A decorative network diagram in the top-left corner, featuring a cluster of interconnected nodes. Some nodes are represented by solid grey circles, while others are open circles with a smaller solid circle inside. They are connected by thin grey lines, some of which are solid and others dashed.

Internet of Things Lab

RPL: Routing Protocol for Lossy networks

Romeo Velvi
0120000304

A decorative network diagram in the bottom-right corner, similar to the one in the top-left. It shows a cluster of interconnected nodes, with some solid grey circles and some open circles containing smaller solid circles, connected by thin grey lines.

RPL - Intro

The RPL acronym of Routing Protocol for Lossy networks strategy was designed for:

- Handling the power consumption.
- Reducing congestions over the network.

Moreover, this protocol implements a **time-window** parameter that:

- ❖ Limits the forwarding of messages on a “expired” criterion.
- ❖ Control the amount of message retransmitted.
- ❖ Decrease the number of “orphan package” in loop.

Power Consumption Problem

Energy consumption in IoT devices, especially sensors, is a critical concern.

The main factors responsible for the power drain are essentially:

- Transmission
- Reception

So, the great percentage of power supply goes to the **bad communication management**.

Congestion Problem

By definition, a congestion refers to a situation where the network becomes overloaded due to an excessive amount of data traffic.


This happens when multiple devices or sensors send data simultaneously.

This brings the network's capacity to exceeds, leading to:

- Delays
- Packet loss
- Degraded performance



Presentation Topics

1. Project Goals
 2. Entities at Stake
 3. Messages Protocol
 4. Optimization Strategies
 5. Pseudo-code Idea
 6. CupCarbon Simulation
 7. Energy Considerations
 8. Improvements and criticalities
- 



Presentation Topics

1. **Project Goals**
 2. Entities at Stake
 3. Messages Protocol
 4. Optimization Strategies
 5. Pseudo-code Idea
 6. CupCarbon Simulation
 7. Energy Considerations
 8. Improvements and criticalities
- 


Project Goals

The goal of this project is to implement the RPL protocol with the time-window expiration parameter.

Moreover, for the sake of efficiency there are also presented additive performance and optimization strategies.



Presentation Topics

1. Project Goals
 2. **Entities at Stake**
 3. Messages Protocol
 4. Optimization Strategies
 5. Pseudo-code Idea
 6. CupCarbon Simulation
 7. Energy Considerations
 8. Improvements and criticalities
- 


Entities at stake

The RPL protocol requires two kind of different entities:

- **ClockSync Node:** responsible to mark the time-window parameters. Moreover, It also decide what sensor can transmit the data. For the project's sake, we can assume that it doesn't have any power limitation due to its relevance (e.g. UPS).
- **Sensor Node:** which can be seen as classic sensor that read and transmit data. In order to work with RPL protocol, they must be phased with ClockSink descision in order to reduce the congestions.



Presentation Topics

1. Project Goals
 2. Entities at Stake
 3. **Messages Protocol**
 4. Optimization Strategies
 5. Pseudo-code Idea
 6. CupCarbon Simulation
 7. Energy Considerations
 8. Improvements and criticalities
- 

Messages Protocol

In order to implement efficiently the RPL protocol, we need to define a few different kind of messages.

- Sync Message sent by the ClockSync Nodes.
- Sensor Message sent by Sensor Nodes.
- Ack Message (for [OTA method](#)) always sent by Sensor Nodes built in order to diversify the package kinds and apply complexity to the model.

Messages Protocol – Basic Structure

The basic general structure is always defined by:

Kind	Other informations...	Hops
------	-----------------------	------

The ***Kind*** part refers to the type of message transmitted/received and can assume three possible values:

- «SYN» for the Sync Message
- «MSG» for the Sensor Message
- «ACK» for the Ack Message

The ***Hops*** part refers to [HPD optimization](#) explained later.

Messages Protocol – Sync Message Pt.1

The *sync message* sent by the ClockSync nodes has this structure:

Kind="SYN"	PackVersion	ChosenSlot	NumberOfSlots	LimitTime	ClockSyncID	...SimulationVariables...	Hops
------------	-------------	------------	---------------	-----------	-------------	---------------------------	------

- **PackVesion**: reports the version of the Sync message sent in order to let the sensor nodes be phased with the new directives decided by the ClockSync nodes.
- **ChoosenSlot & NumberOfSlotS**: are variables needed for [Slot-Based Execution Mechanism](#) explained later.
- **LimitTime**: is the limit time-window and tells the validity of a given message.
- **ClockSyncID**: used by the sensor nodes in order to reduce the number of trasmission through the nework.
- **Hops**: refers to [HPD optimization](#) explained later.

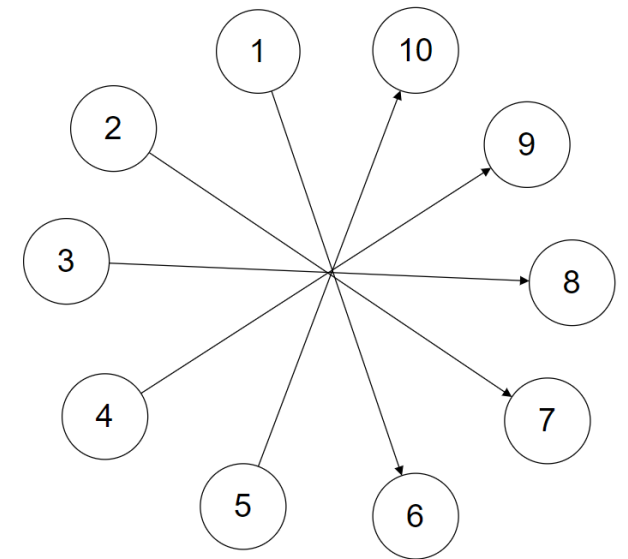
Messages Protocol – Sync Message Pt.2

The *sync message* sent by the ClockSync Nodes has this structure:

Kind="SYN"	PackVersion	ChosenSlot	NumberOfSlots	LimitTime	ClockSyncID	...SimulationVariables...	Hops
------------	-------------	------------	---------------	-----------	-------------	---------------------------	------

- The *SimulationVariables* are defined by:
 - **SensorsNumber**: The number of sensors available in the net.
 - **lowestSensorID**: The lowest id's value in the net.

This set is needed for testing case. In order to bring complexity to the model, each node uses a **mirror mechanism** for which it tries to send a message to another node (hypotetically) situated into the other specular part of the net.



Messages Protocol – Sensor Message

The *sensor message* sent by the Sensor nodes has this structure:

Kind="MSG"	SendingTime	TimeToLive	SenderID	ReceiverID	Data	Hops
------------	-------------	------------	----------	------------	------	------

- **SendingTime:** is the time when the message has been sent.
- **TimeToLive:** reports the time until the message can be forwarded.
- **SenderID:** is the message source sensor ID.
 - This is computed with the criteria mentioned before.
- **ReceiverID:** is the message destination sensor ID.
- **Data:** is the object of a message.
 - For testing scenario, the data is a random generated from a gaussian distribution.
 - It doesn't mean anything for the RPL protocol.
- **Hops:** refers to [HPD optimization](#) explained later.

Messages Protocol – Ack Message


The *ack message* sent by the Sensor Nodes has this structure:

Kind="ACK"	SendingTime	TimeToLive	SenderID	ReceiverID	Data	ReversedHops Hops
------------	-------------	------------	----------	------------	------	---------------------

- **SendingTime**: is the time when the ack has been sent.
- **TimeToLive**: reports the time until the ack can be forwarded.
- **SenderID**: is the ack source sensor ID.
- **ReceiverID**: is the ack destination sensor ID (arose from the message source sensor id).
- **Data**: is the object of the previously received message.
- **Hops**: refers to [HPD optimization](#) explained later.
- **ReversedHops**: is a concept that takes advantage of HPD method for a fast forwarding mechanism. This is explained later.



Presentation Topics

1. Project Goals
 2. Entities at Stake
 3. Messages Protocol
 4. **Optimization Strategies**
 5. Pseudo-code Idea
 6. CupCarbon Simulation
 7. Energy Considerations
 8. Improvements and criticalities
- 

Optimization Strategies

In order to increase the efficiency and the performance of the RLP protocol, some optimization strategies has been adopted:

- **Slot-Based Exclusion Mechanism**
- **HPD: Hops in Pack Data Method**
- **OTA: One-Time-Ack Application**
- **Clever Transmissions Mechanism**
- **Buffered Message for Loop Avoidance**
- **Time Relaxation on Receiver**
- **Try-and-Repeat Method**

O.S. – Slot Based Mechanism

In order to limits the whole network congestion the strategy adopted next to the time-window is the Slot Based Mechanism.

The idea is:

1. At each ΔT the ClockSync Node sends the «SYN» pack that handles the node communication. Inside this pack there are two variable **ChosenSlot** and **NumberOfSlot**.
2. Each node automatically generated a slot id (based on the **NumberOfSlot**).
3. Once the sensors receive the «SYN» pack, they have to initially check if their slot generated is equals to the **ChosenSlot** and then:
 1. If it is eqal, a sensor can send the message to another node, i.e. put message into the network.
 2. If it is not equal, a sensor can be in IDLE state (preserve power) or used as «means of transport» of messages through the net.

O.S. – Hops in Pack Data Method Pt.1

One of the problems that needs to be avoided is the «naive re-transmission».

Usually, when a node send the data to its neighbors, in order to let the message walks into the net, could happen that the information is back-propagated and the re-transmission flux is redundantly repeated.

To solve this problem, it is introduced in the protocol the HPD in the RPL protocol.

O.S. – Hops in Pack Data Method Pt.2

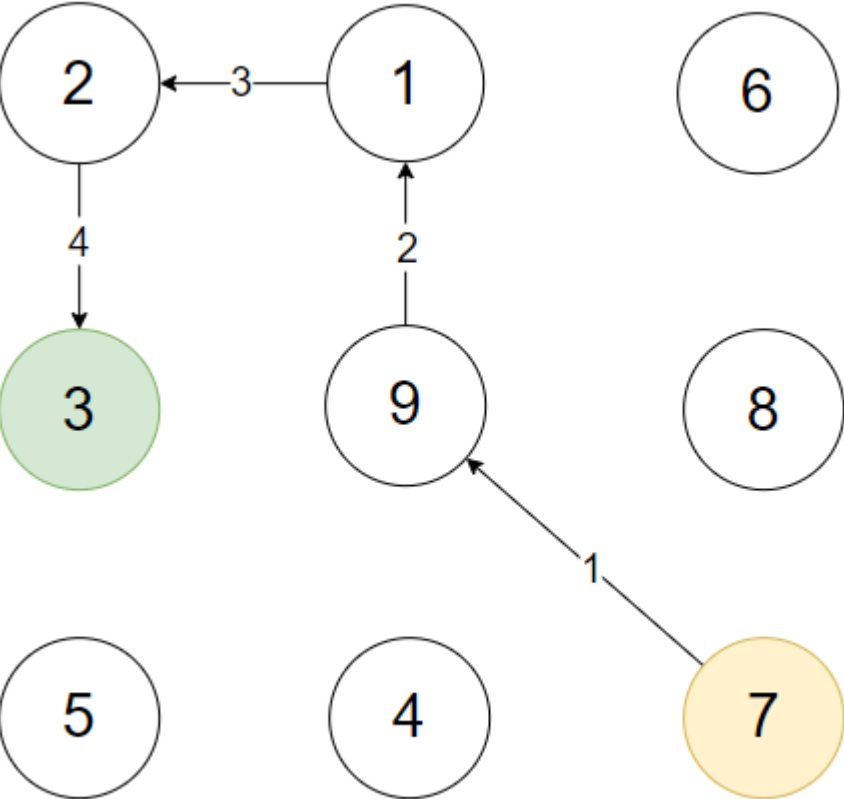
The main idea of HPD is to store some of the nodes who got this pack in a sort of FIFO/STACK data structure into the message transmitted.

Doing this, we can:

- Reduce the number of data transmitted just by filtering the nodes which already got the message.
 - Reduce the congestion and decrease the energy drain beca.
- Retrieve (portion of) of the message traversal path.
 - Also used as hint-trace for the [One-Time-Ack method](#).

O.S. – Hops in Pack Data Method Pt.3

Here is an example with three-tier stack (used in the project):



Sending a message from node 7 to node 3

STEP 1	#	#	7
STEP 2	#	7	9
STEP 3	7	9	1
STEP 4	9	1	2

The slot for the STACK is just made by three elements

O.S. – One-Time-Ack Application Pt.1

In order to diversificate as much as possible, for a more realistic way that the RLP could handle different types of the data, a new kind of message has been introduced, i.e. the ACK message.

This, as the name says, is the message sent by a destination node toward the source node to inform that the data (i.e. the «MSG») has been received.

Since we are in a lossy environment, the ACK is sent just once and it must follow the HPD stack in reverse in order to optimize the flux and reduce congestions.

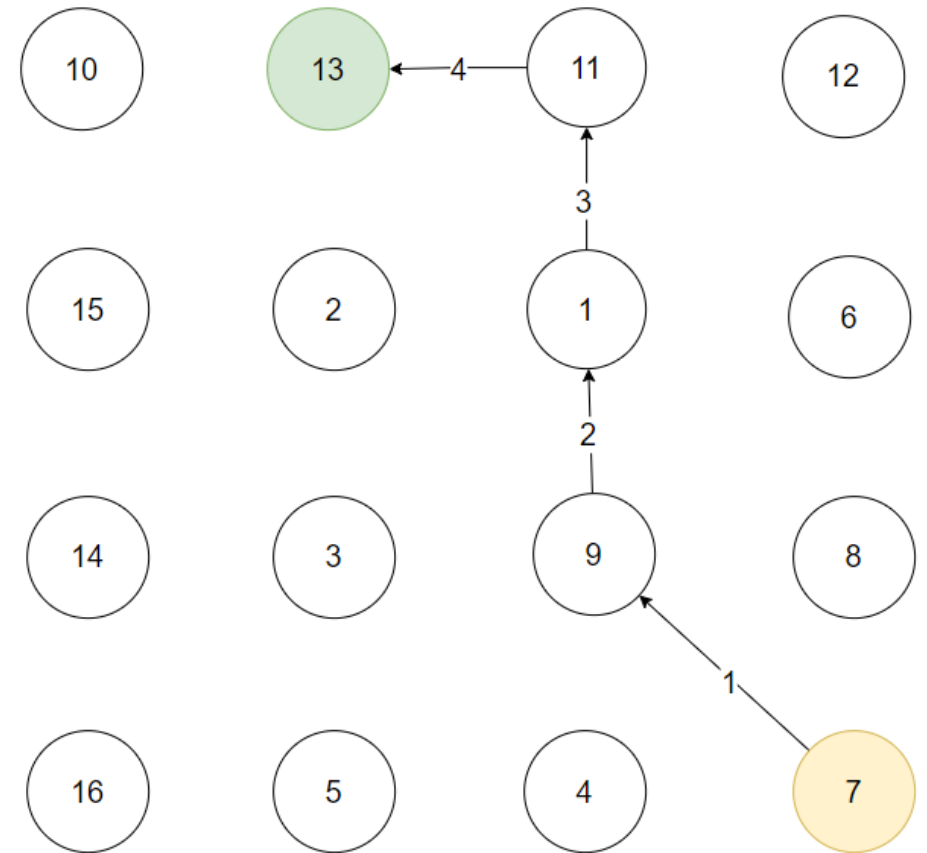
O.S. – One-Time-Ack Application Pt.2

An example of path retrieving using the reversed HPD stack:

When the message start from node 7 to the node 13, the stack read will be:



If 13 has to send the ack to 7, it just sends the data containing the path that other nodes must follow in order to speed-up the «delivery».



O.S. – Clever Transmission Mechanism Pt.1

In the previous slides there was analyzed the main factors which RPL aims to solve, i.e.:

- The power consumption for bad transmission handling (main cause of power drain).
- The congestion due to the many messages sent from/to a single node.

The least common multiple found is the *intelligent and careful sending of data* that doesn't waste useless transmission.

O.S. – Clever Transmission Mechanism Pt.2

The transmission mechanism is theoretically easy to explain and it requires these simple step:

1. Initially, the node scan the area in order to check its neighbors.
2. If one of its neighbors is the destination node, it directly send the message to it.
3. Otherwise:
 1. Send the message to anyone except the nodes who give the message and the ClockSync node.
 2. If we are sending and ACK, pick the next hop in sequence.

O.S. – Buffered Message for loop Avoidance

This method is another mechanism that helps the loop detection with the usage of some buffer (of configurable sizes).

Every node, before processing the message, check the buffer:

1. If the message is read for the first time, it stores the message into the buffer.
2. If the buffer stores a message equals the one received, it can do two things:
 1. Discard the message to avoid the loop.
 2. Resend the message (continue the loop) for a given number of times (configurable value).

O.S. – Time Relaxation on Receivers

Since the RPL protocol uses a time window parameter that identifies the expiration of transmission, in this protocol is also applied **another strategy in order to optimize the overall number of packets successfully delivered.**

The strategy is explained below:

1. Check the data transmitted
2. If the data is expired,
 1. discard the message
3. If the data is expired and the sensor which read the data is the destination sensor, then:
 1. Do not reject the message
 2. Read the message

O.S. – Try and Repeat Method


Since the RPL protocol is based also for lossy networks, when a node wants to transmit the data to another one, the data transmitted can be easily discarded by other nodes for various reason.

In order to cope with this problem the strategy adopted is:

1. Send the message
2. If we do not receive the ACK, retry to send the message.
 1. The number of possible retry can be decided (if the retry number is 0, this strategy is basically avoided).
3. If we receive the ACK the communication is ended successfully .



Presentation Topics

1. Project Goals
 2. Entities at Stake
 3. Messages Protocol
 4. Optimization Strategies
 5. **Pseudo-code Idea**
 6. CupCarbon Simulation
 7. Energy Considerations
 8. Improvements and criticalities
- 

Pseudo-Code Idea

In order to explain the working principle of the RPL advanced protocol, it is required to understand:

- The logic behind the choices made by the **ClockSync Node**.
- The processing mechanism of the **Sensor Nodes**.

To help with this topic it is needed to illustrate both concept with their pseudo-codes.

Pseudo-Code Idea – Sensor Node Pt.1

```
let canSend = false;
let sendingKind = "MSG";
let versionPhased = null;
let directive = null;
let retryCounter = 0;
let versionPhased = 0;

let packToBeAked = null;
let messageSent = null;

let loop = true;

do{

let T = curreTime();
let neighborsNumber = neighboursNumber();
let neighborsSet = neighboursSet();

let pack = receivePack();
let kindPack = getKindPack(pack);
let data = readAllDataFromPack(kindPack);

if (data.kind == "SYN" && data.packVersion > versionPhased) {
    directive = data;
    versionPhased = directive.packVerion;
    let slot = generateSlot(1,directive.numberOfSlots);
    canSend = (slot == directive.chosenSlot) ? true : false;
    let packToTransmit = clonePackWithNewStack(directive,this.sensorID);
    cleverSend(packToTransmit);
}

. . .
```


Pseudo-Code Idea – Sensor Node Pt.2

```
...  
  
if (data.kind == "MSG" || data.kind == "ACK") {  
    let loopPackCount = getBufferedPackCount(data);  
    if (loopPackCount == 0){  
        addPackToBuffer(data);  
    }  
    else if (loopPackCount > this.maxNumberLoop){  
        discardPack(pack);  
    }  
}  
else if (data.kind == "MSG") {  
    if (data.receiverID == this.sensorID){  
        packToBeAked = pack;  
        sendingKind = "ACK";  
    }  
    else if (data.messageTime >= data.limitTime) {  
        discardPack(pack);  
    }  
    else {  
        let packToTransmit = clonePackWithNewStack(pack,sensorID);  
        cleverSend(packToTransmit);  
    }  
}  
...
```

Pseudo-Code Idea – Sensor Node Pt.3

```
...  
  
else if (data.kind == "ACK") {  
    if (data.receiverID == this.sensorID) {  
        log("communication succeeded!");  
        sendingKind = "MSG";  
    }  
    else if (data.messageTime >= data.limitTime) {  
        discardPack(pack);  
    }  
    else{  
        let hopIndex = findHopIndex(data.hops,sensorID);  
        if(hopIndex > 0 && hopIndex+1 < this.hopSize) {  
            send(pack, data.hops[hopIndex+1]);  
        }  
        else {  
            let packToTransmit = clonePackWithNewStack(pack,sensorID);  
            cleverSend(packToTransmit);  
        }  
    }  
}  
  
...
```

Pseudo-Code Idea – Sensor Node Pt.4

```
...  
  
if (canSend == true && sendingKind == "MSG") {  
    if (retryCounter == 0) {  
        let dataMessage = rgauss();  
        let hops = [];  
        let receiverID = specularSensor(  
            this.sensorID,  
            directive.sensorsNumber,  
            directive.loswestID  
        );  
        let message = generateNewMessage(  
            T,  
            directive.limitTime,  
            this.sensorID,  
            receiverID,  
            dataMessage,  
            hops  
        );  
        cleverSend(message);  
        messageSent = message;  
        retryCounter ++;  
    }  
    else if (retryCounter < this.maxRetry) {  
        messageSent.timeToLive = directive.limitTime+(10*directive.limitTime)/100;  
        cleverSending(messageSent);  
        retryCounter ++;  
    }  
}  
...
```

Pseudo-Code Idea – Sensor Node Pt.5

```
...  
  
else if (canSend == true && sendingKind == "ACK") {  
    let dataToBeAked = readAllDataFromPack(packToBeAked);  
    let ack = generateNewAck(  
        T,  
        directive.limitTime,  
        this.sensorID,  
        dataToBeAked.senderID,  
        dataToBeAked.data,  
        reverseStack(dataToBeAked.hops)  
    );  
    sendingKind = "NIL";  
    packToBeAked = null;  
}  
  
loop = (this.sensorBattery>0) true : false;  
} while(loop);|
```

Pseudo-Code Idea – ClockSync Node Pt.1

```
let loop = true;
let simulationVariables = {this.sensorsNumber, this.lowestID}


let syncVersion = 0;

do(){

  let T = curreTime();
  let limitTime = T+simulationVariables*10;
  let chosenSlot = 1+(syncVersion%this.numberOfSlots);
  let hops = [];
  let newSyncMessage = generateNewSyncMessage(
    syncVersion,
    chosenSlot,
    this.numberOfSlots,
    limitTime,
    this.id,
    simulationVariables,
    hops
  );
  send(newSyncMessage,*);
  syncVersion ++;
  delay this.sensorsNumber*100
} while(loop);
```



Presentation Topics

1. Project Goals
 2. Entities at Stake
 3. Messages Protocol
 4. Optimization Strategies
 5. Pseudo-code Idea
 6. **CupCarbon Simulation**
 7. Energy Considerations
 8. Improvements and criticalities
- 

CupCarbon Simulation – Test 1 – RPL optimized

A more accurate analysis is presented in the following scenarios both for RPL simple and for the RPL optimized:

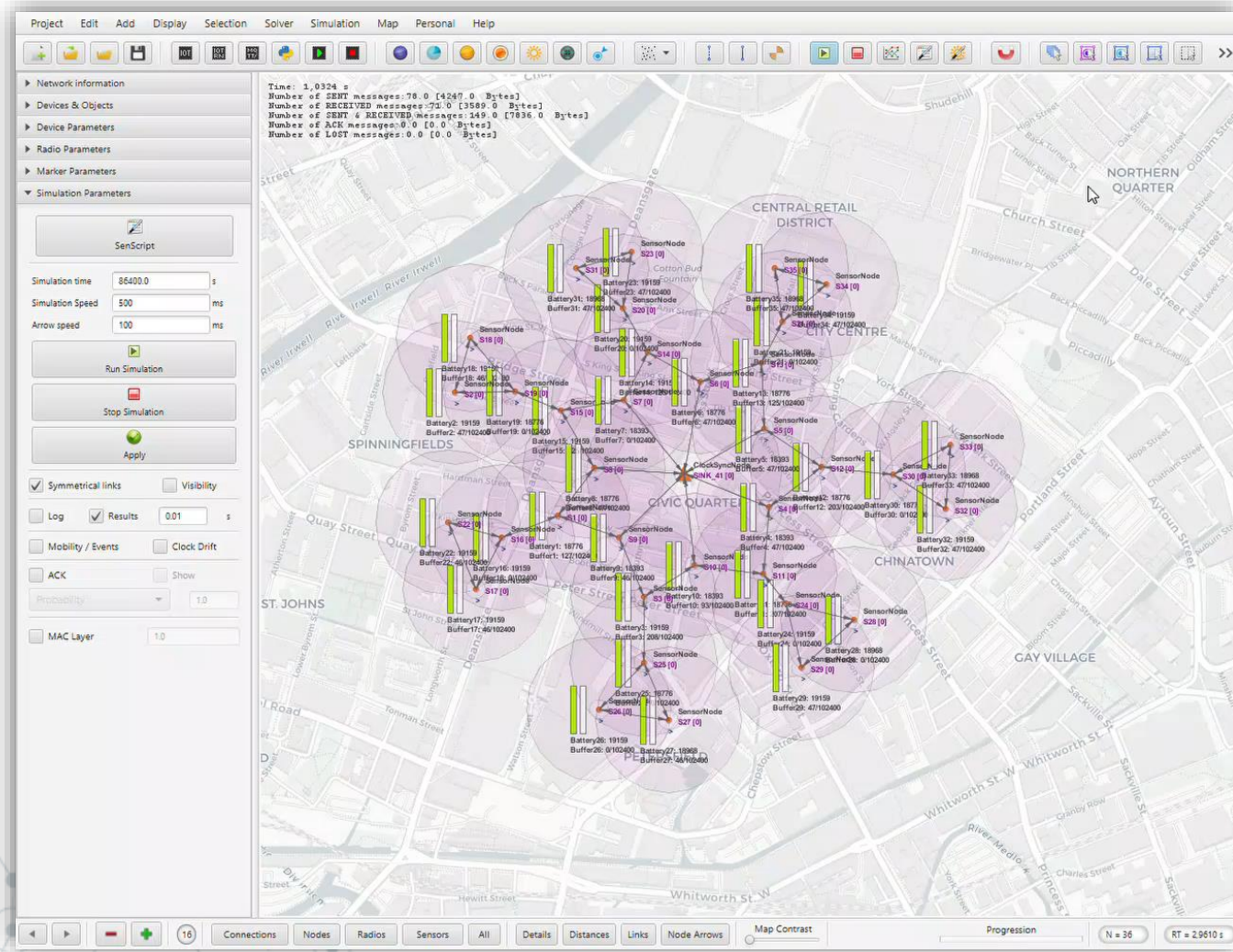
1. Test 1 – Snowflake topology
2. Test 2 – Square topology
3. Test 3 – Fisheye topology

Each one of these test are made with this battery drain criteria:

$$battery = battery - \left(battery * \frac{\#MessageSentPerLoop}{100} \right)$$

So, it is not based on a fixed number, but it depends on how many data a sensor has transmitted and the actual battery status.

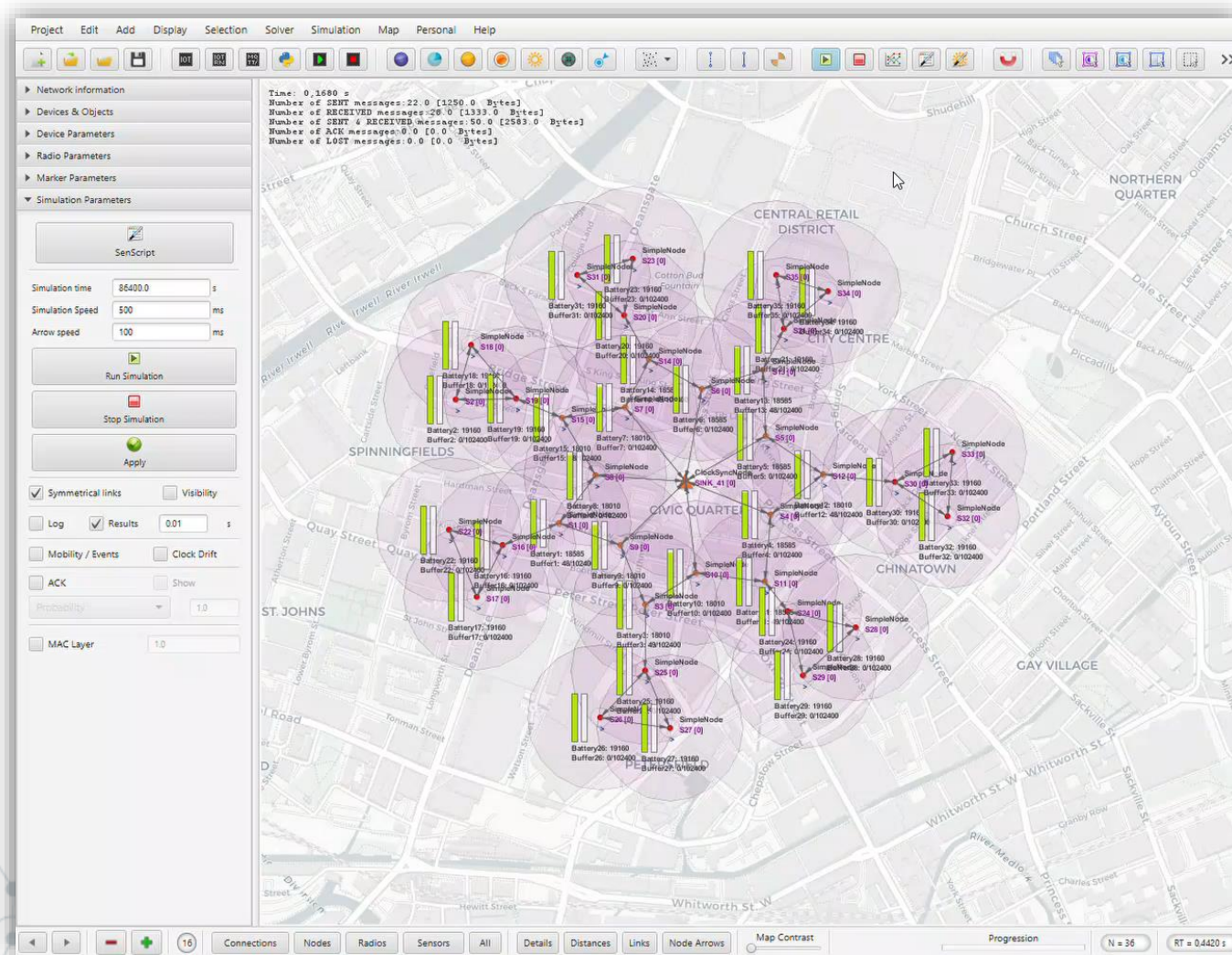
CupCarbon Simulation – Test 1 – RPL optimized



Network parameters:

- Number Nodes: 35
- Number Slots: 2
- Buffer size: 10
- Max loop: 2
- Max retry: 3
- Sync Refresh: $35 \times 100\text{ms}$
- Time: 25min

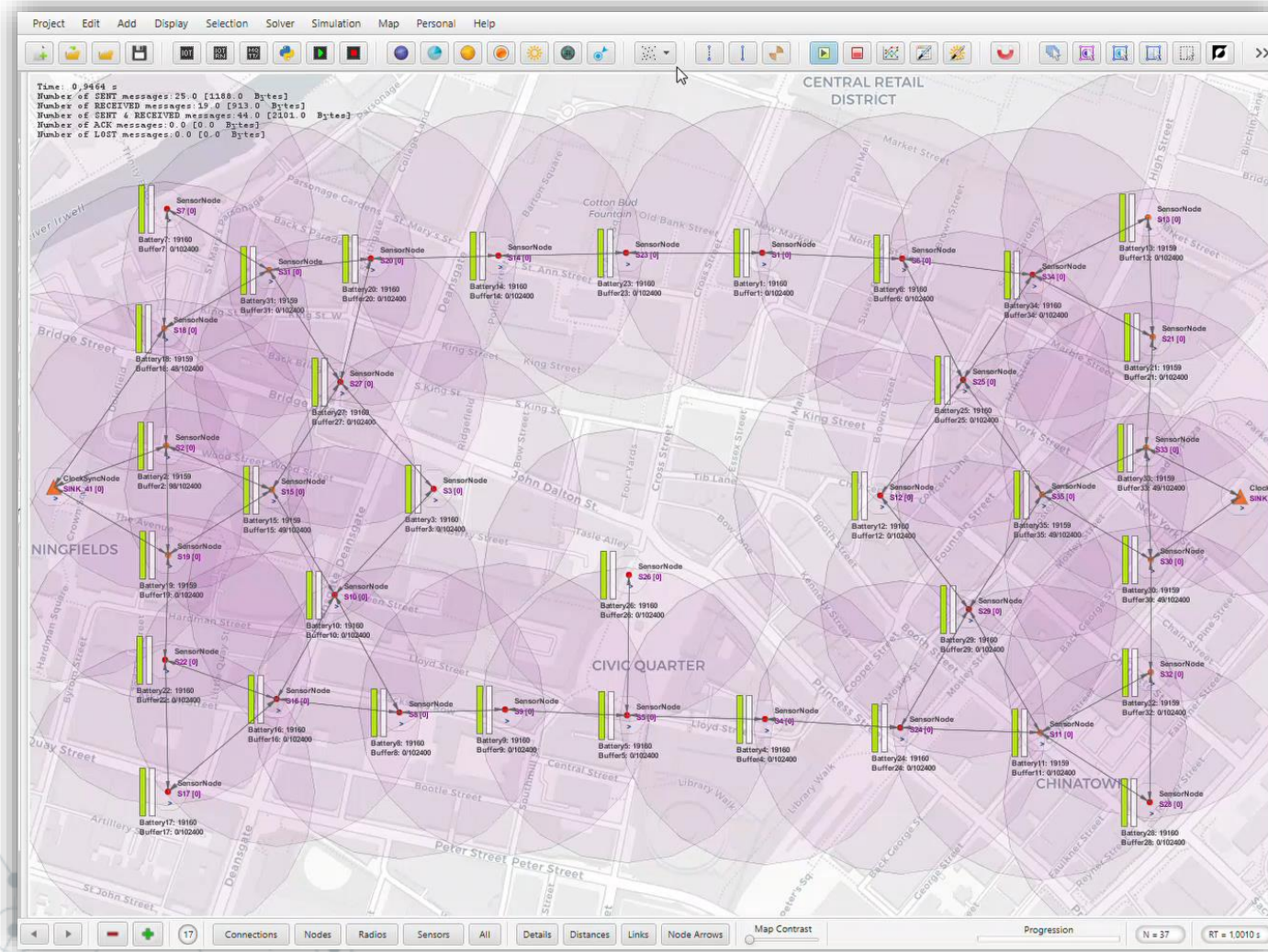
CupCarbon Simulation – Test 1 – RPL optimized



Network parameters:

- Number Nodes: 35
- Number Slots: 2
- Sync Refresh: $35 \times 100\text{ms}$
- Time: 8min

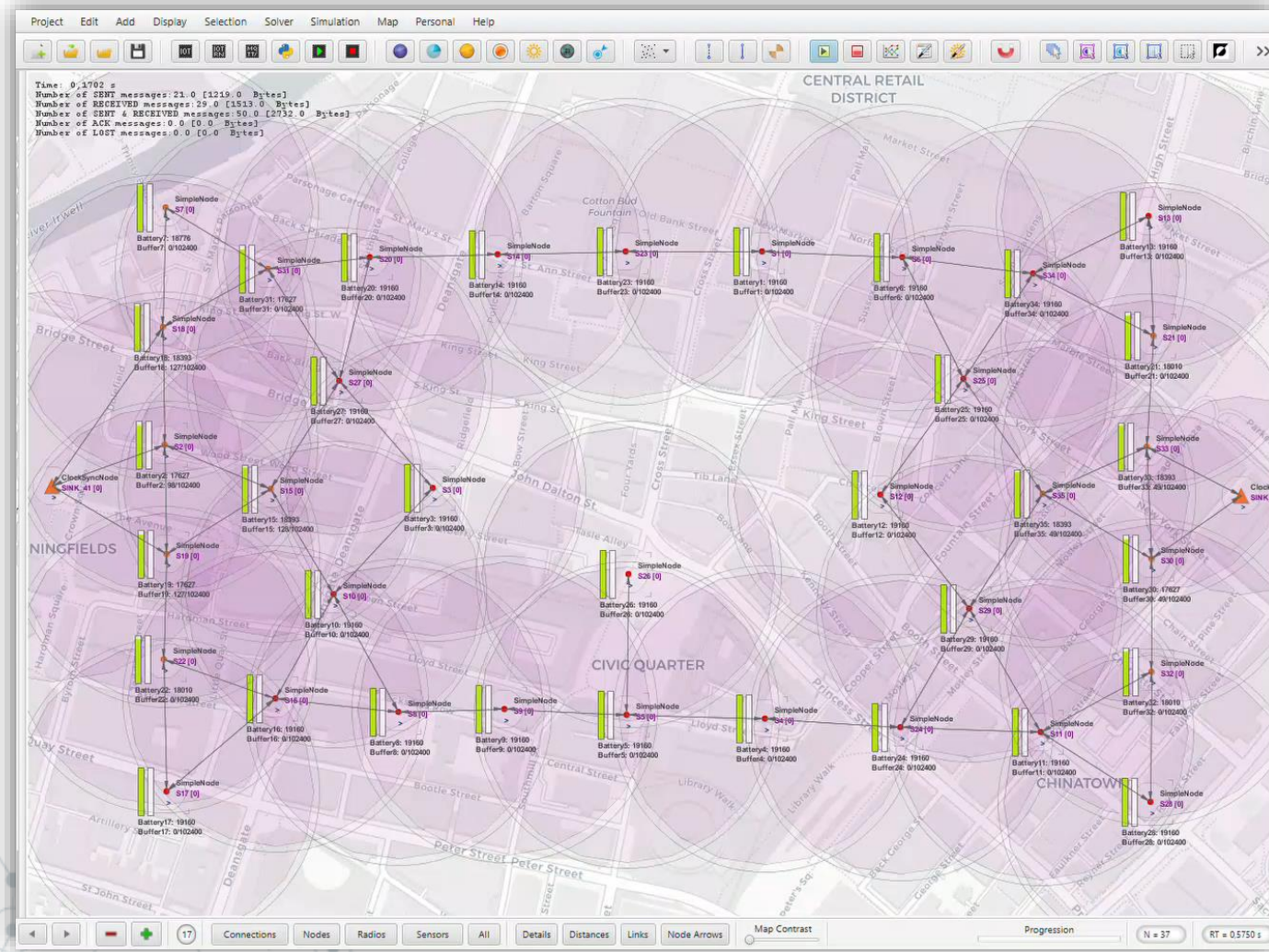
CupCarbon Simulation – Test 2 – RPL optimized



Network parameters:

- Number Nodes: 35
- Number Slots: 2
- Buffer size: 15
- Max loop: 3
- Max retry: 3
- Sync Refresh: $35 \times 100\text{ms}$
- Time: 27min

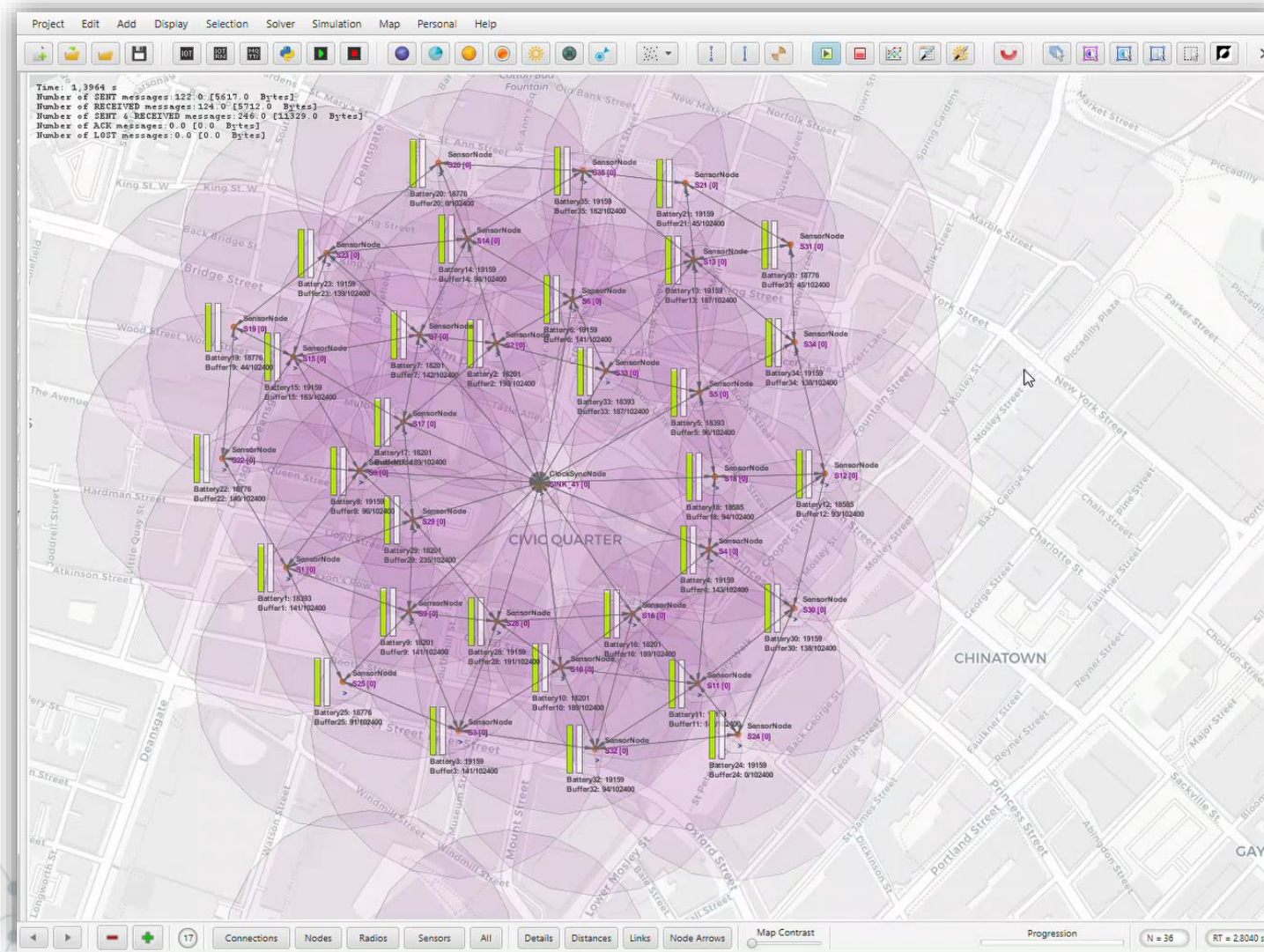
CupCarbon Simulation – Test 2 – RPL flooding



Network parameters:

- Number Nodes: 35
- Number Slots: 2
- Sync Refresh: $35 \times 100\text{ms}$
- Time: 7min

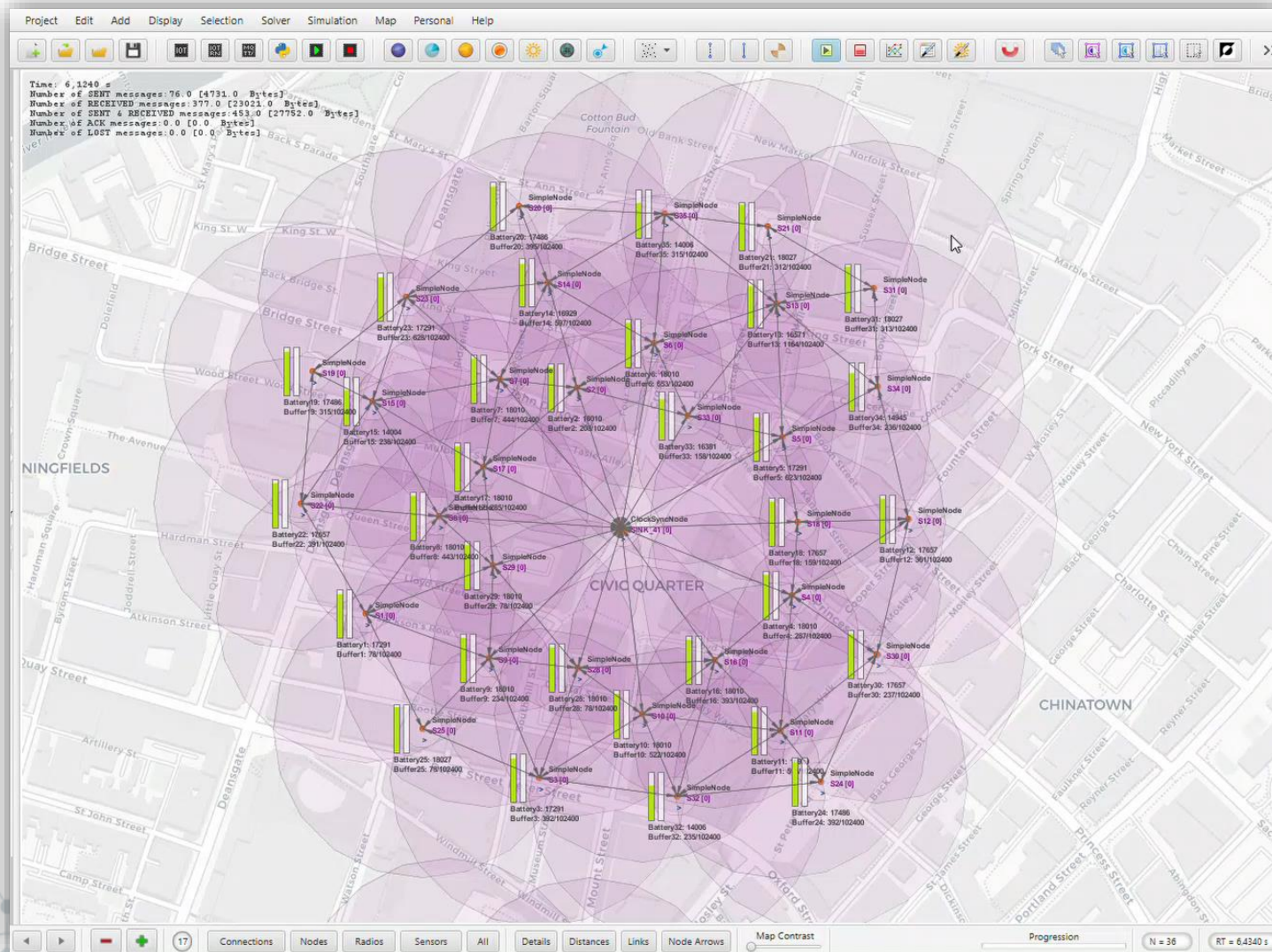
CupCarbon Simulation – Test 3 – RPL optimized



Network parameters:

- Number Nodes: 35
- Number Slots: 5
- Buffer size: 20
- Max loop: 3
- Max retry: 3
- Sync Refresh: $35 \times 150\text{ms}$
- Time: 25min

CupCarbon Simulation – Test 3 – RPL flooding




Network parameters:

- Number Nodes: 35
- Number Slots: 5
- Sync Refresh: $35 \times 150\text{ms}$
- Time: 5min



Presentation Topics

1. **Project Goals**
 2. **Entities at Stake**
 3. **Messages Protocol**
 4. **Optimization Strategies**
 5. **Pseudo-code Idea**
 6. **CupCarbon Simulation**
 7. **Energy Considerations**
 8. Improvements and criticalities
- 

Energy considerations Pt.1

For the energy analysis the test are essentially made with the support of two main things:

- Some use-case scenarios (e.g. the topologies shown before).
- A way to consume the energy.

For the experiment's sake, each one of the following simulations are done with the same battery drain method which is:

- 0.6 for the transmission
- 0.1 for the reception

This is an amount of energy draining way bigger than the usual, which is:

- $6 * 10^{-5}$ for transmission
- $1 * 10^{-5}$ for reception

Energy considerations Pt.2

This wants to say that each simulation time must be increased 4 order of magnitude w.r.t. the actual result.

An example, if a sensor has 30000J and after 70s the sensor dies, the «standard» measurement will be 1050000s i.e. less then two weeks.

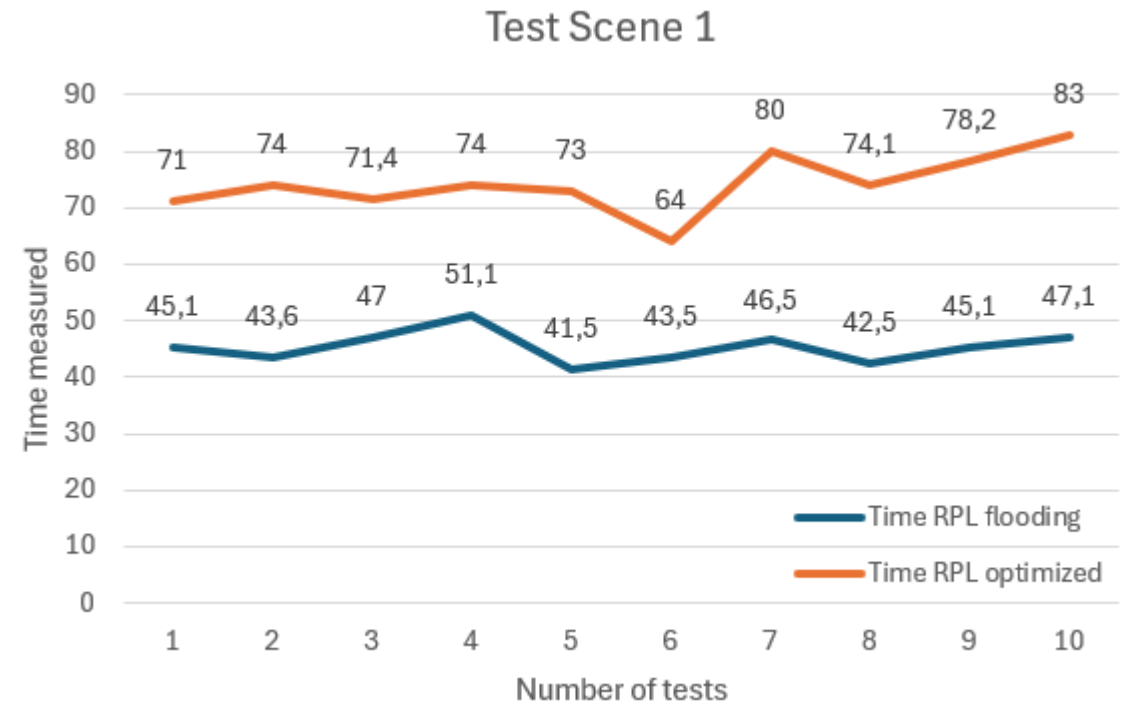
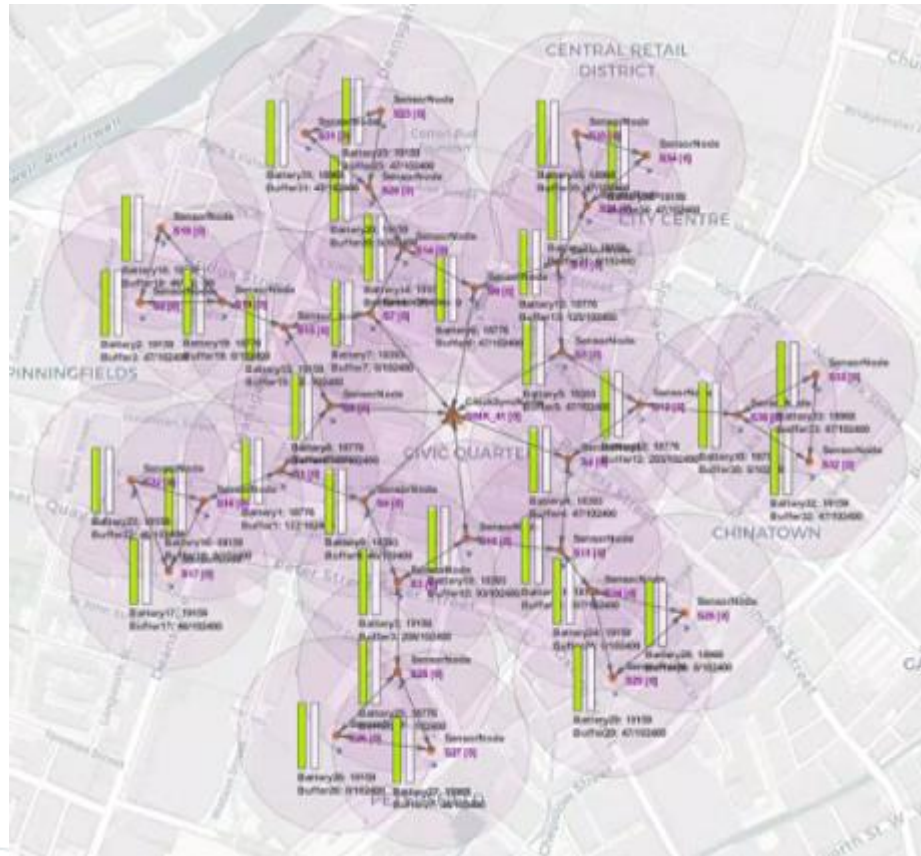
A more accurate analysis is presented for each one of the scenarios previously presented:

1. Test 1 – Snowflake topology
2. Test 2 – Square topology
3. Test 3 – Fisheye topology

We must also say that the battery capacity settled for each test is: 20000J

Energy considerations – Test 1 – Pt.1

Analysis of snowflake scenario:



Energy considerations – Test 1 – Pt.2

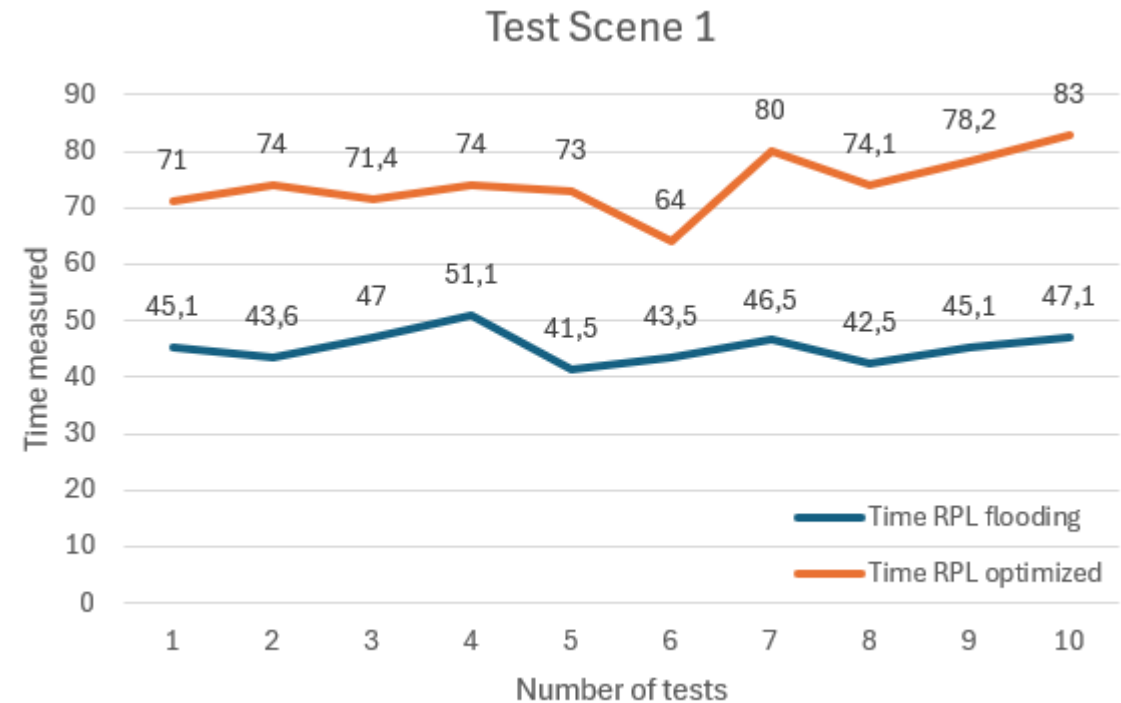
Analysis of snowflake scenario (20000J) for a full-transmission network:

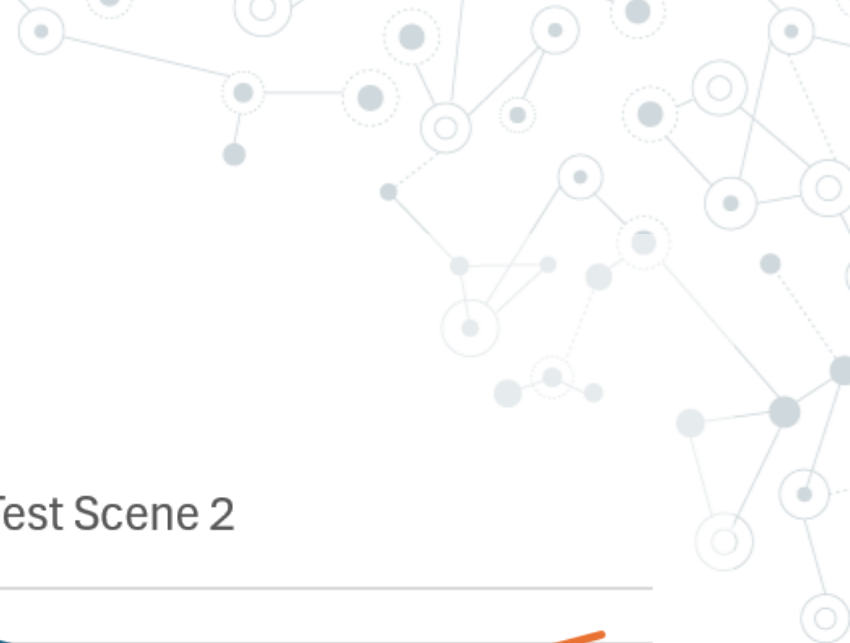
On the average:

- For RPL optimized we have
 - 75.1s duration.
- For RPL flooding we have
 - 50s duration.

The «standard» time will say that

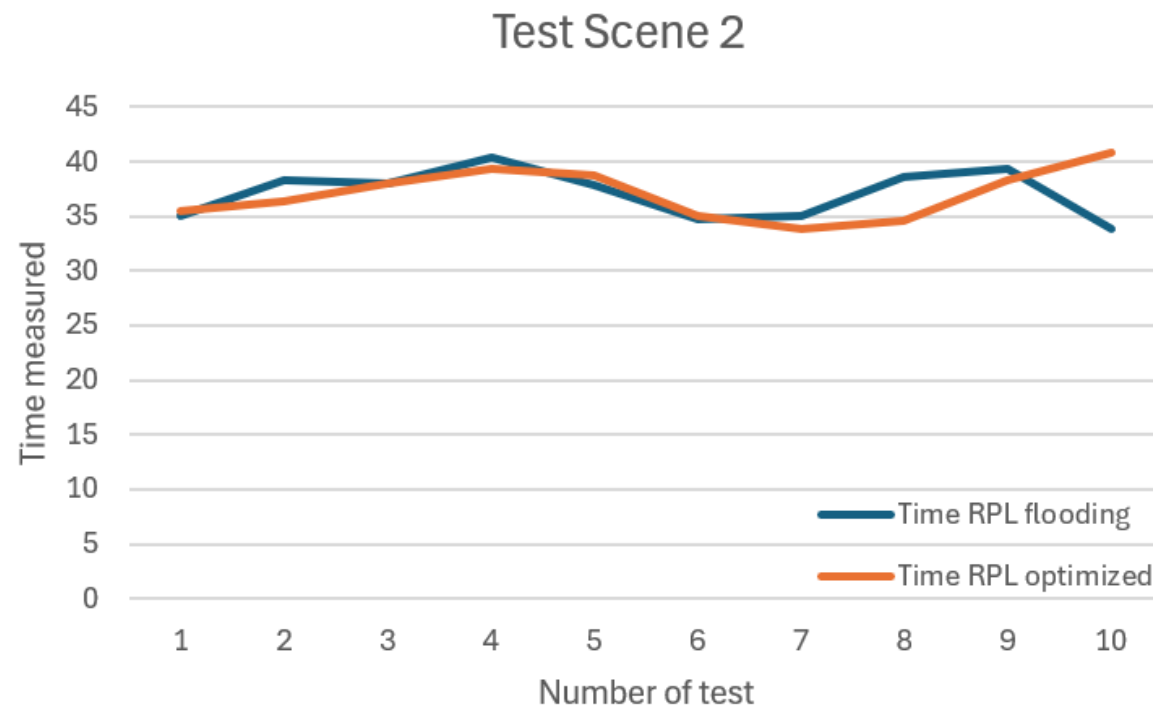
- For RPL optimized we have
 - < 1 week per sensor
- For RPL flooding we have
 - > 5 days per sensor





Test Scene 2

Analysis of square scenario:



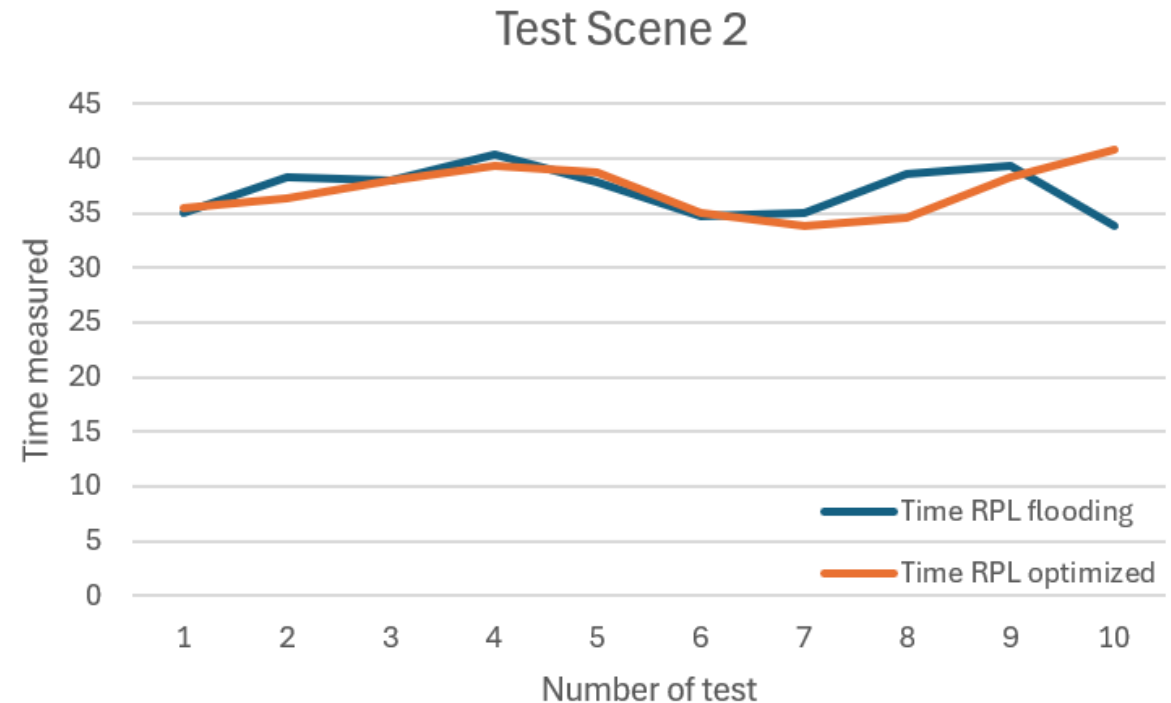
Energy considerations – Test 2 – Pt.2

Analysis of square scenario (20000J) for a full-transmission network :

On the average:

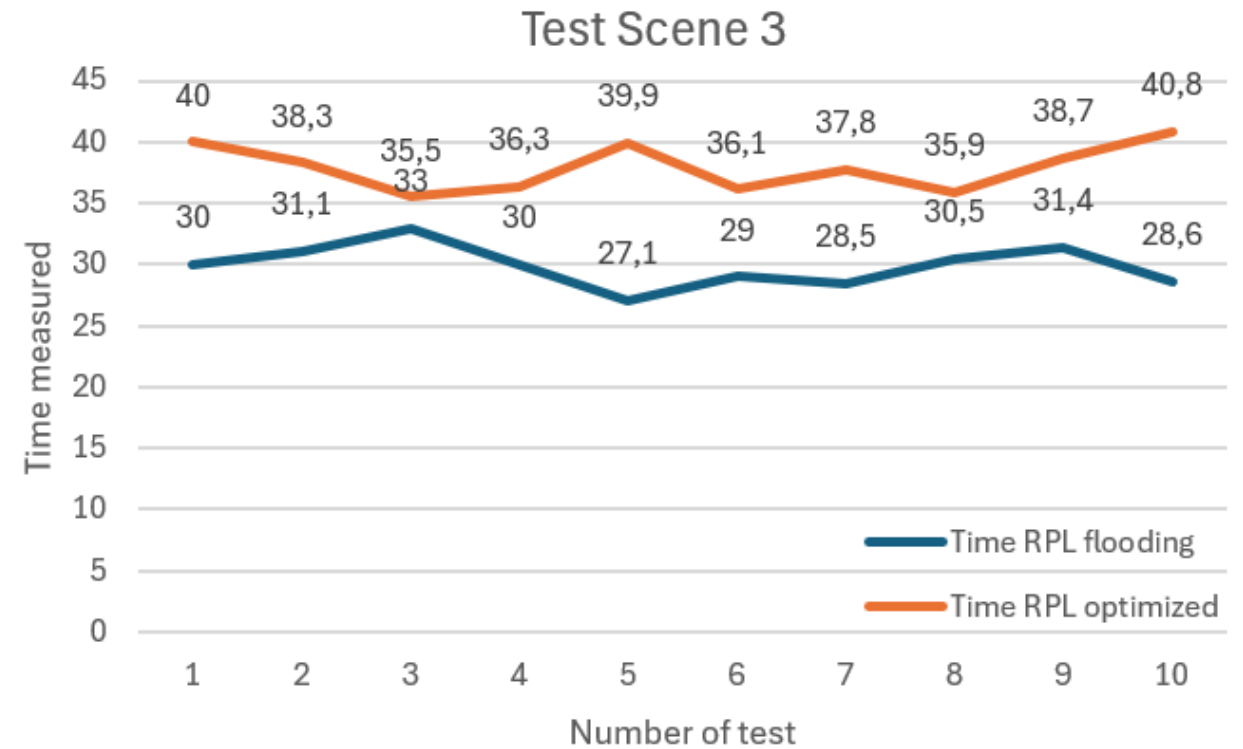
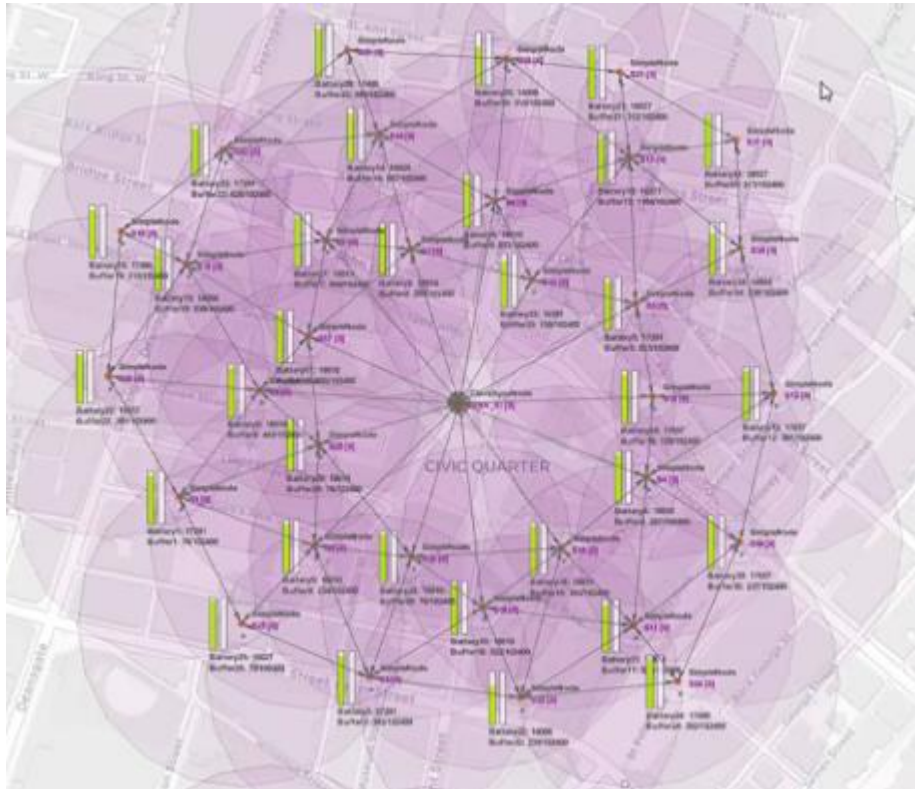
- For RPL optimized we have
 - 37.04s duration.
- For RPL flooding we have
 - 37.11s duration.

The «standard» time will say that
both for RPL optimized and flooding
< 4 days per sensor.



Energy considerations – Test 2 – Pt.1

Analysis of fisheye scenario:



Energy considerations – Test 2 – Pt.2

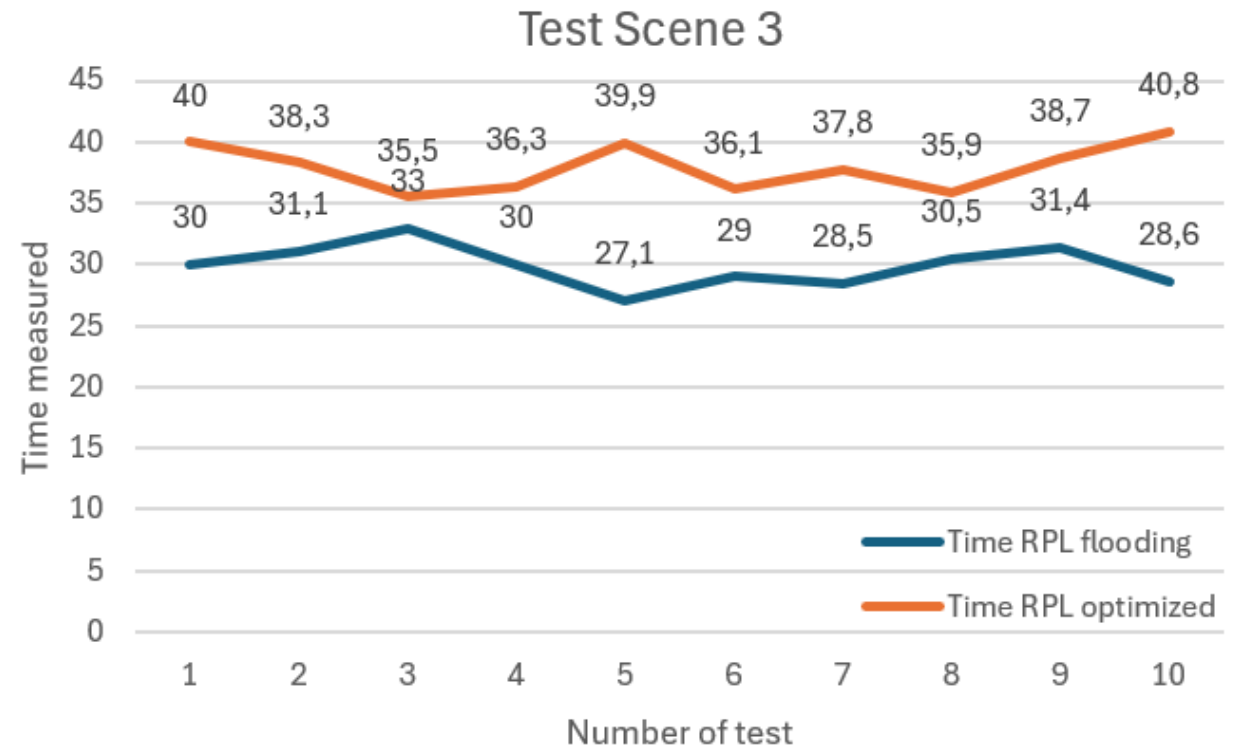
Analysis of fisheye scenario (20000J) for a full-transmission network :

On the average:

- For RPL optimized we have
 - 37.53s duration.
- For RPL flooding we have
 - 29.92s duration.

The «standard» time will say that

- For RPL optimized we have
 - > 1 week per sensor
- For RPL flooding we have
 - > 4 days per sensor





Presentation Topics

1. Project Goals
 2. Entities at Stake
 3. Messages Protocol
 4. Optimization Strategies
 5. Pseudo-code Idea
 6. CupCarbon Simulation
 7. Energy Considerations
 8. **Improvements and criticalities**
- 

Improvement and Criticalities

Further improvement will be:

- ☐ The nodes can automatically decide if transmits some messages or not based on:
 - ☐ Remaining Energy
 - ☐ Buffer Saturation
- ☐ Enable the dying sensor to send a message to the ClockSync Node in order to influence:
 - ☐ The slot decision mechanism.
 - ☐ The time-window mechanism
- ☐ Using a HPD mechanism also for populate internal “routing table” in order to address the right message to the right sensor. Doing so, it can be enhanced the power of the clever sending mechanism

Improvement and Criticalities

The major criticalities are:

- ❖ The RPL protocol built in this project better perform in topologies that do not have to lose connections. For example, the square topology presents more or less then 2 connection for nodes. In this case, using a protocol that works with filtering and resending is pretty much useless, it is may also degrade the overall performance, preferring an approach orient to the naked forward mechanism like the RLP with flooding.
- ❖ When the ClockSync nodes into some areas die, the nodes will be in a state for which only the sensor that has the last chosen slot can send their messages.



Thank you.