on adaptive trees were consistent with the results presented below.

We measured the power consumption of a mote with a single child, physically close to the root of the multihop network. Its power reflects a mote that overhears a lot of traffic but which sends relatively few messages (a

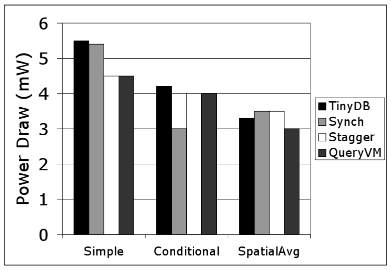

Figure 10: Power consumption of TinyDB, QueryVM, and nesC implementations. Synch is the nesC implementation when nodes start at the same time. Stagger is when the nodes start times are staggered.

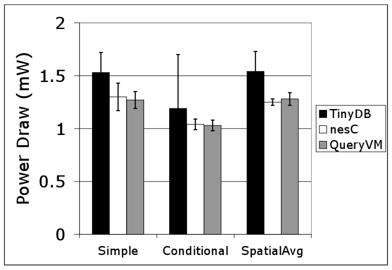

Figure 11: Average power draw measurements in a two node network. For the Conditional query, the monitored node has parent = 0, so sends no packets. The error bars are the standard deviation of the per-interval samples.

common case). In each of the queries, a node sends a data packet every 50 seconds, and the routing protocol sends a route update packet every two epochs (100 seconds). We measured the average power draw of the instrumented node over 16 intervals of 100 seconds, sampling at 100 Hz (10,000 instantaneous samples).

Table 7 presents the results from these experiments. For the three sample queries, QueryVM consumes 5% to 20% less energy than TinyDB. However, we do not believe all of this improvement to be fundamental to the two approaches. The differences in yield mean that the measured mote is overhearing different numbers of messages — this increases QueryVM's power draw. Conversely, having larger packets increases TinyDB's power draw — based on the 325μJ per-packet cost, we estimate a cost of 0.2–0.5mW depending on the query and how well the measured mote hears more distant motes.

However, these are not the only factors at work, as shown by experiments with a native TinyOS implementation of the three queries. We ran these native implementations in two scenarios. In the first scenario, we booted all of the nodes at the same time, so their operation was closely synchronized. In the second, we staggered node boots over the fifty second sampling interval. Figure 10 shows the power draw for these two scenarios, alongside that of TinyDB and QueryVM. In the synchronized case, yields for the native implementations varied between 65% and 74%, in the staggered case, yields were between 90% and 97%. As these results show, details of the timing of transmissions have major effects on yield and power consumption. To separate these networking effects from basic system performance, Section 4.7 repeats our experiments in a two-node network.

4.7 Efficiency: Interpretation

In our two-node experiments, the measured mote executes the query and sends results, and the second mote is a passive base station. As the measured node does not forward any packets or contend with other transmitters, its energy consumption is the cost of query execution and reporting. The extra cost of sending TinyDB's larger result packets is negligible (.01mW extra average power draw). We ran these experiments longer than the full network ones: rather than 16 intervals of length 100 seconds (25 minutes), we measured for 128 intervals (3.5 hours).

The results, presented in Figure 11, show that QueryVM has a 5–20% energy performance improvement over TinyDB. Even though an ASVM based on reusable software components and a common template, rather than a hand-coded, vertically integrated system, QueryVM imposes less of an energy burden on a deployment. In practice though, power draw in a real network is dominated by networking costs — QueryVM's 0.25mW advantage in Figure 11 would give at most 8% longer lifetime based on the power draws of Figure 10.

To determine where QueryVM's power goes, we compared it to four hand coded TinyOS programs. The first program did not process a query: it just listened for messages and handled system timers. This allows us to distinguish the cost of executing a query from the underlying cost of the system. The other three were the nesC implementations of the queries used for Figure 10. They allow us to distinguish the cost of executing a query itself from the overhead an ASVM runtime imposes. The basic system cost was 0.76 mW. Figure 11 shows the comparison between QueryVM and a hand-coded nesC implementation of the query. The queries cost 0.28–0.54 mW, and the cost of the ASVM is negligible.

This negligible cost is not surprising: for instance, for the conditional query, QueryVM executes 49 instructions per sample period, which will consume approximately 5ms of CPU time. Even on a mica2 node, whose CPU power draw is a whopping 33 mW due to an external oscillator (other platforms draw 3–8 mW), this corresponds to an average power cost of 3.3μW. In the 40 node network, the cost of snooping on other node's results will increase power draw by another 20μW. Finally, QueryVM sends viral code maintenance messages every 100 minutes (in steady state), corresponding to an average power draw of 1.6μW.

From the results in Table 7, with a power consumption of 4.5mW, a pair of AA batteries (2700mAh, of which approximately two thirds is usable by a mote) would last for 50 days. By lowering the sample rate (every fifty seconds is a reasonably high rate) and other optimizations, we believe that lifetimes of three months or more are readily achievable. Additionally, the energy cost of ASVM interpretation is a negligible portion of the whole system energy budget. This suggests that ASVM-based active sensor networking can be a realistic option for long term, low-duty-cycle data collection deployments.

5. RELATED WORK

The Maté virtual machine [15] forms the basis of the ASVM architecture. ASVMs address three of Maté's main limitations: flexibility, concurrency, and propagation. SensorWare [5] is another proposal for programming nodes using an interpreter: it proposes using Tcl scripts. For the devices SensorWare is designed for — iPAQs with megabytes of RAM — the verbose program representation and on-node Tcl interpreter can be acceptable overheads: on a mote, however, they are not.

SOS is a sensor network operating system that supports dynamic native code updates through a loadable module system [8]. This allows small and incremental binary updates, but requires levels of function call indirection. SOS therefore sits between the extremes of TinyOS and ASVMs, where its propagation cost is less than TinyOS and greater than ASVMs, and its execution overhead is greater than TinyOS but less than ASVMs. By using native code to achieve this middle ground, SOS cannot provide all of the safety guarantees that an ASVM can. Still, the SOS approach suggests ways in which ASVMs could dynamically install new functions.

The Impala middleware system, like SOS, allows users to dynamically install native code modules [18]. However, unlike SOS, which allows modules to both call a kernel and invoke each other, Impala limits modules to the kernel interfaces. Like ASVMs, these interfaces are event driven, and bear a degree of similarity to Maté. Unlike ASVMs, however, Impala does not provide general mechanisms to change its triggering events, as it is designed for a particular application domain, ZebraNet [13].

Customizable and extensible abstraction boundaries, such as those ASVMs provide, have a long history in operating systems research. Systems such as scheduler activations [3] show that allowing applications to cooperate with a runtime through rich boundaries can greatly improve application performance. Operating systems such as exokernel [14] and SPIN [4] take a more aggressive approach, allowing users to write the interface and improve performance through increased control. In sensor networks, performance — the general goal of *more*, whether it be bandwidth, or operations per second — is rarely a primary metric, as low duty cycles make resources plentiful. Instead, robustness and energy efficiency are the important metrics.

ANTS, PLAN, and Smart Packets are example systems that bring active networking to the Internet. Although all of them made networks dynamically programmable, each system had different goals and research foci. ANTS focuses on deploying protocols in a network, PLANet explores dealing with security issues through language design [10, 9], and Smart Packets proposes active networking as a management tool [24]. ANTS uses Java, while PLANet and Smart Packets use custom languages (PLAN and Sprocket, respectively). Based on an Internet communication and resource model, many of the design decisions these systems made (e.g., using a JVM) are unsurprisingly not well suited to mote networks. One distinguishing characteristic in sensor networks is their lack of strong boundaries between communication, sensing, and computation. Unlike in the Internet, where data generation is mostly the province of end points, in sensor networks every node is both a router and a data source.

Initial mote deployment experiences have demonstrated the need for simple network programming models, at a higher level of abstraction than per-node TinyOS code. This has led to a variety of proposals, including TinyDB's SQL queries [19], diffusion's aggregation [12], regions' MPI-like reductions [28], or market based macroprogramming's pricings [21]. Rather than define a programming model, ASVMs provide a way to implement and build the runtime underlying whichever model a user needs.

6. DISCUSSION AND FUTURE WORK

Section 4 showed that an ASVM is an effective way to efficiently provide a high-level programming abstraction to users. It is by no means the only way, however. There are two other obvious approaches: using a standard virtual machine, such as Java, and sending very lightweight native programs.

As a language, Java may be a suitable way to program a sensor network, although we believe a very efficient implementation might require simplifying or removing some features, such as reflection. Java Card has taken such an approach, essentially designing an ASVM for smart cards that supports a limited subset of Java and different program file formats. Although Java Card supports a single application domain, it does provide guidance on how an ASVM could support a Java-like language.

Native code is another possible solution: instead of being bytecode-based, programs could be native code stringing together a series of library calls. As sensor mote CPUs are usually idle, the benefit native code provides — more efficient CPU utilization — is minimal,

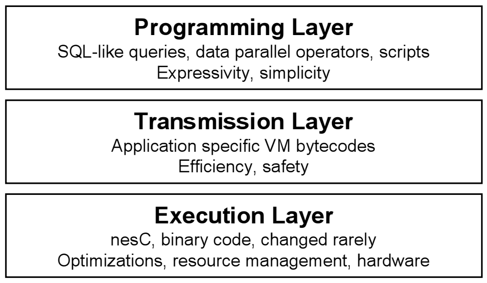
Programming Layer
SQL-like queries, data parallel operators, scripts
Expressivity, simplicity
Transmission Layer
Application specific VM bytecodes
Efficiency, safety
Execution Layer
nesC, binary code, changed rarely
Optimizations, resource management, hardware

Figure 12: A layered decomposition of in-situ reprogramming.

unless a user wants to write complex mathematical codes. In the ASVM model, these codes should be written in nesC, and exposed to scripts as functions. Additionally, native code poses many complexities and difficulties, which greatly outweigh this minimal benefit, including safety, conciseness, and platform dependence. However, the SOS operating system suggests ways in which ASVMs could support dynamic addition of new functions.

ASVMs share the same high-level goal as active networking: dynamic control of in-network processing. The sort of processing proposed by systems such as ANTS and PLANet, however, is very different than that which we see in sensor nets. Although routing nodes in an active Internet can process data, edge systems are still predominantly responsible for generating that data. Correspondingly, much of active networking focused on protocol deployment. In contrast, motes simultaneously play the role of both a router and a data generator. Instead of providing a service to edge applications, active sensor nodes are the application.

Section 4.6 showed how an ASVM — QueryVM — can simultaneously support both SQL-like queries and motlle programs, compiling both to a shared instruction set. In addition to being more energy efficient than a similar TinyDB system, QueryVM is more flexible. Similarly, RegionsVM has several benefits — code size, concurrency, and safety — over the native regions implementation.

We believe these advantages are a direct result of how ASVMs decompose programming into three distinct layers, shown in Figure 12. The highest layer is the code is a user writes (e.g., TinyScript, SQL). The middle layer is the active networks representation the program takes as it propagates (ASVM bytecodes). The final layer is the representation the program takes when it executes on a mote (an ASVM).

TinyDB combines the top two layers: its programs are binary encodings of an SQL query. This forces a mote to parse and interpret the query, and determine what actions to take on all of the different events coming into the system. It trades off flexibility and execution efficiency for propagation efficiency. Separating the programming layer and transmission layer, as QueryVM does, leads to greater program flexibility and more efficient execution.

Regions combines the bottom two layers: its programs are TinyOS images. Using the TinyOS concurrency model, rather than a virtual one, limits the native regions implementation to a single thread. Additionally, even though its programs are only a few lines long — compiling to seventy bytes in RegionsVM — compiling to a TinyOS image makes its programs tens of kilobytes long, trading off propagation efficiency and safety for execution efficiency. Separating the transmission layer from the execution layer, as RegionsVM does, allows high-level abstractions to minimize execution overhead and provides safety.

7. CONCLUSION

The constrained application domains of sensor networks mean that programs can be represented as short, high level scripts. These scripts control — within the protocols and abstractions a domain requires — when motes generate data and what in-network processing they perform. Vision papers and existing proposals for sensor network programming indicate that this approach will not the exception in these systems but the rule. Pushing processing as close to the data sources as possible transforms a sensor network into an active sensor network. But, as sensor networks are so specialized, the exact form active sensor networking takes is an open question, a question that does not have a single answer.

Rather than propose a particular active networking system, useful in some circumstances and not in others, we have proposed using application specific virtual machines to easily make a sensor network active, and described an architecture for building them. Two sample VMs, for very different applications and programming models, show the architecture to be flexible and efficient. This efficiency stems from the flexibility of the virtual/native boundary, which allows programs to be very concise. Conciseness reduces interpretation overhead as well as the cost of installing new programs. “Programming motes is hard” is a common claim in the sensor network community; perhaps we have just been programming to the wrong interface?

Acknowledgements

This work was supported, in part, by the Defense Department Advanced Research Projects Agency (grants F33615-01-C-1895 and N6601-99-2-8913), the National Science Foundation (grants No. 0122599 and NSF IIS-033017), California MICRO program, and Intel Corporation. Research infrastructure was provided by the National Science Foundation (grant EIA-9802069).

8. REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 81–91. ACM Press, 1998.

[2] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedia transcoding. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 178–189. ACM Press, 1998.

[3] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[4] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, 1995.

[5] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the First International Conference on Mobile Systems, Applications, and Services (MobiSys 2003)*, 2003.

[6] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.

[7] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 201–213. ACM Press, 2004.

[8] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSYS '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

[9] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A packet language for active networks. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 1998.

[10] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. Planet: An active internetwork. In *Proceedings of IEEE INFOCOM*, 1999.

[11] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second International Conferences on Embedded Network Sensor Systems (SenSys)*, 2004.

[12] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking*, Aug. 2000.

[13] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, oct 2002.

[14] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[15] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[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] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM Press, 2003.

[19] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *Transactions on Database Systems (TODS)*, 2005.

[20] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[21] G. Mainland, L. Kang, S. Lahaie, D. Parkes, and M. Welsh. Using virtual markets to program global behavior in sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.

[22] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conferences on Embedded Networked Sensor Systems (SenSys)*, 2004.

[23] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic role assignment for wireless sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, 2004.

[24] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets: Applying active networks to network management. *ACM Transations on Computer Systems*, 2000.

[25] C. Sharp, S. Shaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. Proceedings of the Second European Workshop of Wireless Sensor Networks (EWSN 2005), 2005.

[26] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[27] D. Tennenhouse and D. Wetherall. Towards an active network architecture. In *Computer Communication Review, 26(2)*, 1996.

[28] M. Welsh and G. Mainland. Programming sensor networks with abstract regions. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[29] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.