Wireless Sensor Networks project: Source Routing for Download Data Traffic

July 17, 2017

Ardino Pierfrancesco 189159

The aim of this project is to implement a multi-hop source routing protocol. The routing protocol will support both many-to-one and one-to-many traffic patterns. In the first one all nodes will send data packets up to the sink, while in the second one the root has the possibility to send unicast data packets to other nodes in the collection tree. The project consists of these files:

- `MyCollection.h`: defines the structs needed for the project;
- `RoutingC.nc`: defines the Routing interface that provides the creation of the collection tree and wires the modules used and provided by RoutingP;
- `RoutingP.nc`: contains the implementation of the Routing interface;
- `OneToManyC.nc`: defines the OneToMany traffic pattern interface that provides the sending of unicast packet from the sink to a specific node in the network, moreover it wires the modules used and provided by OneToManyP;
- `OneToManyP.nc`: contains the implementation of the OneToMany interface;
- `ManyToOneC.nc`: defines the ManyToOne traffic pattern interface that provides the sending of packets from all the nodes in the networks to the sink, moreover it wires the modules used and provided by ManyToOneP;
- `ManyToOneP.nc`: contains the implementation of the ManyToOne interface;
- `AppP.nc, Test`: contain the implementation of the application that uses the Routing protocol with the two traffic patter interfaces.

1 Implementation

1.1 Initialization

At the beginning all nodes boot with the node with ID 1 set as the Root. In addition to that they all start *StartTimer*. Once the timer fired, the Root will start the **Build Tree** procedure and start the *PeriodicTimer* timer, which is started also by the other nodes, that periodically start a *OneToMany* or *ManyToOne* communication based on the type of node.

1.2 BuildTree

The sink is responsible for the building of the routing tree. When it receives the `buildTree` command from the application, it sets the `i_am_sink` variable to **TRUE**, starts the *RefreshTimer* timer that when fired will restart the **Build Tree** procedure and posts the `send_beacon` task that will broadcast a beacon. The payload of the beacon will contain the sequence number of the beacon and the current hops to the sink, which will be used in combination with the **RSSI**

as metric for choosing the parent of a node.
When a node has to send a beacon, it first check the value of the `sending_beacon` variable, if this variable is **TRUE** it means that the node is already sending another beacon and so it can not send the current one, while if the variable is **FALSE** it will fill the beacon payload and send the beacon, if there are some problems during the sending of a beacon the message will be rescheduled using the *NotificationTimer* timer.
When a node receives a beacon it first checks if it is the root, if so the beacon is discarded. Otherwise it will compute the **RSSI** between him and the sender of the beacon. If the **RSSI** is lower than a threshold the beacon is discarded, otherwise the node will compare the sequence number of the beacon, if the sequence number is older than the previous beacon than the packet is dropped otherwise if it is newer it means that the root started to rebuild the tree and so the parent of the node is set to the sender of the beacon. Lastly, if the sequence number is the same the node will check the hops to the root. If the number of hops to the root contained in the beacon is lower than the one stored by the node than the parent is updated to the sender of the beacon, otherwise if the number of hop is equal, the node will compare the **RSSI** and then update the parent. If the parent is updated the node will broadcast a new beacon with the newer distance from the root.

1.3 Routing Table creation

A routing Table is maintained by the root, each entry of the routing table contains information about a node and its parent. When a node updates its parent, it sends a message to the root containing the information about the parent. If the node is already sending a **InfoBeacon** to the root, it will put the message in a temporary variable and reschedule the sending using the *InfoTimerRescheduling* timer that when fired, will fetch the information of the packet from the temporary variable.
When the root receives the message it will update its Routing table, adding a new entry or updating a existing entry.

1.4 Many-To-One traffic pattern

As said before, in Many-To-One traffic pattern the nodes in the network send messages to the sink. The payload of these messages contains a sequence number and the hops between the node and sink. The sending of a message is controlled by the *JitterTime* timer that when fired calls the **send** procedure of the **ManyToOne** interface passing as parameter a `MyData` variable containing the sequence number of the message.
The **send** procedure, when called, fills the payload of the message setting up the source of the message, initializing the number of hops and including the struct received as input. If the node is already sending a message, defined by the `sending_data` variable that has the same behaviour of the `sending_beacon` in the **RoutingP** interface, the message is copied into a temporary variable and rescheduled using the *RetryForwardingTimer* timer. Otherwise if the `sending_data` variable is **FALSE** the **send_data** procedure will be called passing as parameter the payload.
The **send_data** procedure will set the number of max retries of the message, find the parent of the node calling the **getParent** routine of the **RoutingP** interface and try to send the message. If the send is successful the `sending_data` variable will be set to **TRUE** otherwise the message will be rescheduled. When the **sendDone** is triggered the `sending_data` variable is set to **FALSE**.
When a node receive a message it checks if it is the root, if so the node signal the receiving of a message that will be caught by the **AppP** module. If it is not the root the node will forward the message to its parent updating the hops number in the payload.

1.5 One-To-Many traffic pattern

In One-To-Many traffic pattern the root has the possibility to send unicast data packets to other network nodes down the collection tree. The root will use its `RoutingTable` to build the path from the root to the node chosen as destination.
First of all the root calls the **getRandomNode** routine of the **RoutingP** that returns a random node from the routing table. Then it calls the **send** procedure of the **OneToMany** interface passing as parameter a `MyData` variable containing the sequence number of the message and a variable that contains the destination of the message.

The path to the destination is maintained by an array where each element represent the next hop to the destination. This array is built by the **getDestinationRoute** routine of the **RoutingP** interface. This routine iterates the routing table several time in order to create a path to the destination node. It starts by assign to `currDest` the destination node and put it in a reverse path array. Then the routine search in the routing table an entry which has as *childAddress* the `currDest` variable, if it finds an entry it change the `currDest` to the parent of the node and insert it into the reverse path array, this procedure is repeated until the `currDest` variable is equal to the root or the path length is greater than the max path length defined into the **Mycollection.h** file, checking the length of the array will avoid the creation of loop because as soon as the path length is greater than *MAX_ROUTE_LENGTH* (which is 30 for this project) the procedure is stopped and a *NULL* pointer is returned to the **OneToManyP** interface. If the creation of the path is successful the path is returned in the correct order. Then the path and the destination node is inserted into the payload of the message and it is forwarded to the next hop in the path if the node is not already sending another message, otherwise the message is rescheduled. When a node receive a message it first check the destination of the message, if the message is for him it will signal the Application interface otherwise it will delete itself from the path and forwards the message to the next node in the path.

2 Testing

The implementation has been tested using the Cooja simulator with 15 nodes in the network. The first test was made disabling the LPL from the Makefile and sending 200 message both for One-To-Many and Many-To-One communication. The probability of receiving a message has been set to 20% while the probability of sending a message to 100%. The second test has been made activating the LPL with an interval of 64ms and sending only 20 messages due to crashes in the Cooja simulators.

2.1 Without LPL

The results of the first test show an average PDR of 0.77 for the Many-To-One communication with a min PDR of 0.57. Each node in the network is able to send messages to the root. As expected the nearest nodes to the root have a really high PDR while the farthest nodes have a PDR of about ∼0.60, which is normal giving that the probability of a message being lost increase with the increasing of hops between a node and the root. I've also noticed an increasing in the PDR wrt to the original Many-To-One protocol rescheduling message when they can not be sent instead of simply drop that messages.
The average PDR of the One-To-Many is 0.79 with a minimum of 0.50. Packets dropped due to cycles in the path are dropped and counted as not received. The probability of create a loop in the routing table is not so high, in fact during the simulation only 6 messages are dropped due to cycles in the routing table.

2.2 With LPL

As expected the results of the second test are worse that the one of the first test. For Many-To-One communication the average PDR has dropped to 0.60 while minimum is 0.25. While the PDR of nodes near to the root is quite high and comparable the ones of the first simulation, the PDR of nodes far from the root is very low, in fact there are 5 nodes with a PDR lower than 0.40.
The PDR of the One-To-Many communication follows the same pattern of the Many-To-One communication, both average and min PDR are lower than the one in the first simulation. Again there are only few packets that are dropped due to the finding of cycles in the routing table.