Matteo Beltrame, mat: 231443

LPIoT project report: multi-hop wireless sensor network

Protocol design

The goal is to design a multi-hop wireless sensor network protocol enabling many-to-one and one-to-many communication using source routing.

In the network, a single node is the sink, and it is responsible for keeping an updated routing table, to enable source routing.

Routing Metrics

The protocol uses two routing metrics combined:

- *Hop count:* the number of hops required to reach the sink
- *RSSI:* the relative signal strength index

The primary metric is the hop count, this means that a node will change its parent provided the hop count is lower, independently of the RSSI value provided it is above a certain threshold.

In fact, the RSSI value is used in two different ways:

1. If a link has an RSSI that is below a certain threshold, the link is automatically discarded
2. If two links have the same hop count, the one with the better last RSSI value is selected

The first constraint on the RSSI threshold allows to discard very low-quality links, while the second one allows to always select the best path among multiple with the same primary metric value, increasing the general reliability.

Topology reconstruction and communication

Periodically, the sink sends a beacon in broadcast and the topology is adjusted with a flooding algorithm. Upon receiving a beacon, nodes update their parent based on the routing metrics.

In the implemented protocol, whenever the beacon sequence number is greater than the last received beacon, the node forces the update of the parent independently of the routing metrics, to ensure responsivity to topology changes. However, nodes communicate to the sink the topology update only if the parent has been changed. This means that, whenever a new flooding algorithm is carried out and the nodes are forced to update the parent, an additional check is run to determine whether the new assigned parent is different from the old one. This simple check allows to greatly reduce traffic in static situations in which the topology is changed with low frequency, as the nodes almost never change the parent.

When nodes change parent, they have two different ways to inform the sink about it:

- Piggybacking: embed the new parent information in the header of a normal application message
- Dedicated topology update: send a dedicated message to inform the sink about the topology change

The balance between the two is dictated by a parameter called topology update delay. This value determines the maximum delay to wait before sending a dedicated topology update to the sink unless an update with piggybacking has not already been sent. This parameter can be seen as the maximum time between a node changing parent and the sink being informed about it, without considering lost messages due to other concurrent topology changes or collisions.

The protocol uses both broadcast and unicast communication primitives:

- Broadcast: is used to perform the flooding algorithm for topology reconstruction
- Unicast: is used to perform communication from nodes to sink and from sink to nodes with source routing

In unicast there are two different packets, for this reason each packet has an identifier in the header, used to determine its type.

To communicate with the sink, nodes simply send a packet towards their parent, which in turn will forward to their parent and so on until the sink is reached.

The sink keeps an updated routing table in which the parent of each node is known. To send a message to the specific node, the route is calculated and embedded in the header. When receiving a source route packet, nodes will pop the first entry of the routing list in the header and send it in unicast towards that entry and so on, until the desired node is reached.

Implementation
In order to simplify the development of the protocol, some constructs have been designed. In the following paragraph their structure and purpose are explained.

The routing table has been encapsulated in a struct that takes care of the basic primitives: *add, update, remove, allocate, free.* The source code of the routing table is defined in the header `routing-table.h` and in the source file `routing-table.c`. Abstracting the functionality of the routing table from the protocol allows to perform tests and spot bugs more easily. The routing table is allocated in the sink only, as the nodes only required the parent to be stored.

To simplify various tasks such as the dynamic allocation of the header and the automatic calculation of offsets, another struct called buffer has been created to encapsulate primitives such as: *allocate (read,write), free, read, write.* The source code of the buffer is defined in `buffer.h` and `buffer.c`.

Topology updates
Upon changing the parent during the topology reconstruction protocol, the node schedules a function to be run after a certain time τ defined as:

$$\tau = T_{ud} + \gamma$$

Where T_{ud} is the topology update delay and $\gamma \in [0, 1]$ is a random factor expressed in seconds and used to introduce stochasticity in the communication protocol with the aim of reducing collisions.

Whenever a message is sent towards the sink, the node will check if the topology is set to dirty and, if no topology update has been piggybacked in a previous message, the header containing the information of the new parent is embedded in the message. At this point, the dirty flag is set to `false` and the flag indicating that the notification has been set during this topology epoch is set to `true`.

When τ time has passed and the callback is called, if the topology is dirty and no update message has been piggybacked, a dedicated update is sent, otherwise the callback simply returns.

This small protocol ensures that exactly one update is sent to the sink every τ time.

However, it is important to keep in mind that lost topology messages caused by collisions or concurrent topology updates are not taken into consideration.

Performance evaluation

The performance evaluation of the protocol has been carried out in different scenarios and with different parameters. In all experiments, the MAC layer used is the `csma_mac` defined in ContikiOS.

Parameters table

Beacon period	T_B
Source route msg period (one-to-many)	T_{S_r}
Node msg period (many-to-one)	T_m
Topology update delay	T_{ud}
Percentage of piggyback updates	U_p
Percentage of dedicated updates	U_d

Cooja
In Cooja, the expriments have been tested with various mac layers and parameters to validate the protocol in every situation.

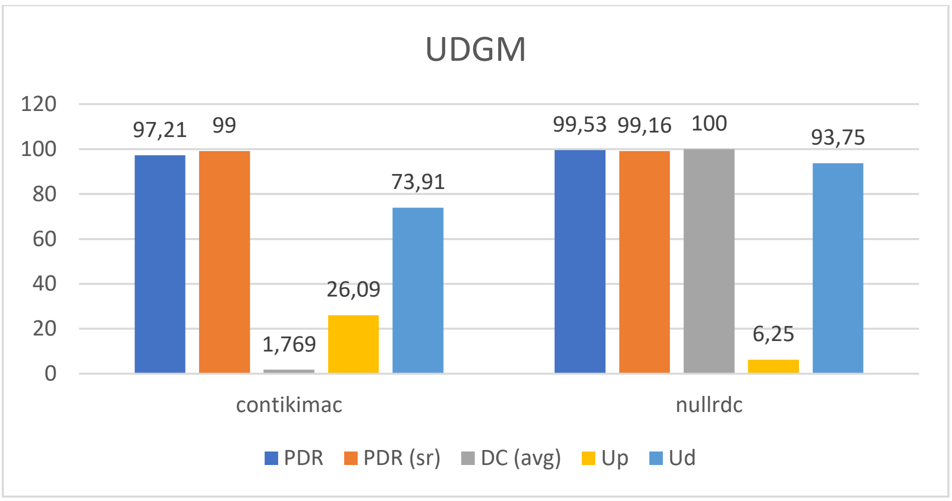
T_{S_r}	15 s
T_m	30 s

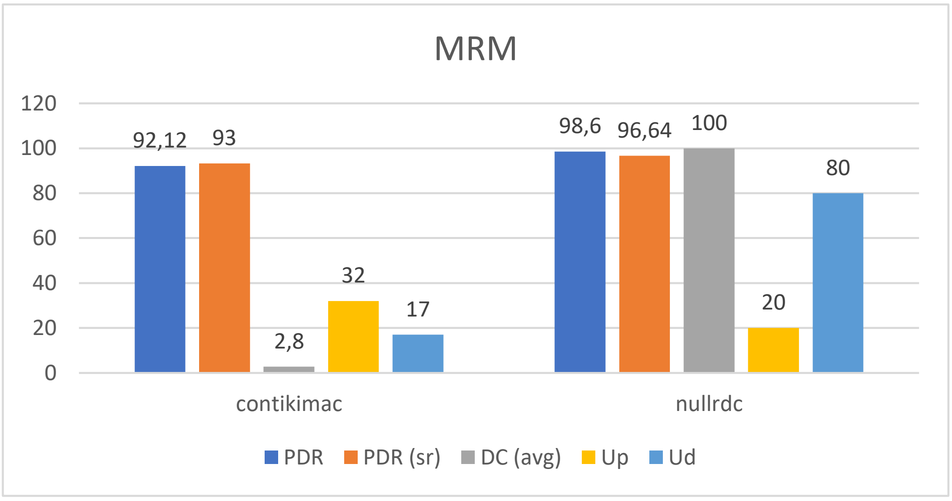
RDC: `nullrdc`, `contikimac`

Radio medium: `MRM`, `UDGM`

Simulations are run with a simulated time of `1800000 ms` that is, `30 minutes`.

T_B	30 s
T_{ud}	15 s





Dynamic environment: mobile sink
RDC: `contikimac`

Radio medium: `MRM`

In this custom configuration, the Cooja's test script has been edited to introduce changes in the network topology. The topology selected is changed at runtime: the sink is moved in a circular trajectory every $n = 64$ exchanged messages (steps). Figure 1 represents the configuration of the topology and the trajectory of the sink. The selected topology is very interesting as at each time the graph is connected, preventing communication errors due to isolated nodes.

The trajectory of the sink allows the nodes to exploit the protocol at its best: nodes dynamically adjust their parent based on the hop count and on the RSSI value. In case of draw in the hop count, the parent corresponding to a higher RSSI value is selected.

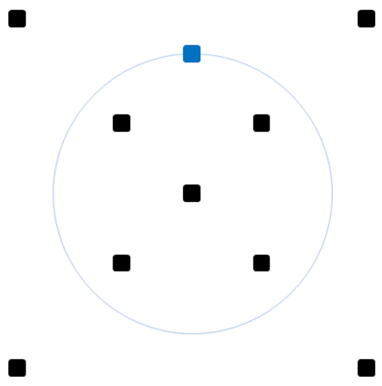
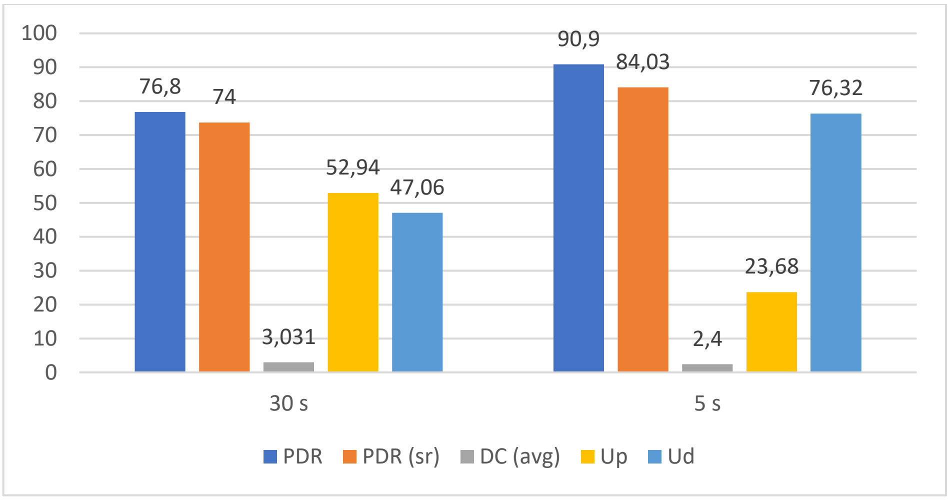
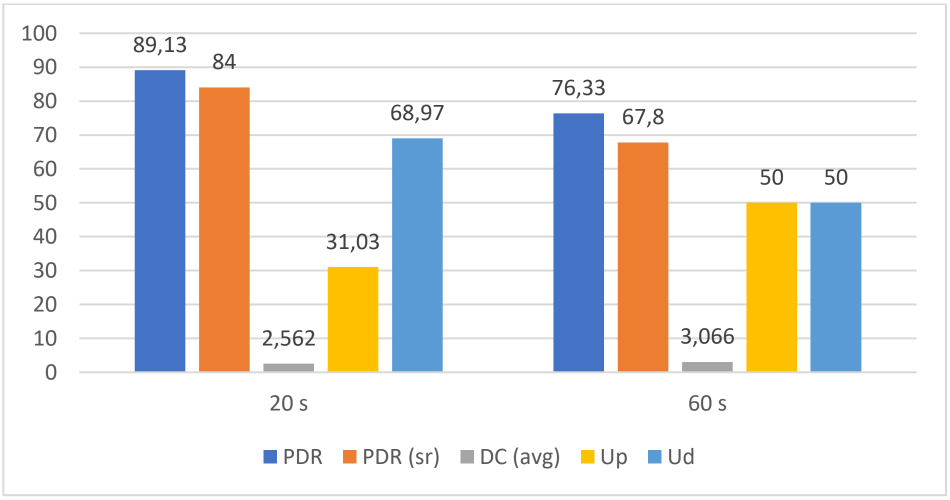


Figure 1: nodes are black, sink is blue, the circumference highlights the sink's trajectory

In the dynamic environment, benchmarks are run with different values for T_{ud} and T_B to highlight the influence on the performance of the protocol. The following graphics show the various results.



$T_{ud}, T_B = 30s$

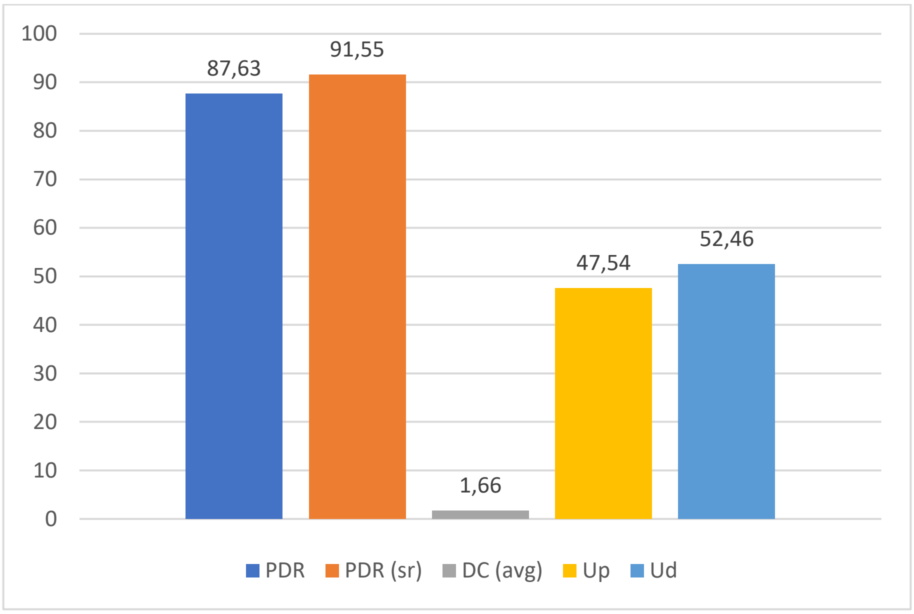


$T_B, T_{ud} = \frac{T_B}{2} s$

CLOVES
In the CLOVES testbed, experiment uses the `contikimac` RDC layer and is run for a total of `20 minutes`. The following results have been achieved by using all the nodes in the `DEPT` area (`disi_povo2`).

Parameters have been slightly adjusted to cope with the large topology. The table below shows the selected values.

T_B	90 s
T_{ud}	30 s
T_{S_r}	15 s
T_m	60 s



Conclusions
Experiments show deep correlations between mainly two parameters: the beacon delay T_B and the topology update delay T_{ud}.

In dynamic environments such as the custom Cooja simulation, decreasing the beacon delay, allows to respond quicker to topology changes, resulting overall in a better performance. However, decreasing it too much, can cause more collisions as the flooding algorithm is performed more times each epoch.

The topology update delay T_{ud} also plays an important role as it indirectly determines the main modality in which topology updates are propagated to the sink.

Increasing the T_{ud} will cause most topology updates to be sent by piggybacking information into messages headers while decreasing it enforces the protocol to send dedicated topology updates more frequently. The optimal value for this parameter is greatly influenced by the application purpose. In an application in which messages toward the sink are exchanged very frequently, increasing the value can result in most of the updates being propagated using piggybacking, reducing the overall traffic. If an application instead does not exchange messages frequently, piggybacking topology information into messages headers could not be enough, in this case a smaller value can ensure a higher responsiveness to topology changes.

Simulations run on the CLOVES testbed highlight how real environment factors influence the efficiency of the protocol. Even though nodes are static, the protocol experiences numerous topology updates, mainly because of the fluctuations of RSSI values. Another factor to take into consideration is the number of nodes involved in the experiment. The simulation was run on the `disi_povo2` area which is composed of 36 nodes. The topology reconstruction flooding algorithm causes a lot of messages to be exchanged and requires more time to complete, inevitably causing more collisions.

Problems

One of the main problems of the current protocol is the choice of the primary routing metric.

Using the hop count as the primary metric is not always the best choice as it implicitly forces nodes to select the shortest path which, in many cases, is the weakest. In fact, each node will tend to select the route that reaches the sink with less hops, hence using longer connections. Although using the RSSI both as threshold and as parameter to select different routes of equal length increases the performance of the protocol, it is not enough.

Possible solutions would be to use other, more complicated, routing metrics such as the Link Quality Indicator and the Expected Transmissions Count.

Another problem is how the RSSI is used. The real-world simulation on the CLOVES testbed highlights how the RSSI value can fluctuate a lot, resulting in a lot of topology changes and, consequently, in more traffic.

A possible solution could be using a moving average of the RSSI value. This solution allows to filter out high frequency fluctuations and use a smoothed value to decide which parent to select. On the other hand, it would force each node to keep a dynamic growing list of RSSI values to update the moving average of each neighbor, resulting in a larger memory footprint.