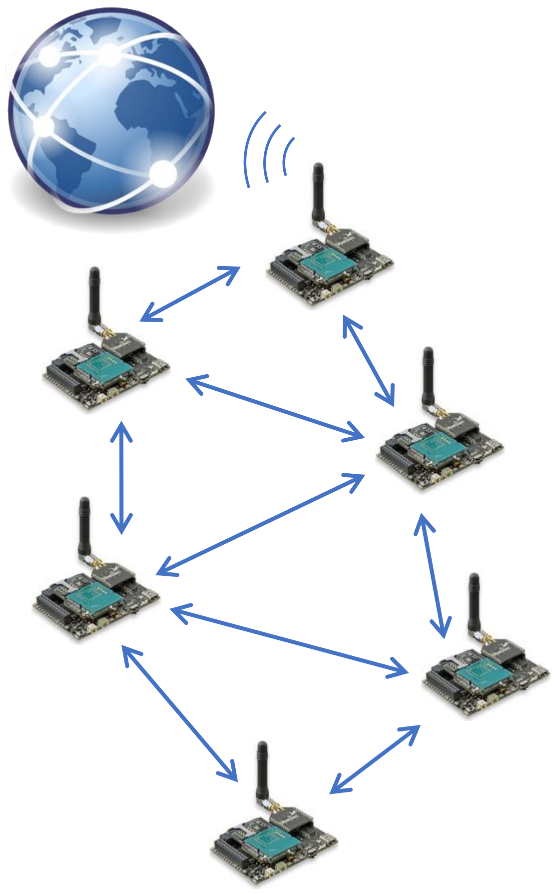
Low-power Wireless Networking for the Internet of Things

Lab2: Events, Protothreads and Rime

Matteo Trobinger (matteo.trobinger@unitn.it)



Credits for some slides to:

Timofei Istomin, Pablo Corbalán, Ramona Marfievici, Carlo Alberto Boano

Event timers and callback timers

```
#define PERIOD CLOCK_SECOND
```

Event timers

```
static struct etimer timer; // event timer structure
etimer_set(&timer, PERIOD); // start timer
PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));
```

Wait until an event occurs and the condition is true

Callback timers

```
static struct ctimer timer;
ctimer_set(&timer, PERIOD, timer_cb, ptr);
```

Function to be called

Pointer passed to the function; can be NULL

Events in Contiki

Why? Contiki is an *event-driven* OS, processes only run when an event occurs

Events in Contiki

```c

#define PERIOD_ON (CLOCK_SECOND / 10)
#define PERIOD_OFF (CLOCK_SECOND * 9 / 10)
/*---------------------------------------------------------------------------*/
// Declare a process
PROCESS(hello_world_process, "Hello world process");
// List processes to start at boot
AUTOSTART_PROCESSES(&hello_world_process);
/*---------------------------------------------------------------------------*/
// Implement the process thread function
PROCESS_THREAD(hello_world_process, ev, data)
{
  // Timer object
  static struct etimer timer; // ALWAYS use static variables in processes!

  PROCESS_BEGIN();            // All processes should start with PROCESS_BEGIN()

```

Events in Contiki

Why? Contiki is an *event-driven* OS, processes only run when an event occurs

ID of the process

Current event

Event data

```
PROCESS_THREAD(demo_process, ev, data){
 PROCESS_BEGIN()
 // ...
 PROCESS_WAIT_EVENT();
 // ev and data are updated
 if(ev==PROCESS_EVENT_TIMER && etimer_expired(&timer))
 {
  //...
 }
 else if (ev==some_other_event)
 {
  //...
 }
 PROCESS_END()
}
```

When the process continues `ev` and `data` contain info about the event that woke up the process

An event from some timer

Check that it was "our" timer

Other events defined in `contiki-uwb/contiki/core/sys/process.h` e.g., PROCESS_EVENT_INIT, PROCESS_EVENT_POLL, PROCESS_EVENT_EXIT...

Custom events

You can create your own event types!

1. **Define your event:**

```
process_event_t alarm_event;
```

2. **Allocate the event:**

```
PROCESS_THREAD(demo_process, ev, data) {
  PROCESS_BEGIN();
  alarm_event = process_alloc_event();
  //...
  PROCESS_END();
}
```

Signaling events

Send an event to process(es):

Destination process

Event type

```
process_post(&a_process, alarm_event,
             &alarm_event_data);
```

Any pointer (or NULL)
It will be passed to the processes

Send to all processes in the system

```
process_post(PROCESS_BROADCAST,
             alarm_event,
             &alarm_event_data);
```

Process switching in Contiki

What is the system doing while a process is waiting for an event?

`PROCESS_WAIT_EVENT();`

It might be running other processes or sleeping if all processes are waiting!

`PROCESS_WAIT_EVENT()` **and similar calls are the only way of switching between processes in Contiki!**

- No preemption between processes
    - Except for interrupts
- These calls give the control back to the system
- If one of your processes does not do that, it will be the only one running (and the system will never sleep) until the process finishes

No need to worry about race conditions between processes

Wait for a specific event (condition)

```
PROCESS_WAIT_EVENT(); // waits for ANY event

// Often we need a specific event (or condition)
do {
   PROCESS_WAIT_EVENT();
} while (!(ev==PROCESS_EVENT_TIMER &&
        etimer_expired(&et)) )

// There is a shorter way of doing exactly that:
PROCESS_WAIT_EVENT_UNTIL(ev==PROCESS_EVENT_TIMER &&
                              etimer_expired(&et) );

// It still wakes up on any event, but goes back to sleep
if the condition is not true
```

Any condition

Prototthreads

Process: high-level, event-driven task managed by the Contiki kernel
Protothread: low-level, memory efficient mechanism that provide a process with a thread-like, sequential control flow

Processes in Contiki are based on *protothreads*

Protothreads are just weird functions:

- They have multiple entry points (like *coroutines*)
- They memorize the point where they exited the last time and start from that point the next time they are called
- Implemented with a (hackish) use of C preprocessor macros and **switch…case** statements

```
PROCESS_THREAD(some_process, ev, data){
 PROCESS_BEGIN()
 // do something
 PROCESS_WAIT_EVENT();
 // do something more
 PROCESS_WAIT_EVENT();
 // do something more
}
```

Here the process function exits (returns)

Here it will continue when called again

Contiki's process scheduler

```
while (1) { // pseudocode!
  sleep();
  // Woken up by something
  while (!event_queue_empty()) {
    event, data = event_queue_pop();
    for (p: all_processes) {
      if (destination(event) == p ||
          destination(event) == BROADCAST) {
        p(event, data);
      }
    }
  }
}
```

Interrupt handlers may generate events (put to the event queue)

Call the protothread of the process **p**, passing it the event and its data

Processes may generate events, too. They will be handled in the following iterations of the scheduler loop

So far

- Contiki processes

```
PROCESS_THREAD(name, ev, data) {
 PROCESS_BEGIN();
 ...
 PROCESS_END();
}
```

- **Cooperative behavior** through `PROCESS_WAIT_EVENT()` and similar calls

- Communication between processes with **events and event data**

- Timers
  - **etimer** (event timer) → post the event `PROCESS_EVENT_TIMER` to the process that set the timer when it expires
  - **ctimer** (callback timer) → calls the specified function when the timer expires

Next: communication **between devices**

The Contiki Network Stack

Abstraction of the radio

RADIO Layer: `NETSTACK_RADIO`

- 2.4GHz IEEE 802.15.4 (several radios: CC2420, CC2538, CC2650, etc.)
- Sub-1GHz IEEE 802.15.4g (CC1350)
- Limited support for BLE (CC2650 and CC1350)

When radio is on

RDC Layer: `NETSTACK_RDC`

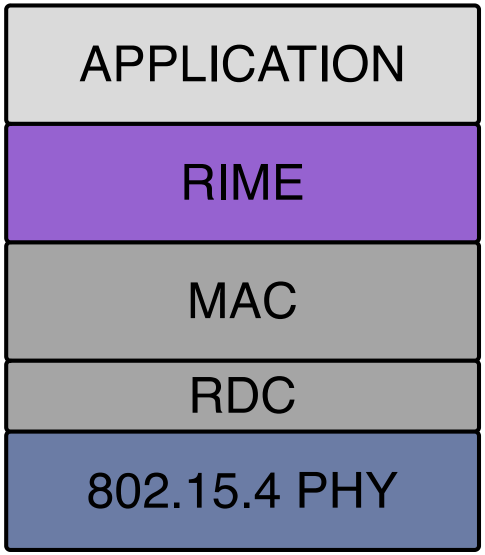
- NullRDC, ContikiMAC, X-MAC

MAC Layer: `NETSTACK_MAC`

- NullMAC, CSMA, TSCH

Network Layer: `NETSTACK_NETWORK`

- **Rime**:
  - several communication primitives
  - protocols atop (e.g., Collect)
- **uIPv6:**
  - RPL
  - TCP/UDP
  - CoAP, HTTP, etc.



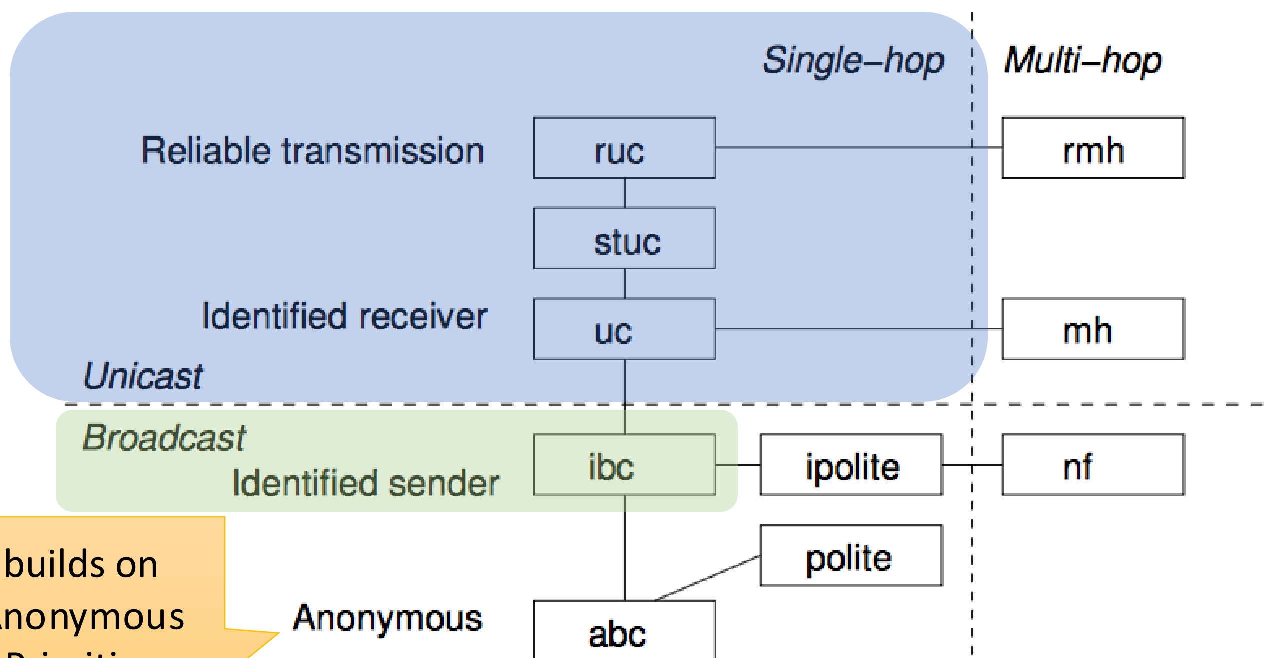
The Rime Stack

Layered network architecture:

- Simplified protocol implementation
- Each layer is a communication abstraction (a primitive)
- Protocols are built on top of a set of these primitives

Next Lab

Today



Everything builds on top of the Anonymous Broadcast Primitive

The Rime Stack: Addresses

Rime Address: 2 Bytes

- Short address of IEEE 802.15.4: provided by the radio chip
- E.g.: 143.130, 87:BB

What's my Rime address?

- Set in: `linkaddr_node_addr`
- To print your address:

```
printf("Node address = %02X:%02X\n",
       linkaddr_node_addr.u8[0],
       linkaddr_node_addr.u8[1]);
```

Using Rime primitives

Before you can send or receive data, you **should initialise the primitive** you want to use — in terms of Rime, ***open a connection***.

```
broadcast_open(
    &connection,
    CHANNEL,
    &callbacks)
```

A data structure (connection object) holding the state associated with the connection. You should have it defined as a global variable

Just a number to distinguish among several connections that are used in the program (similar to a UDP port)

A structure holding pointers to *callbacks*: functions of **your** program that Rime will call to inform you about something related to this connection. You should define this structure as a global variable and assign pointers to your functions to the elements of the structure.

Rime broadcast API (broadcast.h)

```
/* Open a Broadcast connection */
void broadcast_open(
struct broadcast_conn *c,
uint16_t channel,
const struct broadcast_callbacks *u);

/* The callbacks structure */
struct broadcast_callbacks {
  void (*recv)(struct broadcast_conn *conn, const linkaddr_t *sender);
  void (*sent)(struct broadcast_conn *conn, int status, int num_tx);
}
/* Close a Broadcast connection */
void broadcast_close(struct broadcast_conn *c);

/* Send a Broadcast packet */
int broadcast_send(struct broadcast_conn *c);
```

Packet received callback

Packet sent callback

Example: using Rime broadcast

```
/* Open a Broadcast connection */
void broadcast_open(
struct broadcast_conn *c,
uint16_t channel,
const struct broadcast_callbacks *u);

/* The callbacks structure */
struct broadcast_callbacks {
  void (*recv)(struct broadcast_conn *conn, const linkaddr_t *sender);
  void (*sent)(struct broadcast_conn *conn, int status, int num_tx);
}
/* Close a Broadcast connection */
void broadcast_close(struct broadcast_conn *c);

/* Send a Broadcast packet */
int broadcast_send(struct broadcast_conn *c);
```

```
struct broadcast_conn my_bcast_conn;

void my_recv(struct broadcast_conn *conn, const linkaddr_t *sender) {
  // ... Your code here
}
```

The definition of ***your*** **receive** callback function

```
struct broadcast_callbacks my_cb = {.recv = my_recv, .sent = NULL};
```

Pointer to the callback function

NULL = not interested

```
// ... Somewhere in a process:
broadcast_open(&my_bcast_conn, 123, &my_cb);
```

Open the connection

```
// ... When you need to send data
broadcast_send(&my_bcast_conn);
```

What Packet?

The Packet Buffer (packetbuf.h)

A single shared buffer for incoming and outgoing packets

- Motivation: to reduce memory footprint
- To queue data: **queuebuf** `(core/net/queuebuf)`

To send a packet:

1. Clear packetbuf: **`packetbuf_clear()`**`;`
2. Copy your message to packetbuf:

```
packetbuf_copyfrom(
    const void *from,  /* Message to copy */
    uint16_t len);  /* Message length */
```

3. Send message: **`broadcast_send`**`(&my_bcast_conn);`

Note: These steps must be carried out one after the other! Other processes or the radio driver could overwrite the packetbuf if your process is suspended

Receiving a packet

Rime will call it when a packet arrives on the specified channel

```
void my_recv(struct broadcast_conn *conn, const linkaddr_t *sender)
{
  // ... Your code here
}
```

When a packet has arrived: useful functions

1. The received data length: `packetbuf_datalen()`
2. Pointer to the received data in packetbuf:
   `void* packetbuf_dataptr();`
3. Use `memcpy(void *to, void *from, int length)`
   function to retrieve data from the packetbuf

Code template

Download and unzip the provided code

- unzip `Lab2.zip`

Go to the code directory

- `cd Lab2/bcast-exercise`

  ⟶ The file you should edit is `node.c`

Compile:

- `make` or `make TARGET=sky`

When everything seems to work, run it in Cooja:

- `cooja sim.csc &`

Exercise – Part one

Description

- Implement a sender/receiver using the Rime broadcast primitive.
- A template with TODOs is provided (`node.c`).

Implement the sender and the receiver

- Initialize the Rime broadcast primitive.
- Prepare and send a broadcast message with a string "Hello World!" every 5–8 seconds.
- Implement the `broadcast_recv`: print every received message, alongside with the address of the sender node.

If everything works ...

- Implement the sent callback (`broadcast_sent`): print the status and number of TX after every message TX.

Hints

Creating the text to embed in the packet

- `char msg[] = "Hello World!";`
- `strlen(msg);` gives the number of characters

Reading the received text

- `size_t len = packetbuf_datalen();` to find out the lenght of the message
- `char msg[len + 1];` allocates an array of the proper size.
- Why `len + 1`? Remember a string must end with `'\0'`!
- If your receiver prints unexpected characters, make sure you are terminating the string correctly with `'\0'`!

Exercise – Part two

Extra features

- You can send 2 different messages, e.g., “Hello from Matteo!” or “Hello from XX!”;
- Keep sending ***<u>the same</u>*** message every 5-8 seconds (e.g., “Hello from Matteo!”) …
- … Until a button is pressed. Upon this event, start sending ***<u>the other</u>*** message every 5-8 seconds …
- … Until a button is pressed. This event will make you change message again.

The button is a sensor

- An additional event is assigned to sensors: `sensors_event`
- Activate the button right after PROCESS_BEGIN() with `SENSORS_ACTIVATE(button_sensor);`
- `button_sensor`, the event data that comes with the event, is referenced in dev/button-sensor.h (included)

Make sure the sensor event is the one you expect by checking that `data==&button_sensor`

How to press a button in Cooja