Politecnico di Milano - Department of Electronics, Information and Bioengineering

POLITECNICO
MILANO 1863

MSc Computer Science and Engineering
Academic Year 2016-2017

Internet of Things Project

Lightweight publish-subscribe application protocol

Student Testa Filippo
ID 875456

Advisor Redondi Alessandro
Professor Cesana Matteo

August 24, 2017

1 Project Specifications

The project requires to design and implement a lightweight publish-subscribe application protocol similar to MQTT and to test it with simulations on a star-shaped network topology composed by 8 client nodes connected to a PAN coordinator, which act as a MQTT Broker. The features that must be implemented refer to connection (CONN), publish (PUB) and subscribe (SUB) phases and it is required also to provide a QoS management with two levels (0 or *at most once* and 1 or *at least once*).

2 Design and Implementation

First of all, it's important to remark that the implementation of the two types of node, Broker and Client, is separated in two different source files: this allows to maintain quite low requirements in terms of both ROM and RAM size especially on the Client side, which are more constrained devices with respect to Broker node. As a consequence, this separation helps to preserve scalability, which is one advantage of MQTT-like protocols, but also changes applied to one node type don't affect the other one, given that the communication interface and the exchanged messages remain untouched.
Even though the implementation of Broker and Client is split up, there are some common values (such as maximum number of clients, number of supported topics and their names, Broker address) and data structures, related to the messages, that are gathered together in a shared header file, called *definitions*. Also, through this file, changing some values it is possible to activate a debug-like mode, where the verbosity of the console messages from all the nodes is higher, and a static/random mode where the subscription preferences of the Clients are fixed at compile time, and equal for all of them, or chosen dynamically in a random way for every Client.
The addressing schema adopted is quite simple, indeed the Broker node is fixed at address 1 while the Client nodes have addresses equal or greater than 2; this choice is helpful to support a variable number of clients, even higher than the requested 8 from the project specification.
The designed protocol actually supports only the three topics indicated in the specifications, however the implementation of both node types with small adjustments can support a larger number of them: in fact, the common header file contains an enumeration type with the identifiers of the various topics and an array of strings which is used to stored their names, for displaying purposes.
Finally, the designed protocol has been implemented on TinyOS and simulated with Cooja.

2.1 Messages

The messages sent between Clients and Broker have a precise and fixed structure: they have a node identifier, a message type and a payload.
The type of the message can be one of the following six: CONN, CONNACK, SUB, SUBACK, PUB and PUBACK; they are quite self-explanatory and each couple (that is a request and its relative ACK) refer to a specific feature that has been implemented in the protocol.
The initial part of every message is the node identifier: it is used to track the source node of each CONN, SUB and PUB message received at the Broker, but, differently, when a PUB message is received from a Client, the node identifier refers to the Client node that originated the publish message, thus allowing to better distinguish the received PUB messages and also to print on the console more detailed information.
Finally, the last part of a message is the payload and, exploiting the *union* construct, it can be a SUB or a PUB one; in order to correctly handle this payload, every node, before accessing it, should check the message type and act accordingly. The PUB payload contains the topic identifier of the publication, the

data value and the QoS level. When a PUB message is received from the Broker, it could answer with a PUBACK depending on the QoS level, while there isn't a specific type of acknowledgement when a PUB message is sent from the Broker to the subscribed nodes but it suffices to use the synchronous ACK provided by TinyOS, this also allows a simplification of the message management, as explained later in the Broker section.
The second type of payload refers to the subscription and for this reason it contains two arrays of boolean, which sizes depends on the maximum number of supported topics; the former one indicates whether the requesting Client wants or not to subscribe to every topic (1 means subscribe, 0 otherwise) while the latter contains the desired QoS level for each topic.
There is also another type of message that however is not sent on the network but it is only used by the Broker node to deal with retransmission (see Broker section for a better explanation).

2.2 Broker

The Broker node is the fundamental element of the protocol because it is responsible for managing all types of messages received, replaying to Clients' requests and also for relaying publish messages between nodes.
The connection management is rather simple and exploit a BitVector component to store the actual connection of each Client: upon reception of a CONN message, this data structure is checked and properly updated; also, a preliminary verification step on node connection is done before reading every received SUB message.
For the subscription management, the Broker relies on two different matrices of boolean, whose sizes are related to the total number of supported nodes and topics; the first one tracks the actual subscription of every node respectively for each topic while the second stores the QoS level for each couple node-topic.
Furthermore, to properly support the message reception, the Broker uses a first small queue (called handling queue) where all the received messages are stored waiting to be processed while another, bigger, queue (called retransmission queue) contains all the ready messages that will be sent on every periodic timer expiration.
On CONN, SUB or PUB reception, the Broker immediately tries to answer with the respective ACK message (remember that for PUB, the PUBACK is sent only if QoS is 1); if, for any reason, is impossible to complete this operation, the Broker displays an error on the console and it is the requesting Client node that, after a time-out expiration, will resend the message that wasn't acknowledged.
Differently, when the Broker relays PUB messages towards subscribed nodes, it uses the synchronous ACK mechanism provided by TinyOS to ensure that the message has been correctly delivered to the destination, and if it is not acknowledged, the message is re-enqueued in the retransmission queue, waiting to be sent again. In this case, it has been used this OS service instead of ad-hoc defined message and procedure, because it is simpler and otherwise it would require unnecessary additional data structure and complex tasks to deal with retransmissions.
In embedded devices, dynamic memory is not advisable and for this reason the two queues can't have unlimited size; various simulations have shown the chosen time-out interval and queue sizes values do not fill them entirely, however this could happen: in this case, the policy adopted on the Broker is to simply discard the message which is not possible to enqueue.

2.3 Client

The Client node can connect, subscribe and publish through specific messages sent to the Broker; this three activities depends on the values of two flags: they represent whether the Client is connected with

the Broker and has already subscribed to some topics.
At start-up, Clients can only connect to the Pan Coordinator and when the CONN message is sent, a timer is started; if the CONNACK isn't received within a defined time interval, the timer fires and the Client tries to resend again a CONN message.
Only after having received the CONNACK, the Client can send a SUB message but, also, it starts another timer used only for sampling. The retransmission procedure for subscription is the same as the connection phase and it exploits the same timer, but its behaviour changes accordingly to the flag values.
Once the SUBACK is received, the Client continues to sampling data from the attached Fake Sensor and for every received PUB message, it displays on the console the payload information.
As mentioned, the sampling activity is regulated by a dedicated timer that ticks on periodic time interval: at each firing, data is collected from the Fake Sensor, encapsulated in PUB message and sent to the Broker; the latest retrieved data is also saved for retransmission in the case the publish message is not acknowledged.
The attached Fake Sensor module is simply used for simulating a real sensor that, when called, return a data measurement but, in this case, the data returned is just a random one.
The Client node, on start-up, has to decide on which topics subscribe and chose a topic for publishing data; for this purpose, two specific functions have been implemented. Since each Client can publish on at most one topic, this one and its QoS level are chosen based of the node identifier, thus, when the number of clients equals or exceeds the number of topics it is automatically guaranteed that there is at least one node which publish on each topic.
On the other hand, it is required that at least three nodes will subscribe to more than one topic: in order to ensure this condition, a static assignment for topic subscription and related QoS is already provided and hard-coded; even though this assignment is equal for all Client motes and it allows to satisfy and to test the project specifications, it could seem not so realistic. For this reason, the static assignment mode can be disabled changing a specific flag in the *definitions* file, thus activating the dynamic one, where the subscriptions on topics and relative QoS levels are chosen in a random way. Notice, finally, that to fulfil the requirements in case of random assignment, every node should have a quite complex function that also must know the decision taken from the other motes which would cause an increase in exchanged messages but also an overhead for their handling.

3 Simulation

The simulation environment chosen is Cooja because it allows to create scenarios where nodes can execute different programs, thus is it possible to run both Broker and Client mote types; contrarily, this could not be possible with TOSSIM.
As requested by the project requirements, the implemented protocol has been tested with 8 Client nodes connected to the Broker, there are at least 3 motes that subscribe to more than one topic and the payload of publish messages is a random number from the Fake Sensor component.
The provided companion log file shows the behaviour and the interactions of the nodes through console messages; in order to properly verify the acknowledgement and retransmission procedures, at specific time points a random Client node has been moved in and out of the communication range with the Broker to purposefully trigger these procedures.