Maté: A Tiny Virtual Machine for Sensor Networks

Philip Levis and David Culler
UC Berkeley
Intel Research: Berkeley
A Sensor Network
Sensor Network Motes
A Sensor Network
A Sensor Network
Bottom Line

• Need in-situ programming
• Has to be:
  – Small
  – Expressive
  – Concise
  – Resilient
  – Efficient
  – Tailorable
  – Simple
Proposal

Maté: A Tiny Virtual Machine for Sensor Networks
Outline

• Sensor networks
• Requirements
• Maté
• Evaluation
• Conclusion
### Technological Constraints

<table>
<thead>
<tr>
<th>Mote Type</th>
<th>Date</th>
<th>WeC</th>
<th>Rene</th>
<th>Rene2</th>
<th>Dot</th>
<th>Mica</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |

|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |
|                |     |     |      |     |     |      |

<table>
<thead>
<tr>
<th>Microcontroller (4MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>Prog. Mem. (KB)</td>
</tr>
<tr>
<td>RAM (KB)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Communication</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Radio</strong></td>
</tr>
<tr>
<td>Rate (Kbps)</td>
</tr>
<tr>
<td>Modulation Type</td>
</tr>
</tbody>
</table>
Example Application Scenario

- Monitor Storm Petrel nesting on Great Duck Island
- Inaccessible: 50 nodes in bird nests
- Simple sense and send loop
- Runs every 8 seconds – low duty cycle
- Frequent reprogramming would be useful
  - Biologists don't know what they need until they see it!
Proposal: Use a Virtual Machine!

- Can express a wide range of applications
- Abstraction of complex operations
- Safe execution environment
- Interpretation overhead small
- Customizable instruction sets
- VM can handle code dissemination
System Requirements

Requirement
• Small
• Expressive
• Concise
• Resilient
• Efficient
• Tailorable
• Simple

Maté provides
• 7286B code, 603B RAM
• Bytecode interpreter
• GDI app is 19 bytes
• Safe execution environment
• Small CPU overhead
• User-definable instructions
• Viral self-programming
Why We Need a New VM

• Communication centric
• Extensibility
• Power a critical consideration
  – JVMs (KVM, PicoJava, etc.)
    – Need over 50 KB of RAM
    – Strings? Are you crazy?
• FORTH
  – How do you install code?
  – Maté draws on FORTH's design decisions
Maté in a Nutshell

- Built on TinyOS, runs on renes and micas
- Three concurrent execution contexts
- Execution triggered by predefined events
- Two stack architecture
- Tiny code capsules self-propagate
  - built-in multihop routing
Maté Architecture

Subroutines

0 1 2 3

gets/sets

Events

Clock Send Receive

Code

PC

Operand Stack

Return Stack

Mate

Context
Maté Instructions

• One byte per instruction

• Three classes: basic, s-type, x-type
  – basic: data, arithmetic, communication, sensing
  – s-type: message headers
  – x-type: embedded operands (e.g. push constant)

• \texttt{usr}0–7 instructions: tailorability

\begin{tabular}{|c|c|}
\hline
\textbf{basic} & \texttt{00iiiiii} & i = instruction \\
\hline
\textbf{s-type} & \texttt{01iiixxx} & x = argument \\
\hline
\textbf{x-type} & \texttt{1ixxxxx} & \\
\hline
\end{tabular}
Maté Sense and Send

pushc 1   # Light is sensor 1
s
sense   # Push light reading on stack
pushm     # Push message buffer on stack
clear     # Clear message buffer
add       # Append reading to buffer
send      # Send message on built-in
halt      # ad-hoc protocol
Maté Capsules

- Hold up to 24 instructions
- Small enough to fit in a single TinyOS packet
  - atomic installation
  - no buffering
- Four types: send, receive, clock, subroutine
  - context-specific: send, receive, clock
  - shared: subroutines 0-3 (call, ret)
But, How Do Capsules Get There?
Viral Code

• Every capsule contains a version number
• Maté installs newer capsules it hears
• Programs can forward capsules
  – local broadcast
  – forw, forwo
pushc 1  # Light is sensor 1
sense   # Push light reading on stack
pushm  # Push message buffer on stack
clear  # Clear message buffer
add    # Append reading to buffer
send   # Send message
forw   # Forward this capsule
halt
Propagation Example
Propagation Example
Propagation Example
Propagation Complete
Node Enters the Network
Node Joins the Network
Evaluation

• What do we care about?
  – CPU cycles
  – bandwidth
  – energy
• Execution rate
• Code propagation behavior
Maté Interpretation Overhead

- ~10,000 instructions per second
- 34:1 to 1.03:1 compared to native code

<table>
<thead>
<tr>
<th>Operation</th>
<th>Maté</th>
<th>Native</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>469 inst</td>
<td>14 inst</td>
<td>34:1</td>
</tr>
<tr>
<td>rand</td>
<td>435</td>
<td>45</td>
<td>9.5:1</td>
</tr>
<tr>
<td>sense</td>
<td>1342</td>
<td>396</td>
<td>3.4:1</td>
</tr>
<tr>
<td>send</td>
<td>685 + ~20,000</td>
<td>~20,000</td>
<td>1.03:1</td>
</tr>
</tbody>
</table>
### Where Do the Cycles Go?

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
<th>Time portion</th>
</tr>
</thead>
<tbody>
<tr>
<td>pushc 1</td>
<td>40 us</td>
<td>0.06 %</td>
</tr>
<tr>
<td>sense</td>
<td>240 us</td>
<td>0.24 %</td>
</tr>
<tr>
<td>pushm</td>
<td>40 us</td>
<td>0.06 %</td>
</tr>
<tr>
<td>clear</td>
<td>40 us</td>
<td>0.06 %</td>
</tr>
<tr>
<td>add</td>
<td>50 us</td>
<td>0.08 %</td>
</tr>
<tr>
<td>send</td>
<td>60,000 us</td>
<td>99.44 %</td>
</tr>
<tr>
<td>halt</td>
<td>40 us</td>
<td>0.06 %</td>
</tr>
</tbody>
</table>

- Dominated by send
- Aggregate overhead: ~1.15:1
Code Propagation Methodology

- 42 node network
- 3 x 14 grid, spaced 20 cm apart
- 3 hop network (radio at very low power)
  - Cells were 15-30 nodes
- TinyOS 0.61 10Kb networking stack
Time to Complete Infection

- Self-forwarding timer capsule runs every 20 seconds
- Measures a quiet network (< 10% bandwidth)

![Network Programming Rate Graph](attachment:image.png)
Propagation Rate Scalability

- Timer capsule ran every second
- Capsule had a forwarding probability:
  - if ((rand & 0x1) == 0x1) forward();
- Network cell bandwidth: 16 packets/second

<table>
<thead>
<tr>
<th>Probability</th>
<th>Expected Interval</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>12.5%</td>
<td>8 s</td>
<td>23 s</td>
</tr>
<tr>
<td>25%</td>
<td>4 s</td>
<td>10 s</td>
</tr>
<tr>
<td>50%</td>
<td>2 s</td>
<td>21 s</td>
</tr>
<tr>
<td>100%</td>
<td>1 s</td>
<td>400 s</td>
</tr>
</tbody>
</table>
Energy Consumption

• Maté imposes a CPU overhead
• Maté provides a reprogramming savings

• Rough energy cost comparison (1 hop)
  – full active: ~15mA x 3V x seconds
  – sense and send overhead/sample (2.5 ms)
  – sleep (~15 uA)
  – reprogramming savings (120 seconds)

⇒ 50,000 samples equals one reprogram budget
⇒ 400,000 seconds, 5 days
Conclusions

- Maté can **conserve** energy
- Spectrum of reprogramming emerges
  - hardware
  - native code
  - bytecode interpreter
Future Work

- VM-land can replace user-land
- Higher-level languages: motlle
- Concurrency control
- Code propagation
- Bombilla: application specific Maté flavors
event result_t Timer.fired() {
    if (state == IDLE && call Photo.sense()) {state = SENSE;}
    return SUCCESS;
}

event result_t Photo.dataReady(uint16_t data) {
    if (state == SENSE) {
        packet->reading = data;
        if (call SendMsg.send(packet, sizeof(DataBuf))) {
            state = SENDING;
        } else {state = IDLE;}
    } else {state = IDLE;}
}
return SUCCESS;

event result_t SendMsg.sendDone(TOS_MsgPtr msg) {
    if (state == SENDING) {state = IDLE;}
    return SUCCESS;
}