UNIVERSITÀ DEGLI STUDI
DI TRENTO

Load Balanced Routing Protocol

Wireless Sensor Networks

Fabrizio Zeni
153465

Abstract

This report describes the project developed for the wireless sensor networks laboratory. It starts from the project specifications and continues through the implementation choices. It ends with the experimental results and their analysis.

Contents

1 Introduction

1.1 Project Specifications

The project specifications ask to implement a *Load Balanced Routing Protocol* (LBRP) over *TOSSIM* which is capable to route the data traffic through a wireless sensor network up to a node called **sink**, which is responsible of the data collection. Each node is responsible to mantain its own "routing table", which in this case is represented by a list of *parent nodes* as the next-hop node toward the sink. To balance the traffic among the parents, the node has to check the amount of forwarded message of each parent and verify, for each pair of parents, the following formula:

$$0 \leq msg_{p_a} - msg_{p_b} \leq 1$$

where msg_{p_i} are the messages forwarded to the parent p_i of the node.
The picture below (Figure 1), shows how the LBRP should act on a simple network.

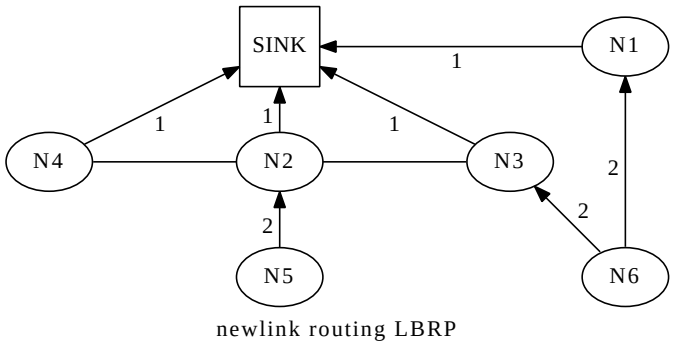


Figure 1: Example of load balancing routing

2 Project Implementation

2.1 File structure

In the project root are present the following items:

- *Tree* directory, which contains the implementation of the application using the Tree-based routing
- *Graph* directory, which contains the implementation of the application using the LBRP
- *results* directory, which contains the results obtained from some simulations
- *test*[1] bash script, used to perform sequential tests
- *statistics.awk* and *mean.awk*, two AWK script used to manage the data gathering through simulations

The Tree directory contains almost the same files of the Graph on, so let's have a look to the latter:

- *.py, python scripts to run simulation on specific scenarios
- *.out, topologies for specific scenarios
- *DataToNetwork.nc*, interface which provides communication from the data layer to the network layer
- *DataLayerC.nc*, data layer configuration
- *DataLayerP.nc*, data layer component
- *DataMsg.h*, header file for the data layer
- *Debug Makefile Reliable Remove*, makefiles that will be better described in the next subsection
- *NetworkLayerC.nc*, network layer configuration
- *NetworkLayerP.nc*, network layer component
- *NetworkToData.nc*, interface which provides communication from the network layer to the data layer
- *RoutingMsg.h*, header file for the network layer

[1] test bash script: described in Section 2.6.1 at page 10

2.2 How to run the project

Looking at the project root, the code is contained mainly in two directories: **Graph** and **Tree**. The implementation of the LBRP is in the Graph one, while the other contain the *single-parent* implementation, which was used to make some comparison over the developed protocol. The application is *modular*, in the sense that from the same code is possible to compile any subversion of the application. That is made possible through some *preprocessor conditional branches* inside the code (*#ifdef keyword*), which are triggered when the compilation is done using a specific makefile. These are the makefiles defined so far:

- Makefile - the default one, which compiles the standard version of the application
- Reliable - which compiles the reliable 1-1 communications version
- Remove - which compiles the Cascade Parent Removal version
- Debug - which compiles the debug version of the program

If the version to compile is the standard one, it will be suffies to run "*make micaz sim*", else the command to input will be "*make -f* ***makefile_name*** *micaz sim*", where **makefile_name** is the makefile of the version to compile.

Then to run the compiled program, one of the python scripts has to be chosen.

- 25.py - this script will execute the script over a topology generated through the topology generator contained in tinyos, with 25 nodes
- 49.py - as the previous, the topology was generated, but this time it has 49 nodes
- newlink.py - executes the script over an ad-hoc scenario, built to check the response of the protocol
- test.py - as for the previous, executes over an another ad-hoc scenario
- stable.py - executes the script on a toplogy of 8 nodes with a stable topology and a good connectivity

To run the simulation type "*python name_of_script*" and the simulation will begin. To simplify this procedure, I prepared a *bash script*, which is described in section 2.6.1.

2.3 Protocol description

The difference between a tree routing protocol and the one implemented, can be seen in the following example.

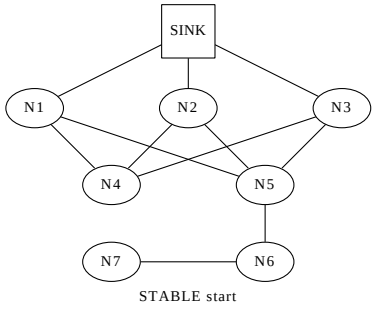


Figure 2: Stable scenario start

This is a simple topology, which will be used for some tests, with 3 layer[2] 1 nodes (1,2,3), 2 layer 2 nodes (4,5) and a chain of two nodes as descendants of node 5.

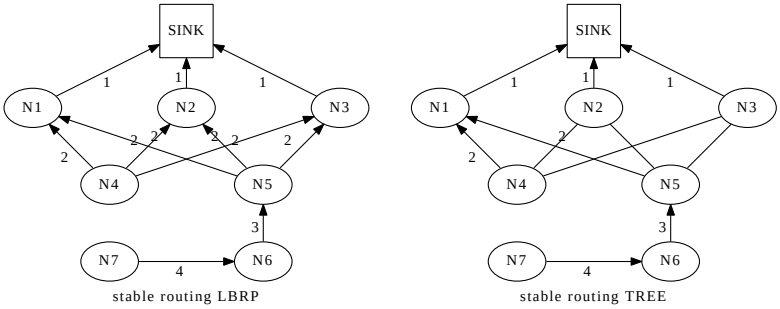


Table 1: Comparison: Tree and LBRP

The differences of the two protocols can be easily seen in Table 1, in fact the tree protocol builds the routing chain by selecting the lower ID parent among candidates with the best cost, while LBRP chooses all those candidates and balance the traffic following the formula written in Section 1.1 at page 1.

[2]layer X: means that the node/group of nodes can reach the sink in X hops

2.3.1 LBRP structures

The first thing to take into consideration when coding a multiple-parent routing is to fix a structure to mantain the information. I defined the following structure:

```
typedef nx_struct Parent{
  nx_uint16_t id; // Node ID of the parent
  nx_uint16_t forwarded; // messages forwarded from the parent
  #ifdef BUFFER_METRIC
  nx_uint16_t buffer; // buffer usage of the parent @ last beacon
  #endif
  #ifdef TRAFFIC_METRIC
  nx_uint16_t traffic; // traffic on the parent @ last beacon
  #endif
  #ifdef ALIVE
  /*
  * state of the link to the parent
  * 3: the link is ALIVE
  * 2: the link is on YELLOW ALERT, the data layer will still try
      to send DataMsg
  * 1: the link is on RED ALERT, the data layer will send an
      AliveMsg instead of a DataMsg
  * 0: the link is DEAD, then it can be substituted
  */
  nx_uint16_t state;
  #endif
} Parent;
```

Listing 1: Parent Structure of LBRP

Actually the paremeters used are the ones out of any preprocessor guards: *id* and *forwarded*, which makes the structure as simple and straightforward as possible. The other parameters were supposed to be used for further developments of the protocols, but by now they are just coded in the header file.
The RoutingMsg/Beacon looks like the structure:

```
typedef nx_struct RoutingMsg{
  nx_uint16_t seq_no;
  nx_uint16_t metric; // from the node to the sink (hop-count)
  #ifdef BUFFER_METRIC
  nx_uint16_t buffer; // actual usage of the buffer
  #endif
  #ifdef TRAFFIC_METRIC
  nx_uint16_t traffic; // traffic on the node
  #endif
} RoutingMsg;
```

Listing 2: Routing message

The id of the sender is just a redundant information, because it can be retrieved by calling *AMPacket.source* over the received message, while we need a *sequence number* to organize the beacons with an ordering dictate by the sink.

2.3.2 LBRP flow

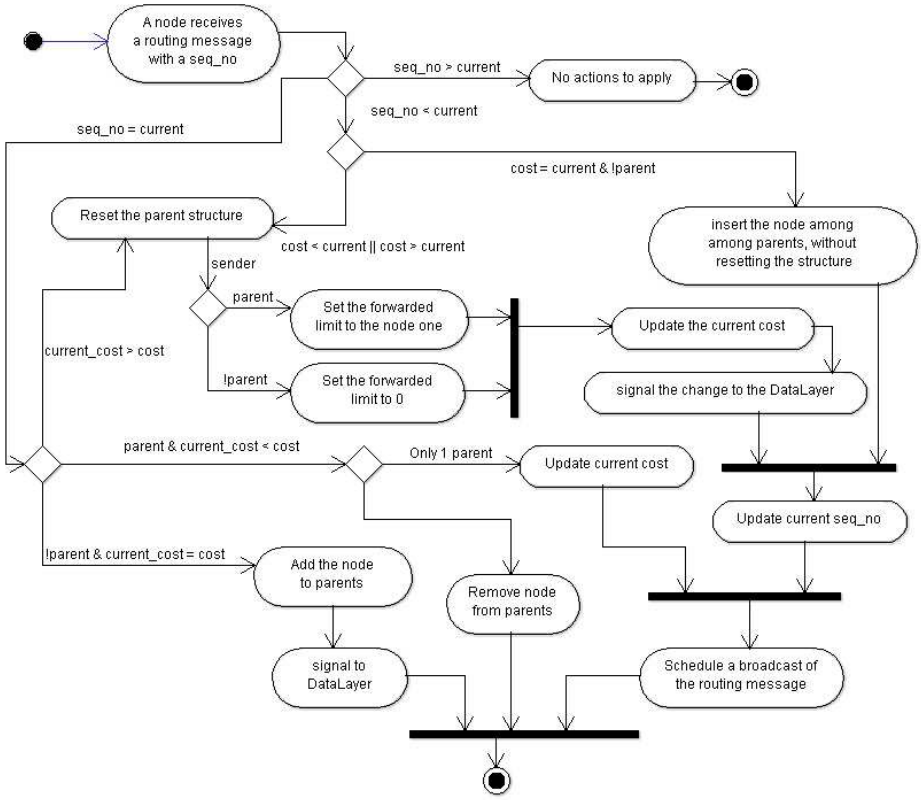


Figure 3: Activity Diagram of the LBRP

The protocol flow is described in Figure 3. An important thing that should be noticed, is that a node will broadcast a routing message, (almost) only when its current cost to the sink is somehow changed. This avoids the genearation of traffic which should not give any benefit to the network. The flow start from the receipt of a RoutingMsg, then the flow continues only if the *sequence number* is greater or equal to the current stored by the receiving node. In both cases the next branch checks the cost, lets look first to the case when the sequence number is equal to the current (which means that a a message with that seq_no has already reached the node).

Sequence number equal to the current

If a message with a *lower* cost reaches the node, then there is a new better path to the sink and then the structure is *reset* (because it might contain parents with the previous cost). I will describe the rest of the flow from the reset when

dealing with the other major branch regarding the sequence number lower than the current.
If the cost is *equal* to the current and the sender is not a parent, then the node is added to the parents and the new parent is signalled to the DataLayer, because it is the newest parent, and it is likely that some messages should be forwarded to it, in order to achieve the same load of older parents. There is no need to have a branch for the case in which the sender is a parent because it will not add any new information and is a wast of time and resources.
If the cost is *greater* than the current and the sender is a parent, then there is another branch checking the number of parents in the structure. If more than one parent are present, then the sender is just removed, because there is at least another parent that can reach the sink with the current cost. Otherwise an update of the current cost is performed and a broadcast of a RoutingMsg is scheduled to advertise the neighbors of the change.

Sequence number lower than the current

If the cost is *equal* to the current and the sender is not a parent, then the node is added to the parents, an update of the sequence number is performed and a broadcast of a RoutingMsg is scheduled. The broadcast can be skipped, because no changes are made when this case occurs, but I just kept this procedure because the broadcast is made for the other the case of the branching.
If the current cost is *greater* or lower than the current, the structured is reset. The reset is performed even if the incoming cost is greater, on the assumption that if a message with a greater cost has reached the node before the ones with the same cost, probably there were some change on the network and so to mantain a path to the sink, a parent with a greater cost in this case is acceptable. Then there is a step that depends on whether the sender is a parent or not. If it is a *parent*, the algorithm takes the past forwarded message count to the node and set that for the new position of the parent (head of the structure), *otherwise* the messages are set to 0 and the flow can go on. An update of the cost is performed, the new parent is signalled to the data layer, the current sequence number is updated and finally a broadcast is scheduled.

2.4 Implemented Features

During the development of the project, I inserted into the project several features (modes), which alter the behaviour of the protocol. The major difference stands in the reliability of the communication, in fact there is the baseline versione which has estEffort 1-1 communications, while I developed another version with Reliable 1-1. I decided to develop several version in order to run them over different topologies and compare their results.

2.4.1 Best-Effort 1-1

This was the requested mode for the developed project, which substancially does not take care of any retransmission in a best effort trasmission flavour. So that any trasmission to a parent will be counted, without taking into consideration its result. As said in the section 2.2, to compile this version it would be suffice to type "make micaz sim"

2.4.2 Reliable 1-1

Thanks to the use of acks and their managment, this version work over a reliable 1-1 channel. In this case, only successful communication are counted for each parent. The differences in the code are almost only in the DataLayer module.

2.4.3 CPR - Cascade Parent Removal

The Cascade Parent Removal is a feature which can be used by the Reliable 1-1 version. It comes from the idea that when the communications to a parent are unsuccessfull for a given number of sequential attempts, it ends up in the assumption that such parent is no more reachable (might be dead, moved away, etc). But as the name might suggest, this feature does a bit more, infact whenever a parent is removed following the procedure before and that parent was the only one the node have, a NO_PARENT_MSG is broadcasted. Each node that will receive such message, will check if the send is among its parents and in that case the sender will be removed from the parents. This procedure will be repeated until a node which has a "backup" route is found (or even a node which is not the son of the sender of the NO_PARENT_MSG), and this explains the term *cascade*. Furthermore, CPR implements a *recovery* procedure, so that when a NO_PARENT_MSG reaches a node that has a "safe" route, that node will schedule the broadcast of a routing message, so that the orphan node(s) (a cascade might have happened) will eventually rebuild a path to the sink.

2.5 Not-Yet-Implemented Features

Here I list some features that I started to implement, but by now are not yet completed. They are listed just for explain why in the code there might be some reference to structures, methods,etc that might seem not connected to the goals of the project.

2.5.1 Alive Messages

The Alive procedure was one of the first features on which I started to work at the begining of the development. It can be said that it was the *parent of CPR*. However, Alive was rather much complex than CPR, it involved both Data and NetworkLayer, and even a specific *timer* was used. The NetworkLayer was *constantly* looking at the *state* of any parents, so that when a communication from a parent is received, the state of such parent is set to ALIVE, on the contrary whenever the alive timer is triggered, the state of each parent is decremented. For each value of the state of a parent, a behavior is codified, such that:

- 3 (ALIVE): the parent is alive, the normal behavior is allowed
- 2 : tollerance level, the behavior will be kept as normal even in this stage
- 1 : the absence of parent communications is rather suspicious, stop data communications, send an AliveMsg
- 0 (DEAD): the parent is marked as dead, it can be removed and/or substituted

An Alive Message, was a message with a minimal dummy payload, an integer containing the nodeId of the sender, for which a acks is explicitly requested. In the original idea, the Alive mechaninsm was planned to be applied to all version, so that even with the Best-Effort a way to substitute broken/bad links would be available.

2.5.2 Other Balancing Metrics

The balancing of the load over a count mantained locally by each node, could be improved somehow. That was my tought, so I started thinking what can be exploited to have a better view of the congestion of parent nodes. I reached two possibilities: **buffer usage** and **traffic** of parent. The first idea comes from the fact that if we can retrieve the buffer usage of each parent, it would be nice to send the traffic to the one with the *highest buffer availability*. Moreover I looked over the buffer usage and it turned out that with my settings for the timers that *send messages* (from the buffer) and *generate messages* (insert in the buffer), which are respectively set to *5s* and *10,5s*, if we use a time τ_i, where $\tau_{i+1} - \tau_i = 5s$, a buffer of size n, will be filled at time τ_n when having two descendants (e.g. two children or a child and a grandchild). So a tuning over the buffer might be a good idea.
The traffic metric, looks over the forwarded messaged FROM each parent. This is much more general than the previous, but can anyway give the flavour of the congestion of a node.
Both the metrics are quite easy to implement, but the problem consists in the rate at which the refresh of that information is provvided to the children. In fact if it is too large (e.g. beaconing period) the information could became rather old and the balancing cannot be such effective, on the other hand if the refresh happens too often it could introduce too much overhead into the network.

2.5.3 Rebuild Messages

This was meant to be an *extension* of CPR, where a **request of rebuild** (ROR_MSG) would be routed to the sink in order to ask for a new *beacon*. For the *rebuild period* I thought at two possibilities:

- Single rebuild beacon, the sink sends a single rebuild beacon for each request
- Adaptive beacon, inspired to the one of *CTP*[3] (which is itself inspired to *Trickle*[4]), but applied to the receipt of rebuild messages, so that whenever a ROR_MSG is received by the sink the beaconing period is set to a minimum, while at each activation of the beaconing timer, the period is doubled up to the max value.

2.6 Tools

2.6.1 Bash script

I wrote a bash script to make easier the job of running series of tests and perform data analysis. By typing "./test help", the script returns a small documentation. The script call format is "./test test_mode number_of_tests number_of_nodes name_of_test python_script_name [compile]" where *test_mode* can be:

- normal, for running the best-effort version of the project
- rel, for running the reliable version of the project
- remove, for running the CPR version of the project
- tree, for running the best-effort tree-single-parent version
- reltree, for running the reliable tree-single-parent version

The *python_script_name* is the name of the python script used for the simulations and could be one of those named in Section 2.2.

2.6.2 AWK

The AWK[5] language was used in order to work over the data gathered by the simulations and make some stastistics over them.

2.6.3 DOT

I used DOT[6] language to give a graphical description of the network graphs

[3]Collection Tree Protocol: http://www.tinyos.net/tinyos-2.x/doc/html/tep123.html
[4]Trickle Algorithm: http://www.cs.berkeley.edu/ pal/pubs/trickle-nsdi04.pdf
[5]AWK wikipage: https://en.wikipedia.org/wiki/AWK
[6]DOT wikipage: http://en.wikipedia.org/wiki/DOT_%28graph_description_language%29

3 Experimental Results

In this section I will show the result of a series of sequential test ran for all version of the project. Each subsection contains the results of a specific scenario, whose detailed description can be found in *Appendix A*. In order to do a comparison, for each testng session I took the forwarded message to each parent from each node (as suggested by the project specifications) and the number of arrived packet for each node, which can rate the reliability of the tested version. Anyway, please note that only some results are reported here, but the complete list of results can be seen in the subdirectory *results* of the project.
In this section the names of the tests are shortened to fit into tables, so that:

- **betree**, for the best-effort 1-1 version of the tree procotol
- **belbrp**, for the best-effort 1-1 version of the LBRP
- **reltree**, for the reliable 1-1 version of the tree protocol
- **rellbrp**, for the reliable 1-1 version of the LBRP
- **cpr**, for the CPR version of the LBRP

The tests were done with the parameters given in the project specification, so the *send period* was set to 5sec, while the **beacon period** was set to 5minutes. Moreover these were some parameters set by me:

- **data generation period**, the period at which each node generates its own data to send, set to 10.5s (something more of the double of the *send period*, to avoid that the buffer will become full too early.
- **length of the simulation**, set to 20 minutes (=1200000ms), so that we will be able to see about 4 beacons, and keep a reasonable time of simulation
- **number of tests**, for the smaller topologies (newlink,test,stable) for each protocol were done 200 sequential simulations, while for the topologies made of 25 and 49 nodes the simulations were 100

The choises of the *simulation length* and the *number of tests* were driven by the need to keep the "real" time of each simulation "brach" as much reasonable as possibile. The problem was due to the fact that the time spent for each simulation was almost proportional to the number of nodes, so on the 49 topology 1 simulation took about 3'39", which implies that the "pack" of simulations (for one protocol) took 6 hours and 5 minutes. Then simply multiply it for 5 (the developed version of the protocols) and we get that for getting the data for the topology, it was needed 1 day, 6 hours and 25 minutes.

3.1 Newlink Results

In this scenario, the connectivity in the network is subject to some changes, infact 3 links are removed in the early phase of the simulation (30% of the simulation time), and two links are added respectively at 55% and 60%.

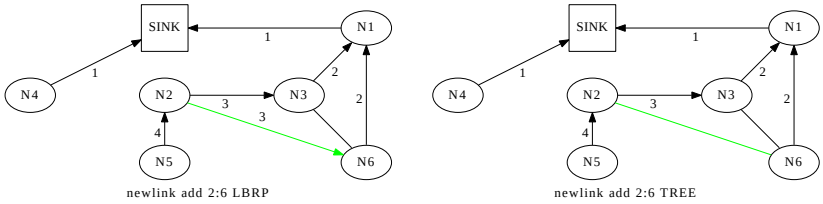


Table 2: Newlink Final stage

When the network reaches its final configuration, the different behavior between the Tree protocol and the LBRP can be easily seen from the Table 3:

TEST NAME	**FORWARDED MESSAGES TO**							
	0	1	2	3	4	5	6	TOT
betree	54.94	-	-	118.58	-	-	∼1	174.41
belbrp	54.82	-	-	59.22	-	-	47.88	161.92
reltree	9.30	-	-	103.92	-	-	∼1	113.98
rellbrp	9.22	-	-	59.76	-	-	45.85	114.83
cpr	9.33	-	-	96.08	-	-	32.72	138.13

Table 3: Messages forwarded by node 2

It can be seen that both versions of the tree protocol report only an average of forwarded messages to 6 less to 1, probably due to some temporary problem inside the network. The majority of messages are forwarded to 3, because it is the lower ID parent candidate. On the other hand, when the link between 2 and 6 came up, 6 is added as parent and in order to balance the load, the messages are sent almost to 6 in order to bring the two parents to the same level of load.

MSG FROM	**TEST NAME**				
	betree	belbrp	reltree	rellbrp	cpr
2	13.06	11.48	22.08	19.48	30.48

Table 4: Messages arrived to sink from 2

The table above (Table 4) shows the number of messages originally sent by 2 which reached the sink. The reliable versions got a way better deliver rate if compared to the forwarding messages report in Table 3. In particular, the CPR

version did a really good job by having more than the double of best-effort versions and one third more than reltree and rellbrp.

3.2 Test Results

Here the same 3 links of Newlink are removed in the early phase, while a new node 7 is added in the late phase of the simulation (75%). The node is configured to have two links one to the sink and one to the periphery of the network (node 5), which should force the network to reconstruct the routing schema. Moreover the link between between 1 and the sink has a really low gain, which makes the communication chain till the sink a bit problematic.

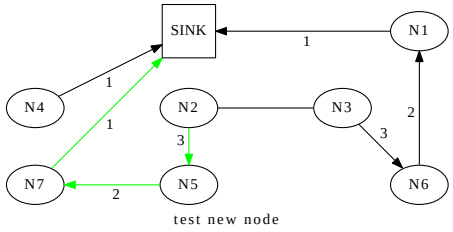


Figure 4: Final configuration of Test scenario

Even in this case, could be interesting to see the forwarded messages from 2:

TEST NAME	FORWARDED MESSAGES TO								
	0	1	2	3	4	5	6	7	TOT
betree	57.42	-	-	116.14	-	1.56	-	-	175.13
belbrp	52.95	-	-	105.03	-	2.44	-	-	160.41
reltree	10.52	-	-	107.33	-	1.23	-	-	119.07
rellbrp	10.57	-	-	109.49	-	2.86	-	-	122.92
cpr	10.63	-	-	55.70	-	4.8	-	-	71.14

Table 5: Messages forwarded by node 2

In the table above, the gap of *total forwardings* between cpr and the others is quite big, and that need some investigation over the topology. Anyway with a bit of debug it turned out, that the loosy gain between 1 and the sink, which was the gateway for all the network until the arrival of 7, was the cause of the trigger of the cascade parent removal, which starting from 1 stopped all the network until the next successful beacon. Another thing that can be said from the Table 5 is that the tests with LBRP and its variants seems to be faster to adapt to the topology changes.

3.3 Stable Results

This test was done on a stable topology with a good connectivity.

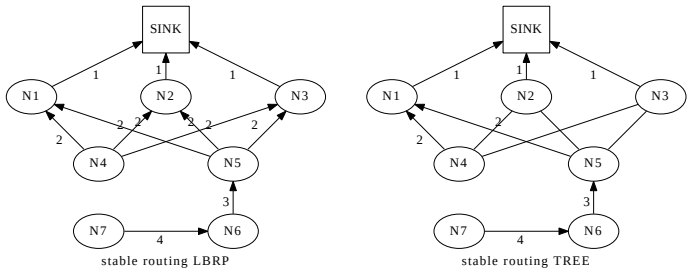


Table 6: Stable Routing

This time the interesting part of the test resides in the node 5, because, as can be seen in Table 6, the forwarding results of the node 5 can show the differences between the routing protocols and because it is the head of a node chain (5<-->6<–>7), which can show how the protocols behave on chain of nodes.

TEST NAME	**FORWARDED MESSAGES TO**								
	0	1	2	3	4	5	6	7	TOT
betree	-	183.01	-	1.99	-	-	-	-	185
belbrp	-	61.19	61.91	61.90	-	-	-	-	185
reltree	-	156.97	-	1.73	-	-	-	-	158.74
rellbrp	-	52.27	52.70	52.34	-	-	-	-	157.31
cpr	-	50.77	50.51	49.49	-	-	-	-	150.86

Table 7: Messages forwarded by node 5

The two besteffort versions (and even reltree and rellbrp) show the different behavior of the tree protocol and LBRP, in fact as Table 6 depicts, the tree protocol forwards almost everything to node 1 (with a small amount which is forwarded to node 3 because of a temporary choice of 3 as parent), while LBRP divides the traffic between 1,2 and 3.

MSG FROM	**TEST NAME**				
	betree	belbrp	reltree	rellbrp	cpr
5	21.92	16.41	9.25	18.20	16.86
6	47.17	42.91	36.38	42.91	50.08
7	46.85	40.10	35.32	37.98	45.52

Table 8: Messages arrived to sink from 5,6,7

From Table 8 a couple of thing can be observed. The performances of the besteffort version are higher than the standard reliable ones (reltree,rellbrp), and are almost the same of CPR. This can be explained by the good connectivity, so that the probability that a message is sent correctly is high, which implies that the besteffort approach can land a good number of deliveries to the sink, while whenever a fail happen in the reliable versions (in the send itself or in the ack delivery), it turns out in an overhead which is a waste in such scenario. Furthermore, the fact that the less messages from the head node (5) are received than the ones from its descendants, can be due to the buffer problem, which was brefly took into consideration in Section 2.5.2 at page 9.

3.4 Random topologies Results

The LBRP seems to work as expected when the topology is randomly generated and the number of nodes larger than in the previous scenarios. The 25 and 49 nodes generated topologies gives the flavour of how the protocol works. Take for instance the forwardings of the node 22 of the test made over the 25 nodes scenario (**NOTE**: the results are cut to show only some of the meaningfull data, the complete results can be seen on the file *forwarded_22.csv*):

TEST	**FORWARDED MESSAGES TO**										
	0	1	2	4	5	6	7	8	9	10	11
betree	-	35.62	27.31	20.67	-	4.37	-	-	-	-	-
belbrp	-	4.06	5.28	6.46	2.42	7.32	1.76	3.54	3.85	8.38	5.56
reltree	-	16.23	10.33	12.59	-	3.65	-	-	-	-	-
rellbrp	-	2.74	2.91	3.87	1.12	4.86	2.04	1.53	2.02	4.81	3.39
cpr	-	5.02	2.56	2.47	∼1	9.67	4.35	3.4	2.22	5.08	4.36

Table 9: Messages forwarded by node 22

TEST	**FORWARDED MESSAGES TO**									
	12	13	15	16	17	18	20	21	23	TOT
betree	-	-	-	-	-	-	-	-	-	89.01
belbrp	1.42	-	3.16	7.24	7.21	7.28	6.86	7.25	-	89.31
reltree	-	-	-	-	-	-	-	-	-	44.15
rellbrp	1.64	-	1.53	4.04	4.17	4.31	5.11	4.28	-	54.72
cpr	4.64	2.29	4.33	11.03	9.10	7.53	4.48	5.44	3.11	97.14

Table 10: Messages forwarded by node 22

The links of the generated topology are not strong, which lead to the loss of routing packets which implies that the routing protocols sometimes acts as non-optimal. Nevertheless, the tests of the tree protocols choose as main parent

the node 1, followed by 2,4 and 6, while the rest of nodes does not take any forwarding from 22. From the tree results, we can infere that 1,2,4,6 have almost the *same cost* to reach the sink, but the beacons reach the node 22 in the inverse order, so that first 6 is chosen, then 4, then 2 and finally 1 as the *smallest id*. Nevertheless when using LBRP 1,2,4 are not chosen so often, that is because the number of parent is fixed (10) until they are removed from the structure (because they got an increased cost or a better way is found), their slots are used.

```
DEBUG (22): NEW SEQNO 21  COST  2{0}
DEBUG (22): ADD PARENT  5 SAME  2{0}
DEBUG (22): ADD PARENT  4 SAME  2{0}
DEBUG (22): ADD PARENT  18  SAME  2{0}
DEBUG (22): ADD PARENT  10  SAME  2{0}
DEBUG (22): ADD PARENT  2 SAME  2{0}
DEBUG (22): ADD PARENT  11  SAME  2{0}
DEBUG (22): ADD PARENT  17  SAME  2{0}
DEBUG (22): ADD PARENT  9 SAME  2{0}
DEBUG (22): ADD PARENT  16  SAME  2{0}
DEBUG (22): NEW SEQNO 16  COST  2{9}
DEBUG (22): NEW SEQNO 1 COST  2{0}
DEBUG (22): REMOVE  PARENT  2 COST  3>2
DEBUG (22): REMOVE  PARENT  18  COST  3>2
DEBUG (22): REMOVE  PARENT  17  COST  3>2
DEBUG (22): REMOVE  PARENT  11  COST  3>2
DEBUG (22): REMOVE  PARENT  10  COST  3>2
DEBUG (22): REMOVE  PARENT  21  COST  4>2
```

Listing 3: Routing debug messages of node 22 with rellbrp

In the Listing 3 can be seen the routing debug messages of the node 22 on a rellbrp simulation loop. After the first beacon the parent structure contains 2,4,5,9,10,11,16,17,18,21, then the maximum number of parents its reached. The second beacon comes from 16, which is already a parent of 22, so no problems come from that and for that application cycle there are no change in the structure. The third and last beacon is forwarded by 1, which is not a parent and because it comes with a cost equal to the current and the structure is full, the last parent (16 in this case) is substituted by 2. During this cycle, the network had some problem which is highlighted by the fact that 6 of the previous parents are removed by the structure due to their increased cost to reach the sink. The CPR version[7] is even more interesting and give some information on the network, infact a lot of NO_PARENT_MESSAGES are sent and the network are not able to reach a stable routing. So in general, it can be said that CPR introduces some overhead in this scenario, which is due to the fact that the links are not good and the delivery of the beacons can be seen as a matter of "luck". In cases like these (and in real world development), would be better to have a link quality estimator, which can help in the choice of parents by adding a knowledge of the "goodness" of the link along to their cost.

[7] if interested the log is on *results/25/routing_22*

TEST NAME	MIN	MAX	AVG	VAR	ST.DEV
betree	3	34	15.87	40.27	6.35
belbrp	6	23	13.54	15.53	3.94
reltree	0	73	10.24	111.24	10.55
rellbrp	0	30	7.90	31.17	5.58
cpr	2	29	13.93	42.29	6.50

Table 11: Messages arrived to sink from 22

Anyway from the Table 11 it can be seen that even if some overhead was introduced by the CPR algorithm, the performance of the protocol were near to the ones of the best-effort tree and by looking at the results of other nodes, in the majority of the cases the delivered messages are even more than the ones done in the other versions.

4 Conclusions

In general the performances of LBRP were good and the algorithm performed as expected. Sometimes the overhead, introduced by the need of using a specific structure and the routines needed for the load balancing, were affecting the numbers of forwarding w.r.t. the tree protocol, but the differences was almost negligible. Even if some tweak is still needed, the CPR was working well, but when I will have some time I will implement the *rebuild messages*[8] feature to see its influence on the routing process. However, as shown for the bigger and random topologies, the quality of the channel and link have a huge impact on the performances of the algorithms e on their choices.I think that in order to obtain better results, would be necessary to integrate some kind of lin quality estimator (RSSI,LQI,ETX), so that a metric on the communication environment can help to choose whether a neighbor can be good parent or not. Without that, the routing protocols might be subjected to the delivery of beacons from not-so-reliable links, which can compromise the operating principle of the network. Moreover, the introduction of a data aggregation scheme could be an interesting choice to enhance the performance on the data layer, which might prevent the buffer congestion problem. For instance, the whole current buffer can be forwarded, so that at each trigger of the send timer, the buffer is emptied. However it should be better analysed, because in some scenario it could lead to "bottle-neck" problem on some nodes.

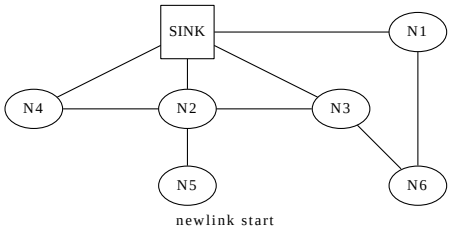
[8] Sec 2.5.3 at page 10

5 Appendixes

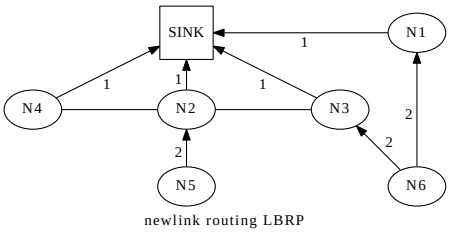
5.1 Appendix A: Scenarios in-depth look

5.1.1 Newlink

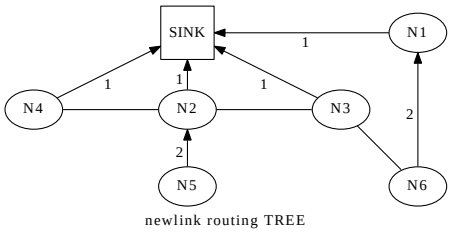
This is the configuration at the start



newlink start



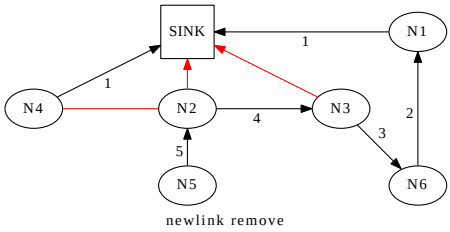
newlink routing LBRP



newlink routing TREE

After the first beacon, the difference between the Tree protocol and the LBRP is that in the first the node 6 will have just 1 as parent, while in the latter 3 will be chosen as another parent, because both 1 and 3 reach the sink through 1 hop

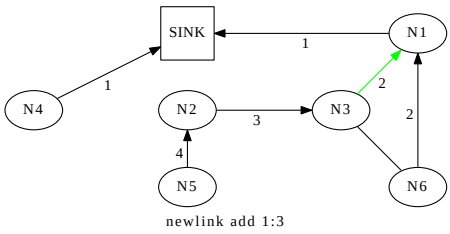
At the 30% of the simulation the red links will be removed, so the network will reconstruct in the way depicted in the figure on the right, so that a long chain will be used to deliver messages

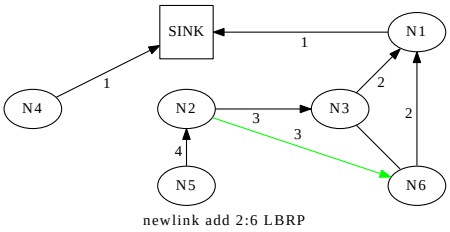


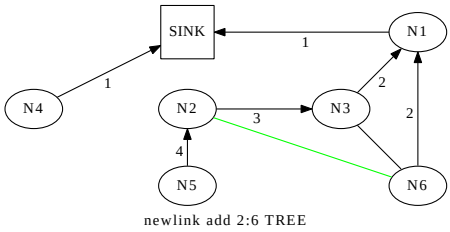
newlink remove

continues on the next page ...

At 55% of the simulation, a link between 3 and 1 is added, so both tree and LBRP will choose 1 as the new parent for 3 because of the minor cost than 6.





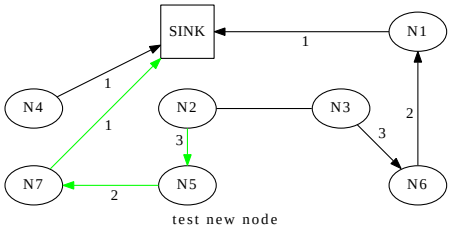


At 60% of the simulation, another link is added, this time between 2 and 6. That link will be chosen by LBRP based versions, to add 6 as parent of 2 (along with 3), while it will be almost ignored by Tree versions

5.1.2 Test

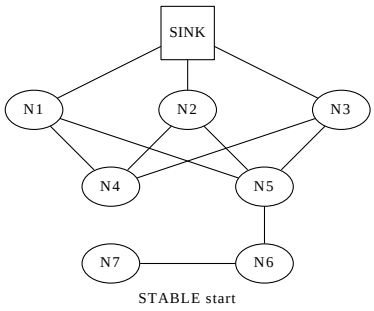
The first three steps are equal to the Newlink ones, that can be seen on the previous page.

At 75% of the simulation a new node (7) is added with links to the sink and to 5. This will force the network to choose a new configuration, because the introduction of 7, bring a new path with a reduced cost to 5 and 2

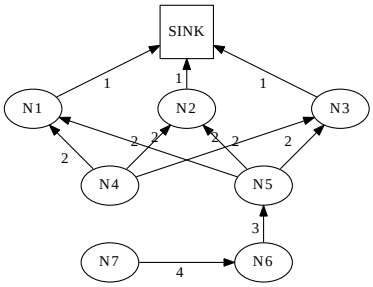


5.1.3 Stable

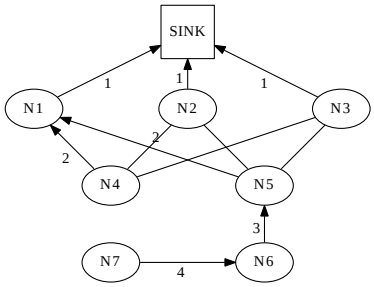
This topology is characterised by a stable configuration and links with a good gain.



STABLE start



stable routing LBRP



stable routing TREE

After the first beacon, the difference between the Tree protocol and the LBRP is that the "second layer" nodes will route their traffic to 1 when using a Tree version, while by using LBRP the traffic will be split between 1,2,3