ADAPTABLE NETWORK INFRASTRUCTURE

Dimitrios Amaxilatis
Research Academic Computer Technology Institute
Patras, Greece
amaxilatis@cti.gr

Shlomi Dolev
Ben-Gurion University of the Negev
Beer-Sheva, Israel
dolev@cs.bgu.ac.il

Christos Koninis
Research Academic Computer Technology Institute
Patras, Greece
koninis@cti.gr

1. Introduction

In this chapter we present a set of self-stabilizing software components, that can be composed into a whole self-organizing and adaptive layer that runs on a network of tiny artifacts. It is the basic layer in which the system "comes together", organizes continuously, and is always ready to react. Here the basic components that work together are:

1 Adaptive neighbors discovery: continuous detection of nearby nodes, passing-by nodes, etc.

2 Adaptive topology control: establishment of Leaders in neighborhoods, and dynamic creation of clusters.

3 Adaptive cluster radio: formation of a layer of abstraction for leader to leader communication.

Each component periodically (or at external request) should re-organize with respect to neighbors, local leaders and clusters. Note that this Layer reacts to "internal" changes , due e.g., to mobility of nodes, local failures etc.

2. Neighborhood Discovery module

One of the most important issues faced by algorithms running on WSNs is the erratic behavior of the of the communication channel. It is quite common for sudden changes in the quality of the communication medium or nodes with poor communication channel (e.i. due to local interference, or node positioning), to greatly affect the stability and performance of an entire network running an otherwise stable software module. The ND module offers local connectivity awareness and based on a number of indicators is able to characterize the stability of the neighboring nodes so that the higher modules can adjust their operation to frequent and/or significant topology changes.

2.1 Neighborhood Discovery Methodology

The main operation is based on a broadcasting mechanism that periodically sends beacon messages. For each received beacon message the ND module examines a number of indicator values which define the quality of the communication channel with it's neighboring node. Then, based on these indicators, decides if a node should be considered as a "stable" neighboring node. More specifically, if the ND receives at least 2 consecutive beacon

messages that their indicator values are above certain thresholds then the node is marked as "stable". The stable neighboring nodes are attached to each broadcast packet and when a node receives a beacon, it checks whether its address is listed in the received message. In this way , the module is able to recognize whether the communication with the neighboring nodes is bi-directional or not. If the ND module stops receiving beacons from a neighbor, or if the beacons received do not match the defined indicators, it will remove the node from the list of neighbors after a predefined amount of time.

The indicators used for examining the quality of the communication channels are (a) the LQI of the received beacons, (b) the RSSI of the received beacons, (c) the link associativity (the time, in beacons, that nodes are associated, i.e., they retain a connection), (d) the node stability (the average link associativity for all neighboring nodes) and (e) the reverse node stability (defined as a result of experimentation, indicating the average numbers of consecutive beacons that successfully delivered on a destination).

The goal is to filter out the neighbors that have poor communication channel quality. The nodes will consider the broadcaster as a neighboring node only if it receives a number of consecutive beacon messages with RSSI/LQI above a certain threshold. In order to avoid occasional short channel degradation to negatively impact the above strategy, we allow a node to miss a number of beacons within a given period of time before removing it (called the *timeout period*). However we set a second RSSI/LQI threshold; message beacons with RSSI/LQI below this threshold are dropped (and therefore count towards the number of beacons that are missed).

The ND module can adopt its mode of operation based on the requirements of the higher-level modules. Depending on the requests it fine tunes (i) the periodicity of examining the condition of the nearby devices, (ii) the frequency of generating notifications for the changes in the nearby topology, (iii) the indicators that will be used to characterize the quality of the communication channel, and (iv) the contents of the beacon message.

2.2 Piggybacking Mechanism

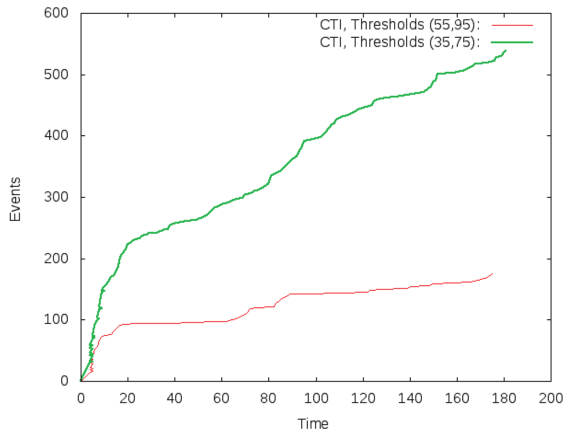
It is common among many existing algorithms to periodically exchange data as new events are generated. Normally this is done by broadcasting messages containing the updated information. This can often lead to the generation of excess messages waste of precious memory space when more that one algorithm is running on the same device and needlessly duplicate the previous mechanism. The ND provides a piggybacking technique that can be significantly reduce the rate of broadcasting messages, by allowing higher-layer modules to take advantage of the broadcasting mechanism that is already in use by the module. The piggybacking technique exposes an interface that allows other modules to attach their data on the beacon messages of the ND module, and get notified by a callback mechanism then new data are received. The ND will gather all the data from the different modules and will add them to the payload of the next beacon message, and upon receiving the beacon it will unpack the data and deliver them to the relevant modules. For example, the Clustering module is using this technique so that the nodes also broadcast their role (cluster head, cluster member).

2.3 Evaluation

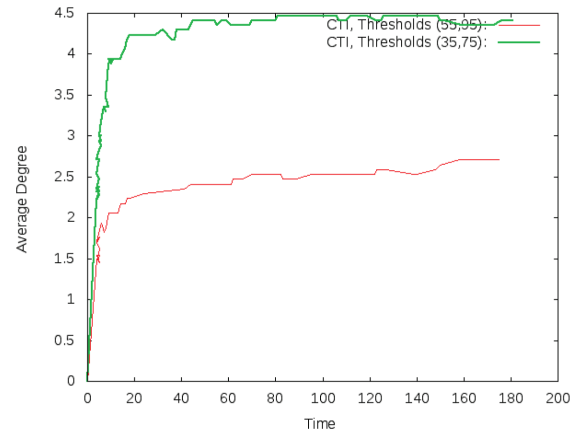
We have conducted a series of experiments in order to determine the effect that different parameters have in the ND mechanism. We measure the *average size of the resulting neighborhoods* (i.e., we calculate the size of the neighborhood of each node, and average over all nodes) and the *total events produced* (i.e., the total number of events produced by the ND module of all nodes). Remark that in the figures we use the term *node degree* to refer to the size the node's neighborhood of the current topology. We start by measuring the effect of the LQI thresholds in the performance of the ND module, for that we fix the beacon interval to $1000ms$ and the period for removing a node (i.e., if no beacon is received) to $5000ms$.

We assess the performance of the ND module based on two LQI threshold pairs: $(35, 75)$ and $(55, 95)$. The resulting neighborhood sizes are depicted in Fig. 1(b) and the total number of events generated is shown in Fig. 1(a). As expected the stricter the LQI thresholds, the smaller and more stable the neighborhood sizes will be. While when relaxing the LQI thresholds, the neighborhood sizes increase but also the system become prone to channel quality fluctuations.

We conclude that the LQI threshold pair $(55, 95)$ is more suitable for the experimental testbeds used. For this range the ND module generates stable neighborhoods within a short period of time and the resulting topology is dense enough to allow the other modules to operate properly.

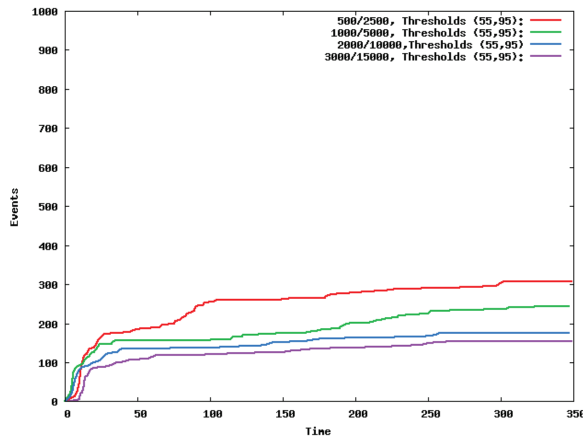


(a) Total number of events generated

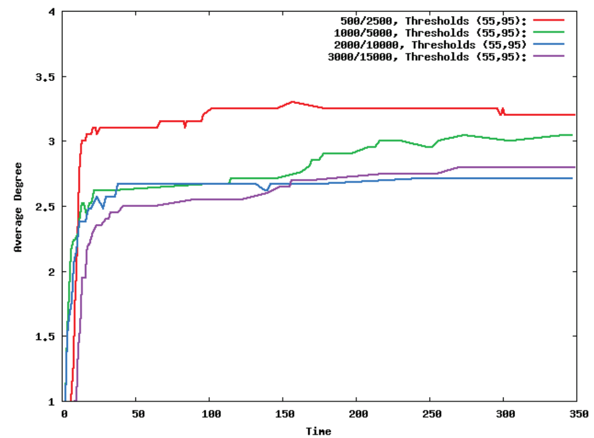


(b) Average degree

Figure 1: Events generated and average degree reported by ND for different LQI threshold values



(a) Total number of events generated



(b) Average degree

Figure 2: Events generated and average degree reported by ND for different beacon interval periods

In the previous set of experiments we measured the impact of the LQI thresholds and kept fixed the values for the beacon interval period (i.e., the period between consecutive transmissions of message beacons) and the neighbor timeout period. We continue with a second set of experiments to determine the impact of beacon interval period and the neighbor timeout period in the ND module. One would expect there exists a linear correlation between the time required for the module to stabilize (i.e., correctly detect the neighboring nodes) and the beacon interval. Reducing the beacon interval period (i.e., increasing the rate of transmission) should lead to a quicker response to changes in the topology (i.e., shorter delays in detecting changes to the neighboring nodes).

We execute the ND module for four beacon/timeout period pairs: $500ms/2500ms$, $1000ms/5000ms$, $2000ms/10000ms$, $3000ms/15000ms$. Notice that all pairs maintain the same ratio $1 : 5$ between the beacon interval period and the timeout period. Recall that in the experiments reported in this Section, all nodes nodes remain static for the full duration of the experiment. Therefore any changes in the topology are the result of the fluctuation of the wireless transmission medium due to people moving around in the offices and their actions in their working environment (e.g., doors being opened, mobile phones operated etc.). The resulting *average size of the neighborhoods* for the four beacon/timeout period pairs are shown in Fig. 2(b) and the results for the *total*

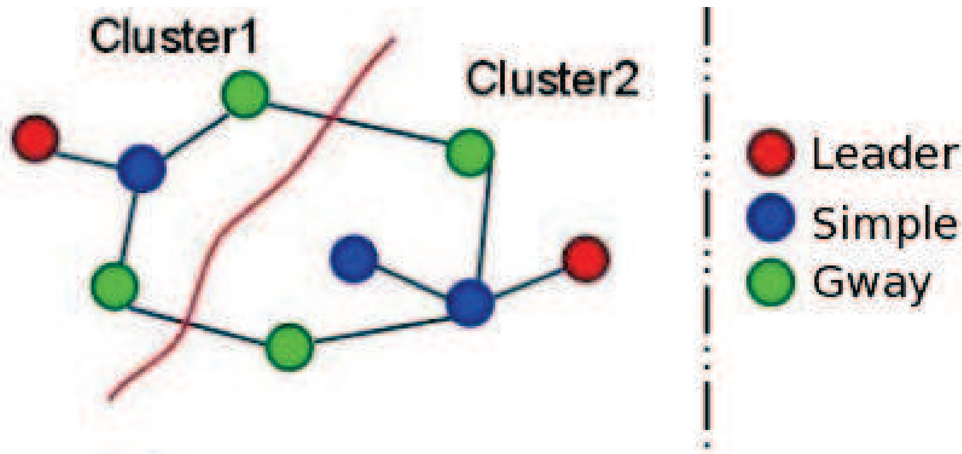


Figure 3: Example of two formed Cluster

events produced are depicted in Fig. 2(a).

Surprisingly, the results show that our intuition was wrong, the experiments reveal that for small beaconing values (less than $1000ms$) the time the ND module needs to stabilize is longer. This leads to a larger number of events generated. This is caused by the excess traffic generated due to the short beacon interval, which itself creates interference that leads to losses of message beacons. Thus many neighbors are falsely removed and then re-added to the neighborhoods. Compared to the with the $1000ms/5000ms$ the $500ms/2500ms$ Beacon Interval/Neighbor Timeout needs almost double the time to stabilize and generates 30% more events.

3. Clustering module

The CL module is responsible for organizing the network into multiple coherent groups. Each group has its own hierarchy with an elected leader and nodes of different ranks. The implemented algorithm designates 3 roles to cluster nodes: `Leader, Gateway and Simple` node. `Leaders` are responsible for the formation of the cluster. They always advertise their cluster and try to persuade more nodes to join them. `Gateways` are nodes on the outskirts of the cluster, responsible for communicating with other nearby clusters. Finally, `Simple` nodes are just members of the cluster. Fig 3 is a simple sketch with two clusters in a 9 node network, showing how roles are distributed to nodes.

3.1 The Algorithm used

To achieve this structure inside the network CL module executes two phases. In the **first phase**, nodes are elected as leaders if they have the minimum ID within their $k - hop$ neighborhood. In the **second phase**, the formation of the clusters is performed and every node joins the nearest leader. As the status of the wireless network is not stable, the module enters a **third phase**, which is continuously executed and maintains the clusters by making all the necessary adjustments to make sure that at any given time every node can communicate with its leader.

The **first phase** is actually an exchange of ids where all nodes flood the network with the minimum id they know of. They start by broadcasting their own id and continue to broadcast the minimum value amongst the ids they receive and their own. This phase actually consists of k id exchanges, where k is the desired maximum distance between a leader and all cluster nodes. After k messages are exchanged, all nodes check their final minimum known ID. If it is their own id they become leader

The **second phase** starts right after leaders are elected. All leaders broadcast advertisement messages. Each non-leader node that receives an advertisement message checks its status and joins the cluster if the leader is closer to him than its current leader (if the node is unclustered then the distance to current leader is ∞), or the leader with the minimum ID if both leaders (current and advertised) are at the same distance. A node that joined a new cluster is responsible for propagating the advertisement message up to k hops away. The advertisement propagation is key to the formation of multi-hop clusters.

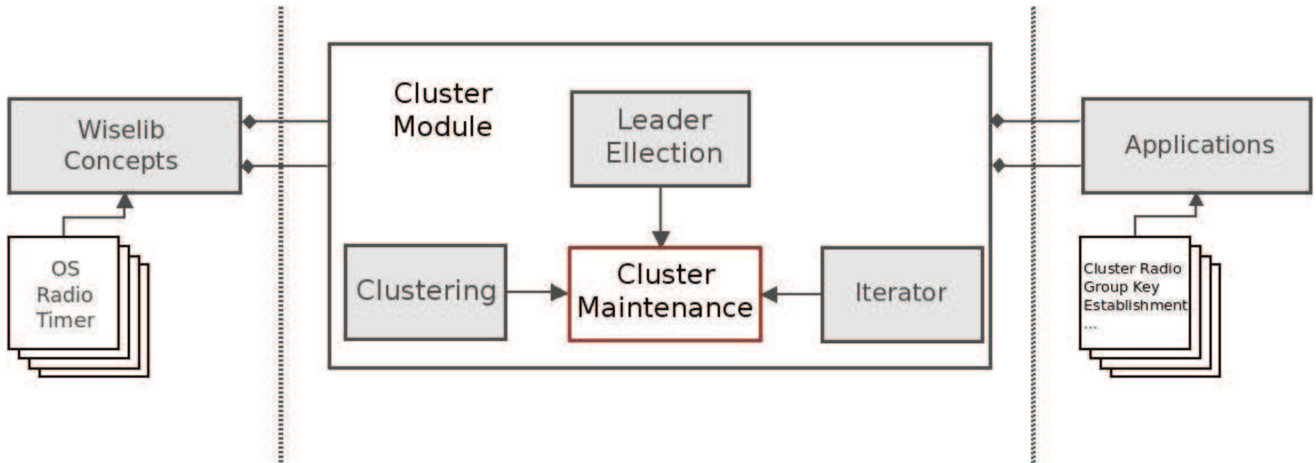


Figure 4: The component based implementation of Clustering Module

3.2 Implementation

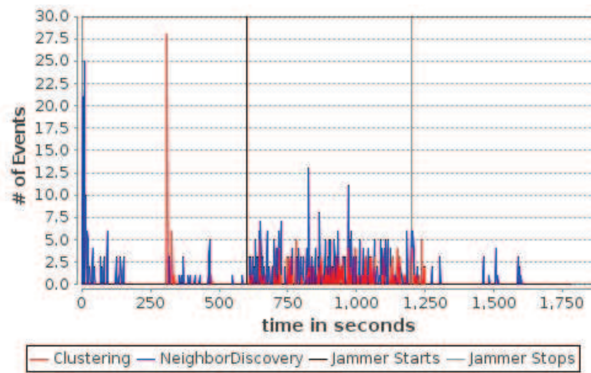
The implementation of the module follows a component based approach. The concept behind the component implementation is that the phases described above are common in many clustering algorithms. For example the LCA algorithm forms clusters in almost the same way, while MaxMinD elects leaders using an id exchange phase. By using the component implementation, components that are common can be reused, reducing the effort needed to implement new algorithms. After studying many surveys about clustering algorithms and the classifications proposed, we decided that Leader Election and Clustering can be implemented as two different components. A third component is responsible for cluster maintenance and interfacing the clustering algorithm with other modules and a forth component called an Iterator is used to store all information needed such as ClusterId, Parent, ClusterNeighbors or Gateway status which should be available for all clustering algorithms. How the components form Clustering, interface with other algorithms and use Wiselib Concepts such as Radio or Timer is also presented in Fig 4.

3.3 Evaluation

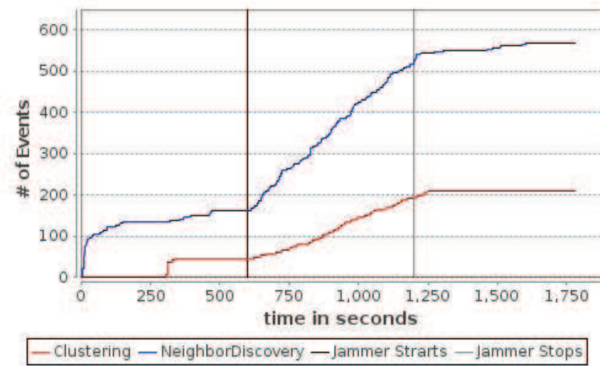
The most illustrative experiment to present how the CL module functions and maintains clusters is the one presented in Fig 5. CL is enabled after the ND module stabilizes and all bidi links are identified. After a period of message exchange and event generation, clustering stabilizes and as long as bidi links, provided by ND, are stable clusters remain the same. When the jammer is enabled, link quality is severely degraded. This causes great instability and ND continuously generates events. As a result CL module becomes unstable too (generating messages, events, and changing the cluster structure). After the jammer is switched off, both ND and CL stabilize again.

4. Cluster Radio Module

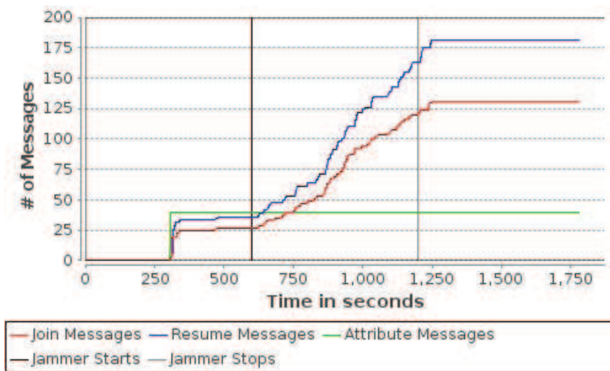
The CLR Module provides a layer of abstraction for leader to leader communication. After the whole network is organized in multiple clusters and leaders become responsible for all the nodes that have joined them, there is no need for complete node to node communication. In that way traffic is reduced and the network's life can be prolonged. Also, after this virtual network between the leaders is formed, we can introduce a second level of organization (using again CL, CLR and ND) in order to create a more complex hierarchy. In order to achieve all these, CLR should provide valid routes between leaders of nearby clusters, creating a virtual network over the physical links with reduced diameter.



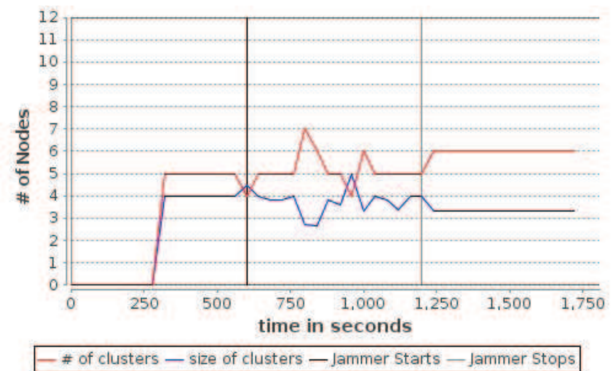
(a) # of Events generated



(b) Total # of Events generated



(c) Messages generated



(d) Cluster Sizes

Figure 5: Experiment Statistics using ND and CL module

4.1 CLR Routing Tables

All nodes in the network have their own `Routing Table` for routing packets of the CLR. A routing entry contains information about the `destination cluster`, the `link` where the packet should be forwarded through and an indicator of how `old` the information on the routing table is. A routing table entry is refreshed every time the ND module send a new beacon. When a beacon is received, the CL module notifies CLR for the route as if it was a new one. Then CLR updates its routing table (setting the age indicator of the link to new) and forwards the notification to its `cluster head`. When a node receives a message for a new CLR route (or an update for an existing) checks its own routing table. If the entry **existed** in exactly the same way in its Routing Table then the entry is refreshed and the message is **propagated** to its parent. If the entry is **different**, because the node uses a different route, the message is **ignored**. In order to detect routes that are no longer valid a **timeout period** is defined and if the link is older than the timeout period it is removed.

4.2 Generating unique routes

Routes are generated by the `Gateway` nodes.`Gateway` nodes play the most important role in the process. A CLR route is formed whenever a node becomes `Gateway`. As messages propagate from gateways to `Leaders`, all nodes that forward the message also **register** the respective route in their routing tables. After all such messages are delivered, `Leaders` know all their nearby Clusters and a valid route to them. Also, all nodes in the path have their routing tables set.

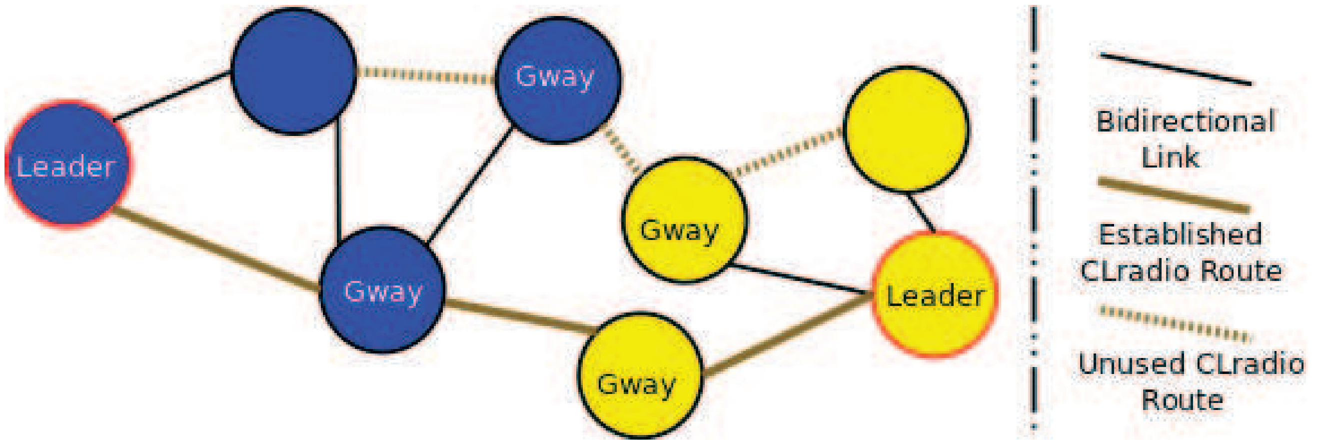


Figure 6: Example of formed Cluster Radio routes

4.3 Sending Messages

When a `Leader` needs to send a message to a selected cluster, the message is sent to the node designated in the Leader's routing table. The node that receives the message forwards it to the next hop, as pointed in its routing table. When the message passes from the `Gateway` node to a node of the destination cluster, it is forwarded using the tree formed from CL (using the value returned from the `parent()` call to clustering). Using this concept routing decisions are made locally and paths can be more adaptive than when Leaders decide the path when originally sending the message.

5. Interfaces

5.1 ND Interface

The Neighborhood Discovery Module follows the general Wiselib interface. It provides an enable and disable function and a {reg/unreg}_event_callback to register for notifications about changes in the neighborhood.

```
template<...> class NeighborhoodDiscovery {
 int enable(void);
 int disable(void);

 init(RadioT, ClockT, TimerT , DebugT);

 set_beacon_period(uint16_t);
 set_timeout_period(uint16_t);

 uint8_t reg_event_callback(uint8_t alg_id, uint8_t events_flag, T *obj_pnt);
 void unreg_event_callback(uint8_t alg_id);

 uint8_t register_payload_space(uint8_t payload_id);
 uint8_t unregister_payload_space(uint8_t payload_id);

 uint8_t set_payload(uint8_t payload_id, uint8_t *data, uint8_t len);
 bool is_neighbor(node_id_t);
 bool is_neighbor_bidi(node_id_t);
 uint8_t nb_size();
}
```

Interfaces with other components:

Radio The module uses the radio interface to access the wireless communication device of the node.

5.2 CL Interface

Clustering Module follows the general Wiselib interface. It provides an enable and disable function and a {reg/unreg}_event_callback to register for notifications about changes in clusters.

```
template<...> class ClusteringAlgorithm {
 int enable(void);
 int disable(void);

 init(RadioT, TimerT , DebugT , RandT);

 set_components(CHD_t,JD_t,IT_t);
 set_maxhops(int);

 node_id_t parent();
 cluster_id_t cluster_id();
 int hops();
 int node_type();
 bool is_gateway();
 bool is_cluster_head();

 int reg_event_callback(T *obj_pnt);
 int unreg_event_callback(int idx);
}
```

To enable Cluster Maintenance and adapt to topology changes CL uses both ND event notifications and ND payload piggybacking. As clusters require bidirectional communication within the cluster, all nodes accept messages only from nodes that ND identifies as bidi. Also when ND reports a lost link with a cluster node, the failure is detected by the CL module and all needed actions are made. Also, the piggybacked payload is in fact an advertisement message, broadcasted periodically to detect new clusters that did not exist previously or clusters of mobile nodes that were formed somewhere else and moved to the area.

5.3 CLR Interface

Cluster Radio actually follows the Radio Concept of Wiselib. Thus implements the same functions as the normal radio. For starters, it `provides enable_radio()` and `disable_radio()` functions to switch the radio on and off. In order to send a message there is a `send` function that is implemented exactly as the send function of Radio. To send a message you need to provide the `destination`, the `size` of the payload to be sent and a `pointer` to the payload. The destination address can either be a Broadcast Address (`0xffff`) or the id of the destination Cluster. In case of a Broadcast Address, CLR sends the message to all known Clusters. Also, each algorithm should register a function, as a receive callback, (same way as in Wiselib physical radio) to be able to receive cluster radio messages.

```
template<...> class ClusterRadio {
 int enable_radio(void);
 int disable_radio(void);
 void send(node_id_t receiver, size_t len, block_data_t *data);
 node_id_t id();
 int reg_recv_callback(T *obj_pnt);
 int unreg_recv_callback(int idx);
}
```

In order to build the routing tables, CLR needs to receive notifications from the CL. So CLR registers itself to receive notifications for all the events of Clustering. When a neighbor node changes its cluster, the new cluster is examined. If the new cluster is different than the node's cluster, a new Routing rule is generated and the Leader is notified as described before.