RPL

standard and TinyOS implementation

Marica Barrano
Matr. 766185, (marica.barrano@gmail.com)

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: April, 01 2011

Abstract

As known, the WSNs have significant differences compared to the world of the Internet, especially as regards the routing. The crucial aspects are energy constraints, limited processing power, storage motes and topologies of sensor networks that are very different from those of traditional IP networks. It is therefore necessary to find the routing algorithms that know the entire network topology, and use this information in order to find the most convenient route according to the metric used and introduce less overhead as possible. Meet all these requirements simultaneously is a very complicated task and necessarily implies the search for a fair balance between the various conflicting factors. Developing efficient, fast and unexpensive routing algorithms,is a little challenge still open and it's of primary importance to the world of wireless sensor networks. My analysis aims to present the RPL routing protocol and its implementation in TinyOS, a popular operating system in the field of sensor networks.

Introduction

RPL is a proactive distance vector protocol that form a particular structure of a graph in which there is a root node or sink.This protocol has been developed by ROLL (Routing Over Low power and Lossy networks) with the aim of standardizing a new routing protocol that is well suited to the peculiar needs of this type of network. This protocol is based on completely separate management of procedures for processing and forwarding from those of network optimization. RPL support complex objective functions subject to constraints imposed by the actual situation of the network.The optimal solution related to the path is selected based on metric and constraints and represented by a DODAG (Destination-Oriented Direct Acyclic Graph).Each instance RPL has its own objective function, which may include the establishment of multiple DODAG. Different instances implement different objective function.The main goal of RPL is avoid constant updating of topological information caused by constant changing of the sensors network, limiting the information travelling on the network. Within this document I carry out a detailed study of the operation of the protocol RPL, as specified in the draft [1]. This detailed analysis will allow me to present the implementation of routing protocol discussed in TinyOS. Furthermore, it is presented a brief introduction to TinyOS and an analysis of the results of some tests of its performance.

Topology and parameters

First, I have to explain the meaning of the word DODAG. DODAG is a directed acyclic graph (DAG) with a single root node that is the target of all routes (destination-oriented). The RPL protocol create a structure called DODAG among all nodes, a tree whose root is the root DODAG: for this reason a node can play the role of parent, child, ancestor and leaf. A network on which it operates RPL may involve several DODAG according to the needs of the application. In reference to the above, the protocol provides includes 4 parameters:

1. RPLInstanceID: Identifies an instance of RPL. Includes one or more DODAG who share metrics and functions to calculate the rank. A network can have multiple instances of RPL with different IDs, each of which identifies DODAG independent and unrelated based on metrics and different functions. A node can belong to a single DODAG within the same instance RPL.

2. DODAGID: DODAG identifies a root and the corresponding DODAG. The pair (RPLIstanceID, DODAGID) uniquely identifies a DODAG the network.

3. DODAGVersionNumber: identifies together with the two previous version of a DODAG.

4. Rank: establishes an order in a version of DODAG and identifies the location of each node at the root, increases or decreases depending on whether we are moving away or closer to the root node. The value MinHopRankIncrease identifies the minimum increment between a node and its parent in the graph, that for an efficient implementation should be a power of 2.

RPL supports three types of traffic flow: multipoint-to-point (MP2P), point-to-multipoint (P2MP) and point-to-point (P2P).

Metrics

Some implementations may decide to adopt an extremely simple approach based on the use of a single metric with no constraints, whereas other implementations may use a larger set of link and node routing metrics and constraints. New routing metrics and constraints could be defined in the future, as needed. Metrics and constraints are transported in optional field of DIO messages (DAG Metric Container [2] and are therefore optional. The working group has identified some constraints and metrics broken down by node and link. The metric/constraint is specified by the Type field option DAG Metric Container, which can hold values from 0 to 255. The value 0 is not used, while those currently used are:

1. Node State and Attribute
2. Node Energy
3. Hop Count
4. Link Throughput
5. Link Latency
6. Link Quality Level
7. Link ETX
8. Link Color

Metrics that are implemented at this moment are ETX and energy (energy not in TinyOS but in Contiki OS).

Control Messages

Control messages are defined within a package and ICMPv6. ICMPv6 have a header and a body that includes the message itself and a number of possible options.I have a Type field that takes the value 155 and the Code field that takes one of the following codes:

- 0x00: DODAG Information Sollecitation (DIS)
- 0x01: DODAG Information Object (DIO)
- 0x02: Destination Advertisement Object (DAO)
- 0x03: Destination Advertisement Object Acknoledgment (DAO-ACK)
- 0x80: Secure Information DODAG Sollecitation
- 0x81: Secure DODAG Information Object
- 0x82: Secure Destination Advertisement Object
- 0x83: Secure Destination Advertisement Object Acknowledgment
- 0x8A: Consistency Check

DIO

this message is used for the formation and maintenance of the paths towards the root node (broken up). Contains the necessary information for a node to know the existence and configuration parameters of an instance RPL. It also contains the set of nodes that the node can be elected as his father and then joining the DODAG. The configuration parameters are typically initiated by the sink that assumes the role of root in DODAG. The package contains the following fields:

- RPLInstanceID: 8-bit field set by DODAG root with the identifier of the RPL to which it belongs
- Version: full field of 8-bit unsigned set by DODAG root that identifies the version of DODAG
- Rank: Field 16-bit integer that identifies the rank of a node
- Grounded G: is a bit to define whether the DODAG is grounded or if it offers connectivity to nodes necessary for the application to carry out its task.The nodes are called floating not grounded.
- MOP (Mode Of Operation): 3 bits field defined by the root node that establishes the technique used to build the paths from the root to the other nodes. Each node of the DODAG must be able to fulfill the role of routers that is established by the sink and highlighted in this field, otherwise it can engage the DODAG mode only leaf. The various MOP are:
  - 000: no path from the root node
  - 001: not storing mode
  - 010: storing mode without multicast
  - 011: storing mode with multicast

  not storing mode exploits the technique of source routing to build the paths, while the storing mode uses the routing tables.
- DODAGPreference (Prf): Field 3-bit unsigned integer set by the root node that defines the level of preference DODAG (0x00 to 0x07) with respect to another within the instance RPL.
- DSTN (destination advertisement triggered sequence number): 8-bit field unsigned integer, is used for the maintenance of the routes from the root node.
- DODAGID: field of 128-bit unsigned integer that identifies the DODAG RPL within the instance.
- Options:
  - 0x00 Pad1

  - 0x01 PadN
  - 0x02 Metric Container
  - 0x03 Routing Information: carries the same information as an IPv6 ND (neighbor discovery)
  - 0x04 DODAG Configuration
  - 0x08 Prefix Information

DIS

This message can be used in order to request the sending of a DIO packet to another node in order to discover or verify the presence of neighboring nodes.

DAO

this message is used for the construction of the paths from the root node to the other nodes of the DODAG and is sent in unicast mode by a child or by one or more parents. Can be confirmed (but this is optional) from the parents by sending a DAO-ACK in response to the corresponding child. The form contains the following fields:

- RPLIstanceID: 8-bit field that identifies me whole instance RPL and the value is learned from the DIO message.
- K: bit that tells me whether the source of the message expects a DAO DAO-ACK confirmation.
- D: bits indicating the presence or less of the value DODAGID in the package, which must necessarily be present in the case where I DODAG more in the same instance.
- DAOSequence: increased value to each sending and receiving the same in DAO DAO-ACK response.
- DODAGID: optional field 16-bit integer, is present only if the D bit is active.
- Options

DAO-ACK

This message is sent by the parent node to the child node in response to a DAO. The packet format includes the following fields:

- RPLInstanceID: 8-bit field that identifies the entire RPL instance, the value is learned from the DIO message
- D: bits that indicates whether or not the DODAGID is present in the package.Must be present in the case of multiple DODAG.
- DAOSequence: the value is the same content in DAO received.
- Status: If the value is above 128 then it means that the node should select a parent alternative.
- DODAGID: 16-bit field unsigned integer that identifies the DODAG. The field is present if the D bit is active.

OPTIONS

options that can be included in control packets RPL are:

- 0x00 Pad1: allows you to insert one octet of padding in the message with the purpose of obtaining an array of options following the convention adopted for the IPv6 protocol.
- 0x01 PadN: Enter N octets of padding to the message always with the purpose of alignment.
- 0x02 Metric Container: to disseminate and report metric values along the DODAG.
- 0x03 Route Information: is used to indicate if there is connectivity from the root node to a destination specified by a prefix.
- 0x04 DODAG Configuration: is used to spread configuration parameters needed for the operation of the protocol RPL. These variables must be specified from the root node and only he should be allowed to change them.
- 0x05 RPL Target: is used to indicate an IPv6 address for a particular destination.
- 0x06 Transit Information: mainly used when the protocol works in storing mode, is used to indicate the attributes of the paths that match the specified destinations in the RPL Target.
- 0x07 Solicited Information: it is used by a node to request the sending of a DIO message by one or more neighboring nodes.
- 0x08 Prefix Information: allows you to distribute routing prefixes used in DODAG.

More options all have a common header that includes the following fields:

- Option Type: 8-bit field integer that identifies the type of option
- Option Length: 8-bit field unsigned integer representing the length in octets of the option excluding the Option Type and Option Length fields.
- Option Date: field length equal to that specified Length into the option that contains the information specified in the option

Routes creation and maintenance

The paths to the root node are constructed and maintained by RPL through the sending of a DIO packet between nodes of the network. The fields in the DIO packet G, MOP, Prf, Version, and RPLIstanceID DODAGID are set at the root node and the nodes that receive the message must adopt a consistent configuration with these parameters and forward unchanged. The nodes themselves may instead update the fields rank and DSTN.

Discovery of neighboring nodes, selection of parent nodes and DODAG

receiving a DIO message allows a node to discover the existence of a DODAG. The origin of this message is the DODAG root and propagated in the network by other nodes. This message is then received by a node by an arbitrary number of other nodes, which selects between them a set of neighbor and a set of parent. I have chosen the method by which these collections is implementation-defined even if there are rules to follow.They are mostly common sense and say that the neighbor nodes should be accessible directly via the wireless transmission medium, while the parent are those neighbors who have rank strictly less than that of the node in question, which is determined according to the following formula:

$$rank(new) \leq L + DAGMaxRankIncrease \quad (1)$$

where L is the lowest rank of received DIO packets. This value is then propagated to rank in the rest of the network. Between the parent node is selected only preferred (preferred parent) that represents the node connecting the path to the root node (root), which clearly has a set of parent nodes empty. Hence I conclude that the rank decreases as they bring us closer to the root node. This node propagates a rank equal to ROOTRANK which constitutes the minimum value of rank within the DODAG. If there are more DODAG inside RPL instance, a node chooses what to snap on the basis of the functions specified in the implementation (this is application dependent), which, however, must take into account the variable Prf, or by degree of preference has a DODAG than the other.

Version management of a DODAG

a version of a DODAG is defined by the triple (RPLIstanceID, DODAGID, DODAGVersionNumber) established by the root node. The nodes of a parent node must all have the same version as the node itself. In the event that a node receives a DIO with a higher version than the current which then has been updated from the root node, the new one must migrate to and communicate it in the DIO transmitted.

DIO Processing and transmission

when a node receives a DIO packet processes it so as to determine whether the node from which it has received can be inserted within nodes parent and if possibly elect preferred parent.A preferred parent is a parent with rank greater than or equal to that of all nodes that are part of the set of parent (the one closest to the node in question). RPL from the possibility to decrease the traffic control as needed through the algorithm said Trickle Timer. It requires three parameters provided in the DIO packet through DODAG Configuration Option:

- Imin: time of departure of the algorithm
- Imax: its value is the maximum number of doublings of the value Imin and is a natural number
- K: natural number greater than 0 that is the constant of redundancy to these are added other three variables needed to run the algorithm:
- I: current time
- t: time value within the current
- c: counter variable

the initialization of the algorithm is:
I = Imin
c = 0
At the beginning of an interval, which ends after the time unit, t is chosen randomly within the range [I / 2, I). Each time it detects a transmission consisting, the variable c is incremented by one unit and ends when the interval defined by t, the algorithm allows the transmission of DIO is verified only if the condition c <K. Finally, at the end of the interval defined by I, its value is doubled until it reaches the value Imax, in which case remains I = Imax and is set to c = 0. If a program is found inconsistent, the algorithm resets the variables again by setting I = Imin and c = 0. As part of RPL define consistent reception of DIO that does not change the parent's preferred or rank. The transmission is rather inconsistent in the moment in which one has the reception of a DIS multicast without the option Solicited Information or, in the case option is present, the node checks the parameters defined within it before judging inconsistent. The node should reset the Trickle timer when receiving a packet DIS directed specifically to him, but DIO simply send a package with options DODAG Information Option in unicast to the source node, and if there is the option Solicited Information, the form is bound to the coincidence of the parameters in the option itself.

Creating paths from the root node

the management and the creation of paths from the root node to the other nodes is via DAO packets sent towards the root node. This is made possible by the selection of one or more next-hop, called DAO parent, within the set

of DODAG parent. Fields RPLIstanceID and DODAGID Package DAO must have the same values in DIO, while the field DAOSequence is increased to the maximum of one unit for each new DAO sent. The request of a DAO-ACK in response to a message DAO and the consequent activation of the bit K is optional, also a DAO-ACK can be sent even in the absence of K bits enabled when you want to report an error. You can then disable broken down or decide how it is maintained through the DIO field MOP Package. If the value is 0, no node needs to send messages DODAG DAO and ignore them in reception. Since, by definition, the DAO packages go towards the root node, a node that receives one might have to retransmit. The sending of this DAO should not be immediate, but delayed in order to allow the aggregation of data from other nodes.

Non-storing mode

Non-storing mode is an operation mode in which the paths are constructed using the technique source routing without maintaining routing tables in the nodes. The paths are populated with the root node and are constructed using DAO containing options Transit information and RPL Target.In the Parent Field Transit information option specifies one or more DAO parent of a child node, the address of which is nell'RPL Target information that must precede the other option. DAO messages back through the DODAG forward next to the root node, which aggregates information to calculate the paths to each node.

Storing-mode

This mode uses the routing tables of the nodes to determine the paths downwards. At the base are exploited destination and origin of the DAO packets that are exchanged over the network: a node that receives a packet DAO can store, based on the node that it was originated, the node around which allows to reach it. The table and is recorded information of the node that generated the DAO and that of the neighboring node that allows you to achieve it.

Failure Management

Global Repair

for the repair of DODAG there is a mode called global repair, which provides that the root node publishes a new DIO with a version of DODAG increased. This operation allows the nodes to update their rank without being bound to the rank of DODAG previous, so as to avoid possible loops that might come and create in the case where a node would increase its rank within the same DODAG. The decrease in rank rather not hazardous. This mode has the disadvantage of having to wait for a new version of DODAG order to repair a fault.

Local Repair

Another mode of repair of DODAG is the local repair that allows the node to undertake the repair of the fault without waiting for the sending of a DIO with updated version of DODAG of which it forms part. This mechanism is activated when a node becomes aware of no longer being able to communicate with its next-hop, through the non-receipt of one or more ACK confirmation. If communication is interrupted with a preferred parent or simply a parent I have 2 different procedures: in the first case, I have a procedure to search for a new parent preferred while in the second case simply delete the node from the set of parent. In fact, if the communication is interrupted with a preferred parent, the node must first seek an alternative route choosing the new parent preferred among the remaining parent. If there are no other parent in the parent September then the node must try the alternative route between the neighbors. Not all the neighbors, however, go well: a neighbor to become the new parent must have rank equal to that of the node and a reception power value in excess of the threshold. In the event that the search is successful then the node to update its rank and increase it, for this reason shall also send a notice of infinite rank their children to warn them of the change and release them. In the case in which even this search fails, then the node will be forced to wait or the receipt of a new DIO to engage the new DODAG or request the sending of the packet through a packet DIS.

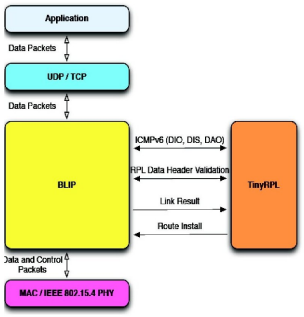

Figure 1: Blip TinyRPL interaction

RPL in TinyOS

The RPL implementations in TinyOS is centered in two layers of the software stack, blip, the TinyOS 6LoWPAN stack, and TinyRPL the actual implementation of the RPL standard (Figure 1). This section introduces the interfaces and the interactions between TinyRPL and BLIP and also the performance of TinyRPL/BLIP compared with CTP (Collection Tree Protocol).

TinyOS

TinyOS is an open source operating system for embedded devices such as motes. It is write in Nesc programming language, a dialect of C. The Kernel has a hierarchical two-layers scheduler: one for events and one for tasks.Events have right of first refusal on other events and tasks. An event could therefore interrupt another event to avoid race conditions requires the use of keywords Atomic. The code structure is based on intefaces, modules and commands whereas files are arranged in configuration files, headers and module files.

BLIP (Berkley Low-Power IP)

Blip, is the IPv6/6LoWPAN stack for TinyOS 2.x for embedded systems. The goal of this implementation is provide a suite to build new real world applications. Blip implements the 6LoWPAN header compression, 6LoWPAN neighbour discovery and DHCPv6 to support the use of IPv6 in the upper layers [3]. Blip also provides a layered IP forwarding abstraction which allows routing protocols such as RPL to be implemented on top of the ICMP engine. Regarding communication, it offers the possibility of two different addressing modes: static address mode and DHCPv6 address mode. In static address mode a static address is configured at compile time adding the unique node ID to the IPv6 prefix. BLIP also offers two link-local address:

1. link local address (type 1): is obtained from the unique EUI-64
2. link local address (type 2): refers to RFC4944

BLIP stack is implemented using 4 different main layers:

1. protocol
2. routing
3. neighbor discovery
4. dispatch

In its interaction with TinyRPL, blip initiates the TinyRPL operations once a node is assigned a global address. Once TinyRPL establishes a route using the RPL-related ICMPv6 messages the route is added to blip's routing table. The forwarding engine in blip makes routing decisions by performing lookups in this table and also includes optional up-calls to the routing layer for each packet being forwarded; this allows the routing protocol to implement non-standard tests for packet forwarding. For instance, TinyRPL checks for an optional routing header to discover the existence of any routing loops. Blip also manages per-interface message queues which are used to buffer outgoing packets, necessary to support bursts of outgoing packets such as those generated from sending a fragmented packet and that caused by head-of-line blocking during retransmissions. Blip version 2.0 provides the implementation of the last standard header compression described in the draft [4]. As regards the consumer, in the Figure 2 we can see that the implementation of Blip and TinyRPL requires a small consumption of resources. For example, the average consumption of the memory depends on the type of downward routing used.

Implementation	ROM (B)	RAM (B)
6LoWPAN header compression	7222	1581
TinyRPL — non-storing mode	9166	308
TinyRPL — storing mode	8428	1454
TinyRPL — no downward routes	6990	264

Figure 2: Memory Usage

TinyRPL

TinyOS, open source software platform for wireless sensor network, is one of the first platforms to provide an open implementation of 6LowPAN and RoLL standards. TinyRPL is the TinyOS implementation of the IETF's IPv6 Routing Protocol for Low-power and Lossy Networks (RPL) and deeply depends on the intefaces provided by Blip. As a sample implementation, TinyRPL supports the RPL draft's basic mechanisms [1], while omitting some of RPL's optional features, such as the security options. TinyOS implements routing support for the three different traffic that RPL supports: P2P, P2MP, MP2P. Once a mote boots up with TinyRPL, TinyRPL will operate in the 'background' of an application to exchange route related messages with all nodes. The RPL Routing Engine begins its operations once a global address is allocated using the DHCPv6 process. Once the RPL routing engine starts exchanging DIO and DAO messages, it can receive packets from the application layers. The packet-sending IP interface can be connected identically to the ways that they are wired in blip. Once the packet reaches the point to discover the next hop address (on the blip stack), RPL's routing table will be called to retrieve the next hop node's IPv6 address for the specified destination.TinyRPL installs its routes separately at blip's packet forwarding module once the DODAG is constructed using DIO messages. Since the routes to any known destination (including the default route) are installed in the forwarding module,upper-level protocols can directly send their messages through the forwarding module in blip. The parent node selection and rank calculation policies are controlled using either Objective Function 0 (OF0) or MRHOF. For both objective functions, a new node is added to the parent set (i.e., the set of

nodes with lower rank values in a node's single hop neighborhood) with a local expected number of transmissions (ETX) value of 3.5. OF0 [5] objective function is essentially a hopcount-based parent selection metric. MRHOF [6] objective function is a minimization of path-ETX values to root node of the DODAG trough the parent node. Due to TinyRPL's modular design, additional objective functions can also be supported easily. TinyRPL updates the ETX values after each unicast transmission. Updating the ETX metric is performed in each node by using an exponentially weighted moving average (EWMA) of = 0.5, calculated based on retransmission attempts occurred for a correct transmission. Whenever a new ETX is set for a node in the parent set, the protocol verify if there are new elements in its potential parent set and select the node with the best metric to be its desired parent in the DODAG.

TinyRPL + BLIP performance

As shown in [7], TinyRPL provide a more efficient routing by its interaction with transport protocol, not included in CTP (Collect Tree Protocol).Tests were performed in a testbed of 51 telosb, in wich one of the 51 nodes is configured as root. Below I show the results of tests comparing the performance of tinyRPL and those of CTP:

- Packet Reception Ratio: the first important result is that TinyRPL and CTP have comparable PRR, always greater than 99.8
- Control Overhead: it has been found that the overhead of TinyRPL control packets is greater (8.96 DIO per hour) than those of CTP (8.29 control packets per hour). This result can be motivated by two explanations. The first reason is that TinyRPL has a parents selection rate greater than those of CTP and therefore produces more overhead. The second reason is that they have a different method for calculating the estimation of the quality of the link. CTP uses a 4-bit estimator while TinyOS uses MRHOF with the metric-path ETX.
- Routing Protocol Overhead: it is the overhead produced by attachment and transport of packets routing headers. This value of the two protocols is comparable.
- Selection of the path that minimize retransmissions: it is the same and high value in both cases.
- Downstream Routing: TinyRPL combined with a transport level protocol can offer a level of connectivity that is the same of real applications need.

code structure

Header (RPL.h)

is located in tos/lib/rpl and contains the definition of the messages exchanged in the network defined by the protocol, which I reproduce below:

DIO

```
struct dio_base_t {
  struct icmpv6_header_t icmpv6;
  struct rpl_instance_id instance_id;
  // used to be instanceID
  nx_uint8_t version; //used to be sequence
  nx_uint16_t dagRank;
  uint8_t flags;
  uint8_t dtsn;
  nx_uint16_t reserved;
  struct in6_addr dodagID; // was dagID
}__attribute__((packed));

struct dio_body_t{ // type 2 ; contains metrics
  uint8_t type;
  uint8_t container_len;
  //uint8_t *metric_data;
};

struct dio_dodag_config_t{
// type 4 ; contains DODAG configuration
  nx_uint8_t type;
  nx_uint8_t length;
  uint8_t flags : 4;
  uint8_t A     : 1;
  uint8_t PCS   : 3;
  nx_uint8_t DIOIntDoubl;
  nx_uint8_t DIOIntMin;
  nx_uint8_t DIORedun;
  nx_uint16_t MaxRankInc;
  nx_uint16_t MinHopRankInc;
  nx_uint16_t ocp;
  nx_uint8_t reserved;
  nx_uint8_t default_lifetime;
  nx_uint16_t lifetime_unit;
};

struct dio_metric_header_t{
  uint8_t routing_obj_type;
  uint8_t reserved    :  2;
  uint8_t R_flag      :  1;
  uint8_t G_flag      :  1;
  uint8_t A_flag      :  2;
  uint8_t O_flag      :  1;
  uint8_t C_flag      :  1;
  nx_uint16_t object_len;
};

struct dio_etx_t{
  nx_uint16_t etx;
};

struct dio_latency_t{
  float latency;
};

struct dio_prefix_t{
  uint8_t type;
  nx_uint16_t suboption_len;
  uint8_t reserved : 3;
  uint8_t preference : 2;
  uint8_t reserved2 : 3;
  nx_uint32_t lifetime;
  uint8_t prefix_len;
  struct in6_addr prefix;
};
```

DAO

```
struct dao_base_t {
  struct icmpv6_header_t icmpv6;
  struct rpl_instance_id instance_id;
  uint16_t k_bit : 1;
  uint16_t d_bit : 1;
  uint16_t flags : 6;
  uint16_t reserved : 8;
  uint8_t DAOsequence;
  struct in6_addr dodagID;
  struct target_option_t target_option;
```

```
  struct transit_info_option_t transit_info_option;
}__attribute__((packed));
```

DIS

```
struct dis_base_t {
  struct icmpv6_header_t icmpv6;
  nx_uint16_t reserved;
};
```

Furthermore, options of the respective packages and the common header of icmpv6 packets are specified.

Interfaces

Note that the application does not need direct connections to any of these interfaces. These interfaces are internally connected to the blip/6lowpan layer . These interfaces are positioned within tos/lib/rpl/.

Interface RPLRank

- RPLRank.nc: Contains the definition of the interface with the list of commands related to the implementation in the file RankP (module);

```
interface RPLRank {
 command void declareRoot();
 command void cancelRoot();
 command bool isRoot();
 command uint16_t getRank(struct in6_addr *node);
 command bool isParent(struct in6_addr *node);
 command void inconsistencyDetected();
 command uint8_t hasParent();
 command bool isLeaf();
 command uint16_t getEtx();
 command bool compareAddr(struct in6_addr *node1,
 struct in6_addr *node2);
 command bool validInstance(uint8_t instanceID);
 event void parentRankChange();
 command void setQueuingDelay(uint32_t delay);
 command error_t getDefaultRoute(struct in6_addr
 *next_hop);
}
```

- RPLRankC.nc: configuration file that contains the mapping between components and interfaces;
- RPLRankP.nc: module file that contains the implementation of all the real protocol specifications relating to the management of the rank and parent.

RPLRoutingEngine

includes the definition of the interfaces, components and commands for maintaining the DODAG:

- RPLRoutingEngine.nc: definition of the interface with its controls used in the module.

```
interface RPLRoutingEngine{
 command void resetTrickle();
 command bool hasDODAG();
 command error_t getDefaultRoute(struct in6_addr
 *next_hop);
 command uint16_t getRank();
 command uint8_t getInstanceID();
 command bool validInstance(uint8_t instanceID);
 command struct in6_addr* getDodagId();
 command void setDODAGConfig(uint8_t
        DIOIntDouble,
        uint8_t DIOIntMin,
        uint8_t DIORedun, uint8_t MaxRankInc
        uint8_t MinHopRankInc);
 command uint8_t getMOP();
 command void setDTSN(uint8_t dtsn);
 command uint8_t getDTSN();
 command void inconsistency();
}
```

- RPLRoutingEngineC.nc: configuration file. Contains the definition of the components and their association with their interfaces.
- RPLRoutingEngineP.nc: module that implements the routing within the DODAG and commands defined in the interface definition files RPLRoutingEngine.nc. In particular, in this file are implemented the tasks of sending messages across the network (eg DIO, DIS) but does not implement the DAO which are implemented in a file devoted specifically to this type of messages. It also implements all operations performed on the DIO message and operations triggered for the maintenance of the network and fault management. In particular, the DIO packet is created through a series of successive assignments and a bundling carried out with the use of a vector. In detail, the code is as follows, where we see the distinction between operating modes with respect to the objective functions MRHOF (with metrics) and 0OF (without metric).

```
#ifdef RPL_OF_MRHOF
   length = sizeof(struct dio_base_t)
   + sizeof(struct dio_body_t) +
   sizeof(struct dio_metric_header_t)
   + sizeof(struct dio_etx_t) +
   sizeof(struct dio_dodag_config_t);
   ADD_SECTION(&msg, sizeof(struct dio_base_t));
   ADD_SECTION(&body, sizeof(struct dio_body_t));
   ADD_SECTION(&metric_header,
   sizeof(struct dio_metric_header_t));
   ADD_SECTION(&etx_value,
   sizeof(struct dio_etx_t));
   ADD_SECTION(&dodag_config,
   sizeof(struct dio_dodag_config_t));
#else
   length = sizeof(struct dio_base_t) +
   sizeof(struct dio_dodag_config_t);
   ADD_SECTION(&msg, sizeof(struct dio_base_t));
   ADD_SECTION(&dodag_config,
   sizeof(struct dio_dodag_config_t));
#endif
#undef ADD_SECTION
      }
      // TODO: add prefix info (optional)
      v[0].iov_base = (uint8_t*)&data;
      v[0].iov_len = length;
      v[0].iov_next = NULL;
      pkt.ip6_hdr.ip6_nxt = IANA_ICMP;
      pkt.ip6_hdr.ip6_plen = htons(length);
      pkt.ip6_data = &v[0];
      //iov_print(&v[0]);
    } else {
      length = sizeof(struct dio_base_t);
      pkt.ip6_hdr.ip6_nxt = IANA_ICMP;
      pkt.ip6_hdr.ip6_plen = htons(length);
      v[0].iov_base = (uint8_t *)&msg;
      v[0].iov_len  = sizeof(struct dio_base_t);
      v[0].iov_next = NULL;
      pkt.ip6_data = &v[0];
}
```

Morover the timer for sending DIOs is set through the following lines of code:

```
tricklePeriod = 2 << (DIOIntMin-1);
randomTime = tricklePeriod;
```

```
randomTime /= 2;
randomTime += call Random.rand32() % randomTime;
call TrickleTimer.startOneShot(randomTime);
```

the same procedure is adopted for DIS packets that I need in the event that a node wants to request the sending of DIO packets. RPL provides that a node can solicit its neighbors to send DIO packets to be able to attach to DODAG, which is done to avoid having to wait every time the periodic sending of DIOs which can have a relatively long period. It is therefore set a timer expiration by a periodic sending of DIOs, after which, if the event has not yet occurred, it will automatically send DIS packets. In the code this condition is implemented as follows:

```
call InitDISTimer.startPeriodic(DIS_INTERVAL);
```

RPLDAORoutingEngine

includes interface definition and components and the module that implements the routing of the only DAO packets. Specifically we have:

- RPLDAORoutingEngine.nc: file that defines the commands for the mentioned interface.

```
interface RPLDAORoutingEngine{
 command error_t startDAO();
 command bool getStoreState();
 command void newParent();
}
```

- RPLDAORoutingEngineC.nc: is the configuration file which lists the interfaces provided and used, in addition to doing the component-interface wiring.
- RPLDAORoutingEngineP.nc: is the file that implements the module DAORoutingEngine. Here, are implemented all the rules of creation and routing of DAO packets. As all the messages defined up to now, also the DAOmessages provides a timer that sets the frequency with which they are sent. In the code, the initial value of the sending rate is set to 10000 and updated according to the rule:

```
  if (dao_double_count < dao_double_limit) {
     dao_rate = (dao_rate * 2) +
     call Random.rand16()%100;
     dao_double_count ++;
   }
call GenerateDAOTimer.startOneShot(dao_rate +
call Random.rand16()%50);
```

Because DAOs flow upwards, receiving a unicast DAO can trigger sending a unicast DAO to a DAO parent.

1. On receiving a unicast DAO message with updated information, such as containing a Transit Information option with a new Path Sequence, a node SHOULD send a DAO. It SHOULD NOT send this DAO message immediately. It SHOULD delay sending the DAO message in order to aggregate DAO information from other nodes for which it is a DAO parent.
2. A node SHOULD delay sending a DAO message with a timer (DelayDAO). Receiving a DAO message starts the DelayDAO timer. DAO messages received while the DelayDAO timer is active do not reset the timer. When the DelayDAO timer expires, the node sends a DAO.
3. When a node adds a node to its DAO parent set, it SHOULD schedule a DAO message transmission.

```
command void RPLDAORouteInfo.newParent()
{
 /*
  dao_rate = INIT_DAO;
  dao_double_count = 0;
  call GenerateDAOTimer.stop();
  call GenerateDAOTimer.startOneShot(dao_rate);
  */
  post initDAO();
}
```

where the task initDAO () is used to go to see if there are messages to be sent. First checks to see if there are messages in the buffer allocated for sending operations through a SendPool.get (). In the case in which there are then puts them in another buffer for sending, while otherwise initializes the values for a new packet to send and puts the packet in a waiting queue buffer.

RPLParentTable

contains the definition of a single command, which is:

```
command parent_t* get(uint8_t parent_index)
```

RPLRoutingC

it is the configuration file in which components are associated with their interfaces of the routing part implemented in the following files denoted as RPLRoutingEngine.

RPLOF

This interface consists of 3 files in which the Objective Functions for the DODAG creation is defined. The criterias for creating a DODAG are mainly 2:

- minimum rank objective function with hysteresis (RPLMRHOFP.nc). MRHOF uses hysteresis while selecting the path with the smallest metric value. Its goal is to determine the least-cost path from the node to the root of DODAG and at the same time prevent an excessive disruption of the network. To do so passes to the path of minimum cost only in the case where the cost of the current path exceeds a certain threshold the cost of the path of minimal cost. Makes use of metrics that can be related to the link or to the node. The optimum value of the objective function is calculated on a chosen metric, ETX at the moment.

In addition implements the rules for the selection of the parent node and the calculation of rank.

- zero objective function (RPLOF0P.nc) Objective Function 0 is designed to find the nearest Grounded root. The algorithm does not use the metric details but refers only to the information contained in the packages refer to DIOs and rank preference DODAG. Finally implements the selection rules of the parent node and updating the rank.

In addition there is the RPLOF.nc file that contains the definition of the interface commands.

RootControl

TinyRPL uses the generic RootControl interface to set the root of the RPL DODAG. For the node that will act as the root of the routing tree (e.g., the edge router), the implementation file should use the RootControl interface. The RootControl interface should be connected to the RPLRoutingC component or the RPLRoutingEngineC component.

Conclusion

RPL protocol is implemented in TinyOS based guidelines of the draft [1] . At the moment only two objective functions are implemented and only ETX metric. Also TinyRPL supports only a single RPLInstanceID while supporting multiple DODAGIDs and it does not support security options. Due to the configuration of TinyRPL, RPL require the interfaces of BLIP to interface with the link layer. For this reason packets can then be sent through a transport layer component such as the UDP component supported by BLIP 2.0 (the interaction between TinyRPL and Blip is shown in figure 1). Currently TinyRPL is only supported on the telosb and epic platforms. Therefore, TinyRPL is not supported on the TOSSIM simulator which requires a micaz binary.The only simulation environment available for TinyRPL simulation is Cooja. All the tests regarding RPL protocol implemented in TinyOS are simulated by Cooja platform of Contiki OS. In fact, this platform can run TinyOS applications by selecting the result of the compilation of the mote telosb. The application to test the RPL is called TestRPL but you can also use other applications such as PPPRouter combined with UDPEcho. TestRPL currently does not work correctly on Cooja simulator.

Future works

RPL protocol in TinyOS could be improved and thus increase its performance, going to implement other metrics that produce different objective functions and implementing security options. Purpose might be going to make different DODAGs based on the resource and not on minimization of the energy used to maintain the paths. This would lead to a more smart routing.

References

[1] Winter, T., Brandt, A., Clausen, T., Hui, J., Kelsey, R., Levis, P., Pister, K., Struik, R.: Rpl: Ipv6 routing protocol for low power and lossy networks, draft-ietf-roll-rpl-19. (March 13, 2011)

[2] Vasseur, J.P., Kim, M., Pister, K., Dejean, N., Barthel, D.: Routing metrics used for path calculation in low-power and lossy networks. Internet Engineering Task Force (IETF) (March 2012)

[3] Ko, J., Dawson-Haggerty, S., Gnawali, O., Culler, D., Terzis, A.: Evaluating the performance of rpl and 6lowpan in tinyos. (April, 2011)

[4] Ko, J., Terzis, A., Dawson-Haggerty, S., Culler, D.C., W.Hui, J., Levis, P.: Connecting low-power and lossy networks to the internet. (April 2011)

[5] Thubert, P.: Rpl objective function 0, draft-ietf-roll-of0-05. ROLL Internet-Draft (July 9, 2011)

[6] Gnawali, O., Lewis, P.: The minimum rank objective function with hysteresis, draft-ietf-roll-minrank-hysteresis-of-00. Networking Working Group Internet-Draft (April 14, 2011)

[7] Ko, J., Eriksson, J., Nicolas, T., Dawson-Haggerty, S., Terzis, A., Dunkels, A., Culler, D.: Contikirpl e tinyrpl: Happy together. Proceedings of the workshop on Extending the Internet to Low power and Lossy Networks (Chicago, USA, Apr. 2011)