TinyOS:
Active Messages and Communications tidbits...

Ajay Gupta
WMU-CS

Parts taken from TinyOS boot camp slides, TinyOS docs and C. Lu's lecture at Washington U, St. Louis

Hardware Constraints

- Severe constraints in power, size, and cost
  translated to:
- Slow CPU
- Short-distance, low-bandwidth radio
- Small memory
- Limited hardware parallelisms
  - CPU hit by many interrupts!
- Support sleep mode in hw components

MICA Mote

- CPU: 4 MHz, 8 bit
  - NO kernel/user protection
- Raw peripherals - a lot of work for CPU:
  - Collect data from sensors
  - Process every bit to/from radio
  - Arbitrate bus
- Radio: 916/433 MHz
  - Rene: 19.2 kbps
  - Mica: 50 kbps (max), 200 feet (power adjustable)
  - NO byte level processing
- Memory
  - Rene: 512 B data; 8K code
  - Mica: 4 KB data; 128 KB code
- Two AA battery
  - 3 days of continuous active operation
- Sleep modes: idle/power-down/power-save
Software Challenges

- Small memory footprint
- Efficient in power and computation
- Lack hardware parallelism → OS provides concurrency-intensive operation
- Real-time
- Robust
- Diversity in applications and design →
  - Efficient modularity
  - Reconfigurable hardware
  - Software & hardware codesign

How about a traditional embedded OS?

- Multi-threaded architecture
  - Large number of threads → large memory
  - Context switch overhead
- I/O model
  - Blocking I/O (stop and go): waste memory on blocked threads
  - Polling (busy-wait): waste CPU cycles and power
- Protection between applications and kernel
  - Overhead for crossing kernel/user boundary & interrupt handling
- Pros
  - Clean & simple programming model
  - Priority-based scheduling support
  - Robust (protect kernel)

Example: Existing embedded OS

1. Thread 1 (high prio) runs
   - read from socket 1
   - block
2. Thread 2 (medium prio) runs
   - read from socket 2
   - block
3. Thread 3 (low prio) runs
4. Thread 2 unblocked, preempt thread 3
5. Thread 1 unblocked, preempt thread 2
6. Threads 1,2,3 complete in order

3 TCB's, 6 context switches, 7 kernel/user switch
TinyOS Solutions

- Support concurrency: event-driven architecture
- Modularity: application = scheduler + graph of components
- Efficiency: Get done quickly and sleep
- Event = function calls
- Less context switch: FIFO/non-preemptable scheduling
- No kernel/application boundary

Component model

- Component has:
  - Frame (memory)
  - Tasks: thread (computation)
  - Interface:
    - Command
    - Event
- Frame: static storage model – compile-time allocation
- Command and events = function calls
- Clean (hw-like) interface
  - No shared memory or global variables
  - Replace hw with sw and vice versa

TOS Component

```c
TOS_MODULE AM //AM.comp/

// AM comp

TOS_SUBMODULE AM

// AM comp

char AM_SUB_TX_PACKET(char* data);
void AM_SUB_POWER(char mode);
char AM_SUB_INIT();

SIGNALS

char AM_MSG_REC(char type,
char* data);
char AM_MSG_SEND_DONE(char success);

HANDLES

char AM_TX_PACKET_DONE(char success);
char AM_RX_PACKET_DONE(char* packet);

USES

char AM_SUB_TX_PACKET(char* data);
void AM_SUB_POWER(char mode);
char AM_SUB_INIT();
```
A Complete Application

TinyOS Two-level Scheduling

How to handle multiple data flows?
Receiving a message

Timing diagram of event propagation

How should network msg be handled?

- **Socket/TCP/IP?**
  - Too much memory for buffering and threads
  - Data are buffered in network stack until application threads read it
  - Application threads blocked until data is available
  - Transmit too many bits (sequence #, ack, re-transmission)
  - Tied with multi-threaded architecture

- **TinyOS solution:** *active messages*

Active Message

- Every message contains the name of an **event handler**
- **Sender**
  - Declaring buffer storage in a frame
  - Naming a handler
  - Requesting Transmission; exit
  - Done completion signal
- **Receiver**
  - The event handler is fired automatically in a target node
  - No blocked or waiting threads on sender or receiver
  - Behaves like any other events
  - Single buffering
Send Message

```c
CHAR TOS_COMMAND(INT_TO_RFM_OUTPUT)(int val) {
    int_to_led_msg* message = (int_to_led_msg*)VAR(msg).data;
    if (!VAR(pending)) {
        message->val = val;
        if (TOS_COMMAND(INT_TO_RFM_SUB_SEND_MSG)(TOS_MSG_BCAST, AM_MSG(INT_READING), &VAR(msg))) {
            VAR(pending) = 1;
            return 1;
        }
    }
    return 0;
}
```

Analysis and Evaluation

Let's take apart Space, Power and Time

Space Breakdown...

![Code size for ad hoc networking application](image)

- Intermittent
- Message Dispatch
- Initialization
- C-Runtime
- Light Sensor
- Clock
- Scheduler
- Led Control
- Messaging Layer
- Packet Layer
- Radio Interface
- Routing Application
- Radio Byte Encoder

- Scheduler: 144 Bytes code
- Totals: 3430 Bytes code
- 226 Bytes data

© Culler et al., TinyOS benchmark presentation, Feb 2001
Power Breakdown...

- Lithium Battery runs for 35 hours at peak load and years at minimum load!
- That's three orders of magnitude difference!
- A one byte transmission uses the same energy as approx 11000 cycles of computation.

<table>
<thead>
<tr>
<th>Components</th>
<th>Active (mA)</th>
<th>Idle (mA)</th>
<th>Sleep (µA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>5</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Radio</td>
<td>7 (TX)</td>
<td>4.5 (RX)</td>
<td>5</td>
</tr>
<tr>
<td>EEPROM</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LED's</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Photo Diode</td>
<td>200</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Temperature</td>
<td>200</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

Time Breakdown...

- 50 cycle thread overhead (6 byte copies)
- 10 cycle event overhead (1.25 byte copies)

- AM
  - 0.05% 0.20% 0.33
  - CPU Utilization 1.12% 0.81% 7.58
  - Radio handler 26.87% 12.16% 182.38
  - Radio decode thread 5.48% 2.48% 37.2
  - Radio Reception 66.48% 36.58% 461.17
  - Idle 54.75%
- Total 100.00% 100.00% 2028.66

---

Applications

- Multi-hop routing
- Active badge
- Vehicle sensing
- Air-to-ground communication
- Habita monitoring @ Great Duck Island
Routing

Each node needs to determine its parent and its depth in the tree:
- Each node broadcasts out <identity, depth, data> when parent is known.
- At start, Base Station knows it is at depth 0.
  - It sends out <Base ID, 0, **>
- Individuals listen for minimum depth parent.

![Diagram of a tree structure with nodes labeled 0, 1, 2, 3.]

Active badge

- 16 motes deployed on 4th floor of Soda Hall.
  - 10 round motes as office landmarks.
  - 2 base stations around corners of the building.
  - 4 Rene motes as active badges for location tracking.
  - AA batteries (3 weeks) for tracking.
  - Tracking precision +/- one office.

[Image of a tracking device and barcode]

Vehicle sensing

- Unmanned airplane dropped motes from an unmanned airplane.
- Motes automatically form a network.
- Motes detect passing vehicles through magnetic sensors.
- Unmanned airplane sends a query to motes to get the passing time of the vehicle.

[Image of an unmanned airplane dropping motes.]

TinyOS: Pros

- Small memory footprint
- Non-preemptable FIFO task scheduling
- Power efficient
  - Put microcontroller and radio to sleep
- Efficient modularity
  - Clean function call (event, command) interface between components
- Concurrency-intensive operations
  - Event/command + tasks
  - Efficient interrupts/events handling (function calls, no user/kernel boundary)

TinyOS: Cons

- Messy/difficult programming model
- Explicit negotiation for data/resource
  - No "long-running" things in command/event handlers
  - No kernel/user protection → NOT robust
    - An infinite loop in application: all dead!
    - Zero compatibility → implement everything from scratch
- No overload protection
  - "Livelock": interruptions/events consume all CPU cycles → NO real-work get done
- No real-time support/analysis
  - Non-preemptable FIFO task scheduling
  - How do we know the performance?
  - Over constraining the platform?