Kiel University
Christian-Albrechts-Universität zu Kiel

Research Group
Distributed Systems

Internet of Things & Wireless Networks

Programming IoT with Contiki-NG

Olaf Landsiedel

Summer Term 2020

Last Time?

- Motivation and Introductions
- Platforms
- Course Organization

Today

1. **Overview of Contiki**
2. Programming basics

Contiki

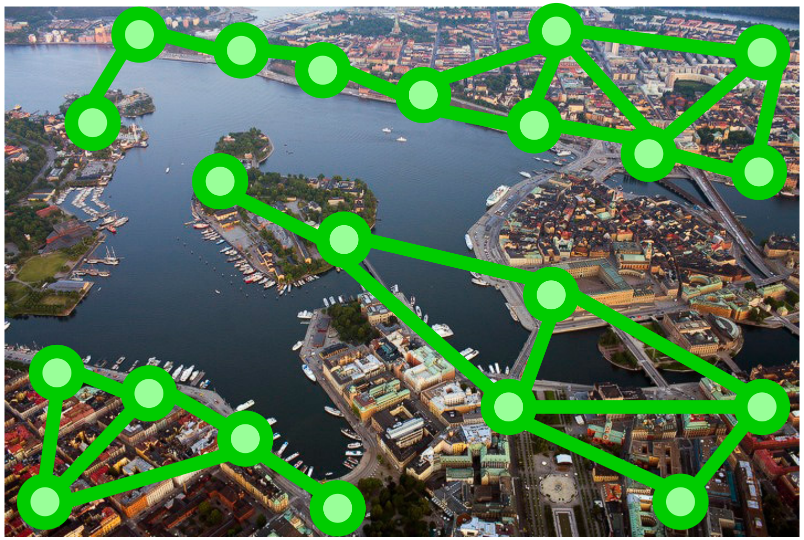
- The Open Source OS for the Internet of Things
- Open source: BSD
- C-based (+ *protothreads*)
- Supports many embedded platforms
- Supports standard low-power IPv6
- Includes Cooja simulator

Contiki
The Open Source OS for the Internet of Things

- Created in 2003
- Used both by Academia and Industry
  - Lots of papers referring to it (1850+ citations)
  - More and more commercial products
- Brings standards to the most constrained devices

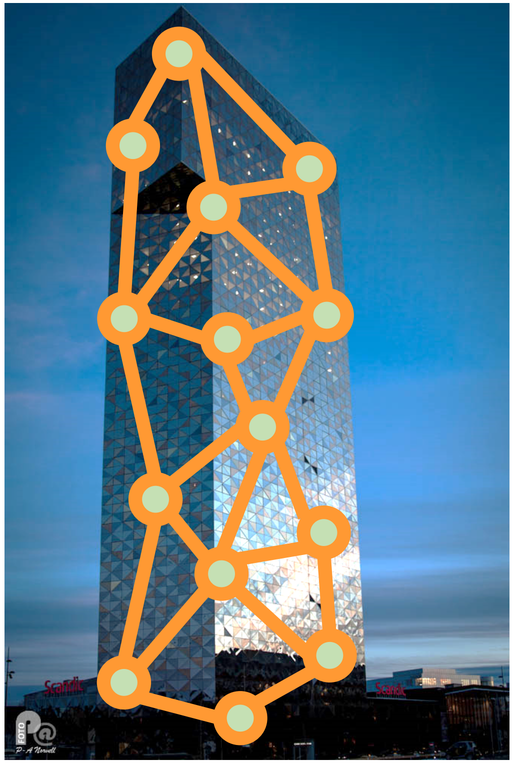
Typical Applications

- IoT scenarios: smart cities, building automation, …
- Multiple hops to cover large areas
- Low-power for battery-powered scenarios
- Nodes are interoperable and addressable (IP)



Traffic lights
Parking spots
Public transport
Street lights
Smart metering
…

Light bulbs
Thermostat
Power sockets
CO2 sensors
Door locks
Smoke detectors
…



Contiki Community

- Open source: BSD (business-friendly)
- Led by Adam Dunkels (Thingsquare, formerly SICS)
- 11 maintainers (Thingsquare, SICS, Bristol University, Inria, Zolertia, CETIC, …)
- 125 contributors
- 1755 followers
- Many more users!

Contiki Contenders

- Contiki is not the only IoT OS around ☺
  - But a very widespread one, in both research and industry
- TinyOS
  - Historical sensor OS, from US
  - Hugely used in research, now declining
- RIOT OS
  - Recent, similar focus to Contiki, LGPL
- ARM mbed, Apache MyNewt, Zephyr, …

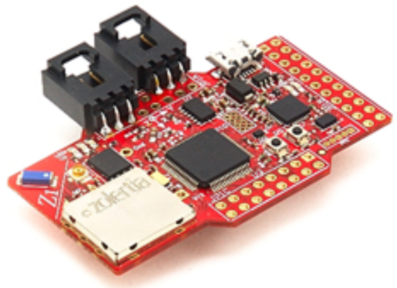
Contiki Programming model

- Standard C
- Uses protothreads (eases event-driven programming)

1. Create application
2. Configure Network stack
3. Compile application with core (OS)
4. Flash device with resulting firmware

Contiki Platforms

- Many different platforms (currently 33)
- 8, 16, 32 bits MCUs
- Typically an IEEE 802.15.4 radio
- Battery-based
- RAM (4-10s of kB)
- ROM (10-100s of kB)
- Sensors / actuators



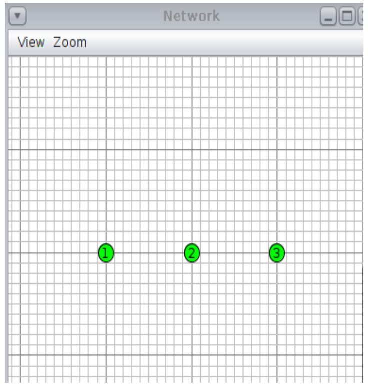
Contiki’s low-power IP stack

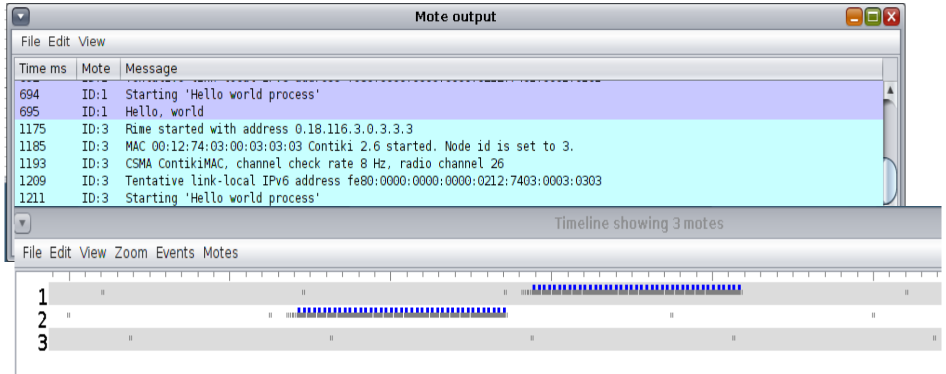
- Early implementation of standards
- IPv6
  - Certified IPv6 stack from CISCO in 2008
- 6LoWPAN
  - IPv6 for IEEE 802.15.4
- RPL
  - Routing in “low-power and lossy networks”
- CoAP
  - Application layer (HTTP-lite)
- TSCH and 6TiSCH
  - Time Slotted Channel Hopping MAC of IEEE 802.15.4e

Contiki’s Network Simulator

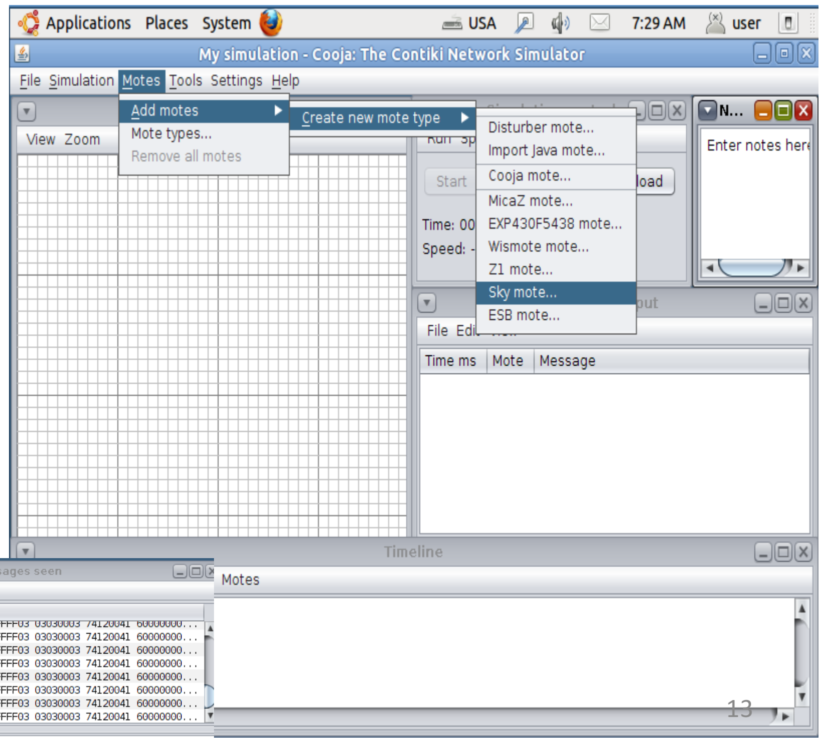
- COOJA: extensible Java-based network simulator
  - Java nodes
  - Contiki nodes (deployable code)
  - Emulated nodes (deployable firmware, not necessarily Contiki)
- MSPSim: sensor node emulator for MSP430+cc2420 nodes:
  - Tmote Sky, Zolertia Z1, Wismote, etc.
  - Enables cycle counting, debugging, power profiling etc.
- COOJA + MSPSim
  - Simulate the network + emulate the nodes’ firmware

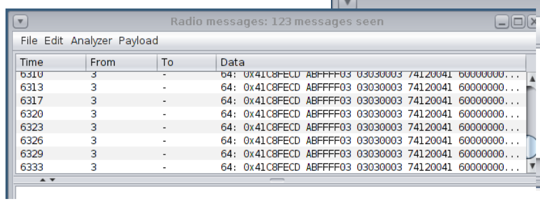
Cooja features





- 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



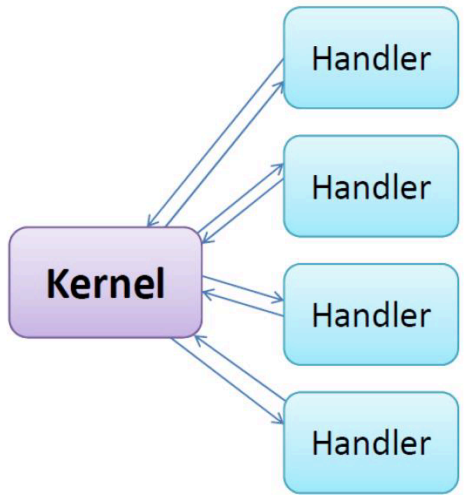


Outline

1. Overview of Contiki
2. **Programming basics**

Contiki kernel and Protothreads

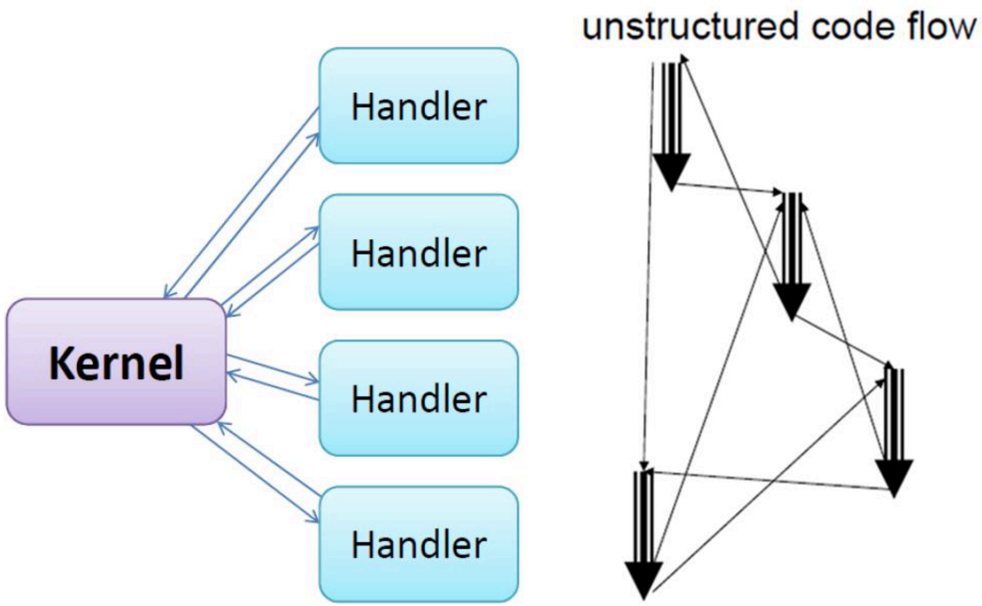
- The Contiki kernel is event-based
  1. Wait for interrupt
  2. Run interrupt handler
  3. Call kernel and applications
  4. Sleep
- Three types of events
  1. Timer events
  2. External events (e.g., hardware interrupts)
  3. Internal events (e.g., inter-process communication)



Contiki kernel and Protothreads

- Event-driven programming is not straightforward!
- Using threads would be much easier!

Event-driven approach



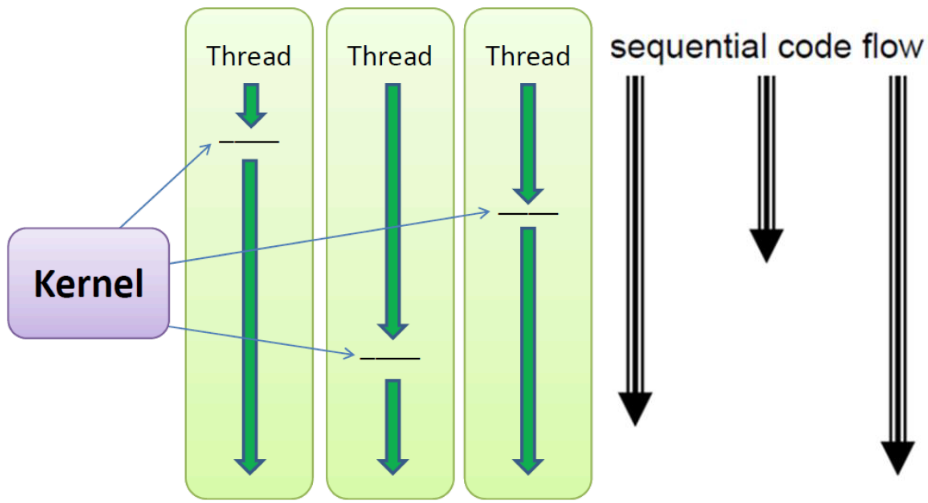
- Not all programs easily expressible as state machines
- Many pitfalls (example: recursive callbacks!)
- Memory efficient (same stack for all processes)
- Low context switching overhead
- No preemption possible
- No need for locking mechanisms

Contiki kernel and Protothreads

- Event-driven programming is not straightforward!
- Using threads would be much easier!

Multithreaded approach

- Suitable for long computations
- Preemption is possible
- Locking mechanisms required (reentrant code needed)
- High memory consumption (each thread has its own stack, typically over-provisioned)
- Large code overhead



Prothothreads

Example: conditional blocking wait

- A sequential control flow on top of an event-based kernel
  - Easy to program
  - Comes with some limitations

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

  PT_WAIT_UNTIL(pt, condition1);

  if(something) {

    PT_WAIT_UNTIL(pt, condition2);

  }
  PT_END(pt);
}
```

Block until condition is true

Protothreads

- Single stack
  - Cost: Two bytes of RAM, ~10 CPU cycles per protothread, instead of expensive context switching and one stack per thread.
- Implemented using **local continuations** – ANSI C
  - When set, capture the state of a function
  - When resumed, perform a jump

Stack information lost across blocking calls

If you want to preserve variable state across YIELD & WAIT:

→ Use **`static`** rather than automatic variables!

+ **Constraints on the use of** `switch()`

Six-line implementation

Protothreads implemented using the C switch statement

```
struct pt { unsigned short lc; };

#define PT_INIT(pt)              pt->lc = 0
#define PT_BEGIN(pt)             switch(pt->lc) { case 0:
#define PT_EXIT(pt)              pt->lc = 0; return 2
#define PT_WAIT_UNTIL(pt, c)     pt->lc = __LINE__; case __LINE__: \
                                 if(!(c)) return 0
#define PT_END(pt)               } pt->lc = 0; return 1
```

C-switch expansion

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


  PT_WAIT_UNTIL(pt, condition1);


  if(something) {


    PT_WAIT_UNTIL(pt, condition2);


  }

  PT_END(pt);
}
```

```
int a_protothread(struct pt *pt) {
  switch(pt->lc) { case 0:


  pt->lc = 5; case 5:
  if(!condition1) return 0;

  if(something) {


    pt->lc = 10; case 10:
    if(!condition2) return 0;

  }

  } return 1;
}
```

Line numbers

Protothreads

- Cooperative multithreading: Preempt at defined points **only**
  - Unlike classic threads which preempt at the OS scheduler will.
- Conditional blocking waits
  - `PT_WAIT_UNTIL (pt, condition*)`
    The protothread is blocked until condition becomes true
  - `PT_WAIT_WHILE (pt, condition*)`
    The protothread is blocked while condition is true
- Yielding a protothread
  - `PT_YIELD (pt)`
    Yield the protothread, thereby allowing other processing to happen.

Contiki Processes

Contiki processes are based on protothreads:

- `PROCESS_THREAD(name,events,data)`: defines a new process
- `PROCESS_BEGIN()` and `PROCESS_END()` : Enclose the process
- `PROCESS_WAIT_EVENT()` or `PROCESS_YIELD()`
  Preempt the process and wait for new event to be posted to process
- `PROCESS_WAIT_EVENT_UNTIL(condition)`
  waits for an event to be posted with extra condition, e.g.,
  - Button:
    `PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event&&data == &button_sensor);`
  - Timer expiration:
    `PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));`

Two ways to make a process run

- Post an event
  - `process_post(process_ptr, eventno, ptr);`
    - Process will be invoked later
  - `process_post_synch(process_ptr, eventno, ptr);`
    - Process will be invoked now
  - Must **not** be called from an **interrupt** (device driver)
- Poll the process
  - `process_poll(process_ptr);`
    - Sends a PROCESS_EVENT_POLL event to the process
  - Can be called from an interrupt

Starvation in Contiki

- Thread scheduling is cooperative
  - play nice
- The watchdog is on
  - on some platforms unable to turn off
- Don't hog the CPU
- Long-running, do
  - watchdog_periodic();
  - PROCESS_WAIT();

Example: Hello World!

- Simple process that prints "Hello World!" tostdout

Define a new process

PROCESS_THREAD (name, events, data)

Enclose the process

PROCESS_BEGIN()

PROCESS_END()

Wait for an event to be posted to the process PROCESS_WAIT_EVENT()

```
PROCESS_THREAD(hello_world_p, ev, data) {
    PROCESS_BEGIN();
    printf("Hello world!\n");
    while(1) {
        PROCESS_WAIT_EVENT();
    }
    PROCESS_END();
}
```

Waits forever!

Example Process: Hello World

– See examples/hello-world: **hello-world.c,** Makefile

```
#include "contiki.h" // Contiki core
#include <stdio.h>  // Necessary for the printf
/* Declare the process */
PROCESS(hello_world_process, "Hello world process");
/* 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 */

   while(1) {
      PROCESS_WAIT_EVENT();         /* Wait for events */
   }

  PROCESS_END();                /* Must always come last */
}
```

Example Process: Hello World

– See examples/hello-world: hello-world.c, **Makefile**

```
# Target platform
ifndef TARGET
TARGET=...
endif
```

```
TARGET=sky
```

```
TARGET=z1
```

```
TARGET=srf06-cc26xx
BOARD=sensortag
```

```
# name of the .c file containing the main application
CONTIKI_PROJECT = hello-world
# what to compile
all: $(CONTIKI_PROJECT)
# optional: path to custom configuration file
# CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
# Additional source files to be compiled (if any)
#CONTIKI_TARGET_SOURCEFILES += library.c
# path to Contiki
CONTIKI = ../..
# include Contiki makefile
include $(CONTIKI)/Makefile.include
```

Example Process: Hello World

- See examples/hello-world: hello-world.c**, Makefile**
- Compile for target sky
  - **`$ make TARGET=sky hello-world`**
- Delete all compiled code
  - **`$ make TARGET=sky clean`**

Contiki Timers

- 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
  - **Only one in the system, reserved for OS internals**

Timers in Contiki

contiki\core\sys\etimer.h

- Etimer
  - Active timer: sends an event when it expires
  - Declaration of timer
    - `static struct etimer et;`
  - Activate and deactivate the timer
    - `etimer_set(&et, AMOUNT_OF_TICKS);`
    - `etimer_stop(&et);`
  - Set `AMOUNT_OF_TICKS` as a function of `CLOCK_SECOND`
  - Keep track of expirations
    - `etimer_pending(); // Is there a non-expired event?`
    - `clock_time_t next_expiration_time();`

Example: Periodic Hello World!

- Printing "Hello World!" every two seconds. Using an etimer.

```
#include "contiki.h"
#include <stdio.h>
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_EXITHANDLER()
    PROCESS_BEGIN();
    static struct etimer et;
    while(1) {
        etimer_set(&et, (CLOCK_SECOND*2));
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        printf("Hello world!\n");
    }
    PROCESS_END();
}
```

We add an etimer that fires every second

We wait for the timer to expire!

Example: Periodic Hello World!

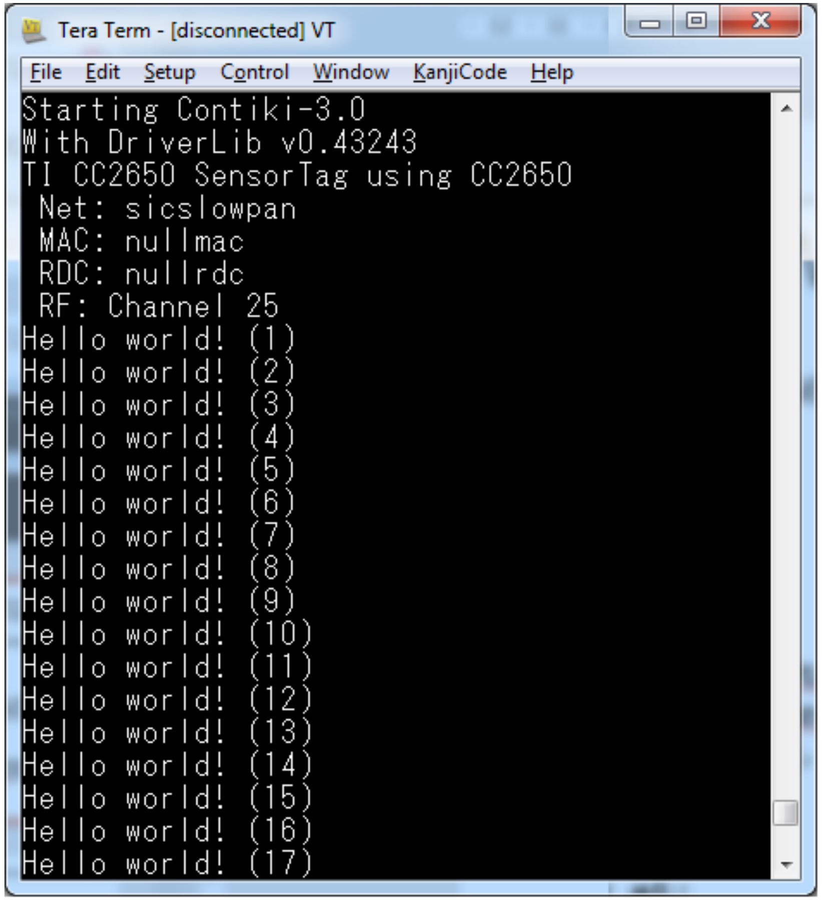
- Printing "Hello World!" every two seconds. Using an etimer.


```
#include "contiki.h"
#include <stdio.h>
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_EXITHANDLER()
    PROCESS_BEGIN();
    static int counter = 0;
    static struct etimer et;
    while(1) {
        etimer_set(&et, (CLOCK_SECOND*2));
        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
        printf("Hello world! (%d)\n", ++counter);
    }
    PROCESS_END();
}
```

We add an etimer that fires every second

We wait for the timer to expire!

Example: Periodic Hello World!



Timers in Contiki

- Ctimer
  - Active timer, calls a function when it expires
  - Declaration of timer and callback
    - `static struct ctimer timer1;`
    - ```
      static void ctimer1_callback(void *ptr) {
          …
      }
      ```
  - Activate and deactivate the timer
    - `ctimer_set(&timer1, AMOUNT_OF_TICKS, ctimer1_callback, NULL);`
    - `ctimer_stop(&timer1);`
  - Set `AMOUNT_OF_TICKS` as a function of `CLOCK_SECOND`

contiki\core\sys\ctimer.h
      ```

Example: Periodic Hello World!

- Printing "Hello World!" every two seconds Using a ctimer

```
#include "contiki.h"
#include <stdio.h>

static struct ctimer timer;

static void tout_cback(void *ptr){
    printf("%s", (char *) ptr);
    ctimer_set(&timer, EXPIRATION, tout_cback, ptr);
}

PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
PROCESS_THREAD(hello_world_process, ev, data)  {
    PROCESS_EXITHANDLER()
    PROCESS_BEGIN();
    ctimer_set(&timer, CLOCK_SECOND*2, tout_cback, "Hello world!\n");
    while(1) {
        PROCESS_WAIT_EVENT();
    }  PROCESS_END();
}
```

ctimer declaration

ctimer callback

ctimer activation

Contiki's SW-based Energy Estimation

- Energest helps you measuring the energy consumption of your application
  - Uses a timer to count the amount of time in which a module is active in a certain mode
  - Surround the piece of code between two instructions `energest_type_time(ENERGEST_TYPE);`
  - Energest types
    - Microprocessor `ENERGEST_TYPE_CPU, ENERGEST_TYPE_LPM`
    - Radio transceiver `ENERGEST_TYPE_TRANSMIT, ENERGEST_TYPE_LISTEN`
  - There are `RTIMER_SECOND` ticks in one second (65.536 ticks for the TelosB)

Contiki's SW-based Energy Estimation

- Example: Monitoring how long the radio module has been active in send or receive mode

```
// Variables declaration
static unsigned long tx_ticks, rx_ticks;

// Starting the computation
rx_ticks = energest_type_time(ENERGEST_TYPE_LISTEN);
tx_ticks = energest_type_time(ENERGEST_TYPE_TRANSMIT);

/* --- CODE TO BE MEASURED --- */

// Finishing the computation
rx_ticks = energest_type_time(ENERGEST_TYPE_LISTEN) - rx_ticks;
tx_ticks = energest_type_time(ENERGEST_TYPE_TRANSMIT) - tx_ticks);
printf("Rx ticks: %lu, Tx ticks: %lu\n", rx_ticks, tx_ticks);

// Compute the energy consumption
…
```

Data types

int: 16 bits signed
long: 32 bits sig

- Integer numbers:
  - int and long are not portable: Depend on CPU.
  - 8 bits signed and unsigned: int8_t, uint8_t
  - 16 bits signed and unsigned: int16_t, uint16_t
  - 32 bits signed and unsigned: int32_t, uint32_t
- Booleans, characters and strings
  - Use uint8_t for Boolean and char
  - and uint8_t* for strings

int: 32 bits signed
long: 32 bits sig

Contiki Memory Management

- Memory allocation
  - `memb`: block memory allocation - statically allocated
    - used whenever N elements of some type is needed
    - heavily used in Contiki
  - `memm`: managed memory (indirection) - dynamic
    - almost unused in Contiki
  - `nbr-table`: centralized network neighbor table
    - ensures all modules keep information about the same neighbor set
    - used by IPv6, RPL state, routing entries, MAC, security
- Data structures
  - List: linked list
  - Ringbuf: ring buffer with atomic put/get

Contiki Resources

- Code
  - https://github.com/contiki-ng/contiki-ng
- Docs
  - https://github.com/contiki-ng/contiki-ng/wiki
- Setup with Docker
  - https://github.com/contiki-ng/contiki-ng/wiki/Docker
- Tutorials
  - https://github.com/contiki-ng/contiki-ng/wiki#tutorials

Demo

See Video

Questions?

Please stay safe and healthy!

Slides Credits

- This presentation borrows heavily from:
  Simon Duquennoy (SICS), Carlo Alberto Boano (TU Graz), Marco Zimmerling (TU Dresden), Alessandro Redondi (Politecnico Milano), Thiemo Voigt (SICS), Adam Dunkels (Things Square), Beshr Al-Nahas (Chalmers)