Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors

Adam Dunkels, Björn Grönvall, Thiemo Voigt

Swedish Institute of Computer Science

IEEE EmNetS-I, 16 November 2004
Sensor OS trade-offs:

*static vs dynamic*

*event-driven vs multi-threaded*
What we have done

- **Contiki** – an OS for sensor network nodes
- Ported Contiki to a number of platforms
  - MSP430, AVR, HC12, Z80, 6502, x86, ...
  - Simulation environment for BSD/Linux/Windows
- Built a few applications for experimental network deployments
Contributions

- **Dynamic loading** of programs
  - Selective reprogramming
  - Static vs dynamic linking
- **Concurrency** management mechanisms
  - Events vs threads
  - Trade-offs: preemption, size
Contiki design target

- “Mote”-class device
  - 10-100 kilobytes of code ROM
  - 1-10 kilobytes of RAM
  - Communication (radio)
- ESB from FU Berlin
  - MSP430, 2k RAM, 60k ROM
# Contiki size (bytes)

<table>
<thead>
<tr>
<th>Module</th>
<th>Code MSP430</th>
<th>Code AVR</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>810</td>
<td>1044</td>
<td>10 + e + p</td>
</tr>
<tr>
<td>Program loader</td>
<td>658</td>
<td>-</td>
<td>8</td>
</tr>
<tr>
<td>Multi-threading library</td>
<td>582</td>
<td>678</td>
<td>8 + s</td>
</tr>
<tr>
<td>Timer library</td>
<td>60</td>
<td>90</td>
<td>0</td>
</tr>
<tr>
<td>Memory manager</td>
<td>170</td>
<td>226</td>
<td>0</td>
</tr>
<tr>
<td>Event log replicator</td>
<td>1656</td>
<td>1934</td>
<td>200</td>
</tr>
<tr>
<td>µIP TCP/IP stack</td>
<td>4146</td>
<td>5218</td>
<td>18 + b</td>
</tr>
</tbody>
</table>
Run-time reprogramming and loadable programs
Reprogramming sensor nodes

- **Software development** for sensor nets
  - Need to reprogram many nodes quite often
- Utilize radio for reprogramming
  - Radio inherently **broadcast**
- Reprogram many nodes at once
  - **Much** faster than firmware download via cable or programming adapter
- Reprogram deployed networks
Traditional systems: entire system a monolithic binary

- Most systems statically linked at compile-time
  - Entire system is a monolithic binary
  - Compile-time optimizations, analysis possible
  - Makes code smaller
- But: hard to change
  - Must download entire system
Contiki: loadable programs

- Contiki: one-way dependencies
  - Core resident in memory
    - Language run-time, communication
  - Programs “know” the core
    - Statically linked against core

- Individual programs can be loaded/unloaded
Loadable programs

- Programs can be loaded from anywhere
  - Radio (multi-hop, single-hop), EEPROM, etc
- During software development, usually change only one module
How well does it work?

- Works well
  - Program typically much smaller than entire system image (1-10%)
    - Much quicker to transfer over the radio
  - Reprogramming takes seconds
- Static linking can be a problem
  - Small differences in core means module cannot be run
  - We are implementing a dynamic linker
Concurrency in Contiki
Concurrency is tricky!

- Event-driven vs multi-threaded
- Event-driven (TinyOS)
  - Compact, low context switching overhead, fits well for reactive systems
  - Not suitable for e.g. long running computations
    - Public/private key cryptography
- Multi-threading
  - Suitable for long running computations
  - Requires more resources
Event-driven

- Event-driven (TinyOS)
  - Processes do not run without events
  - Event occurs: kernel invokes event handler
  - Event handler runs to completion (explicit return;)

Kernel

Handler

Handler

Handler
Multi-threaded

- Threads blocked, waiting for events
- Kernel unblocks threads when event occurs
- Thread runs until next blocking statement
- Each thread requires its own stack
  - Larger memory usage
# Event-driven vs multi-threaded

<table>
<thead>
<tr>
<th>Event-driven</th>
<th>Multi-threaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>- No <code>wait()</code> statements</td>
<td>+ <code>wait()</code> statements</td>
</tr>
<tr>
<td>- No preemption</td>
<td>+ Preemption possible</td>
</tr>
<tr>
<td>- State machines</td>
<td>+ Sequential code flow</td>
</tr>
<tr>
<td>+ Compact code</td>
<td>- Larger code overhead</td>
</tr>
<tr>
<td>+ Locking less of a problem</td>
<td>- Locking problematic</td>
</tr>
<tr>
<td>+ Memory efficient</td>
<td>- Larger memory requirements</td>
</tr>
</tbody>
</table>

Why don't we try to **combine** them?
Contiki: event-based kernel with threads

- Contiki: **kernel** is **event-based**
  - Most programs run directly on top of the kernel
- **Multi-threading** implemented as a library
- Threads only used if **explicitly** needed
  - Long running computations, ...
- **Preemption** possible
  - Responsive system with running computations
Contiki: implementing threads on top of an event-based kernel
Conclusions
Conclusions

- Contiki – OS for “mote”-class sensor nodes
- Contiki explores trade-offs in
  - static vs dynamic
  - event-driven vs multi-threaded
- Loadable programs, works well
  - Static linking can be a problem
- Threads on an event-driven kernel
  - Multi-threading suitable for certain applications
Thank you!

Adam Dunkels <adam@sics.se>

http://www.sics.se/~adam/contiki/
Backup slides
Memory management

- Memory allocated when module is loaded
  - Both ROM and RAM
  - Fixed block memory allocator
- Code relocation made by module loader
  - Exercises flash ROM evenly
Protothreads: light-weight stackless threads

- Protothreads: mixture between event-driven and threaded
  - A third concurrency mechanism
- Allows blocked waiting
- Requires per-thread no stack
- Each protothread runs inside a single C function
- 2 bytes of per-protothread state
**Embedded operating systems**

- **Linux**: Heavy-weight processes with memory protection
- **eCos, OSE, Mantis**: Lightweight multi-threading, preemption
- **Contiki**: Event-driven system
- **TinyOS**: Simple control loop