Course Project: Event-Triggered Control

Low-power Wireless Networking for the Internet of Things
University of Trento, Italy
2021-2022

Project goal

Implement a low-power wireless networking protocol for ***event-triggered control***, namely a low-power wireless communication system capable to detect and timely react to unpredictable events

Protocol rationale

Absence of events:

Minimise network overhead → reduce nodes communications to spare energy

Events detection:

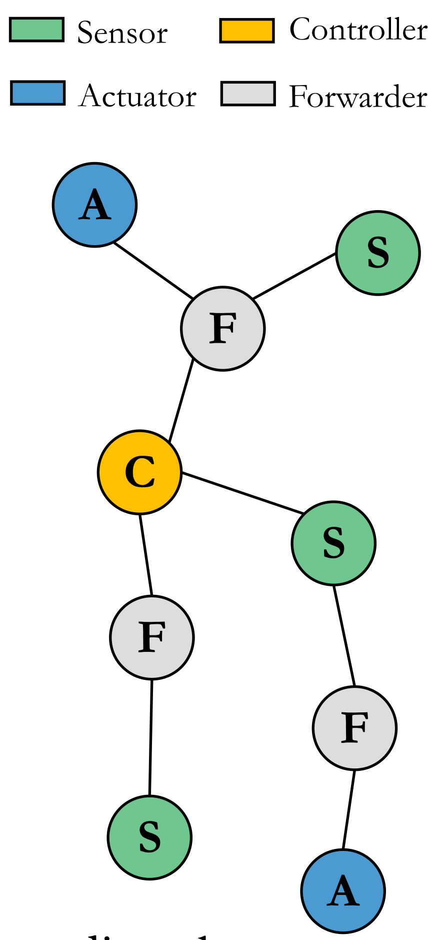
1. Inform the network that an event has been detected and reactions are needed
2. Collect a snapshot of the system status at a central entity (the controller)
3. Distribute control commands over a multi-hop wireless network

System structure & node roles

Nodes are organized in a control loop, holding a predefined role among **sensors**, **controller**, **actuators**, and **forwarders.**

Sensors:

- Periodically sense the environment and evaluate a local triggering condition
- Upon detecting a violation of their triggering condition, distribute an event message
- React to events (detected or notified) by communicating updated readings to the controller



Controller:

- Collects and process sensor readings from all the sensors
- Generates and communicates actuation commands to the actuators

Actuators:

- Wait for commands from the controller and execute them

Forwarders:

- Route traffic between controllers, sensors, and actuators, extending the physical coverage of the system.

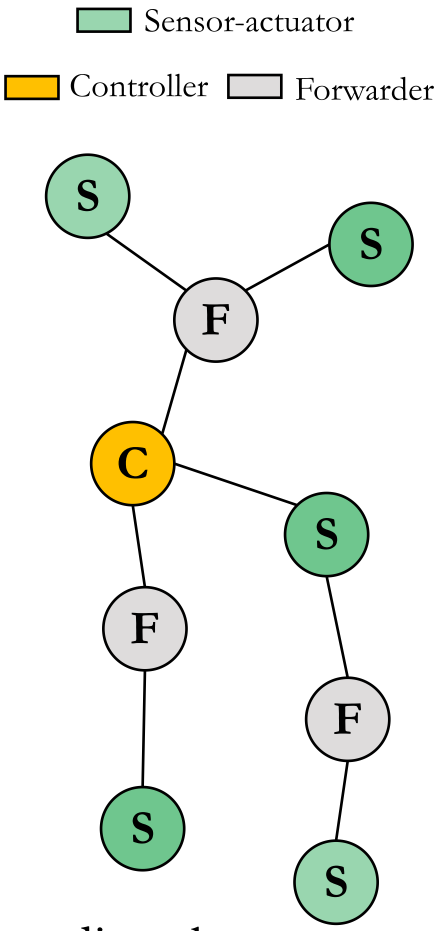
System structure & node roles

In your project sensors & actuators coincide!

Nodes are organized as in a control loop, holding a predefined role among **sensors**, **controller**, **actuators**, and **forwarders.**

Sensors:

- Periodically sense the environment and evaluate a local triggering condition
- Upon detecting a violation of their triggering condition, distribute an event message
- React to events (detected or notified) by communicating updated readings to the controller



Controller:

- Collects and process sensor readings from all the sensors
- Generates and communicates actuation commands to the actuators

Actuators:

- Wait for commands from the controller and execute them

Forwarders:

- Route traffic between controllers, sensors, and actuators, extending the physical coverage of the system.

Protocol Phases

(To be implemented in `etc.c`)

open several connections!!! you need a broadcast for the tree, a broadcast for flooding, etc… this simplifies sw-side stuff. when callback from bc object you know source. data collection is a unicast connection (is it? or maybe not? who knows….). same shit for controller->actuator data

1. Tree construction

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

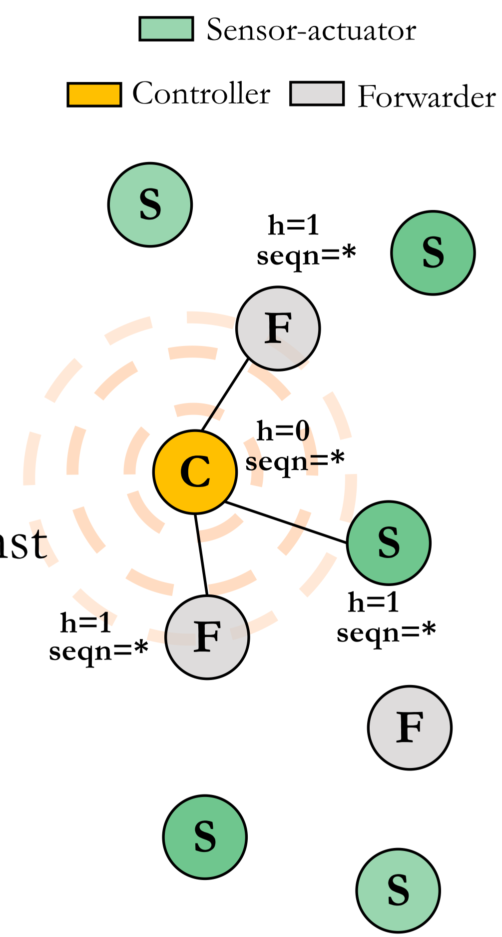
→ Use the **hop count** as your primary routing metric
→ Start from Lab 6-7 code, adapt it, and when it works try to enhance its performance!

Basic tree construction logic

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

Node: Compares **h**, **seqn** of the received message against its current metrics, if better

- Consider the source of the beacon as its parent
- Update its own metric and local information



1. Tree construction

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

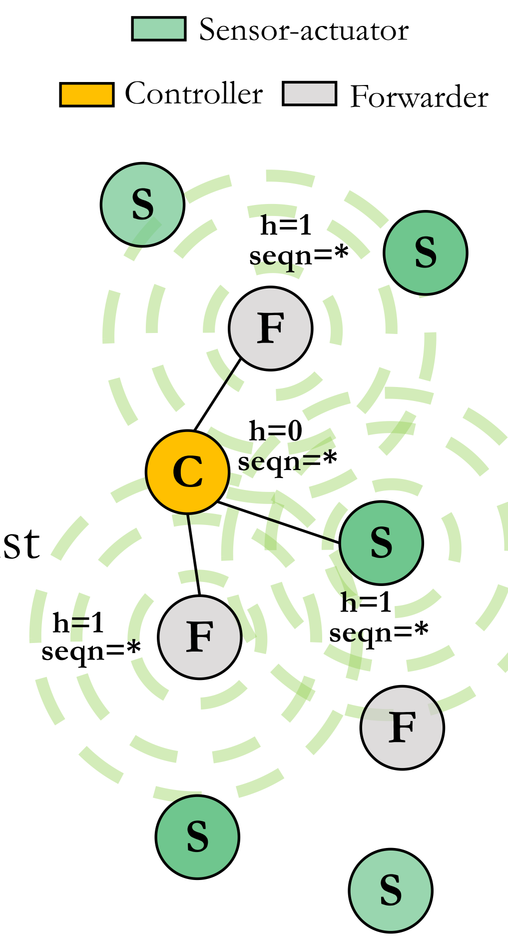
→ Use the **hop count** as your primary routing metric
→ Start from Lab 6-7 code, adapt it, and when it works try to enhance its performance!

Basic tree construction logic

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

Node: Compares **h**, **seqn** of the received message against its current metrics, if better

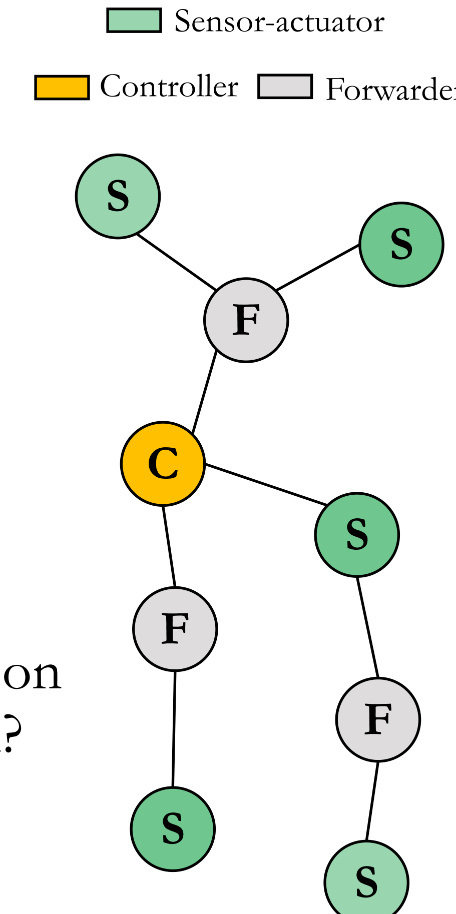
- Consider the source of the beacon as its parent
- Update its own metric and local information
- Broadcast an updated beacon message after a small, random delay



1. Tree construction

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

→ Use the **hop count** as your primary routing metric
→ Start from Lab 6-7 code, adapt it, and when it works try to enhance its performance!



To think about … (Some suggestions)

- How to choose the preferred parent among candidate nodes at the same hop distance?
- Is it enough to consider a single parent per node?
- What is the impact of the tree (re)construction period on the performance (reliability, duty-cycle) of your system?
- …

2. Event detection & dissemination

Goal: Identify when the system is stable and when instead it is perturbed, and let the network act accordingly

Key steps & logic:

1. Sensors periodically acquire new readings and evaluate a local triggering condition → see `sensor_timer_cb()`, already provided!
2. If the system is stable no communication occur
3. When the triggering condition is violated (`value > threshold`) sensors need to be timely informed to communicate their updated readings to the controller
   1. The sensor node detecting the violation starts an `EVENT` flood, which should be propagated network-wide
   2. Upon receiving an `EVENT` message, all sensor nodes prepare themselves to communicate their readings upwards (data collection)
   3. When the controller receives an `EVENT` message, it calls the `ev` application callback and prepares itself for the collection phase

2. Event detection & dissemination

A tip to reduce contention & unneeded transmissions

After detecting or receiving an `EVENT`, nodes should avoid to generate and/or propagate **new** `EVENT` messages for some time!

Rationale

Sensors should be already communicating their most recent readings to the controller, which in turn will compute new actuation commands
→ Let's wait a bit and see if the controller's intervention is enough to bring the system back to stability! If not, the system will trigger again soon

How to?

Exploit `suppression_timer` and/or `suppression_prop_timer` to temporarily suppress the generation and/or propagation of new `EVENTS`.

→ We already provide you with reasonable periods for these `ctimers` (`SUPPRESSION_TIMEOUT_NEW`, `SUPPRESSION_TIMEOUT_PROP`); feel free to modify/adapt them to better fit your specific solution!

`ev_cb-> check seq to see if the message is old`

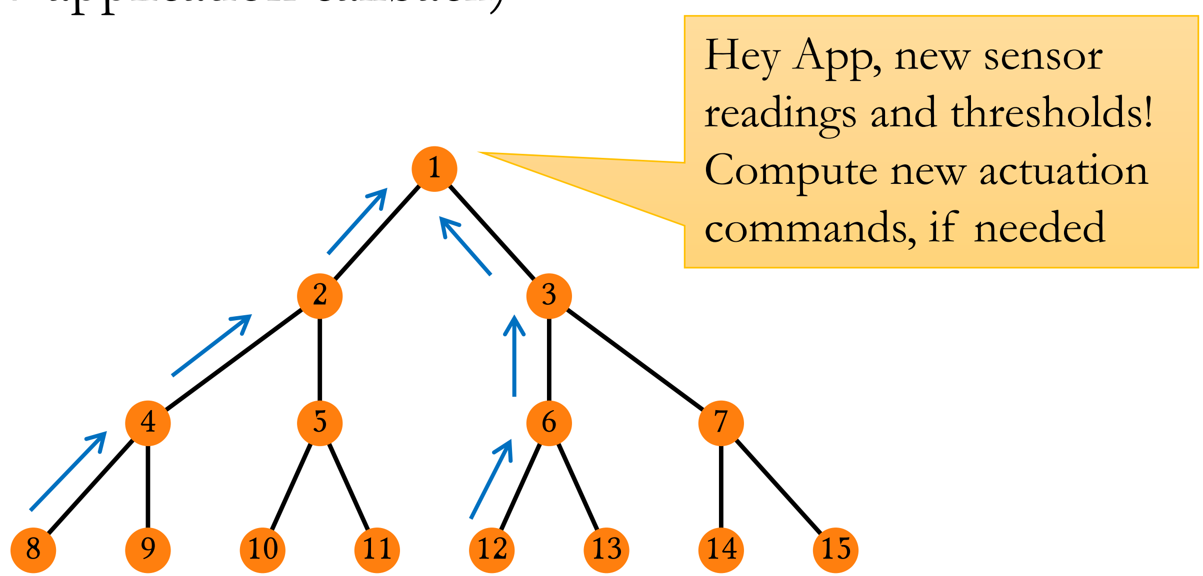
3. Data collection

Goal: Let <u>sensor nodes</u> communicate their current `sensor_value` and `sensor_threshold` to the controller, potentially across multiple hops

How: Follow an approach similar to the one discussed in Lab 7

- <u>Non-controller nodes</u>: forward `COLLECT` messages upwards by leveraging previously learned topology information (phase 1)
- <u>Controller</u>: upon receiving a new `COLLECT` message inform the application (`recv` application callback)

suppose sensors 1, 2 both send their event. ofc s2 event is suppressed, but 2 will still think that it's his event thats being handled. so when the data from sensors 1 and 2 arrive root gonna print collect(s1, 1) even if 2 sent (s2,1). this is done for poor guy parser. same shit for the actuation: the node should follow whatever root says, just actuate and stay shut



4. Actuation command dissemination

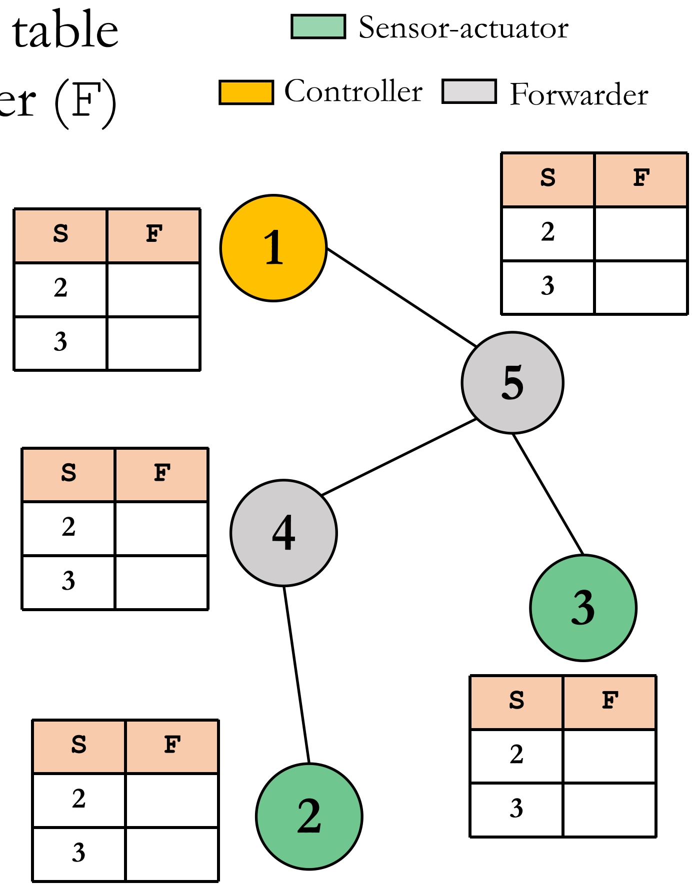
root is not an actuator so we set his actuator callback to null

Goal: Let the controller communicate updated actuation command to the designated sensor-actuator node(s)

How: Exploit the same routing tree used in collection, but **downwards!**

1. Every node locally stores a downward routing table
   Sensor node (S) → 1-hop downward forwarder (F)

implement some logic if you receive a collection but not an event before

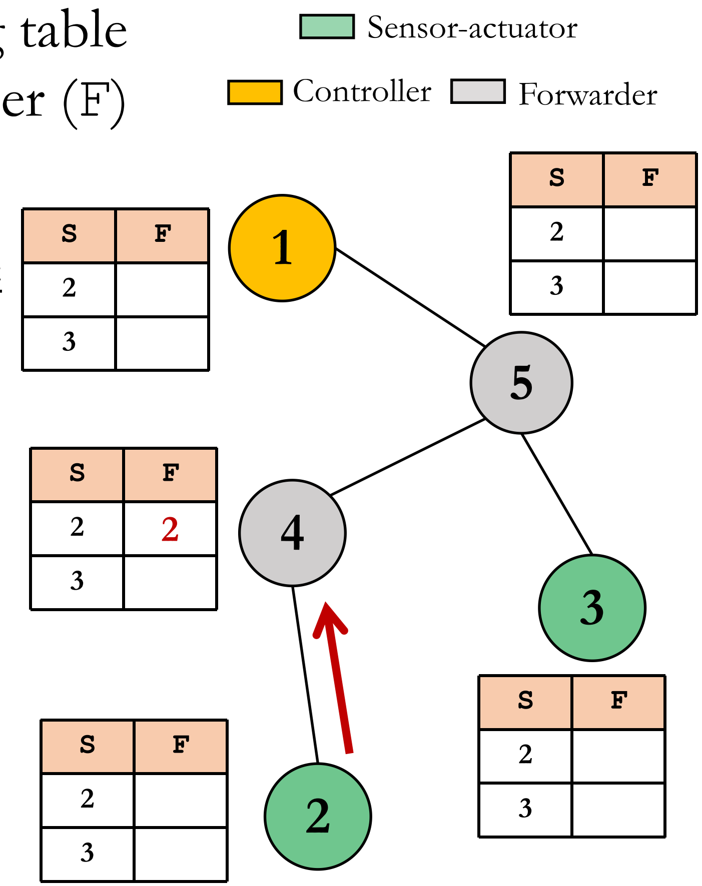


4. Actuation command dissemination

Goal: Let the controller communicate updated actuation command to the designated sensor-actuator node(s)

How: Exploit the same routing tree used in collection, but **downwards!**

1. Every node locally stores a downward routing table
   Sensor node (S) → 1-hop downward forwarder (F)
2. Upon receiving a `COLLECT` message, nodes populate their tables by associating the source of the sensor reading with the sender of the received message

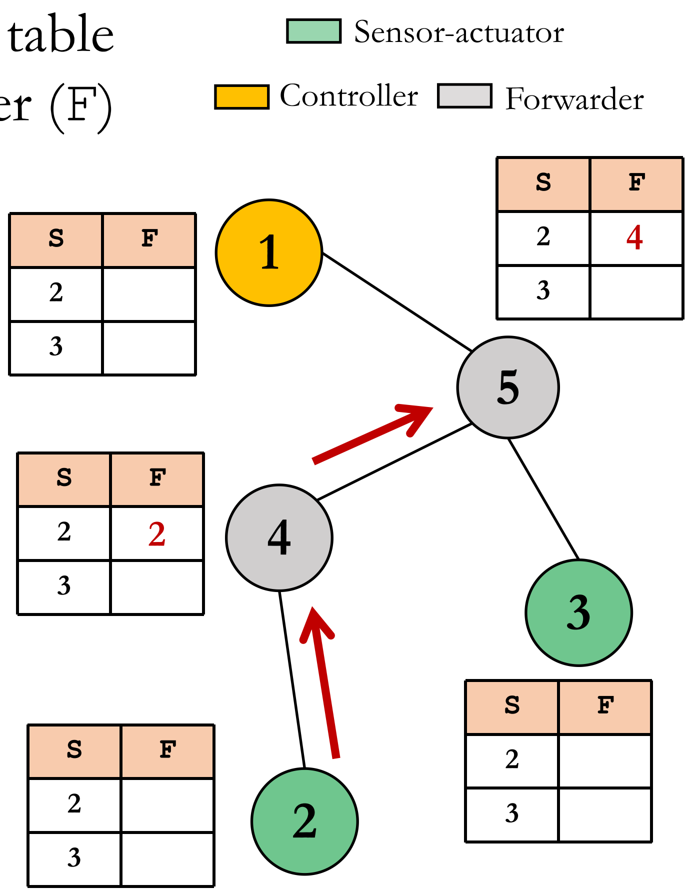


4. Actuation command dissemination

Goal: Let the controller communicate updated actuation command to the designated sensor-actuator node(s)

How: Exploit <u>the same</u> routing tree used in collection, but **downwards!**

1. Every node locally stores a downward routing table
   Sensor node (S) → 1-hop downward forwarder (F)
2. Upon receiving a `COLLECT` message, nodes populate their tables by associating <u>the source</u> of the sensor reading with <u>the sender</u> of the received message

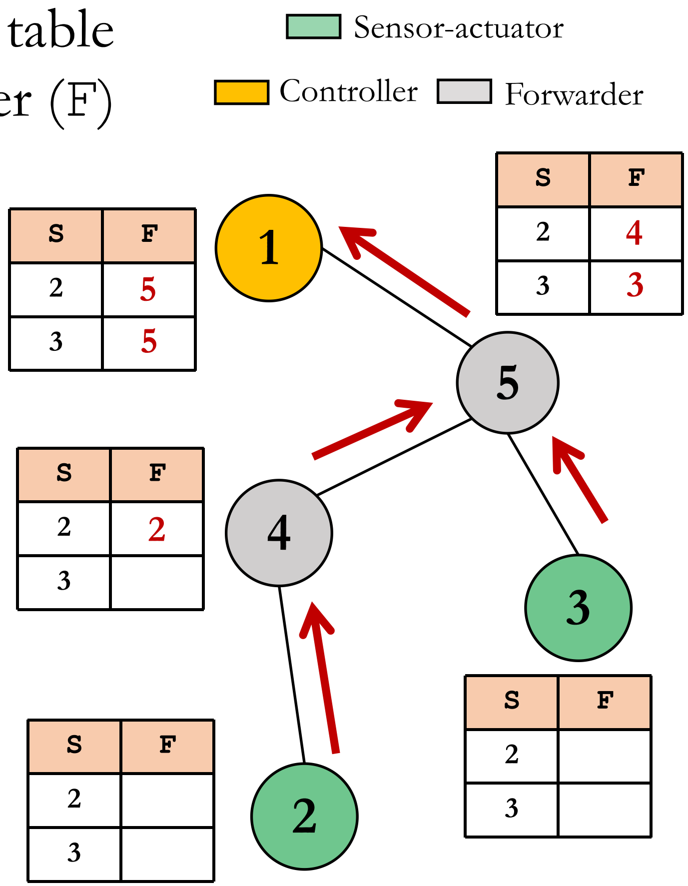


4. Actuation command dissemination

Goal: Let the controller communicate updated actuation command to the designated sensor-actuator node(s)

How: Exploit the same routing tree used in collection, but **downwards!**

1. Every node locally stores a downward routing table
   Sensor node (S) → 1-hop downward forwarder (F)
2. Upon receiving a `COLLECT` message, nodes populate their tables by associating the source of the sensor reading with the sender of the received message

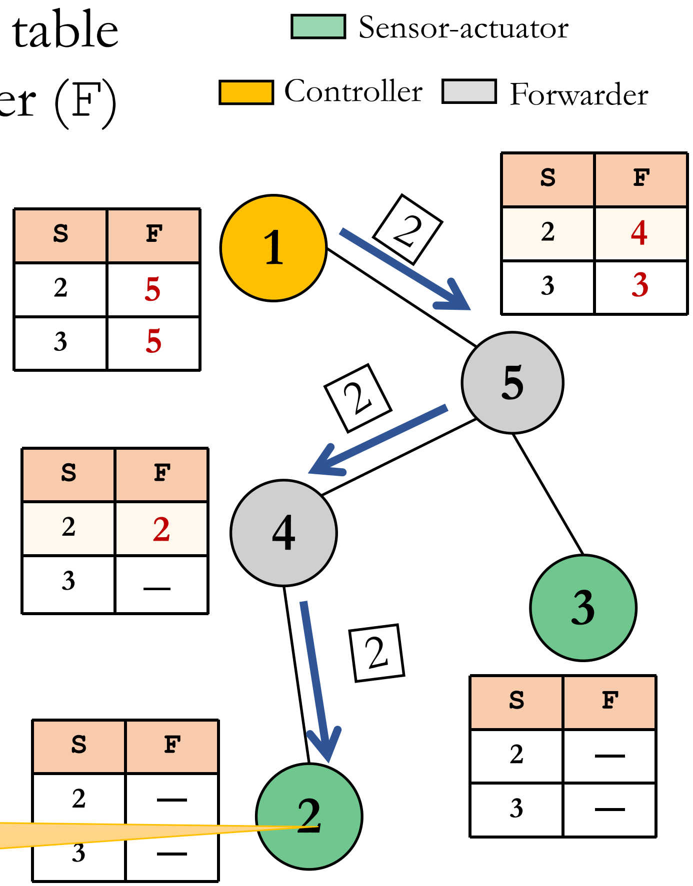


4. Actuation command dissemination

Goal: Let the controller communicate updated actuation command to the designated sensor-actuator node(s)

How: Exploit <u>the same</u> routing tree used in collection, but **downwards!**

1. Every node locally stores a downward routing table
   Sensor node (`S`) → 1-hop downward forwarder (`F`)
2. Upon receiving a `COLLECT` message, nodes populate their tables by associating <u>the source</u> of the sensor reading with <u>the sender</u> of the received message
3. Upon receiving an `ACTUATION` message, nodes check in their downward routing tables the next forwarder for the intended destination



S	F
2	5
3	5

S	F
2	4
3	3

S	F
2	2
3	—

S	F
2	—
3	—

S	F
2	—
3	—

Let's actuate!
(`com_cb` actuation callback)

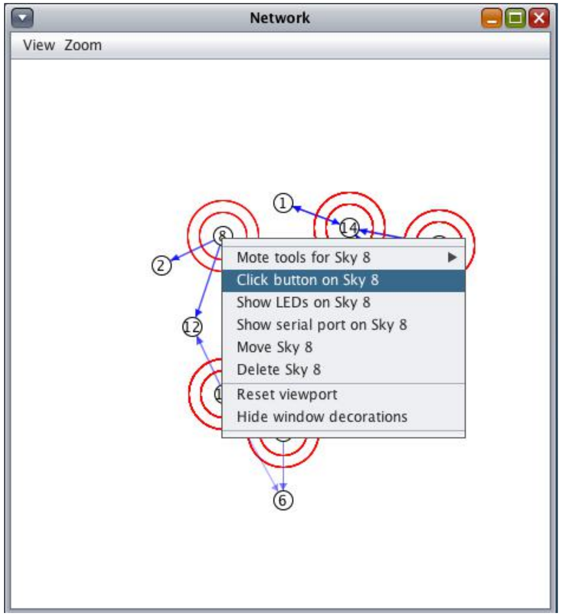
Node failures

As in a real system, nodes of your wireless network can suddenly **stop working**, potentially hampering the performance of your protocol

→ Try to explore ***dedicated*** strategies to reduce the impact of node failures both on the collection and actuation performance

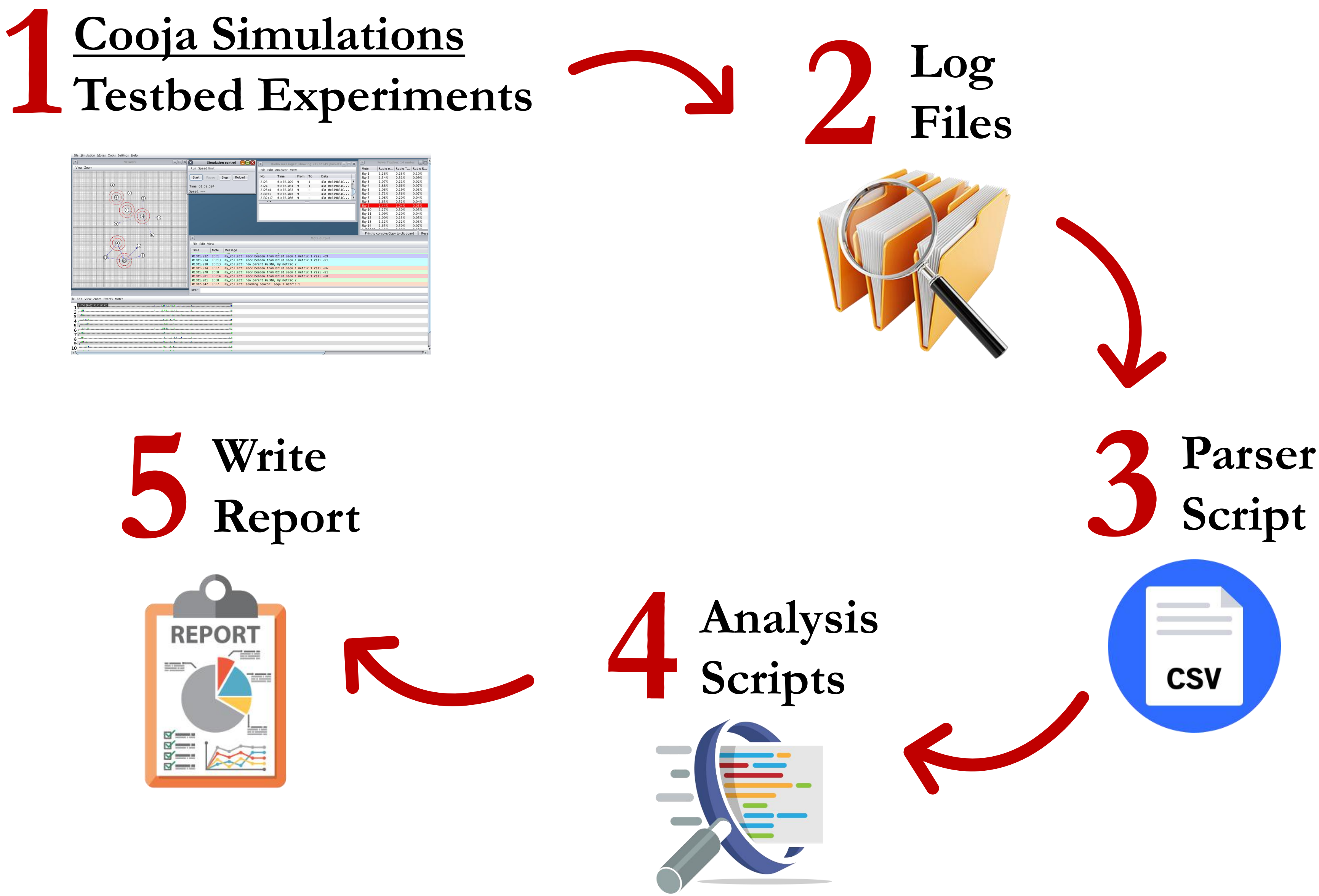
Emulate node failures in Cooja

- Exploit a button sensor event (see Lab 2)
- Upon receiving a button sensor event, nodes should immediately stop working, neglecting received packets and avoiding to forward messages (`etc_close`) `Close all connections here`
- As soon as the node's button is clicked again, the node should resume working as usual



(i) Expect **only forwarders** to experience malfunctions, and (ii) **limit** the node failure analysis **to Cooja experiments**

Performance Evaluation



Cooja simulations

Simulation files:

- 2 simulation scenarios: `scenario1` and `scenario2`, emulating <u>different</u> network topologies. With and without GUI simulation files provided!
- In both scenarios: node 1 is the controller, node 2-6 are source-actuators, the other nodes act as forwarders

How to run simulations:

- `$ cooja scenario*_gui_mrm.csc` → Debug & node failure analysis
- `$ cooja_nogui scenario*_nogui_mrm.csc` → Automatize testing

Approach:

- Run multiple simulations per scenario, changing the random seed, mote start delay, etc
- For each simulation:
  - Store a log file (`scenario*_mrm.log`)
  - Analyse the protocol's performance (PDRs, DC)
    `$ python parse-stats.py scenario*_mrm.log`
- Discuss the results of your performance evaluation in the final report!

Node failures analysis

A few potentially interesting situations to analyse for each scenarios (you are <u>not</u> supposed to try them all)

How to:

Start a normal Cooja simulation experiment <u>with</u> GUI. After some time, press the Cooja button of a specific node to emulate a failure.

Infer (visually in Cooja or via a dedicated script) the impact of the node failure on the system's performance

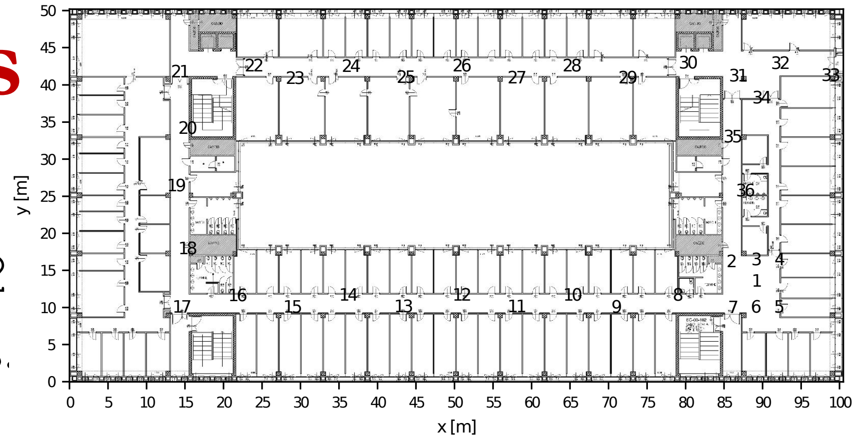
Scenario 1:

- Node 8 failure ⟶ Check sensor 3 performance
- Node 9 failure ⟶ Check sensor 4 performance
- Node 14 failure ⟶ Check sensor 6 performance

Scenario 2:

- Node 8 failure ⟶ Check sensor 2 performance
- Node 9 failure ⟶ Check sensor 5 performance
- Node 14 failure ⟶ Check sensor 3 performance

Testbed experiments



Experiment setup – DISI PovoII:

- Node `1` is the controller, nodes `3`, `12`, `18`, `22`, and `30` the source-actuators. All other nodes act as forwarders.
- You can consider this topology only, or test different ones!

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

- Connect to UNITN VPN
- `$ make TARGET=zoul`
- `$ python3 $TESTBED_CLIENT schedule experiment.json`
- `$ python3 $TESTBED_CLIENT download EXP_ID -u`

Analysis:

- `$ python parse-stats.py job_ID/test.log --testbed`
- <u>**No need**</u> to simulate/analyze node failures with testbed experiments!

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