Surviving Sensor Network Software Faults

Yang Chen (University of Utah)
Omprakash Gnawali (USC, Stanford)
Maria Kazandjieva (Stanford)
Philip Levis (Stanford)
John Regehr (University of Utah)

22nd SOSP
October 13, 2009
In Atypical Places for Networked Systems

- Forest fires
- Volcanoes
- Landmarks
- Really tall trees
Challenges

- Operate unattended for months, years
- Diagnosing failures is hard
  - Input is unknown, no debugger
- Memory bugs are excruciating to find
  - No hardware memory protection
Safe TinyOS
(memory safety)
Safety Violation

• Lab: blink LEDs, spit out error message

• Deployment: reboot entire node (costly!)
  • Lose valuable soft state (e.g., routing tables)
    ‣ takes time and energy to recover
  • Lose application data
    ‣ unrecoverable
Neutron

- Changes response to a safety violation
- Divides a program into recovery units
- Precious state can persist across a reboot
- Reduces the cost of a violation by 95-99%
- Applications unaffected by kernel violations
- Near-zero CPU overhead in execution
- Works on a 16-bit low-power microcontroller
Outline

• Recovery units
• Precious state
• Results
• Conclusion
Outline

- Recovery units
- Precious state
- Results
- Conclusion
A TinyOS Program

- Graph of software components
- Code and state, statically instantiated
- Connections typed by interface
- Minimal state sharing

![Graph of software components]
A TinyOS Program

- Graph of software components
  - Code and state, statically instantiated
  - Connections typed by interface
  - Minimal state sharing

- Preemptive multithreading
  - Kernel is non blocking, single-threaded
  - Kernel API uses message passing
Recovery Units

• Separate program into independent units

• Infer boundaries at compile-time using:
  1. A unit cannot directly call another
  2. A unit instantiates at least one thread
  3. A component is in one unit exactly
  4. A component below syscalls is in the kernel unit
  5. The kernel unit has one thread
Recovery Units

Application Threads

Kernel Thread

syscalls
Recovery Units

Application Threads

Kernel Thread

syscalls
Recovery Units

Application Threads

Kernel Thread
Recovery Units

Application Threads

Kernel Thread

syscalls
Recovery Units

Application Threads

Kernel Thread

syscalls
Rebooting Application Units

- Halt threads, cancel outstanding syscalls
- Reclaim malloc() memory
- Re-initialize RAM
- Restart threads
Canceling System Calls

- Problem: kernel may still be executing prior call
- Next call will return EBUSY
- Pending flag in syscall structure
- Block if flag is set
- On completion, issue new syscall
Memory

• Allocator tags blocks with recovery unit
  • On reboot, walk the heap and free unit’s blocks
  • Must wait for syscalls that pass pointers to complete before rebooting

• On reboot, re-run unit’s C initializers
  • Each unit has its own .data and .bss

• Restart application threads
Kernel Unit Reboot

• Cancel pending system calls with ERETRY
• Reboot kernel
  • Maintain thread memory structures
• Applications continue after kernel reboots
Outline

- Recovery units
- Precious state
- Results
- Conclusion
Coupling

Application Threads

Kernel Thread

syscalls
Coupling

Application Threads

Kernel Thread

syscalls
Precious State

- Components can make variables “precious”
- Precious groups can persist across a reboot
  - Compiler clusters all precious variables in a component into a precious group
  - Restrict what precious pointers can point to

```c
TableItem @precious table[MAX_ENTRIES];
uint8_t @precious tableEntries;
```
Persisting

- Precious variables must be accessed in \texttt{atomic}\{\} blocks
- Only current thread can be cause of violation

- Static analysis determines tainted variables
  - Tainted precious state does not persist on violation
Persisting Variables

• If memory check fails, reboot unit
  • Reset current stack, re-run initializers, zero out .bss, restore variables
  • Need space to store persisting variables

• Simple option: scratch space, wastes RAM

• Neutron approach: place on stack
  • Stack has been reset
  • Often smaller than worst-case stack
Outline

- Recovery units
- Precious state
- Results
- Conclusion
Methodology

• Evaluate cost of a kernel violation in Neutron compared to Safe TinyOS

• Three libraries, 55 node testbed (Tutornet)
  • Collection Tree Protocol (CTP), 5 variables
  • Flooding Time Synch Protocol (FTSP), 7 variables
  • Tenet bytecode interpreter in the paper

• Quantifies benefit of precious state
Kernel Reboot: CTP

Cum. Beacon Count

Time (minutes)
Kernel Reboot: CTP

Cum. Beacon Count

Time (minutes)
Kernel Reboot: CTP

![Graph showing cumulative beacon count over time for different scenarios: No Reboot, Reboot, and Reboot with Neutron.](image)
Kernel Reboot: CTP

99.5% reduction
Kernel Reboot: FTSP
Kernel Reboot: FTSP

CDF (nodes)

Unsynchronized Time

- No Reboot
- Reboot
Kernel Reboot: FTSP

![CDF (nodes) vs. Unsynchronized Time](image)

- No Reboot
- Reboot
- Reboot w/ Neutron
Kernel Reboot: FTSP

94% reduction
Fault Isolation

- CTP/FTSP persist on an application fault
- Application data persists on a kernel fault
## Cost (ROM bytes)

<table>
<thead>
<tr>
<th></th>
<th>Safe TinyOS</th>
<th>Neutron</th>
<th>Increase</th>
<th>Increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blink</td>
<td>6402</td>
<td>8978</td>
<td>2576</td>
<td>40%</td>
</tr>
<tr>
<td>BaseStation</td>
<td>26834</td>
<td>31556</td>
<td>4722</td>
<td>18%</td>
</tr>
<tr>
<td>CTPThreadNonRoot</td>
<td>39636</td>
<td>43040</td>
<td>3404</td>
<td>8%</td>
</tr>
<tr>
<td>TestCollection</td>
<td>44842</td>
<td>48614</td>
<td>3772</td>
<td>8%</td>
</tr>
<tr>
<td>TestFtsp (no threads)</td>
<td>29608</td>
<td>30672</td>
<td>1064</td>
<td>3%</td>
</tr>
</tbody>
</table>

Customized reboot code is small, still fits on these devices.
## Cost (reboot, ms)

<table>
<thead>
<tr>
<th></th>
<th>Node</th>
<th>Kernel</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blink</td>
<td>12.2</td>
<td>11.4</td>
<td>1.16</td>
</tr>
<tr>
<td>BaseStation</td>
<td>22.1</td>
<td>14.1</td>
<td>9.18</td>
</tr>
<tr>
<td>CTPThreadNonRoot</td>
<td>15.6</td>
<td>15.5</td>
<td>1.01</td>
</tr>
<tr>
<td>TestCollection</td>
<td>15.6</td>
<td>15.5</td>
<td>0.984</td>
</tr>
<tr>
<td>TestFtsp (no threads)</td>
<td>14.8</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
## Cost (reboot, ms)

Kernel fault: CPU busy for 10-20 ms

<table>
<thead>
<tr>
<th></th>
<th>Node</th>
<th>Kernel</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blink</td>
<td>12.2</td>
<td>11.4</td>
<td>1.16</td>
</tr>
<tr>
<td>BaseStation</td>
<td>22.1</td>
<td>14.1</td>
<td>9.18</td>
</tr>
<tr>
<td>CTPThreadNonRoot</td>
<td>15.6</td>
<td>15.5</td>
<td>1.01</td>
</tr>
<tr>
<td>TestCollection</td>
<td>15.6</td>
<td>15.5</td>
<td>0.984</td>
</tr>
<tr>
<td>TestFtsp (no threads)</td>
<td>14.8</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Outline

• Recovery units
• Precious state
• Results
• Conclusion
What’s Different Here

• Persistent data in the OS  (RioVista, Lowell 1997)
  • Neutron: no backing store, modify in place

• Microreboots (Candea 2004)
  • Kernel and applications, rather than J2E
  • Doesn’t require a transactional database
What’s Different Here

• Rx (Qin 2007) and recovery domains (Lenharth 2009)
  • Almost no CPU cost in execution, microreboots

• Failure oblivious computing (Rinard 2004)
  • Recover from, rather than mask faults
What’s Different Here

• Changing the TinyOS toolchain is easy

• Changing the TinyOS programming model isn’t (e.g., adding transactions)
  • 90,000 lines of tight embedded code
  • 35,000 downloads/year
Neutron

- Divides a program into recovery units
- Precious state can persist across a reboot
- Near-zero CPU overhead in execution
- Applications survive kernel violations
- Reduces the cost of a violation by 95-99%
- Works on a 16-bit low-power microcontroller
Questions
Diagnosing Faults

At label (2) on August 8, a software command was transmitted to reboot the network, using Deluge [6], in an attempt to correct the time synchronization fault described in Section 7. This caused a software failure affecting all nodes, with only a few reports being received at the base station later on August 8. After repeated attempts to recover the network, we returned to the deployment site on August 11 (label (3)) to manually reprogram each node....

...In this case, the mean node uptime is 69%. However, with the 3-day outage factored out, nodes achieved an average uptime of 96%.


Given the logistics of our deployment we weren’t really able to do much information gathering once Deluge went down in the field, as we simply couldn’t communicate with the testbed until the problem was resolved and it was more important to us, at the time, to get our system back on its feet than to debug Deluge. Note that I believe that the reboots were really more the *symptom*, not the *cause* of the Deluge issue (I think)....

....Anyway, in short this is a long way of saying that we actually have no idea what happened to Deluge.

From: challen@eecs.harvard.edu
Subject: Re: reventador reboots
Date: July 18, 2009 9:15:26 AM PDT