# Lightweight Publish-Subscribe Application Protocol

Matteo M. Fusi, Paolo Mosca

August 17, 2017

## Contents

# 1 Introduction and General Informations

## 1.1 Purpose of the Document

This document is part of the project of the *Internet of Things* course at Politecnico di Milano for the Academic Year 2016/2017. This document is associated to the source code at the link https://github.com/fusiled/tinyos-simple-mqtt.

## 1.2 Essential Informations

The project is written with the TinyOS framework. We developed two kinds of nodes that must be run in Cooja simulation environment: ***PanC*** is the PAN Coordinator and ***NodeC*** is responsbile of collecting measurements and sending them to the PAN Coordinator. *NodeC* components can send/receive messages to/from *PanC*. There can be only one *PanC* in the network and up to 8 *NodeC* components. How the two components are wired and structured is described in the section named *Description of the Components*.

### 1.2.1 *TOS_NODE_ID and Address Layout*

*PanC* must have *ActiveMessageAddressC__addr* set to 9. *NodeC* components rely on this information when they have someting to send. *TOS_NODE_ID* can have any value $\geq 8$. *NodeC* components has a *TOS_NODE_ID* in the range $[1; 8]$ and they have *ActiveMessageAddressC__addr* equal *to TOS_NODE_ID.* Every message has a *node_id* field attached to it and the meaning changes on the message_type (See *Layout of the Messages* for further details). *node_id* is set in messages as *TOS_NODE_ID-1.* With this method it is possible to save 1 bit in the *node_id* field of the messages. The method of how to properly set the *TOS_NODE_ID* symbols is described in *Simulation Environment* section.

# 2 Description of the Components

There are two approaches to handle messages: one it is based on timeout and it's used with *connect/connack* and *subscribe/suback* messages and the other uses a resend buffer when a send fails for some reason. The first method is ok for simple send-and-ack messages (like, in fact, connect and subscribe message sequences): we don't have any kind of timing constraints, so it's tolerable to connect a node not immediatly or know which topics a node is interested. What we wanted was to ensure that informations (the measurements) arrived at their destination, because they're the most important data in the network. That's why we implemented the resend buffer: to ensure that surely *publish* messages will arrive to thei destination.

## 2.1 Node

A Node can send only *connect,subscribe,publish* and *puback* messages. For the sake of simplicity, interested topics and QOS of the topics are hardcoded and are based on the *NODE_ID* of the node.

**The State Machine** A node has a state variable (called *state*) which controls the behaviour of the component. If *state* is set to:

***NODE_STATE_CONNECTING*** The node can only try to connect to PAN Coordinator. Other operations are not possible

***NODE_STATE_SUBSCRIBING*** The node has received *connack.* Now it's sending the *subscribe* message and/or is waiting for the *suback.* In this state the node can collect measurements and publish them.

***NODE_STATE_PUBLISHING*** The node has received the *suback.* The node now must only collect and publish measurements.

The behaviour is briefly explained by the following finite state machine: .

***connect* and *subscribe* handling** These two messages are controlled by a timeout timer: when there's an attempt to send one of these messages a timer is started. If i don't receive the realted ack (a *connack* or a *suback*), then the node will try to send another message of the same kind. Note that *PacketAcknowledgment.requestAck* call is not needed.

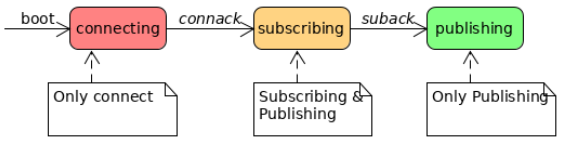


Figure 1: Control State Machine of a Node

***publish* and *puback* handling** They use a different mechanism: at first there's an attempt to send the message, if the sending procedure fails for some reason (the network component is busy or the ack set by the *PacketAcknowledgment* interface is not received when QOS=1), then the message is stored into a resend buffer that it's a simple queue. This resend buffer try to send a message every *RESEND_DELTA_TIME* milliseconds. If the buffer is full then a packet is discarded, so it's important that the parameters of the buffer are well-sized. If the resend buffer fails to send a message, then this message is put at the tail of the queue.

**Collecting measurements and sending them** The node implements a timer which triggers periodically every *SENSOR_PERIOD* seconds. When the time triggers one of the three sensors is selected and the related command needed to fetch the measurement is called. The three possible type of measurements are temperature, humidity and luminosity.

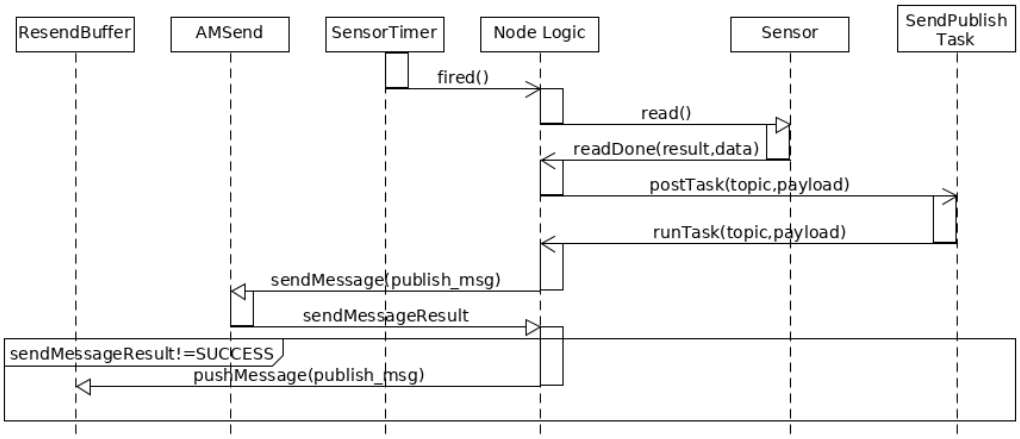


Figure 2: Sequence Diagram of the Collection and Sending of a measurement

## 2.2 PAN Coordinator

PAN Coordinator is a passive agent because it acts only when it receive a message.

***connect* and *subscribe* handling** When the PAN Coordinator receives a *connect* it marks the node who sent the *connect* as active (it keeps an array of boolean to keep the in memory the state of every node) and replies with a *connack*. The same methodolgy is used with *subscribe* messages but the node saves the topic in which a node is interested and the associated QOSs. The peculiarity is that the PAN Coordinator acts lazy when it must reply to *connect* and *subscribe* messages: if the PAN Coordinator can't send *connack* or *suback* the it just fails and it will do nothing. The PAN Coordinator will try to resend another *connack* or *suback* when the node who has not received the associated ack message will resend the *connect* or the *subscribe* because the timeout associated is expired.

***publish* and *puback* handling** When the PAN Coordinator receives a *publish* it replies immediatly with e *puback* and it forward the *publish* to all the interested nodes. It uses *PacketAcknowledgment.requestAck* in case the destination of the nodes required QOS=1 for the topic of the inquiring *publish*. If one of the messages cited fails then it is put in a resend buffer which acts exactly like described in the Node section: it tries to send the element in its

queue every *RESEND_DELTA_TIME* milliseconds. *PacketAcknowledgment.requestAck* in case QOS=1 guarantees that the packets are received by the destination node and the mechanism helps to know if we must retransmit the message without complex mechanism: if the ack is not received, then resend.

# 3 Messages

## 3.1 Layout of the Messages

The messages have been designed in order to be as small as possible. The structures and the functions realted to messages are contained in the source file *commons/Commons.h*. What and when a message is sent is descibed in the section named *The Implemented Publish-Subscribe Mechanism*. Two fields are common in every type of message and their size and positioning in the message layout is fixed:

**code_id** It is a 3bit field. It occupies the 3 less significant bits. This field contains a code that identifies the message.

**node_id** It is a 3bit filed. It occupied the 3 less significant bts after the *code_id* field. This filed contains a node_id. It is usually identifies the source of a message if sent from a *NodeC* components. If a message is sent from the *PanC* module, then it identifies the destination of the message, but when the mesaage is a publish then it contains the id of the node who published.

**CONNECT** The *code_id* is *1*. It is 8 bits big. It has not any other field other than *node_id* and *code_id*. The *node_id* tells to then PAN Coordinator which node requested a connection.

**CONNACK** The *code_id* is *2*. It is 8 bits big. It has not any other field other than *node_id* and *code_id*. The *node_id* tells to which node the *connack* message is destined.

**PUBLISH** The *code_id* is *3*. It is 32 bits big. The other fields after *node_id* and *code_id* are*:*

***publish_topic*** This field is 2 bits big. It identifies the topic of the publish: 0 is temperature, 1 is humidity and 2 is luminosity.

***publish_id*** This field is 8 bits big. It uniquely identify a publish. The range of the ids is limited, but we assume that it's not possible that there are in the network 2 messages with the same id at the same time. This id is used to match *publish* with related *puback* messages.

***publish_qos*** This field is 1 bit big. If the source of the message is *NodeC*, then it tells to PAN Coordinator if send a *puback* or not. If the source of the message is *PanC*, then it is the qos related to *publish_topic* of the destination *NodeC* specified previously int the *subscribe* message.

***publish_payload*** This field is 15 bits big. It's the value of the collected measure.

**PUBACK** The *code_id* is *4*. It is 16 bits big. The other fields after *node_id* and *code_id* are*:*

***puback_topic*** This field is 2 bits big. This field contains the topic associated to the related *publish* message.

***puback_publish_id*** This field is 8 bits big. This id is used to match *publish* with related *puback* messages.

**SUBSCRIBE** The *code_id* is *5*. It is 16 bits big. The other fields after *node_id* and *code_id* are*:*

***topic_mask*** This field is 3 bits big. This is mask tells to PAN Coordinator if the source of the message is interested or not into a specific topic. The least significant bit is associated to temperature, the second least significant bit is associated to humidity and the remaining bit is associated to luminosity. If a bit has value 0, then the node is not interested into a topic, otherwise yes.

***qos_mask*** This field is 3 bits big. It is composed exactly as the *topic_mask* field, but it expresses how the PAN Coordinator must behave when it sends a *publish* to the inquiring node.

**SUBACK** The *code_id* is *6*. It is 8 bits big. It has not any other field other than *node_id* and *code_id*. The *node_id* tells to which node the *suback* message is destined.