Course Project:
Source Routing for Downward Data Traffic

Laboratory of Wireless Sensor Networks
2017–2018

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 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, since the packets themselves already contain all the necessary information.

Routing: How are the routes constructed?
For the root to be able to construct the routes it needs to know the network topology (connectivity graph). In the case of one-to-many routing a spanning tree would suffice 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 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).

Topology Reports: Dedicated reports and Piggybacking
An important aspect to implement source routing is to acquire the required topology information (i.e., the tree) in the source node (the sink) to build the routes. Changes in the network topology tree should trigger new topology reports to maintain an up-to-date topology view in 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.

To cope with this issue, your protocol should send topology reports in two different ways: *i)* **dedicated topology reports** and *ii)* **piggybacking**. For instance, each node after joining the network should send a dedicated topology report to inform the sink about its selected parent, enabling the sink to gather enough topology information for downwards source routing. On the other hand, to reduce control traffic, nodes should piggyback topology information in data collection packets, i.e., application packets whose destination is the sink. To this end, when the application calls the `send` function of your collection protocol, the protocol may add the link layer address of the node's parent to the packet, allowing the sink to refresh its topology information and reducing the need of dedicated control traffic. Nonetheless, in applications with a low data rate, i.e., infrequent data collection packets, piggybacking information may not be sufficient. Hence, your protocol should also send dedicated topology reports. Consider also that if topology reports are lost, the sink may not be able to send traffic downwards, reducing the reliability of your protocol.

Your task is to balance dedicated control traffic with piggybacking topology information in data packets, aiming to refresh the node's parent in the sink, e.g., once every 30s or every minute.

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, node **5** should send 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**) should store in a source routing table an entry for node **5**, setting as its parent node **2** (see Table 1). This indicates that to reach **5**, a packet needs to be forwarded through **2**. If an entry for **5** was already present, it should be updated with the newly received information. Hence, to send a packet from the sink **1** down to node **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.

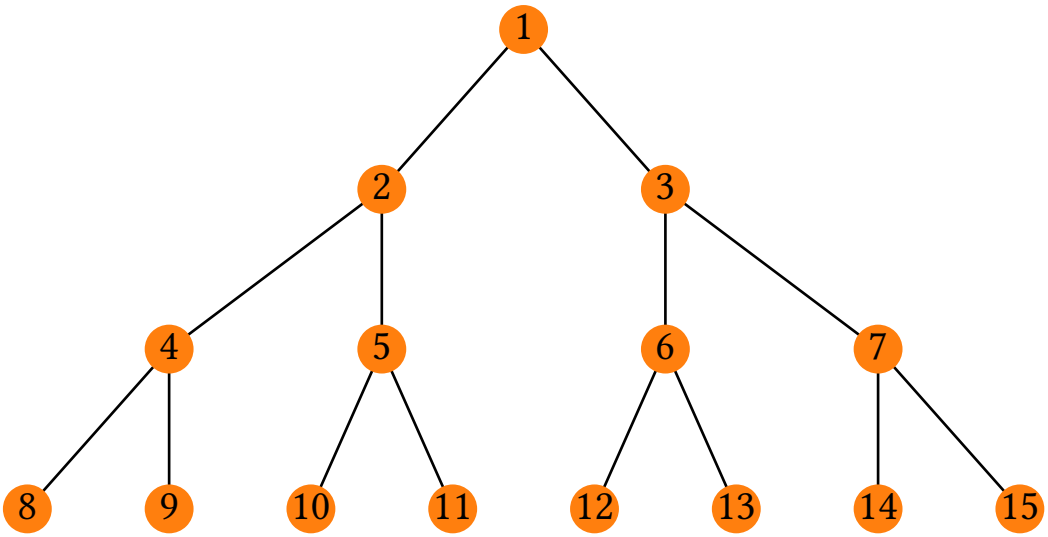

Figure 1: Example data collection network topology.

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
...	...

Packet Forwarding (one-to-many communication).
In addition to the many-to-one forwarding that you already have, your protocol should provide one-to-many delivery service to the application. This means that the application running on the root may request sending data to any node in the network. Similarly, the application at an ordinary network node should be notified when a packet bound to the current node arrives. To implement this, the root should build the path (the sequence of forwarders) to the destination and put this list into the packet header. The intermediate nodes (forwarders) when receiving a packet should forward it to the next forwarder 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 node **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 == root, transmit the packet to 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 an intermediate node receives a packet to be forwarded, it modifies the routing header by removing the next hop node address from the list and transmits the reduced packet with the payload to the next hop node. If a node receives a packet with empty forwarding list, it delivers the packet to the application. 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 illustrates the frame format for a data packet sent by the sink **1** to node **8** through the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$.

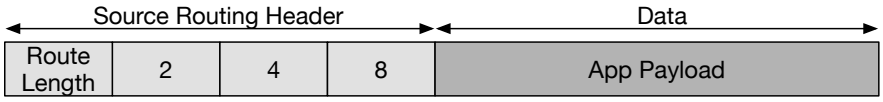

Figure 2: Source routing frame format to send a data packet from the sink **1** to node **8** following the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$.

Miscellaneous 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 broadcast beacons. You SHOULD consider how to avoid or reduce collisions.
- **Source Routing Header.** All the information necessary for forwarding should be contained in the data 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 will cause 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.
- **Radio Duty Cycling (RDC).** Your protocol will be tested/evaluated with two RDC layers: NullRDC and ContikiMAC. Make sure your protocol works appropriately with both RDC layers.
- **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=30, MAX_PATH_LENGTH=10).

Supported data flows. The system should support bidirectional communication: one-to-many unicast and many-to-one data collection traffic to and from a predefined sink node **S**.

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

Listing 1: Source Routing API.

```
/* Source routing send function:
 * Params:
 *      c: 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 *c, const linkaddr_t *dest);

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

Rules of the game.

- The project is individual and the student is suggested to deliver it before the beginning of the second semester, and in any case no later than September 2018.
- You should submit *i)* the Contiki source code and *ii)* a short report with a description of your solution. Afterwards, you should demonstrate that the project works as expected using the COOJA Simulator.
- The code MUST be properly formatted. Follow style guidelines (e.g., this one).
- You MUST contact through e-mail the instructor (gianpietro.picco@unitn.it) AND the teaching assistants (p.corbalanpelegrin@unitn.it, timofei.istomin@unitn.it), well in advance, i.e., a couple of weeks, before the presentation.
- Both the code and the documentation must be submitted in electronic format via email at least three days before the meeting. The documentation must be a single self-contained pdf/txt. All code must be sent in a single tarball consisting of a single folder (named after your surname) containing all your source files.
- The project will be evaluated for its technical content (algorithm correctness and efficiency). *Do not* spend time implementing unrequested features—focus on doing the core functionality, and doing it well.
- The project is demonstrated in front of the instructor and/or assistant.

Plagiarism is not tolerated. An incomplete project will be considered more positively than one with parts of the code adapted from someone else.