Time Synchronization and Periodic Data Collection

Low-power Wireless Networking for the Internet of Thing Course Project

Riccardo Scilla - 213723
University of Trento
riccardo.scilla@studenti.unitn.it

ABSTRACT

This project is about the implementation of a protocol for periodic data collection. It is based on a schedule that is built to the reduce the interference between nodes. Results are shown for both simulation and testbed experiments to assess the performance of the implementation. It is also quantitatively compared with a baseline protocol based on transmission with random delays.

1 INTRODUCTION

The aim of this project is to build a data collection protocol for many-to-one routing, also called Convergecast. We consider the case of a multi-hop wireless network in which we are able to send data from any node to a common destination (the sink) which can perform analysis over it.
Data Collection is a very important communication paradigm in WSN and a core building block for many applications. The idea for a multi-hop data collection protocol is to have a network composed by many nodes and a single sink. In this situation, not all the nodes are connected to the sink directly, so they will forward their data to other nodes and eventually reach the sink. To do this, nodes need to know how to build the routes and how to adapt them to network changes. This is done with Routing, in this case many-to-one, for multi-hop delivery, which means that many nodes are involved in a single route. A natural and efficient way to implement collection is by using a tree topology rooted at one collection points, the sink. The tree is built considering a cost metric, which allow to find, for every node in the network, the minimum cost path to the root.

The protocol described in this project aims to perform data collection using scheduled transmissions, which means that once all the nodes are synchronized with the sink and the tree is built, they will be able to transmit their data to the sink, following the route, in a shorter time window and without interference.

The report is structured as follow: first the protocol is described, defining each phase in the epoch that allow to perform synchronization and data collection. Then follows a description of the implementation choices. The performance evaluation is done both in simulation in Cooja and in testbed and the results are shown; the protocol is also compared with a baseline one.

2 PROTOCOL DESCRIPTION

The protocol follows periodic structure, defining an *epoch*. Within each epoch four phases are handled:

1. **Synchronization phase**: The topology is rebuilt and all the nodes in the network are time synchronized.
2. **Data Collection phase**: All nodes in the network transmit their packet to the sink or are listening to forward the packet or their children nodes.
3. **Radio OFF**: All nodes switch off the radio until the beginning of the next epoch, saving energy.
4. **Guard phase**: Nodes wake up a bit earlier the beginning of the next epoch to account for time synch misalignment.

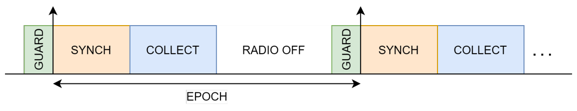

Figure 1: Epoch structure of the collection protocol

The epoch structure with each phase is shown in Figure 1.

2.1 Synchronization phase

During this phase the radio of all nodes is always ON.

The first part of this phase is building the topology. As previously introduced, the routes are built based on a minimum-cost spanning tree, where the path cost is identified by a routing metric which reflect the effort required to deliver packets along the path. A common routing metric is the hop count, which is the number of transmissions that nodes should perform in order to deliver a packet to the sink. In this way the cost path express the its length in terms of hop and therefore the longer the path, the higher the cost. Usually this metric is not enough for wireless because not always the shortest route is the best route. For this reason, the metric is enhanced by filtering very bad links considering a RSSI threshold. This helps to improve the reliability of the links: if we get a low RSSI, is very likely that many packets are lost, thus that link is discarded.

The topology used is a tree where the sink is considered as the root. To build it, the sink initially sends a broadcast beacon; nodes within communication range receive it and schedule a new beacon after a random delay. This is received by farther nodes which again schedule a new beacon after a random delay. The beacons sent in this process include the metric of the sender (e.g. 0 for the sink, 1 for the 1-hop nodes, etc.) which allow to select as parent the closest one, and a sequence_number, which is send periodically by the root at every epoch and the nodes must accept its changes without considering the metric in the beacon.

The second part of this phase is about network synchronization. The goal is to make every node aware on the time the epoch current started. As said before, every time a node receives a routing beacon, it schedules a new transmission after a random delay. Nodes embed in their packet the information about the accumulated delay, which is the total delay from the transmission of the first packet by the sink. Considering the case of node i receives the beacon at the reception timestamp Ti_{RX} and the accumulated delay embedded in the payload is Di, it can get the timestamp of the beginning of the epoch Ti_{epoch} as:

$$Ti_{epoch} = Ti_{RX} - Di$$

In this way all the nodes are synchronized because the reception is not affected by the accumulated delay. Some misalignment may still occur due to different processing delays or time of flight of the beacons.

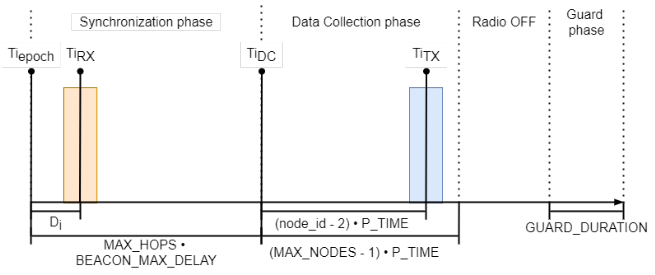

Figure 2: Node epoch phases scheduling

2.2 Data Collection phase

Once all the nodes are synchronized, the Data Collection can start at a "known" time t after the transmission of the first sink beacon. The time t is computed considering the maximum number of hops of the network and the maximum random delay in the Synchronization phase: given that the network has MAX_HOPS and $BEACON\_MAX\_DELAY$ we know that, in the worst case, the farthest node will receive the routing beacon after $MAX\_HOPS \cdot BEACON\_MAX\_DELAY$. So the beginning of the Data Collection Ti_{DC} is computed as:

$$Ti_{DC} = Ti_{epoch} + MAX\_HOPS \cdot BEACON\_MAX\_DELAY$$

In Data Collection phase, nodes schedule the transmission of their packet based on their node ID and the expected time for a packet from the farthest hop to reach the sink. In fact, nodes 1 hop away to the sink can transmit directly to it but nodes that are more hops away need to transmit to their parent first, which means that the farther they are, the longer it takes for their packets to reach the sink. This needs to be taken into account because we want that the data from the different nodes reaches the sink in an ordered way so we must allocate enough time to each packet to reach the sink. If we consider P_TIME the maximum time interval for a packet to reach the sink, MAX_HOPS the maximum number of hops in the network and P the processing time per hop we get that:

$$P\_TIME = MAX\_HOPS \cdot P$$

and the node with ID $node\_id$ will transmit at time Ti_{TX}:

$$Ti_{TX} = Ti_{DC} + (node\_id - 2) \cdot P\_TIME$$

All the timestamps computation are explained in Figure 2.

2.3 Radio OFF

After the collection, turn off the radio to reduce energy consumption. During this phase the application could tell some nodes to schedule a packet. This will not be send directly, because all the radios are OFF, but stored in a buffer and scheduled during the next Data Collection phase.

2.4 Guard phase

Nodes are woken up slightly before the beginning of the new epoch to take into account some possible synchronization misalignment of the nodes. The period duration is fixed and set as $GUARD\_DURATION$. During this phase, nodes can still receive routing beacon of the next epoch and in such situation, the nodes switch to the Synchronization phase and wait for the Data Collection phase.

3 IMPLEMENTATION

To implement the project a mixed programming style is used, including processes and function callbacks: in particular processes are used to schedule the phases of the protocol and to notify the reception of a synchronization beacon. Instead function callbacks are used to schedule the flooding of the beacons in the synchronization phase.

The code is structured in 3 main files: application, sched_collect and sched_beacon. The different operations done in each file are briefly described in the following subsections.

3.0.1 ***application***.

A process defines different operations if working with the sink or a normal node:

- sink: wait for all the nodes to bootstrap;
  call *sched_collect_open* to open the connection with sink flag.
- node: call *sched_collect_open* to open the connection;
  every epoch call *sched_collect_send* to set the data packets to be sent to the sink.

Defines a callback function *recv_cb* that is called to inform the application that a packets is received at the sink.

3.0.2 ***sched_collect***.

Defines the core functions for Data Collection protocol.

- *sched_collect_open*: Initialize the connector structure with the information of metric, parent, seqn and callback.
  Open connections broadcast (for beacons) and unicast (for collection).
  Start different processes if the node is the sink or not.
  - sink: set the metric to 0 and set a ctimer with the callback *beacon_timer_cb* to periodically transmit the synchronization beacon.
  - node: manage the phases of each epoch. Allocate a custom process event *BEACON_ARRIVED* that notify when a beacon arrives.
    * synch_phase: wait for an event if the beacon didn't arrive in the previous guard_phase. Compute the time that remain before entering the collect_phase and set an etimer.
    * collect_phase: wait until it's node time to transmit to the sink. Send the message inside the buffer calling *send_msg*. Wait until the collect_phase is over.
    * radio_OFF: turn off the radio to for the *RADIO_OFF* time to save energy.
    * guard_phase: wait for the entire duration. If it get a *BEACON_ARRIVED* event, set a flag to true for the next synch_phase.
- *sched_collect_send*: Store packet in a local buffer to be sent during the data collection time window. If the packet cannot be stored, e.g., because there is a pending packet to be sent, return zero. Otherwise, return non-zero to report operation success.
- *send_msg*: Copy the buffer in the packetbuf. Create a header with the node address and the hop count, allocate space for it in the packetbuf and copy it. Send the packet to the parent using unicast.
- *uc_recv*: Node receives the unicast packet from a child. Copy the header of the packet. If the node is the sink, remove the header and call the *recv_cb* callback in application. Otherwise, increment the hop count and send the packet to the parent using unicast.

Name	Description	Value
EPOCH_DURATION	Duration of an Epoch.	30 · CLOCK_SECOND
BEACON_INTERVAL	Time before to flood a new beacon by the sink to rebuild the topology.	30 · CLOCK_SECOND
BEACON_MAX_DELAY	Maximum delay to schedule a beacon by a node.	CLOCK_SECOND/N
SYNCH_DURATION	Total duration of the Synchronization phase.	MAX_HOPS · (BEACON_MAX_DELAY+30)
SCHED_COLLECT	Schedule the collect phase for a node.	(node_id-2) · P · (MAX_HOPS)
COLLECT_DURATION	Total duration of the Data Collection phase.	(MAX_NODES-1) · P · (MAX_HOPS)
GUARD_DURATION	Total duration of the Guard phase.	30
RADIO_OFF	Total duration of the Radio OFF (remaining time in the Epoch).	EPOCH_DURATION-SYNCH_DURATION-COLLECT_DURATION-GUARD_DURATION

Table 1: Time and duration to schedule the protocol

3.0.3 ***sched_beacon*.** Defines the functions for flood the network and synchronize using beacon delay.

- *beacon_timer_cb*: call *send_beacon* to broadcast the beacon. If it is the sink, increment the seqn and wait for the next *BEACON_INTERVAL*.
- *send_beacon*: create a beacon message with the seqn, the metric and the accumulated delay so far. Copy it in the packetbuf and send it broadcast.
- *bc_recv*: get the beacon. If the RSSI of the message is over a threshold, check the condition for the update of the metric:
  - if the beacon metric improve the current metric more than 1, or if receive a new seqn, without checking the metric, then:
    - update the seqn, the metric, the parent and sample a delay in $[0, BEACON\_MAX\_DELAY - 1]$;
    - save the current reception time; compute the accumulated delay as the sum the received delay and the sampled delay;
    - compute the begin of the epoch as the difference of the current reception time and the received delay;
    - post a *BEACON_ARRIVED* event to notify the synch_phase or guard_phase in *sched_collect*;
    - Wait the sampled delay and broadcast the synchronization beacon.

All the times and duration used to schedule the protocol are described in Table 1. The two main parameters are N and P:

(1) N tunes the max delay that can be sampled to forward the beacon during the Synch phase; the higher this value, the lowest the range so lower the duration of this phase (affect *SYNCH_DURATION*) but also the higher the probability that two nodes samples the same value, possibly colliding.
(2) P tunes the start time of the Collect phase as described in the Section 2.2. The lower the value, the fewer time nodes have to forward the packet to the sink; it should be tuned in a way that also nodes multiple hops away have the time to reach the sink without intersecting with the collect phase of nearby nodes.

The *SYNCH_DURATION* value is slightly increased for every hop: this is due to the fact that there is a difference between the time each node schedules the transmission of the beacon and the actual transmission, so that the Synch phase would terminate before the reception of the beacon for further nodes, breaking the protocol. This effect can be easily seen setting higher values of N, because all nodes will sample similar random delays.

4 EXPERIMENTAL EVALUATION

4.1 Tests

To measure the performance of the protocol different tests are performed in simulations with Cooja changing the topology, the configuration settings and the radio models.
The protocol is tested first using network topology consisting of 9 nodes with max_hops 3. The topology is shown in Figure 3. Here the radio model used is Unit Disk Graph Model (UDGM) with distance Loss. With this setting are run 3 Tests for 30 mins with random seed and the results are grouped.

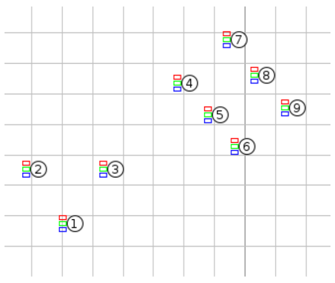

Figure 3: udgm distLoss Topology

The second test performed is changing the radio medium, in this case an enhanced radio model Multi-ray Radio Model (MRM) is used to provide more realistic simulations. In this case, the network topology consists of 10 nodes with max_hops 4. The topology is shown in Figure 4. With this setting are run 3 Tests for 30 mins with random seed and the results are grouped.

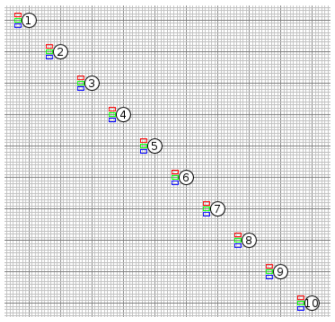

Figure 4: mrm Topology

A comparison between the main schedule protocol and another protocol, here called collect_protocol, in which:

- nodes find their parent in a similar way described before;
- the data collection phase is not scheduled but the packets are sent just after a random delay.

In the collect_protocol, the Radio Duty-Cycle used is ContikiMAC, in contrast to the nullRDC used in the main protocol of this work. The comparison is done using the topology of Figure 3 with the udgm distLoss radio model.

The protocol is tested also in a real Testbed to understand the effect of the environment on this solutions. The Testbed used is the one provided in the DISI floor of the University of Trento, shown in Figure 5: it consists of 36 nodes with max_hops 4.
A comparison with the ContikiMAC is also done in this settings.

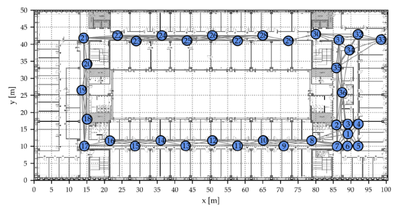

Figure 5: DISI Topology

4.2 Results

4.2.1 ***udgm distLoss.*** The results obtained in the first test are described in Table 2 and Plot 7.

N values	P value	PDR (%)	DC (%)
10	3	63.29 ± 35.94	1.38 ± 0.02
	5	78.95 ± 27.25	1.53 ± 0.01
	7	84.41 ± 20.78	1.68 ± 0.00
	10	99.64 ± 0.70	1.91 ± 0.00
	13	100.0 ± 0.0	2.15 ± 0.00
	15	99.85 ± 0.48	2.30 ± 0.00
	17	100.0 ± 0.0	2.46 ± 0.00
	20	100.0 ± 0.0	2.70 ± 0.00
20	3	64.80 ± 32.84	1.07 ± 0.02
	5	71.26 ± 32.88	1.22 ± 0.01
	7	88.86 ± 16.17	1.37 ± 0.00
	10	99.86 ± 0.48	1.61 ± 0.00
	13	100.0 ± 0.0	1.84 ± 0.00
	15	100.0 ± 0.0	2.00 ± 0.00
	17	100.0 ± 0.0	2.16 ± 0.00
	20	100.0 ± 0.0	2.39 ± 0.00
50	3	59.91 ± 35.44	0.86 ± 0.02
	5	73.06 ± 30.34	1.01 ± 0.01
	7	86.21 ± 19.54	1.17 ± 0.00
	10	100.0 ± 0.0	1.40 ± 0.00
	13	100.0 ± 0.0	1.63 ± 0.00
	15	100.0 ± 0.0	1.79 ± 0.00
	17	100.0 ± 0.0	1.95 ± 0.00
	20	100.0 ± 0.0	2.19 ± 0.00

Table 2: Schedule Protocol Results - udgm distLoss PDR and DC per N and P values

As we can see, increasing the N value has a strong impact on the Duty Cycle for every value P chosen; looking at the PDR, it follows the same trend for every for every value P. Increasing the P value has an impact on PDR, that we can see it reaches lower values for low Ps, because the process time required for further nodes to reach the sink is higher than the one provided by the P.

This can be better seen in Figure 6 where we can see that with P=3, the nodes with id 2 and 3, that are 1-hop from the sink, receive all the packets with a good reliability. This value is lower by a 20% for the nodes 4 5 and 6 that are 2-hops from the sink and even lower for nodes 7 8 and 9 that are 3-hops from the sink. On equal hop distance, nodes with lower id have priority so they have a slightly higher probability to deliver the packet successfully.
Increasing the P value from 3 to 7, the time to reach the sink is enough for 2-hops distance node, so the PDR of node 4 5 and 6 reach good values.

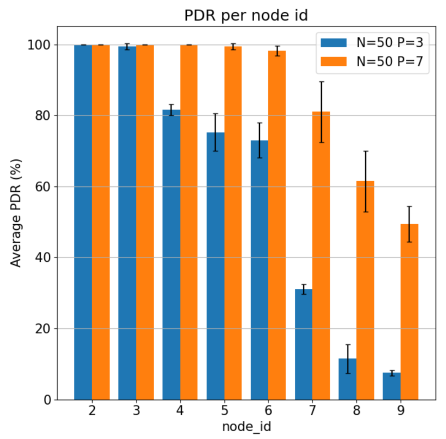

Figure 6: PDR per node_id

4.2.2 ***mrm.*** The results obtained testing the mrm radio model are described in Table 3 and Plot 8. We can see that it follows the same trend as the previous test: the PDR is comparable to the one with udgm radio model, while the DC is increased because of the different topology that contains more nodes and hops.

N values	P value	PDR (%)	DC (%)
10	3	77.39 ± 34.37	2.57 ± 0.01
	7	95.40 ± 11.86	3.16 ± 0.00
	10	99.74 ± 0.70	3.60 ± 0.00
	15	100.0 ± 0.0	4.34 ± 0.00
20	3	76.37 ± 33.66	1.90 ± 0.01
	7	95.02 ± 14.16	2.48 ± 0.00
	10	99.68 ± 1.05	2.93 ± 0.00
	15	99.94 ± 0.33	3.50 ± 0.00
50	3	77.52 ± 34.12	1.39 ± 0.01
	7	93.36 ± 18.90	1.98 ± 0.00
	10	99.04 ± 2.63	2.42 ± 0.00
	15	100.0 ± 0.0	3.15 ± 0.00

Table 3: Schedule Protocol Results - mrm PDR and DC per N and P values

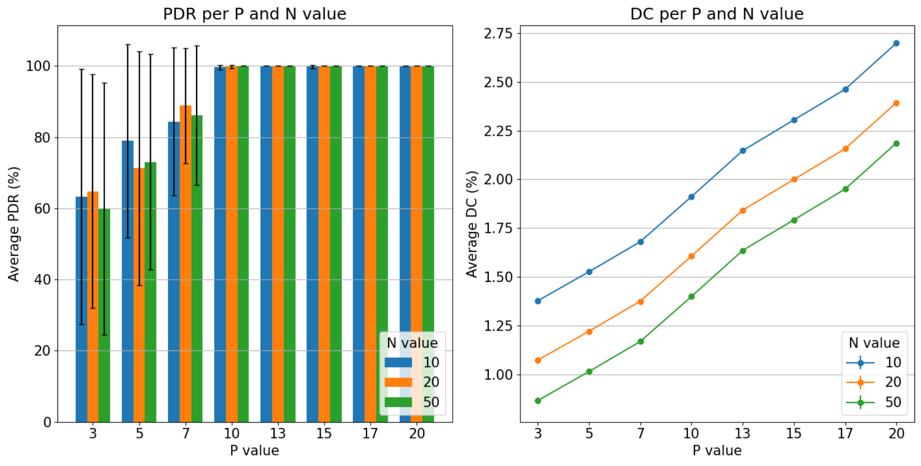

Figure 7: Schedule Protocol Plot - udgm distLoss - PDR and DC per P and N values

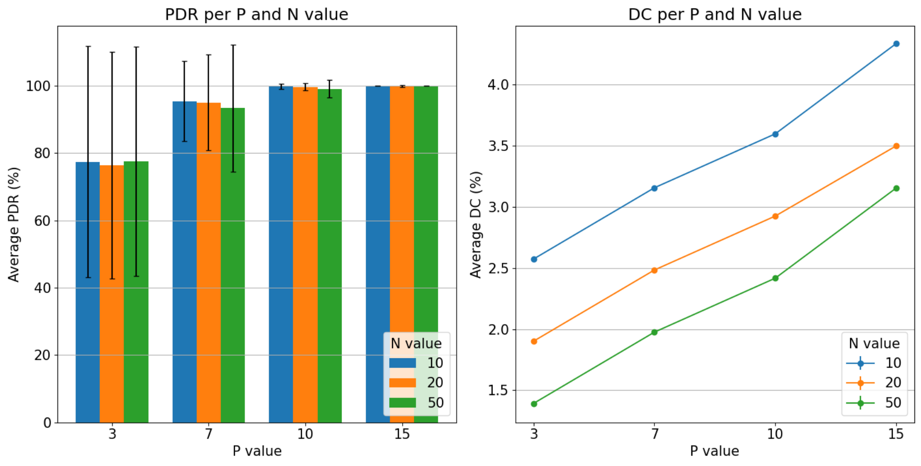

Figure 8: Schedule Protocol Plot - mrm - PDR and DC per P and N values

4.2.3 ***Collect_protocol***. The results obtained testing the collect_protocol are described in Table 4 and Plot 9. Comparing the nullRDC and the ContikiMAC we can see that the DC reaches reasonable values for ContikiMAC, while it is of course 100% for nullRDC. The collect_protocol is tested also increasing packet loss up to 80% depending on the distance. This has a strong effect on the PDR for the nullRDC, which is halved, while it is just few percentage points lower for the ContikiMAC.

Mode	PDR (%)	DC (%)
nullRDC	99.47 ± 0.94	99.99 ± 0.00
nullRDC_rx20	46.95 ± 21.46	100.0 ± 0.0
ContikiMAC	98.73 ± 1.62	1.53 ± 0.57
ContikiMAC_rx20	96.40 ± 3.35	1.01 ± 1.01

Table 4: Collect Protocol Results - udgm distLoss PDR and DC per RDC

Comparing the main protocol with the ContikiMAC, using the setting N=50 P=10 we can reach better performance both in PDR and DC.

4.2.4 ***Testbed***. The results obtained int the Testbed are described in Table 5 and Plot 10. As we can see the protocol reaches perfect results in PDR, even though these values come from a single test. As we found in the other tests in Cooja, the Duty-Cycle increases by incrementing the P value and also is higher overall because of the higher number of nodes in the topology.

N value	P value	PDR (%)	DC (%)
50	5	100.0 ± 0.0	3.04 ± 0.00
	10	100.0 ± 0.0	5.31 ± 0.00
	15	100.0 ± 0.0	7.59 ± 0.00
ContikiMAC		99.70 ± 1.22	3.04 ± 0.01

Table 5: Testbed Results PDR and DC per P value and ContikiMAC

The collect_protocol with ContikiMAC RDC is also tested in the the Testbed. For the results we can see that the DC is similar to the one with P=5 but the PDR is lower because few nodes (ids 23 27) get only 95% of packets.

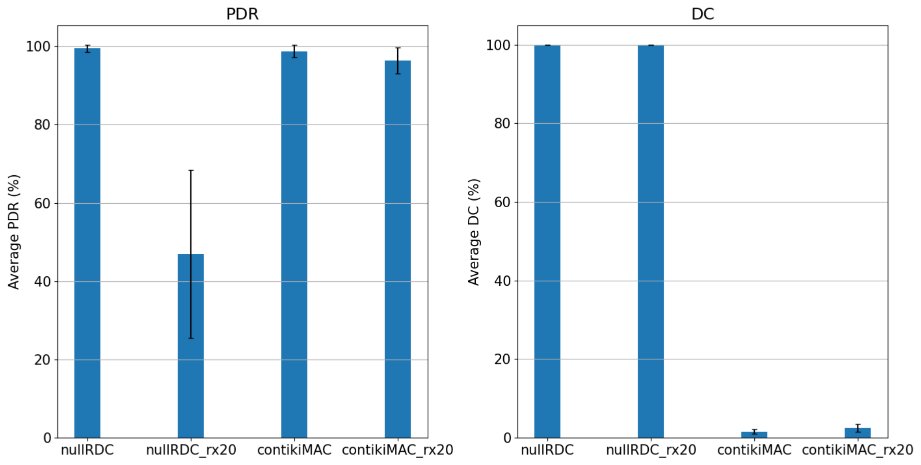

Figure 9: Collect Protocol Plots - udgm distLoss - PDR and DC per RDC

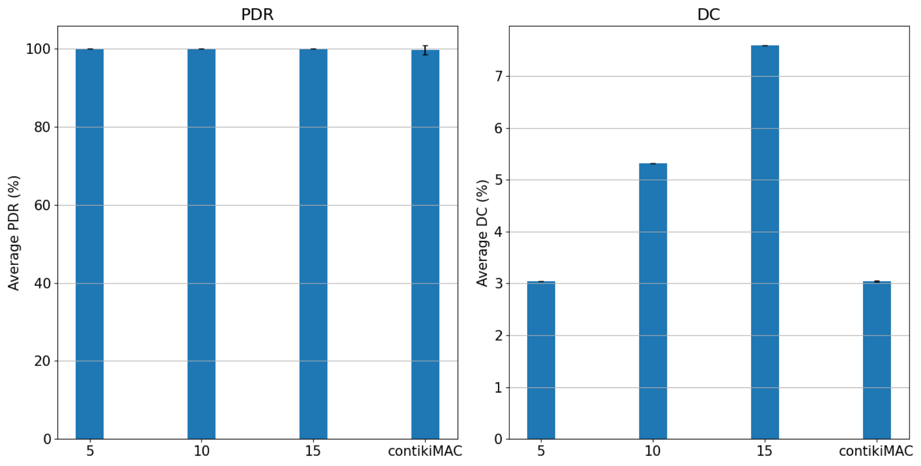

Figure 10: Testbed Plots
PDR and DC per P value and ContikiMAC

5 CONCLUSION

The aim of this work was to implement a protocol for periodic data collection in an IoT scenario. This is achieved having a first synchronization phase to allow nodes discovering their parents in the network, followed by a data collection phases that schedules the time period for which each node can talk. Simulations in Cooja and in a real testbed are performed together with a comparison with a simpler protocol. The results obtained in this experiments are explained and discussed.