Implementation Details of FL-RPL

Summary of ContikiRPL Operation [1]

ContikiRPL implements the RPL protocol, as specified RFC 6552 RPL specification, and two objective functions— OF0 and the Minimum Rank Objective Function with Hysteresis (MRHOF). ContikiRPL has been successfully tested for interoperability through the IPSO Alliance interop program, where it was used on three different platforms and ran over two different link layers, IEEE 802.15.4 and the Watteco low-power power-line communication module.
The ContikiRPL implementation separates protocol logic, message construction and parsing, and objective functions into different modules.

- The **protocol logic module** manages DODAG information, maintains a set of candidate parents and their associated information, communicates with objective function modules, and validates RPL messages at a logical level according to the RPL specification.
- The **message construction and parsing module** translates between RPL's ICMPv6 message format and ContikiRPL's own abstract data structures.
- Finally, **the objective function modules** implement an objective function API. To facilitate simple replacement of and experimentation with objective functions, their internal operation is opaque to ContikiRPL.

ContikiRPL does not make forwarding decisions per packet, but sets up forwarding tables for uIPv6 and leaves the actual packet forwarding to uIPv6. Outgoing IPv6 packets flow from the uIPv6 layer to the 6LoWPAN layer for header compression and fragmentation. The 6LoWPAN layer sends outgoing packets to the MAC layer. The default Contiki MAC layer is a CSMA/CA mechanism that places outgoing packets on a queue. Packets from the queue are transmitted in order through the **radio duty cycling (**RDC) layer.
The RDC layer in turn transmits the packets through the radio link layer. The MAC layer will retransmit packets until it sees a link layer acknowledgment from the receiver. If a collision occurs, the MAC layer does a linear back-off and retransmits the packet. Outgoing packets have a configurable threshold for the maximum number of transmissions.
An **external neighbor information module** provides **link cost estimation** updates through a callback function. Within one sec of such an update, ContikiRPL recomputes the path cost to the sink via the updated link, and checks with the selected objective function whether to switch the preferred parent. The link cost reflects the per-neighbor ETX metric, which is calculated using an exponentially-weighted moving average function over the number of link-layer transmissions with $\alpha = 0.2$. The ETX is used to compute the rank for the MRHOF objective function and in parent selection for the OF0 objective function. New neighbor table entries have an initial ETX estimate of 5. The ContikiRPL neighbor eviction policy is to keep neighbors that have good ETX estimates and low ranks.

1) What are the existing files that implement basic RPL protocol

The implementation of RPL in contiki is located under the contiki folder : core/net/rpl
This folder contains the following files:

- **Rpl-dag.c**: identifies the logic of Directed Acyclic graphs in RPL
- **Rpl-ext-header.c**: management of extension headers for ContikiRPL
- **Rpl-icmp6.c**: identifies the ICMP6 I/O for RPL control messages.
- **Rpl-timers.c**: RPL timer management.
- **Rpl.c**: ContikiRPL: an implementation of RPL as it is described by the IETF RFC 6550.
- **Rpl.h**: a configuration file of RPL: It contains the definition of all the variables and constants that RPL uses.
- **rpl-conf.h**: it is a configuration file.it contains the specification of the routing metrics to be used, the selection of the objective function, select the DAG instance, the mode of the node (router or leaf node), the maximum number of DAGs per instance and the DIO parameters (Imin, Idoublings, DIO redundancy).
- **rpl-private.h**: this file contains the definition of the functions and constants that the RPL implementation requires, such as the definition of the trickle timer parameters, the RPL message types, the RPL control messages options, etc.
- **Makefile.rpl**: - specifies the contiki source files that we want to call.

ContikiRPL contains the implementation of the two objective functions OF0 and MRHOF:

- **Rpl-of-etx.c**: The minrank-hysteresis objective function (OCP 1). This implementation uses the estimated number of transmissions (ETX) as the additive routing metric.
- **Rpl-of0.c**: contains an implementation of RPL's objective function 0.

In addition to the RPL implementation, ContikiRPL contains applications to test the behavior of RPL in simulation and in real experimentation. The examples are located under examples/ipv6/. the examples are:

- **rpl-border-router:** contains an implementation of an rpl border router which builds a border router that interfaces through a **tunslip6** connection to Linux. Apparently only one border router is allowed in the specification.

[Anis: Olfa, please clarify what are the RPL libraries described above that are used in RPL Border. The same for the other types of nodes]

As a response to your question: Any application uses all the RPL libraries that are listed above. In any application, the line #include "net/rpl/rpl.h" is present in the top of the .c file, and this means that all the mechanisms and operations of RPL that are implemented are called when

constructing the DAG when installing the applications on the nodes. In addition, any modification in the libraries (change of the objective function, change of the DIO parameters, etc.) will affect automatically the behavior of the nodes in the DAG. So if we want to change the specification of RPL, we don't change the applications located in examples/ipv6, we change just the RPL implementation and this will automatically be called in the applications. The applications contain simple treatments such as the sending of data messages, the creation of a UDP connection with the sink, etc.

- **rpl-collect:** contains two types of nodes: one is the sink and the other is the sender. the sender sends a data packet to the sink every 60-62 seconds.
- **rpl-udp:** contains two types of nodes: udp-client and udp-server. the client creates a UDP connection with the server and begins sending hello messages, and the server sends an acquittement.
- simple-udp-rpl: contains two types of UDP connections : Unicast and broadcast.

2) MAC and Network Layer Protocols for RPL

Anis: Olfa please add one section that briefly the main files used for MAC and Network Layers. This is also important

In the contiki protocol stack, ContikiRPL is on top of uIP protocol stack which is composed by the MAC layer, the RDC layer, the 6LoWPAN layer and the uIPv6 layer:

1) The contiki MAC Layer:

Contiki has several MAC layers such as NullMAC, CSMA, ContkiMAC, XMAC, etc. The MAC layer sits on top of the RDC layer. The MAC layer is responsible for avoiding collisions at the radio medium and retransmitting packets if there were a collision.

1) General MAC files:

- **mac.c:** mac driver
- **mac.h:** mac driver header file
- **framer.h**: A MAC framer is responsible for constructing and parsing the header in MAC frames

2) ContikiMAC:

- **contikimac.c**: Implementation of the ContikiMAC power-saving radio duty cycling protocol.
- **contikimac.h**: Header file for the ContikiMAC radio duty cycling protocol

3) CSMA

- **csma.c**: A Carrier Sense Multiple Access (CSMA) MAC layer
- **csma.h**: A MAC stack protocol that performs retransmissions when the underlying MAC layer has problems with collisions.

4) CXMAC

- **xmac.c**
- **xmac.h**
- **cxmac.c:** A simple power saving MAC protocol based on X-MAC [SenSys 2006].
- **4) IEEE 802.15.4**
- **frame802154.c:** 802.15.4 frame creation and parsing functions
- **frame802154.h**
- **framer-802154.c:** MAC framer for IEEE 802.15.4.
- **framer-802154.h:** A MAC framer for IEEE 802.15.4

5) NullMAC

- **framer-nullmac.c:** MAC framer for nullmac
- **framer-nullmac.h:** MAC framer for nullmac
- **nullmac.c: :** A MAC protocol that does not do anything
- **nullmac.h:** A MAC protocol implementation that does not do anything

6) Low Power Probing

- **lpp.c**
- **lpp.h:**

7) SICSLOMAC

- **sicslowmac.c:** MAC interface for packaging radio packets into 802.15.4 frames
- **sicslowmac.h:** header file

8) TDMA MAC Protocol

- **tdma_mac.c**
- **tdma_mac.h**
- **ctdma_mac.c**
- **ctdma_mac.h**

2) The Radio Duty Cycling Layer:

- **nullrdc-noframer.c:** A MAC protocol that does not do anything
- **nullrdc-noframer.h:** A null RDC implementation that uses framer for headers
- **nullrdc.c:** A null RDC implementation that uses framer for headers
- **nullrdc.h:** A null RDC implementation that uses framer for headers
- **phase.c:** Common functionality for phase optimization in duty cycling radio protocols
- **phase.h:** Common functionality for phase optimization in duty cycling radio protocols

- **rdc.h**: RDC driver header file

3) uIP layer

Contki has two stacks which are uIP TCP/IP stack and Rime Stack. RPL is part of the uIP stack.

1) The uIP TCP/IP stack

- **uip.c** : The uIP TCP/IP stack code
- **uip.h** : Header file for the uIP TCP/IP stack
- **uip6.c** : The uIP TCP/IPv6 stack code

2) Protosockets library

The protosocket structrure is an opaque structure with no user-visible elements.

- **psock.c** : The representation of a protosocket.
- **psock.h** : Protosocket library header file

3) uIP hostname resolver functions

The uIP DNS resolver functions are used to lookup a hostname and map it to a numerical IP address

- **resolv.c** : DNS host name to IP address resolver
- **resolv.h** : uIP DNS resolver code header file

4) Simple UDP module

The default Contiki UDP API is difficult to use. The simple-udp module provides a significantly simpler API.

- **simple-udp.c**
- **simple-udp.h**

5) Serial Line IP (SLIP) protocol

The SLIP protocol is a very simple way to transmit IP packets over a serial line.

- **slipdev.c** : SLIP protocol implementation
- **slipdev.h** : SLIP header file

6) The Contiki/uIP interface

- **tcpip.c** : Code for tunnelling uIP packets over the Rime mesh routing module
- **tcpip.h** : Header for the Contiki/uIP interface

7) uIP packet forwarding

- **uip-fw.c** : uIP packet forwarding.

- **uip-fw.h** : uIP packet forwarding header file.

8) uIP TCP throughput booster hack

- **uip-split.c** :
- **uip-split.h** : Module for splitting outbound TCP segments in two to avoid the delayed ACK throughput degradation

9) uIP Address Resolution Protocol

- **uip_arp.h**: Macros and definitions for the ARP module.
- **uip_arch.h**: Implementation of the ARP Address Resolution Protocol.

10) Configuration options for uIP

- uipopt.h

11) uIP IPv6 specific features:

- uip-ds6.c: IPv6 data structures handling functions.
- uip-ds6.h: Network interface and stateless autoconfiguration (RFC 4862)
- uip-icmp6.c: ICMPv6 echo request and error messages (RFC 4443)
- uip-icmp6.h: ICMPv6 echo request and error messages (RFC 4443)
- uip-nd6.c: Neighbor discovery (RFC 4861)
- uip-nd6.h: Neighbor discovery (RFC 4861)
- uip6.c: The uIP TCP/IPv6 stack code.

In addition to the RPL functions.

12) Netstack

- Netstack.c: Initialiation file for the Contiki low-layer network stack (NETSTACK)
- Netstack.h: Include file for the Contiki low-layer network stack (NETSTACK)
- rawpacket-udp.c : Implementation of the raw packet API with broadcast UDP packets
- rawpacket-udp.h : A brief description of what this file is
- rawpacket.h : Header file for the Contiki raw packet API interface
- TCP dump
- tcpdump.c
- tcpdump.h
- Neighbor Discovery:
- uip-nd6.c : Neighbor discovery (RFC 4861)
- uip-nd6.h : Neighbor discovery (RFC 4861)
- Database of neighbors:
- uip-neighbor.c : Database of link-local neighbors, used by IPv6 code and to be used by a future ARP code rewrite
- uip-neighbor.h : Header file for database of link-local neighbors, used by IPv6 code and

to be used by future ARP code

- UIP packets over Rime Mesh:
- uip-over-mesh.c : Code for tunnelling uIP packets over the Rime mesh routing module
- uip-over-mesh.h : Header file for tunnelling uIP over Rime mesh
- UIP packet queue:
- uip-packetqueue.c
- uip-packetqueue.h
- Sending UDP packets :
- uip-udp-packet.c : Module for sending UDP packets through uIP
- uip-udp-packet.h : Header file for module for sending UDP packets through uIP
- Libraray and configuration Functions:
- uIP IPv6 specific features
- The uIP IPv6 stack provides new Internet communication abilities to Contiki
- uiplib.h : Library and configuration macros
- uiplib.c :

4) 6LoWPAN Layer

- sicslowpan.c : 6lowpan implementation (RFC4944 and draft-ietf-6lowpan-hc-06
- sicslowpan.h : Header file for the 6lowpan implementation, (RFC4944 and draft-hui-6lowpan-hc-01)

3) How objective functions are implemented.

As mentioned before, To facilitate simple replacement of and experimentation with objective functions, their internal operation is opaque **[Anis: what do you mean by opaque?]** to ContikiRPL. That means that the objective function files are independent from the other contikiRPL files and they are simply called by the other files.
Thus, to add a new objective function named for example rpl_of_new.c, we need just to do the following steps:
1) create a new implementation of the objective function in the **/core/net/rpl** file
2) in the **makefile.rpl**, add the name of the file:
the makefile looks:

```
CONTIKI_SOURCEFILES += rpl.c rpl-dag.c rpl-icmp6.c rpl-timers.c \
      rpl-of-etx.c rpl-ext-header.c
```

and becomes:

```
CONTIKI_SOURCEFILES += rpl.c rpl-dag.c rpl-icmp6.c rpl-timers.c \
      rpl_of_new.c rpl-ext-header.c
```

3) In the rpl-conf.h (v2.6) (earlier in rpl.h)
Replace the line:

```
/* ETX is the default objective function. */
```

```
#define RPL_OF rpl_of_etx
```
with `#define RPL_OF rpl_of_new`
and this will allow RPL to work with the new objective function.

3) Describe the implementation of basic OF0 and OF1.

The implementations follow the specifications of the RFCs 6552 and 6719.

Both the two objective functions are implemented by using the same functions which are:
- reset
- best_parent
- best_dag
- calculate_rank
- update_metric_container

ContikiRPL follows the specifications of the two RFCs 6552 and 6719.

- Implementation of OF0:

As mentioned in RFC 6552, OF0 does not work with metric containers. The file that contains the implementation of OF0 is named `rpl-of0.c`.
It contains the following functions:

1- **best_parent:** Compare two parents by looking both at their rank and at the ETX for that parent, and choose the parent that has the most favorable combination. Thus, the ETX is used in the process of parent selection.

```
static rpl_parent_t * best_parent(rpl_parent_t *p1, rpl_parent_t *p2)
{
```

r1 nd r2 are the ranks of the two candidate neighbors p1 and p2 respectively. p1-> link_metric=etx (configured in rpl.c)

```
 r1 = DAG_RANK(p1->rank, p1->dag->instance) * NEIGHBOR_INFO_ETX_DIVISOR
  +  p1->link_metric;
        r2     =     DAG_RANK(p2->rank,     p1->dag->instance)     *
  NEIGHBOR_INFO_ETX_DIVISOR +  p2->link_metric;
```

2- **calculate_rank**: calculates the node's rank according to the base rank which is the rank of the parent node.

```
Rank     =     base_rank     +     increment     (min_hoprankinc     :
DEFAULT_RANK_INCREMENT)
```

[Anis: can you explain what is the relation of that equation with ETX] I cannot see ETX in the equation for calculating the rank
The ETX is concluded from the rank of the parent, where base_rank=parent->rank., the

parent is concluded from the best_parent function which considers the ETX in selecting the best parent function.

3- **update_metric_container**

`dag->mc.type = RPL_DAG_MC_NONE;`

(this is because OF0 does not work with any routing metric).

- Implementation of MRHOF with ETX

This implementation uses the estimated number of transmissions (ETX) as the additive routing metric. the file is named `rpl-of-etx.c`.

Like OF0, rpl-of-etx contains 4 principle functions which are:

1- **calculate_path_metric**: this function calculates the path metric by adding the ETX contained in the DIO sent by the neighbor with the ETX of the link of that neighbor.
it returns the following value:

`p->mc.obj.etx +p-> link_metric;`

2- **calculate_rank**:
Calculate the rank based on the new rank information from DIO message sent by the neighbor or stored otherwise.

```
new_rank = base_rank + rank_increase;
Where base_rank  =  p->rank and
```

3- best_parent

This function is the same as the function best_parent in OF0 implementation. It compares two parents and chooses the neighbor that has a lower rank while considering the hysteresis.

4- **update_metric_container**
This function chooses the metric container that chould be used by the objective function. The function supports only two metric containers which are ETX (RPL_DAG_MC_ETX) and Energy (RPL_DAG_MC_ENERGY).

```
#if RPL_DAG_MC == RPL_DAG_MC_ETX

  dag->mc.type = RPL_DAG_MC_ETX;
  dag->mc.length = sizeof(dag->mc.obj.etx);
  dag->mc.obj.etx = path_metric;

  PRINTF("RPL: My path ETX to the root is %u.%u\n",
     dag->mc.obj.etx / RPL_DAG_MC_ETX_DIVISOR,
     (dag->mc.obj.etx % RPL_DAG_MC_ETX_DIVISOR * 100) /
```

```
RPL_DAG_MC_ETX_DIVISOR);

#elif RPL_DAG_MC == RPL_DAG_MC_ENERGY

  dag->mc.type = RPL_DAG_MC_ENERGY;
  dag->mc.length = sizeof(dag->mc.obj.energy);

  if(dag->rank == ROOT_RANK(dag)) {
    type = RPL_DAG_MC_ENERGY_TYPE_MAINS;
  } else {
    type = RPL_DAG_MC_ENERGY_TYPE_BATTERY;
  }

  dag->mc.obj.energy.flags = type << RPL_DAG_MC_ENERGY_TYPE;
  dag->mc.obj.energy.energy_est = path_metric;

#else

#error "Unsupported RPL_DAG_MC configured. See rpl.h."
```

In our FL-RPL, we have implemented the other metric containers. We will give more details in what follows.

4) Describe the new function that you added to implement FL-RPL

To implement FL-OF, we have added the following files in core/net/rpl:

- rpl-of-fuzzy.c:

This file has the same structure as the implementations of OFO and MRHOF.
It contains the same functions as the rpl-of-etx.c:
1- **calculate_rank:**
the rank is computed by adding the information
2- calculate_path_metric
this function is maintained (from the file rpl-of-etx). It contains the computation of the path ETX.
3- calculate_fuzzy_metric
this function computes the path metric while considering the 4 selected routing metrics.

```
for (m = list_head(p->mcs);m != NULL; m = m->next){
   switch(m->type){
```

```
case RPL_DAG_MC_ENERGY:
  energy = m->obj.energy.energy_est;
  break;
case RPL_DAG_MC_HOPCOUNT:
  hopcount = m->obj.hopcount;
  break;
case RPL_DAG_MC_LATENCY:
  latency = m->obj.latency;
  break;
case RPL_DAG_MC_ETX:
  etx = m->obj.etx;
  break;
```

4- calculate_etx_path_metric
this function simply calls calculate_path_metric.
5- calculate_hopcount_path_metric
this function computes the hop count of the neighbor
6-calculate_latency_path_metric
this function computes the latency of the overall path to the dag root
7-calculate_energy_path_metric
this function computes the energy of the neighbor
8-best_parent: it is the same function as the one located in rpl-of-etx. However, the ranks are compared based on the computation of the function `calculate_fuzzy_metric.` We maintain also the hysteresis in the process of parent selection.
9- update_metric_container: this function updates the metric container. We have defined the metric container as containing four routing metrics instead of 1:

```
static rpl_metric_container_t *energy,
  *etx,
  *hopcount,
  *latency;
```

these metrcis are computed from the previously defined functions calculate_etx_path_metric, calculate_hopcount_path_metric, calculate_latency_path_metric and calculate_energy_path_metric.

- fuzzy.c

This file contains the implementation of the aggregation process.
The aggregation process occurs when the output metric depends on more than two input fuzzy sets.

- fuzzy2.c

this file contains the definition of the rules with if then relationship. In our Implementation, we have defined 27 rules for the quality output metric.

- fuzzy.h

In this file we defined the membership input fuzzy sets and the output fuzzy sets of FL-RPL.
1) Definition of input fuzzy sets:
The four routing metrics membreships are dfined here with the definition of their trapezoidal forms and thresholds.
- Energy : energy_low, energy_avg and energy_full
- ETX: etx_short, etx_avg and etx_long
- Hop count: HC_near, HC_avg and HC_far
- Lateny: latency_short, latency_avg and latency_long

2) Definition of the output fuzzy sets : the quality which ranges from 0 to 100. The quality output set is defined with 9 linguistic variables which are: quality_awful, quality_very_bad, quality_bad, quality_degraded, quality_avg, quality_accept, quality_good, quality_very_good and quality_excellent.

-quality.c

in this file we compute the global output metric which is the result of the defuzzification of the different fuzzy sets. To simplify the implementation, we have combined the latency and the hop count into one fuzzy output which we named Delay, this delay is an input variable to the final fuzzy logic system along with energy and ETX.
We have established 27 inference rules. Table 1 shows the inference rules used in the fuzzy logic system.

Delay	ETX	Energy	Output: quality
near	short	low	good
near	short	average	very good
near	short	high	excellent
near	average	low	low good
near	average	average	good
near	average	high	very good
near	long	low	bad

near	long	average	low good
near	long	high	good
vicinity	short	low	low good
vicinity	short	average	good
vicinity	short	high	very good
vicinity	average	low	bad
vicinity	average	average	low good
vicinity	average	high	good
vicinity	long	low	low bad
vicinity	long	average	bad
vicinity	long	high	low good

far	short	low	bad
far	short	average	low good
far	short	high	good
far	average	low	low bad
far	average	average	bad
far	average	high	low good
far	long	low	awful
far	long	average	low bad
far	long	high	bad

We used the centroid defuzzification which computes the central of gravity of the different fuzzy sets of quality (from very bad to excellent).
This is the equation that we have used:
return (aw*12 + lb*24 + b*36 + lg*48 + g*60 + vg*72 + ex*84)/(vb + b + bb + avg + lg + g + vg);

5) Explain how you integrated your code into RPL routing

1) Integration of FL-RPL in contiki:

- The implementation of RPL in contiki has been used without modifying the RPL code.
- A RPl router has an `rpl_dag` structure in the DODAG to which it belongs. It has a list of `rpl-parent` and a single reference to the preferred parent.
- The objective function in RPL is also part of the `rpl-dag` structure.
- The default implementation supports only one metric container (by default ETX). We extended the DIO structure by a list of four metrics which are etx, energy, latency and hop count.
- We has replaced the default objective function based on the ETX with our objective function based `FL-OF`.
- We extended the length of the DIO message as the metric container contains 4 routing metrics instead of 1. This extension respects the standard specification (see RFC 6550 https://tools.ietf.org/html/rfc6550#page-49)

2) ContikiRPL rank calculation and fuzzy metric:

As mentioned above, the rank is computed in Contiki RPL from the ETX metric.
This metric is updated by RPL directly in the link_metric attribute of each node's parents. A node computes its rank by adding the link_metric of its preferred parent with the rank value obtained in received DIO message.
It will send its own rank into its DIO and thus the rank increases between the node and the sink.
The preferred parent is chosen by its ETX metric and a hysteresis mechanism disable a change of preferred parent if the difference with the new rank is smaller than a given constant.
We have kept the ETX based rank calculation but we use our fuzzy metric to choose the best parent. We maintain the stability just when two parents have the same fuzzy metric value and the same rank. In this case, if one parent is the preferred then it keeps its role. In the other cases, we select the parent with the lower rank or the higher fuzzy metric.

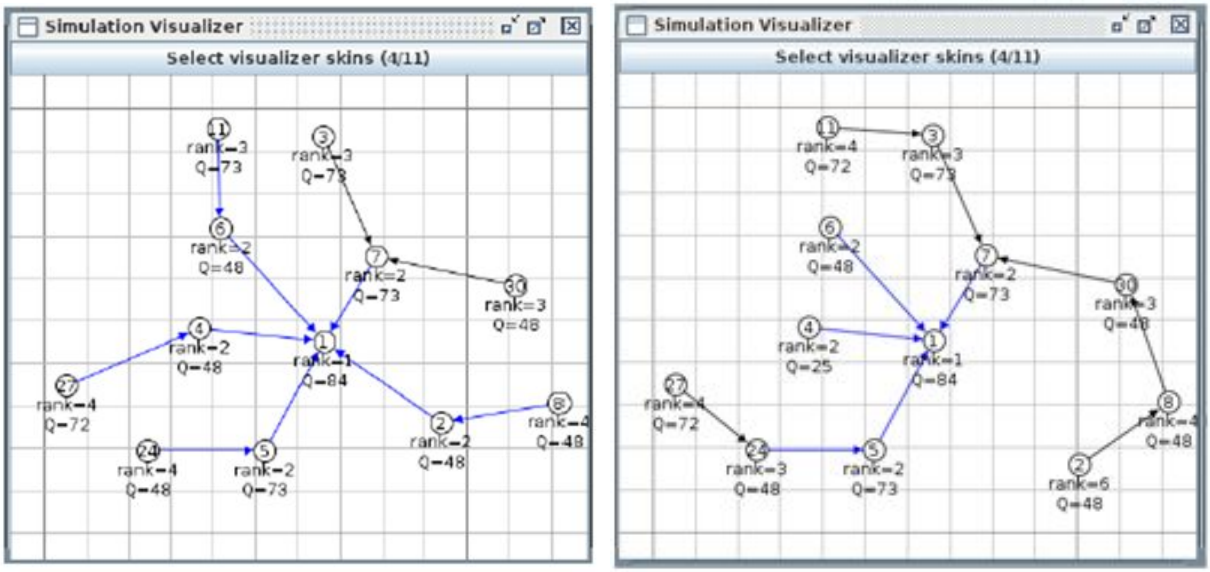
3) DIO message

In the new DIO message, all the metrics are concatenated behind the type 2 for metrics and the total length. The order of the different metrics is not significant as each one is preceded by its Routing-MC-Type that identifies a metric. The additional overhead of this message in comparison with a DIO message composed of a single routing metric is 160 bits, which is equivalent to a ratio from of at most 35% of the DIO size (If the DIO has no more options than the DAG metric container).

6) Explain how did you perform testing of new added functions

6.1) Validation testing:

We have performed several tests to be sure of the correctness of the implemented objective function.
The following figure (in the left) represents the initial topology of an example of RPL network deployed with the proposed OF (after convergence). We enabled the plotting of the rank and the fuzzy metric Q beside each node. Arrows match each node with its preferred parent and node 1 is the DAG root. We can see that our implementation behaves well as each node has a preferred parent with the best rank and best quality.



initial DODAG Final DODAG

With this initial simulation we act on two parameters which are

- The quality for one node (we forced the degradation of the quality of node 4)
- The position of another node (node 2)

In the figure on the right, the topology is changed for these reasons :

- the node 11 has a better parent which is node 3, even if it has a bigger rank. We allow a node to loose one rank if the quality of the parent is better.
- we decrease the quality of node 4. As expected, its quality is not interesting to node 27 anymore.

- we move the node 2 to be out of connectivity with the sink. So the node 8 found a better parent in node 30 and becomes the parent of the 2.

6.2) Defect Testing:

References:

[1] JeongGil Ko; Joakim Eriksson, Nicolas Tsiftes,, Stephen Dawson-Haggerty,Andreas Terzis, Adam Dunkels, and David Culler, ContikiRPL and TinyRPL: Happy Together, IPSN 2011.