NSDS CONTIKI, NODE-RED, SPARK PROJECT

Contiki

Introduction and problems encountered

In this document we present the supposed functionality of the program that we created. The program does not work and it is partially written in pseudo-code. The reason for this is that we encountered a large amount of problems while trying to implement it, we had a huge amount of type-errors whenever we wanted to use a contiki library, especially the LIST library and the ip_helpers ,for example the distinction between *uip_ipaddr_t*, *uip_ip6addr_t* and *uip_ip4addr_t*. We looked at the header files where the structures are defined (https://docs.contiki-ng.org/en/master/_api/uip_8h_source.html) to get a better understanding, but still would encounter errors whenever we tried to code with them. Moreover some feature that are present in the documentation are straight up missing, like the *uip_hostaddr* field which should contain the address of the host-node; including this variable in the code will not produce any errors, as it is present in the header files but Cooja will refuse to start the simulation for reasons that we are yet to understand.
We could not manage to overcome those problems, so we decided to partially use pseudocode in order to illustrate the functioning of our program. That said, all of the steps that require the use of contiki specific functions and features are written in C to show that we understand the functioning of contiki but could not manage to make the code work for the previously mentioned problems.

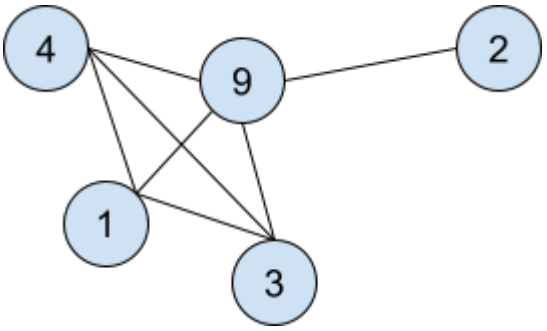
Data structures and design objectives

Every node in the simulation will contain a list of the neighbor nodes and a list containing the different groups which the node is part of. All the choices we made are made with the intent of minimizing the memory footprint and the amount of message exchanges required between IoT devices due to the lack of reliability of the network.
Every node in the simulation is identified by its IP, a group is a list of IPs.

Program functioning

The program Is composed of only one protothread that wakes up, initiates the discovery of the neighbor nodes then goes to wait until the timer expires; once the timer expires, this process is repeated. Before this cycle begins, the node will send to the backend the information regarding its age and nationality. We chose to send it to the backend at the beginning as this will reduce the memory needed to store the information of a node. Given the nature of IoT devices, we decided that the extra work of associating every node with its age and nationality that are needed by the Apache Spark component of the project was best placed in the Node-RED backend.
The discovery process starts with the node sending a “*discover*” message containing its own IP as a link-level broadcast which translates to every node that is in range of the radio. The

node then waits for the responses; each response will contain the address of the node that is responding, with this we can build a set containing all the addresses of the neighbors. In order to make sure that we are accounting for delays in the transmission, we set a timer so that the node will wait before proceeding further, giving the time for all the neighbors to acknowledge the *discover* message.
Once the timer expires the program will check if it is already part of any groups. If it is, it checks if any of the nodes of the group where it is present is missing. If anyone is missing it notifies all the members of the group, including the node that is missing, and if it is the leader it also notifies the backend.
In this step we make use of the assumption that all the nodes are always reachable by a root and thus they will all be available for communication through the RPL tree.
Also note that in this step we introduce a kind of race condition as every node is capable of removing any other nodes in the group like in the following example.



Let's suppose that there is a group formed by 1,2,3,4,9 and after a while 2 goes away but maintains a connection with one node.
In this case, for example we might have that the node 2 realizes before 1,3 or 4 that the others are missing and proceeds to remove 1,3 and 4 from the group.
If that is the case the correct group 1,3,4,9 will be reformed in the next iteration

After checking for missing neighbors, the node will send the newly constructed neighbor set to all its neighbors.
When a node receives a neighbor list from another node it will scan it for the possibility of adding it to one of its own groups. To do so, it checks if any of its groups is a subset of the list that is has just received, if that is the case and the node from which he received the list is not already in the group, it gets added to the group and, if the local node is the leader of the group, it notifies the backend.
After checking for subsets there will be another function called which will check for possible new groups to create. To do so we evaluate all the 3-members groups possible between the 2 nodes. Those are the groups that contain the local node, the one that sent the neighbors list and a node that is in common between the 2. If any of those groups is not a subset of another established group then a new group is created, the node that sent the list is informed of the creation and the backend is informed.

The notification of the backend is done through MQTT and 4 subjects:

- node registration
- group creation
- group deletion
- group update cardinality

In the first one the node address, age and nationality are passed as data. In the 2nd and 3rd the group is passed as data. In the last one we will pass the group, the operation (increase/decrease cardinality) and the address that is missing or that is to be added.

Backend Implementation (Node-RED)

The backend, implemented in Node-RED, listens for messages published by the nodes and processes them based on the topic received via MQTT. Each topic triggers different actions, allowing the backend to track and manage the state of the groups formed by the nodes.

1. Node Creation
When a message is received on the "NODE_INSERT_TOPIC" topic, it is used to register information about individual nodes. Each time a message with node details is received, the backend saves the node's information using the IP address as the hash key. This allows the backend to associate specific information about each node, such as the owner's age and nationality.

2. Group Creation
When a message is received on the "MQTT_CREATE_TOPIC" topic, the backend processes the creation of a new group. Each group consists of multiple nodes, identified by their IP addresses. To handle the creation:

- The backend first extracts the IP addresses of the nodes forming the group.
- It then generates a unique key for the group by concatenating the IP addresses in a consistent order.
- A timestamp is captured at the moment the group is created, and this timestamp is associated with the group in a map. In this map, the unique key (composed of the IP addresses) serves as the hash, while the timestamp serves as the value. This allows the system to track exactly when the group was formed.

3. Update Cardinality
The " MQTT_CARDINALITY_TOPIC" topic is triggered when there is a change in the membership of an existing group, either due to the addition or removal of a node. This requires the backend to handle two tasks simultaneously:

- Updating the Group Map: The backend first locates the group in the map using the hash key of the IP addresses. Depending on whether the cardinality is increasing (a new node joins the group) or decreasing (a node leaves the group), the backend modifies the group by adding or removing the relevant IP address. This ensures that the group structure remains accurate in real-time, reflecting the current members of the group.
- Updating Group Statistics: In addition to updating the group's structure, the backend maintains a second map dedicated to tracking statistics for each group. These statistics include:
    - Cardinality Variations: An array is maintained that tracks how the group's size (cardinality) changes over time. Every time the group's membership changes, this array is updated with the new cardinality.

    - Minimum and Maximum Cardinality: The backend keeps track of the smallest and largest size that the group has reached since its creation. If the new cardinality is lower than the current minimum, the minimum is updated. Similarly, if the new cardinality exceeds the maximum, the maximum value is updated.
    - Average Cardinality: The average size of the group is calculated dynamically by averaging all the cardinalities observed during the group's lifetime. This provides insight into the typical size of the group over time.

4. Group Deletion
When a "MQTT_DELETE_TOPIC" message is received, the backend handles the removal of a group.

- Removing the Group: The backend uses the hash of the group's IP addresses to locate the group in the map and remove it.
- Calculating the Group's Lifetime: Before deleting the group, the backend calculates how long the group existed. This is done by subtracting the timestamp of the group's creation (stored when the group was first added) from the current time. The result is the total lifetime of the group, typically expressed in minutes. This data may be used for reporting or statistical purposes.

Every time a group is created or its cardinality changes (due to a node joining or leaving), the backend sends individual MQTT messages for each node (IP) involved in the group. These messages contain the node's age and nationality, providing detailed information about each participant in the group.

Member statistics with Apache Spark

In order to compute the requested statistics over the various members that enter the groups during the life of the application, we need the help of the powerful tools offered by Apache Spark. Specifically we used the structured streaming paradigm in order to manage the incoming streaming of data sent by the node red backend.

Structured streaming idea is to treat a live data stream as a table that is being continuously appended. This leads to a new stream processing model that is very similar to a batch processing model. You will express your streaming computation as a standard batch-like query as on a static table, and Spark runs it as an incremental query on the unbounded input table.

In our case the data stream is supposed to arrive via MQTT connection, since we identify this as the best way to deal with IoT data. The data will arrive as a JSON string and will be parsed by the application, then assigned to a predefined schema, ready to be queried.

In order to deal with late time events, we use watermarking, which lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly. We introduce a global watermark of 1 day on the attribute timestamp of the

incoming data, ensuring that the data will be dropped in the case they arrive too late, in order to bound the memory usage of the application.

To perform the weekly moving average we firstly compute the daily average of members' age per nationality by defining a sliding window of 1 day over the nationalities and grouping by this window to average the age of users registered in this period of time, defining 'date' column the window end. Again we defined another 7 days watermark on the date column of the new table in order to process only data that are at most 7 days far from the latest data processed. We defined another 7 days window over the nationalities, and took the first and last day averages of the window to compute the weekly percentage increase. The top nationalities are then found by computing the max of the percentage increase per nationality.