IoT 2023 PROJECT 1

Name: Mirko Bitetto Person Code: 10928506

Name: Giulio Saulle Person Code: 10626444

Boot Sequence

The `Boot.booted()` event handler is triggered when the application finishes booting. It proceeds to start the radio using the `AMControl.start()` function. The `AMControl.startDone()` event handler is triggered when the radio start operation completes. Upon successful startup, if the current node is not the PAN Coordinator (Node 1), it starts **Timer1** in periodic mode with a specified timeout (Used to send the CONN message). On the other hand, if the current node is the PAN Coordinator, it initializes the **CommunicationNetwork** with a provided **networkTable** and starts **Timer4** (used for periodic publish on NodeRed) with a defined interval. However, if the radio startup fails, it attempts to restart the radio by calling the **AMControl**.**start()** interface again.

`Timer1.fired` triggers the nodes to send **CONNECT** packets. If the current node ID is not the PANC, it retrieves the payload pointer and sets the message type to **CONNECT** and the node ID to the current node ID. It then generates and schedules the transmission of the message by calling the `generate_send()` function. It is set as periodic Timer to retransmit in case the **CONNECT** or the **CONNECT_ACK** is lost.

`Timer2.fired` is responsible for triggering the nodes to send **SUBSCRIBE** packets. When **Timer2** expires, it prepares and sends a **SUBSCRIBE** packet to the **PANC**. The payload of the packet is filled with the appropriate values, such as **messageType**, **nodeID**, and **subTopic** based on the **clientInterest** array. The message transmission is then scheduled by calling `generate_send()`.
It is set as periodic Timer to retransmit in case the **SUBSCRIBE** or the **SUBSCRIBE_ACK** is lost.

`Timer3.fired` is responsible for triggering the nodes to send **PUBLISH** packets. When **Timer3** fires, it prepares and sends a **PUBLISH** packet to **PANC**. The payload of the packet is filled with the appropriate values, such as **messageType**, **nodeID**, **pubTopic**, and **payloadData**. The **pubTopic** is determined based on the **publishTopic** array, and the corresponding **payloadData** is generated randomly based on the topic (See sensor read section). The message transmission is then scheduled by calling `generate_send()`.

The **actual_send()** event is responsible for performing the actual sending of a packet using the TinyOS interfaces. It retrieves the payload of the packet and checks if the radio is locked. If locked, an error message is printed, and the function returns **FALSE**. Otherwise, it uses the **AMSend.send** function to send the packet to the specified address. If the send operation fails or the message type is invalid, an error message is printed, and the function returns **FALSE**.

The **generate_send()** function is used to store a packet and address into a global queue for sending messages. It checks if the message queue is full and adds the message to the queue if there is space. If timer0 (used to call **actual send()**) is not running, it starts the timer with a random delay.

The **AMSend.sendDone** event is triggered when a message has been sent. It checks if the packet was sent successfully. If so, it unlocks the radio and prints a debug message indicating the successful transmission. If there are remaining messages in the queue, it shifts the queue to remove the sent message and sends the next message in the queue if available. Timer0 is restarted with a random delay if there are more messages to be sent.

Receive Event

The **Receive.receive()** event is responsible for handling received messages. It first checks if the length of the payload matches the expected size. If it does, it proceeds to process the message based on its type.

If the node is the PAN Coordinator, it handles **CONNECT** messages by first checking if the sender is already connected. If not, it adds the sender to the list of connected nodes. Then, it sends a **CONNECT_ACK** message back to the sender, indicating successful connection.

Similarly, for **SUBSCRIBE** messages, the PAN Coordinator checks if the sender is connected and processes the subscription requests for temperature, humidity, and luminosity topics. It adds the sender to the respective subscription lists if not already subscribed and responds with a **SUBSCRIBE_ACK** message.

If the node receiving the message is not the PAN Coordinator, it manages timers based on **CONNECT_ACK** and **SUBSCRIBE_ACK** messages. Upon receiving a **CONNECT_ACK**, it stops Timer1 and starts **Timer2** with a periodic retransmission timeout. When receiving a **SUBSCRIBE_ACK**, it stops **Timer2** and starts **Timer3** with a periodic publishing interval.

When a **PUBLISH** message is received, the PAN Coordinator processes it by adding it to the transmit buffer for later transmission to Node-RED and forwarding it to other subscribed nodes (excluding the original sender). It schedules the transmission of the PUBLISH message to the subscribed nodes by using the generate_send() function.

Throughout the function, various checks are performed, data structures are updated, and debug messages are printed to facilitate message handling and monitoring. This ensures proper communication and coordination between nodes in the network

Transmitting Data to Node-RED

In the receive function, the transmitBuffer is filled by iterating over the received data and copying it to the corresponding elements of the transmitBuffer. This ensures that each element in the buffer stores the received data accurately. On the other hand, in the Timer4.fired() event, the transmitBuffer is used to send data to the Node-RED server. A UDP socket is created, and the data is converted into a string format. The server address and port are set, and the message is sent using the sendto() function. If the send operation is successful, a debug message is printed. The loop in the Timer4.fired() event iterates through the transmitBuffer, sending each entry to the Node-RED server. Once all the packets have been sent, the bufferIndex is reset to 0, ready to be filled with new data in subsequent operations.

Pubsub.h

The pubsub.h header file defines constants and data structures for thr publish-subscribe system. It sets the maximum number of connected clients (**MAX_CLIENTS**) and topics (**MAX_TOPICS**). It also defines timeout values for message retransmission (**RETRANSMISSION_TIMEOUT**), publish intervals for each client (**PUBLISH_INTERVAL**), and transmit interval to NodeRed (**TRANSMIT_INTERVAL**).

The header file includes an enumeration of topic types (**Topic**) such as **TEMPERATURE**, **HUMIDITY**, and **LUMINOSITY**. It also declares a boolean array **clientInterest** representing the interest of each client in subscribing to it.

The **publishTopic** array stores the topic on which each client is publishing.

The **pubsub_message** struct represents the message structure that will be exchanged between the nodes, with fields such as **messageType**, **nodeID**, **subTopic** (representing client's subscription topic), **pubTopic** (representing client's publishing topic), and **payloadData** (payload information like sensor readings). The distinction between **nx_uint8_t pubTopic** and the **nx_struct** block for **subTopic** allows the nodes to subscribe to multiple topics using a single subscription message.

Communication

Communication.nc defines two structs, **NodeInfo** and **CommunicationNetwork**, and provides several functions for managing a communication network of clients. The **NodeInfo** struct holds information about a client node, including its **nodeID**, **isConnected** status, and an array **isSubscribed** that keeps track of its subscription status for each topic. The **CommunicationNetwork** struct represents the overall network and contains an array of clients, which are instances of NodeInfo structs. The **initializeCommunicationNetwork()** function initializes the network by setting default values for each client, such as **nodeID**, **isConnected** (initialized as **FALSE**), and **isSubscribed** (all set to **FALSE**). Functions like **isConnected()** check if a given node is connected by iterating through the clients array and returning the corresponding **isConnected** value. The **addConnection()** function sets the **isConnected** status of a node to **TRUE**. **isSubscribed()** checks if a node is connected and subscribed to a specific topic, returning the corresponding **isSubscribed** value. The **subscribe()** function allows a node to subscribe to a topic by setting the corresponding **isSubscribed** value to **TRUE**. These functions enable the management of client connectivity and subscriptions within the communication network.

Sensor Read

SensorRead.nc includes three functions responsible for generating random sensor readings: temperature, humidity, and luminosity. The **generateRandomTemperature()** function generates a random temperature reading in Celsius, ranging from 0°C to 40°C, by utilizing the **rand()** function from the **<stdlib.h>** library. It calculates a random temperature within the specified range and returns it as an **uint16_t**. Similarly, the **generateRandomHumidity()** function generates a random humidity reading as a percentage, ranging from 0% to 100%. It uses the **rand()** function to produce a random number, scales it to fit the desired humidity range, and returns it as a **uint16_t**. Furthermore, the **generateRandomLuminosity()** function generates a random luminosity reading, ranging from 0 to 1023, representing the minimum and maximum luminosity values. It employs the **rand()** function to generate a random number, scales it to fit the desired luminosity range, and returns it as a **uint16_t**. These functions will be utilized by the nodes to simulate realistic random sensor readings for temperature, humidity, and luminosity.

PubSubAppC

The configuration, named "**PubSubAppC**" defines the components used in the application. The configuration consists of components such as "**MainC**," "**PubSubC**," "**AMSenderC**," "**AMReceiverC**," "**TimerMilliC**," and "**ActiveMessageC**" instantiated with specific configurations. The components are connected to their respective interfaces to establish communication and coordination within the application. The code sets up the necessary connections for booting, timers, message sending and receiving, and control of active messages.

Simulation with TOSSIM

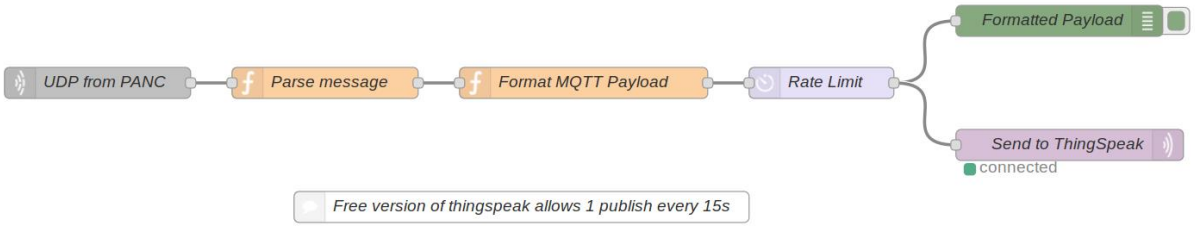
The simulation script is a modified version of the one used in Challenge 3. it includes support for more nodes, adds the debug channel for Node Red and uses a fixed length simulation time set to 180 seconds which is enough for 2 rounds of publish.

The topology has been set accordingly to the specification (8 client nodes and 1 PAN Coordinator), with all links bidirectional and -60dBm as gain for all entries.

The simulation log is saved in “**simulation.txt**”.

Node Red flow

The Node-RED flow is designed to receive UDP packets on port 3030, extract and parse the received data, format it, and then send it to ThingSpeak using the MQTT protocol. The flow starts with an "UDP in" node, which listens for incoming UDP packets and converts the payload to a string format. The parsed values, including **pubtopic**, **payloadData**, **nodeid**, and **simtime**, are extracted from the string and organized into an object called **parsedData** using a "Parse message" function node. The "Format MQTT Payload" function node takes the **pubtopic** and **payloadData** values, and constructs a payload string in the format **'fieldX=Y&status=MQTTPUBLISH'**, where X represents the incremented pubtopic value and Y represents the **payloadData** value. The flow includes a "Rate Limit" delay node, which introduces a delay between messages to adhere to the free version's restriction of one publish every 15 seconds on ThingSpeak. The formatted payload is then displayed in the Node-RED debug sidebar by a "Formatted Payload" debug node for debugging purposes. The payload is subsequently sent to ThingSpeak using the "MQTT out" node, which publishes it to the specified topic **"channels/2177976/publish"** on the ThingSpeak MQTT broker. The configuration of the connection to the ThingSpeak MQTT broker, including the broker's address, port, client ID, and other settings, is handled by an "MQTT-broker" node. This setup ensures the UDP packets are received, parsed, formatted, and sent to ThingSpeak at the appropriate rate, complying with ThingSpeak's limitations.



ThingSpeak channel

The data generated by the Nodes can be visualized more effectively from a user perspective by accessing the public ThingSpeak channel (https://thingspeak.com/channels/2177976). This channel is publicly viewable by anyone who has the channel ID (**2177976**) on ThingSpeak.