Active Sensor networks

Philip Levis†, David Gay‡, and David Culler
††EECS Department University of California, Berkeley
NSDI ’05
Outline

- INTRODUCTION
- Background
- Design
- Evaluation
INTRODUCTION
Introduction

- **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
Introduction

• User often does not know exactly what the sensor data will look like
  – Reprogram sensor network nodes after deployment
• Propose an architecture for implementing a programming model’s underlying runtime.
• ASVMs
  – Application specific virtual machines.
  – Can support dynamically reprogramming for a wide range of application domains.
• Allowing a user to customize both the instruction set and execution triggering events.
Introduction

• ASVMs provide flexibility to an application developer, who can pick the right level of abstraction based on the particulars of a deployment.

• Two contributions
  – It shows a way to introduce a flexible boundary between dynamic and static sensor network code.
  – Presents solutions to several technical challenges.
BACKGROUND
Background

• Mate fails to meet three requirements.
  – Flexibility
    • Supporting a range of application domains requires two forms of customization
      – The execution primitives of the VM
      – The set of events it executes in response to.
  – Concurrency
    • Mate only have a single shared variable.
  – Propagation
    • In Mate, Every handler could fit in a single packet.
    • Not all programming models can fit their programs in a single packet.
DESIGN
Design

Figure 1: The ASVM architecture.
Design

• Three major abstractions
  – Handlers
    • Run in response to system events.
  – Operations
    • Units of execution functionality.
    • Defined by the Bytecode nesC interface.
    • Primitives
      – Language specific
    • Functions
      – Language independent
  – Capsules
    • Units of code propagation.
Design

• For examples
  – Primitives
    • Conditional jumps
    • Pushing a constant onto the operand stack
  – Function
    • Sending a packet.
Design

- Execute
  - How a thread issues instructions
- byteLength
  - Scheduler correctly control the PC.

```c
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();
}
```
Design

• ASVMs have a threaded execution model and a stack-based architecture.
• The components of an ASVM can be separated into two classes
  – The template
    • Every ASVM includes.
  – Extensions
    • The application-specific components that define a particular ASVM.
Design

- The template includes
  - Scheduler
    - Executes runnable thread in a FIFO RR fashion.
  - Concurrency manager
    - Controls what thread are runnable
  - Capsule store
    - Manages code storage
    - Loading
    - Propagating code capsules
    - Notifying the ASVM when new code arrives.
Design – Data Model

Each thread has an operand stack for passing data between operations.

• The architecture defines a minimal set of standard simple operand types as 16-bits values.

• For elaborate types
  – Languages must provide serialization support for their data types.
Design – Scheduler: Execution

- How scheduler executes a thread
  - Fetching its next bytecode from the capsule store.
  - Dispatching to the corresponding operation component.

- Operations
  - Primitives can have embedded operands.
  - \([\text{width}]<\text{name}>[\text{operand}]\)
    - pushc6
Design – Scheduler: Execution

- **Width** denotes how many bytes wide the operation is
  - Corresponds to the “byteWidth” command of the Bytecode interface
- **Operand** is how many bits of embedded operand the operations has.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Width</th>
<th>Name</th>
<th>Operand Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rand</td>
<td>1</td>
<td>rand</td>
<td>0</td>
<td>Random 16-bit number</td>
</tr>
<tr>
<td>pushc6</td>
<td>1</td>
<td>pushc</td>
<td>6</td>
<td>Push a constant on stack</td>
</tr>
<tr>
<td>2jumps10</td>
<td>2</td>
<td>jumps</td>
<td>10</td>
<td>Conditional jump</td>
</tr>
</tbody>
</table>
Design-Concurrency Manager: Parallelism

- An operation component can register with the concurrency manager.
  - When it need to accesses a shared resource.
- Allow a handler to run when it can exclusively access all of the shared resources it needs
  - Two-phase locking.
- When new code arrives
  - Reboots the asvm.
Design-
Capsule Store: Propagation

• ASVM capsule store maintains three network trickles.
  – Version packets
  – Capsule status packets
  – Capsule fragments
Design-
Capsule Store: Propagation

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

Request → Maintain → Respond → Request (Request timeout)
Maintain → Request → Maintain (Receive complete capsule)
Respond → Maintain → Respond (Hear older version or status for current)
Design – Building an ASVM

• Building an ASVM and scripting environment requires specifying three things
  – A language
  – Functions
  – Handlers
Design – Building an ASVM

<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">

</HANDLER>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
</FUNCTION>
Evaluation - Concurrency

- 50 samples.
- 24 shared resources.
- 128 byte handler.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cycles</th>
<th>Time (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>32</td>
<td>8</td>
</tr>
<tr>
<td>Unlock</td>
<td>39</td>
<td>10</td>
</tr>
<tr>
<td>Run</td>
<td>1077</td>
<td>269</td>
</tr>
<tr>
<td>Analysis</td>
<td>15158</td>
<td>3790</td>
</tr>
</tbody>
</table>
Evaluation - Propagation

71 motes
Experiment fifty times.

Seems to have suffered from bad connectivity or inopportunune suppressions

<table>
<thead>
<tr>
<th></th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mote Retasking</td>
<td>20.8s</td>
<td>7.4s</td>
<td>85.8s</td>
</tr>
<tr>
<td>Network Retasking</td>
<td>39.5s</td>
<td>10.4s</td>
<td>85.8s</td>
</tr>
<tr>
<td>Vector Packets Sent</td>
<td>1.0</td>
<td>1.1</td>
<td>13</td>
</tr>
<tr>
<td>Status Packets Sent</td>
<td>3.2</td>
<td>2.4</td>
<td>19</td>
</tr>
<tr>
<td>Fragment Packets Sent</td>
<td>3.0</td>
<td>2.5</td>
<td>22</td>
</tr>
<tr>
<td>Total Packets Sent</td>
<td>7.3</td>
<td>3.6</td>
<td>30</td>
</tr>
</tbody>
</table>
### Evaluation - Propagation

<table>
<thead>
<tr>
<th></th>
<th>Native</th>
<th>RegionsVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code (Flash)</td>
<td>19kB</td>
<td>39kB</td>
</tr>
<tr>
<td>Data (RAM)</td>
<td>2775B</td>
<td>3017B</td>
</tr>
<tr>
<td>Transmitted Program</td>
<td>19kB</td>
<td>71B</td>
</tr>
</tbody>
</table>
Evaluating Flexibility: Languages

- ASVMs currently support three languages
  - TinyScript
    - RegionsVM
  - Mottle
    - QueryVM
  - TinySQL

```plaintext
buffer packet;
bclear(packet);
packet[0] = light();
send(packet);
```

(a) TinyScript  
(b) ASVM Bytecodes
Evaluation
Efficiency: Microbenchmarks

<table>
<thead>
<tr>
<th></th>
<th>None</th>
<th>Operation</th>
<th>Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration</td>
<td>16.0</td>
<td>16.4</td>
<td>62.2</td>
</tr>
<tr>
<td>Sort Time</td>
<td>-</td>
<td>0.4</td>
<td>46.2</td>
</tr>
</tbody>
</table>
## Evaluation

### Efficiency: Application

<table>
<thead>
<tr>
<th>Name</th>
<th>Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>SELECT id, parent, temp INTERVAL 50s</td>
</tr>
<tr>
<td>Conditional</td>
<td>SELECT id, expdecay(humidity, 3) WHERE parent &gt; 0 INTERVAL 50s</td>
</tr>
<tr>
<td>SpatialAvg</td>
<td>SELECT AVG(temp) INTERVAL 50s</td>
</tr>
</tbody>
</table>
Evaluation

Efficiency: Application
## Evaluation

**Efficiency: Application**

<table>
<thead>
<tr>
<th>Query</th>
<th>Size (bytes)</th>
<th>Energy (mW)</th>
<th>Yield</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TinyDB</td>
<td>VM</td>
<td>TinyDB</td>
</tr>
<tr>
<td>Simple</td>
<td>93</td>
<td>105</td>
<td>5.6</td>
</tr>
<tr>
<td>Conditional</td>
<td>124</td>
<td>167</td>
<td>4.2</td>
</tr>
<tr>
<td>SpatialAvg</td>
<td>62</td>
<td>127</td>
<td>3.3</td>
</tr>
</tbody>
</table>
Evaluation

Efficiency: Application
Evaluation

Efficiency: Interpretation

![Power Draw (mW) Graph](image)

- **Simple**
- **Conditional**
- **SpatialAvg**

- TinyDB
- nesC
- QueryVM
Note

- TinyOS Split-Phase Operation
  - Solutions: Split-Phase
  - A split-phase operation separates the *initiation of a method call* from the *return of the call*. (Similar to Asynchronous methods call in distributed computing)
  - A call to a split-phase operation returns immediately, without actually performing the body of the operation.
  - The true execution of the operation is scheduled later.
  - Execution of the body finishes, the operation will notifies the original caller through a separate method call.