Hands on Contiki OS and Cooja Simulator (Part I)

Ing. Pietro Gonizzi
Wireless Ad-hoc Sensor Network Laboratory (WASNLab), University of Parma
pietro.gonizzi@studenti.unipr.it

Dr. Simon Duquennoy
Swedish Institute of Computer Science (SICS)
simonduq@sics.se
Outline

- Contiki Overview
- Basics
- Programming your first application
- The Cooja simulator
- IPv6 Networking
Goal of this Course

- Introduction to Contiki and the Cooja network simulator
  ‣ Help you to start writing Contiki applications
  ‣ Basis for further exploration
  ‣ No low level details
  ‣ Will not be able to cover everything on the slides
  - Together with the notes, you should be able to continue
Wireless Sensor Networks

- Consist of many embedded units called sensor nodes, motes etc.
  - Sensors (and actuators)
  - Small microcontroller
  - Limited memory
  - Radio for wireless communication
  - Power source (often battery)
- Motes form networks and in a one hop or multi-hop fashion transport sensor data to base station
Applications

- Classic WSN applications
  - volcano monitoring
  - wildlife monitoring
  - tunnel monitoring and rescue
- …and many IoT-based applications
  - Smart Parking
  - Smart Lighting
  - Smart Plants
  - Smart Toys
  - Building/Home Automation
WSN Operating Systems

- OS is interface between hardware and programmer
  - Hides many details
- Contains drivers to radio and sensors, scheduling, network stacks, process & power management
- Due to memory constraints and target (embedded) not as convenient as OS for PCs
  - Limited user interaction
- TinyOS, Contiki, FreeRTOS, Mantis OS
Contiki Overview

- Contiki – a dynamic operating system for networked embedded systems
  - Main author and project leader: Adam Dunkels (Thingsquare, earlier SICS)
- Small memory footprint
  - Event-driven kernel, multiple threading models on top
- Designed for portability
  - Many platforms (Tmote Sky, Zolertia, RedBee etc.), several CPUs
  - Code hosted on github
- Used in both academia and industry
  - Contributors from Atmel, Cisco, Redwire LLC, SAP, SICS, Thingsquare, and others
Contiki Overview

Basically, Contiki is:

- A scheduler (event handler)
  - Loop that just takes the next event and processes it
  - Nothing to do->goes to sleep (MCU low power mode)
- Set of services
  - Networking, storage, timers, and others
Contiki Programming Model: Protothreads

- The Contiki kernel is event-based
  - invokes processes whenever something happens:
    - sensor events, processes starting, exiting

- **Protothreads** provide sequential flow of control on top of an event-based kernel
  - Easy to program
  - Also comes with some limitations, discussed later
Protothreads: Example

```c
int a_protothread(struct pt *pt) {
    PT_BEGIN(pt);

    PT_WAIT_UNTIL(pt, condition1);

    if(something) {
        PT_WAIT_UNTIL(pt, condition2);
    }
    PT_END(pt);
}
```
Protothreads

- **Single** stack
  - Low memory usage, like events
- Sequential flow of control
  - No explicit state machines, just like threads
- Implemented using **local continuations** (a continuation is an abstract representation of the control state of a program)
  - When **Set**, capture the state of a function
  - When **resumed**, resume the state and perform a jump
  - Stack information across blocking calls must be manually stored and retrieved (e.g. static variable). See issue with protothreads next
Protothreads – Symplifying Event-driven Programming

1. Turn radio on.
2. Wait until $t = t_0 + t_{awake}$.
3. If communication has not completed, wait until it has completed or $t = t_0 + t_{awake} + t_{wait\_max}$.
4. Turn the radio off. Wait until $t = t_0 + t_{awake} + t_{sleep}$.
5. Repeat from step 1.

No blocking wait!

Problem: with events, we cannot implement this as a five-step program!
Protothreads-based Implementation

```c
int protothread(struct pt *pt) {
    PT_BEGIN(pt);
    while(1) {
        radio_on();
        timer = t_awake;
        PT_WAIT_UNTIL(pt, expired(timer));
        timer = t_sleep;
        if(!comm_complete()) {
            wait_timer = t_wait_max;
            PT_WAIT_UNTIL(pt, comm_complete() || expired(wait_timer));
        }
        radio_off();
        PT_WAIT_UNTIL(pt, expired(timer));
    }
    PT_END(pt);
}
```

- Code uses structured programming (if and while), mechanisms evident from code

  → Protothreads make Contiki code nice
Contiki Processes

Contiki processes are protothreads:

- PROCESS_THREAD defines a new process
- PROCESS_BEGIN() and PROCESS_END()
- PROCESS_WAIT_EVENT() or PROCESS_YIELD() wait for new event to be posted to process
- PROCESS_WAIT_EVENT_UNTIL(condition c) waits for an event to be posted with extra condition, e.g.
  - Button has been pressed
  - Timer has expired
Protocol stacks

Protocol stacks in Contiki:

› uIP: world's smallest, fully compliant TCP/IP stack

- Both IPv4 and IPv6, 6LoWPAN, routing RPL, TCP/UDP support
- Also higher layer protocols: HTTP, CoAP and many others

› Rime stack: protocol stack consisting of simple primitives

› MAC layers in Contiki:

- Carrier Sense Multiple Access (CSMA)
- NullMAC

› Radio Duty-Cycling (RDC) layers

- ContikiMAC (default on Tmote Sky)
- NullRDC (duty cycle off)
- NullRDC (duty cycle off)
- And others (less tested): LPP, X-MAC
Cooja simulator

- COOJA: extensible Java-based network simulator for Contiki-based applications
  - Cross-level: Java nodes, Contiki nodes (deployable code), emulated nodes (deployable firmware, not necessarily Contiki)
- MSPSim: sensor node emulator for MSP430-based nodes:
  - Tmote Sky, Zolertia Z1, Wismote, etc.
  - Enables cycle counting, debugging, power profiling etc.
  - Integrated into COOJA or standalone
- COOJA +MSPSim
  - Simulate the network, emulate every nodes’ firmware
  - Also enables interoperability testing for MSP-based platforms (e.g. IPv6 interop testing)
Cooja features

- Network Visualizer
  - mote type, grid, radio environment, radio traffic, etc.
  - Enables changes to the TX/INT range

- Mote output
  - serial output of the nodes (e.g. `printf()`)  

- Timeline
  - radio activity of the nodes in real-time
  - E.g., radio status, ongoing packets

- Radio messages
  - capturing radio packets
  - Useful for Wireshark analysis
/** Declare the process */
PROCESS(hello_world_process, "Hello world");
/** Make the process start when the module is loaded */
AUTOSTART_PROCESSES(&hello_world_process);

/** Define the process code */
PROCESS_THREAD(hello_world_process, ev, data) {
    PROCESS_BEGIN();  /* Must always come first */
    printf("Hello, world!\n"); /* code goes here */
    PROCESS_END();  /* Must always come last */
}
CONTIKI_PROJECT = hello-world

all: $(CONTIKI_PROJECT)

UIP_CONF_IPV6=1

CONTIKI = /home/user/contiki

include $(CONTIKI)/Makefile.include
Running Hello World

- native platform (your VM)
  
  \[\text{cd contiki/examples/hello-world} \]
  
  \[\text{make hello-world.native} \]

  ▶ After the compilation, start the program with
  
  \[./hello-world.native \]

  ▶ The program prints “Hello, World” and finishes (appears to hang). Interrupt it by pressing Ctrl-C

- Tmote sky platform
  
  ▶ place Tmote in a USB and it will appear in the top of instant Contiki as “Future Technologies Device”. Click on name to connect it to Instant Contiki.
  
  \[\text{cd contiki/examples/hello-world} \]
  
  \[\text{make TARGET=sky hello-world.upload} \]

  ▶ When the compilation is finished, the uploading procedure starts (LEDS blink like crazy).

  ▶ You can see the output of the program by logging into the node
  
  \[\text{make login TARGET=sky} \]

  ▶ Press the reboot button to see some output
Contiki directories

- **contiki/core**
  - System source code; includes (among others)
  - net: rime, MACs, IP etc;
  - sys: processes

- **contiki/examples**
  - Lots of nice examples, see /ipv6 for examples with uIP stack

- **contiki/apps**
  - System apps (telnet, shell, deluge), *not your application code*

- **contiki/platform**
  - Platform-specific code:
    - platform/sky/contiki-sky-main.c
    - platform/sky/contiki-conf.h

- **contiki/cpu**
  - CPU-specific code: one subdirectory per CPU

- **contiki/tools**
  - e.g. cooj, start with “ant run”
  - tools/sky contains serialdump (start with “./serialdump-linux -b115200 /dev/ttyUSB0”) and other useful stuff
Timers in Contiki

- **struct timer**
  - Passive timer, only keeps track of its expiration time

- **struct etimer**
  - Active timer, sends an event when it expires

- **struct ctimer**
  - Active timer, calls a function when it expires

- **struct rtimer**
  - Real-time timer, calls a function at an exact time. *Reserved for OS internals*
Events and Processes

PROCESS_WAIT_EVENT();
Waits for an event to be posted to the process

PROCESS_WAIT_EVENT_UNTIL(condition c);
Waits for an event to be posted to the process, with an extra condition. Often used: wait until timer has expired

PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));

PROCESS_POST(...) and PROCESS_POST_SYNCH(..)
Post (a)synchronous event to a process.
The other process usually waits with PROCESS_WAIT_EVENT_UNTIL(ev == EVENTNAME);
- By default, Contiki on Tmote sky uses ContikiMAC

<table>
<thead>
<tr>
<th>Networking</th>
<th>Rime, SICSLoWPAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAC</td>
<td>CSMA, NULLMAC</td>
</tr>
<tr>
<td>RDC</td>
<td>Framer</td>
</tr>
<tr>
<td>Radio</td>
<td>CC2420</td>
</tr>
</tbody>
</table>

Framer: 802.15.4, NULL
2 functions: create, parse
Measure Power Consumption with Energest

```c
PROCESS_BEGIN();
    static struct etimer et;
    static unsigned long rx_start_time;
...
...
    rx_start_time = energest_type_time(ENERGEST_TYPE_LISTEN);
    lpm_start_time = energest_type_time(ENERGEST_TYPE_LPM);
    cpu_start_time = energest_type_time(ENERGEST_TYPE_CPU);
    tx_start_time = energest_type_time(ENERGEST_TYPE_TRANSMIT);
...
    printf("energy listen %lu tx %lu cpu %lu lpm %lu\n",
            energest_type_time(ENERGEST_TYPE_LISTEN) - rx_start_time,  // in while loop
            energest_type_time(ENERGEST_TYPE_TRANSMIT) - tx_start_time,
            energest_type_time(ENERGEST_TYPE_CPU) - cpu_start_time,
            energest_type_time(ENERGEST_TYPE_LPM) - lpm_start_time);

PROCESS_END();
```
Measure Power Consumption with Energest

- Now we have the times a component was on, eg
  - CPU on ("cpu"), CPU idle ("lpm"), Radio tx, Radio rx, Radio idle, Flash operations, etc
- Note: the cpu is always either on or idle, total runtime = "cpu" + "lpm"
- Can be used to estimate energy consumption
  - Based on power draw (from datasheet or measured)
  - Using other metrics, such as "duty cycle", the portion of time with radio on
    - Duty cycle = (tx+rx) / (cpu+idle)
Measure Power Consumption with Energest

Figure 4: Measuring communication energy expenditure with Powertrace: the radio duty cycling layer maintains energy capsules for wake-ups, transmissions, and receptions. In the figure, capsules for wake-up and transmissions are shown. The transmission capsule is split across two activities: the first transmission attempt at 40 ms, which sensed another transmission in the ether and backed off, and the retransmission at 100 ms.
Thank you