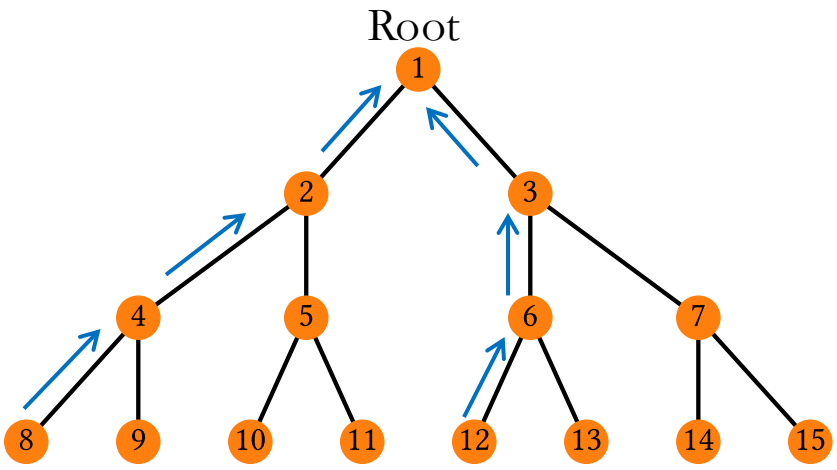
Course Project:
Multi-hop Communication with Data Collection & Source Routing

Low-power Wireless Networking for the Internet of Things
University of Trento, Italy
2025-2026

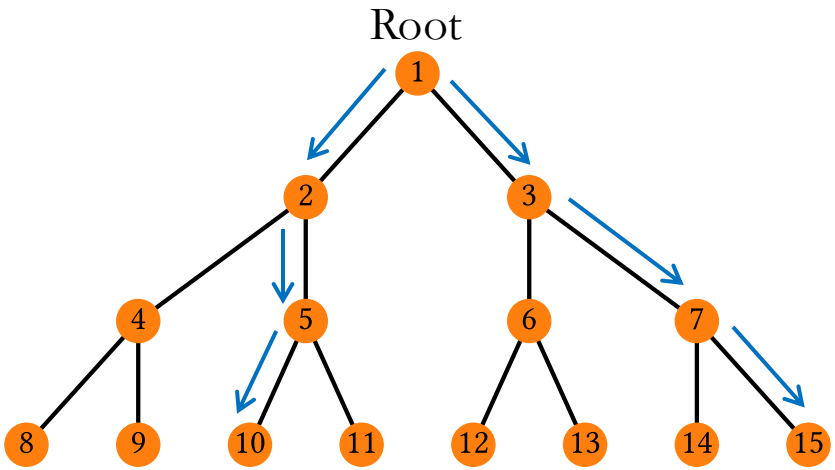
Project Goal

Implement a low-power wireless networking protocol that supports two traffic patterns:

1. **many-to-one**, allowing network nodes to send data packets up to the sink (root)
2. **one-to-many**, enabling the sink to send unicast data packets to other network nodes down the collection tree



Data collection
(many-to-one)



Downward source routing
(one-to-many)

Data Collection

1. Tree construction

To enable data collection, the protocol must build and maintain a tree rooted at the sink

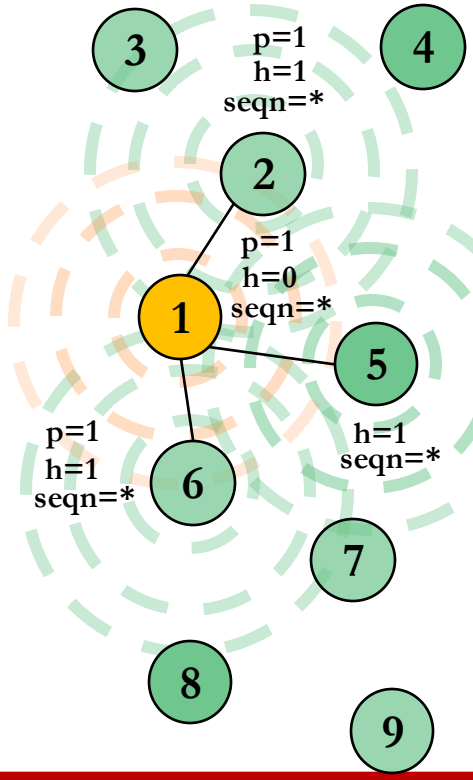
→ Use the **hop count** as your primary routing metric

Basic tree construction logic

Sink: sends broadcast beacon messages with h = 0 and seqn increased upon each new beacon flood transmission

Node: compares h, seqn of the received beacon message against its current metrics, if better

- Considers the source of the beacon as its parent
- Updates its own metric and local information
- After a small random delay TX an updated beacon message in broadcast with h = its own hop count



→ **Periodically** rebuild the tree **from scratch** to cope with network changes and node failures

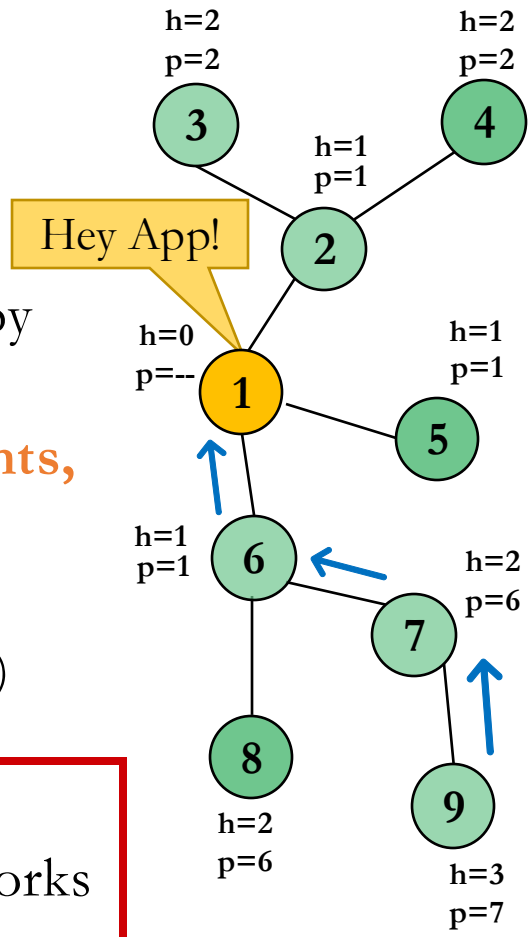
→ *Potentially,* filter out very bad links, e.g., based on the RSSI

Data Collection

1. Tree construction

To enable data collection, the protocol must build and maintain a tree rooted at the sink

→ Use the **hop count** as your primary routing metric



2. Upward data forwarding

Node:
- The originator sends data towards the sink by communicating with **its parent in unicast**
- Forwarders relay data packets **to their parents, again in unicast**

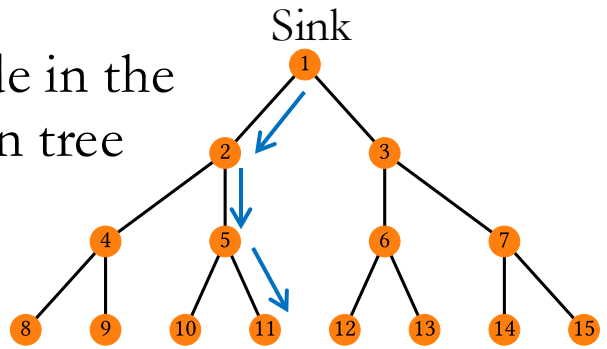
Sink:
- Upon receiving a data packet, **inform the application** (`recv_cb` application callback)

Tip:

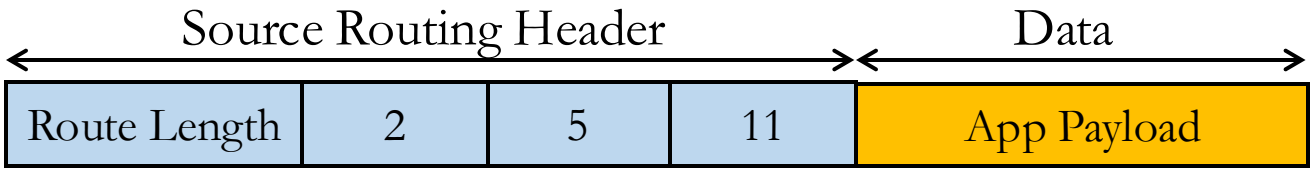
Start from Lab 6-7 code, adapt it, and when it works try to enhance its performance!

Source Routing – Overview

Goal: enable the sink to communicate with *any* node in the network by sending data packets down the collection tree



Basic principle: the sink computes the **entire packet route** (i.e., the complete path from source to destination) and **includes it** in the packet header



→ Non-sink nodes exploit the information stored in the packet header to forward the message to the specific next hop (e.g., 2 →5 →11)

⋙ **No need** to maintain routing tables at non-sink nodes!

How:

1. Acquire complete topology information at the sink
2. Implement the downward forwarding logic (sink and nodes)

Collecting Routing Information

Goal: enable the sink to get topology information to build downward routes
⟶ For every node who its parent is

How: exploit the same routing tree and communication layer used for data collection!

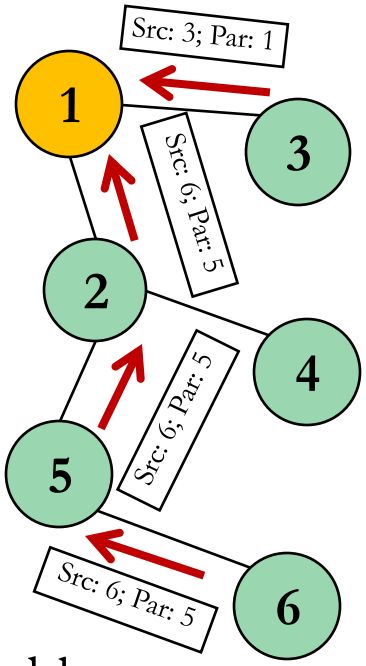
1. Every node should report the link layer address of its parent to the sink (topology report)

2. The sink should keep an up-to-date routing table with child-parent relations for each node

Routing table

Child	Parent
3	1
6	5
2	1
4	2
5	2

Complete view of the network topology at the sink, key for downward source routing. Needs to be **properly maintained**!



Upon receiving a topology report, the sink checks whether an entry for the source of the message exists in the routing table

⟶ If yes, check if to update it with the newly received information

⟶ If not, add a new entry in the routing table

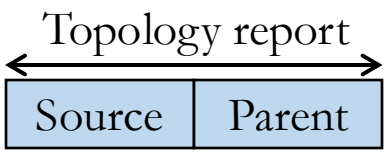
Topology Reports

Q When and how to send topology reports?

→ For you, to think about

How: You might consider sending topology reports in two different ways

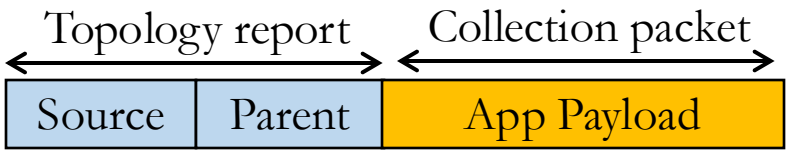
1. **Dedicated reports:** dedicated control traffic to inform the sink of the parent of each node



On demand, reactive to topology changes

Increase contention and consumption

2. **Piggybacking:** attach topology reports to data collection packets
   → When the app sends a packet (`my_collect_send()`) add a header!



Minimal additional overhead

If data rate low, (too) slow reactions

* Credits to "https://www.vecteezy.com/free-vector/like-dislike"

Try to carefully **balance** dedicated reports and piggybacking!

Remember: link failures, packet collisions, and node failures can lead to topology report loss, **compromising** the reliability of downward source routing!

Downward Forwarding – Sink

Goal: (i) compute the **entire route** to the intended destination, (ii) include it in the packet header, and (iii) TX the packet to the 1-hop neighbour

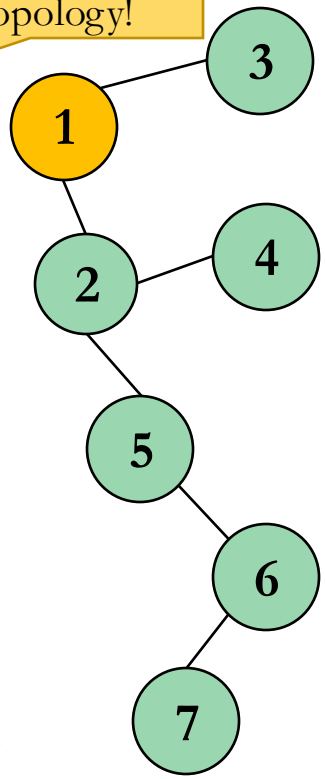
How: exploit the information stored in the **routing table**!

Algorithm:

1. Assign N ≡ intended destination
2. Search for N in the table to find N's parent P
3. If N is not found, drop the packet
4. If P == sink, compute path length and transmit the packet to the next-hop node N
5. Else add N to the source routing list of the packet, assign N ≡ P, go to step 2

Complete view of the network topology!

Child	Parent
3	1
4	2
5	2
2	1
6	5
7	6



Example: Node 7 is our intended destination

N = 7; P = 6

Downward Forwarding – Sink

Goal: (i) compute the **entire route** to the intended destination, (ii) include it in the packet header, and (iii) TX the packet to the 1-hop neighbour

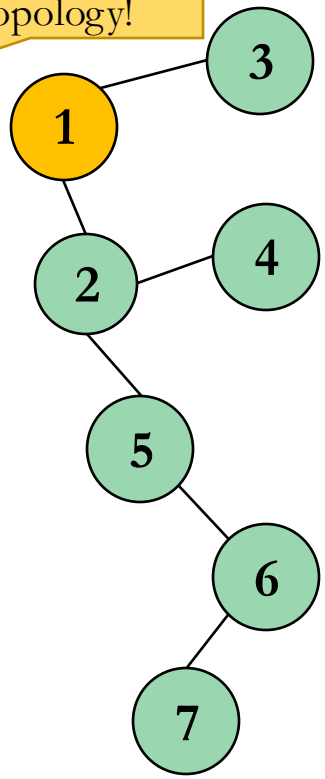
How: exploit the information stored in the **routing table**!

Algorithm:

1. Assign N ≡ intended destination
2. Search for N in the table to find N's parent P
3. If N is not found, drop the packet
4. If P == sink, compute path length and transmit the packet to the next-hop node N
5. Else add N to the source routing list of the packet, assign N ≡ P, go to step 2

Complete view of the network topology!

Child	Parent
3	1
4	2
5	2
2	1
6	5
7	6



Example: Node 7 is our intended destination

N = 6; P = 5

Downward Forwarding – Sink

Goal: (i) compute the **entire route** to the intended destination, (ii) include it in the packet header, and (iii) TX the packet to the 1-hop neighbour

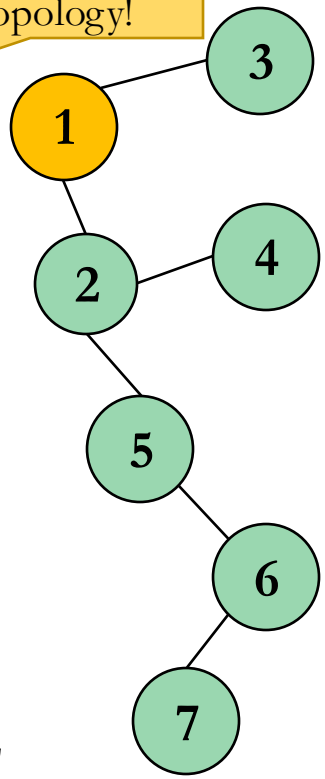
How: exploit the information stored in the **routing table**!

Algorithm:

1. Assign N ≡ intended destination
2. Search for N in the table to find N's parent P
3. If N is not found, drop the packet
4. If P == sink, compute path length and transmit the packet to the next-hop node N
5. Else add N to the source routing list of the packet, assign N ≡ P, go to step 2

Complete view of the network topology!

Child	Parent
3	1
4	2
5	2
2	1
6	5
7	6



Example: Node 7 is our intended destination

5	6	7	Sink App Payload

N = 5; P = 2

Downward Forwarding – Sink

Goal: (i) compute the **entire route** to the intended destination, (ii) include it in the packet header, and (iii) TX the packet to the 1-hop neighbour

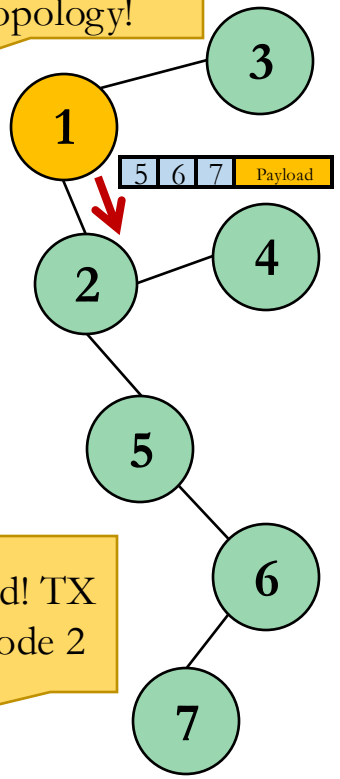
How: exploit the information stored in the **routing table**!

Algorithm:

1. Assign N ≡ intended destination
2. Search for N in the table to find N's parent P
3. If N is not found, drop the packet
4. If P == sink, compute path length and transmit the packet to the next-hop node N
5. Else add N to the source routing list of the packet, assign N ≡ P, go to step 2

Complete view of the network topology!

Child	Parent
3	1
4	2
5	2
2	1
6	5
7	6



Route completed! TX the packet to node 2

N = 2; P = 1

Example: Node 7 is our intended destination

CAUTION

In the process, check for routing loops.
If you detect one, **drop** the packet and **take countermeasures!**

Downward Forwarding – Non-sink

Goal: forward the sink-generated source routing packet down the collection tree towards the intended destination. When reaching it, notify the application

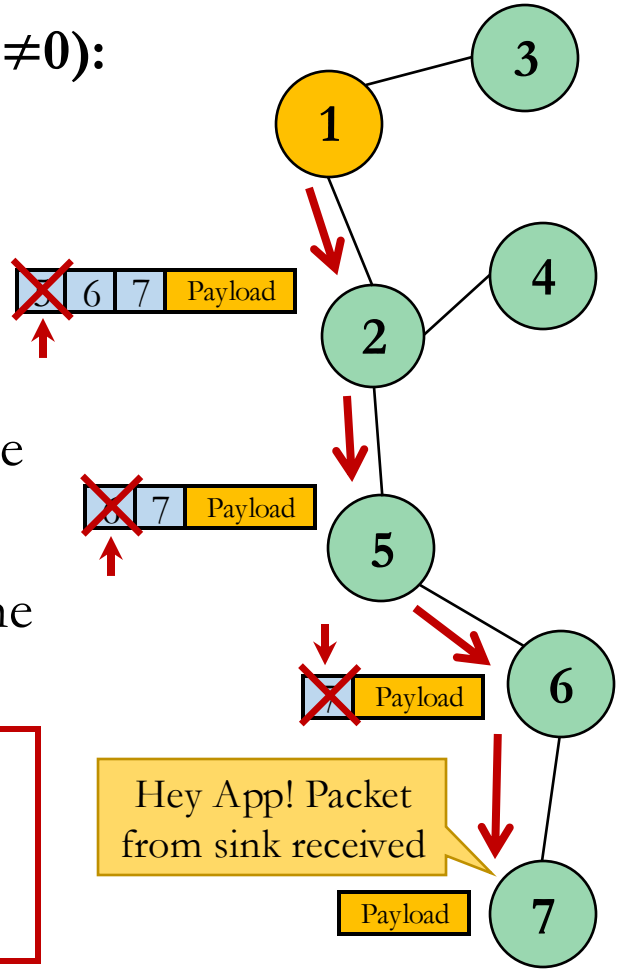
How: exploit the information stored in the routing header of the received packet!

If the routing list is not empty (i.e., route length≠0):

- Extract the address of the next-hop node
  → first element in the routing list
- Remove this information from the packet header
- Decrease the route length by one
- Transmit the reduced packet to the next-hop node

Else:

- The final destination has been reached! Deliver the packet to the application (`sr_recv` callback)



Tip: If you have doubts on how the packetbuffer is managed in Contiki, check `contiki-uwb/contiki/core/net/packetbuf`

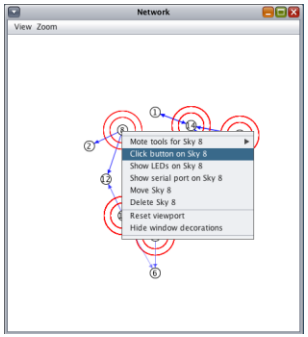
Node failures

As in a real system, rare node failures can occur. Your protocol should preserve *reasonable performance* even if a node becomes temporally unavailable, e.g., by

- Discovering alternative routes that should also be notified at the sink
- Enabling nodes to *quickly* re-join the network after recovering from a failure

Can you do anything better than just wait for the next tree (re)construction?

Simulate node failures in Cooja to asses your system performance!



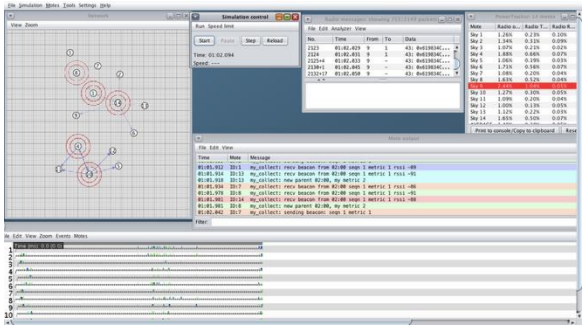
- `app.c` is designed to react to a button sensor event (see Lab 2) by deactivating the node
- Implement `my_collect_close()` to (i) close all active connections, and (ii) stop the running timers (if any)
- As soon as the node's button is clicked again, the node resumes working by calling `my_collect_open()`

NO previously learned information can be exploited by a recovered node!!!

- Assumptions: (i) only one non-sink node at a time can experience malfunctions (ii) nodes failures never split the network topology

Performance Evaluation

1 Cooja Simulations
Testbed Experiments



2 Log
Files

3 Performance
Analysis

4 Write the
Project Report

Cooja simulations

Simulation files:

- 2 simulation scenarios: `test_nogui.csc` and `test_nogui_mrm.csc`
- Same network topology but different channel models (`UDGM` vs. `MRM`)
  → 10 node network, node 1 is the sink

How to run simulations:

- `$ cooja test_nogui*.csc` → Debug, initial experiments, test node failures
- `$ cooja_nogui test_nogui*.csc` → Speed up & automatize testing

Approach:

- Test your solution with two RDC layers: `NullRDC` and `ContikiMAC`
- Run a few simulations per scenario, e.g., changing the random seed, mote start delay →

```
11      <randomseed>123456</randomseed>
12      <motedelay_us>1000000</motedelay_us>
```

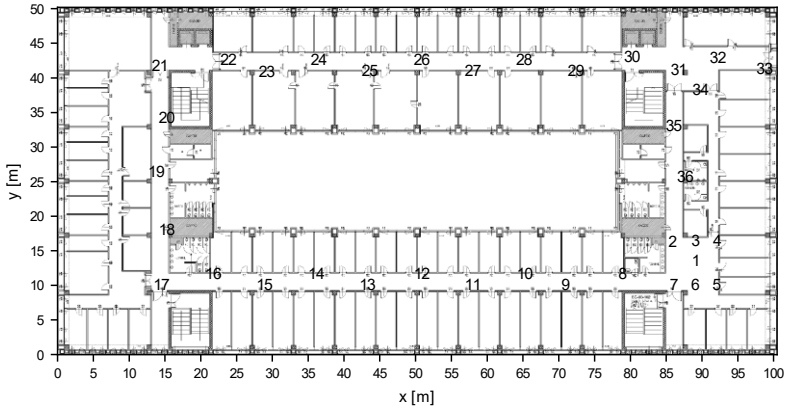
- For each simulation:
  - Store a log file (`test*.log`)
  - Analyse the protocol's performance (PDRs, DC)
    `$ python3 parse-stats.py test*.log`
- Discuss the results of your performance evaluation in the final report!

Suggestion: Run this script on your laptop (not VM), where you can easily install the required Python modules (NumPy & Pandas). Problems? Contact me!

Testbed experiments

Experiment setup – DISI PovoII:

- Node 1 is the sink, all 36 nodes active

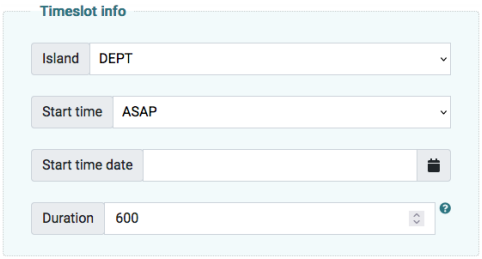


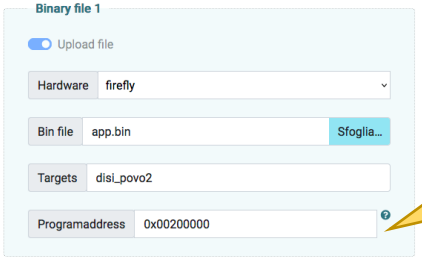
How to run CLOVES experiments: [check Lab 5-7 for further details]

1. **Compile your code for the Zolertia Firefly platform:**

```
$ make TARGET=zoul
```

2. **Log into the CLOVES interface and prepare a job:**





Don't forget to specify it! If not, your test **will not** work as expected

3. **Download, parse and analyze the logs:**

- `https://research.iottestbed.disi.unitn.it/jobs_completed/`
- `$ python3 parse-stats.py job_ID/job.log --testbed`

IMPORTANT NOTE: testbed experiments are *optional*.
However, the **maximum** mark *without* testbed experiments is ***27/30***

Project evaluation

- Implementing the basic functionalities (data collection and source routing) using appropriate communication primitives (broadcast vs. unicast), ***without dedicated techniques for reliability and energy efficiency***, is sufficient to pass the project part of the exam

Even a subset of the requested features may suffice to pass. If you have trouble completing the project, reach out to me!

- To get higher marks, introduce (some) features to reduce contention and collisions, to enhance reliability (e.g., under nodes failure), and to reduce energy consumption
- With few testbed experiments demonstrating that your protocol can work in a real scenario with reasonable performance, you can reach the 30/30 mark

Note:

- The report and the live presentation are always considered in the evaluation
- [OPTIONAL] You are free (and welcome) to introduce new experiments and perform your own data analysis