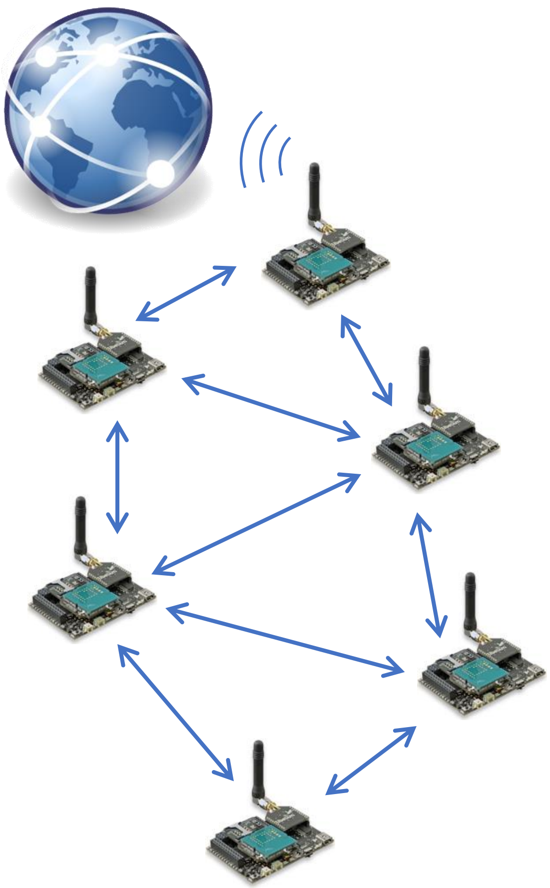
Low-power Wireless Networking for the Internet of Things

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

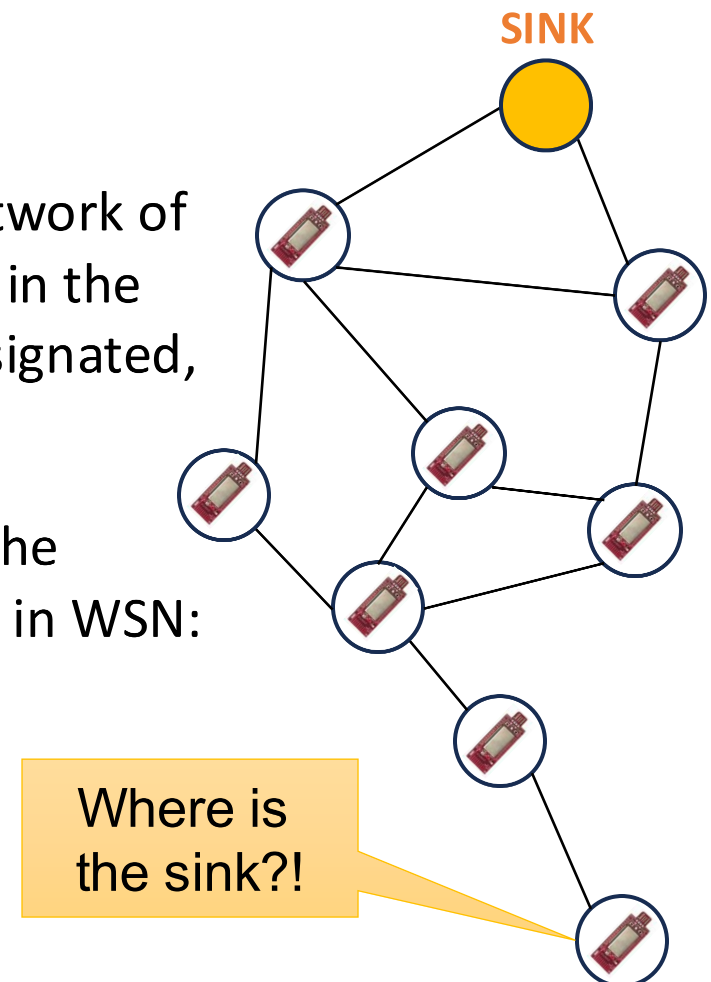
Matteo Trobinger (matteo.trobinger@unitn.it)

Credits for some slides to:

Pablo Corbalán, Timofei Istomin



Multi-hop data collection



Basic idea: Given a multi-hop wireless network of any topology, we want to enable *any* node in the network to communicate with a single, designated, common destination, i.e., **the sink**.

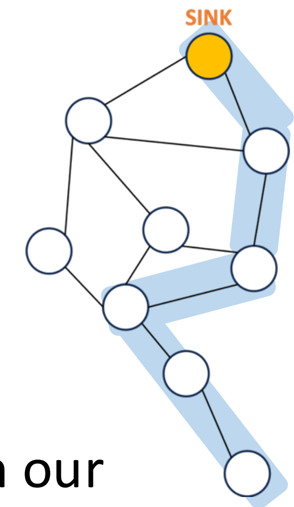
Why? Multi-hop data collection is one of the *most important* communication paradigms in WSN:

- Environmental monitoring
- Smart agriculture
- Industrial monitoring and control

Key challenges:

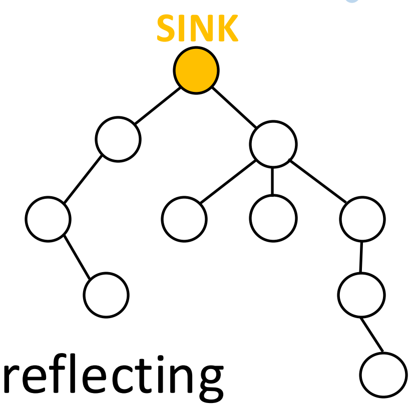
1. <u>Where is the sink?</u> Nodes need to acquire information about the network topology **[Lab 6]**
2. <u>How to communicate with the sink?</u> Nodes should exploit such information to forward data packets across the network, from sources to the sink node **[Lab 7]**

Many-to-one routing



Routing — building *routes* (multi-hop delivery chains) from sources to destination

Many-to-one routing — from *many* sources (*all sensors* in our case) to just *one* destination (the sink)



How to build these routes?

- We will construct a *minimum-cost spanning tree rooted at the sink* based on a path cost function
- The path cost (based on a *routing metric*) is a value reflecting the effort required to deliver packets along the path

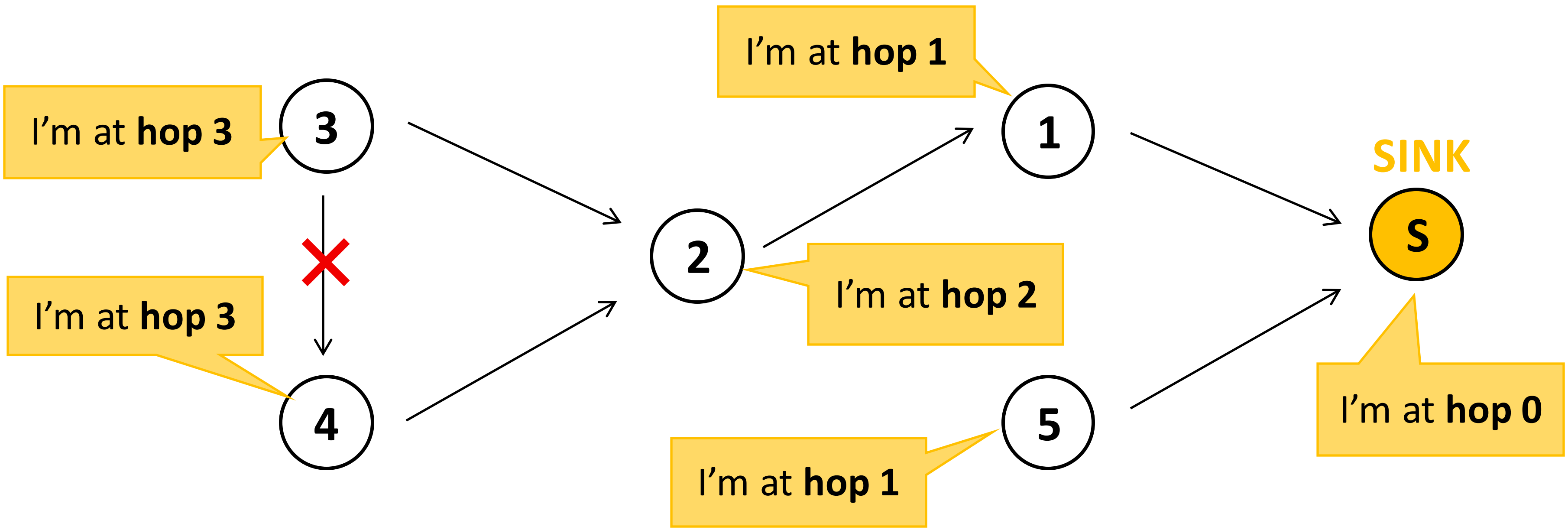
Many different routing metrics exist:

- RSSI, LQI, hop count, ETX, …
- Different trade offs in terms of implementation complexity, reliability, energy cost, …

Our choice for today

Our routing metric: The hop count

Intuition: Number of *intermediate devices* a data packet should pass through to reach the sink



The simplest routing metric

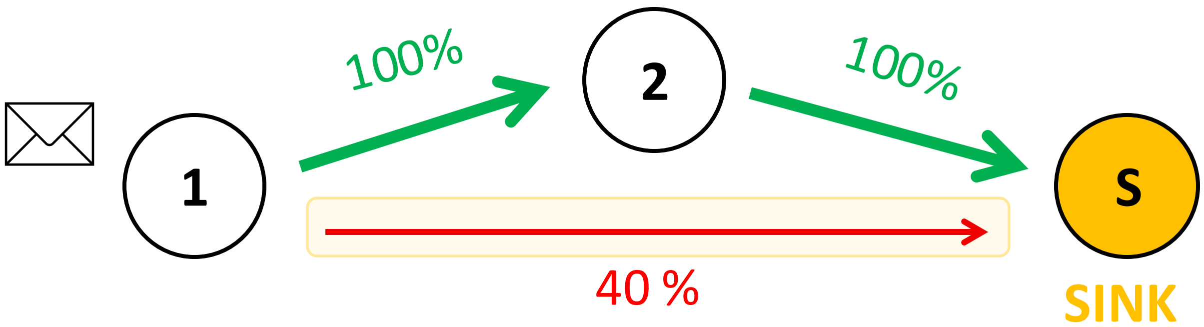
- The path cost = its length (# of hops) ⟶ To be ***minimized***
- The longer the path, the higher the cost (the *worst* the metric)

NB: It is simple but (can be) bad for wireless

Why is the hop count metric potentially bad?

Nodes will try to *minimize* the number of hops in the path

⟶ choosing ***longer*** (and therefore ***weaker***) links



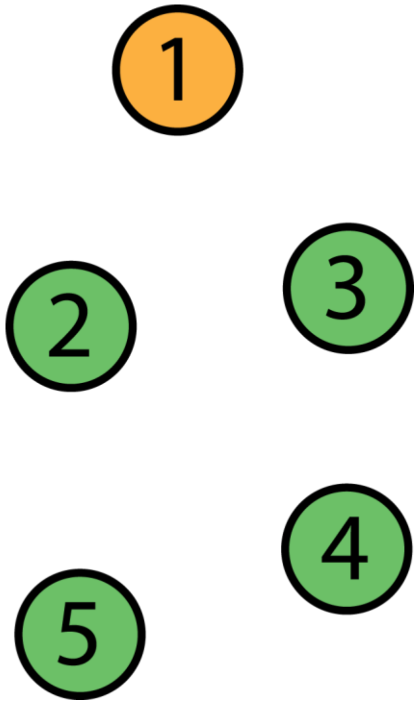
Better would be to measure the reliability (PDR) of all links and consider it when computing the path costs

⟶ *not that easy* unfortunately …

Our approach: use the hop count, but filter out very bad links based on a RSSI threshold

Building the tree (routing)

The **sink** (i.e., the root of the tree) initiates the process by ***sending* a *beacon message* with *h=0 in broadcast***



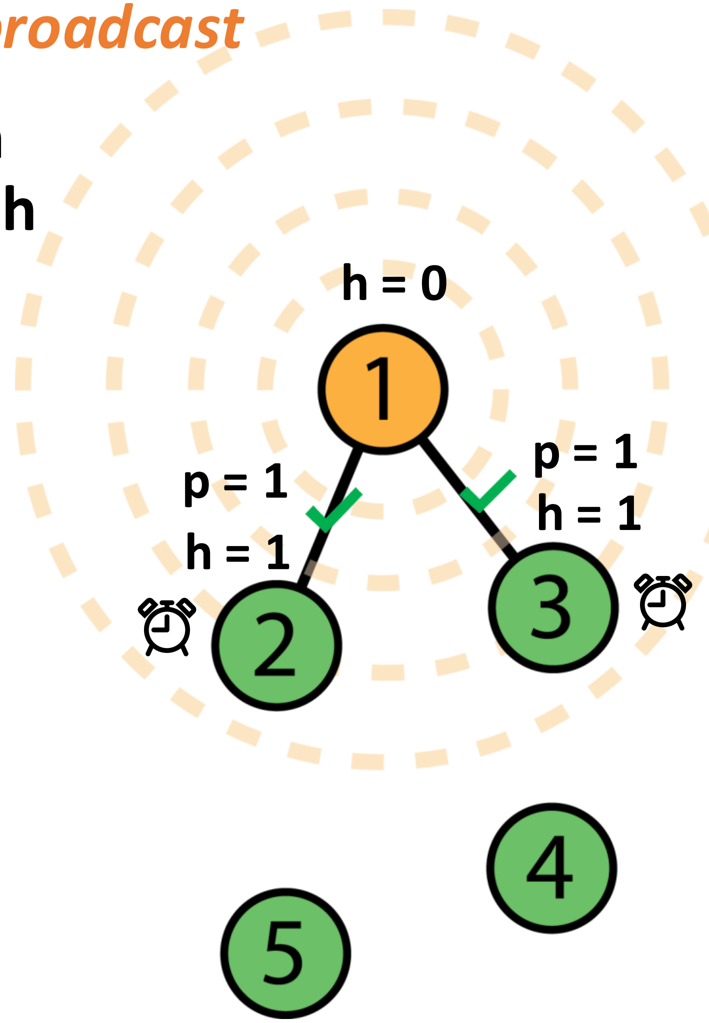
Building the tree (routing)

The **sink** (i.e., the root of the tree) initiates the process by ***sending* a *beacon message* with *h=0 in broadcast***

When a **non-sink node** receives a beacon with RSSI > threshold, it should **compare h of the received message with that of its current parent (if any)**.

If the one in the received message is better (i.e., lower), it

1. Considers the source of the beacon as the ***parent***
2. Sets its own hop-count to ***h+1***
3. (After a *small, random* delay) *broadcasts* a new beacon message with its own hop-count (i.e., ***h+1)*** as the path metric



LAB 6

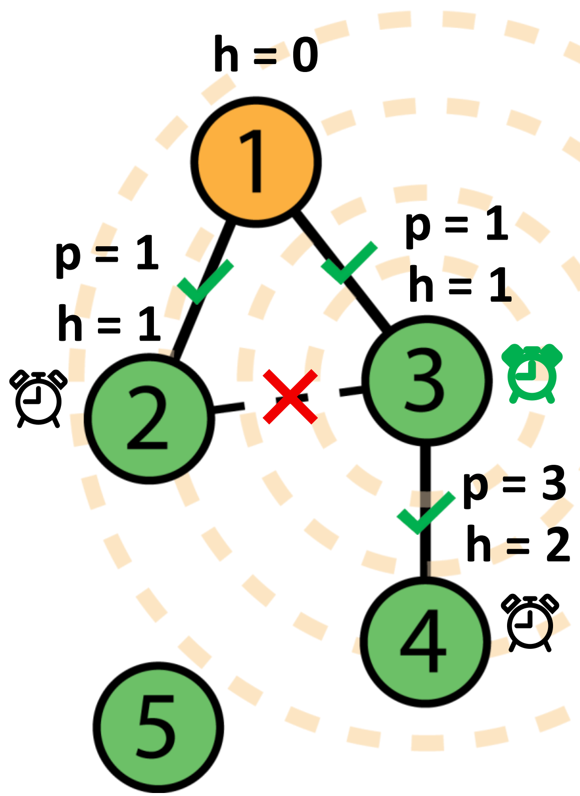
Building the tree (routing)

The **sink** (i.e., the root of the tree) initiates the process by ***sending* a *beacon message* with *h=0 in broadcast***

When a **non-sink node** receives a beacon with RSSI > threshold, it should **compare h of the received message with that of its current parent (if any)**.

If the one in the received message is better (i.e., lower), it

1. Considers the source of the beacon as the ***parent***
2. Sets its own hop-count to ***h+1***
3. (After a *small, random* delay) *broadcasts* a new beacon message with its own hop-count (i.e., ***h+1)*** as the path metric



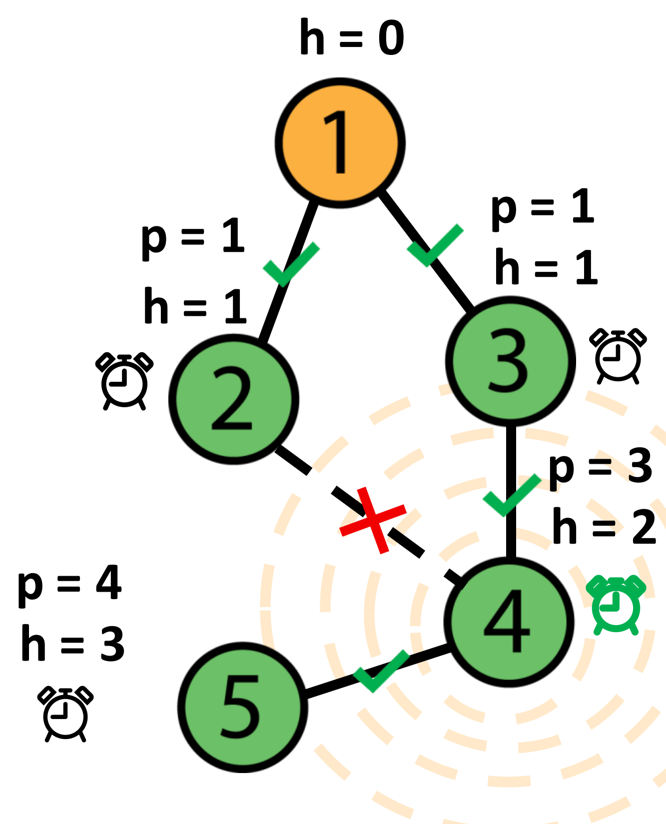
Building the tree (routing)

The **sink** (i.e., the root of the tree) initiates the process by ***sending* a *beacon message* with *h=0 in broadcast***

When a **non-sink node** receives a beacon with RSSI > threshold, it should **compare h of the received message with that of its current parent (if any)**.

If the one in the received message is better (i.e., lower), it

1. Considers the source of the beacon as the ***parent***
2. Sets its own hop-count to ***h+1***
3. (After a *small, random* delay) *broadcasts* a new beacon message with its own hop-count (i.e., ***h+1)*** as the path metric



LAB 6

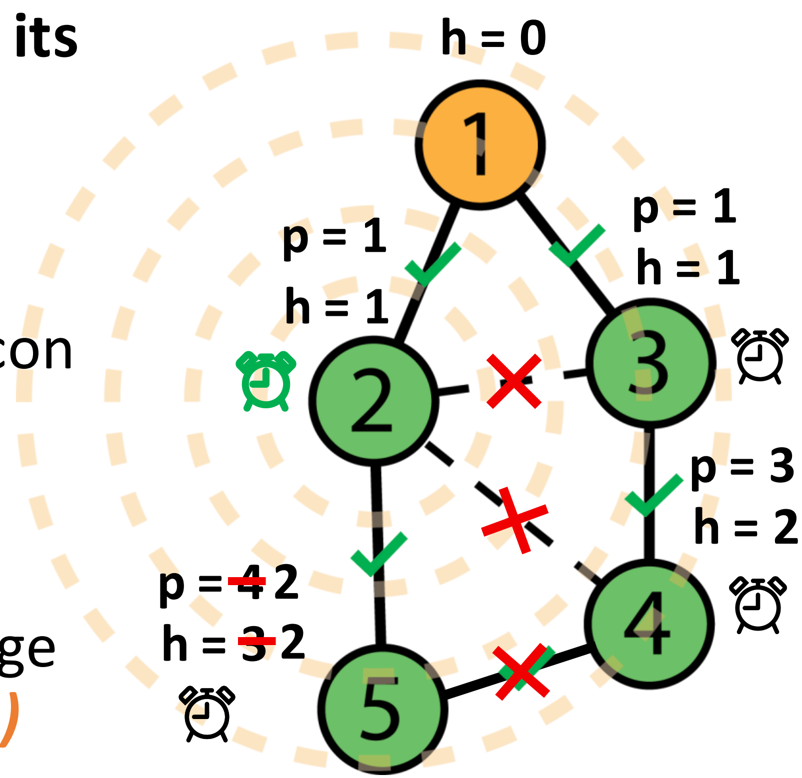
Building the tree (routing)

The **sink** (i.e., the root of the tree) initiates the process by ***sending* a *beacon message* with *h=0 in broadcast***

When a **non-sink node** receives a beacon with RSSI > threshold, it should **compare h of the received message with that of its current parent (if any)**.

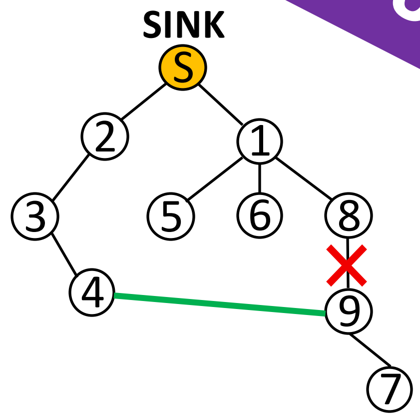
If the one in the received message is better (i.e., lower), it

1. Considers the source of the beacon as the ***parent***
2. Sets its own hop-count to ***h+1***
3. (After a *small*, *random* delay) *broadcasts* a new beacon message with its own hop-count (i.e., ***h+1)*** as the path metric



LAB 6

Periodic tree updates



The protocol should adapt to network changes

- The environment **is dynamic:**
  - Links quality can rapidly change modifying the previously learned topology,
  - Nodes may fail or run out of battery,
  - New nodes may be deployed
- The build-once-forever approach **will not work!**

NB: From time to time the routes should be refreshed!

***One* way of doing it:**

Periodically rebuild the tree from scratch

- The sink *periodically* sends a **new** beacon
- When a non-sink node receives a **new** beacon with RSSI > threshold, it accepts the new routing information (hop count, parent) **without checking it against the old one**

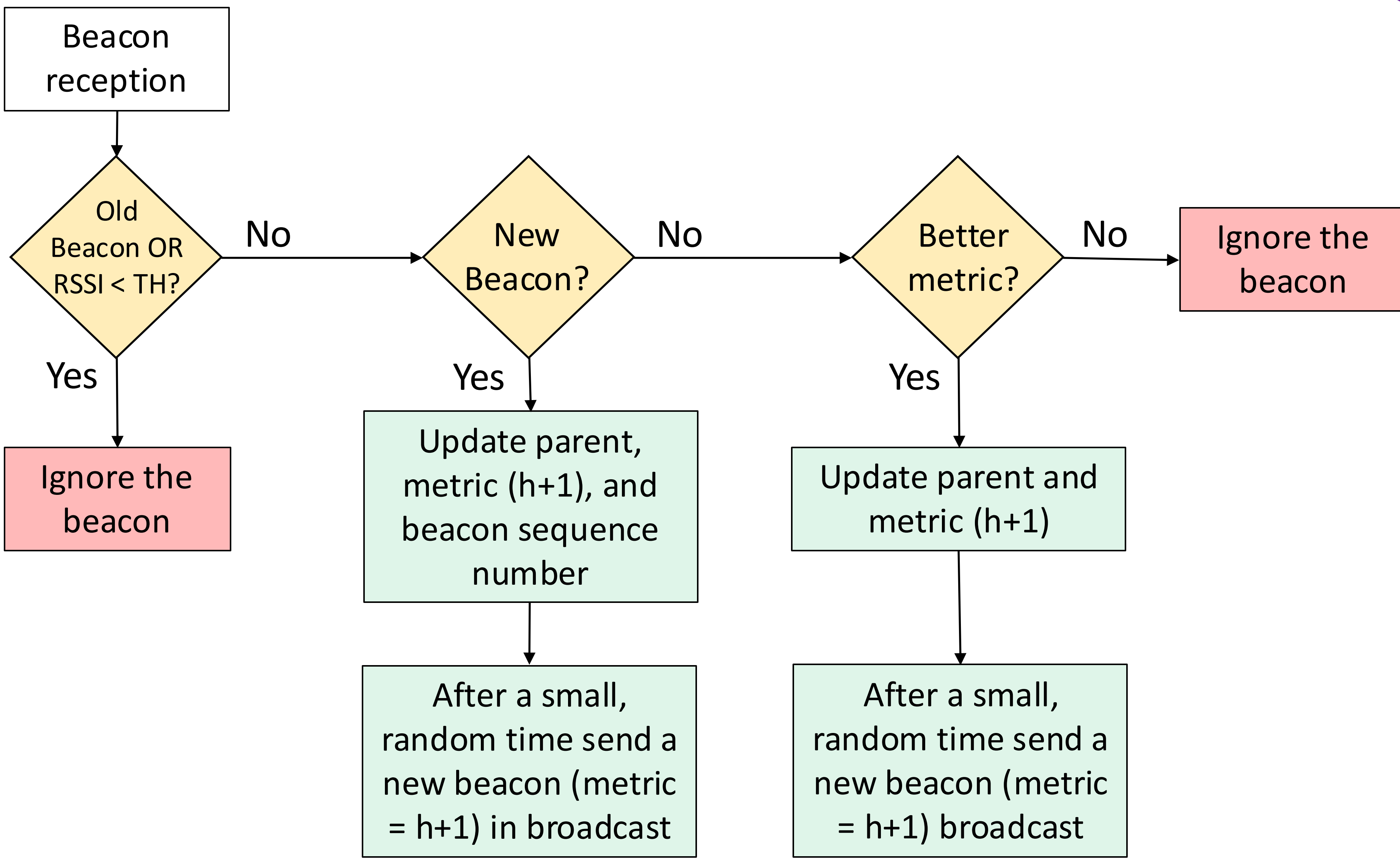
ONLY the sink can update this field (upon sending a new beacon)!

metric	seqn

Q How to tell a new beacon from an old one?

Use sequence numbers! (Embed this information in *every* beacon)

Non-sink node logic to build the tree



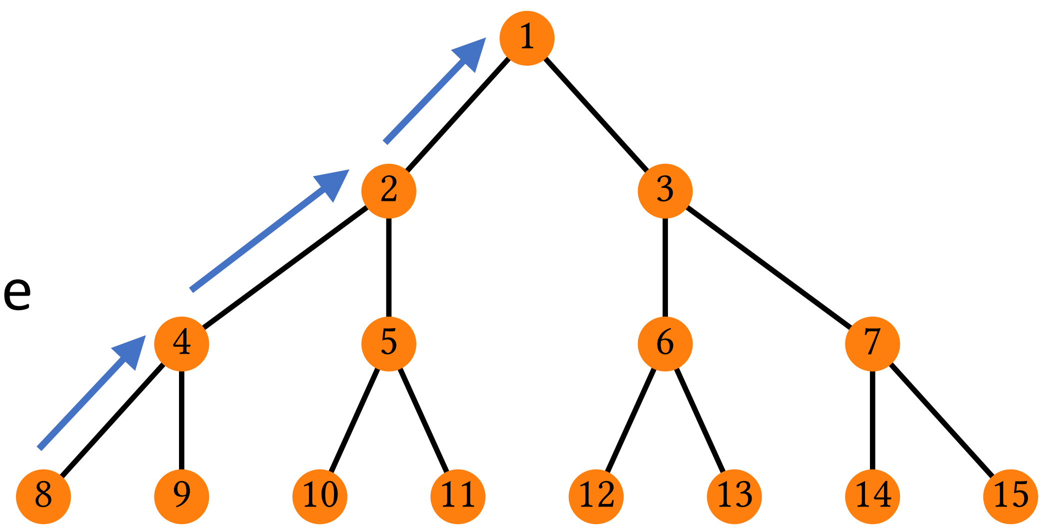
Data forwarding

After the routes have been constructed, nodes can send data *towards* the sink **by sending them to *their parents* (*in unicast*)**

When a node receives a data packet, it **should forward (relay) it to its parent** (again in *unicast*)

Q

What will happen if node 4 will suddenly stop working?



my_collect: our new RIME primitive

Today and in Lab 7 we are going to implement **our own RIME primitive**: **`my_collect()`** that will exploit

- The *identified sender broadcast primi*tive to periodically build the routing tree [Lab 6]
- The *identified receiver unicast primitive* to forward data packets towards the sink [Lab 7]

In practice, we are going to:

- Define our own connection object: `struct my_collect_conn`
- Implement our own initialization function: `my_collect_open` (similar to e.g., broadcast_open)
- Implement the protocol logic (as always mostly within the callback functions)
- [Lab 7] Implement the send function (`my_collect_send`), to enable non-sink nodes to start sending data towards the sink

Code template

Download and unzip the provided code

- `$ unzip Lab6-exercise.zip`

Go to the code directory

- `$ cd Lab6-exercise/data-collection-template`

To compile:

- `$ make` to compile for the Tmote Sky platform (Cooja)
- `$ make TARGET=zoul` to compile for the Zolertia Firefly platform (testbed)

When everything seems to work, test it in Cooja and in the testbed!

Program structure

1. **app.c** – the application that will use our collection layer
   - Already implemented, you do not need to modify it
   - **NB**: please ***do not*** change the print format!
2. **my_collect.h** – the header file specifying the application interface
   - ready to use, modify if needed
3. **my_collect.c** – the collection layer
   - To be completed: implement routing (TODAY) and forwarding logic (LAB 7) here

Today you should implement
ONLY TODO's 1-4!

Notes on the program design

Our data collection layer relies on two Rime primitives:

1. Broadcast to send beacons (Lab 6)
2. Unicast to forward data packets (Lab 7)

Both connections are stored in the `my_collect_conn` structure, together with all the protocol state data:

```
struct my_collect_conn {
      struct broadcast_conn bc;
      struct unicast_conn uc;
      struct my_collect_callbacks* callbacks;
      struct ctimer beacon_timer;
      linkaddr_t parent;
      uint16_t metric;
      uint16_t beacon_seqn;
      bool is_sink;
};
```

Current parent

Current metric

Last beacon seqn

Implementing routing

Edit my_collect.c

- **TODO 1 (my_collect_open()):**
  - 1.1: Initialize the connection structure
  - 1.2: Use the ctimer in `my_collect_conn` to schedule periodic beacons *on the sink node*
- **TODO 2 (beacon_timer_cb()):**
  - Send the beacon (use the `send_beacon` function)
- **TODO 3 (bc_recv()):**
  - Filter out <u>*very bad links*</u>, based on a RSSI threshold (e.g., consider everything below –95 dBm unacceptable, *ignoring* such beacons)
  - Analyze the received beacon and update the routing information <u>if needed</u>
- **TODO 4 (bc_recv()):**
  - If the `metric` or `seqn` was updated on the previous step, retransmit the updated beacon after a small random delay

NB: Other **TODOs** are **for the next time**!

A few tips …

- Use a **(strictly) monotonous routing metric,** otherwise routing loops will appear
  - The hop count is such,
  - In case you want to explore other routing metrics *be careful* about that!

- TODO 3: To read the RSSI of the last reception:

```
int16_t rssi;
rssi = packetbuf_attr(PACKETBUF_ATTR_RSSI);
```

When the rest is done: try to explicitly handle the overflow of the sequence number!

Test your solution in Cooja!

3 Cooja simulation files already provided, start testing your protocol with them!

- **test_linear.csc** → linear topology, every node has 1 parent and 1 child: simple to check whether your protocol works as expected
- **test_circle.csc** → circular topology, beacons propagate along two path! Check if the farthest nodes from the sink (e.g., nodes 5 and 7) handles the routing metric properly (updating it when needed)
- **test.csc** → A more complex topology

Do you want to test more? *Create <u>your own</u> topology!*

1. Start Cooja (type `cooja` in the terminal)
2. File → New simulation → UDGM: constant loss → Create
3. Motes → Add motes → Create new mote type → Sky mote
   - Contiki process/Firmware: select app.sky

When everything works, switch to a lossy model (e.g., UDGM distance loss or MRM); do you notice any difference?