Lightweight publish-subscribe application protocol

Luca Fochetta, Andrea Martino

September 17, 2017

1 Introduction

In this document we will try to summarise how we implemented a lightweight version of *MQTT* on *TinyOS* running on *TelosB* motes. We have extensively tested our implementation in a simulated environment provided by *Cooja*. You can find the complete implementation on *https://github.com/fochetz/IoTProject2017*.

2 Modules

2.1 Common

We have two components that are in common between the *PanC* and the *Client*:

- **QueueSenderC**: a special sender that receives a generic message, inserts in a queue and then sends it. If the packet is not acknowledge it is reinserted in the queue tail. The component will periodically try to send the first packet in the queue. *QueueSender* can potentially be used by any components that needs to reliably send a message.
- **PublishModuleC**: This module sends the publish message(using the *QueueSender*) and signals trough an event whenever a publish message is received.

2.2 Client

Client is built upon a main module, **ClientC** that uses the two common components and the following ones:

- **ConnectionModuleC**: sends *CONNECT* message to the *PanC* and receives *CONNACK* message. When received, it signals an event to the *Client* main component.
- **SubscribeModuleC**: sends *SUBSCRIBE* message (if the node wants to subscribe to some topic) and receives *SUBACK* message. When received, it signals an event to the *Client* main component.
- **FakeSensorP**: simulates different sensors and signals when a specific sensor is read to the *Client* main component.

2.3 PanC

PanC is built upon a main module, **ServerC** that uses the two common components and the following ones:

- **ConnectionModuleC**: handles the receive of the *CONNECT* message, signals it to the *Client* main component and implements the necessary method to send *CONNACK* to the node and add the node to the list of connected devices.
- **SubscribeModuleC**: handles the receive of the *SUBSCRIBE* message, signals it to the *Client* main component and implements the necessary method to send *SUBACK* to the node and add the node to the list of subscribed devices.

3 Execution flow

Here we summarise the execution flow of the simulated environment. For more details please check the log included in the repo.

3.1 Client

The node boots and starts the radio. When it's ready it starts a timer that will periodically ask the *ConnectionModule* to send *CONNECT* to *PanC*. When *CONNACK* is received then the main module is notified and then another timer is started to periodically send, if necessary, a *SUBSCRIBE* message through *SubscribeModule*. When *SUBACK* is received then the main module is notified and it starts to periodically read data from sensors. Then, if needed, it asks *PublishModule* to the data to the *PanC* using the desired QoS. In our implementation QoS, publish topic and subscribe topic are function of the *TOS_NODE_ID* but it can be easily customized. Meanwhile *PublishModule* will receive published data by other nodes to which this node is subscribed to and will notify the main module passing the newly received data.

3.2 Server

The node boots and starts the radio. *ConnectioModule* signals the main module on every *CONNECT* received. The main module then asks *ConnectionModule* to reply to the sender with a *CONNACK* message to end the connection process. *SubscribeModule* notifies the main module on every *SUBSCRIBE*. The main module then ask *ConnectionModule* to check if the device is already connected. If so the main module accepts the subscription sending back a *SUBACK*. *PublishModule* signals the main module on every *PUBLISH* received. The main module then checks if the node is connected and if the node is always sending message with the same topic. If it's the case then it asks *SubscribeModule* the list of subscribed nodes to the specified topic and what QoS they prefer. The main module then finally sends the data through *PublishModule*.

4 Implementation choices

In this section we try to justify and explain our most important design choices.

4.1 ActiveMessages

In this project we need to handle different kind of messages, with different content and different purposes. One of the possible choices was to add a *packetType* field in the packet payload. Doing so we would have been able to understand the packet type just by simply looking at the first four bits of the payload. For example we could have used *0010* as the type id for the *PUBLISH* message and *0011* as the type id for the *SUBSCRIBE* message.
We didn't particularly like the idea of having one single component doing this check passing through a single *Receive*. So we decided to follow a different approach. We use as many components as the number of different packet we need to sort. For example in the *PanC* we use three different modules (*PublishModule*, *SubscribeModule*, *ConnectionModule*) that implements different *Receive.receive* each. Every *AMReceiverC* is built with different *Active Message ID*. So when *PanC* receive a *Publish* message only one of those *receive* event will be signaled.
Doing this kind of check at *Active Message* level makes our code cleaner and more expandable. It's very easy to add another type of message. You only need to initialize a *AMReceiverC* with a not used *Active Message ID*.

4.2 Events

Every component in our project heavily relies on events signaling in order to never wait for some data to be available. For example a publication is received in *PublishModule* the *PublishModule* itself will signal *PublishModule.OnPublishReceive* that is implemented by the *PanC* or the *Client*. Doing so allow us to split some of the logic between modules and the main component without ever incurring in heavy coupling between components. For example in the *PanC PublishModule.OnPublishReceive* publish message data is handled by *SubscribeModule* to get the list of subcribed nodes and then sent to the proper nodes via *PublishModule*.

4.3 Acks

We use "explicit" *SUBACK* and *CONNACK* messages in *SubscribeModule* and *ConnectionModule* but we have decided to use "implicit" automatic lower level *Active Message* acks to easily handle ack requests for messages with QoS 1.

4.4 Messages

We use three different message structures. *struct* details can be seen in *Common/packets.h*.

- **simple_msg_t**: it only contains the sender ID. It is used for "simple" messages like *CONNECT*, *CONNACK* and *SUBACK*;
- **sub_msg_t**: contains every information for *SUBSCRIBE* needed to handle subscriptions.

- **pub_msg_t**: contains every information for *PUBLISH* needed to handle publishes.

5 Diagrams

Here we provide some example diagrams to give an idea on how our implementation works. You can find some insights on how *QueueSender* works in *DocsDiagramSend Procedure.jpg*.

Figure 1: Component diagram: shows how components are linked to each other.

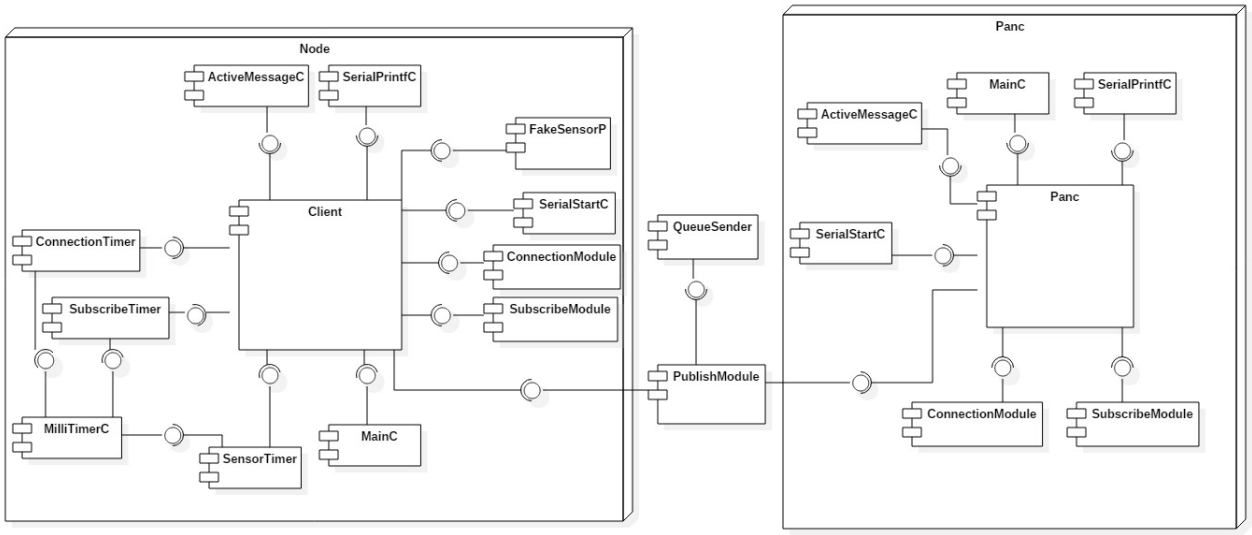

Figure 2: Sequence diagram of *Publish* procedure.