Lightweight publish-subscribe application protocol

Project Report

University:

Politecnico di Milano

Academic Year:

2016/2017

Course:

Internet of Things

Students:

- **Giuseppe Manzi**
  mat. 854470
  giuseppe1.manzi@mail.polimi.it
- **Vito Matarazzo**
  mat. 853095
  vito.matarazzo@mail.polimi.it

1. Introduction

1.1 Goal

The aim of the project is to implement a lightweight publish-subscribe protocol for IoT applications based on personal area networks. This protocol is a simplified version of the MQTT protocol.

1.2 Requirements

The protocol is based on two types of actors: the broker and the client. The broker corresponds to the PAN coordinator and is in charge of receiving and processing connect, publish and subscribe requests coming from clients connected to the same PAN.

- **Connect**: Client must be able to inform the broker that it wants to connect to the network and that it is possibly going to interact with publish and/or subscribe messages in future. Every *CONNECT* message must be acknowledged.
- **Subscribe**: Client must be able to subscribe to one or more of the available topics (*TEMPERATURE*, *HUMIDITY*, *LUMINOSITY*), specifying its ID and a level of Quality of Service (hereinafter QoS) for each topic. Every *SUBSCRIBE* message must be acknowledged.
- **Publish**: Client must be able to publish data on one topic, specifying the desired level of QoS and the value for that topic. A *PUBLISH* message must be acknowledged by the broker only if the message specifies QoS greater than 0. This type of message must be forwarded by the broker to the clients subscribed to the publishing topic, with the QoS specified by each client.

Finally, the protocol must manage two levels of QoS:

- **QoS = 0**, meaning that the message must be sent at most once (no acknowledgement);
- **QoS = 1**, meaning the message must be sent at least once (if it isn't acknowledged it is resent).

2. Design and implementation

We chose to implement the application protocol using TinyOS, because it's more used among the community with respect to Contiki and its documentation is wider. Since the protocol involves two different types of actors (the client and the broker), we designed them as 2 different applications that interact through radio messages.

2.1 Messages

Every message specifies its type in a specific field (***msg_type***) so that the receiver can check it to choose what to do. We decided to manage all acks using the PacketAcknowledgements interface natively provided by TinyOS.

- **Connect message**: It has no additional fields.
- **Subscribe message**: It is sent from clients to broker and specifies the ***topic*** to which the client wants to subscribe and the ***QoS*** to be used when a related "publish" message will be forwarded by the broker to the client. We decided to divide each subscription that a client wants to do into different Subscribe messages to avoid dynamic message length or oversizing it.
- **Publish message**: It carries the publishing ***topic***, the published ***data*** on that topic and the ***QoS*** for the current transmission (which changes when it is forwarded by the broker).

Subscribe and publish messages also carry a ***msg_id*** field that univocally identifies the message sent by a specific mote independently of the receiver (i.e. the tuple <sender, *msg_id*> is univocal).

2.2 Client

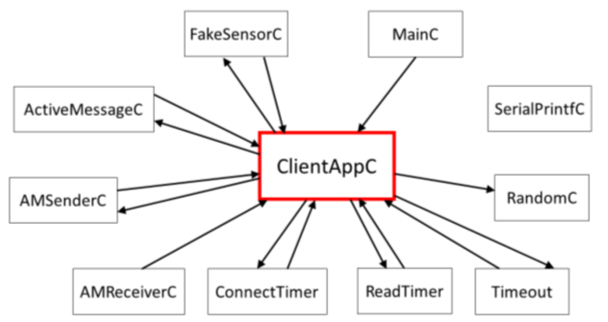


Fig 1: Client Application with its connected components.[1]

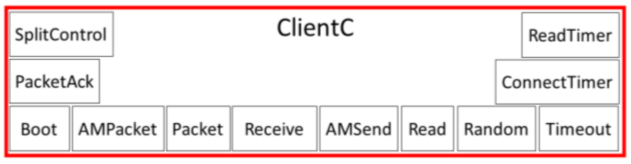


Fig 2: View of the component ClientC and the interfaces it uses. ReadTimer, ConnectTImer and Timeout are three different Timer interfaces.

The Client App is made of one component (*ClientC*) which contains all the application logic of a client mote. For the sending and the receiving of messages, it uses the Active Message (AM) layer to allow multiple accesses to the radio (in our case AM type is 6) and to specify addresses for each transmission. Once booted, each client waits, using a one-shot timer (*ConnectTimer*), for a time equal to its own ID times 500ms before sending the Connect message to the broker, in order to avoid possible collisions at the beginning. After receiving the Connack message (through the PacketAcknowledgments interface), the client can start publishing or subscribing to a specific topic.
After sending a Subscribe message to a topic, the client checks if the packet was acked, and if not, the client resends it immediately.
As for the publishing, each client chooses a single topic (*my_topic*) on which it wants to publish and starts a periodic timer (*ReadTimer*) that triggers the reading of a new value from the sensor. We simulate it using *FakeSensorC*, which reads a random value. When the read is done, the client sends a Publish message, with a specific QoS (in our case randomly chosen), to the broker. If the selected QoS is greater than 0, the client waits for the ack, and if it doesn't arrive, the client waits, using the *Timeout* timer, for a random timeout between 100ms and 500ms before resending the message. The random timeout is useful to avoid repeated failed transmissions. Moreover, if a Publish message hasn't been sent successfully before a new value is read, the old value is discarded and the new one is sent, since it usually doesn't make sense for the subscribers to receive a non-up-to-date value.

2.3 Broker

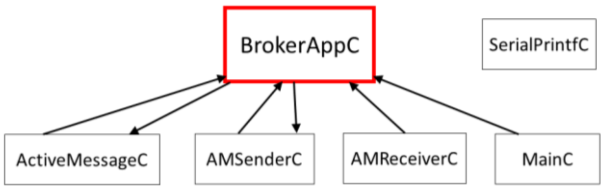


Fig 3: Broker Application with its connected components.[1]

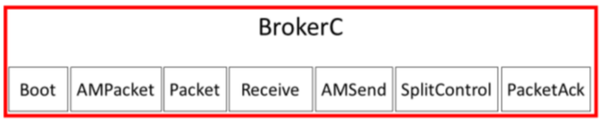


Fig 4: View of the component BrokerC and the interfaces it uses.

The Broker App is made of one component (*BrokerC*) containing all the application logic of the broker node. For the sending and the receiving of messages, as the client, it uses the Active Message (AM) layer. Similarly to clients, also the broker has an ID, that is always the highest in the network, since it is equal to MAX_CLIENTS + 1.
Once booted, the broker waits for any message received from the clients. In order to save the status of connection and subscription of each client, the broker uses a *subscription_matrix*, where the element in position (i, j) represents the QoS level of the subscription or the connected/disconnected status (in case there isn't a subscription for that specific position yet) for topic i and client j.

[1] Arrows from the App to another component indicate that the App calls that component's commands. Arrows from a component to the App indicate that the component signals events to the App. The component SerialPrintfC is automatically wired to the configuration and is called through printf functions. That's why it has no arrows towards or from the App.

When a Connect message arrives, the broker saves the connected status for that client ID, writing NOT_SUB in the column of the matrix associated to that client.
When a Subscribe message arrives, the broker saves the subscription writing the received QoS level in the corresponding cell of the matrix.
When a Publish message arrives, the broker forwards the message to all the subscribed clients one by one using the QoS value stored in the subscription matrix. Unlike the client, if the broker has to retransmit a packet with QoS greater than 0, it retries immediately, to avoid delays that would slow down the broker, that has to process a higher quantity of messages.
If a new Publish message is received while the broker is forwarding a previous one, it saves the new message in the one-element queue related to that topic (*queued_msgs*). This is to avoid that, since the send procedure is asynchronous, the last received packet overwrites the one that is being forwarded at that time. Once finished the forwarding, the queued messages are processed in the same way.

2.4 General coding approach

The main idea we followed during the implementation of the project was to try to minimize the memory consumption of the application, since it is intended to run on devices with limited resources, in particular Telosb motes. That's why we avoided to declare constants in the typical C way, but we used enumerations. We also avoided to declare enum variables, but we used the values directly. We also tried to not declare variables unless it was strictly necessary, losing some readability but improving the efficiency.

3. Comments on the simulation

3.1 Simulation flow

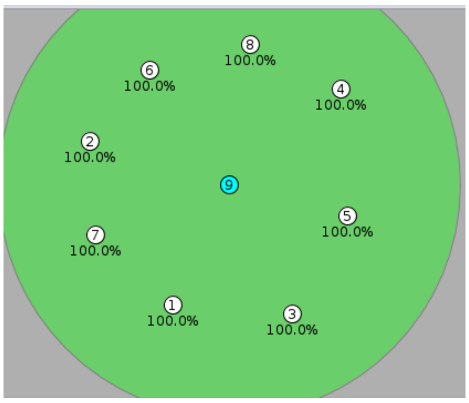
For the simulation execution, each client follows this flow: connection, subscription to all the available topics except for its own (*my_topic*), periodic readings and publications (every 4 seconds). The broker, instead, has not a specific flow but it just reacts to the received messages.

3.2 Comments on the log file

The log was made using printf functions (provided by SerialPrintfC component). Each print starts with the name of the application that is printing (CLIENT or BROKER) followed by the name of the function in execution.

3.3 Key events

Time 00:10.435: A publish message arrives to the broker while it is forwarding the previous one so it enqueues it.
Time 00:10.440: The previously enqueued message is dequeued and forwarded to clients subscribed to the related topic.
Time 00:13.230: We moved node 3 out of the broker's radio range. Acks for the publish message sent by the broker with id 9 are not received, so the broker retries. (Fig 5)
Time 00:16.552: Node 3 reenters the radio range and therefore it receives the message sent by the broker.



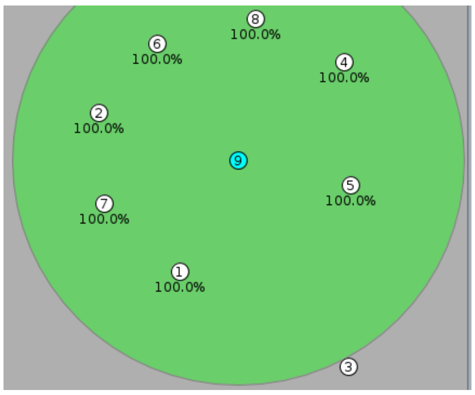


Fig 5: On the left: Starting topology of the network during the simulation. On the right: Node 3 is moved out of the broker's radio range.