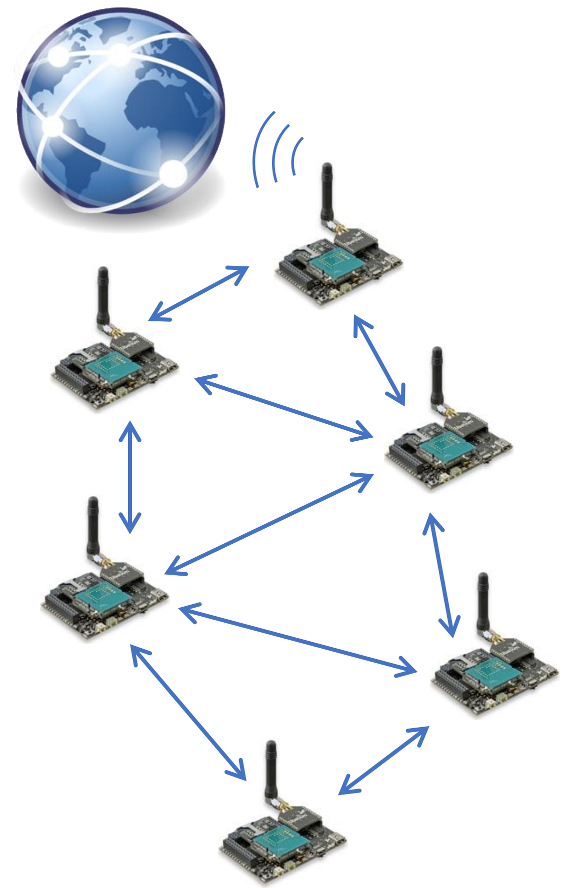
# Low-power Wireless Networking for the Internet of Things

## Lab (6 and) 7:
## Data collection with many-to-one routing



**Matteo Trobinger** (matteo.trobinger@unitn.it)

Credits for some slides to:

Pablo Corbalán, Timofei Istomin

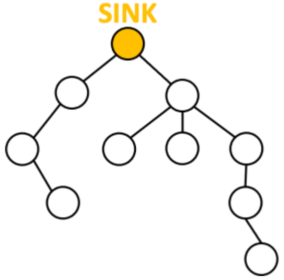
# Labs 6-7 Goal

Implement a *many-to-one routing-based data collection protocol* capable of collecting sensor readings from *any* node of a *multi-hop* wireless network towards a *single* intended destination, called the *sink*.

**Step 1 — Lab 6:**

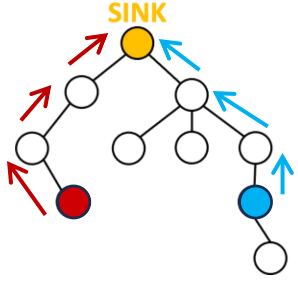
Acquire topology information: Building a routing tree

⟶ Routing metric: Hop count




**Step 2 — Lab 7:**

Leverage such routes to reliably forward data packets across the network, *from sources to the sink node*



# Lab 6 Recap:

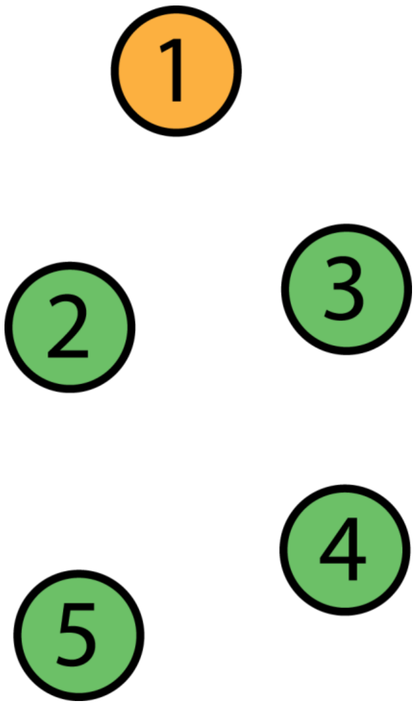
## How to build a routing tree for an unknown topology

# Lab 6 — Building the three (routing)

**S0:** Initially—when the application calls `my_collect_open()`—we have a *disconnected* network.
We need to:

1. Initialize our connection object
2. Open a connection with the underlying RIME primitives (broadcast and unicast)
3. Start the procedure to construct our routing tree **for the first time**

→ We need to build a routing tree with **the sink** (node 1) as the **root** (data collection point).

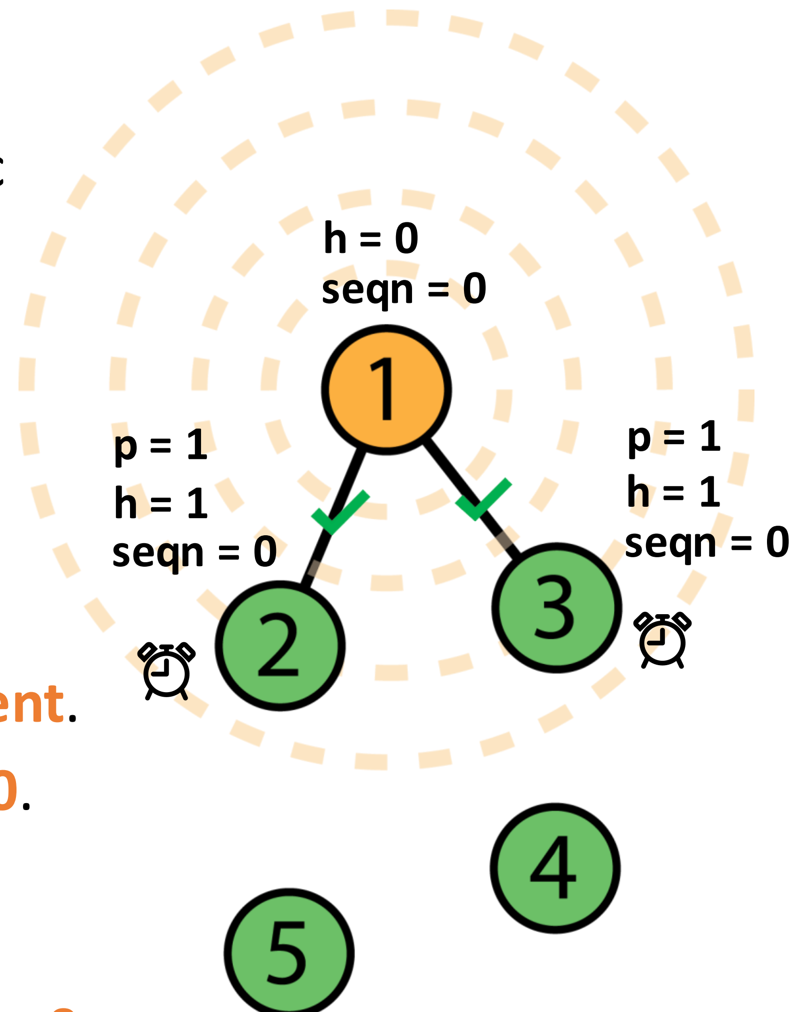


# Lab 6 — Building the three (routing)

**S1:** The sink sends a beacon message in broadcast with **seqn = 0** and metric **h = 0**.

**S2:** If RSSI of the beacon > threshold, nodes 2 and 3 compare h, seqn of the received message against their current metrics and join the network.

- Select node 1 (the sink) as their **parent**.
- Set their own metrics to **h=1**, **seqn=0**.
- Prepare themselves to send, after a **small random delay**, a new beacon message in broadcast with **h=1**, **seqn=0**.

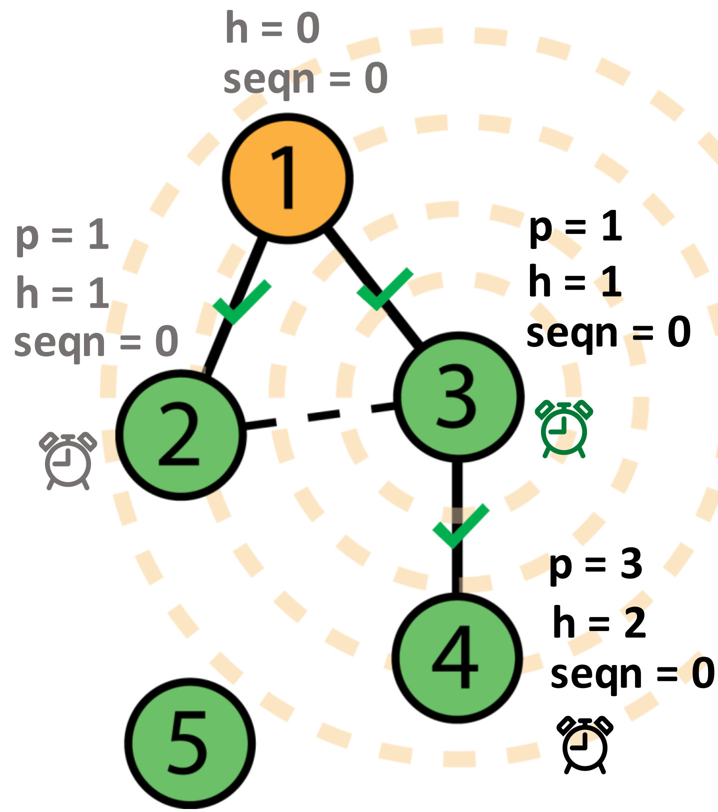


# Lab 6 — Building the three (routing)

**S3:** Node 3 sends a broadcast message with **seqn = 0** and metric **h = 1**.

Node 4 (i) checks RSSI > threshold, (ii) joins the network with metrics **h = 2**, **seqn = 0**, (iii) selects node 3 as the parent, and (iv) prepares itself to TX after a **small, random delay** a beacon message in broadcast.

Nodes 1, 2 **do not** update their routing information as their metrics **do not improve** by selecting node 3 as parent.

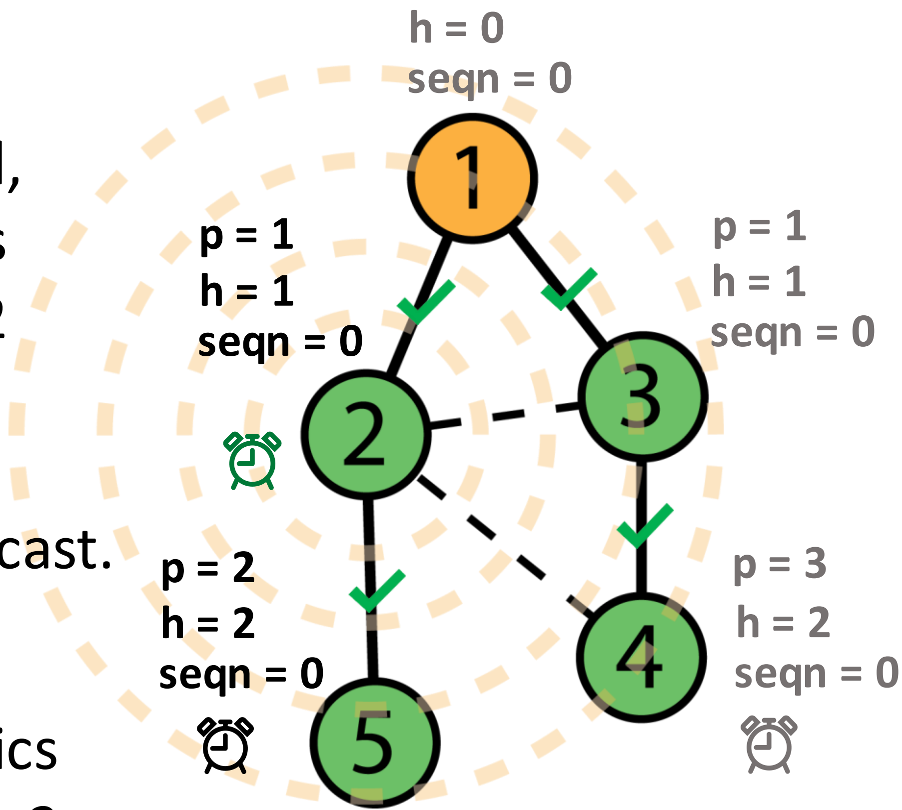


# Lab 6 — Building the three (routing)

**S4:** Node 2 sends a broadcast message with **seqn = 0** and metric **h = 1**.

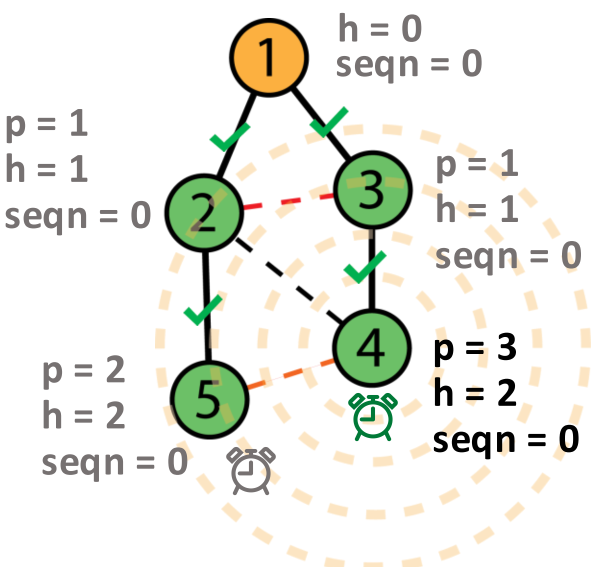
Node 5 (i) checks RSSI > threshold, (ii) joins the network with metrics **h = 2**, **seqn = 0**, (iii) selects node 2 as the parent, and (iv) prepares itself to TX after a **small, random delay** a beacon message in broadcast.

Nodes 1, 3, 4 **do not** update their routing information as their metrics **do not improve** by selecting node 2 as parent.

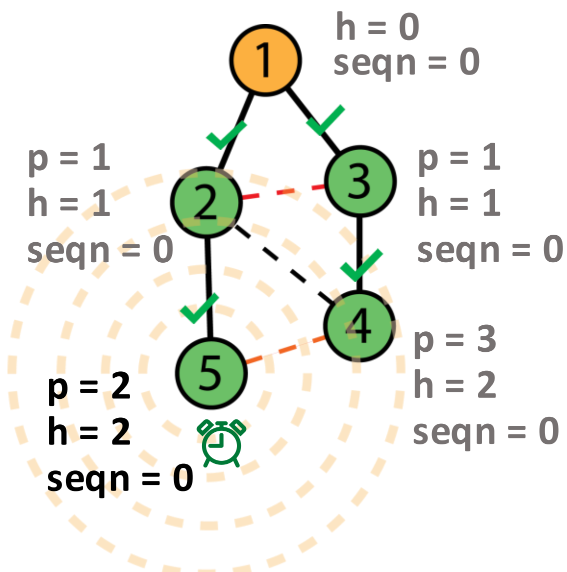


# Lab 6 — Building the three (routing)

**S5:** Node 4 sends a broadcast message with **seqn = 0** and metric **h = 2**.




**S6:** Node 5 sends a broadcast message with **seqn = 0** and metric **h = 2**.




In both cases **no routing changes** occur in the network!

# Lab 6 — Periodic tree updates

**Afterwards:** Refresh routes *periodically* by repeating the **whole process** *from scratch*.

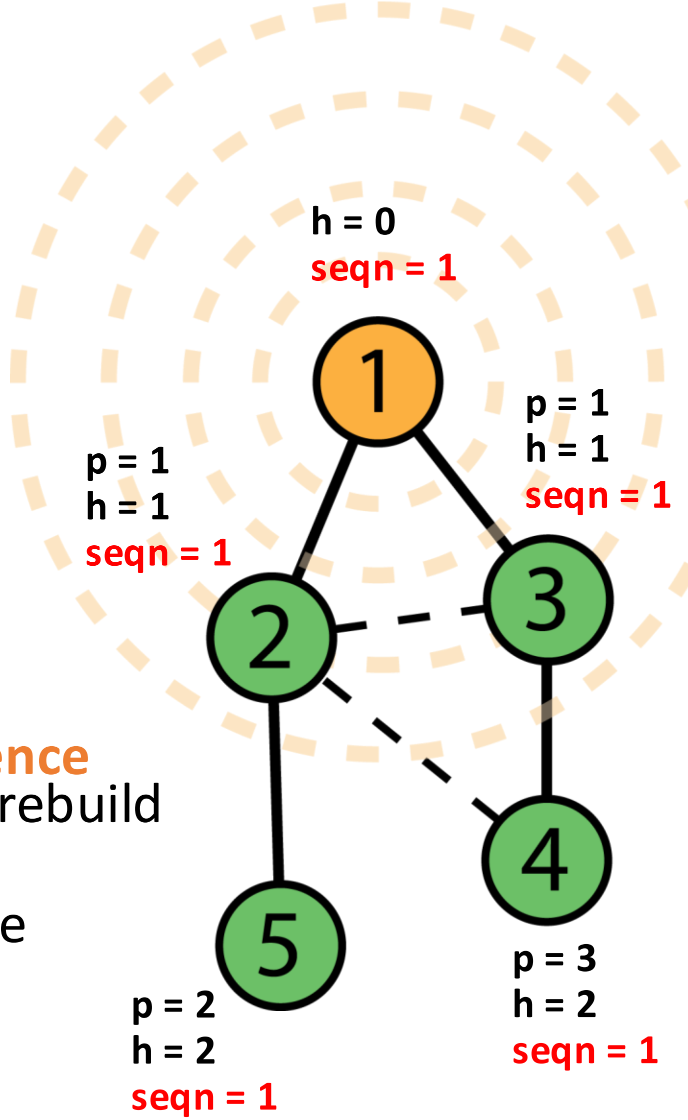
**Why?** To cope with potential network changes that could *drastically modify* our previously learned topology.

**How?**

**Sink:** *Periodically* **increases the beacon sequence number** and starts a *new beacon flood* to (re)rebuild the routing topology

**Nodes:** Upon receiving a **new** beacon message (i.e., seqn > highest seqn ever seen)

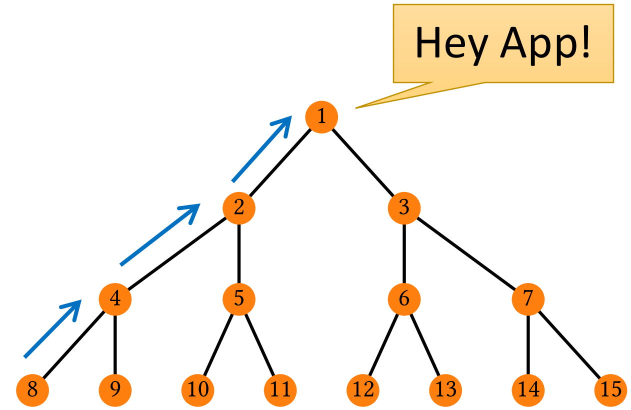
- Check RSSI > threshold,
- Accept the new metric (hop count, parent, …) **without** checking it against the old one, and
- After a small random delay TX the updated beacon message in broadcast



# Lab 7

## Let’s start forwarding!

# Implementing forwarding




<u>When</u> the routing is done,

implement the forwarding mechanism!

**You just need to complete two functions in my_collect.c:**

1. `my_collect_send()` — called periodically *on non-sink nodes* by the application (app.c) to start sending data towards the sink


   - *Prepare* the data packet
   - Send the packet to the source's parent in *unicast*

2. `uc_recv()` — called by the underlying unicast layer to inform the my_collect layer when a data packet arrives


   - <u>Forwarder (non-sink node)</u>: relay the received packet *to the node's parent* (again in unicast) to ensure forwarding progress
   - <u>Sink</u>: inform the application that a new data packet has been received (i.e., call `recv_cb`, the recv callback of my_collect)

# Forwarding at the data source — TODO 5

The application calls `my_collect_send()` to send data to the sink. This function should:

- Return **-1** (failure) if the node is not yet connected to the network
- Otherwise,

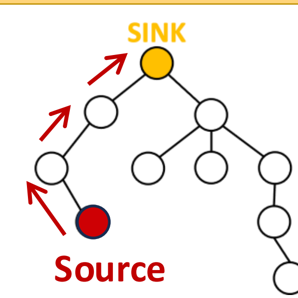
1. *Prepare* and *attach* a protocol header to the data packet;
2. Send the data packet to the source's parent in unicast
3. Return the status returned by the unicast send function

**Our protocol header:**

```
struct collect_header {
  linkaddr_t source;
  uint8_t hops;
} __attribute__((packed));
```

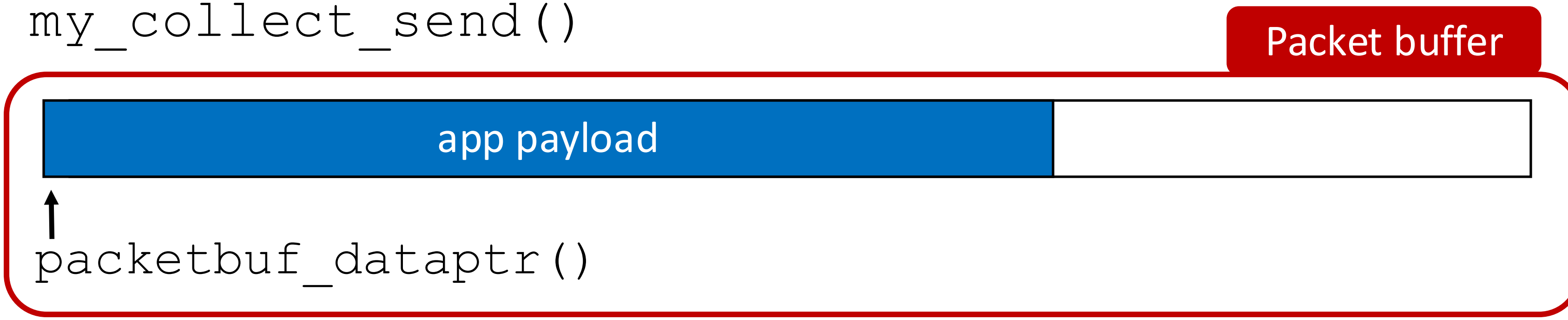
Originator (source) node address: *set* at the source node, ***never modified*** along the path

Hop count, set to 0 at the source, **incremented every time** the packet is **received**



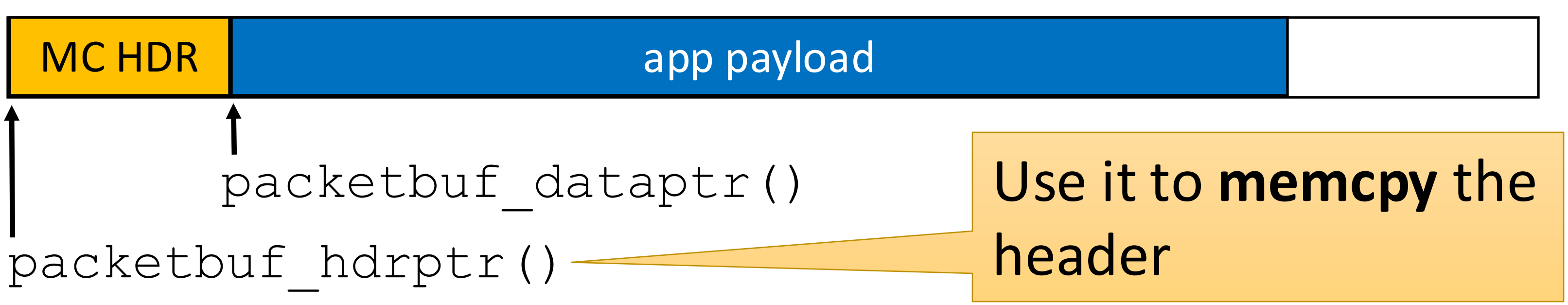
# Adding a header (at the source) — TODO 5

When the application sends a packet, it (i) **fills the packet buffer** with its message **starting from** `packetbuf_dataptr()`, and (ii) calls `my_collect_send()`




The collect layer needs to insert its <u>header</u> in the packet buffer, **<u>in front of</u>** the application payload. Towards this end we need to:

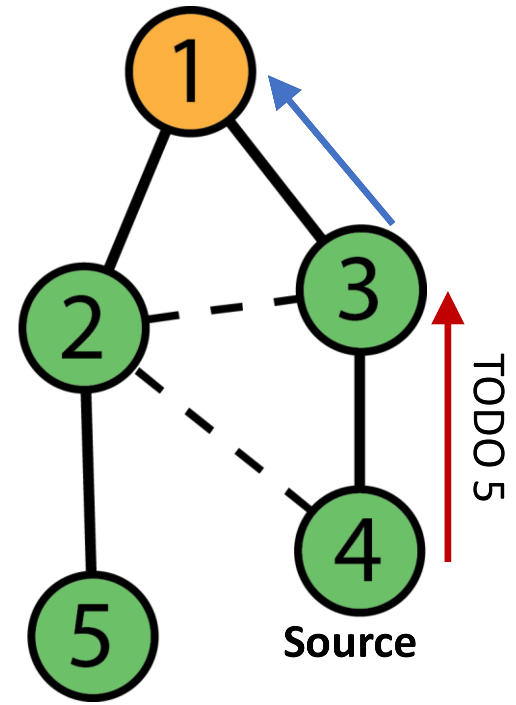
1. **Allocate** space for the header → call `packetbuf_hdralloc(size)`
2. **Insert** the header in the packet buffer. **How?**



# Forwarding at a relay node — TODO 6

When a **non-sink node** receives a data packet, it should **forward** it to its **parent in unicast:**

1. Extract the collection header from the received payload
2. Increment the hop count in the header
3. Put the *updated* header back in the packet buffer
4. Send the packet to its current parent using unicast



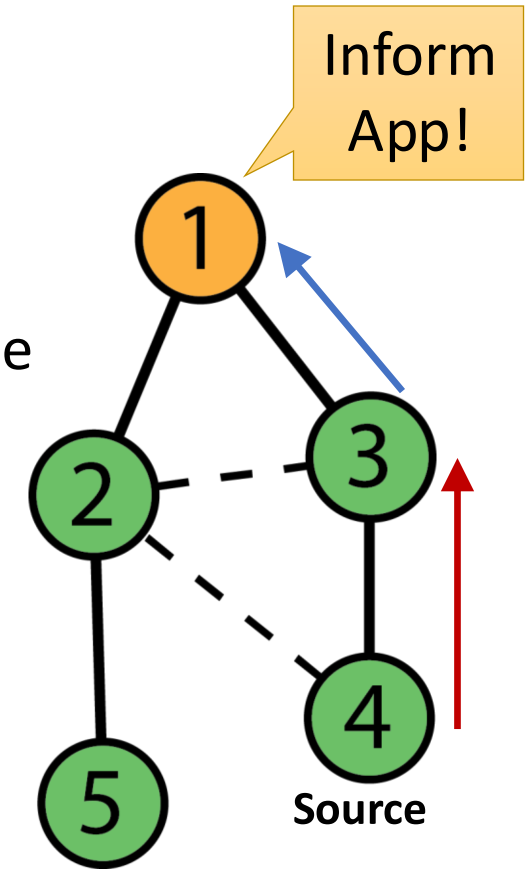

```
struct collect_header {
  linkaddr_t source;
  uint8_t hops;
} __attribute__((packed));
```

Set to 0 at the source. It must be incremented by 1 every time the packet is relayed

# Forwarding at the sink — TODO 6

When the **sink node** receives a data packet, it should (i) communicate the packet's source and hop count, and (ii) **deliver** the **app payload** to the **application:**

1. Extract the collection header from the received payload
2. Keep trace of the header's fields (source, hops) and update the hop count (hops + 1)
3. Remove the header from the payload
4. Call the *application* `recv` callback (`recv_cb`) to inform the application about the received data (app payload, source, hops)

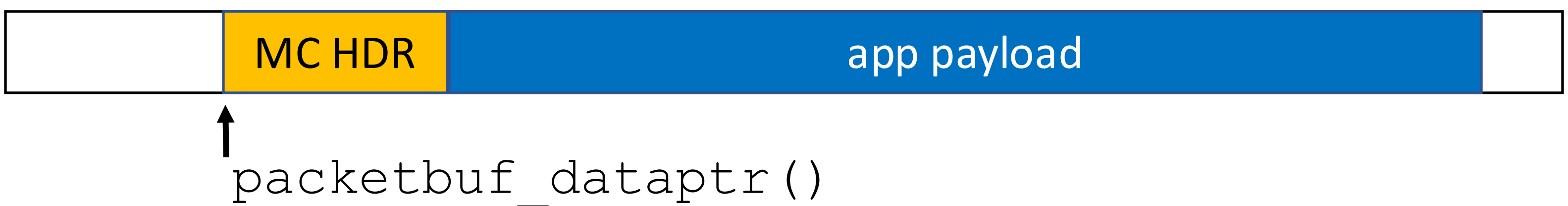



Q What is the "received payload"? How can we "*extract*" and/or "*remove*" our collection header from it?

# Extracting a header — TODO 6 (nodes & sink)

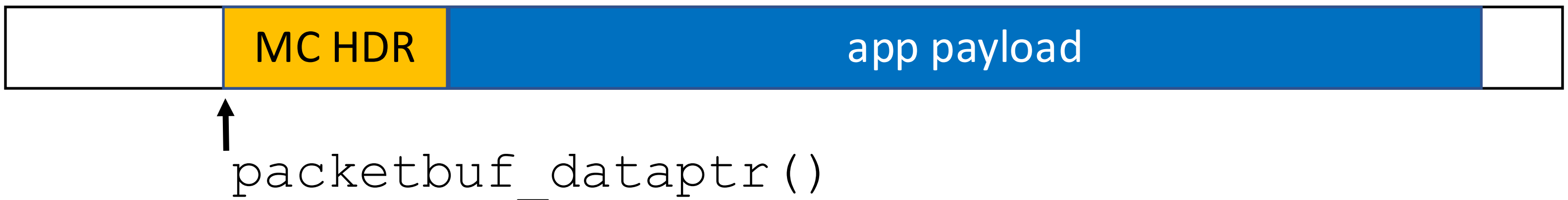
When we receive a packet from the underlying layer (`uc_recv()`), its payload starts at `packetbuf_dataptr()`


$Q_1$ *What is such* ***"payload"****?* It consists of both our **collection header** and the **application payload**.



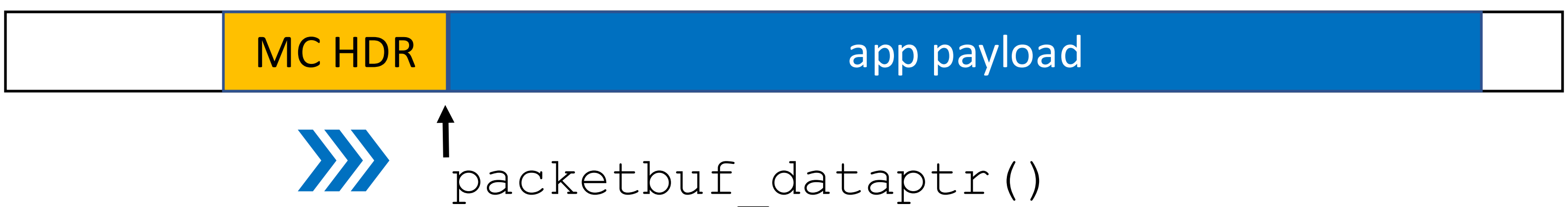

$Q_2$ *How can we* ***extract*** *the* ***collection header*** *from the received payload?* Let's just use the `packetbuf_dataptr()` pointer to `memcpy` the header!

# Removing a header — TODO 6 (sink only)




$Q_3$ *How can we* ***remove*** *the* ***collection header*** *from the received payload?* Let's use **`packetbuf_hdrreduce(size)`**

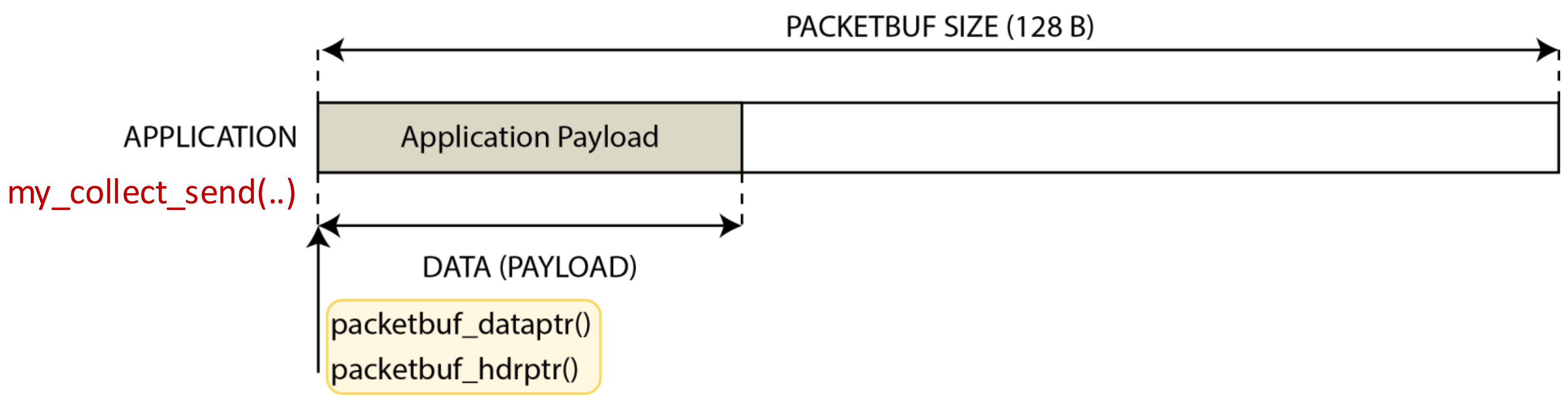
→ Shift the `packetbuf_dataptr()` pointer to the right of "size" bytes, thus making it point to the beginning of the app payload




**N.B.:** If the *application* `recv` callback is called *after* having removed the collection header, by looking at `packetbuf_dataptr()` the application can *directly get* the app payload!

# Packetbuf Management (<u>SENDING</u>)

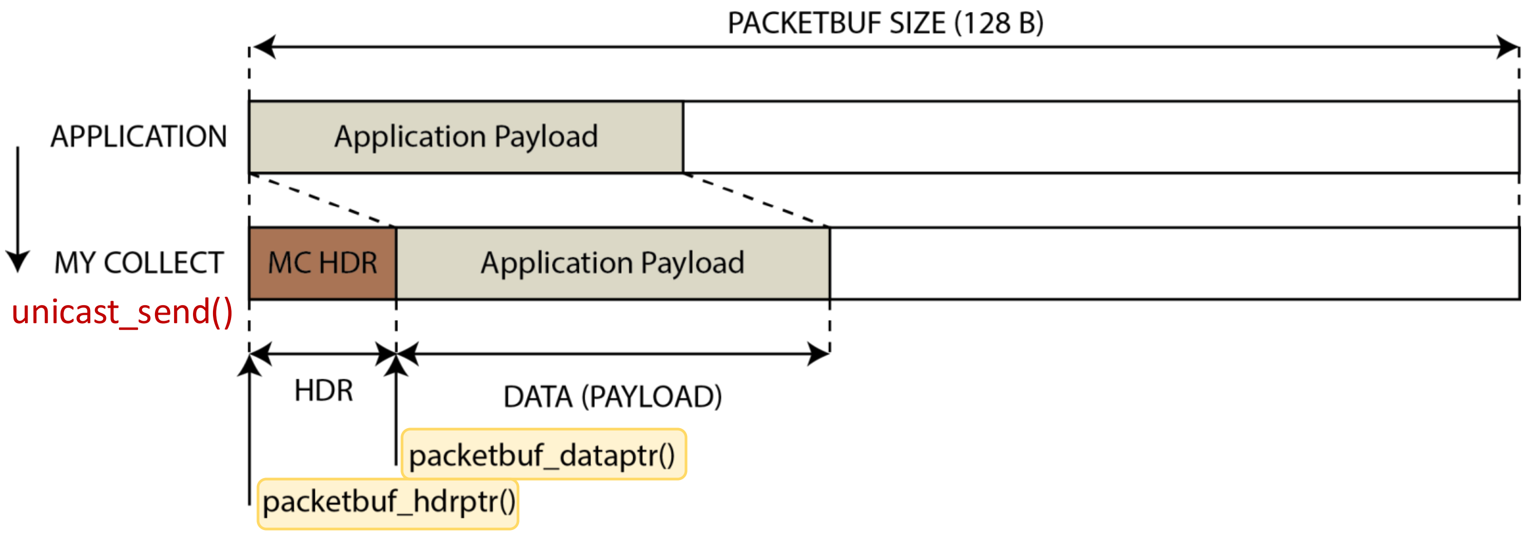
**Application:** (i) clears the packetbuf, (ii) copies the data to the packetbuf, and (iii) sets the data length




**NB:** This is the packetbuf status when my_collect_send(..) is called!

# Packetbuf Management (SENDING)

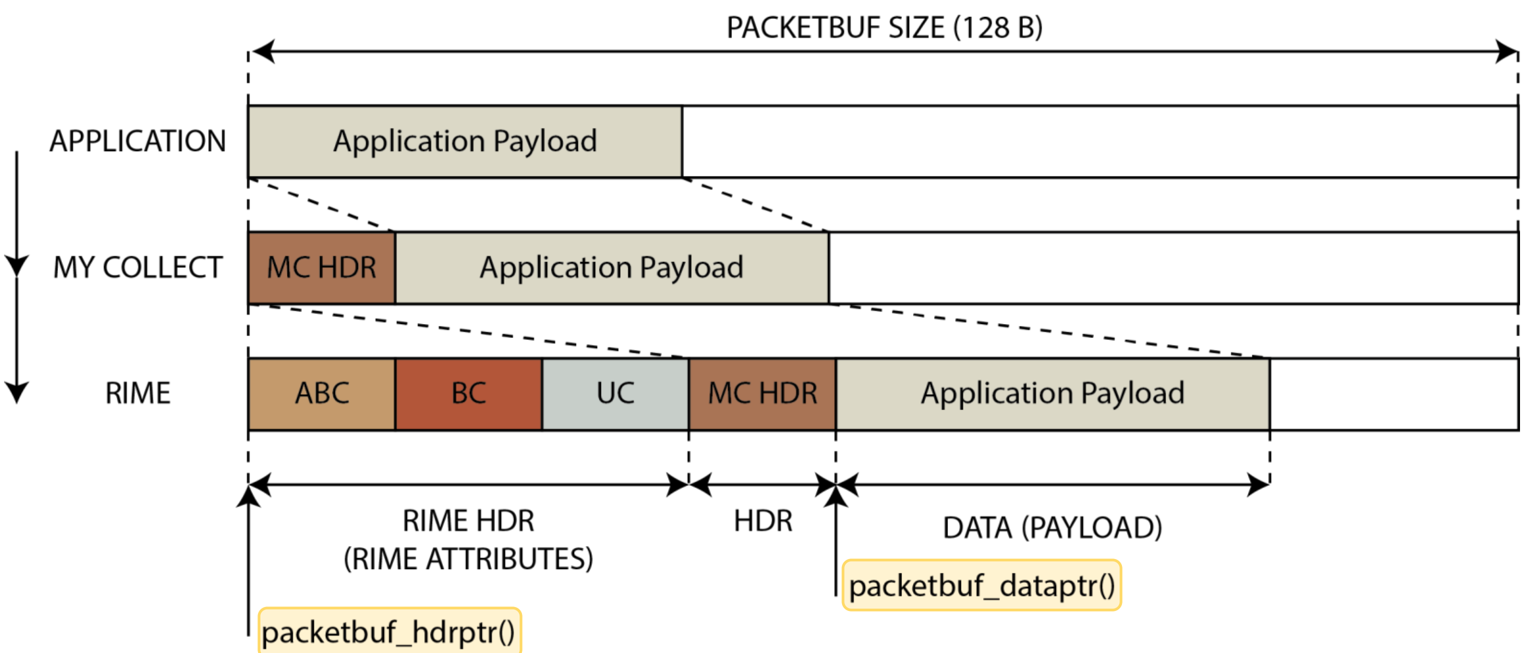
**My Collect:** (i) *allocates* space for the header (shifting the app data to the right → `packetbuf_hdralloc`) and (ii) *copies* its header to the packet buffer




**NB:** This is the packetbuf status when unicast_send() is called!

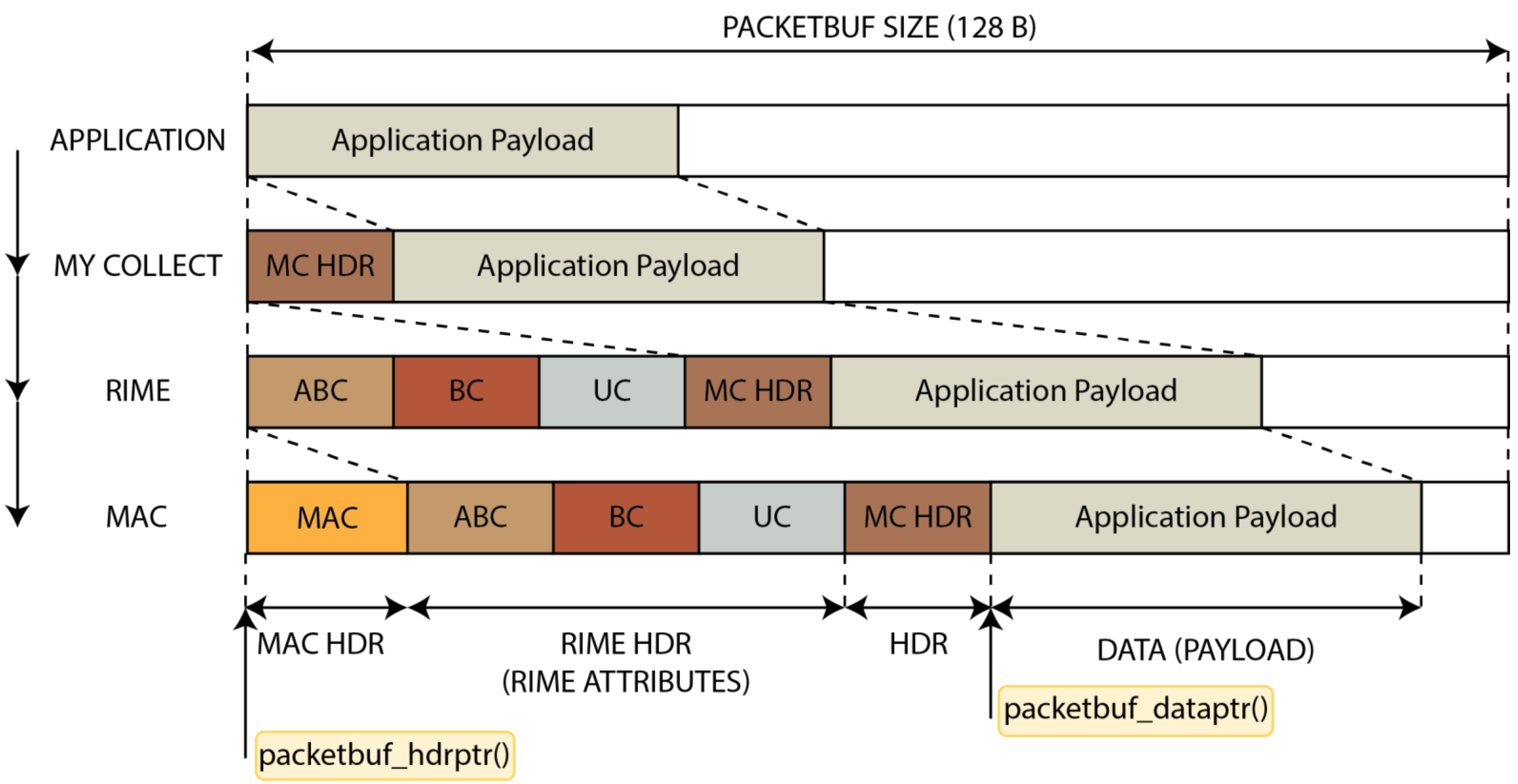
# Packetbuf Management (SENDING)

**Rime:** (i) *allocates* space for its own header (shifting the data to the right) and (ii) *copies* the header to packetbuf



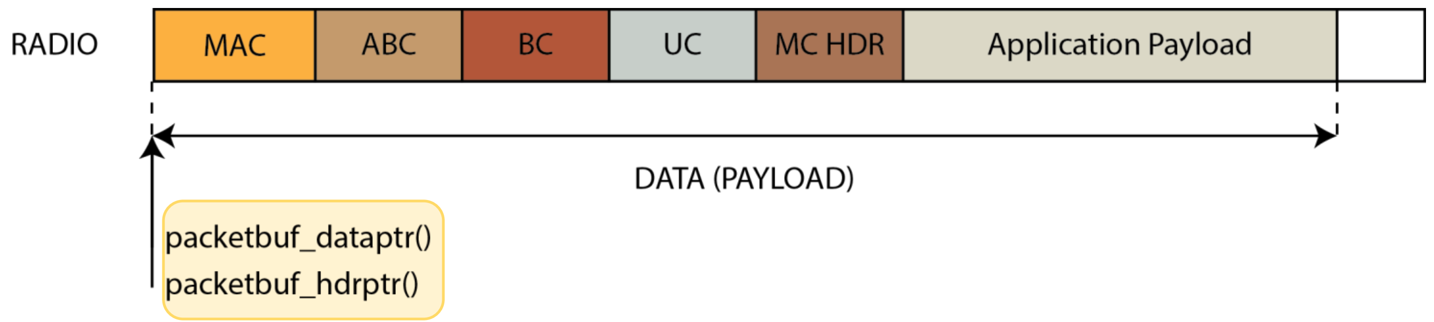
# Packetbuf Management (SENDING)

**MAC:** (i) *allocates* space for its own header (shifting the data to the right) and (ii) *copies* the header to packetbuf



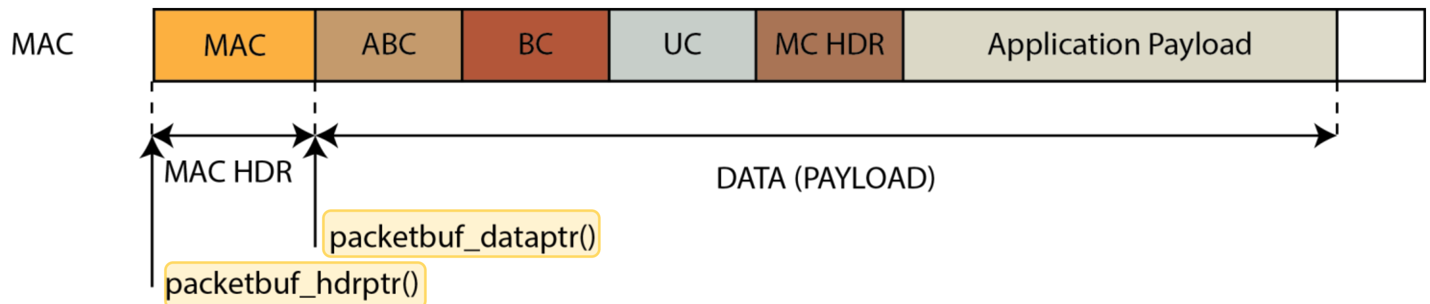
# Packetbuf Management (RECEPTION)

**RADIO:** (i) clears packetbuf, (ii) copies received packet to packetbuf, (iii) sets the data length (MAC + RIME + MC HDR + PLD), and (iv) calls the upper layer (MAC)



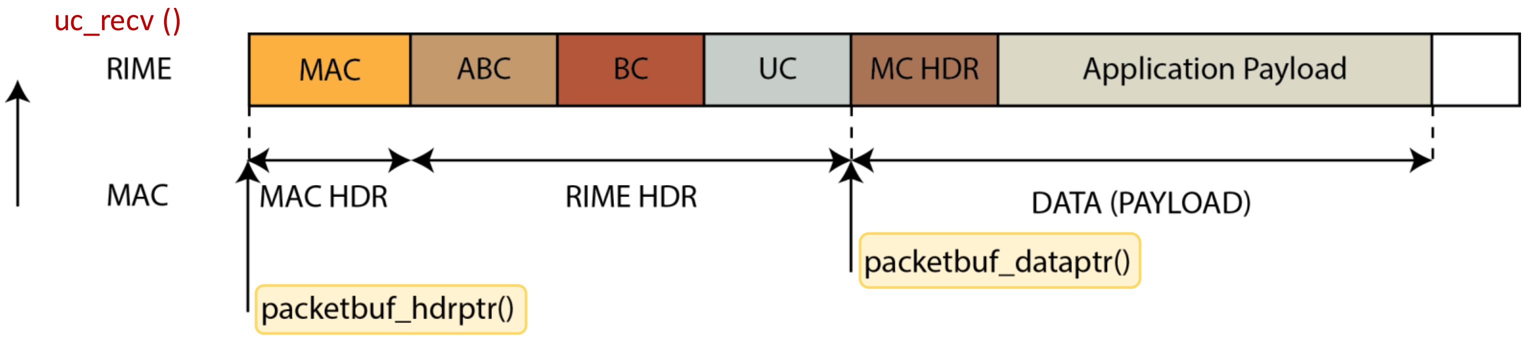
# Packetbuf Management (RECEPTION)

**MAC:** (i) *parses* MAC header, (ii) *reduces the* header (shifting the data pointer to the right of MAC HDR bytes), and (iii) *calls* the upper layer (RIME)



# Packetbuf Management (RECEPTION)

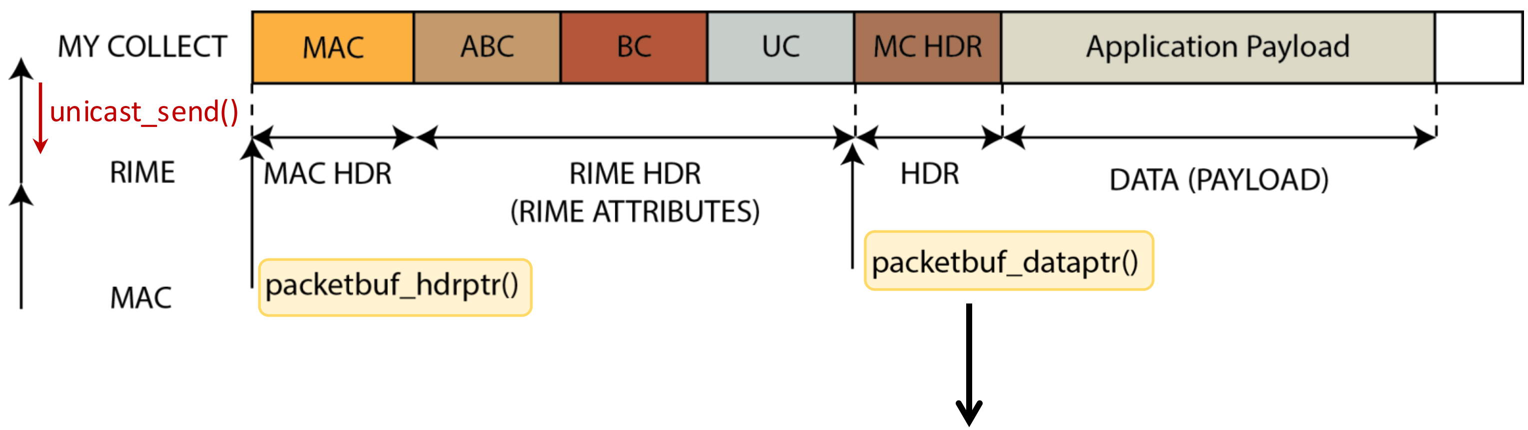
**Rime:** (i) *parses* Rime header attributes, (ii) *reduces* header (shifting the data pointer to the right of RIME HDR bytes), and (iii) *calls* the upper layer (MY_COLLECT)




**N.B.:** This is the packetbuf status when uc_recv () is called!

# Packetbuf Management (RECEPTION)

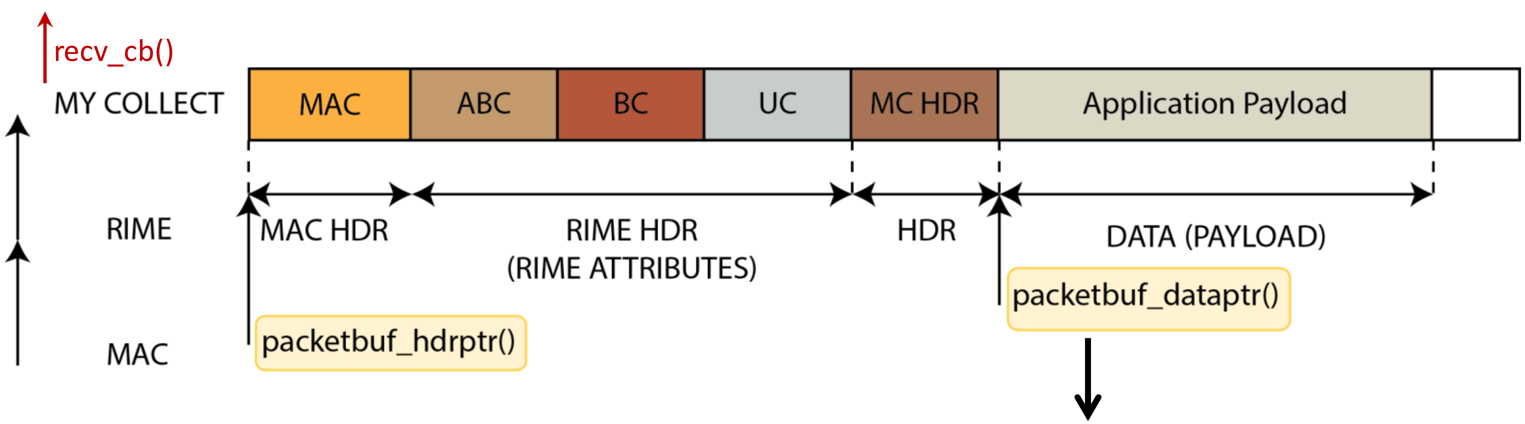
**My Collect—Non-sink:** (i) *parses* and *updates* collect header,
(ii) *copies* the new header to packetbuf,
(iii) *calls* the *lower* layer (RIME) via
`unicast_send()`




**N.B.:** no need to shift it to the right here,
we are *not going* to call the application!

# Packetbuf Management (RECEPTION)

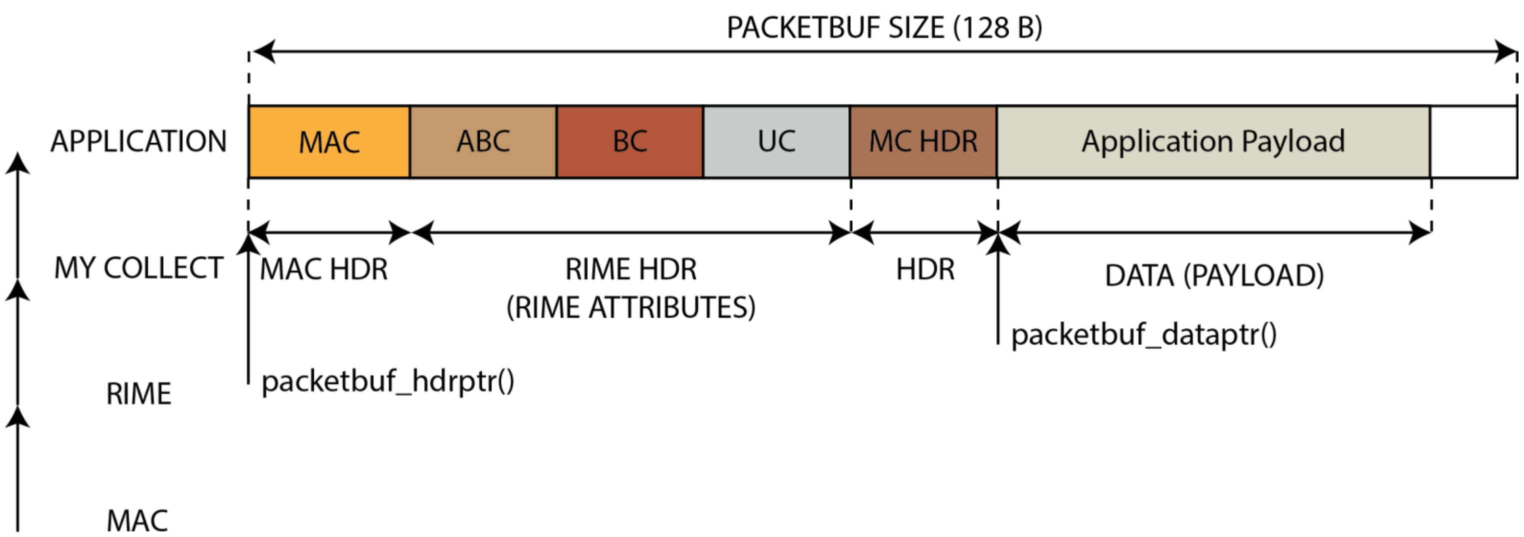
**My Collect—Sink:** (i) *parses* and updates collect header, (ii) *reduces* header (shifting the data pointer to the right of MC HDR bytes), and (iii) *calls* the upper layer (application, via `recv_cb`).




**N.B.:** When the application `recv` callback is called `packetbuf_dataptr()` points to the *beginning* of the app data. The application can thus directly use this pointer to access the app payload!

# Packetbuf Management (RECEPTION)

**Application:** copies payload to a local buffer using `packetbuf_dataptr()` and processes the data



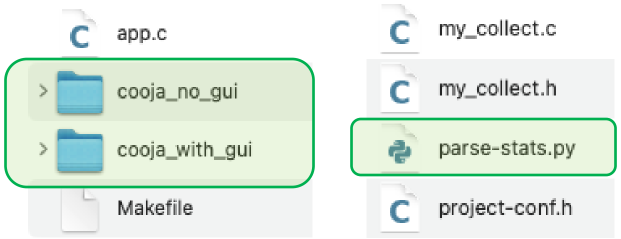
# Code template

## Download and unzip the provided code

- `$ unzip Lab7-exercise.zip`

## Go to the code directory

- `$ cd Lab7-exercise/data-collection-template-Lab7`




## If you have not yet started TODO 1 [Lab 6]

- Directly rely on this new code template!

## Otherwise:

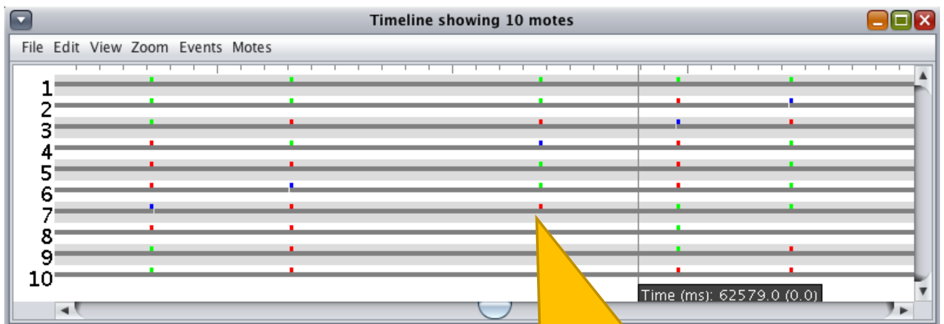
- EITHER replace the `my_collect.c` file in this template with the one you developed in Lab 6, relying on the Lab 7 code template for the rest.
- OR keep working on the Lab 6 template, but (i) set `DATA_FORWARDING = 1` in `app.c`, and (ii) add the `cooja_no_gui` and `cooja_with_gui` folders and `parse-stats.py` to that template.

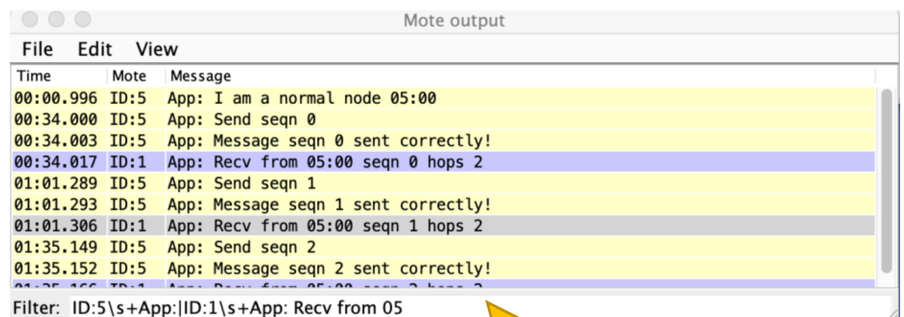
# Test your solution and …

## Step 1 — Check that things work as expected

- Try the different Cooja simulation files in the `cooja_with_gui` folder

  → **Tips:** leverage (i) the Cooja timeline, and (ii) the filter functionality of the mote output tool




Check who TX and RX a given packet

Visualise only the desired output.
**NB: You can use regular expressions here!**

# Test your solution and … Compete!

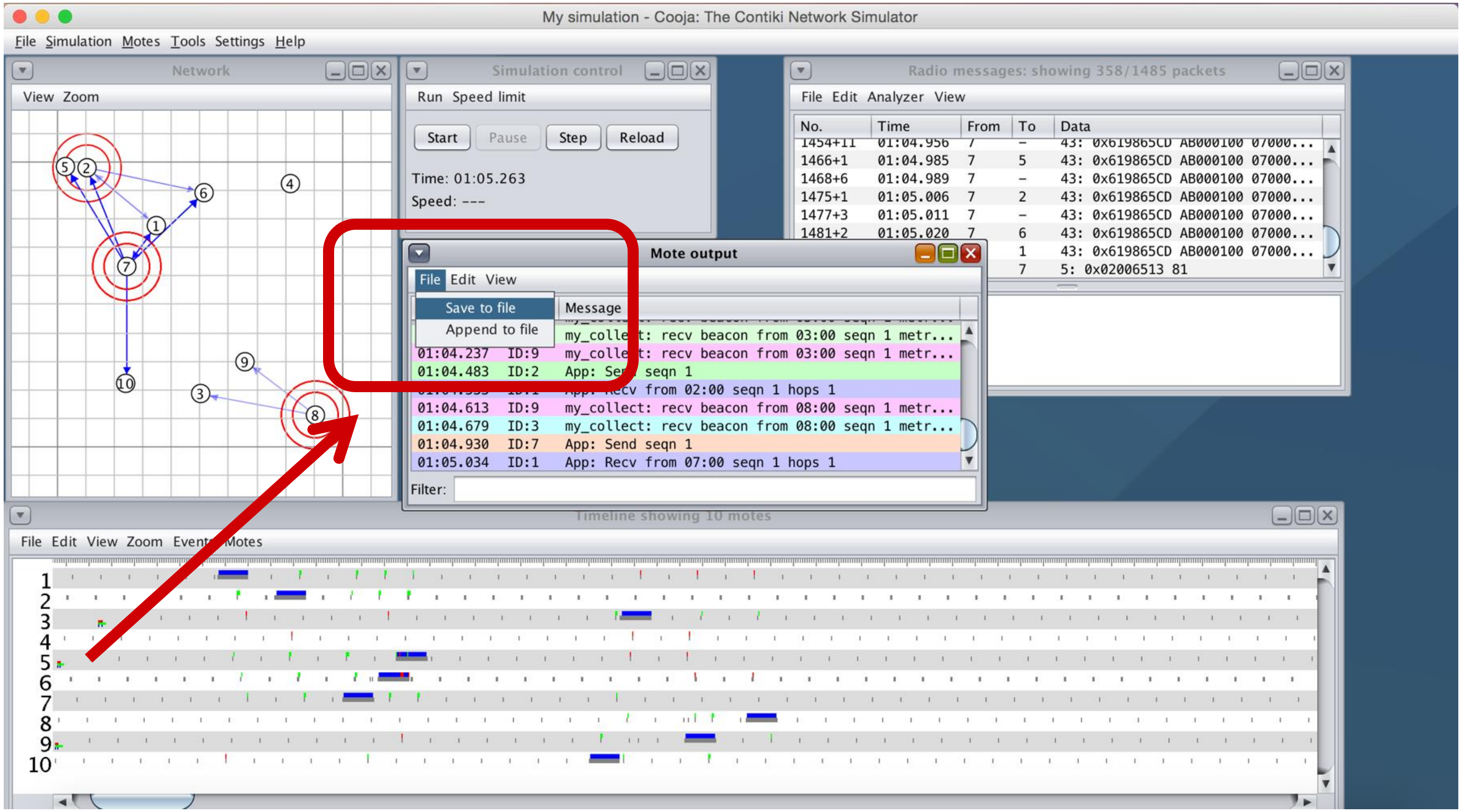
## Step 1 — Check that things work as expected

- Try the different Cooja simulation files in the `cooja_with_gui` folder
  → **Tips:** leverage (i) the Cooja timeline, and (ii) the filter functionality of the mote output tool

## Step 2 — Evaluate the protocol performance and compete with your classmates!

- Run a reasonably long simulation experiment with Cooja:
  `$ cooja testname.csc`
- After 15 simulated minutes, **save a log file** with your results

# How to save a Cooja log file



# Test your solution and … Compete!

## Step 1 — Check that things work as expected

- Try the different Cooja simulation files in the `cooja_with_gui` folder
  → **Tips:** leverage (i) the Cooja timeline, and (ii) the filter functionality of the mote output tool

## Step 2 — Compare your protocol performance with your classmates!

- Run a longer simulation experiment with Cooja:
  `$ cooja testname.csc`
- After 15 simulated minutes, **save a log file** with your results
- Analyze the reliability of your protocol with:
  `$ python3 parse-stats.py my_log_file.log`
- Try to improve your solution!

# Let's automate: Cooja simulations without GUI!

Once your protocol is working reliably, you can run Cooja **without GUI**, increasing the simulation speed and making it easier to automate experiments.

**Steps:**

- Emulate the Cooja simulation files provided in `cooja_no_gui`:
  `$` **`cooja_nogui`** *`testname`*`_nogui.csc`
- By default:
  - 30 (simulated) minutes experiment
  - Motes output saved in *`testname`*`_nogui.log` file
  - Radio-on time saved in *`testname`*`_nogui_dc.log` file
- Analyze the reliability of your protocol with:
  `$ python3 parse-stats.py` *`testname`*`_nogui.log`

# Preparing Cooja simulations without GUI

To prepare other Cooja simulations without GUI, you can just copy and paste in your `*.csc` files the following code snippet (from every `testname_nogui.csc` files we provide):

```
<plugin>
      org.contikios.cooja.plugins.ScriptRunner
      <plugin_config>
            <script>SIM_SETTLING_TIME = 1000
                  TIMEOUT(1800000);
                  .
                  .
                  .
            </script>
            <active>true</active>
      </plugin_config>
</plugin>
```

You can change the experiment duration by modifying this value!

**NOTE: the timeout is in ms**

# Expected performance of your protocol

Once you have implemented all the basic functionalities we have discussed, your system should achieve *similar performance*

```
MacBook-Air-di-Matteo:exp matteo$ python ../parse-stats.py test_nogui.log
Namespace(logfile='test_nogui.log', testbed=False)
Logfile: test_nogui.log
Cooja simulation

########## Node Statistics ##########

Node 2: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 3: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 4: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 5: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 6: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 7: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 8: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 9: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 10: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%

########## Overall Statistics ##########

Total Number of Packets Sent: 531
Total Number of Packets Received: 531
Overall PDR = 100.00%
Overall PLR = 0.00%
```

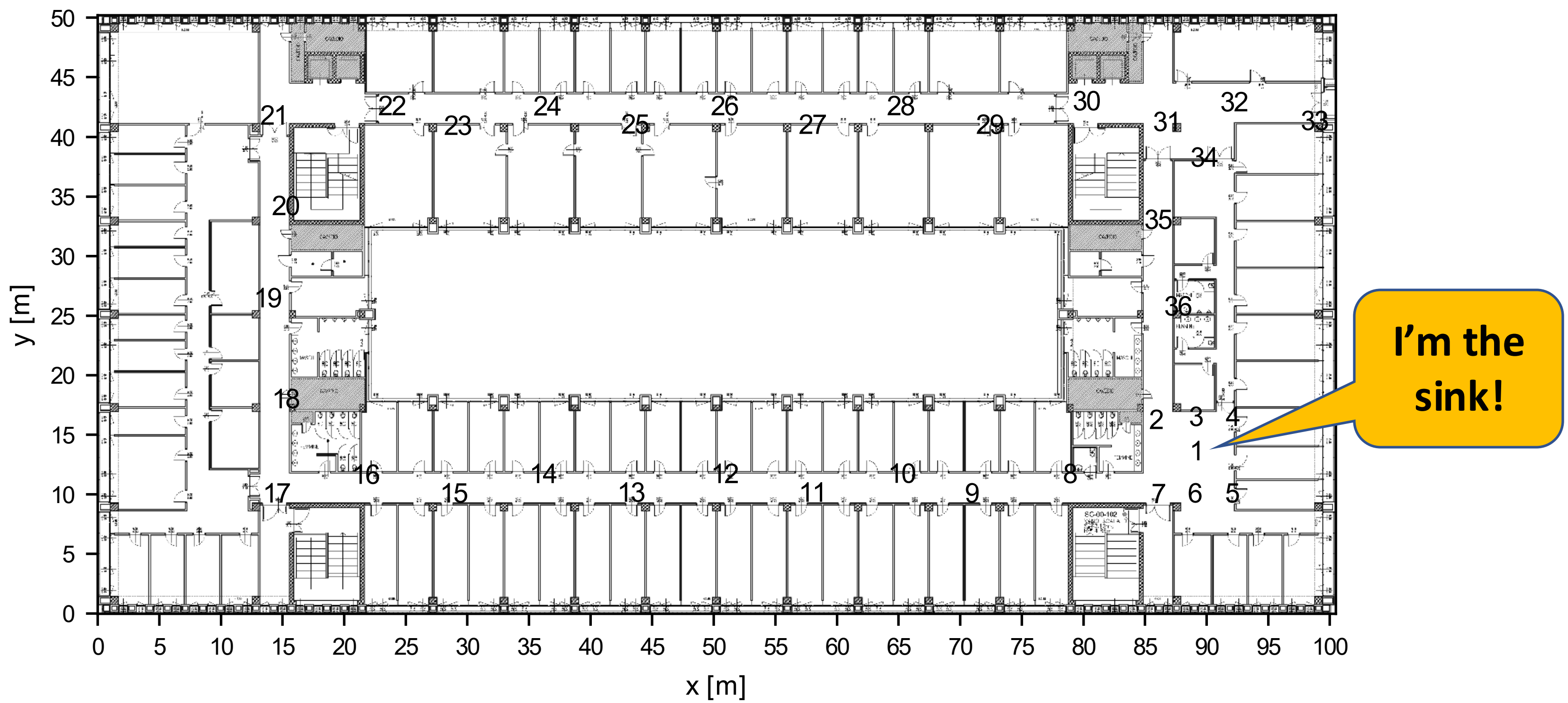
```
MacBook-Air-di-Matteo:exp matteo$ python ../parse-stats.py test_more_random_nogui.log
Namespace(logfile='test_more_random_nogui.log', testbed=False)
Logfile: test_more_random_nogui.log
Cooja simulation

########## Node Statistics ##########

Node 2: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 3: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 4: TX Packets = 59, RX Packets = 58, PDR = 98.31%, PLR = 1.69%
Node 5: TX Packets = 59, RX Packets = 58, PDR = 98.31%, PLR = 1.69%
Node 6: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 7: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 8: TX Packets = 59, RX Packets = 58, PDR = 98.31%, PLR = 1.69%
Node 9: TX Packets = 59, RX Packets = 59, PDR = 100.00%, PLR = 0.00%
Node 10: TX Packets = 59, RX Packets = 58, PDR = 98.31%, PLR = 1.69%

########## Overall Statistics ##########

Total Number of Packets Sent: 531
Total Number of Packets Received: 527
Overall PDR = 99.25%
Overall PLR = 0.75%
```

There is still room for improvements... Can you push it further?

Does it work?
Try it in CLOVES
with Firefly nodes!

# Suggested topology to start with

**DISI POVOII floor** (part of DEPT) ⟶ **36 nodes, use node 1 as sink**




Other topologies are available! Check them at
`https://research.iottestbed.disi.unitn.it/`**`maps/`**

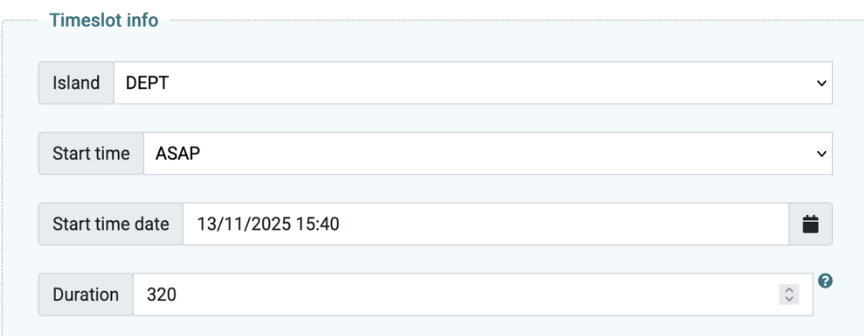
<u>**N.B.:**</u> **Before** to use <u>different</u> topologies, remember to update:

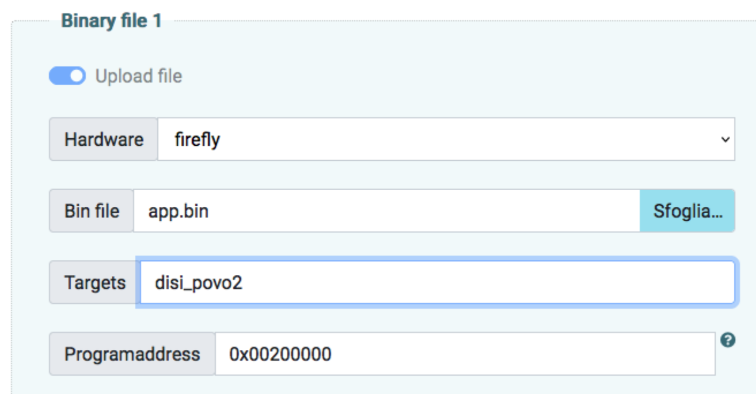
1. `linkaddr_t sink` in `app.c`
2. `sink_id, addr_id_map` in `parse-stats.py`

# Running a test in CLOVES

## Steps:

1. **Compile your code for the Zolertia Firefly platform:**
   `$ make TARGET=zoul`
2. **Log into the CLOVES interface:**
   Start at `research.iottestbed.disi.unitn.it`, and log in with your credentials. Then navigate to "Create job".
3. **Prepare a job:**




4. **Wait for the job to complete & download the logs:**
   Check (and download) completed jobs at:
   `https://research.iottestbed.disi.unitn.it/jobs_completed/`
5. **Parse and analyze the logs:**
   `$ python3 parse-stats.py your_path/job.log --testbed`

# Performance at night

```
Lab6_7-solution/data-collection-template/cloves/job_44552 » python ../../parse-stats.py job.log --testbed
Namespace(logfile='job.log', testbed=True)
Logfile: job.log
Testbed experiment

########## Node Statistics ##########

Node 2: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 3: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 4: TX Packets = 16, RX Packets = 16, PDR = 100.00%, PLR = 0.00%
Node 5: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 6: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 7: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 8: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 9: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 10: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 11: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 12: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 13: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 14: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 15: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 16: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 17: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 18: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 19: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 20: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 21: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 22: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 23: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 24: TX Packets = 16, RX Packets = 16, PDR = 100.00%, PLR = 0.00%
Node 25: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 26: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 27: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 28: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 29: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 30: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 31: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 32: TX Packets = 16, RX Packets = 16, PDR = 100.00%, PLR = 0.00%
Node 33: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 34: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 35: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%
Node 36: TX Packets = 15, RX Packets = 15, PDR = 100.00%, PLR = 0.00%

########## Overall Statistics ##########

Total Number of Packets Sent: 528
Total Number of Packets Received: 528
Overall PDR = 100.00%
Overall PLR = 0.00%
```

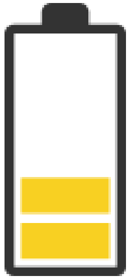
**N.B.:**

- Night-time experiment on *channel 25!* Results might **highly vary** in function of the amount of interference present in the channel!
- Be careful, during the day the amount of interference is typically higher!
- Try to run an experiment during the day on channel 11 (typically highly exposed to Wi-Fi traffic); do you notice any difference?

# What about energy consumption?

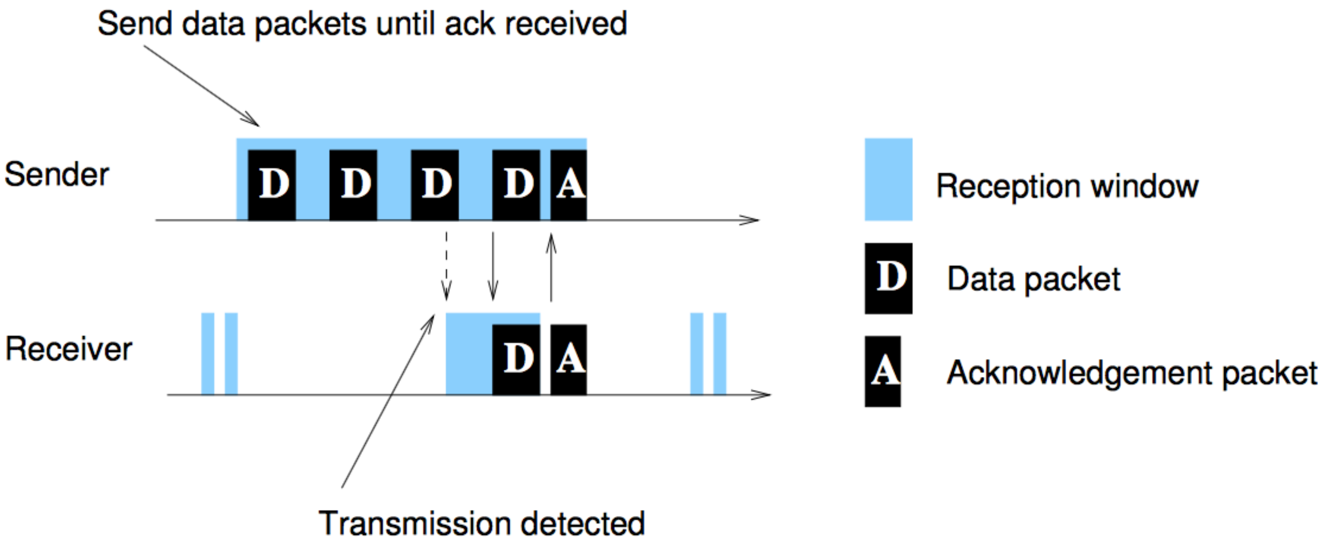
# NullRDC Vs. ContikiMAC

So far, we used **NullRDC** → Radio is ***always ON***, **high power consumption!**



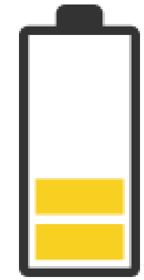
More energy efficient radio duty-cycle (RDC) layers are available! Check them at `contiki-uwb/contiki/core/net/mac`

- Try **ContikiMAC*** and check how it affects performance




* A. Dunkels, "The ContikiMAC Radio Duty Cycling Protocol", *SICS Technical Report*, 2011

# NullRDC Vs. ContikiMAC



So far, we used **NullRDC** → Radio is ***always ON*, high power consumption!**

More energy efficient radio duty-cycle (RDC) layers are available!
Check them in `core/net/mac`

- Try **ContikiMAC*** and check how it affects performance

**Steps:**

1. Change the RDC protocol in your **`project-conf.h`** file:
```
/* #define NETSTACK_RDC nullrdc_driver */
#define NETSTACK_RDC contikimac_driver
```
2. Check the PDR of your system, do you notice any difference?
3. Compare ContikiMAC and NullRDC radio-on time:
   - Cooja Timeline: Can you recognize the peculiar ContikiMAC radio on-off pattern?
   - Radio duty-cycle: Cooja -> Tools -> Mote radio duty cycle