UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Low-power wireless networking for the Internet of Things

2022/01/25

Project Report

Carlo Corradini
223811

Academic year 2021/2022

Contents

1 Introduction

Unfortunately I've not found a nice and fancy name for the project, so from now on it will be called *lpiot*.
lpiot is the abbreviation of the course name:
Low-power wireless networking for the Internet of Things.
All the available `C` code is documented via *Doxygen* to have a more comprehensive view of the overall logic and context.

1.1 Project

This project consists on developing a low-power wireless networking protocol for event-triggered control, capable of reacting to unpredictable events by rapidly and reliably collecting sensor readings at one controller (sink) and distribute actuation commands over a multi-hop wireless network.
The structure of the project is fully based on the *Contiki Operating System* to manage all operations such as processes management, energy estimation, communication, timers, and so on.
lpiot has been tested via simulations (*Cooja*) and a real scenario (*Testbed*) each of which asses the performance of the implementation in different scenarios.

1.2 GitHub Repository

The source code and the documentation of *lpiot* are available on `GitHub` at the following URL:
`https://github.com/carlocorradini/lpiot`

1.3 Node Roles

Nodes in the wireless network is organized in a control loop holding a predefined role.

1.3.1 Controller

A *Controller* is a special entity, where the control logic resides.
It's responsible of:

- Collecting and processing sensor readings from all the sensor nodes in the network.
- Generating new actuation commands, in order to bring the system back to steady state.
- Communicate such actuation commands to all the actuators.

For simplicity, only a single *Controller* is available. In real systems (e.g., industrial plants), multiple control loops can share the same wireless network, requesting for strategies to minimise, if not totally avoid, mutual interference.

1.3.2 Sensor

A *Sensor* monitor the process under study by periodically sensing the environment and evaluating a local triggering condition. Upon detecting a violation of the triggering condition instruct all other sensors in the network to communicate their updated readings to the *Controller*, effectively providing an updated snapshot of the system status.
In this project a *Sensor* is also an *Actuator*.

1.3.3 Actuator

An *Actuator* waits for commands from the *Controller* and execute them to change the state of the system. In this project an *Actuator* is also a *Sensor*.

1.3.4 Forwarder

A *Forwarder* routes the traffic between *Controller*(s), *Sensor*(s), and *Actuator*(s), extending the physical coverage of the system.

1.4 Assumptions

Some assumptions have been considered during the implementation of the system:

- **`Rime, RDC and MAC layers`**
  The system is implemented at the Rime layer. The MAC layer should be CSMA, while RDC should be ContikiMAC.

- **`Application Interface`**
  An `ETC` application interface used to hide all low level logic to nodes.
  It must provide:

  - `void etc_open(uint16_t channel, const struct etc_callbacks_t *callbacks)`
    Open an ETC connection.
    This function is different from the one given in the template. It only requires the channel on which the connection will operate and a pointer to the callback structure. The other variable are automatically handled inside each module.
    No return type.
  - `void etc_close(void)`
    Close an ETC connection.
    This function is different from the one given in the template. It does not require any variable since the connection can be opened and closed via simple function calls.
  - `void etc_update(uint32_t value, uint32_t threshold)`
    Update sensor data.
    This function is equal to the one given in the template.
  - `bool etc_trigger(uint32_t value, uint32_t threshold)`
    Start event dissemination. If events are suppressed no dissemination to avoid contention.
    This function is different from the one given in the template since ETC is automatically handled inside each module.
    Returns `bool` instead of `int`:
    - `true`: Started event dissemination.
    - `false`: Event(s) are suppressed.
  - `bool etc_command(const linkaddr_t *receiver, enum command_type_t command, uint32_t threshold)`
    Send the command to the receiver actuator node.
    This function is different from the one given in the template since ETC is automatically handled inside each module.
    Returns `bool` instead of `int`:
    - `true`: Command sent.
    - `false`: Command not sent.

- **`Simulating node failure`**
  Node failures should be simulated in firmware. This can be accomplished by simply closing all open connections, ignoring any incoming packet, and ending all *Contiki* processes that use those connections to transmit.
  Failure simulation is triggered by the user pressing the button of the node. Failure is only available in *Cooja*.

- `Logging`
  The provided application already includes several `printf` functions to log events, collected data and actuation commands. These functions and their outputs are not modified because they are used to automatically evaluate the performance of the protocol. However, they are disabled by default and they can be enabled by simply compiling with `make STATS=true` instead of `make`.
  Note that enabling the statistics automatically disable the logger.
  For more information see section Logger.

1.5 Building

The entire building operations are managed through *GNU Make*.
GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files. Make gets its knowledge of how to build program from a file called the makefile, which lists each of the non-source files and how to compute it from other files[3].
For more information about *GNU Make* visit `https://www.gnu.org/software/make/`

1.5.1 Recipes

In the following sections are described the available *GNU Make* recipes that can be used to build/compile and clean the project.

1.5.1.1 all | Default

Build and compile the application:
`make`
Enable statistics logging (used by the *Python* script):
`make STATS=true`
Compile for the testbed environment and target *Zolertia Firefly*:
`make TARGET=zoul`
Note that you can combine both statistics and target:
`make STATS=true TARGET=zoul`

1.5.1.2 cleanall

Clean all the generated files produced by the building and compilation phases. Moreover clean additional log and useless files:
`make cleanall`

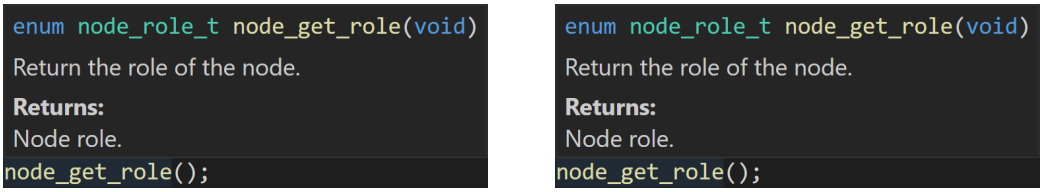
1.6 Contiki Operating System

Contiki is an open source operating system that runs on tiny low-power microcontrollers and makes it possible to develop applications that make efficient use of the hardware while providing standardized low-power wireless communication for a range of hardware platforms.
Contiki is used in numerous commercial and non-commercial systems, such as city sound monitoring, street lights, networked electrical power meters, industrial monitoring, radiation monitoring, construction site monitoring, alarm systems, remote house monitoring, and so on[1].
For more information, see the *Contiki* website at `https://contiki-os.org`

1.7 Doxygen

Doxygen is a documentation generator and static analysis tool for software source trees. When used as a documentation generator, *Doxygen* extracts information from specially-formatted comments within the code. When used for analysis, *Doxygen* uses its parse tree to generate diagrams and charts of the code structure. *Doxygen* can cross reference documentation and code, so that the reader of a document can easily refer to the actual code.
Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL and

to some extent D[2].
Moreover, *Doxygen* is supported by the majority of text editors and IDEs (Integrated development environment) making the overall development experience easier and less heavier.
An example of using *Doxygen* in the project:



(a) Source (b) Result

Figure 1.1: Doxygen comment of a function (left) and the result when using it (right) in *Visual Studio Code*.

1.8 Logger

In the project all messages and errors are logged using a custom handwritten logger.
Logger header (`logger.h`) and source (`logger.c`) files are under the *logger* folder.

1.8.1 Log levels

Log levels are a simple means of classifying log messages based on the information contained.
In the configuration file (`config/config.h`), a level is chosen from those in the list below (ordered by importance) in which it is expected that the lower levels are ignored compared to the higher ones.
The default *log level* is set to **INFO** however when compiling with statistics enabled `make STATS=true` logging is turned off (`DISABLED`). This behavior is fully customizable. E.g When the analysis output report a PDR that is less than 100% is better to have the logger enabled to understand what are the causes rather than pure statistics.

FATAL
: The `FATAL` level designates very severe error events that will presumably lead the application to abort.

ERROR
: The `ERROR` level designates error events that might still allow the application to continue running.

WARN
: The `WARN` level designates potentially harmful situations.

INFO
: The `INFO` level designates informational messages that highlight the progress of the application at coarse-grained level.

DEBUG
: The `DEBUG` level designates fine-grained informational events that are most useful to debug an application.

TRACE
: The `TRACE` level designates fine-grained informational events than the `DEBUG`.

DISABLED
: The `DISABLED` has the highest possible rank and is intended to turn off logging.

Log level \ Output	FATAL	ERROR	WARN	INFO	DEBUG	TRACE
FATAL						
ERROR						
WARN						
INFO						
DEBUG						
TRACE						
DISABLED						

Table 1.1: Correlation between log levels and output

1.8.2 Configuration

```
/* --- LOGGER --- */
/**
 * @brief Logger level.
 */
#ifndef STATS
#define LOGGER_LEVEL LOG_LEVEL_INFO
#else
#define LOGGER_LEVEL LOG_LEVEL_DISABLED
#endif
```

Logger configuration file is under `logger/config.h`.
Default log level is set to `INFO`.
When statistics are enabled during the building phase the logger is automatically disabled via level `DISABLED`.

2 Configuration

Two configuration files are available. Both of them are under the `config` folder.
First file is called `project_conf.h` and is used to configure *Contiki*. No changes have been made with respect to the template.
Second file is called `config.h` and is one of the most important file in the entire application becauseis used to configure and customize the application's behaviour.
Following is a list describing all configuration variables:

- `LOGGER_LEVEL`
  Logger level used by the logger.
  Value: `LOG_LEVEL_INFO`
  Default log level is set to `INFO` however when compiling with statistics enabled (`make STATS=true`) logging is turned off (`DISABLED`).
  For more information see section Logger.

- `ETC_EVENT_FORWARD_DELAY`
  Time to wait before sending an event message.
  Value: `(random_rand() % (CLOCK_SECOND / 10))`

- `ETC_COLLECT_START_DELAY`
  Time to wait before sending a collect message.
  Value: `(CLOCK_SECOND * 3 + random_rand() % (CLOCK_SECOND * 2))`

- `ETC_SUPPRESSION_EVENT_NEW`
  New event generation suppression time.
  Value: `(CLOCK_SECOND * 12)`

- `ETC_SUPPRESSION_EVENT_PROPAGATION`
  Event propagation suppression time.
  Value: `(ETC_SUPPRESSION_EVENT_NEW - CLOCK_SECOND / 2)`

- `ETC_SUPPRESSION_EVENT_PROPAGATION_END`
  Time to wait to disable suppression propagation after received a command message.
  Value: `(CLOCK_SECOND / 2)`

- `CONTROLLER`
  Controller address.
  Value: `{{0x01, 0x00}}`
  Default controller address is set to `{{0x01, 0x00}}` however when compiling with a different target, Zolertia Firefly, (`make TARGET=zoul`) controller address is set to `{{0xF7, 0x9C}}`.

- `CONTROLLER_MAX_DIFF`
  Maximum sensor value difference.
  Value: `(10000)`

- `CONTROLLER_MAX_THRESHOLD`
  Maximum sensor threshold.
  Value: `(50000)`

- `CONTROLLER_COLLECT_WAIT`
  Time to wait before analyzing the Sensor readings.
  Value: `(CLOCK_SECOND * 10)`

- NUM_SENSORS
  Total number of Sensor nodes available.
  Value: (5)

- SENSORS
  Sensors addresses.
  Value:
  ```
  {
   {0x02, 0x00}},
   {0x03, 0x00}},
   {0x04, 0x00}},
   {0x05, 0x00}},
   {0x06, 0x00}}
  ```
  } Default sensor addresses are set to the above values however when compiling with a different target, Zolertia Firefly, (`make TARGET=zoul`) sensor addresses are set to:
  ```
  {
   {0xF3, 0x84}},
   {0xF2, 0x33}},
   {0xf3, 0x8b}},
   {0xF3, 0x88}},
   {0xF7, 0xE1}}
  }
  ```

- SENSOR_UPDATE_INTERVAL
  Interval to sense the value.
  Value: (CLOCK_SECOND * 7)

- SENSOR_UPDATE_INCREMENT
  Random increment to add to the old sensed value.
  Value: (random_rand() % 300)

- SENSOR_INITIAL_VALUE
  Initial sensed value step.
  Value: (1000)

- CONNECTION_CHANNEL
  Channel(s) on which the connection will operate.
  Value: (0xAA)

- CONNECTION_RSSI_THRESHOLD
  RSSI threshold.
  Value: (-95)

- CONNECTION_MAX_HOPS
  Maximum number of hops a message could be forwarded. Defines the threshold on which a message is discarded if has been forwarded too many times. The value is inclusive. Mostly used to detect loops.
  Value: (16)

- CONNECTION_BEACON_MAX_CONNECTIONS
  Maximum number of connections to store.
  Value: (3)

- CONNECTION_BEACON_INTERVAL
  Interval to (re)create the connections tree.
  Value: (CLOCK_SECOND * 30)

- `CONNECTION_BEACON_FORWARD_DELAY`
  Time to wait before sending a beacon message.
  Value: `(random_rand() % CLOCK_SECOND)`

- `CONNECTION_UC_BUFFER_SIZE`
  Unicast buffer size. The maximum number of unicast messages that the buffer could store. A good value is the number of Sensor nodes available.
  Value: `(NUM_SENSORS)`

- `CONNECTION_UC_BUFFER_MAX_SEND`
  Maximum number of send attempts for a packet in the buffer.
  Value: `(1)`

- `CONNECTION_UC_BUFFER_SEND_DELAY`
  Time to wait before sending a message in the unicast buffer that failed at least one time.
  Value: `(random_rand() % (CLOCK_SECOND / 10))`

- `CONNECTION_FORWARD_MAX_SIZE`
  Maximum number of hops in forwarding structure.
  Value: `(3)`

- `CONNECTION_FORWARD_DISCOVERY_TIMEOUT`
  Time to wait before checking the forward hops of the sensor.
  Value: `(CLOCK_SECOND * 1)`

3 Project Structure

I've re-managed the overall project structure to obtain a more organized architecture where each folder is specialized for a unique purpose.

3.1 .vscode | Folder

Visual Studio Code configuration folder.
For more information visit `https://code.visualstudio.com/docs/getstarted/settings`

3.1.1 c_cpp_properties.json | File

C/C++ extension configuration file.
For more information visit `https://code.visualstudio.com/docs/cpp/customize-default-settings-cpp`

3.1.2 extensions.json | File

Recommended list of extensions.
Fore more information visit `https://code.visualstudio.com/docs/editor/extension-marketplace#_workspace-recommended-extensions`

3.1.3 settings.json | File

Visual Studio Code configuration file.
For more information visit `https://code.visualstudio.com/docs/getstarted/settings`

3.2 scenarios | Folder

Scenarios folder where all different scenarios are saved.

3.2.1 scenario_1 | Folder

Cooja scenario 1.

3.2.1.1 gui.csc | File

Cooja simulation file with support for GUI.

3.2.1.2 nogui.csc | File

Cooja simulation file without support for GUI.

3.2.2 scenario_2 | Folder

Cooja scenario 2. *Cooja* simulation file with support for GUI.

3.2.2.1 gui.csc | File

Cooja simulation file with support for GUI.

3.2.2.2 nogui.csc | File

Cooja simulation file without support for GUI.

3.2.3 testbed | Folder

Testbed scenario.

3.2.3.1 experiment.json | File

Testbed configuration file.

3.2.3.2 get-test.sh | File

Utility shell script to obtain testbed logs given an identifier.

3.2.3.3 run-test.sh | File

Utility shell script to run a test bed experiment.

3.2.4 parse-stats.py | File

Python script to extract and analyze statistics.

3.3 src | Folder

Source files folder.

3.3.1 config | Folder

Configuration folder.

3.3.1.1 config.{h|c} | File

Application configuration file(s).

3.3.1.2 project_conf.h | File

Contiki configuration file.

3.3.2 connection | Folder

Connection folder where all related source files regarding networking are saved.

3.3.2.1 beacon.{h|c} | File

Beacon source file(s).

3.3.2.2 connection.{h|c} | File

Connection source file(s).

3.3.2.3 forward.{h|c} | File

Forwarding structure source file(s).

3.3.2.4 uc_buffer.{h|c} | File

Unicast buffer source file(s).

3.3.3 etc | Folder

Event-Triggered Control (ETC) folder.

3.3.3.1 etc.{h|c} | File

ETC source file(s).

3.3.4 logger | Folder

Logger folder.

3.3.4.1 logger.{h|c} | File

Logger source file(s).

3.3.5 node | Folder

Node folder where all node types and their behavior are saved.

3.3.5.1 controller | Folder

Controller node folder.

controller.{h|c} | File

Controller node source file(s).

3.3.5.2 forwarder | Folder

Forwarder node folder.

forwarder.{h|c} | File

Forwarder node source file(s).

3.3.5.3 sensor | Folder

Sensor node folder.

sensor.{h|c} | File

Sensor node source file(s).

3.3.5.4 node.{h|c} | File

Node utility source file(s).

3.3.6 tool | Folder

Tool folder where third-party tools are saved.

3.3.6.1 simple_energest.{h|c} | File

Contiki Process which prints information about Radio and CPU usage obtained with *Contiki* Energest source file(s).

3.3.7 app.c | File

Main application file. It's the starting point of every node.

3.4 lpiot | Root Folder

Project root folder.

3.4.1 .clang-format | File

Clang-Format Style Options configuration file.
For more information visit `https://clang.llvm.org/docs/ClangFormatStyleOptions.html`

3.4.2 .gitattributes | File

A gitattributes file is a simple text file that gives attributes to pathnames.
For more information visit `https://git-scm.com/docs/gitattributes`

3.4.3 .gitignore | File

Specifies intentionally untracked files to ignore.
For more information visit `https://git-scm.com/docs/gitignore`

3.4.4 LICENSE | File

MIT Open source license file.
For more information visit `https://opensource.org/licenses/MIT`

3.4.5 Makefile | File

GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.
For more information visit `https://www.gnu.org/software/make`

3.4.6 README.md | File

README file in Markdown language.
For more information about README files visit `https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes`
For more information about Markdown language visit `https://www.markdownguide.org`

4 Tree Construction

The *Controller* node is in charge of initiating floods of beacon messages to (re)construct the collection tree. To propagate the flood while reducing the chance of collisions, nodes retransmit the received beacon message after a small random delay. See Configuration chapter for more information about the different values used in the application.

4.1 Metrics

In the tree construction protocol there are multiple metrics that the beacon algorithm should consider to correctly select and/or discard a parent node.
These metrics are listed below and they are ordered by importance. This means that to evaluate a new beacon message that has metric textitX equal to the current one it must be evaluated also metric Y even if it's less important that X.

4.1.1 seqn | Sequence Number

Sequence number is a monotonic value increased by 1 each time a new flood to (re)construct the tree is initiated by the *Controller* node. It's used to understand if the received beacon message is new. A metric with greater sequence number always replace an old (saved) metric.
Note that the sequence number could overflow resetting the *seqn* to 0. This behavior is handled, giving special precedence.

4.1.2 hopn | Hop Number

Hop number is the distance between the node and the *Controller* node. A high-hop number means that the node is far from the *Controller* node.
If two beacon messages have the same `seqn`, but the first one has a lower `hopn` compared to the second one, the first one is chosen and the second one saved as a backup (more later).

4.1.3 rssi | Received Signal Strength Indication

Received signal strength indicator (RSSI) is a measurement of the power present in a received radio signal.
When a message is received the first thing to check is that the RSSI does not fall above the `RSSI_THRESHOLD`. Moreover, the RSSI indicates how good the signal between the transmitter and the receiver is. Note that the last is not always true because during transmissions there can be interference and collisions degrading the overall quality.
A parent node that has a lower RSSI (and better `hopn`) compared to the backup has higher chances to fail during transmissions.

4.2 Algorithm

```
142 /* Analyze received beacon message */
143   if (rssi <= CONNECTION_RSSI_THRESHOLD) return; /* Too Weak */
144   if (beacon_msg.seqn != 0 && beacon_msg.seqn < connections[0].seqn)
145     return; /* Old (Keep in mind seqn overflow) */
146   if (beacon_msg.seqn == connections[0].seqn) {
147     /* Same sequence number, check... */
148     for (connection_index = 0;
149          connection_index < CONNECTION_BEACON_MAX_CONNECTIONS;
150          ++connection_index) {
151       if ((beacon_msg.seqn == 0 &&
152            beacon_msg.seqn < connections[connection_index].seqn) ||
153           beacon_msg.seqn > connections[connection_index].seqn) {
154         break; /* Better -> New */
155       }
```

```
156
157       if (beacon_msg.hopn + 1 > connections[connection_index].hopn) {
158         continue; /* Worse -> Far */
159       }
160       if (beacon_msg.hopn + 1 == connections[connection_index].hopn &&
161           rssi <= connections[connection_index].rssi) {
162         continue; /* Worse -> Weak */
163       }
164
165       /* Better -> Near or Strong */
166       break;
167     }
168
169     if (connection_index >= CONNECTION_BEACON_MAX_CONNECTIONS)
170       return; /* Far or Weak */
171   }
```

The above snippet is the core algorithm used to select a beacon message and discard or save as the main or backup parent node.
At line `143` the RSSI is compared to the threshold. If lower than the threshold the received message is discarded.
At line `144` the `seqn` is compared to the current parent `seqn`. If older (lower) the received message is discarded. Note the initial check for overflow. If `seqn` is 0 has the maximum priority.
At line `146` we check if the current `seqn` is the same as the received one. If `true` other metrics should be compared. Otherwise, this is a newer tree construction message and must be used as the primary one.
At line `148` the beacon message is compared against all the current available connections (primary and backups).
At line `151` the `seqn` is compared to the one of the saved messages. If greater, it's newer and must replace the ith connection. Also here is a check for verflow to give importance.
At line `157` the `hopn` metric is compared. If the value is greater than the ith value, this means that being further. Skip the current ith connection and continue with the next one. Otherwise, we found a node that is closer to the *Connection* node or has the same `hopn` value.
At line `160` we check the RSSI value only if the received message and the ith connection are the same. If the RSSI is greater than the current i th connection continue with the next one.
If all the above checks have been passed, we finally found a better parent node so there is no need to continue to check other connection. Terminate the loop with the `break` keyword at line `166`.
At line `169` the index used in the `for` loop is compared with the size of the connections. If greater of equal to the received message is worse than every saved connection so we discard it. Otherwise we are sure that a new better connection is available and should replace the ith connection in the connections array. Note that the index could also be 0 replacing the primary connection.
When all checks and evaluations are done, the next step is to replace the ith connection with the received one. Two things must be noted:

- The ith connection is not removed but a shift operation to the right is performed. The ith connection is removed only if the index is equal to the size of the connections array minus 1 producing a deleting shift.
- In the connections array there are no duplicates of the same node. This means that if multiple messages are received, only the best one is saved. This behavior is used to increase free-spaces and to reduce failure reaction time. If a node is not reachable, it does not make sense to check it again (and again).

4.3 Reliability

Three connections, saved and ordered following the rules described in the previous sections, are stored in the application to increase reliability and react to node failures.

When a message is sent to the current parent (collect message) and a failure is detected the following algorithm is executed. Note that the algorithm is in the textttconnection/connection.c file. This file is an abstraction above Rime to help developing nodes easily.

```
504 /* Give a last chance if receiver is controller */
505 if (!message->last_chance && linkaddr_cmp(receiver, &CONTROLLER)) {
506   retry = true;
507   message->last_chance = true;
508   break;
509 }
510
511 /* Receiver is current parent */
512 if (linkaddr_cmp(receiver, &conn->parent_node)) {
513   /* Invalidate connection */
514   if (connection_invalidate()) {
515     retry = true;
516     message->last_chance = false;
517     message->num_send = CONNECTION_UC_BUFFER_MAX_SEND - 1;
518   }
519 } else {
520   /* Parent changed dinamically */
521   retry = true;
522   message->last_chance = false;
523   message->num_send = CONNECTION_UC_BUFFER_MAX_SEND - 1;
524   break;
525 }
```

If the parent node is the *Controller* node, retry one last time.
Invalidate primary connection if the receiver of the message is the current parent. If the invalidation procedure is successful, retry to send the message to the new parent node. If not, no more available connections are available and the message could not be sent.
If during sending the current parent dynamically changed retry to send the message with the new parent.
Note that a lot of code has been omitted due to space limitations.

4.3.1 Example

Example from scenario 2 where *Sensor* 6 has a faulty parent. Current parent is *Forwarder* node 7 but backup *Sensor* node 5 is available. After the overall procedure we can see that the message has been sent successfully.

```
07:22.514  ID:6   ERROR SENSOR/ACTUATOR connection.c:490: Error sending unicast message to 07:00 on tx 8 due to 2
07:22.523  ID:6   WARN  SENSOR/ACTUATOR connection.c:205: Invalidating connection: { parent_node: 07:00, seqn: 14, hopn: 4, rssi: -93 }
07:22.534  ID:6   INFO  SENSOR/ACTUATOR connection.c:222: Backup connection: { parent_node: 05:00, seqn: 14, hopn: 5, rssi: -88 }
07:22.539  ID:6   INFO  SENSOR/ACTUATOR connection.c:558: Retrying to send last unicast message
07:22.549  ID:6   INFO  SENSOR/ACTUATOR connection.c:600: Sending buffered unicast message: { receiver: 05:00, type: 0, num_send: 1 }
07:22.623  ID:6   INFO  SENSOR/ACTUATOR etc.c:636: Unicast message sent
```

Figure 4.1: Recovery procedure of node 6 due to faulty parent node 7.

5 Downward Forwarding

A forward structure has been built to enable downward forwarding from an upper node to a lower node in the tree. The data collection phase is used to populate this structure. Whenever a node receives a collect message, it associates the source of the sensor reading with the sender of the received message. Node should leverage such information when forwarding command (actuation) messages. The sender of the collect message becomes the next forwarder in the downward path from the *Controller* node to the specific source.
The implementation guarantee that no routing loop is created and also a method to increase the overall reliability when a node is no more reachable.

5.1 Metrics

In the forwarding structure there is a single metric that is considered to correctly select and/or discard a child (hop) node. The metric is listed below.

5.1.1 distance | Hop Distance

Hop distance from the node to the *Sensor* node.
It's used to understand how far a child node is in respect to the receiver *Sensor* node.
The value of this metrics is only used when the reliability procedure has ended (more later) to sort the received responses by distance in ASC order.
Note that the metric is only used during the reliability phase due to no available hops to forward a message.
The most important thing to know is that every time a collect message is received also the forwarding structure is updated. The primary forwarding node is obtained by checking the last received collect message. Therefore, the reception time is the key value. However, global time is hard to achieve a receiv-insert algorithm is more suitable.

5.2 Reliability

The algorithm to recover from a faulty forwarder node (hop) is much more complicated compared to the one for a parent node.
Also in this case an array of forwarding(s) is stored to increase overall reliability and time.
The procedure is divided in two blocks:

1. If sending a message to a child (hop) fails, executes the following algorithm:

```
536   /* Give a last chance */
537   if (!message->last_chance) {
538     retry = true;
539     message->last_chance = true;
540     break;
541   }
542
543   /* Invalidate hop */
544   invalidate_hop(&command_msg->receiver);
545
546   /* Try with new hop or prepare to discovery */
547   retry = true;
548   message->num_send = CONNECTION_UC_BUFFER_MAX_SEND - 1;
```

   If we don't have given a last chance to the needed message, retry for the last time.
   Otherwise, we are sure that the current hop is not valid so invalidate the current hop and retry with the next one (backup) if available.

If no more hop available we need to start a procedure called *Forward Discovery* to learn (if possible) new forwarding nodes. This procedure is shown in the next item.

2. **Forward discovery**
   If no more hops are available we need to start the forward-discovery algorithm and try to discover nodes that can deliver the message to the receiver *Sensor* node. In the image below is shown the initial scenario where node texttt1 tried to forward a command message to its primary hop. However, hop node is unavailable and node `1` has no more hops available to try with.

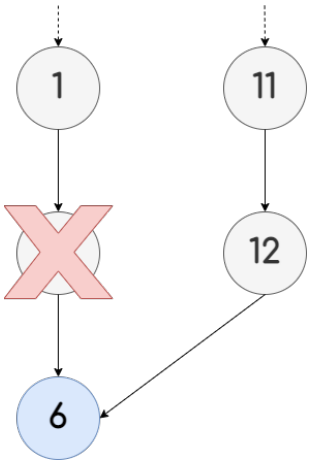


Figure 5.1: Node `1` has no more hop available because the primary one is unavailable

Node `1` send a broadcast message of type `Forward Discovery Request` and ask any receiver node if has a route to *Sensor* `6`.
Moreover, a timeout timer starts.

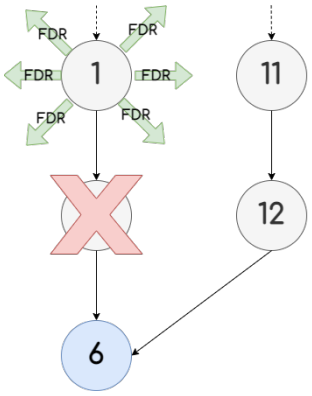


Figure 5.2: Node `1` send in broadcast a FDR message asking if any receiver node has a hop to *Sensor* `6`

Every receiver node checks if has a valid route to *Sensor* `6`. Every receiver node checks if has a valid route to *Sensor* `6`. If not, discard the message and do nothing. Otherwise, check if sender (node `1`) is not primary hop for *Sensor* `6`. If primary, check if another hop is available, otherwise discard the message and do nothing.
If a hop is available, respond to node `1` with hop distance and requested *Sensor* `6` (used to match request).
Note that the response is done in broadcast so other node(s) can learn about the route.

Upon receiving a `Forward Discovery Response` message, node `1` save it to the forwarding structure and sort it by distance.
Note that the distance is increased by one to match the real distance from the node `1`.

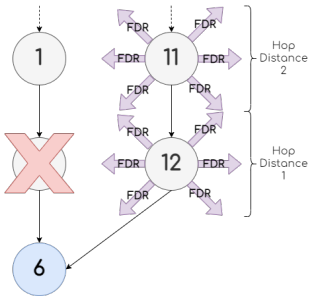


Figure 5.3: Node `11` and `12` respond in broadcast with a FDR message with *Sensor* `6` and hop distance.

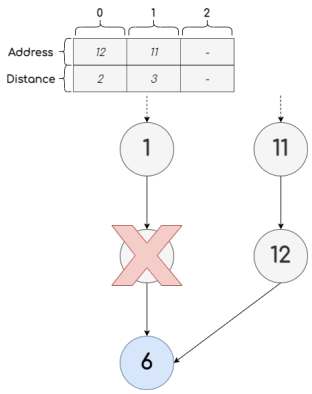


Figure 5.4: Node `1` save sender node as a valid hop and sort forwarding array by distance

When the timer triggers, node `1` checks if *Forward Discovery* procedure has been successfully. If no hop has been found, an error is logged and the message is discarded. Otherwise, redirects the command message to the first available hop in the array (which is the nearest to the *Sensor* `6`). In this case, to node `12`.
Note that node `11` is not removed but is maintained as an available backup hop.

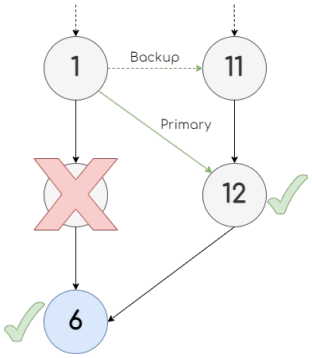


Figure 5.5: Node `1` send unicast message to hop node `12`.

The overall procedure not only helps to handle failures, but also helps to instruct near nodes about possible (new) routes for a specific *Sensor*.

6 Unicast Buffer

Reliability is hard. To maintain a record of the messages that must be sent, a unicast buffer structure has been created.
The structure is specialized on maintaining all the information of the message that must be sent, including retries data.
The structure is a buffer, more simply to an array of `uc_buffer_t` struct. The default size of the buffer is equal to the number of *Sensor*(s) in the application. This number has been extracted by numerous scenarios tentative where the *Controller* node needed to send a command message to every single *Sensor* node. Note that the size of the buffer can be configured in the `config.h` file, located under the `config` folder.
You can find more about unicast buffer in the corresponding source file: `uc_buffer.h` and `uc_buffer.c`

6.1 struct uc_buffer_t

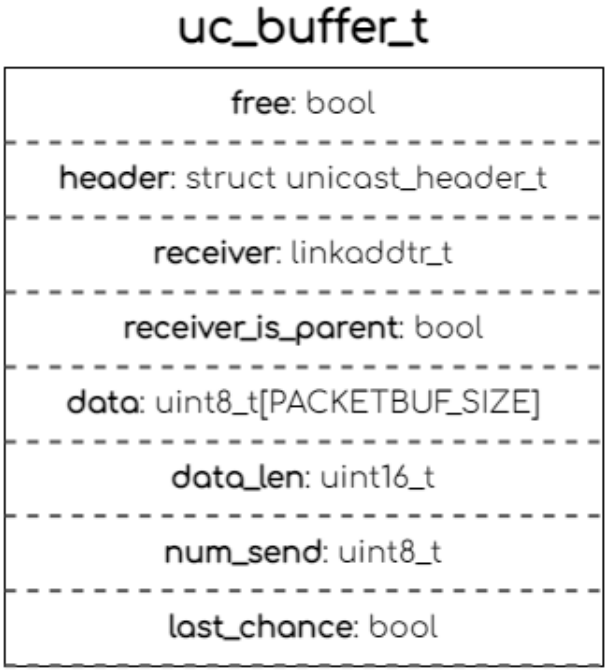


Figure 6.1: Unicast buffer struct entry.

6.1.1 free

Type: `bool`
Flag to inform that the entry is free and can be populated.

6.1.2 header

Type: `struct unicast_header_t`
Original unicast header.

6.1.3 receiver

Type: `linkaddr_t`
Receiver address.
Note that this value can change dynamically if the parent node is equal to the receiver and a

backup is used due to a failure.

6.1.4 receiver_is_parent

Type: `bool`
Flag to inform if the message must be sent to the parent.
This is a direct consequence of dynamic changes of the receiver address.

6.1.5 data

Type: `uint8_t[PACKETBUF_SIZE]`
Original data of the message.
Note that the datatype is `uint8_t`, a byte.

6.1.6 data_len

Type: `uint16_t`
Data length in byte.

6.1.7 num_send

Type: `uint8_t`
Number of times that this message has been sent.

6.1.8 last_chance

Type: `bool`
Flag to inform if this is the last chance for the message to be sent.

7 Scenarios

To understand which node is a *Controller* or a *Sensor* or a *Forwarder*, please refer to the Configuration chapter.
I've tested the application in three different scenarios:
The first and second scenarios are simulated in *Cooja* and are based on the TMote Sky platform. The duration for each simulation is around 30 minutes.
The third scenario is a real testbed with real hardware (Zolertia Firefly) located at the University of Trento. To test the application in this environment, we must compile with a specific target and leverage to an application written in *Python* to schedule a job. Note that we cannot create a job with a duration that is higher than 10 minutes because we can saturate the waiting queue.
The zigzag effect that we can see from the plots is done by the tree (re)construction protocol that every 30 seconds floods the entire network with a beacon message in roadcast. Since each simulation lasts approximately 1800 seconds we have 60 spikes.
All the materials are available at the following Google Drive folder:
`https://drive.google.com/drive/folders/1wenW-CfoMUbLeXvOTO8ZNjgzOD4wd1Q1?usp=sharing`

7.1 Scenario 1

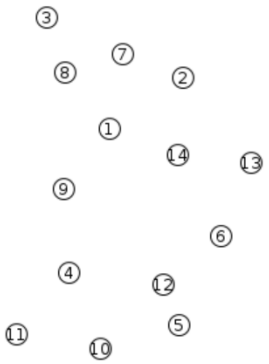


Figure 7.1: Scenario 1.

In the scenario we can clearly see that node number 6 has a huge power consumption towards the end. Following the logs I've found that this is due to an event collection caused by node 5. Node 6 received a bunch of collect message to redirect to the *Controller* node. All the messages are buffered and ready to be sent. However, probably due to collisions, most of the time the operation failed causing a bunch of re-sends and changes in the parent node. Even though we had these issues and a power consumption that is really huge compared to other nodes in the recovery operation successfully delivered the messages to the *Controller* node.
Note that we can see the same behavior in the first half (nodes texttt9, 12, 13, 4, 14) but the impact on energy consumption was way below what happened to node 6.

7.1.1 Statistics

```
Namespace(logfile='scenario1_nogui_mrm.log', testbed=False)
Logfile: scenario1_nogui_mrm.log
Cooja simulation

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

AVERAGE DUTY CYCLE: 1.233%
STANDARD DEVIATION: 0.161
MINIMUM: 1.064%
MAXIMUM: 1.644%

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

# EVENTS AT CONTROLLER: 15
# COLLECT ROUNDS AT CONTROLLER: 15
# 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 7.2: Scenario 1 | Statistics

7.1.1.1 DC

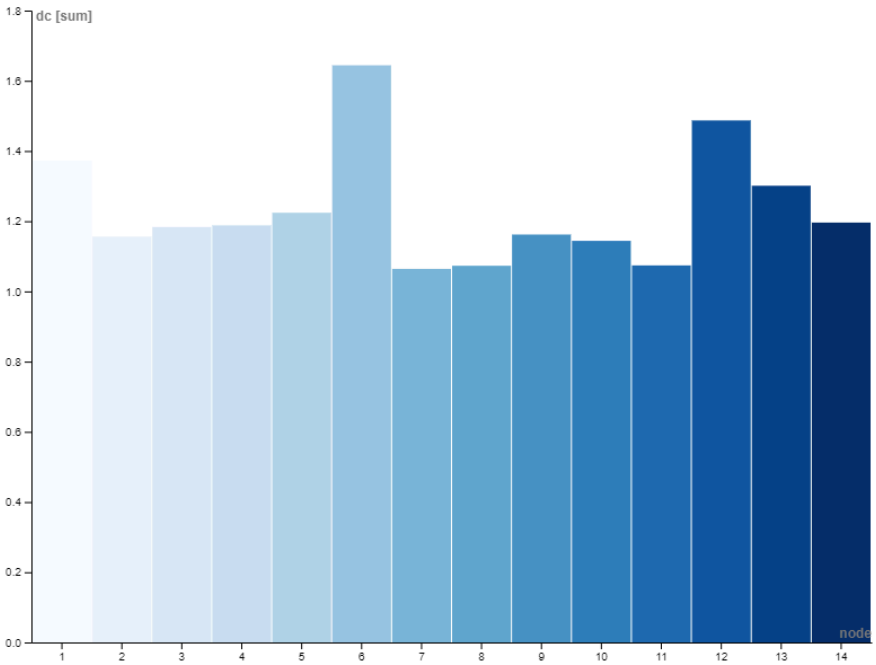


Figure 7.3: Scenario 1 | DC

7.1.1.2 CPU

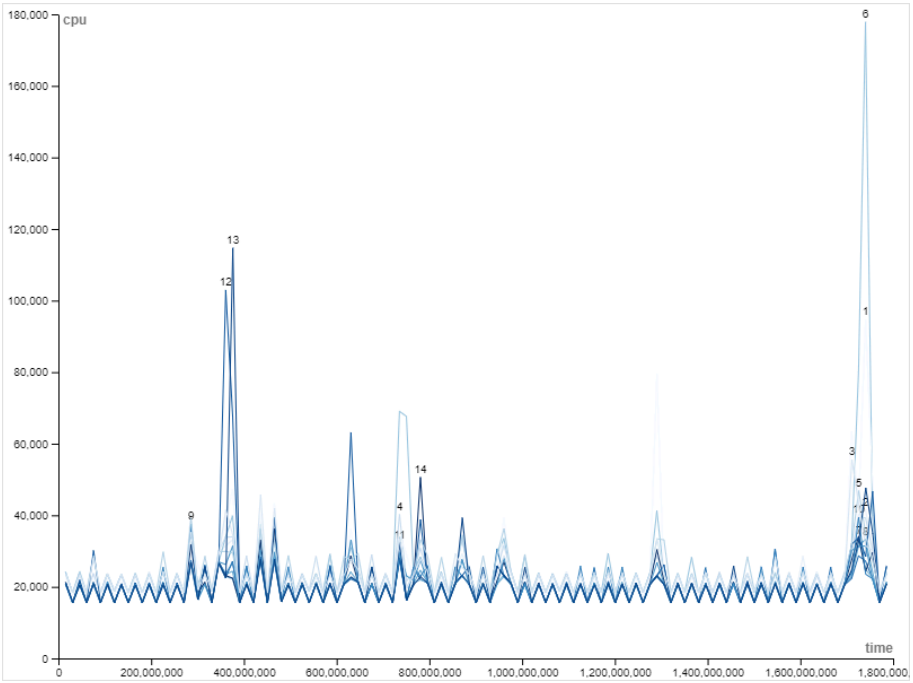


Figure 7.4: Scenario 1 | CPU

7.1.1.3 LPM

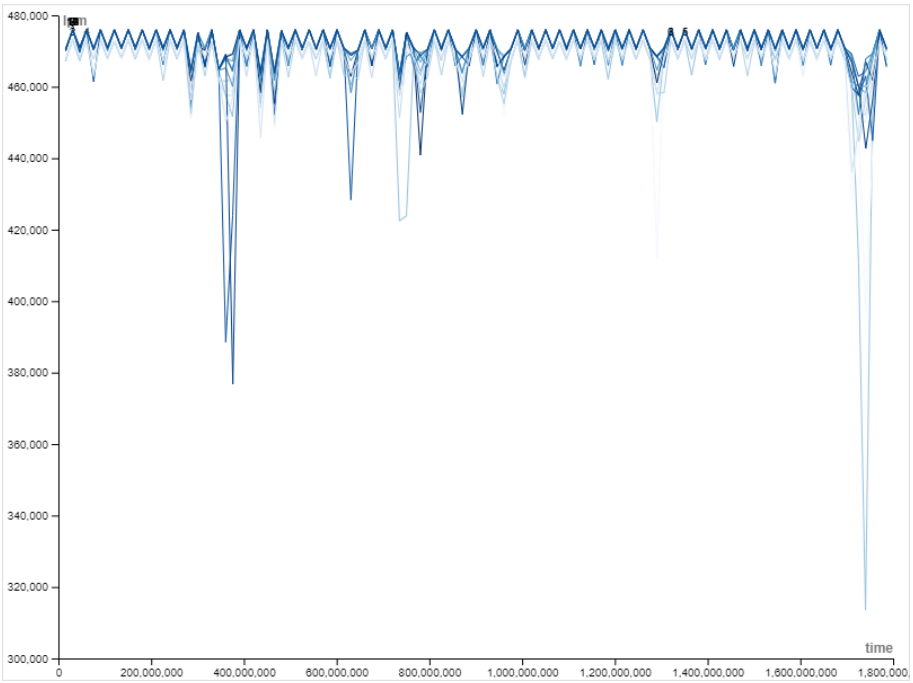


Figure 7.5: Scenario 1 | LPM

7.1.1.4 TX

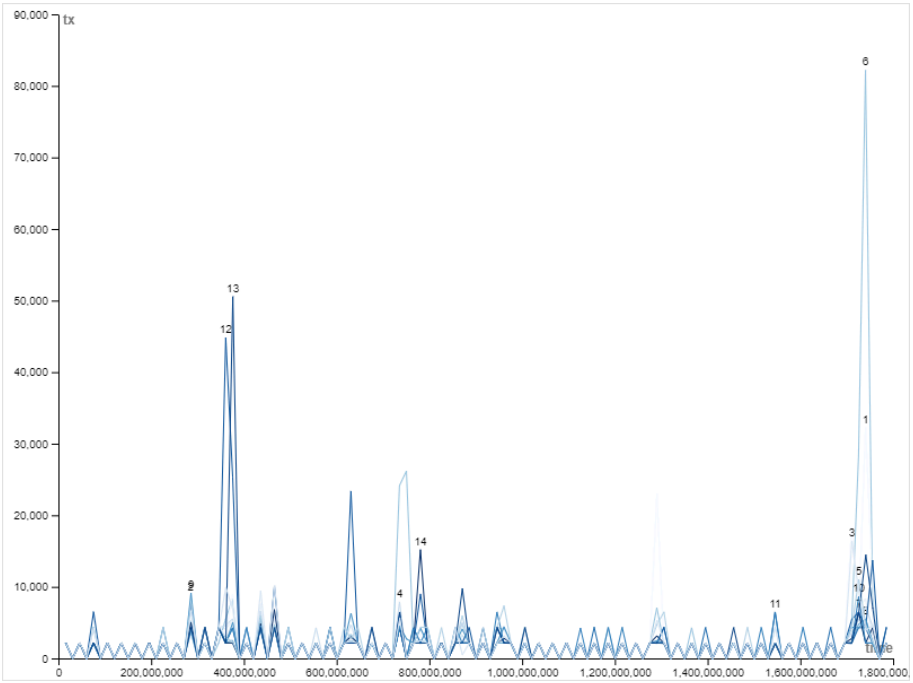


Figure 7.6: Scenario 1 | TX

7.1.1.5 RX

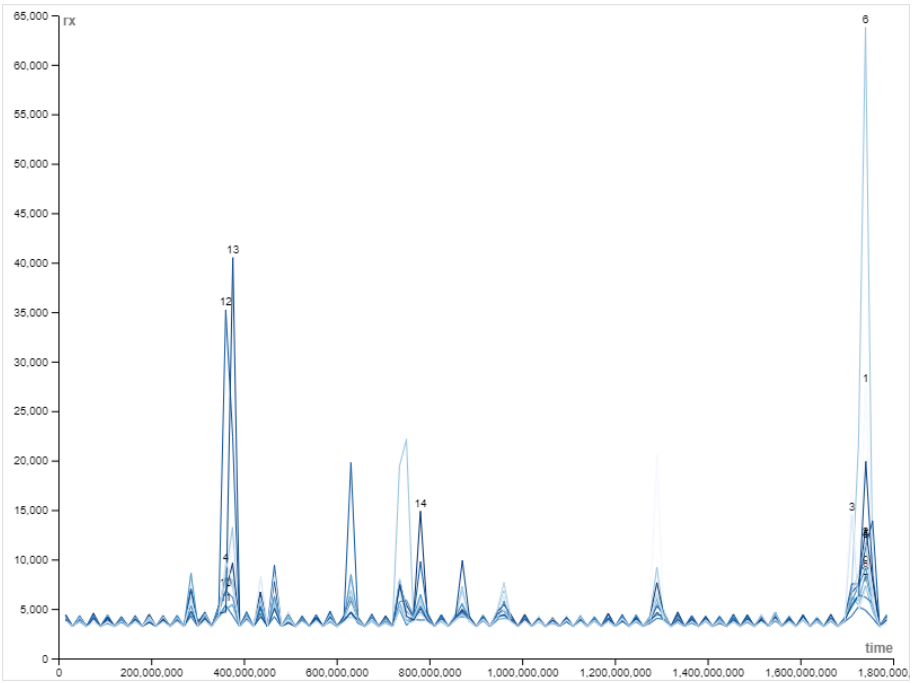


Figure 7.7: Scenario 1 | RX

7.2 Scenario 1 | Node 9 Failure

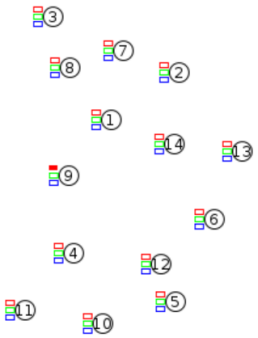


Figure 7.8: Scenario 1 | Node 9 Failure

In this scenario we are simulating the failure of node `9`. Duration is set to 10 minutes.
We can clearly see that without node `9` there is no easy communication between the nodes that are in the lower part and the *Controller*. The majority of the communication must go through node `12`, node `4` and node `6`. This is direclty reflected in the power consumption of the nodes compared to the one without failures.
The PDR is lower than 100% even tough all collect messages has been received by the *Controller*. This is due to the fact that the *Controller* has a collect timer that is triggered before the reception of some messages. Some failures and buffering can increase the overall delivery time of the messages of nodes that are from the lower part.
There is a high probability that node `12` die soon than the others leaving the lower part without any command message but only triggers on every reading.

7.2.1 Statistics

```
amespace(logfile='loglistener.log', testbed=False)
Logfile: loglistener.log
Cooja simulation

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

AVERAGE DUTY CYCLE: 1.392%
STANDARD DEVIATION: 0.578
MINIMUM: 0.000%
MAXIMUM: 2.876%

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

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

COLLECT PDR: 0.8857

# COMMANDS GENERATED BY THE CONTROLLER: 6
# COMMANDS RECEIVED BY ACTUATORS: 6
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 7.9: Scenario 1 | Node 9 Failure | Statistics

7.2.1.1 DC

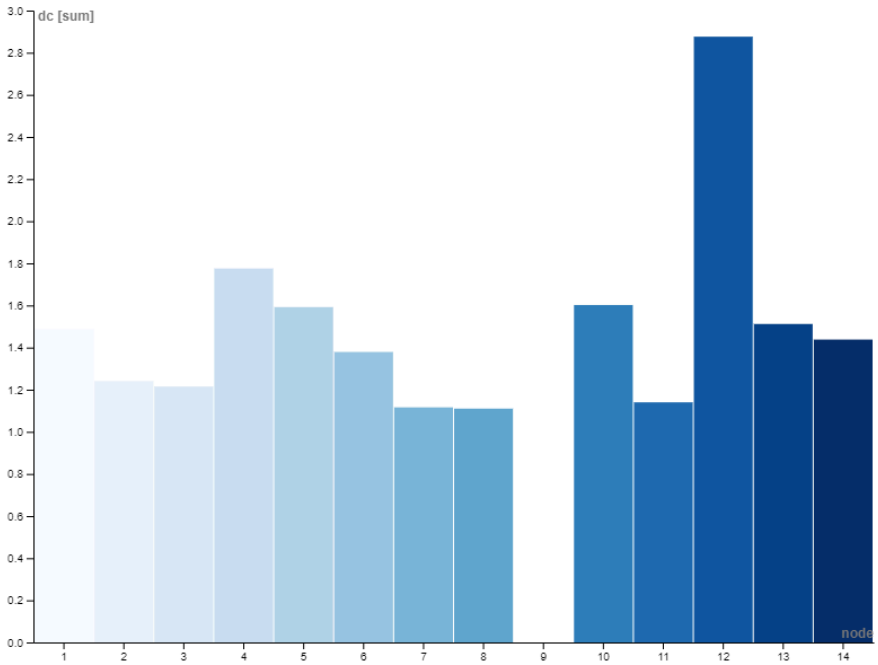


Figure 7.10: Scenario 1 | Node 9 Failure | DC

7.2.1.2 CPU

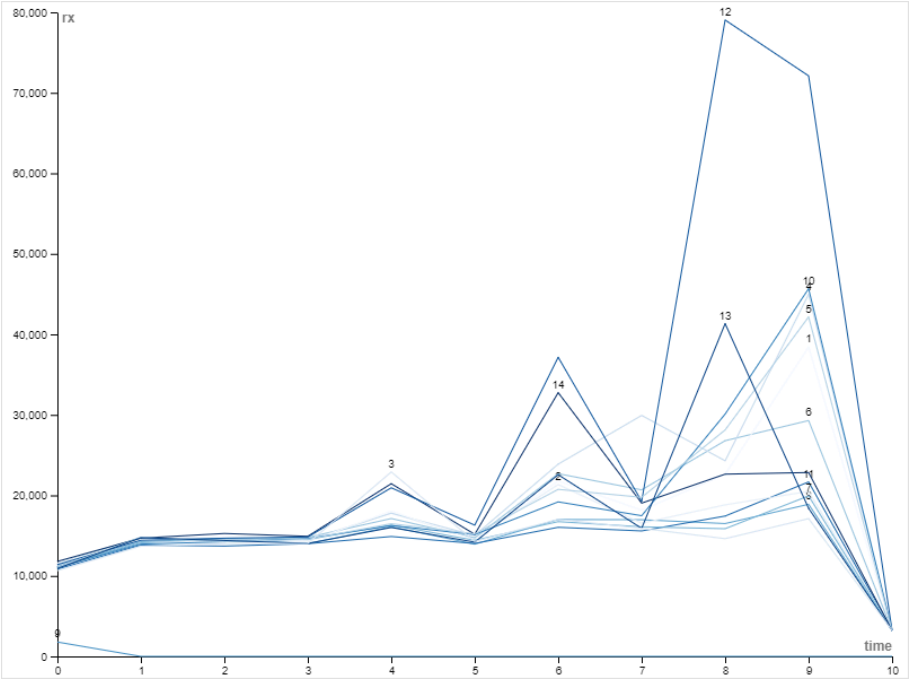


Figure 7.11: Scenario 1 | Node 9 Failure | CPU

7.2.1.3 LPM

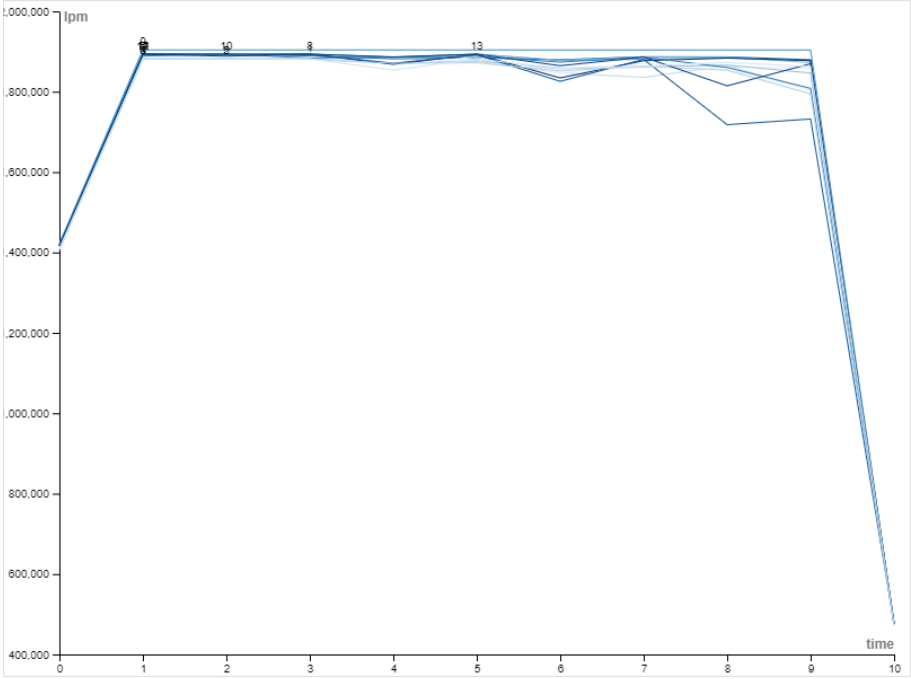


Figure 7.12: Scenario 1 | Node 9 Failure | LPM

7.2.1.4 TX

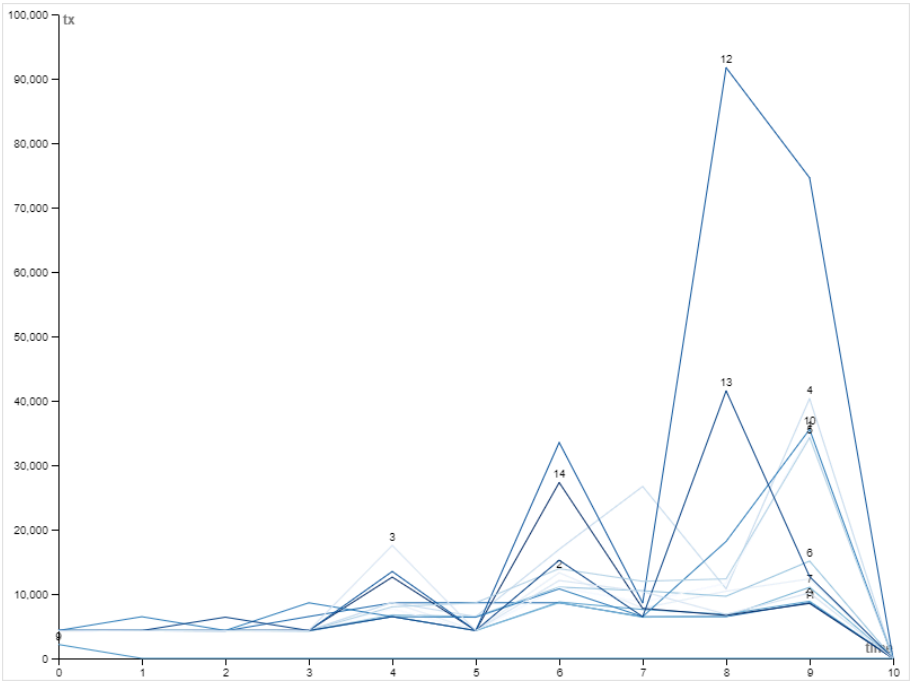


Figure 7.13: Scenario 1 | Node 9 Failure | TX

7.2.1.5 RX

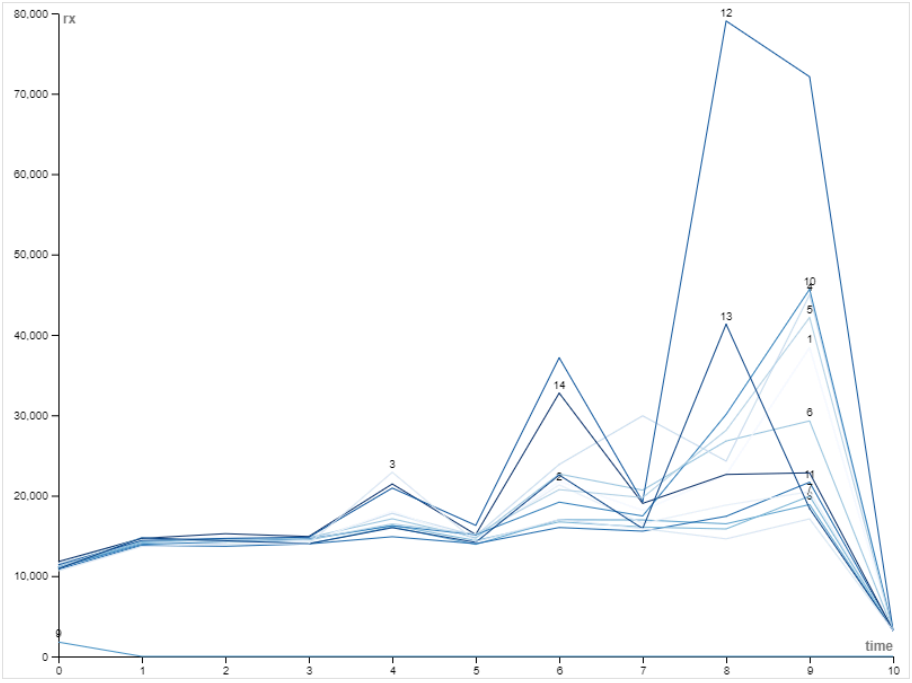


Figure 7.14: Scenario 1 | Node 9 Failure | RX

7.3 Scenario 2

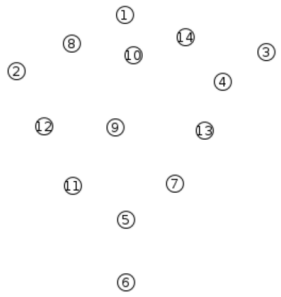


Figure 7.15: Scenario 2.

In this scenario we can clearly see that node `9` is heavily used thanks to its strategic position. Its in the middle, so it can reach the *Controller* node with a low number of hops and can reach a huge number of nodes with a single broadcast message. Moreover, propagating the information to other nodes increase the probability to be chosen as a parent node from the other nodes. In fact, this is reflected by the overall power consumption data.
The scenario 2 is also affected by some failures as in scenario 1. We can clearly see that nodes `6, 9, 7, 4, 1, 5, 10` are affected. Most likely this is due to collisions. Despite these issues that required some nodes retry and/or recover from failures, all the messages have been delivered correctly reporting a PDR or 100%.
Note that even though node `9` can be chosen from a tree (re)construntion it's not said that is the most efficient one. It's heavily used, so the probability of collisions is higher than other nodes. Moreover, if node `9` failed to send a message to all the other buffered messages need to wait all the recovery procedure, this takes an undefined number of resources and time that must be accurately calibrated.

7.3.1 Statistics

```
Namespace(logfile='scenario2_nogui_mrm.log', testbed=False)
Logfile: scenario2_nogui_mrm.log
Cooja simulation

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

AVERAGE DUTY CYCLE: 1.180%
STANDARD DEVIATION: 0.121
MINIMUM: 1.086%
MAXIMUM: 1.567%

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

# EVENTS AT CONTROLLER: 12
# COLLECT ROUNDS AT CONTROLLER: 12
# 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 7.16: Scenario 2 | Statistics

7.3.1.1 DC

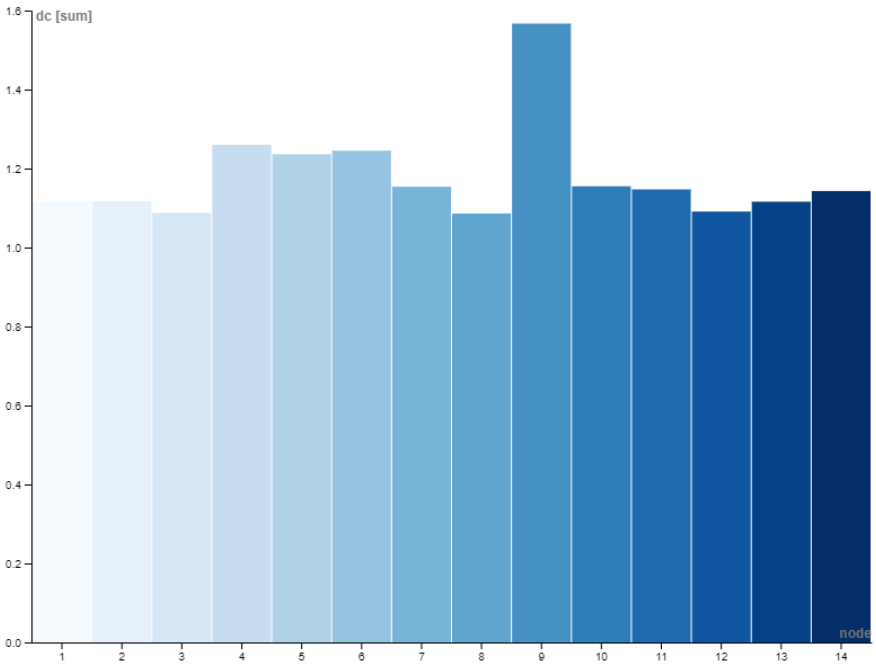


Figure 7.17: Scenario 2 | DC

7.3.1.2 CPU

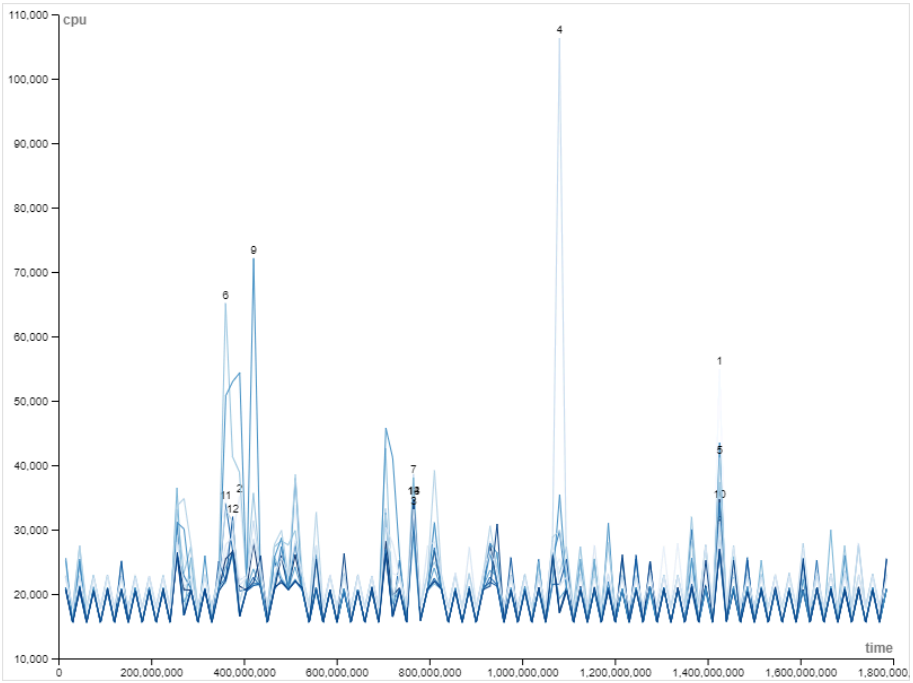


Figure 7.18: Scenario 2 | CPU

7.3.1.3 LPM

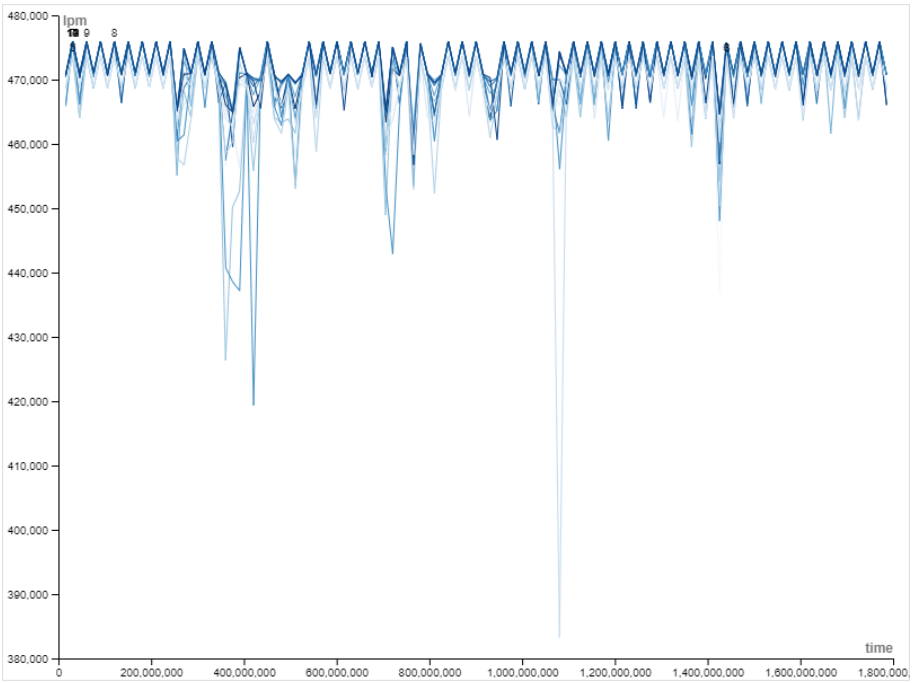


Figure 7.19: Scenario 2 | LPM

7.3.1.4 TX

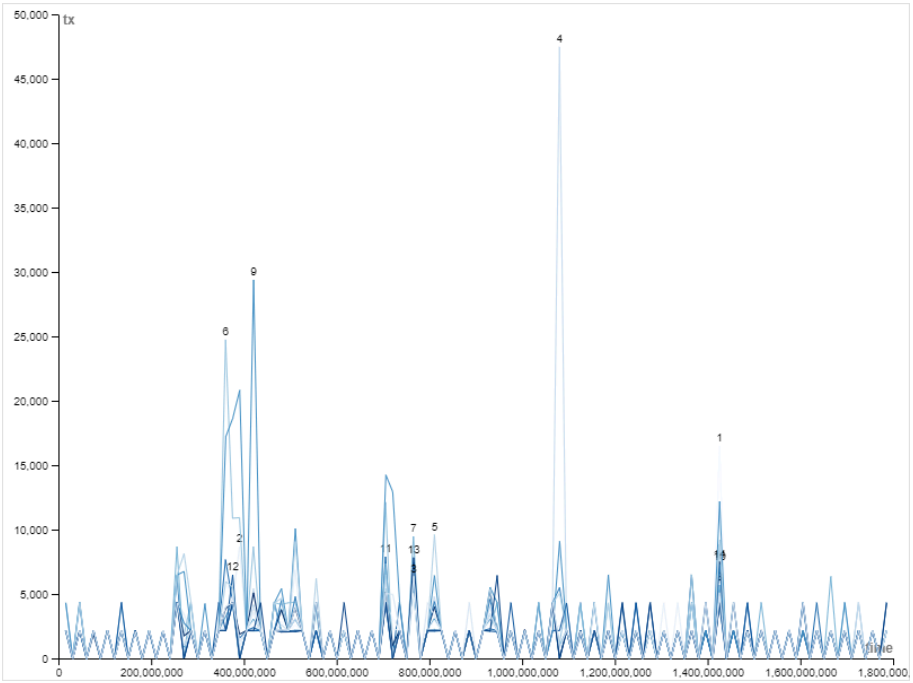


Figure 7.20: Scenario 2 | TX

7.3.1.5 RX

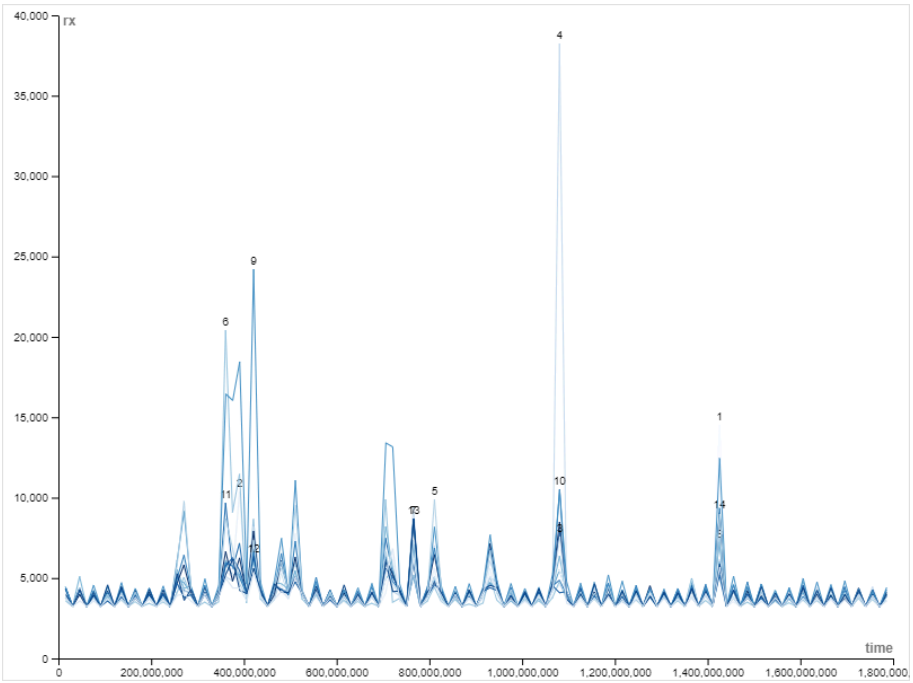


Figure 7.21: Scenario 2 | RX

7.4 Scenario 2 | Node 9 Failure

Figure 7.22: Scenario 2 | Node 9 Failure

In this scenario we are simulating the failure of node 9. Duration is set to 10 minutes.
We can clearly see that without node 9 the heavy lifting is done by node 5. Its power consumption is huge compared to other nodes.
In this scenario the statistics are quite good with a PDR of 100%. This is due to the fact that the network traffic of the lower part is balanced on the left and right sides More or less a V shaped network. Moreover, node 6 is heavily dependent by node 5. If node 5 dies there is a high chance that some collect messages are lost.
Even in this scenario there are some issues during communications that required the nodes to promptly react to them. For sure, node 5 has done some retransmission and (re)evaluation of the parent node.
Note that this time node 9 has been turned off more or less at minute 4. This is clearly seen in the TX | RX graphs where the line goes down to 0.

7.4.1 Statistics

```
Namespace(logfile='loglistener.log', testbed=False)
Logfile: loglistener.log
Cooja simulation

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

AVERAGE DUTY CYCLE: 1.259%
STANDARD DEVIATION: 0.273
MINIMUM: 0.802%
MAXIMUM: 2.145%

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

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

COLLECT PDR: 1.0

# COMMANDS GENERATED BY THE CONTROLLER: 6
# COMMANDS RECEIVED BY ACTUATORS: 6
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 7.23: Scenario 2 | Node 9 Failure | Statistics

7.4.1.1 DC

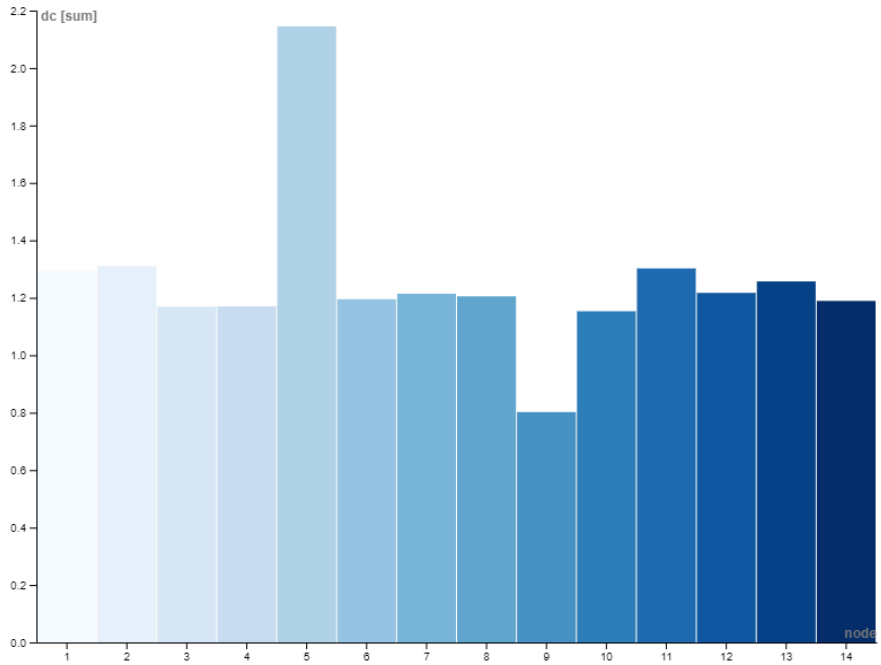


Figure 7.24: Scenario 2 | Node 9 Failure | DC

7.4.1.2 CPU

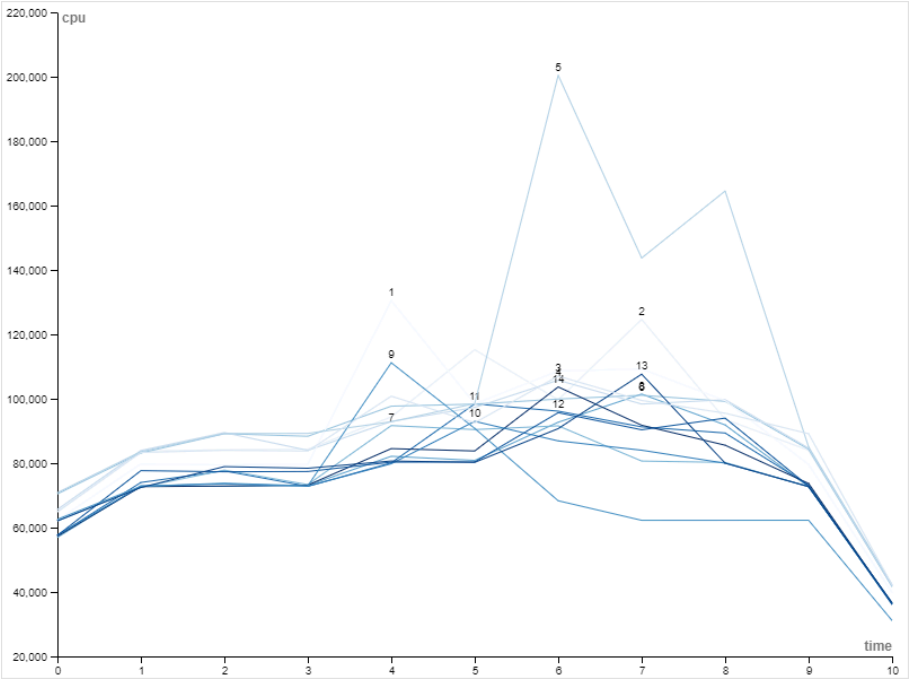


Figure 7.25: Scenario 2 | Node 9 Failure | CPU

7.4.1.3 LPM

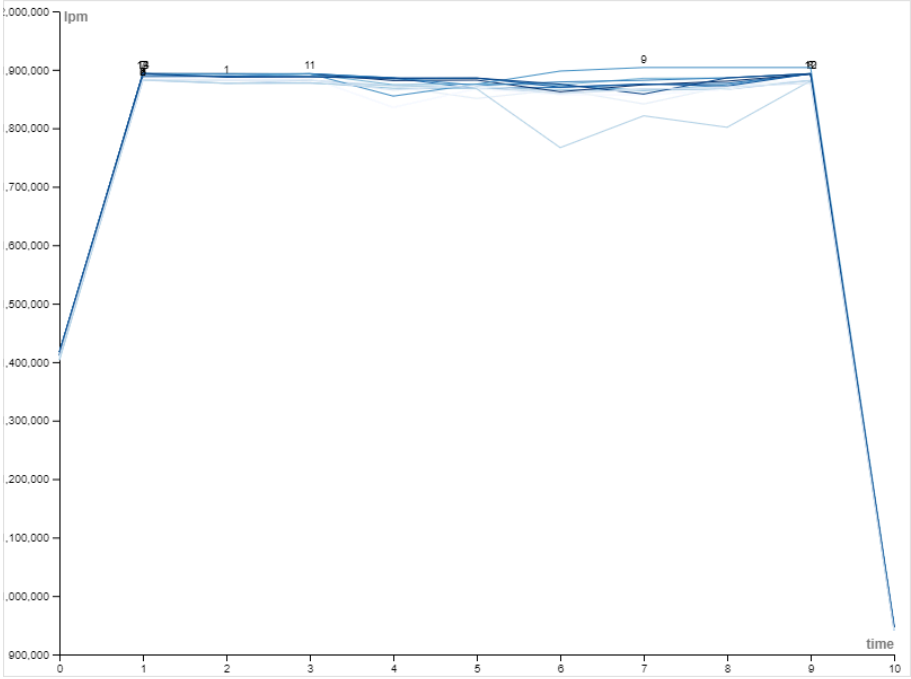


Figure 7.26: Scenario 2 | Node 9 Failure | LPM

7.4.1.4 TX

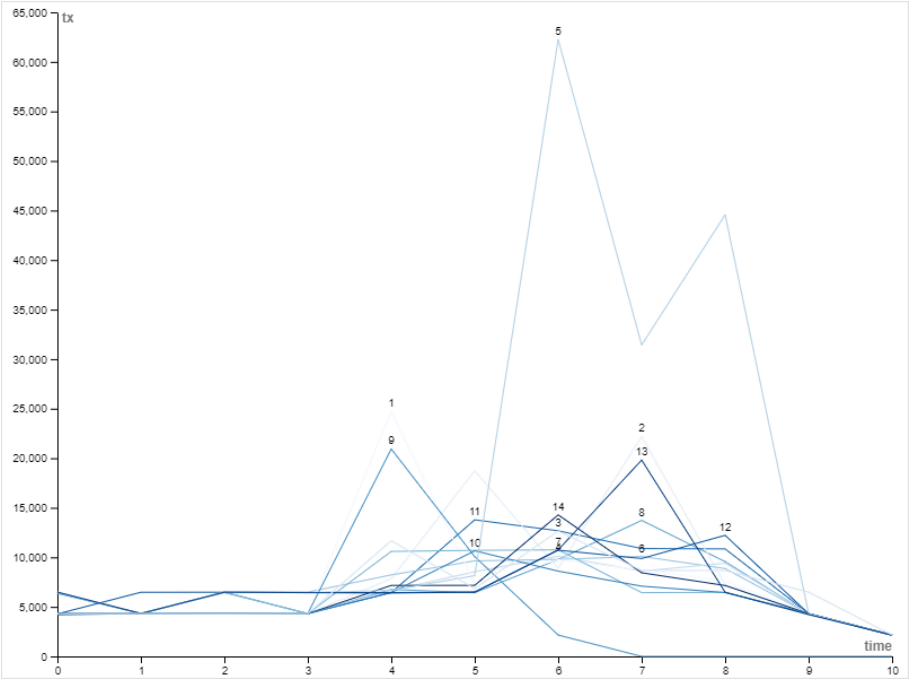


Figure 7.27: Scenario 2 | Node 9 Failure | TX

7.4.1.5 RX

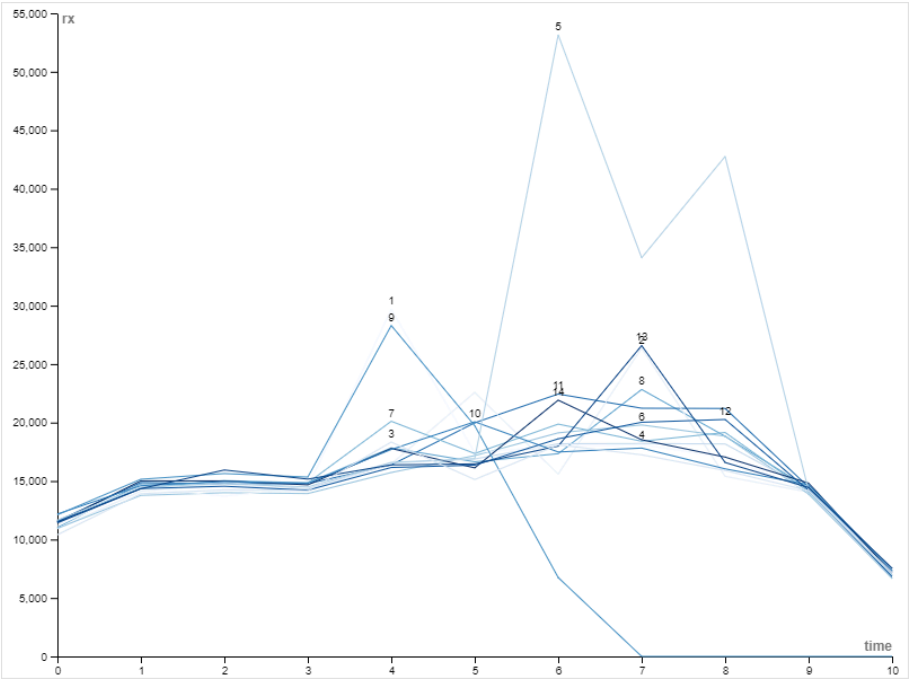


Figure 7.28: Scenario 2 | Node 9 Failure | RX

7.5 Testbed

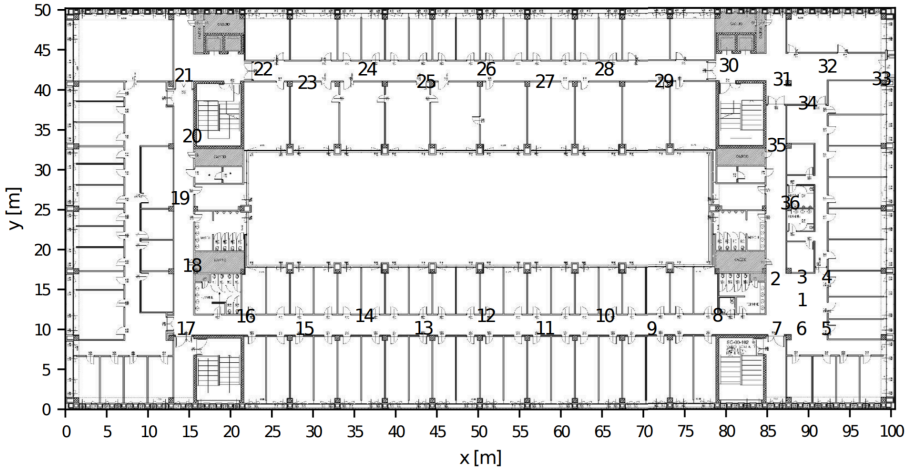


Figure 7.29: Testbed

My experiments have been conducted in the morning so the noise caused by Wi-Fi repeaters, Smartphones, Laptops, and other IT equipment have increased significantly the overall collision probability. I've launched more than 5 jobs and the PDR always fluctuated from 50% to roughly 100%. The logs showed that a lot of messages have been lost or have not been correctly received by the receiver node. Note that in these tests, the number of maximum send allowed, configuration variable `CONNECTION_UC_BUFFER_MAX_SEND`, has not been changed so the value is equal to 1.
As a last chance, I've increased the maximum number of send allowed to 3. Note that in the previous tests, when failing to send a message to the parent node (and the parent is not in the *Controller*), the recovery procedure automatically started removing the current one and trying to obtain a backup parent. Evaluating the scenario of the testbed we can clearly see that the nodes are not randomly sparse like in *Cooja*, increasing the chances of a good multi's parent, but they are inline so the chances of getting a good multi parent architecture are lower. Moreover, as wrote before, starting the recovery procedure as soon as a message has not been correctly delivered increases the losses, reducing the overall PDR. Event though, retrying can increase collection time, undoubtedly increase the overall reliability and strengthness of the application.

One of the major issues of using the testbed is the queue. Sometimes I've waited hours before my job is executed. This report should have been delivered by Monday, however my job has been scheduled for Tuesday's morning, forcing me to postpone it. Sometimes, I was forced to wait for at least one hour because someone before me have scheduled a job that is too long.
The second *"issue"* (that is not a real issue) is that real-time simulations takes real time to evaluate and not seconds/minutes like a simulated scenario in *Cooja*. A testbed is super useful for testing an application in real-world scenarios, but it's super stressful. Sometimes I've made stupid errors and I was forced to re(submit) the job waiting again.
One last minor issue is that the logger does not work in the testbed. I've tried many countermeasures but none of them worked. I don't know why, maybe the zoul platform does not correctly handle variadic functions or?
Nevertheless, I'm very happy that my application has been tested not only in a simulated (boring) scenario but also in a real one. In the previous years I've always wondered what are those Rasperry Pi sockets attached to the ceiling around the University. Today, I know their purpose and how to use them!
Note that in the testbed no failures have been tested since I'm not able to physically turn them off even tough it can be a really cool feature.

7.5.1 Preparation

1. Clean
   `make cleanall`

2. Build/Compile
   `make TARGET=zoul`
   To enable statistics:
   `make TARGET=zoul STATS=true`

7.5.2 Schedule

Schedule the experiment to the testbed:

1. Change directory
   `cd scenarios/testbed`

2. Run test
   `./run-test.sh`
   Remember the job_id!

7.5.2.1 experiment.json

To configure the testbed we need an `experiment.json` file.
The following is the configuration used in the testbed:

```
{
  "name": "Event-triggered control -- LPIoT Project Course",
  "description": "Zolertia Firefly (CC2538) event-triggered control protocol -- LPIoT Project Course",
  "start_time": "asap",
  "duration": 480,
  "logs": 0,
  "binaries": {
    "hardware": "firefly",
    "bin_file": "../../app.bin",
    "targets": "disi_povo2",
    "programAddress": "0x00200000"
  }
}
```

Figure 7.30: Testbed | experiment.json

You can find the configuration file under the `scenarios/testbed` folder. For more information on how to configure a job please visit `https://testbediot.disi.unitn.it/frontend/docs/jobFile.html`

7.5.3 Logs

Retrieve the logs produced during the testbed execution.

1. Get test
   `./get-test <JOB_ID>`

2. Change directory
   `cd job_<JOB_ID>`

3. Log file is available: `test.log`

7.5.4 Statistics

```
Namespace(logfile='test.log', testbed=True)
Logfile: test.log
Testbed experiment

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

AVERAGE DUTY CYCLE: 1.499%
STANDARD DEVIATION: 0.237
MINIMUM: 1.247%
MAXIMUM: 2.275%

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

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

COLLECT PDR: 0.9

# 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 7.31: Testbed | Statistics

7.5.4.1 DC

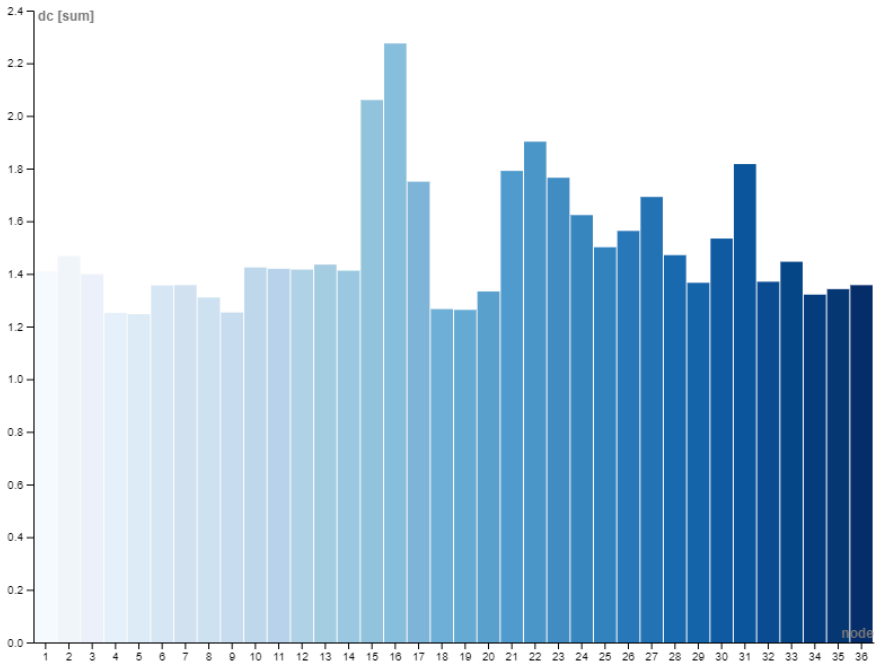


Figure 7.32: Testbed | DC

7.5.4.2 CPU

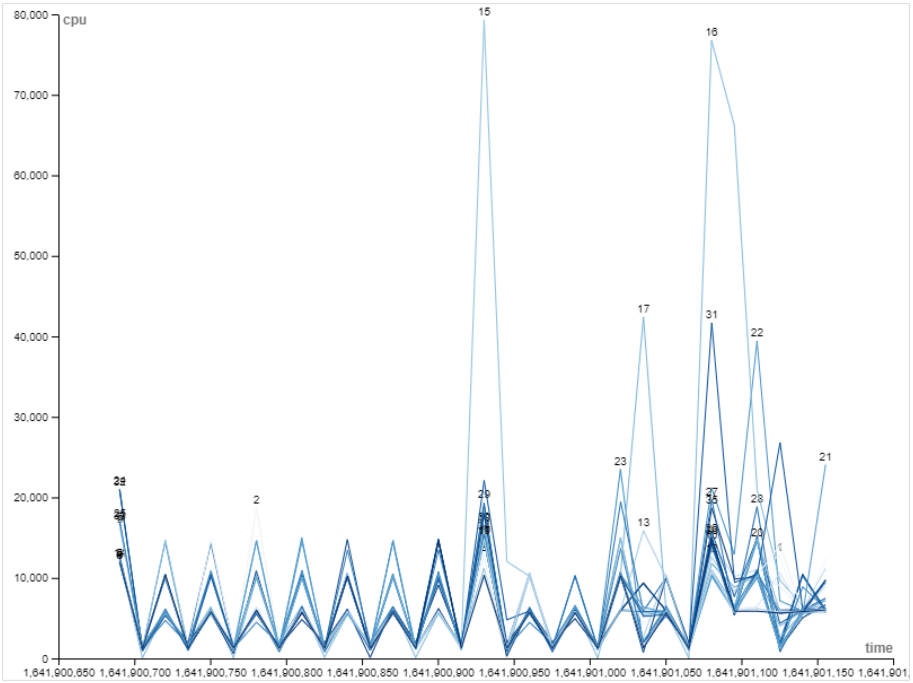


Figure 7.33: Testbed | CPU

7.5.4.3 LPM

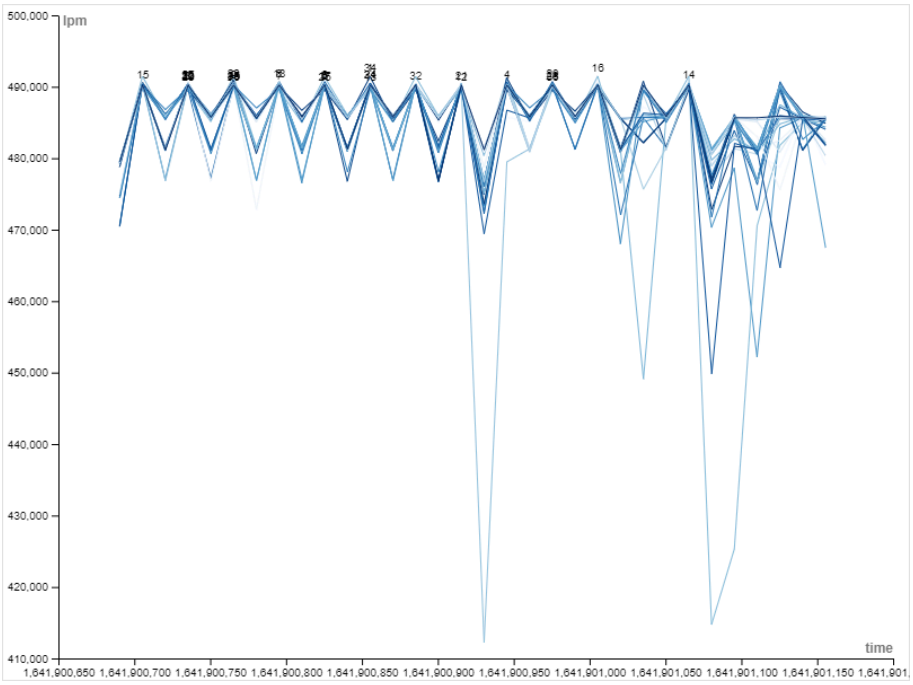


Figure 7.34: Testbed | LPM

7.5.4.4 TX

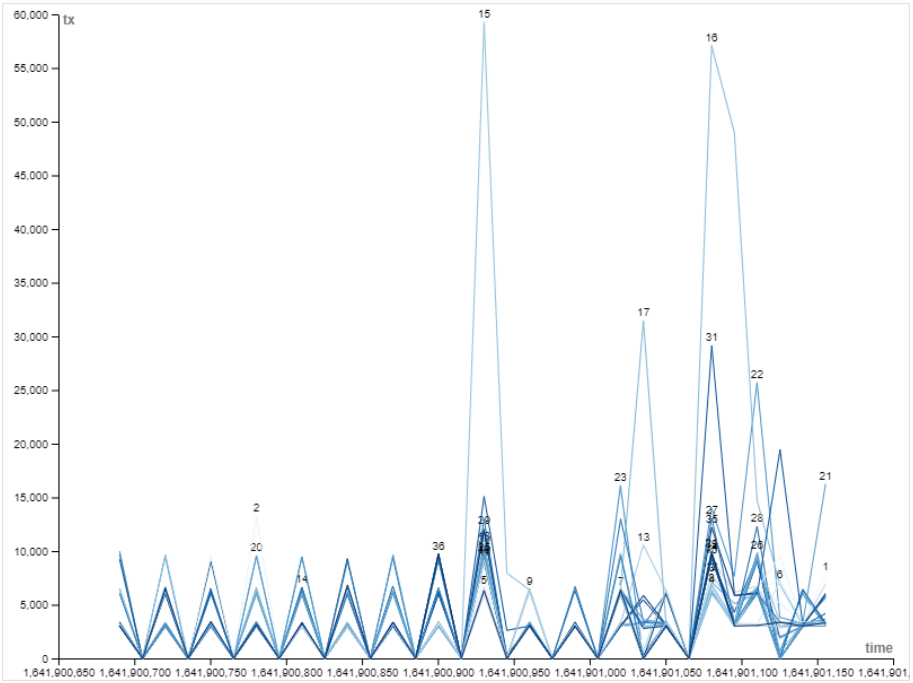


Figure 7.35: Testbed | TX

7.5.4.5 RX

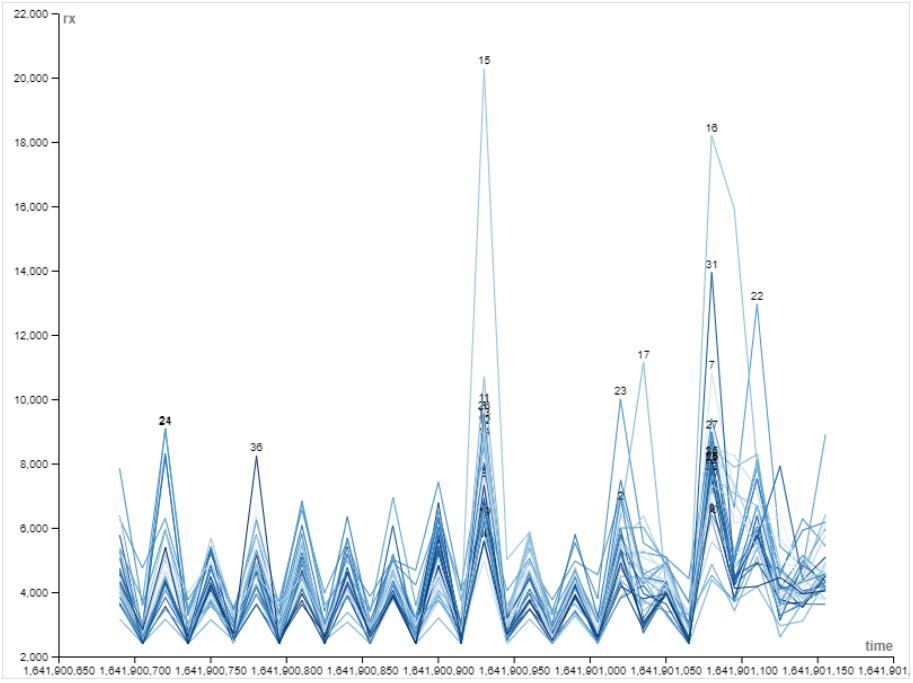


Figure 7.36: Testbed | RX

8 Issues

8.1 Triggers Same Instant

This problem happens when two triggers (events) occurs at the same time. More frequently when the distance between the two *Sensor*(s) are significant. The second trigger is not suppressed by the first one, because the propagation of the event message arrives after the trigger.
By the design, when the propagation of an event occurs, it's saved and "locked" as the current handled event for a certain amount of time (propagation-timer). If a node receives a message that is related to an event, and the handled event is "locked", and it does not correspond to the current one, the message is discarded. Note that the latter behavior is implemented not only in the *Controller*, but in every single node to remove useless communications and reduce energy consumption.
Since some messages are discarded due to an event mismatching the *Controller* reports a PDR that is less than the total number of *Sensor*(s). Moreover, is more likely that the *Controller* handles the event from the nearest *Sensor* rather than the one that are far away and take more time to propagate.
This behavior could also occur in a real industrial scenario and there should be a mechanism to properly handle both of them and not only a single one.

8.2 No Event At Sensor

If an event message does not arrive to a *Sensor* node, no collect message is sent causing a PDR that is less than the total number of *Sensor*(s).
Most of the time this is caused by a collision with another node that it too was propagating the event message. This issue happens sometimes in the simulated scenario 2 on the *Sensor* node 6. Following the logs it's easy to spot that the node doesn't receive any event message and continues doing the readings without preparing for a propagation/collect.
This behavior could also occur in a real industrial scenario and there should be a mechanism to properly know if a *Sensor* have not received the event.

8.3 Collect Message After Timeout

This issue is very rare, but sometimes a collect message arrives to the *Controller* node after the collect timer has triggered causing the message to be discarded.
The proper way to handle this situation is to test the scenario on which our network is operating and calibrate the collect timeout accordingly to the results (choose the highest reception or a mean of all).

8.4 Cycles

In an environment that is based on networking, cycles are inevitable and there should be some mechanisms to detect them and act accordingly. Note that a cycle only happens when using the Rime unicast primitives and does not involves any broadcast communication.
There are two possible ways to handle a cycle in the application:

1. `Discarding`
   A cycle is detected and there are no mechanisms to reroute the message in a correct way. The message is discarded and a warning log is shown.
   This is achieved by a hop counter, similar to the one in TCP/IP stack, where on reception, the counter (present in the header) is increased by 1, and if reaches a predefined threshold it's discarded.

An example of a scenario is when there are three nodes: the first node is connected to the second, the second to the third and the third to the first. Currently, there is no logic to detect this, and the message enters in a loop until the counter reaches the threshold and is discarded.

2. `Rerouting`
   In some cases a cycle is detected and the message is rerouted to another node. Obviously, some constraints must be respected to correctly detect them but as far as I've seen in the simulations it works.
   As an example, when a node receives a collect message and the sender is it's parent something strange is going on. Therefore, the collect message is not sent to the current parent but to the backup parent (if any) removing the possible loop and rerouting the packet to a node that should be good for the delivery.

8.5 Wrong Parent Node

By design, every `X` seconds the *Controller* sends a new beacon message to reconstruct the communication tree and upon reception is automatically chosen because the sequence number is higher than the previous one.
Sometimes it happens that due to collisions the messages from the nodes that are above the current node (closer to the *Controller*) are not received but messages from nodes that are under the current node are received causing to overwrite the old (and possibly correct) parent. This could cause cycles and mechanisms to handle them are already described above.
Note that in the beacon protocol an overflow of the sequence number (`seqn=0`) is handled and treated as a new beacon message.

8.6 Node Slowdown

In *Cooja* simulations, very rarely, happens that a node is affected by severe slowdowns. Unfortunately I don't know the direct cause, but it could be that the execution of a thread is postponed too much until it reaches a critical threshold in the kernel and it's finally handled.
I've seen this in the logs when a node (specifically a *Sensor*) tries to send a unicast message. Following the timestamps (start sending and sent notification) the time elapsed can reach more than five seconds causing a generalized delay in the communications.
Due to this, a collect message could reach the *Controller* node after the collect timer triggered causing the message to be discarded and reducing the overall PDR.

8.7 Beacon & Event

If an event message is propagated during the tree-reconstruction phase the number of overall collisions increases significantly. This is due to the fact that both of them are based on the Rime broadcast primitive that spread the communication to every node that can capture it.
In some cases, to prevail are the beacon messages and in other cases are the event messages. There should be a balance to choose between having an updated tree or the latest sensor readings. This behavior could also occur in a real industrial scenario and there should be a mechanism to properly handle this. A possible solution is to recognize that an event propagation is occuring and delay the tree reconstruction.

9 Future Improvements

There are three main aspects that can be improved in the application:

Add support for multiple *Controller* nodes in combination with an algorithm to synchronize the *Sensor*(s) readings across the network. Increase the reliability thanks to redundancy and reduce the overall network traffic thanks to collect messages gathered only from the nearest *Controller*(s).

Reduce the number of collisions maintaining an acceptable power consumption with a direct result into increasing the number of PDR and maintaining an updated tree.

Develop and/or include a better tree construction algorithm rather than a beacon/timer architecture. One solution is to update the tree only when a failure is detected but other possibilities are available. In my opinion, this is one of the hardest, but most important, aspect of the application since a flood of beacon messages in the entire network not only is costly in terms of time and power consumption but can also interfere with more important events such as *Sensor*(s) readings.

Last but not least, every new feature must be carefully analyzed and tested because can introduce new bugs/issues and can reduce the lifetime of a node due to higher power consumption.

Bibliography

[1] Contiki contributors. The Contiki Operating System. `https://github.com/contiki-os/contiki`.

[2] Doxygen contributors. Doxygen. `https://www.doxygen.nl`.

[3] GNU Make contributors. GNU Make. `https://www.gnu.org/software/make`.