Sensor Networking with SOS

Simon Han – simonhan@cs.ucla.edu
Ram Kumar Rengaswamy – ram@ee.ucla.edu
Roy Shea – roy@cs.ucla.edu
Mani Srivastava – mbs@.ee.ucla.edu
Eddie Kohler – kohler@cs.ucla.edu
SOS Operating System

Problem With Sensor Networks

- Require *uninterrupted* operation indefinitely
- *Post-deployment software updates* are common
  - Customizing the system to the environment
  - Feature upgrades
  - Bug removal
  - Re-tasking of the system
- Re-programming a deployed system is hard
- Remote reprogramming is *essential* for sustainability

The SOS Solution

- Remotely insert binary modules into running kernel
  - Software reconfiguration without interrupting system operation
  - No stop and re-boot unlike differential patching
- Superior performance over virtual machines

CENS '05
Static Kernel

- Hardware abstraction and common services
- Costly to modify after deployment
- Data structures to enable module loading

Dynamic Modules

- Drivers, protocols, and applications
- Inexpensive to modify after deployment
- Position independent
Basic Services

- SOS provides the basic services expected in a sensor operating system
- **Hardware Abstraction Layer** (HAL)
  - Clock, UART, ADC, SPI, etc.
- **Low layer device drivers** interface with HAL
  - Timer, serial framer, communications stack, etc.
- Core kernel **services**
  - Dynamic memory management
  - Scheduling
  - Function control blocks
More Basic Services

Memory Management
- Static allocation is inefficient
- Use fixed-partition dynamic memory allocation
  - Constant allocation time
  - Low overhead
- Memory management features
  - Guard bytes for run-time memory over-flow checks
  - Ownership tracking
  - Garbage collection on completion

Scheduling
- SOS implements non-preemptive priority scheduling using priority queues
- Event is served when there is no higher priority event
  - Low priority queue used for scheduling most events
  - High priority queue used for time critical events including hardware interrupts and sensitive timers
  - Prevents execution in interrupt contexts

```
pkt = (uint8_t*)ker_malloc(hdr_size + sizeof(SurgeMsg), SURGE_MOD_PID);
post_long(TREE_ROUTING_PID, SURGE_MOD_PID, MSG_SEND_PACKET, hdr_size + sizeof(SurgeMsg), (void*)packet, SOS_MSG_DYM_MANAGED);
```
Module Linking

- Independent of *module distribution* protocol
- New module is stored into a free module block located in program memory
- Kernel stores critical information about the module in the module table
- *Initialization routine* for the module is called
  - **Publish** functions for other parts of the system to use
    
    ```
    char tmp_string = {'C', 'v', 'v', 0};
    ker_register_fn(TREE_ROUTING_PID, MOD_GET_HDR_SIZE, tmp_string,
    (fn_ptr_t)tr_get_header_size);
    ```
  - **Subscribe** to functions supplied by other modules
    
    ```
    char tmp_string = {'C', 'v', 'v', 0};
    s->get_hdr_size = (func_u8_t*)ker_get_handle(TREE_ROUTING_PID,
    MOD_GET_HDR_SIZE, tmp_string);
    ```
  - Set initial timers and schedule events
Kernel provides system services and access to hardware

- `ker_timer_start(s->pid, 0, TIMER_REPEAT, 500);`
- `ker_led(LED_YELLOW_OFF);`

Kernel jump table re-directs system calls to handlers

- Upgrade kernel independent of the module
- Interrupts and messages from the kernel are dispatched by a high priority message buffer
- Low latency
- Concurrency safe operation
Inter-Module Message Passing
- Asynchronous communication
- Messages dispatched by a two-level priority scheduler
- Suited for services with long latency
- Type safe binding through publish / subscribe interface

Inter-Module Function Calls
- Synchronous communication
- Kernel stores pointers to functions registered by modules
- Blocking calls with low latency
- Type-safe runtime function binding
· **Problem**: Modules can be remotely added to, removed from, and modified on deployed nodes

· **Accessing a module**
  · If module does not exist, then SOS kernel catches messages for non-existent modules and handles dynamically allocated memory
  · If module exists but can't handle the message, then the module's default handler gets message and kernel handles dynamically allocated memory

· **Subscribing to a module provided function**
  · Publishing a function includes a type description stored in a function control block (FCB) table
  · Subscription attempts include type checks against corresponding FCB
  · Type changes or removal of published functions result in subscribers being redirected to system stub handler function specific to that type
  · Updates to published functions with the same type are assumed to have the same semantics
Many applications are created by combining already written and tested modules.

SOS kernel facilitates loosely coupled modules:
- Passing of memory ownership
- Efficient function and messaging interfaces

Current modules in SOS distribution:
- Aggregation tree, debugging, flooding, mica sensor board drivers, neighbourhood maintenance, surge, tiny diffusion, tree routing, and demonstration modules
Module Design

#include <module.h>

typedef struct {
    uint8_t pid;
    uint8_t led_on;
} app_state;

DECL_MOD_STATE(app_state);
DECL_MOD_ID(BLINK_ID);

int8_t module(void *state, Message *msg){
    app_state *s = (app_state*)state;

    switch (msg->type){
        case MSG_INIT: {
            s->pid = msg->did;
            s->led_on = 0;
            ker_timer_start(s->pid, 0, TIMER_REPEAT, 500);
            break;
        }
        case MSG_FINAL: {
            ker_timer_stop(s->pid, 0);
            break;
        }
        case MSG_TIMER_TIMEOUT: {
            if(s->led_on == 1){
                ker_led(LED_YELLOW_ON);
            } else {
                ker_led(LED_YELLOW_OFF);
            }
            s->led_on++;
            if(s->led_on > 1) s->led_on = 0;
            break;
        }
        default: return -EINVAL;
    }
}

Accessible

- Uses standard C programming language
- General kernel provides common services
- Reusable module library jump starts projects
- Programs created by “wiring” modules together

Clean Modules

- Clean interfaces enhance re-usability
- Simple structure that developers are familiar with
Development Cycle

- **Rapid Development and Deployment**
  - Bugs fixed after deployment
  - Test and develop in real environment
  - Utilize new resources immediately

![Diagram of the development cycle]

SOS: Write → Refine → Update Deployment → Test

Other Solutions: Write → Hope → Deploy → Test
Related Systems

• **TinyOS**
  - Compiles components written in nesC into a static binary image
  - Uses external tools such as MOAP or Deluge to load updated images on remote nodes

• **Mate / Bombilla**
  - Virtual machine generator for use with TinyOS
  - Capsules carry concise byte code for execution on remote nodes

• **Contiki**
  - Very portable system
  - Proof of concept work on dynamic service delivery

• **Impalla**
  - Runs on different classes of nodes
  - Dynamic component support is in development for mote devices

• **Mantis**
  - POSIX for motes
Comparison of application performance in SOS, TinyOS, and MateVM

Surge Tree Formation Latency

<table>
<thead>
<tr>
<th>Platform</th>
<th>ROM</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOS Core</td>
<td>2046</td>
<td>1163</td>
</tr>
<tr>
<td>Dynamic Memory</td>
<td>1536</td>
<td></td>
</tr>
<tr>
<td>TinyOS with</td>
<td>2113</td>
<td>2597</td>
</tr>
<tr>
<td>Mate VM</td>
<td>3974</td>
<td>3196</td>
</tr>
</tbody>
</table>

Memory footprint for base operating system with the ability to distribute and update node programs.

Surge Forwarding Delay

Surge Packet Delivery Ratio

<table>
<thead>
<tr>
<th>System</th>
<th>Time (min)</th>
<th>Time (%)</th>
<th>Relative to TOS (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyOS</td>
<td>3.31</td>
<td># # #</td>
<td>NA</td>
</tr>
<tr>
<td>SOS</td>
<td>3.50</td>
<td># # #</td>
<td>5.70%</td>
</tr>
<tr>
<td>Mate VM</td>
<td>3.68</td>
<td># # #</td>
<td>11.00%</td>
</tr>
</tbody>
</table>

CPU active time for surge application.
Reconfiguration Performance

- Energy trade-offs
  - SOS has slightly higher base operating cost
  - TinyOS has significantly higher update cost
  - SOS is more energy efficient when the system is updated **one or more times a week**
  - Basic optimizations will **continue to improve this break even point**

<table>
<thead>
<tr>
<th>Module Name</th>
<th>Bytes</th>
<th>Code Size (Bytes)</th>
<th>Write Cost (mJ/page)</th>
<th>Write Energy (mJ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sample_send</td>
<td>568</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree_routing</td>
<td>2242</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>photo_sensor</td>
<td>372</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Energy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>46.6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Module size and energy profile for installing surge under SOS.

Energy cost of light sensor driver update.

Energy cost of surge application update.
### Current Feature Comparison

<table>
<thead>
<tr>
<th>Category</th>
<th>SOS</th>
<th>TinyOS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maturity of Code Base</strong></td>
<td>Initial state</td>
<td>Approaching v</td>
</tr>
<tr>
<td><strong>Platform Support</strong></td>
<td>AVR and ARM</td>
<td>AVR and MSP43</td>
</tr>
<tr>
<td><strong>Sensor Driver Support</strong></td>
<td>MTS310 (sensor)</td>
<td>Crossbow series</td>
</tr>
<tr>
<td><strong>Application Support</strong></td>
<td>Developing</td>
<td>Established</td>
</tr>
<tr>
<td><strong>Documentation</strong></td>
<td>Basic tutorial</td>
<td>Established knowledge</td>
</tr>
<tr>
<td><strong>Programming Language</strong></td>
<td>C</td>
<td>nesC</td>
</tr>
<tr>
<td><strong>Optimizations</strong></td>
<td>Per-module</td>
<td>Entire application</td>
</tr>
<tr>
<td><strong>Scheduling</strong></td>
<td>Priority Queue Function Point</td>
<td></td>
</tr>
<tr>
<td><strong>Memory Allocation</strong></td>
<td>Dynamic</td>
<td>Static</td>
</tr>
<tr>
<td><strong>Safety</strong></td>
<td>Run time</td>
<td>Compile time</td>
</tr>
<tr>
<td><strong>Modularity (Development)</strong></td>
<td>Strong</td>
<td>Strong</td>
</tr>
<tr>
<td><strong>Modularity (Deployment)</strong></td>
<td>Strong</td>
<td>Weak</td>
</tr>
<tr>
<td><strong>User Community</strong></td>
<td>Growing</td>
<td>Large</td>
</tr>
<tr>
<td><strong>Classroom Community</strong></td>
<td>Growing</td>
<td>Large</td>
</tr>
</tbody>
</table>

- TinyOS 2.0 is scheduled for release this coming season but does not appear to support dynamic memory, more flexible scheduling, or modular components.
Platform Support

Supported micro controllers

- Atmel Atmega128
  - 4 Kb RAM
  - 128 Kb FLASH
- Oki ARM
  - 32 Kb RAM
  - 256 Kb FLASH

Supported Radio Stacks

- Chipcon CC1000
- BMAC
- Chipcon CC2420
- IEEE 802.15.4 MAC (NDA required)
Simulation Support

- **Source code level** network simulation
  - Pthread simulates hardware concurrency
  - UDP simulates perfect radio channel
  - Supports user defined topology and heterogeneous software configuration
  - Useful for verifying the functional correctness

- **Instruction level simulation** with **Avrora**
  - Instruction cycle accurate simulation
  - Simple perfect radio channel
  - Useful for verifying timing information
  - See [http://compilers.cs.ucla.edu/avrora/](http://compilers.cs.ucla.edu/avrora/)

- **EmStar integration** is under development for use this Spring quarter
Ongoing SOS Development

- RATS *time synchronization* and *low power duty cycling*
- EmSOS and hostmote *integration with EmStar*
- Runtime memory *swapping* to stable storage
- *sQualnet* integration and simulation
- Potential Claremont *deployment Summer '05*
- Continued *classroom use* at UCLA and Yale
Conclusions

- New Architecture for sensor networks
  - Eases application development
  - SOS closes and tightens the development cycle
  - SOS opens new domains of sensor networking
- SOS documentation, tutorials, source, and support
  
  http://nesl.ee.ucla.edu/projects/sos/

Questions?