Low power wireless networking for the Internet of Things

Project report
2021/2022

Nicola Arpino
223025

UNIVERSITÀ DI TRENTO
Department of Information Engineering and Computer Science

Contents

1 Introduction

In this project we were asked to develop a protocol to be used, in the context of WSN, for *event-triggered control* capable of reacting to events and perform control actions. In particular the core phases of the protocol are the data collection and commands dissemination. The solution proposed is based on the Lab 6-7 projects and has been refined multiple times in order to reach the expected performances.

2 Tree construction

The tree construction phase needs to create a tree rooted on the controller node. Such a tree is created periodically via a dedicated broadcast connection `tree_bc_conn`. The controller starts sending in broadcast a beacon message containing the *hop count* and the *sequence number*. When a node receives the beacon it can analyze the RSSI, the sequence number and the metric in order to decide how to handle such information. In case the message signal is not too weak or too old a comparison logic starts. Each node keeps up to two parents (because of reliability requirements), the first one is the *best candidate* and the second one could be defined as the *reserve*. The best candidate ideally is the one that has the strongest signal and the lowest distance (in terms of hops). The protocol starts with nodes having no parents at all so it makes sure to have at least two parents per node.

1. **Is the packet not weak or old?**

   If so the packet can be analyzed. In case the sequence number is greater than the current known by the node, then the beacon sender becomes the best candidate.

2. **Does the node have two non undefined parents?**

   In case there is no parent saved it is the same situation as the above one. In case there is at least one parent saved, the sender of the packet is compared with the saved parents and the appropriate one is chosen to be *"updated"*. The evaluation, as said before, is based on the RSSI and the metric. The sender must have better RSSI or a better metric. In any case if at least one property is better than the one of the actual candidates, the other property must be equal or greater. For example, a new candidate `[address: 06:00, rssi -90 Db, distance 4]` can replace the best candidate `[address 05:00, rssi -91 Db, distance 4]`. Moreover the protocol ensures that the two candidate parents are not the same.

There are some other minor points that are relevant to discuss. The first one is about the *switch* of parents: in case a new candidate parent is better than the actual best candidate, the actual best becomes the reserve one. Another important aspect to state is that both patents will be chosen so that they are at the same distance (metric). This might be not always possible due to the network real topology. It implies that the tree made up by all best candidate parents is a valid tree meanwhile if a node uses the reserve parent, messages might loop. Since all nodes must have two parents this situation may happen; proper loop management has been enforced.

In short after the tree construction each node has up to two different parents chosen among the senders of all the beacon messages that the node receives.

3 Event dissemination

This phase is used by for sending the event message: it notifies that system is perturbed in a broadcast way. This phase uses a dedicated broadcast connection `event_bc_conn`. Each sensor periodically perform new readings and if the triggering condition holds then the application calls the `etc_trigger` function. The sensor should start an EVENT flood network-wide.
The event flood technique is rather simple. Each node receives an EVENT message and, as soon as it has been received, the messages is broadcasted again. The event dissemination requires to address two challenges :

1. The Event flood must be received by all nodes
2. Nodes must take advantage of timers to reduce contention and avoid triggering events in loop

3.1 Event flood

The first point has been addressed using *retransmissions* as reliability mechanism: using the `sent` callback of the broadcast primitive, each node that fails to transmit can repeat the operation after some delay. My initial attempt was to emulate the Glossy [1] protocol by retransmitting all packets at least two times. This turned out to produce bigger amount of traffic w.r.t the one used by the selective resend of faulty messages. Another mechanism that I tried to employ was to use the *constructive interference* by sending two messages after a very short amount of time. The problem with this approach is that to perform well requires almost an always on radio [2], which, for the purpose of the project, cannot be utilized.

3.2 Contention avoidance

As suggested by the project description document, there are two different events to manage: the **self** and the **propagation** events. Self events are the ones produced by the node and must be suppressed for a while after the node has been triggered. In this way the node avoids to trigger multiple times and go into a loop. The propagation events are the ones received by other nodes that must be forwarded. The solution was to employ two different *ctimers* `suppression_timer` and `suppression_prop_timer` that were set up so that the first timer has a longer interval w.r.t the second one.

3.3 Event reception

The event reception makes each sensor to forward in broadcast the very same event. Moreover the node prepares itself to perform the data collection phase, scheduling such phase with a timer. After some attempts and performance analysis I come up with a timer delay calculated as follows.

$$random\_rand()\%(CLOCK\_SECOND) + (CLOCK\_SECOND/2) \tag{1}$$

[1]http://www.olgasaukh.com/paper/ferrari11glossy.pdf
[2]https://ora.uniurb.it/retrieve/handle/11576/2664014/135988/FINAL%20VERSION.PDF

4 Data collection

The data collection phase objective is, from the controller perspective, to collect all the sensors data. Such a data should contain some relevant information, typically the `sensor_value` and the `sensor_threshold`. The messages might traverse many nodes in a multi-hop way taking advantage of the tree built in the previous phase. Before looking into some relevant design choices, let's have a look to the collect message sent by sensors.

4.1 Data collection packet

```

// Structure for additional collection info
struct sensor_data {
  uint32_t value;
  uint32_t threshold;
  linkaddr_t sensor_addr;
}

// Structure for data collection packets
struct collect_msg_t {
  linkaddr_t event_source;
  uint16_t event_seqn;
  struct sensor_data s_data;
  uint16_t retransmitted;
}
```

Listing 1: Collect data messages

The structure `collect_msg_t` represents the collection message sent by a sensor. It includes the `event_source` and the `event_seqn`: they identifies the current event that requested for sensors data. The additional structure `sensor_data` includes the sensed data and the address of the sensor sending this message. In this way, no matter which node of the tree is relaying such message, there is precise knowledge about the originator. The `retransmitted` attribute is used for reliability mechanism discussed later.

4.2 Packet transmission

In order to send packets towards the controller, dedicated unicast connection has been adopted. In particular the *runicast* connection `data_collection_conn` is used. As well as holds for the other connections, it has been opened in a separate virtual channel.
The *runicast* was employed in order to take advantage of the retransmission feature.

4.2.1 Queue based forwoarding

The general idea of forwoarding is based on one of the major challenges that this phase required to face. Since a sensor/forwoarder node can be part of the path to the sink for multiple sensors, it might be requested to forwoard many packets at the same time. The idea to face this scenario was to employ a queue based mechanism implemented using the `LIST` and `MEMB` primitives available in ContikiOs. Each node has a queue implemented with a dynamic linked list. When a sensor is triggered to send the EVENT message, it also schedules itself to send a COLLECT message after some time. In that way the EVENT message and COLLECT ones are not likely to collide. Also the sensor has a single unit queue. The sending process is so managed by two different functions that are in charge of adding elements to the queue and sending elements of the queue.

```
struct collect_msg_list_t {
  struct collect_msg_list_t* next;
  struct collect_msg_t collect_message;
};

/* Buffer per node of collect messages to be delivered */
LIST(collect_to_send); /* The queue name */
MEMB(collect_mem, struct collect_msg_list_t, number_of_nodes);
```

Listing 2: Queue

4.2.2 Pushing and sending

The function `data_collection_trigger` is in charge of allocating the memory for a single element of the queue (using MEMB) and pushing the collect message to the bottom of the queue using `list_add` primitive. Once the operation is done it schedules the actual sending after some short delay via the `data_collection_send`. This second function can pick an item from the queue and actually send it using the `runicast` connection. The message **is not** removed from the queue. It will be removed (and the space freed) in case the successful callback of the runicast is triggered; In this way the *faulty* messages will be still in the queue to be delivered. The function is in fact recursive: it schedules a call to itself if the queue has additional messages w.r.t the one currently being managed. In any case there is a `MAX_DATA_RETRANSMISSIONS` limit above which the collection packet is removed from the queue.

4.3 Collect reliability

Since the sending uses the runicast primitive, the relative callbacks has been used. `sent_data_rc` is called when the message has successfully sent and so it is removed from the node queue. The `sent_data_timedout` callback instead schedules a call to the resend function `data_collection_resend`. This last function does the following:

- Updates the `retransmitted` attribute of the failed message
- Schedules, using another timer, a call to the `data_collection_send` we discussed above. The difference this time is that the message has been marked as *failed* for one time. The send function will manage differently those packets: they will be retransmitted a number of times lower than the threshold mentioned before; in this way loops of retransmissions are avoided.

4.4 Collect reception

The function `data_collection_uc_recv` is used on collect messages reception. In case the controller receives a packet it will call the application logic via `recv_cb`, the other nodes will instead forward packets to their parents. In both cases there are some relevant actions to mention that are performed by all nodes. Generally speaking the main action performed is to store the message to be delivered in the queue and schedule the actual delivery after some amount of time.

Common logic actions

1. Simple duplicate filter:

All nodes records the last successful collect message managed. Such a message is saved if the destination node confirms the reception with an appropriate ACK.
Example: suppose the scenario in which A sends a message to B which then sends it to C. Suppose that the communication between node BC is successful but node A does not receive an ACK from B. At this point A might retransmit the packet to B and, without the `last_collect` information, also B would retransmit the packet to C.

Above situation is mitigated with the last successful collect message information that, in addition to saving from extra communication, avoids the controller to call the application logic multiple times (in the above example C could be the controller).

2. Update the *downward table*:

   Nodes keeps track of the reverse paths e.g the paths from the leaf to the root of the tree. This information is later used by the actuation phase, section 5, in order to send commands.

Sensors have an additional check to perform: *they look for loops.* They have in fact two parents to whom send the COLLECT message, the second of them might be at the same distance from the root as the first parent or it could be a special parent which has the best RSSI among all neighbours, no matter of the metric. Since the sensor could use such a parent in case the first one is faulty, the message may come back to him. In that case the drop action is taken.

4.4.1 Reception at application layer

Upon this point we did not investigate the application behaviour since it was already implemented. The COLLECT message reception, from the application layer perspective, has to take into account some particular situations as described below.

- Duplicate messages
- Old messages
- Messages referring to concurrent events
- COLLECT messages arriving before the EVENT message

The strategy employed to solve these problems is the following:

1. Employ a `current_managed_event` field in which save what is the event the controller is managing. It is the last event received for which the controller is waiting for COLLECT messages. A timer is used to enforce message reception within a fixed time interval: if a defined percentage of messages is not collected then the timeout is fired. Such a parameter is configurable via `COLLECT_MINIMUM_POOL` constant; default is 60%.

2. Old messages are simply dropped; check is based on both `event_source` and `sequence number`. The same action is taken when controller's current managed event is nothing e.g the controller did not received an EVENT. Note that after it successfully managed an event by collecting enough messages and sending all the commands, the current event turns to be noting. In this way it is used as a flag variable.

3. Collect messages triggered by concurrent events are those messages arriving in a valid time interval e.g when the controller expects a message but they refer to a different event. The solution I adopted is not to drop those packets but instead to keep their readings *as*

they were coming from the correct event. Those readings are in fact *recent enough* since concurrent events take place at the same moment of the current managed one, typically right after.

4. Duplicate messages are as well dropped. The check is made using a list of collected messages that the controller keeps updated for each sensor.

5 Actuation

The actuation phase employ the same strategy as the collection phase. The general objective of this phase is to send actuation (or command) messages from the controller to the sensors/actuators. This is achieved using the *reverse paths* built in the previous phase. All the nodes keep a structure of information called `downward_table` that contains, for each sensor, the next node to which forward messages. It contains **only one** parent per sensor. The question then is:

How to deal with *offline* nodes?

Before answering, let me briefly describe the general picture of this phase.

5.1 General workflow

As stated before, the strategy is the same as the collection phase: each node maintain a queue of messages to be delivered. There are two main operation to manage, the push and the pop. The queue is called `commands_to_send` and, as for the collection phase, is implemented using the very same primitives. The callback `etc_command` adds a message to the queue and schedules a later delivery. The function `send_actuation_command` performs the actual delivery using some logic. The send is performed using the `runicast` primitive and, as you might expect, the proper callbacks were set up. As the previous phase sending, there are mechanisms for duplicate checking, for managing the queue elements and space etc. The different approach in this phase is related to retransmissions.

5.2 Actuation reliability

Since there is a single parent to contact, in case it is faulty there could be problems. The solution I adopted is, in first place, perform more retransmissions to such parent since collisions may occur and assuming the node failure because of a single failed transmission attempt is not reliable. After the node tried at least for the half the number of maximum commands retransmission without success, then the opportunistic communication technique can be used.

5.2.1 Opportunistic forwarding

This technique is, in short, the usage of the broadcast channel instead of the unicast to perfrom the sending of messages. The sender of a packet, once failed multiple times to get in touch with the other node, can broadcast such packet to all the neighbours. The `send_actuation_opportunistic` is rather simple: takes the message to be delivered from a single item buffer and, using a dedicate broadcast connection forwards the message. The reception of the packet is performed through `actuation_opp_bc_recv` callback.

Reception keypoints

1. Check for the destination

The node can be a sensor or a forwarder. Both have to forward packets. In case the packet destination is the triggered sensor then the application `com_cb` callback is called.

2. Duplicate filter

   The node avoids to create loops of transmissions: if a packet has already broadcasted it won't be forwarded again.

3. Broadcasting suppression

   After a node broadcasted a packet it sleeps for a while. This technique also helps in avoiding the transmissions loops. There is an exception for the sensors. Might happen that a sensor receives a command, then, according to this feature, it should suspend the repropagation.

   Example: sensor A receives a command whose destination is sensor B. A broadcasts the packet and does not turn off. After a short amount of time another packet might be received by A which, this time, is also the destination. This situation happens when the controller sends *bursts* of commands to be delivered. If A would have dropped packets for a while it could have missed the packet for himself. This is the reason why sensors do not suspend packets forwarding.

6 Performance analysis

6.1 Cooja simulations

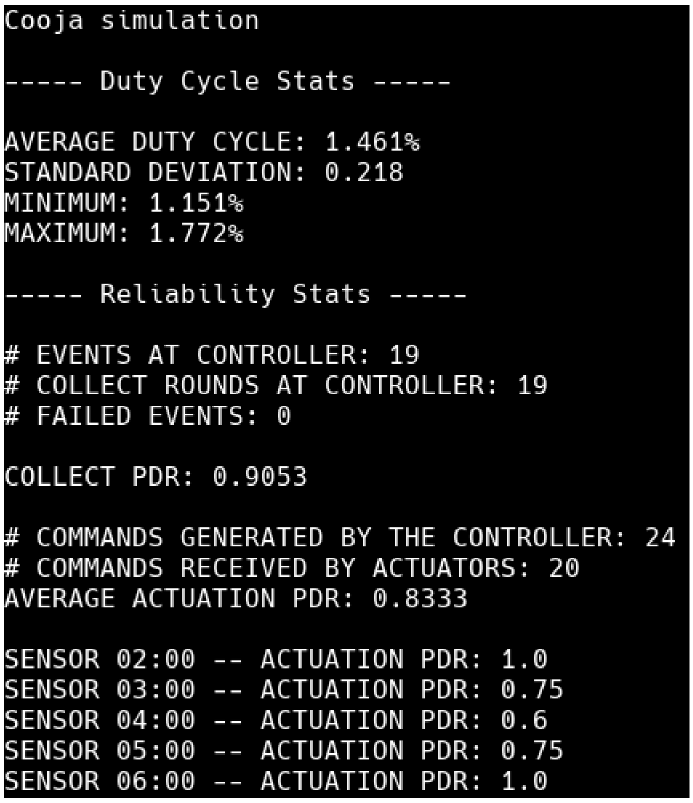
Cooja simulation scenarios have been tested many times during the project development and refinement. The very initial tests showed not high performances when the queue based mechanism was not implemented. The protocol was not able to manage bursts of messages being sent in the network. The tests below reported were executed in the standard conditions of event generation. Simulation duration was set to 30 minutes.

```
#define SENSOR_UPDATE_INTERVAL       (CLOCK_SECOND * 7)
#define SENSOR_UPDATE_INCREMENT      (random_rand() % 300)
#define SENSOR_STARTING_VALUE_STEP   (1000)
#define CONTROLLER_MAX_DIFF          (10000)
#define CONTROLLER_MAX_THRESHOLD     (50000)
#define CONTROLLER_CRITICAL_DIFF     (15000)
```

Listing 3: Default event settings

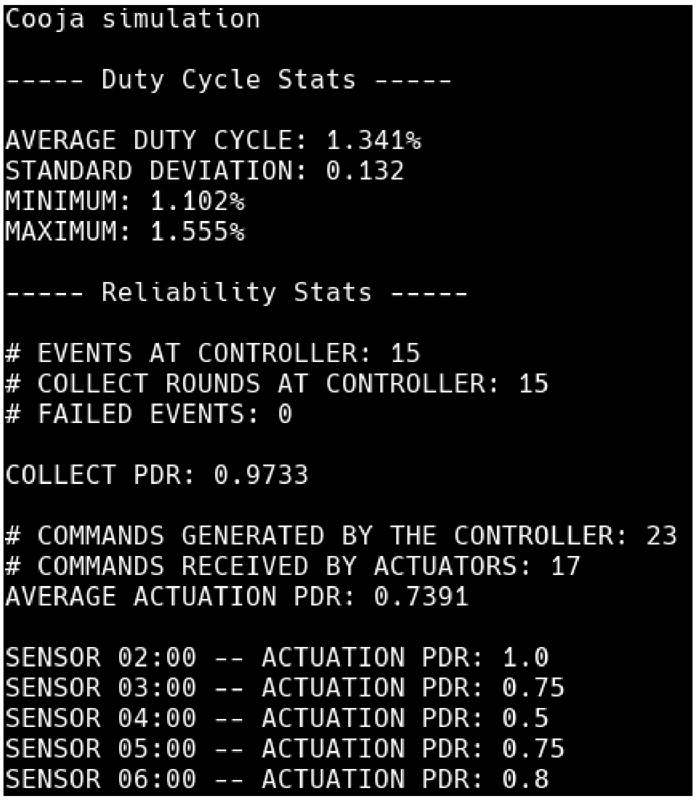
6.1.1 Initial tests

Tests below are related to the Scenario 2 in which the queue based mechanism was not implemented. As you can see, the process of refinement was not straightforward: in the figure 2 the collection PDR is higher than the figure 1 but it comes for a price. Because of collisions and bad timing setting of internal timers the PDR of actuation commands is lower.



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.461%
STANDARD DEVIATION: 0.218
MINIMUM: 1.151%
MAXIMUM: 1.772%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 19
# COLLECT ROUNDS AT CONTROLLER: 19
# FAILED EVENTS: 0

COLLECT PDR: 0.9053

# COMMANDS GENERATED BY THE CONTROLLER: 24
# COMMANDS RECEIVED BY ACTUATORS: 20
AVERAGE ACTUATION PDR: 0.8333

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 0.75
SENSOR 04:00 -- ACTUATION PDR: 0.6
SENSOR 05:00 -- ACTUATION PDR: 0.75
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 1: Scenario 2 - no burst management capability

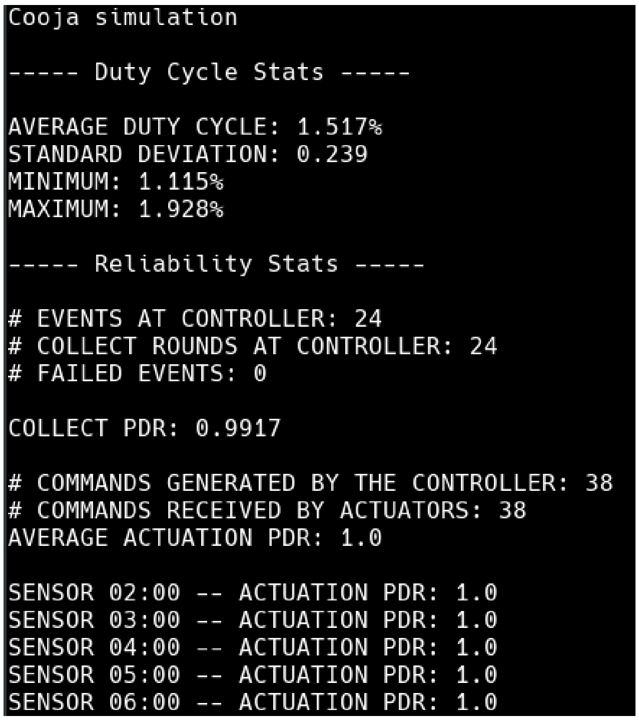


```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.341%
STANDARD DEVIATION: 0.132
MINIMUM: 1.102%
MAXIMUM: 1.555%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 15
# COLLECT ROUNDS AT CONTROLLER: 15
# FAILED EVENTS: 0

COLLECT PDR: 0.9733

# COMMANDS GENERATED BY THE CONTROLLER: 23
# COMMANDS RECEIVED BY ACTUATORS: 17
AVERAGE ACTUATION PDR: 0.7391

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 0.75
SENSOR 04:00 -- ACTUATION PDR: 0.5
SENSOR 05:00 -- ACTUATION PDR: 0.75
SENSOR 06:00 -- ACTUATION PDR: 0.8
```

Figure 2: Scenario 2 - burst little improvement

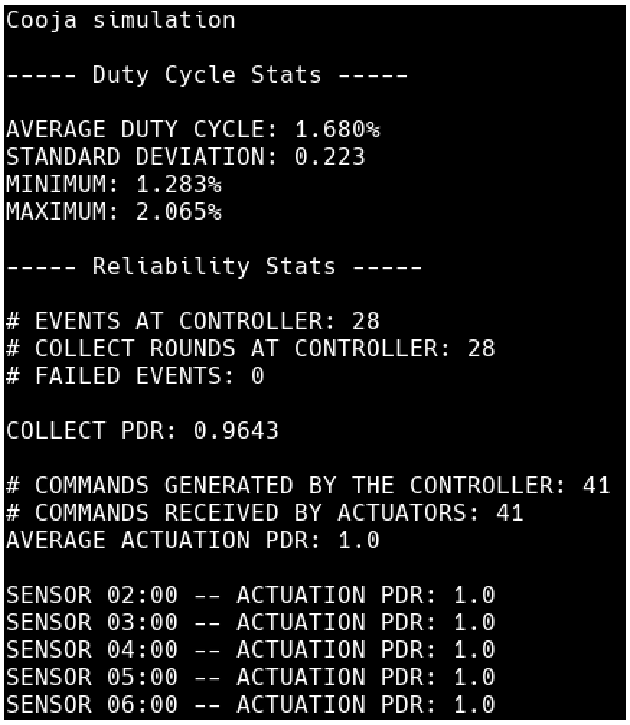
6.2 Doubling events

The following experiments have been conducted by doubling the number of events generated w.r.t the standard conditions: `SENSOR_UPDATE_INCREMENT (random_rand() % 300) * 2`



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.517%
STANDARD DEVIATION: 0.239
MINIMUM: 1.115%
MAXIMUM: 1.928%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 24
# COLLECT ROUNDS AT CONTROLLER: 24
# FAILED EVENTS: 0

COLLECT PDR: 0.9917

# COMMANDS GENERATED BY THE CONTROLLER: 38
# COMMANDS RECEIVED BY ACTUATORS: 38
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 3: Scenario 1 - double events ratio



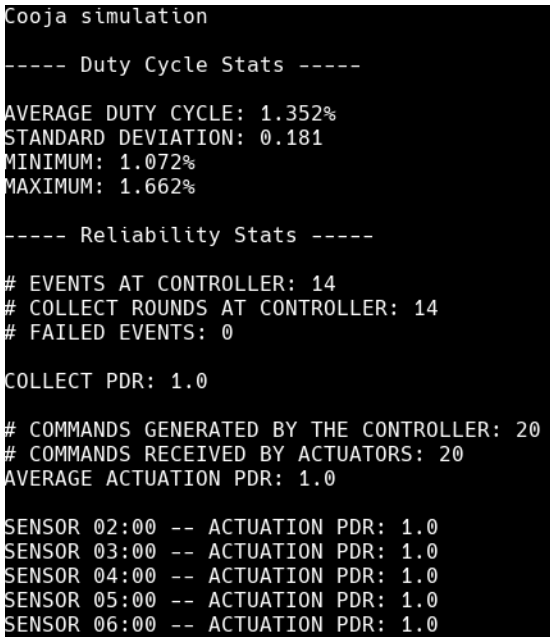
```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.680%
STANDARD DEVIATION: 0.223
MINIMUM: 1.283%
MAXIMUM: 2.065%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 28
# COLLECT ROUNDS AT CONTROLLER: 28
# FAILED EVENTS: 0

COLLECT PDR: 0.9643

# COMMANDS GENERATED BY THE CONTROLLER: 41
# COMMANDS RECEIVED BY ACTUATORS: 41
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 4: Scenario 2 - double events ratio

6.3 Final performances

Below you can find the final performances achieved by the protocol using the standard event ratio as described in 6.1 in the Cooja environment.

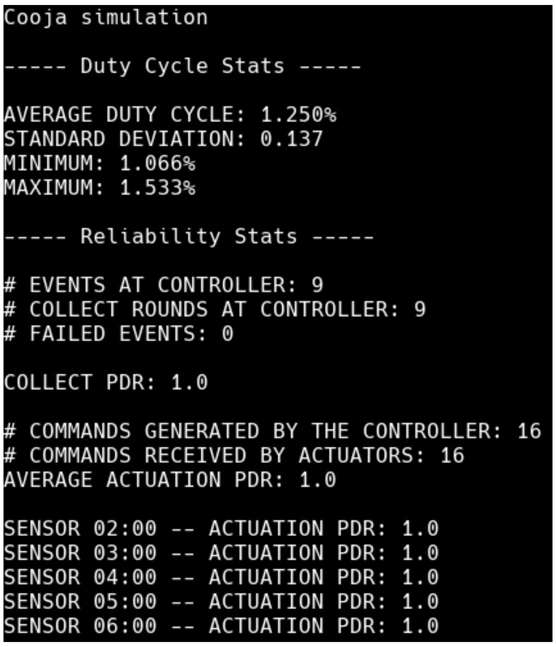
6.3.1 Scenario 1



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.352%
STANDARD DEVIATION: 0.181
MINIMUM: 1.072%
MAXIMUM: 1.662%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 14
# COLLECT ROUNDS AT CONTROLLER: 14
# FAILED EVENTS: 0

COLLECT PDR: 1.0

# COMMANDS GENERATED BY THE CONTROLLER: 20
# COMMANDS RECEIVED BY ACTUATORS: 20
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 5: S1 standard conditions

6.3.2 Scenario 2



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.250%
STANDARD DEVIATION: 0.137
MINIMUM: 1.066%
MAXIMUM: 1.533%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 9
# COLLECT ROUNDS AT CONTROLLER: 9
# FAILED EVENTS: 0

COLLECT PDR: 1.0

# COMMANDS GENERATED BY THE CONTROLLER: 16
# COMMANDS RECEIVED BY ACTUATORS: 16
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 6: S2 standard conditions

6.4 Node failures analysis

As described in the specification document of the project, the node failures analysis was limited to simulated Cooja scenarios. Generally speaking the results have been evaluated with the provided script as done for previous tests. Visual evaluation has been also used to check for specific sensor failed messages. The failure has been caused after some minutes, typically 4, after the system started and the faulty node has been kept in such status for the whole 30 minutes of the simulation. For each failure analysis two experiments were done, the ones reported are those having the worst values in terms of performances.

6.5 Scenario 1

- **Node 8 failure**

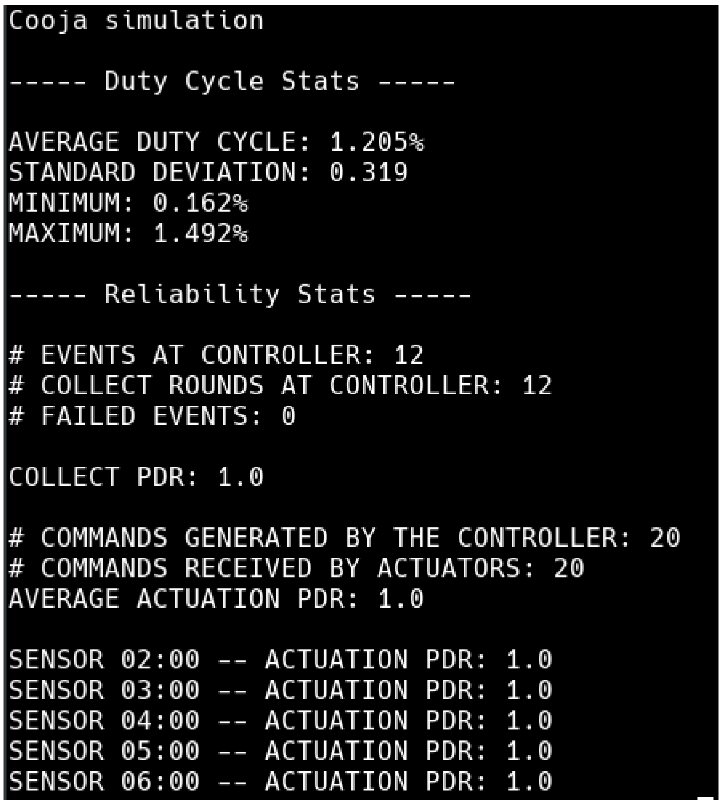
  Node 3 performance not affected. Figure 7.

- **Node 9 failure**

  Sensor 4 performance not affected. Maximum duty cycle, above 2%, registered at node 12. Figure 8.

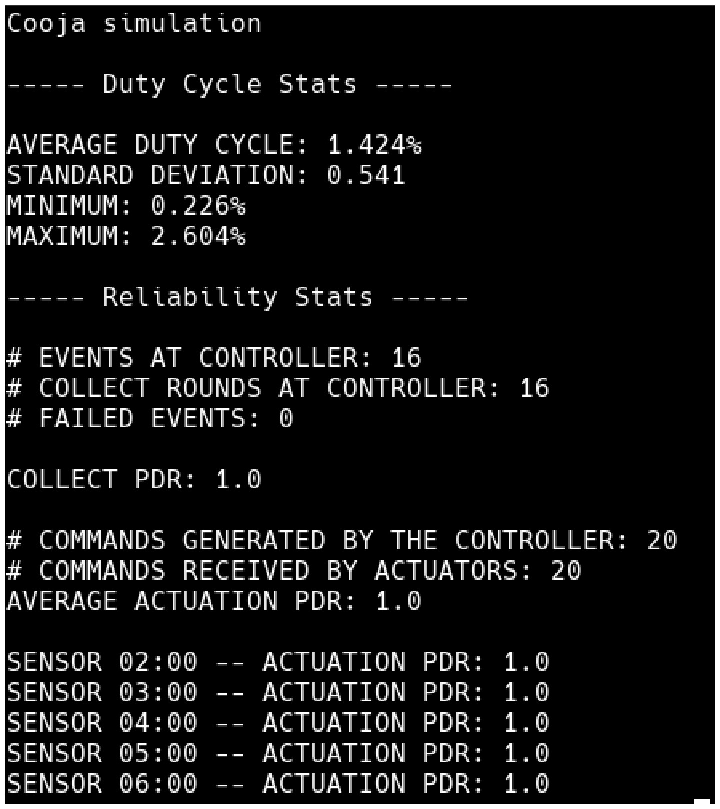
- **Node 14 failure**

  Sensor 6 performance not affected. Only the node 4 failed to send few collect messages making the collect PDR decrease. Figure 9.



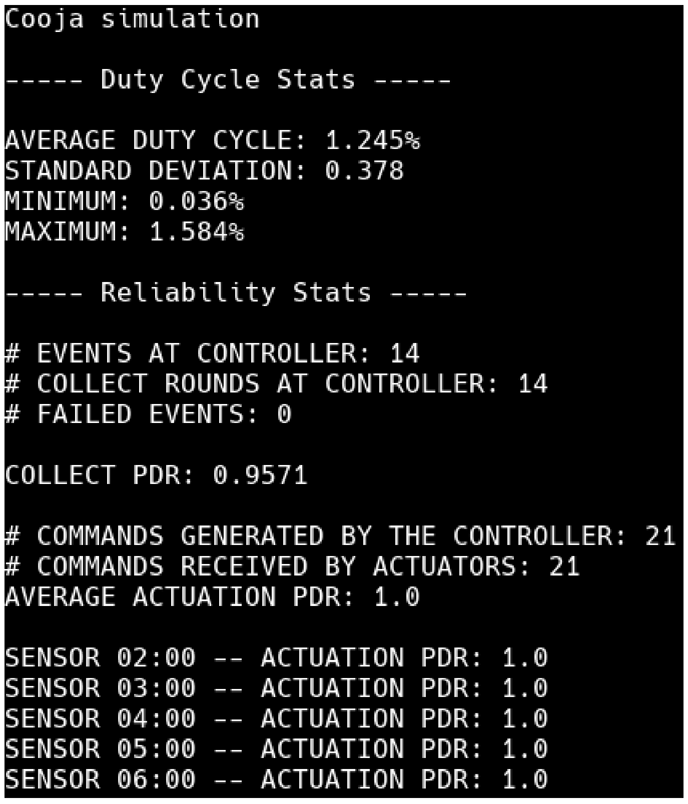
```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.205%
STANDARD DEVIATION: 0.319
MINIMUM: 0.162%
MAXIMUM: 1.492%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 12
# COLLECT ROUNDS AT CONTROLLER: 12
# FAILED EVENTS: 0

COLLECT PDR: 1.0

# COMMANDS GENERATED BY THE CONTROLLER: 20
# COMMANDS RECEIVED BY ACTUATORS: 20
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 7: S1 node 8 failure



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.424%
STANDARD DEVIATION: 0.541
MINIMUM: 0.226%
MAXIMUM: 2.604%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 16
# COLLECT ROUNDS AT CONTROLLER: 16
# FAILED EVENTS: 0

COLLECT PDR: 1.0

# COMMANDS GENERATED BY THE CONTROLLER: 20
# COMMANDS RECEIVED BY ACTUATORS: 20
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 8: S1 node 8 failure



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.245%
STANDARD DEVIATION: 0.378
MINIMUM: 0.036%
MAXIMUM: 1.584%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 14
# COLLECT ROUNDS AT CONTROLLER: 14
# FAILED EVENTS: 0

COLLECT PDR: 0.9571

# COMMANDS GENERATED BY THE CONTROLLER: 21
# COMMANDS RECEIVED BY ACTUATORS: 21
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 9: S1 node 14 failure

6.6 Scenario 2

- **Node 8 failure**

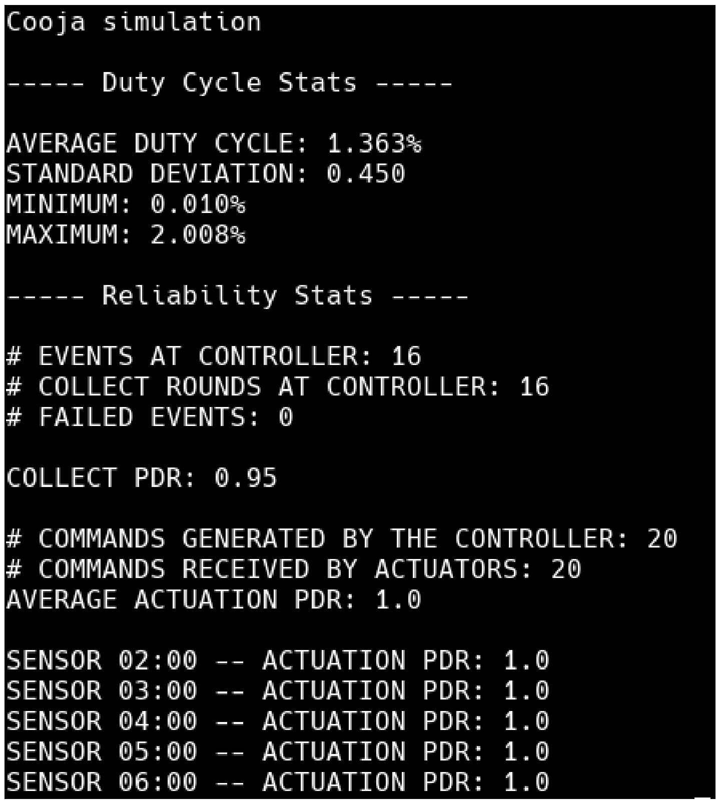
  There are 76 collect messages successfully sent out of the expected 80 e.g 95% of PDR. All the nodes actually sent the messages but they were dropped by the controller due to timeout (10 seconds) in the collection phase; those message arrived late. Sensor 2 failed to send one message, the other three faulty messages were related to sensor 5 and 6.

- **Node 9 failure**

  There are 81 collect messages successfully sent out of 85. The sensor 5 performances were not affected. Sensor 6 and 3 failed to send few collect messages.

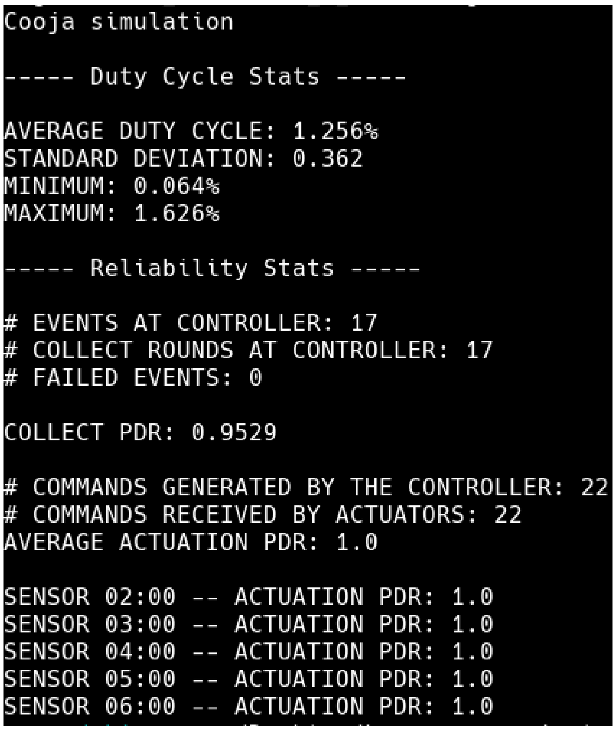
- **Node 14 failure**

  In this case there are 73 out of 80 messages correctly received by the controller. Sensor 3 failed to send two messages, sensor 4 failed to send four messages and sensor 4 failed just one. By looking at the controller dropped messages I could find that, as well as for the previous case, all the nodes actually sent their messages but they arrived late at the controller.



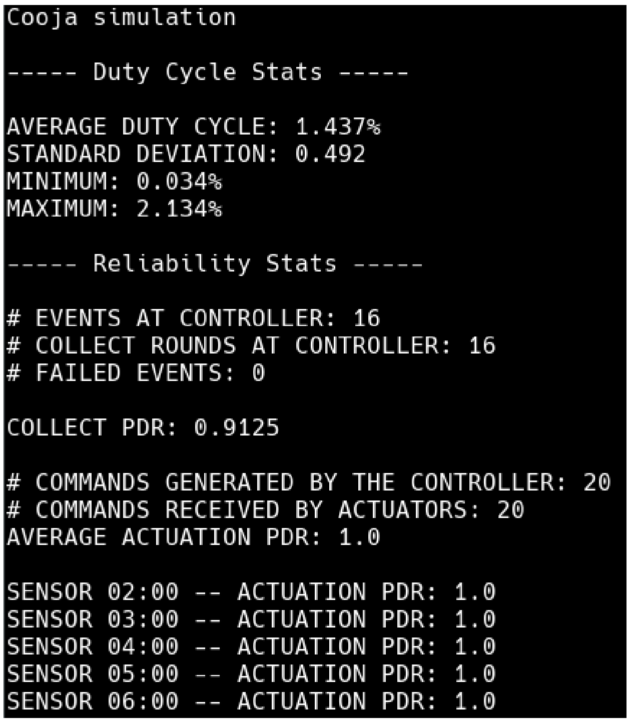
```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.363%
STANDARD DEVIATION: 0.450
MINIMUM: 0.010%
MAXIMUM: 2.008%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 16
# COLLECT ROUNDS AT CONTROLLER: 16
# FAILED EVENTS: 0

COLLECT PDR: 0.95

# COMMANDS GENERATED BY THE CONTROLLER: 20
# COMMANDS RECEIVED BY ACTUATORS: 20
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 10: S2 node 8 failure



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.256%
STANDARD DEVIATION: 0.362
MINIMUM: 0.064%
MAXIMUM: 1.626%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 17
# COLLECT ROUNDS AT CONTROLLER: 17
# FAILED EVENTS: 0

COLLECT PDR: 0.9529

# COMMANDS GENERATED BY THE CONTROLLER: 22
# COMMANDS RECEIVED BY ACTUATORS: 22
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 11: S2 node 9 failure



```
Cooja simulation

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.437%
STANDARD DEVIATION: 0.492
MINIMUM: 0.034%
MAXIMUM: 2.134%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 16
# COLLECT ROUNDS AT CONTROLLER: 16
# FAILED EVENTS: 0

COLLECT PDR: 0.9125

# COMMANDS GENERATED BY THE CONTROLLER: 20
# COMMANDS RECEIVED BY ACTUATORS: 20
AVERAGE ACTUATION PDR: 1.0

SENSOR 02:00 -- ACTUATION PDR: 1.0
SENSOR 03:00 -- ACTUATION PDR: 1.0
SENSOR 04:00 -- ACTUATION PDR: 1.0
SENSOR 05:00 -- ACTUATION PDR: 1.0
SENSOR 06:00 -- ACTUATION PDR: 1.0
```

Figure 12: S2 node 14 failure

6.7 Testbed

Experiments have been made also on the Testbed using the settings suggested both in the project specification document and in the project template. The `experiment.json` file has been configured as follows:

```
{
    "name" : "LpIoT - Arpino",
    "description" : "Nicola Arpino Lpiot project",
    "start_time" : "asap",
    "duration" : 600,
    "binaries" : {
      "hardware" : "firefly",
      "bin_file" : "app.bin",
      "targets" : "disi_povo2",
      "programAddress": "0x00200000"
    }
  }
```

Listing 4: Json configuration of the experiment

The duration for the experiments was 10 minutes and they were scheduled to run in both *day* and *night* scenarios. The experiments, after some failing attempts due to misconfiguration, were scheduled with no problems reported.

6.7.1 Results

- **Experiment at 20:25 p.m**

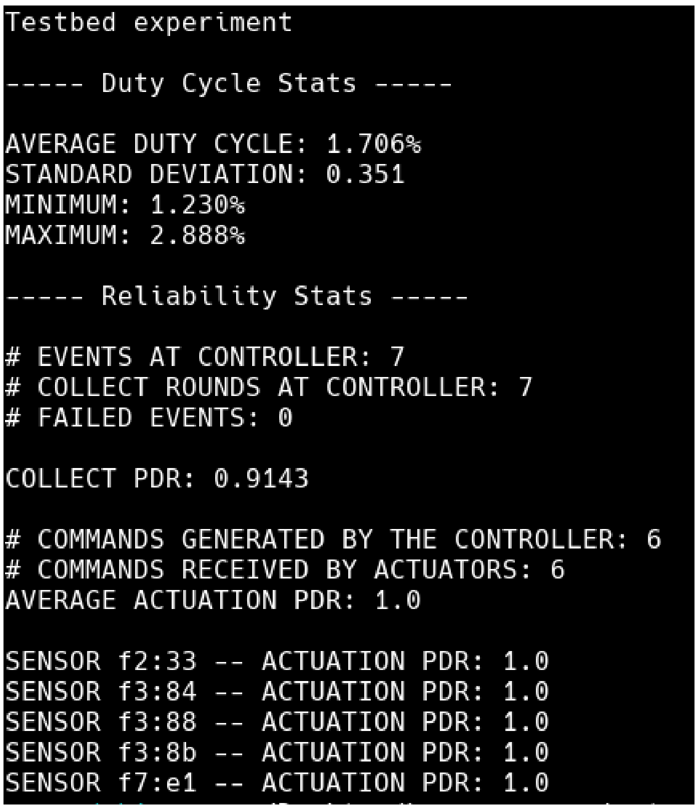
  This was the very fist experiment run, scheduled at 19.00 pm but actually started later. As we discussed in class, the night experiments should provide better results w.r.t the day ones; with this idea in mind I expected good performances. As you can see from the figure 13, the PDR indices are not as high as the simulation ones.

- **Experiment at 01:19 a.m**

  This other experiment has been scheduled right after the previous one finished but, as you can see, it actually started some hours later. The collect PDR is even lower than the previous one. This fact could imply that at the time when the test was running, there where other simulations taking place at nodes nearby.

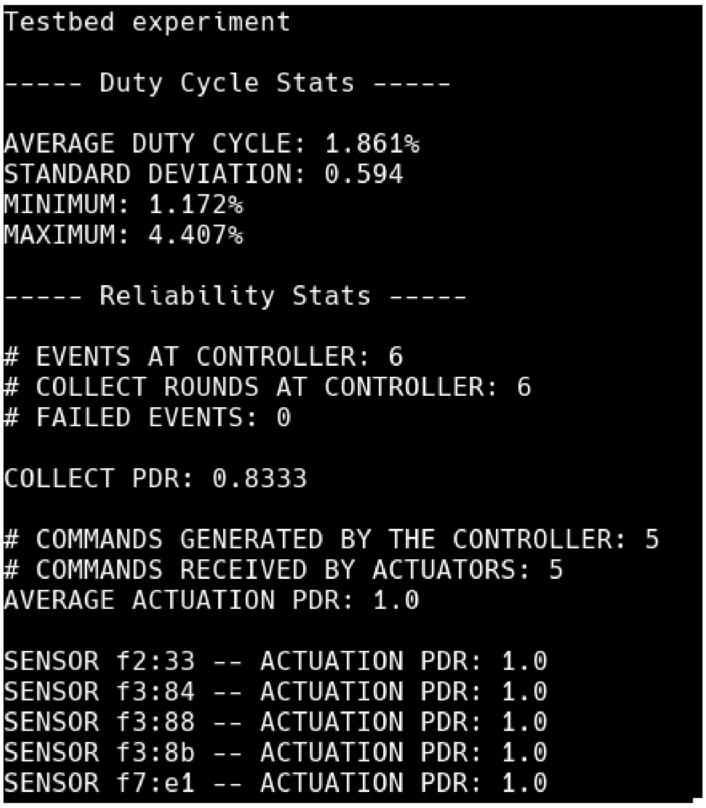
- **Experiment at 12:05 a.m**

  The day context, as expected, provides performances lower than the night scenario.



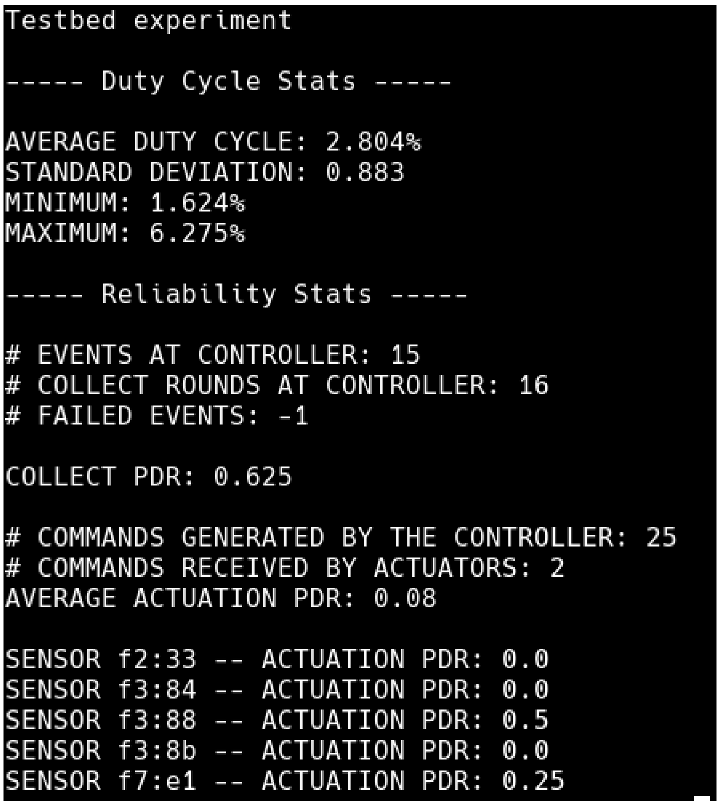
```
Testbed experiment

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.706%
STANDARD DEVIATION: 0.351
MINIMUM: 1.230%
MAXIMUM: 2.888%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 7
# COLLECT ROUNDS AT CONTROLLER: 7
# FAILED EVENTS: 0

COLLECT PDR: 0.9143

# COMMANDS GENERATED BY THE CONTROLLER: 6
# COMMANDS RECEIVED BY ACTUATORS: 6
AVERAGE ACTUATION PDR: 1.0

SENSOR f2:33 -- ACTUATION PDR: 1.0
SENSOR f3:84 -- ACTUATION PDR: 1.0
SENSOR f3:88 -- ACTUATION PDR: 1.0
SENSOR f3:8b -- ACTUATION PDR: 1.0
SENSOR f7:e1 -- ACTUATION PDR: 1.0
```

Figure 13: Testbed 20:25 p.m



```
Testbed experiment

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 1.861%
STANDARD DEVIATION: 0.594
MINIMUM: 1.172%
MAXIMUM: 4.407%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 6
# COLLECT ROUNDS AT CONTROLLER: 6
# FAILED EVENTS: 0

COLLECT PDR: 0.8333

# COMMANDS GENERATED BY THE CONTROLLER: 5
# COMMANDS RECEIVED BY ACTUATORS: 5
AVERAGE ACTUATION PDR: 1.0

SENSOR f2:33 -- ACTUATION PDR: 1.0
SENSOR f3:84 -- ACTUATION PDR: 1.0
SENSOR f3:88 -- ACTUATION PDR: 1.0
SENSOR f3:8b -- ACTUATION PDR: 1.0
SENSOR f7:e1 -- ACTUATION PDR: 1.0
```

Figure 14: Testbed 01:19 a.m



```
Testbed experiment

----- Duty Cycle Stats -----

AVERAGE DUTY CYCLE: 2.804%
STANDARD DEVIATION: 0.883
MINIMUM: 1.624%
MAXIMUM: 6.275%

----- Reliability Stats -----

# EVENTS AT CONTROLLER: 15
# COLLECT ROUNDS AT CONTROLLER: 16
# FAILED EVENTS: -1

COLLECT PDR: 0.625

# COMMANDS GENERATED BY THE CONTROLLER: 25
# COMMANDS RECEIVED BY ACTUATORS: 2
AVERAGE ACTUATION PDR: 0.08

SENSOR f2:33 -- ACTUATION PDR: 0.0
SENSOR f3:84 -- ACTUATION PDR: 0.0
SENSOR f3:88 -- ACTUATION PDR: 0.5
SENSOR f3:8b -- ACTUATION PDR: 0.0
SENSOR f7:e1 -- ACTUATION PDR: 0.25
```

Figure 15: Testbed 12:05 a.m