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

Low-power Wireless Networking for the Internet of Things
2025–2026

In this project, you will extend the data collection protocol that you have implemented in class with a simple multi-hop source routing protocol. As a result, you should have a routing protocol that supports two traffic patterns: *i)* many-to-one, allowing network nodes to send data packets up to the sink (root) and *ii)* one-to-many, enabling the root to send unicast data packets to other network nodes down the collection tree.

Source routing. In source routing, each data packet contains *complete* routing information to reach the destination. These routes are computed by the source node (the sink/root in our case) and included in the packet header. A key advantage of source routing is that intermediate nodes *do not* need to maintain routing tables in order to forward the packets: the packets themselves already contain *all* the necessary information.

Routing: How are the routes constructed? For the sink to be able to construct the routes it needs to know the network topology (i.e., the connectivity graph). Notably, in the case of one-to-many routing a spanning tree suffices instead of the full connectivity graph. You have already implemented a distributed algorithm for building the spanning tree for data collection. The same tree can be reused for one-to-many traffic, provided that the sink collects enough information from the nodes to successfully reconstruct it.

To build a tree it is sufficient to know *the parent of every node*. Therefore, each node should report the link layer address of its parent to the sink, e.g., by using the data collection protocol already implemented in the labs. The sink node should keep an *up-to-date* table of the child-parent relations for each node in the network (see Table 1).

Example. Consider the network shown in Figure 1. This network topology could represent the collection tree built by your protocol at a given time. When node **5** joins the network and chooses **2** as its parent, 5 sends a packet to the root (node **1**), informing the root that **2** is its parent. Hence, data collection packets from **5** will follow the path $5 \rightarrow 2 \rightarrow 1$. Then, the root (**1**) stores in a source routing table an entry for node **5**, setting **2** as its parent node (see Table 1). This indicates that to reach **5**, a packet needs to be forwarded through **2**. If an entry for **5** is already present, it is updated with the newly received information. Hence, to send a packet from the sink **1** down to **5**, the packet should travel the path $1 \rightarrow 2 \rightarrow 5$, i.e., the same path as for data collection but in the opposite direction.

Topology updates: Dedicated reports and piggybacking. Changes in the network topology tree should trigger new topology reports from nodes to maintain an up-to-date topology view at the sink. However, in dynamic environments, this may result in frequent topology updates that could significantly increase the control traffic and the radio duty cycle, leading to high energy consumption and decreased performance (e.g., due to packet collisions).

To cope with this issue, your protocol should send topology updates in two different ways: *i)* dedicated topology reports and *ii)* piggybacking. For instance, each node after joining the network may send a *dedicated topology report* to inform the sink about its selected parent, enabling the sink to gather enough topology information for downward source routing. On the other hand, to reduce control traffic, nodes could *piggyback topology information in data collection packets*, i.e., application packets whose destination is the sink. To this end, when the application of non-sink nodes calls the send function, your collection protocol may add the link layer address of the node parent to the packet, allowing the sink to refresh its topology information and reducing the need of dedicated control traffic. Nonetheless, in applications with infrequent data collection packets, piggybacking information may not be sufficient. Hence, your protocol should also send dedicated topology reports. Consider also that topology updates can be lost, e.g., due to link failures or packet collisions; when this happens the sink may not be able to send traffic downwards, reducing the reliability of your solution. Moreover, node failures can occur, as discussed next, further increasing the likelihood of missing topology

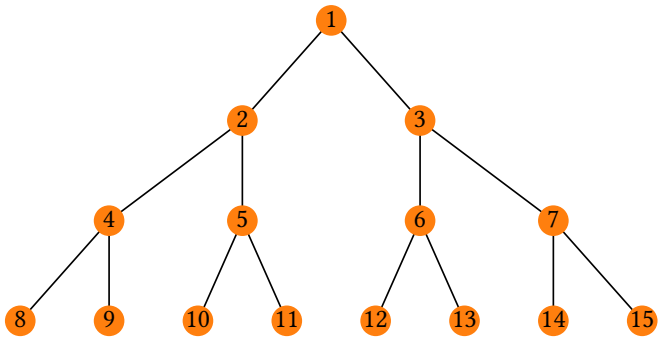

Figure 1: Example data collection network topology.

Table 1: Source routing table based on the network topology of Figure 1.

Child	Parent
2	**1**
3	1
4	2
5	**2**
6	3
7	3
8	4
...	...

updates and data packets.

Your task is to strike a balance between dedicated topology reports and piggybacking to optimize the trade-off between system reliability and energy consumption. Furthermore, your implementation should include mechanisms to reduce the chance of packet collisions and to improve the overall reliability of the system.

Downward packet forwarding (one-to-many communication). In addition to the many-to-one forwarding already seen in class, your protocol should provide one-to-many communication. This means that the application running on the root (i.e., the sink) may request sending data to any node in the network. Similarly, the application at any other node should be notified when a packet addressed to the node arrives. To implement this, the root should build the path (i.e., the sequence of forwarders) to the destination and put this list into the packet header. Forwarders receiving a packet should send it to the next node in the list or deliver it to the application if the destination is the current node. More formally, the algorithms of the root and a forwarder are described in the following.

1. **At the root**: When the application at the root **S** requests to send a packet to the destination node **D** the following algorithm should be executed:
   1. Assign N := D
   2. Search for node N in the routing table as constructed above to find the parent P of N
   3. If N is not found or a loop is detected, drop the packet
   4. If P == S, compute the 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.
2. **At the forwarders**: When a non-root node receives a packet to be forwarded, it modifies the routing header by *removing* the next hop node address from the list, reduces the path length by 1, and transmits the reduced packet with the payload to the next hop node. If a node receives a packet with no forwarding list, it delivers the packet to the application. The same procedure is repeated until the packet reaches the destination.

Packet format. The data frame should contain (*at least*) the route/path length, the source routing list, and the data payload (Figure 2).

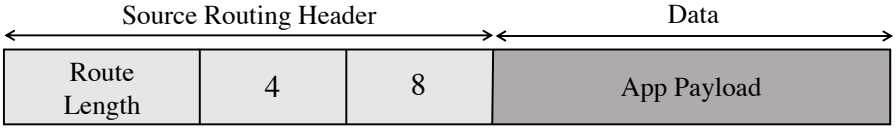

Figure 2: Source routing frame format to send a data packet from the sink **1** to node **8** following the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$.

Node failures. As discussed in class, in real systems nodes failures can occur. Your protocol should preserve reasonable performance even if nodes become temporarily unavailable, e.g., by *i)* rapidly discovering alternative routes that should be also notified to the sink, and *ii)* enabling nodes to quickly re-join the network after recovering from a failure. When a node recovers, no previously learned information (e.g., about the network topology) can be exploited by the node.

By simulating a node failure (and recovery) in Cooja you can easily assess the robustness of your solution to similar circumstances. In the presentation for this project, you can find additional details on how to simulate node failures by leveraging Cooja button sensor events. As simplifying assumptions, you can expect *i)* only non-sink nodes to experience malfunctions, *ii)* failures to never affect multiple nodes simultaneously, i.e., only one node at a time can become temporarily unavailable, and *iii)* nodes malfunctions to never split the network topology in two disconnected parts.

Application interfaces. In addition to the data collection application interface, your networking layer exploits a new interface for one-to-many delivery. Similar to the many-to-one data collection interface implemented in labs 6 and 7, this interface provides a sr_send(struct my_collect_conn *conn, const linkaddr_t *dest) function and a sr_recv(struct my_collect_conn *conn, uint8_t hops) callback. sr_send should accept the packet destination link layer address as a parameter. In a non-root node, sr_send should return an error immediately. The sr_recv callback should be signaled on non-root nodes only upon receiving a source routing packet. Listing 1 shows the source routing API that implementations should follow. The application and header files provided together with this document should further clarify the usage of these functions.

Testbed experiments. Simulations are a very useful tool for protocol design. Nonetheless, testbed experiments are key to understand the effect of the environment on our solutions. Once your protocol achieves satisfactory results in Cooja, you can run the same tests in the testbed (disi_povo2 topology), using Zolertia Firefly nodes. You may need to make small adaptations of your implementation for the testbed.

Testbed experiments are optional. However, the maximum mark for a project submitted with no evaluation in the testbed is 27/30.

Protocol evaluation. You should analyze the performance of your solution in terms of *i)* packet delivery rate (PDR), focusing on the reliability of data collection (many-to-one traffic, from nodes towards the sink) and source routing (one-to-many traffic, from the sink towards nodes); and *ii)* duty cycle (DC). Towards this end, you may use `parse-stats.py`, a Python script provided as part of the project template, which parses your Cooja or testbed `.log` files and computes the above mentioned metrics.

We encourage you to study the impact of the various design decisions you have made, and discuss how you balance reliability and energy efficiency in your system.

The core of your analysis should focus on Cooja experiments with no simulated node failures, which you can run without limitations in a controlled setting ensuring repeatability. As a starting point, evaluate your implementation relying on the Cooja simulation files we provide as part of the project template. *Optionally*, you may define your own simulation scenarios to assess the performance of your system under other topologies and conditions. To evaluate the efficacy of your solution under rare node failures, you can run (a few) additional experiments in Cooja using the graphic user interface: by pressing a Cooja node button you can easily emulate a node failure and recovery. Finally, a few testbed experiments are enough to showcase the performance of your solution in a real environment. No node failure needs to be emulated in the testbed experiments.

Report. You are expected to write a report with a brief description of the design decisions behind your protocol logic, implementation details, results of your performance evaluation, and a concise summary of the findings. To present your results, we encourage you to add tables or make meaningful plots, e.g., showing the node PDR and DC. In Cooja, we suggest repeating experiments with a few different random seeds. You can easily change the random seed value at the top of the Cooja simulation file.

Implementation Notes.

- **Collisions.** Because of the flood employed to build the collection topology, many nodes may send topology reports to the sink simultaneously. This may, in turn, produce collisions between topology reports and/or other packets. You should consider how to reduce collisions.
- **Source Routing Header.** All the information necessary for forwarding a packet from the sink towards a node should be contained in the source routing packet header. The number of address entries in the header is equal to the number of nodes on the path minus one.
- **Routing Loops.** Source route entries should not repeat as this causes unwanted loops. Any loops should be detected by the root while constructing the list of forwarders. If there is a loop, the packet should be dropped.
- **Node failures.** The application file provided (app.c) is designed to emulate a node failure when a button is pressed on a node in Cooja. When a node fails, you should exploit my_collect_close(struct my_collect_conn *conn) to close *all* open connections and stop the running timers (if any). When the button is pressed again, the node recovers. Remember that no previously learned information (e.g., about the network topology) can be exploited by the node. To ensure this, you can simply reset all the node's variables.

- **Radio Duty Cycling (RDC).** Your protocol must be tested/evaluated with two RDC layers: NullRDC and ContikiMAC. Make sure your protocol works appropriately with both.
- **Radio models.** In Cooja, your protocol must be tested/evaluated with the MRM radio model; make sure your solution works appropriately with it by exploiting the `test_nogui_mrm.csc` Cooja simulation file. In the code template, we provide an additional Cooja simulation file (`test_nogui.csc`) that relies on the UDGM radio model. We recommend using it initially to debug and test your solution; if you want, feel free to report also UDGM results in the project report.
- **Assumptions.** Assume that the number of nodes in the network and the maximum path length are bounded. Use C defines to set these parameters using reasonable values (e.g., MAX_NODES=40, MAX_PATH_LENGTH=10).
- **Logging.** The provided application already includes several `printf` functions to log send and receive events. You **should not** modify these output strings, as they are used to automatically evaluate the performance of your protocol.

Code Template. We provide several files to simplify your development:

- **app.c** where you can find the higher-level logic of the sink and network nodes, the implementation of the callback functions, and the code to simulate node failures in Contiki. *Do not* change this file unless you really need it; if you do need to make changes, please *document them clearly* in the project report.
- **my_collect.h**, i.e., the header file of your Rime layer protocol with the minimum API you should provide. You have to complete it; furthermore, feel free to extend the header file with other functions or data structures if needed.
- **tools** to enable duty cycle estimation in both simulated Tmote Sky nodes in Cooja and Zolertia Firefly nodes in the testbed. The files within the `tools` folder must not be changed.
- **parse-stats.py** a Python script that parses your Cooja or testbed `.log` files and computes the PDR and DC of each node alongside aggregated statistics. You may use this script to analyze the performance of your protocol implementation. Feel free to extend/modify the script if needed. For the script to work, you will need to install Python's NumPy and Pandas modules. We recommend running this script on your laptop (rather than on the virtual machine), where you should be able to easily install both modules via the terminal, e.g., as `$ pip install pandas numpy`. In case of problems, don't hesitate to contact the teaching assistant (matteo.trobinger@unitn.it).
- **Makefile** to compile your code. You may add new source code files to it.
- **project-conf.h** to configure your application and protocol. For instance, to change the RDC layer between NullRDC and ContikiMAC, you can simply update the `project-conf.h` file configuration as seen in Lab 7.
- **Cooja simulation files** to test and evaluate your protocol. You are provided with two configurations (`test_nogui.csc`, `test_nogui_mrm.csc`); the first uses the UDGM radio model, while the second is based on MRM. Results for the MRM radio model should be included in the report; if you want, feel free to also report those from the UDGM case. Optionally, you may also define your own simulation files in addition to those provided.

Listing 1: Source Routing API.

```
/* Source routing send function:
 * Params:
 *      conn: pointer to the collection connection structure
 *      dest: pointer to the destination address
 * Returns non-zero if the packet could be sent, zero otherwise.
 */
int sr_send(struct my_collect_conn *conn, const linkaddr_t *dest);

/* Source routing recv function callback:
 * This function must be part of the callbacks structure of
 * the my_collect_conn connection object. It should be called when a
 * source routing packet reaches its destination.
 * Params:
 *      conn: pointer to the collection connection structure
 *      hops: number of route hops from the sink to the destination
 */
void (* sr_recv)(struct my_collect_conn *conn, uint8_t hops);
```

Rules of the game: How to (and how not to) work on the project

- The project is individual. Due to the structure of the course, students are encouraged to deliver it before the beginning of the second semester. However, students have time until the start of the next-year course to submit it. Students who are unable to meet this deadline should contact the instructor well in advance, especially if they have already passed the written exam.
- You should submit *(i)* the Contiki source code and *(ii)* a brief report with the description of your solution, your evaluation results, and your conclusions. The report must be in English.
- You should demonstrate that the project works as expected using the Cooja simulator and/or the testbed in front of the teaching assistants and/or the instructor.
- The code must be properly formatted. Follow style guidelines (e.g., Contiki code style).
- You **must** contact through e-mail the instructor (gianpietro.picco@unitn.it) **and** the teaching assistant (matteo.trobinger@unitn.it) well in advance, i.e., at least a couple of weeks before the presentation. If you have time constraints, it is up to you to make them known to the instructors in due time. Remember that the instructors have their own availability constraints and may not be able to accommodate exactly the date/time you need.
- Both the code and the report must be submitted in electronic format via email at least five days before the meeting. The project report must be a single, self-contained PDF. All code must be sent in a single tarball consisting of a single folder (named after your surname) containing all your source files and any additional files relevant to the project evaluation (if any).
- The code **should not** be published in GitHub or any other online service/tool. While we encourage students to become part of the open source community, we believe sharing your project code can help other students develop their implementation, which can be unfair to fellow students and result in plagiarism (see below).
- The project will be evaluated based on the technical implementation (correctness and efficiency) *and* the report quality, which should demonstrate the student understanding of the problem at hand.

Plagiarism is not tolerated. Students whose project is partially copied/adapted from other students' projects will undergo disciplinary measures. Depending on the gravity of the plagiarism, these may involve reporting the student to the highest disciplinary bodies of the university, therefore possibly jeopardizing the offending student's academic career. If you are afraid you may not complete your project, get in touch with the instructors. Remember: an incomplete project will be considered more positively than one we discover to have been partially or totally copied from someone else's project.