Test cases for the basic features of the CTP protocol

These basic test cases cover most the main features of the CTP protocol. For a guide on how to use the CTP algorithm in an application and set up a test framework, see the document testing_guide.txt.
The following cases can be run on both the iSense nodes and simulated in Shawn.
For running these test cases in Shawn, we need to make the following timing amendments:

In the sendTask() function in the FE module, the reTx timer is set after a message has been sent the first time, and resent if not acknowledged through the following lines of code:

```
startRetxmitTimer(SENDDONE_OK_WINDOW, SENDDONE_OK_OFFSET);
startRetxmitTimer(SENDDONE_NOACK_WINDOW, SENDDONE_NOACK_OFFSET);
```

These lines must each be replaced with the following code:

```
if (!reTxTimerIsRunning) {
        setTimer((void*) RETXTIMER, 2200)
}
```

This compensates for Shawn's one second iterations, as an ack can be received only 2 iterations after the original message has been sent.

1. Loop detection and choosing an alternate route

Initial state:

```
#define CTP_DEBUGGING

#define DEBUG_ETX

#define NODES_NR                4

#define LINKS_NR                5
#define LINKS               {{0,1,1},{1,2,1},{1,3,100},{2,3,1},{3,0,100}}

#define SENDER_NODES_NR         1
#define SENDER_NODES                    {3}

#define ROOT_NODES_NR           1
#define ROOT_NODES                      {0}

#define DEBUG_NODES                     {0,1,2,3}
#define DEBUG_NODES_NR          4
```

Methodology and outcomes:
After the network stabilizes and the routing paths are calculated, the messages sent from the main app on node 3 are routed through 3->2->1->0.
Then we remove or change the link quality between node 0 and 1 to something very big, like 1000. After this change occurs, the messages are forwarded in the loop 3->2->1->3 for a while, until the routes are updated, and then it starts forwarding directly to 0. While the messages are in the loop, nodes 1 and 2 detect a possible loop and force the RE to trigger a route update.
Result: Pass

2. Better route present

Initial state:

```
#define CTP_DEBUGGING

#define DEBUG_ETX

#define NODES_NR                4

#define LINKS_NR                5
#define LINKS                   {{0,1,1},{1,2,60},{1,3,50},{2,3,20},{3,0,100}}

#define SENDER_NODES_NR         1
#define SENDER_NODES                    {3}

#define ROOT_NODES_NR           1
#define ROOT_NODES                      {0}

#define DEBUG_NODES                     {0,1,2,3}
#define DEBUG_NODES_NR          4
```

Methodology and outcomes:

After the network stabilizes and the routing paths are calculated, the messages sent from them main app on node 3 are routed through 3->1->0.

Then we change the link quality between node 1 and 2 to 10. After this change occurs, the messages are still routed on the path 3->1->0 for a while, until the routes are updated, and then they start being routed through 3->2->1->0.

Result: **Pass**

3. No route available – route found

Initial state:

```
#define CTP_DEBUGGING

//#define DEBUG_ETX

#define NODES_NR                4

#define LINKS_NR                4
#define LINKS                   {{0,1,1},{1,3,50},{2,3,20},{100,100,USHRT_MAX}}

#define SENDER_NODES_NR         1
#define SENDER_NODES                    {3}

#define ROOT_NODES_NR           1
#define ROOT_NODES                      {0}

#define DEBUG_NODES                     {0,1,2,3}
#define DEBUG_NODES_NR          4
```

The etx values in the LINKS array are not taken into account since DEBUG_ETX is not defined. The last element in the LINKS array is a dummy one. We will change it to {0,2, USHRT_MAX} later to enable the link between 0 and 2.

In Shawn, in order to speed up the things, the enum member MAX_RETRIES in the FE representing the number of unacked sends before dropping a packet must be set to 10.

Methodology and outcomes:
After the network stabilizes and the routing paths are calculated, the messages sent from the main app on node 3 take the only available route: 3->1->0.
After some time (20 iterations in Shawn), we remove the link 1 – 3 by changing one of its nodes to a value greater than NODES_NR (eg. 100).
After this, since it has no other alternate route, node 3 will continue to forward the messages to node 1. After a while, the unacked messages accumulate, and the etx value for the link between 3 and 1 increases.
During this time, packets unacked for a certain period will be dropped, and it may be reached the point where the send queue gets full, and incoming messages are dropped instantly.
After some more time (40 iterations in Shawn), we enable the link 0-2. Node 3 still tries to forward messages to node 1 for a while, until the link 0-2 becomes mature. Then a new route is computed, and the messages start travelling on the route 3->2->0.
Result: **Pass**

4. Congestion detection – load balancing

Initial state:

```
#define CTP_DEBUGGING

#define DEBUG_ETX

#define NODES_NR                4

#define LINKS_NR                4
#define LINKS                   {{0,1,20},{0,2,19},{3,1,20},{3,2,40}}


#define SENDER_NODES_NR         1
#define SENDER_NODES                    {3}

#define ROOT_NODES_NR           1
#define ROOT_NODES                      {0}

#define DEBUG_NODES                     {0,1,2,3}
#define DEBUG_NODES_NR          4
```

In order to force a congestion situation on node 1, we make the following alterations:
The message sending period in the main app is 500ms.
For node 1, the timer is set in the function post_sendTask() of the FE with a timeout of 1000 instead of 0 as normal.
The limit variable congestionThreshold is initialized in init_variables() of the FE with command_SendQueue_maxSize() >> 2, to consider an earlier congestion.

Methodology and outcomes:
After the network stabilizes and the routing paths are calculated, the messages sent from the main ap on node 3 are routed through 3->1->0.

After a while, node 1 reports that it is congested, sets the congested flag and node 3 detects that its parent is congested. It then triggers a route update, and detects node 2 as a feasible parent. Messages are then routed through 3->2->0, even though, overall, it has a higher ETX than the initial route (3->1->0).
After some time, node 1 is not congested anymore, and adjusts the beacon flags accordingly. Eventually node 3 sets 1 as parent and messages begin being passed on the route 3->1->0 again. Eventually node 1 becomes congested again, and the cycle repeats, ensuring well balanced traffic on the network.
Result: **Pass**