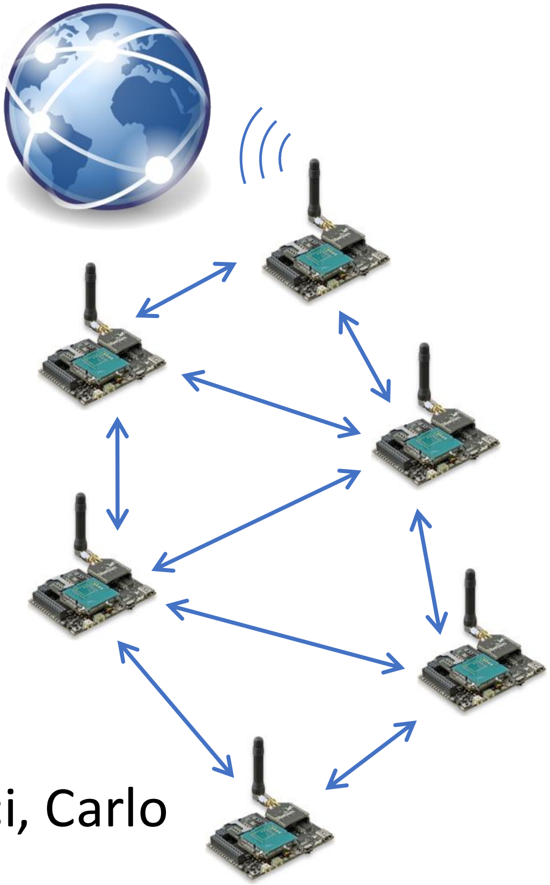
Low-power Wireless Networking for the Internet of Things

Lab3: The Rime network stack

Matteo Trobinger (matteo.trobinger@unitn.it)



Credits for some slides to:

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

Lab 2 — Recap

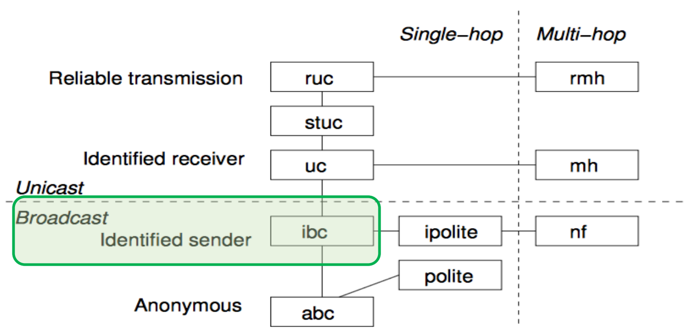
- Events, processes, and protothreads

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

switch between processes

- Processes are **cooperative**
- Events govern the execution of processes

- The Rime network stack: layered architecture based on simple communication primitives



- Broadcast primitive
  - Open a connection: `broadcast_open(&connection, channel, &callbacks)`
  - Implement the `SENT` and `RECEIVED` callbacks

Function Pointers in C

```
#include <stdio.h>
void sum(int a, int b){
  printf("Sum: %d\n", a+b);
}

void diff(int a, int b){
  printf("Diff: %d\n", a-b);
}

/* Function pointer declaration*/
void (*fp)(int, int);

int main() {
  fp = sum;
  fp(5, 3);
  return 0;
}
```

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

```
/* The callbacks structure (broadcast.h) */
struct broadcast_callbacks {
  void (*recv)(struct broadcast_conn *conn,
               const linkaddr_t *sender);
  void (*sent)(struct broadcast_conn *conn,
               int status, int num_tx);
}
```

Function pointer: a variable that stores the memory address of a function. It allows to call a function indirectly through the variable

OUTPUT:
Sum: 8

Function Pointers in C

```c
#include <stdio.h>
void sum(int a, int b){
  printf("Sum: %d\n", a+b);
}

void diff(int a, int b){
  printf("Diff: %d\n", a-b);
}

/* Function pointer declaration*/
void (*fp)(int, int);

int main() {
  fp = diff;
  fp(5, 3);
  return 0;
}
```

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

/* The callbacks structure (broadcast.h) */
struct broadcast_callbacks {
  void (*recv)(struct broadcast_conn *conn,
               const linkaddr_t *sender);
  void (*sent)(struct broadcast_conn *conn,
               int status, int num_tx);
}
```

Function pointer: a variable that stores the memory address of a function. It allows to call a function indirectly through the variable

OUTPUT:
```
Diff: 2
```

Function Pointers in C

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

/* The callbacks structure (broadcast.h) */
struct broadcast_callbacks {
  void (*recv)(struct broadcast_conn *conn,
               const linkaddr_t *sender);
  void (*sent)(struct broadcast_conn *conn,
               int status, int num_tx);
}
```

```
#include <stdio.h>
void sum(int a, int b){
  printf("Sum: %d\n", a+b);
}

void diff(int a, int b){
  printf("Diff: %d\n", a-b);
}
```

```
void math_operation(int a, int b, void (*fp)(int, int)){
  fp(a, b);
}
```

```
int main() {
  math_operation(5, 3, sum);   → Sum: 8
  math_operation(5, 3, diff);  → Diff: 2
  return 0;
}
```

Callback Timer (ctimer): API

```
/* Start the callback timer */
void ctimer_set(
  struct ctimer *c,
  clock_time_t t,
  void(*f)(void *), /* The CALLBACK FUNCTION */
  void *ptr); /* Data Pointer */
```

`contiki-uwb/contiki/core/sys/ctimer.h`

Callback Timer (ctimer)

Declaration:

- `static struct ctimer ct;`

How does it work?

- Calls a function when the timer expires (defined “callback”)
- Built on top of etimer

Usage in Contiki:

- Rime Stack: many primitives (e.g., stubborn unicast, collect, etc.)
- uIPv6: RPL, neighbor discovery and maintenance, etc.

Programming style: callback-based

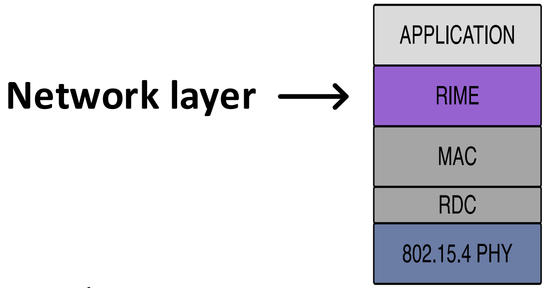
- Different from etimers (protothread-based) --- recall LAB1

Callback Timer (ctimer): API

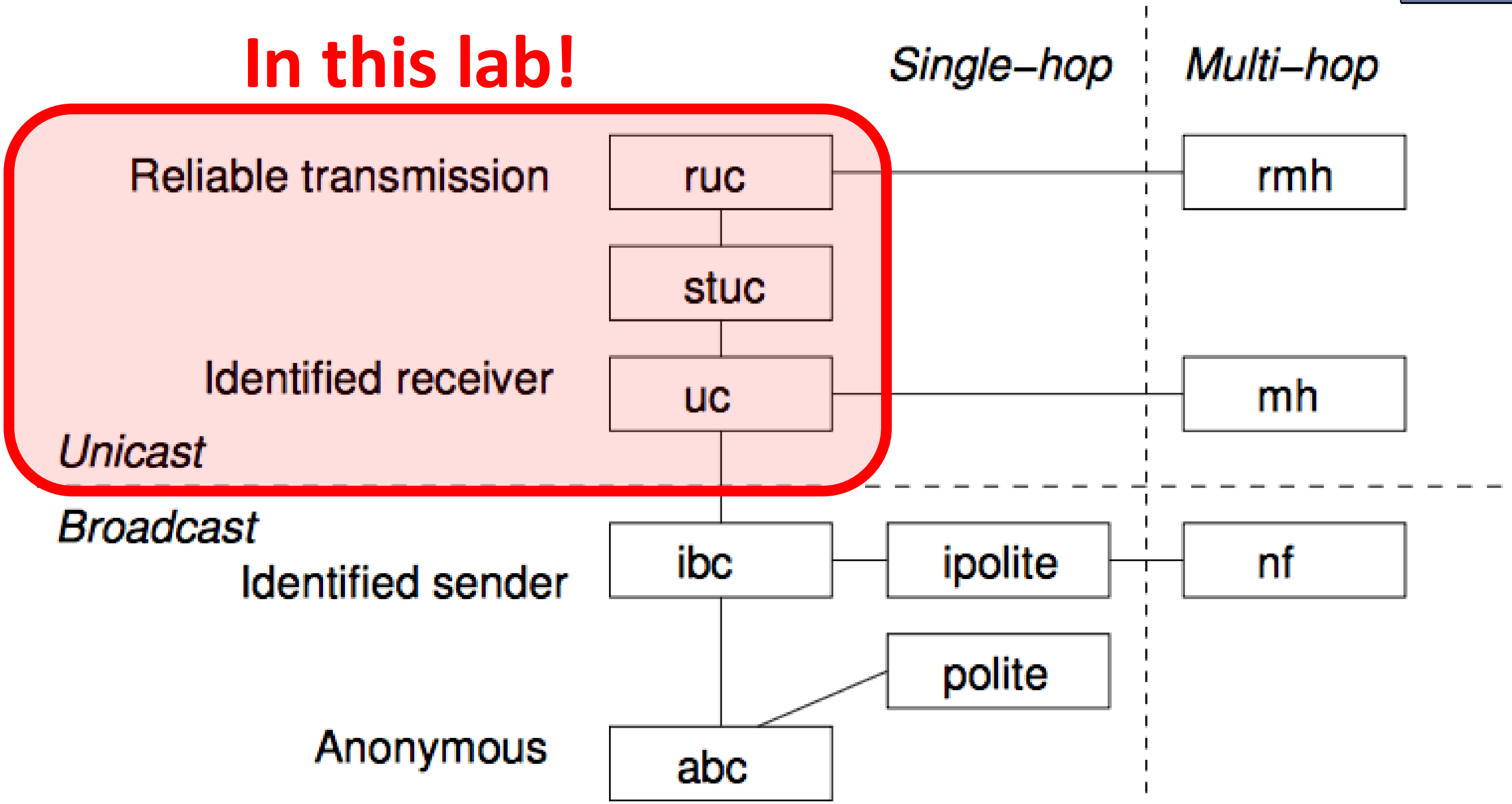
```
/* Start the callback timer */
void ctimer_set(
  struct ctimer *c,
  clock_time_t t,
  void(*f)(void *), /* The CALLBACK FUNCTION */
  void *ptr); /* Data Pointer */

/* Restart timer from the previous expiration time */
void ctimer_reset(struct ctimer *t);

/* Restart the timer from current time */
void ctimer_restart(struct ctimer *t);

/* Stop the timer */
void ctimer_stop(struct ctimer *t);

/* Check if the timer has expired */
int ctimer_expired(struct ctimer *t);
```

The Rime Stack

Contiki Network Stack

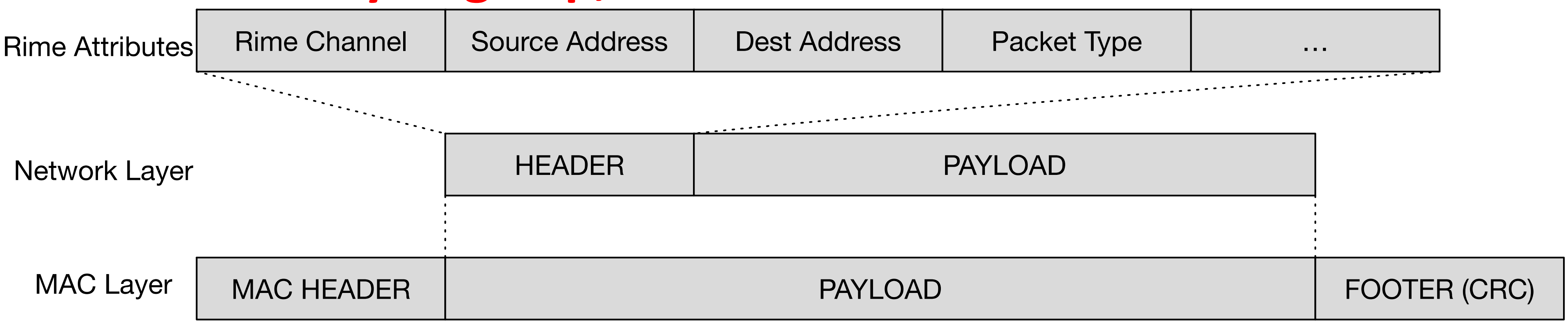


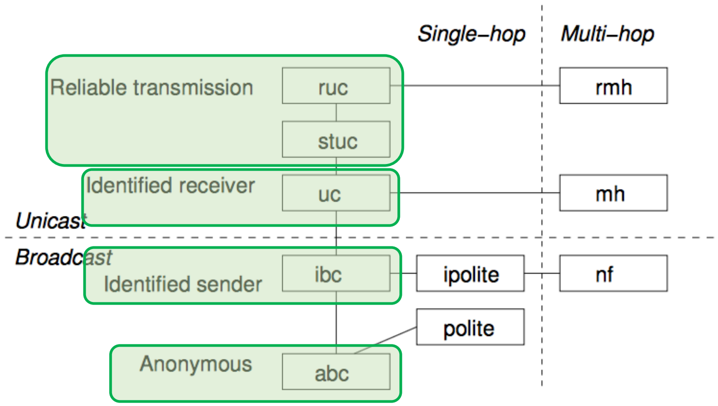
In this lab!



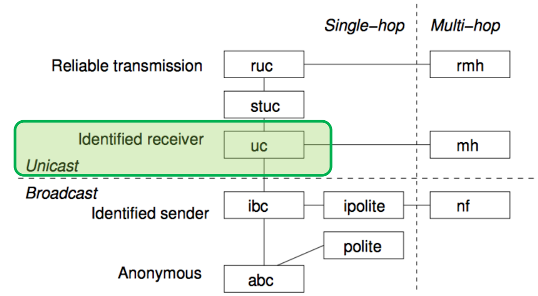
The Rime Stack: Protocol Headers

Rime Attributes change depending on the primitive!
The more you go up, the more attributes will be added!





The Unicast Primitive



How does it work?

- Built on top of broadcast primitive by adding the **destination address**
- No network-layer reliability: *one* message sent, *no* retransmissions

Added Rime attributes:

- Receiver address attribute: `PACKETBUF_ADDR_RECEIVER`

Unicast Connection:

```
static struct unicast_conn uc;
```

Callbacks:

```
static const struct unicast_callbacks uc_callbacks = {
  .recv = recv_unicast,
  .sent = sent_unicast
};
```

The Unicast Primitive: Callbacks

```
struct unicast_callbacks {
  /* RECV called when a packet has been received */
  void (* recv)(
    struct unicast_conn *c,
    const linkaddr_t *from); /* Sender ADDRESS */

  /* SENT called after transmitting a packet */
  void (* sent)(
    struct unicast_conn *c,
    int status, /* From MAC Layer: TX_OK, COLLISION */
    int num_tx); /* From MAC Layer: number of TX */
};
```

The Unicast Primitive: API

```
/* Open a Unicast connection */
void unicast_open(
  struct unicast_conn *c,
  uint16_t channel, /* Similar to TCP port */
  const struct unicast_callbacks *u);

/* Close a Unicast connection */
void unicast_close(struct unicast_conn *c)

/* Send a Unicast packet */
int unicast_send(
  struct unicast_conn *c,
  const linkaddr_t *receiver);
```

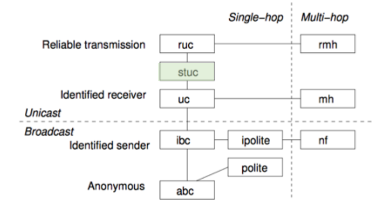
You need to specify the receiver address!

The Unicast Primitive: Example

```
static struct unicast_conn uc;

void my_recv(struct unicast_conn *conn, const linkaddr_t *from) {
  /* Your code here... */
}

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


PROCESS_THREAD(my_process, ev, data)
{
  PROCESS_BEGIN();
  unicast_open(&uc, 146, &my_cb);

  while(1) {
    /* Do something or wait for events */
    unicast_send(&uc, &dest); /* Send packet to dest */
  }
  PROCESS_END();
}
```

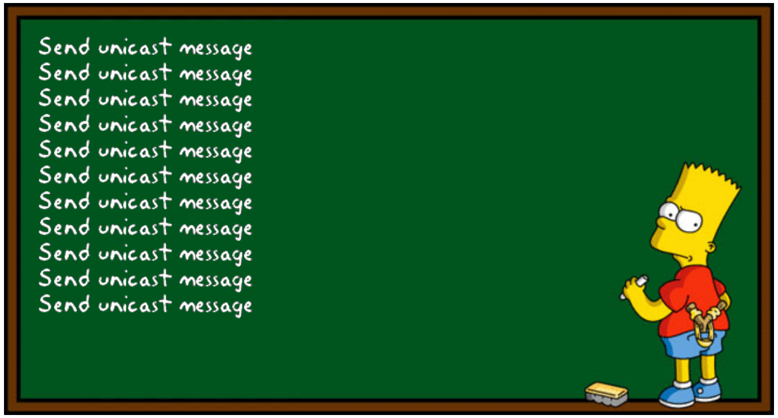
NULL = not interested

Stunicast: The Stubborn Unicast Primitive



Why stubborn?

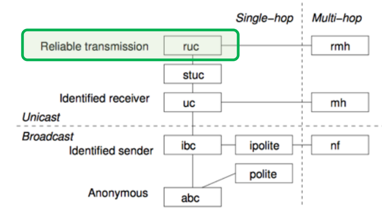
- **Repeatedly** send a packet until an upper layer (e.g., **runicast**) cancels the transmission



In a nutshell:

- Built on top of unicast
- Rime attributes: same as unicast
- Stunicast uses:
  - **`queuebuf`** to store the packet and its attributes
  - **`ctimers`** to schedule packet transmissions
- Behavior: when the `ctimer` expires, `stunicast` copies the data from the `queuebuf` to the `packetbuf` and re-sends the packet

Adding Reliability: the Runicast Primitive



How does it work?

- Built on top of stunicast by adding ACKs and a stop
- **Reliable**: it sends a message until it receives an ACK or reaches the maximum number of retransmissions (configurable)

Added Rime attributes:

- Packet type: data or ACK
- Packet ID: seqno to match packets with ACKs

Declaration:

```
static struct runicast_conn runicast;
```

Callbacks:

```
static const struct runicast_callbacks ruc_callbacks = {
  recv_runicast, sent_runicast, timedout_runicast};
```

The Runicast Primitive: Callbacks

```
struct runicast_callbacks {

  /* RECV called when a packet has been received */
  void (* recv)(
    struct runicast_conn *c,
    const linkaddr_t *from,
    uint8_t seqno); /* Runicast SEQNO for ACKs */

  /* SENT called after successful packet TX */
  void (* sent)(
    struct runicast_conn *c,
    const linkaddr_t *to,
    uint8_t retransmissions);

  /* TIMEDOUT --- packet not received or properly ACK*/
  void (* timedout)(
    struct runicast_conn *c,
    const linkaddr_t *to,
    uint8_t retransmissions);
};
```

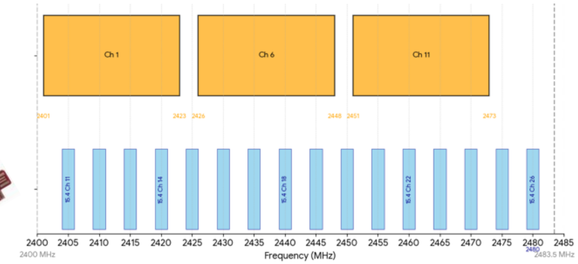
The Reliable Unicast Primitive: API

```
/* Open a Runicast connection */
void runicast_open(
  struct runicast_conn *c,
  uint16_t channel, /* Similar to TCP port */
  const struct runicast_callbacks *u);

/* Close a Unicast connection */
void runicast_close(struct runicast_conn *c);

/* Send a Runicast packet */
int runicast_send(
  struct runicast_conn *c,
  const linkaddr_t *receiver,
  uint8_t max_retransmissions);

/* Is Runicast transmitting a packet? */
uint8_t runicast_is_transmitting(struct runicast_conn *c);
```

Specify the **maximum** number of TXs, e.g., 4 or 8

RF Configuration

Wi-Fi



IEEE 802.15.4 (2.4 GHz):

- RF Channel: For narrowband devices, 16 channels available (from channel 11 to 26). So far channel 26 has been exploited (`project-conf.h`).
- TX power: E.g., you can TX at 0dBm (1mW)
- CSMA Clear Channel Assessment (CCA) Threshold

RF Configuration Parameters: `(contiki-uwb/contiki/core/dev/radio.h)`

- RF Channel: `RADIO_PARAM_CHANNEL`
- TX power: `RADIO_PARAM_TXPOWER`
- CCA Threshold: `RADIO_PARAM_CCA_THRESHOLD`

To read a value:

```
radio_value_t rfval;
NETSTACK_RADIO.get_value(RADIO_PARAM_CHANNEL, &rfval)
```

To set a value:

```
radio_value_t rfval = 26;
NETSTACK_RADIO.set_value(RADIO_PARAM_CHANNEL, &rfval)
```

Noise floor and RSSI

Noise floor: `RADIO_PARAM_RSSI`

```
radio_value_t rfval;
NETSTACK_RADIO.get_value(RADIO_PARAM_RSSI, &rfval);
```

[Last Packet] Received Signal Strength Indicator (RSSI): `RADIO_PARAM_LAST_RSSI`

```
radio_value_t rfval;
NETSTACK_RADIO.get_value(RADIO_PARAM_LAST_RSSI,&rfval);
```

⟶ E.g., to get a rough estimate of the link quality

Code Templates

Download and unzip the provided code

- `Unzip Lab3-exercise.zip`

Go to the code directory

- `$ cd Lab3-exercise/ping-pong-exercise`

The file you <u>should edit</u> is **uc-ctimer.c**

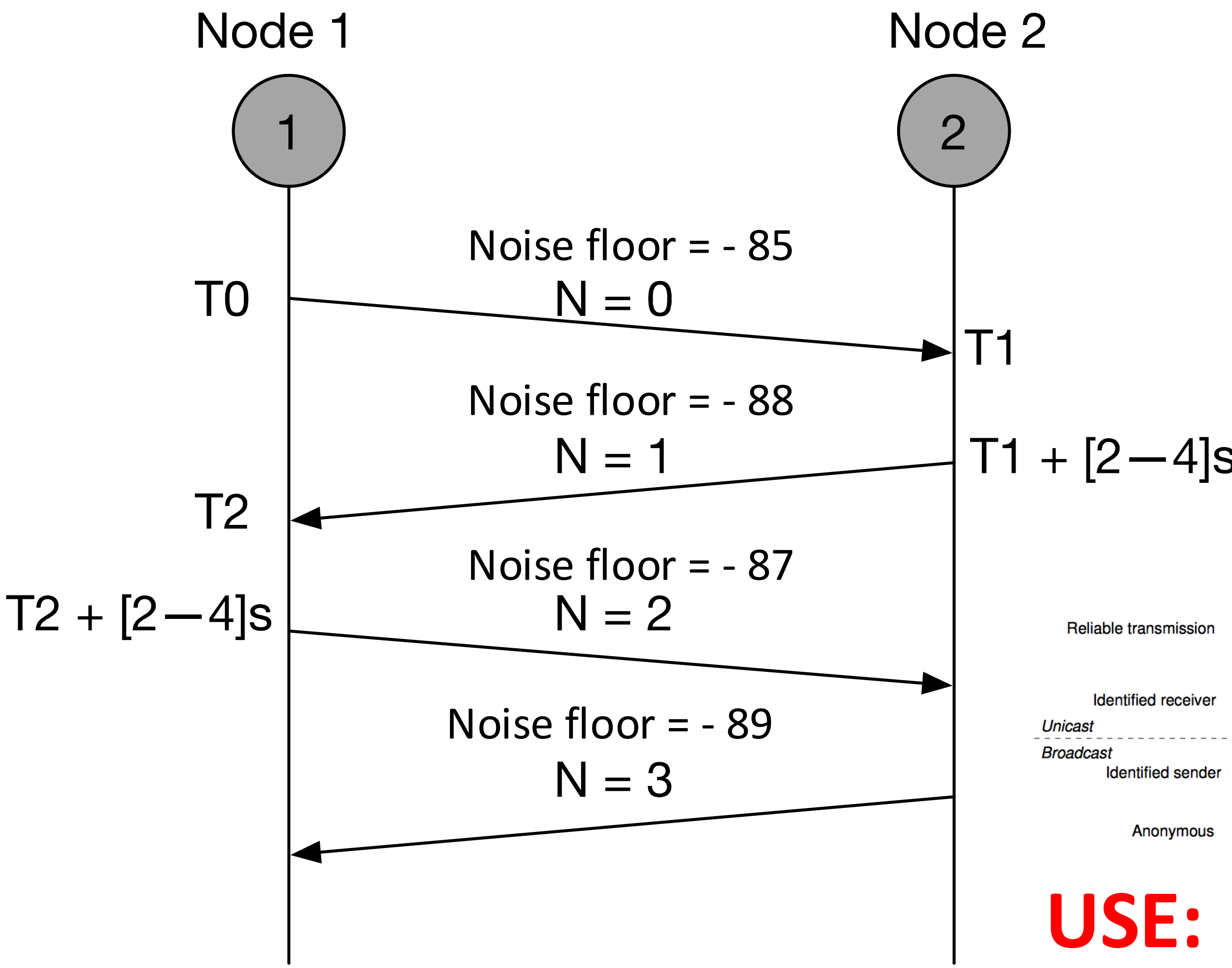
To compile:

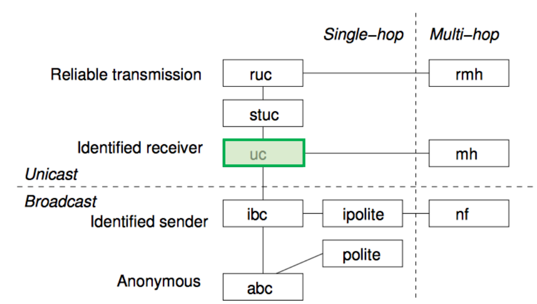
- `$ make` or `make TARGET=SKY`

When everything seems to work, test it in Cooja:

- `$ cooja` **`lab3-ctimer.csc`** `&`

Exercise: Ping-Pong Application





USE:

UNICAST and CTIMERS

Credit for the exercise idea to:
Ramona Marfievici

Exercise – packet structure

```
typedef struct ping_pong_msg {
  uint16_t sequence_number;
  int16_t noise_floor;
} ping_pong_msg_t;
```

Already provided in **uc-ctimer.c**, alongside other template code and several TODOs.
Check them all before start coding!!!

What to do?

- For the first message, set the `ctimer`, when it expires (i) fill in the packet with the `noise_floor` and the `ping_pong_number`, and (ii) send the packet
- Next, consider how (and where) you can use the `ctimer` to let the receiver react to the packet reception, allowing the ping-pong exchange to continue.

Hint:

- Initially set `noise_floor` to 0 and focus only on making the ping-pong application work
- Once it works, add the real noise floor to the message structure

Optional Exercise

If you don't remember something, check the Lab 2 slides!

Same application but with Etimers + Process Events

Instructions:

- Template code in `uc_etimer.c`
- Cooja simulation: `$ cooja lab3-etimer.csc &`

Hints: **(check them ONLY if you are lost)**

- Upon receiving a message: post a custom process event (`app_event`) to the main process
- Handling `app_event`: set an etimer with 2–4s timeout
- Etimer expiration: send the ping-pong message

Finished? Try with runicast!

Recap: Packet Buffer (packetbuf.h)

OPTIONAL!
packetbuf_copyfrom
clears the buffer anyway ☺

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: **unicast_send**(&uc, &dest);

To read a received packet:

1. 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) to retrieve data from the packetbuf

Suggested reading

1. A. Dunkels. **Rime — A Lightweight Layered Communication Stack for Sensor Networks**. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN),* 2007. PDF
2. A. Dunkels, F. Österlind, and Z. He. **An Adaptive Communication Architecture for Wireless Sensor Networks**. In *Proceedings of the 5th ACM Int. Conference on Embedded Networked Sensor Systems* (SenSys), 2007. PDF
3. **Contiki Wiki: Timers** **https://github.com/contiki-os/contiki/wiki/Timers**
4. **Contiki Wiki: Processes** **https://github.com/contiki-os/contiki/wiki/Processes**