Towards standardization of MQTT-alert-based sensor networks: protocol structures formalization and low-end node security

Georgios Vrettos[1] Evangelos Logaras[2] Emmanouil Kalligeros[1]

[1] Dept. of Information and Communication Systems Eng., University of the Aegean, Samos, Greece
[2]Infineon Technologies, Graz, Austria
Email: [gvrettos, kalliger]@aegean.gr, evangelos.logaras@infineon.com

***Abstract*—We present a small scale sensor network application as a testbench to explore different setups in terms of hardware/ software, network protocol and data processing/storage scheme options, focusing on alert-based sensor systems with long idle times. Our specifications required the deployment of the network using the MQTT protocol over an encrypted TLS connection. We focused on using sensor nodes with low-power consumption profile, as well as on the formalization of the MQTT protocol's basic elements, by clearly defining the topic/message scheme used across the network. In addition, we experimented on how to combine locally stored information with data from popular cloud-based platforms and also acquired results regarding the processing performance of the nodes using different data exchange formats and database technologies. Work in progress includes data preprocessing on the network edge targeting distribution of the processing power across the network and network-traffic limitation, and also big data post-processing on server side or on dedicated high performance nodes to reveal hidden data patterns.**

***Keywords*—sensor network, MQTT, hw/sw, cloud, security**

I. Introduction

Within the next few years, billions of different types of connected computing and sensor devices will be used to monitor life in different environments, like home/car applications, industrial/metropolitan areas, remote wildlife areas and in the oceans. With the technological progress that has been achieved during the last decades in domains like networking and Internet of Things (IoT), telecommunications and embedded electronics, it is now possible for all these devices to form a huge network acting like a "skin" for our planet [1]. It is highly likely that even single chip self-powered sensor devices with networking capabilities can be found in the most remote areas, pushing data to the Internet. While adding devices to the network will be a trivial process, computer engineers already face questions on: i) what kind of network protocol technologies are suitable for the next years and decades to support all these new devices, ii) where to store and process the vast amount of collected data, and iii) how to secure all these data transmissions, with nearly every computing device having networking capabilities.

In this paper, by using a small scale sensor network for environmental and infrastructure monitoring, we try to provide/improve solutions on the following hw/sw and network related topics: a) the use of the Message Queuing Telemetry Transport (MQTT) machine-to-machine protocol and the formalization of its data structures (topics/commands format, use of XML/JSON data format). We chose MQTT because it has a very small footprint [2] and its implementation is feasible, even with encrypted data transmissions, on very small devices. b) Different hw/sw options for the network nodes to efficiently handle data in a secure way, c) database management options to store and share data, d) combining local data traffic with data stored/retrieved to/from the cloud, e) data preprocessing on the field to reduce network traffic, and f) data post-processing (online/offline) on server side to extract hidden data patterns, using dedicated hardware co-processors, e.g., Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPUs), etc. The main contribution of our work is twofold: i) we improve the formalization of the MQTT protocol data structures, since no standardized syntax exists at the moment [3]. Also, since security is the biggest concern of IT and semiconductor industry [4], as billions of devices will be connected to the network in the next few years, ii) we experimented on the encryption mechanism of the MQTT protocol and how feasible is to apply it, in terms of processing and power consumption, to very small and resource restricted devices at network edge. To the best of our knowledge, no concrete results exist in the literature in this area.

A. Network Topology, Motivation and Design Goals

The topology of our network architecture is presented in Figure 1. Locally, the highest level of the network is a Linux server running the MQTT broker (level #2), and managing the database, which collects incoming data, and an efficient dashboard to present these data, in a meaningful way, to system administrators. The server might also have high performance computing capabilities to process data and may also push data to other cloud platforms (level #3), like ThingSpeak, Google Cloud, etc. The server may collect data directly from low level MQTT client nodes at the edge of the network (level #0), or intermediate levels with more powerful nodes (level #1) may filter/compress data to reduce network traffic. Nodes at level #0 do not establish long-lived connections with the broker but rather communicate in an alert-based manner to report data periodically or based on interrup/timeout events. Although these nodes make use of very low-end hardware, we want them to be able to: a) connect to the MQTT broker over local/long range WiFi or 3G/4G networks, b) get into low-power/sleep mode to save power, but also include hardware that can support MQTT message encryption using the Transport Layer Security (TLS) protocol, c) support software execution using threading and/or interrupts to handle MQTT traffic, as well as interrupt events from connected sensors, d) support, if possible, Operating System (OS) and file based software, and e) support Over the Air (OtA) re-programming and configuration.

One main concern of our work was to be able to efficiently manage and store incoming data from edge nodes, so that it becomes easier to process and share the information either in local level or in the cloud as open data. Shared data, relating to

978-1-5386-4155-2/18/$31.00

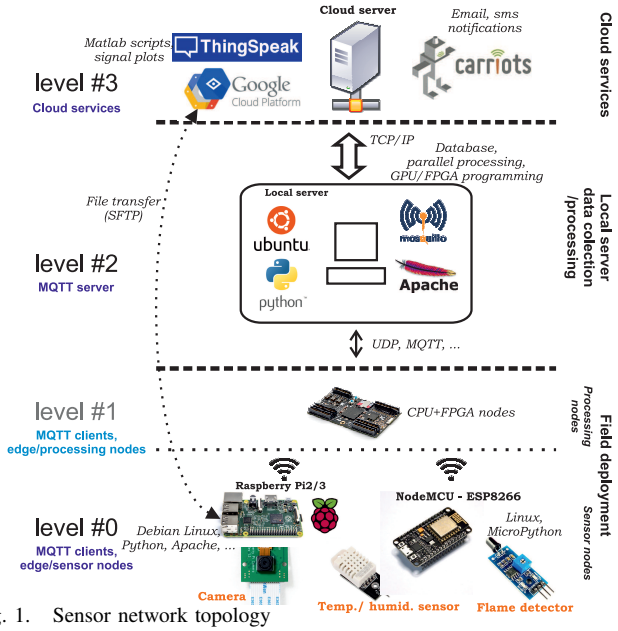

Fig. 1. Sensor network topology

local weather, building surveillance, traffic information, etc., in cloud platforms, can really improve every day life, especially in small communities like the island of Samos in Greece, where IT infrastructure is limited. Having open data as a high level goal, we tried to establish some hard criteria for the topic names and the content of data messages, the two main data structures of the MQTT protocol, for which there are no well-established and accepted formats [3]. Of course, the MQTT protocol has been utilized in many previous Wireless Sensor Network (WSN) applications but, most of the times, along with quite complex and power demanding hardware at the edge of the network and also without really specifying or providing a schema of the MQTT data format [5].

In our application, we support the idea that at the edge of the network, the lowest possible level of hardware should be used that supports encrypted network connectivity with low-power consumption. Although MQTT WSN solutions based on 8-bit uC architectures [6] really favor low power, unfortunately cannot handle TLS encryption. The use of a Python based framework along with the Advanced Message Queuing Protocol (AMQP) has been presented in [7] and while the protocol favors faster communication throughput using long-lived node connections, the authors do not provide clear performance results on the maximum number of supported nodes or data throughput through the network. Using our network architecture, we try to provide experimental results about the MQTT message network throughput supported by our hardware and also about the maximum number of nodes that our server/broker can support, by measuring the server message processing time (refer to Section IV). We have planned also to measure and report the power profile of our devices, related to MQTT message traffic and throughput.

II. Network Layer Using the MQTT Protocol

In our application, we wanted to establish a machine-to-machine communication, where sensor nodes send short messages (less than $1KB$) to report conditions in a certain indoor or outdoor area. The MQTT protocol [8] has been used already in many signaling and control telemetry applications, so that companies can get millions of products connected to their servers, and even as a short message chatting platform (Facebook messenger).

The protocol standard provides the means for the broker and the client nodes (CN) to create message topics, to which each node can subscribe and listen to incoming messages or post data in the form of short messages. The protocol also specifies three different levels of Quality of Service (QoS) for the transmitted messages and also provides an encryption mechanism based on TLS. A connection is made to the broker and a CN may transmit data or may just listen to some topics. So, the broker is not aware about the state of the CNs, in case they are connected but constantly in a passive/listening mode. Also, despite the fact that there are some restricted/adminsitrative topics, the CNs may freely create new topics and publish data to them. In terms of message content, the protocol specifies a type of retained messages (last published data on the topic) and also a type of last-will message, broadcasted by the broker in case a CN disconnects in an abnormal way.

In our network, an XML node configuration file is associated and stored in each network node, including the broker. This file holds important information about each node: a) geolocation information, b) type of hardware and attached sensors, c) type of software and OS, d) type of data types provided by the sensors, and e) network parameters like ID and IP address. The content of the configuration file is parsed during hardware boot by every node and also transmitted by each CN to the broker during a connection/validation process, where the broker validates the XML file against a defined schema. The CN validation process is further explained in the following sections. After a successful CN node validation by the broker, the XML configuration file is stored in the database and associated to the specific CN, while work in progress will also provide a mechanism so that the server can modify and update OtA the configuration file of a CN.

Handling of the XML configuration file in our application is closely related to the formalization of the MQTT protocol data structures. Each CN, according to the number and type of attached sensors, is able to automatically create the name of the topic that will use to publish its data, by extracting the related fields from its configuration file. In this way, each CN subscribes to: a) a control topic used to send and receive commands to/from the broker, b) as many as needed data-related topics to publish data from its sensors, and c) a number of system topics to receive broadcasted information from the broker, e.g., time, network status, etc. The format of these three topics is the following (system topics start with a '$'):

/networkName/nodeID/country/districtState/cityTown/areaDescription/area/buidling/room/control [(str) command, (str) parameter1, (str) paramter2, (str) parameter3]*

/networkName/nodeID/country/districtState/cityTown/areaDescription/area/buidling/room/sensorDataName [(str) dataType, (str) dataRangeUnits, (str, int, float) data]

$/networkName/nodeID/country/districtState/cityTown/areaDescription/area/buidling/room/sysTopicName [(str) dataType, (str, int, float) data]

where in the control topic the supported command may contain up to three parameters, and in the sensor data topic the data type, units and range are provided for each piece of data. Using these topics' formats, data from CNs are tagged and characterized in terms of location and sensor type, while node

administration is very modular, since to add more sensors or change the location of a CN, only an update to the XML configuration file is required. By using a standardized format, collecting data from the topic name or the message content and storing information to a database on the server becomes easier, using powerful text processing languages like Python.

III. Client Nodes (CNs) Specification

Our network configuration on the edge consists of two kinds of client nodes: a) ESP8266 and b) Raspberry Pi, both equipped with software that has a rigid but easily expandable structure. The low-power and efficient Espressif ESP8266 WiFi SoC integrates a 32-bit single core RISC processor clocked at $80MHz$ (up to $160MHz$), combined with $160KB$ of RAM and External Flash support (up to $16MB$). Considered as open hardware, ESP8266 modules come in many variations from different vendors. For our implementation, the NodeMCU development kit was used with an ESP8266 (ESP-12E) WiFi module from Ai-thinker, providing an additional $4MB$ of Flash memory (ROM). The SoC has various interfaces including GPIO and ADC for digital and analog inputs and outputs. As measurement devices, a digital temperature/humidity DHT22 sensor was used, along with an analog/digital flame sensor. Raspberry Pi 3 Model B was our board of choice as a second edge node. This version of the popular board computer utilizes a 64-bit quad core $1.2GHz$ processor and $1GB$ of RAM for more demanding tasks. Combined with Raspberry Pi Camera V2 module, this node handles images and video streams for further analysis and object recognition. In terms of software, we adopted an OS based approach for all the nodes, so that we can easily handle encryption certification files and other file-based operations, and also be able to support, using interrupts/threads, at least the two concurrent operations of: i) MQTT message traffic and ii) sensor data acquisition.

The nodes are constantly connected and able to respond to commands in the control topic. We introduced three client functional modes (INITial, ACTIVE, BLOCKED) that define the connection status of the CNs. During INIT mode, each node is validated by the server according to the content of its XML file and the server sets its functional mode either to ACTIVE or BLOCKED. While in ACTIVE mode, a severity mode should also be set by each CN, related to the status of the acquired measurements. The default severity mode after boot is NORMAL, in which the CN acquires measurements every $60sec$. Every change in the severity mode is reported to the server using dedicated commands in the control topic. Changes of the severity mode are triggered by the CNs, according to data preprocessing functions applied on acquired data. This process allows critical conditions already identified on the field, to be indicated by the CNs with severity mode changes, while further data post-processing is done on server side. The sensor sampling interval is decreased to $40sec$ in WARNING mode and to $20sec$ in DANGER mode.

For the ESP boards we used the MicroPython OS [9]. This lite version of Python 3 has many CPython functions on a small package, well suited for uCs. MicroPython provides a Python programming (cross-)platform (all nodes running Python) with a dedicated file system. It features OtA file updates, making prototyping extremely versatile. File system offers multi-directory distributed coding with separated Python modules that increases flexibility and code maintenance. During code development we came across major RAM memory issues, solved with memory optimized programming and kernel recompilation to increase memory utilization. A lot of effort was required, using multiple interrupt service routines, to implement a pseudo-thread mechanism in Python (MicroPython does not support real threads) to handle concurrently the MQTT client and sensor data acquisition. Using also a data regression function, ESP generates a linear model of the temperature measurements, to handle severity mode transitions according to the temperature increase/descrease rate.

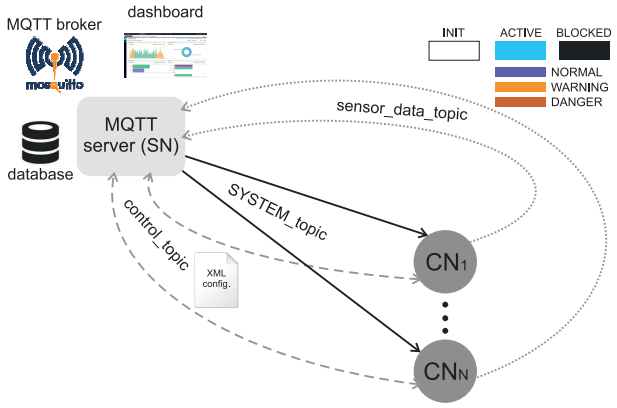

Fig. 2. MQTT main topics and available functional/severity modes for CNs

The Raspberry board hosts a Linux Debian 9 OS (Raspbian Stretch), equipped with Python packages to run our project's scripts. As a programming language we used Python 2.7 to take advantage of its cross-platform compatibility and get all of our CNs running similar Python code. The main purpose of the Raspberry node is to capture images on a timed interval and communicate directly with cloud services, like the Google Vision API, for basic image analysis. Analysis result are filtered and provided to the MQTT server for storage and further analysis. In this way, the Raspberry node provides data to the local MQTT network, but also uses powerful processing resources from the cloud. Work in progress includes a mechanism to be developed, so, if required, images can be transferred from the Raspberry board to the server for further analysis, using an SFTP connection (MQTT does not favor file transfers). Severity mode transitions are handled according to the type and number of identified objects on the images, while the mode transition rules can be changed on the fly (type of identified objects that trigger transitions) using commands on the control topic. Figure 2 summarizes the main MQTT topics and the functional/severity modes of the CNs.

Both nodes in our setup exchange TLS encrypted messages, using locally stored certificates. On the ESP8266 board we had to heavily optimize our code and recompile the MicroPython kernel in order to save some KBs of memory. To the best of our knowledge, the ESP8266 board (NodeMCU) is the smallest, very low-power, embedded CPU plus WiFi transceiver combined System on Chip (SoC), that, along with the MicroPython OS, can handle concurrently the MQTT encrypted messages, sensor data acquisition and support OtA software updates.

IV. Server Node (SN) Specification

In our network topology, we use a server PC that acts both as the MQTT network broker and a network client. A 64-bit $3.3GHz$ Intel $i3$ processor with $3GB$ of RAM can handle all the tasks of our small scale network. The server hosts an Ubuntu 16.04 OS and is used as: a) MQTT broker,

b) database (DB) and DB management, and c) Web Server for data visualization and cloud services communication.

As MQTT broker, we used the Mosquitto Broker with the MQTT protocol ver. 3.1. The broker forces TLS encryption to all the clients, as well as password protected connection. Encryption is achieved with server self-signed certificates. To store and manage our data efficiently, we chose the PostgreSQL database that offers the advantages of relational DBs (indices, triggers, structured data) and some non-SQL DB characteristics (JSON data type), without any major sacrifice on speed. For data visualization and other web services integration, our node acts also as a web server, providing open data in the widely acceptable JSON format.

The server's software is also implemented in Python 2.7 as a state machine, to realize the several states related to the CNs functional/severity modes implementation model. As previously described, a configuration file in XML format that follows a specific XML schema is used for every node, as node identity. There is a strict process of node validation upon connection to the broker that strengthens node integrity. The configuration files of all nodes are stored in our database along with live data, such as the functional and the severity mode of each node. In the event of an invalid configuration file, the node attempting the connection is blocked. The main task of the SN is to receive all incoming messages from clients, perform message analysis, handle the data and respond accordingly.

A multi-threaded approach was used in Python to develop the software on the server, demonstrating the potential of the language in parallel processing environments. In our implementation, we handle the message load by using two buffers and a function that checks for messages periodically. The main buffer holds all incoming messages by continuously checking the broker, while a parallel thread ensures constant availability by moving the captured messages to a second (temporary) buffer for further actions. The content of this buffer is filtered according to message topic and sensor data are stored into the DB using 1 thread per message. With this approach, our low-end server can easily process and store large bursts of sensor measurements from a single client, averaging on a satisfying 60 messages/sec without message loss. To simulate multi-client conditions, we developed some benchmarks that include simultaneous bursts from more clients. For our benchmarks, we used a sample sensor data message ($<$1KB) that we send to the server repeatedly, using QoS of level 2, to ensure that the message is delivered exactly once. As we can see from Table I, throughput tends to decline either with increased client count or burst size. The last case with 500 connected clients sending 1 msg. each is the most realistic one, as it simulates a common condition in alert-based networks. We simulated the 500 clients using separate threads running in parallel, achieving an average delay of $14msec$ between messages. It is very encouraging that our server manages to maintain its peak performance under these circumstances. Experiments with level 1 QoS messages showed a 40% increase in performance, due to the faster transmission process. Work in progress includes hardware upgrades and software optimizations. Python code, state machine diagrams (SN, CN), and the CN XML configuration file and schema can be found in our code repository [10].

TABLE I. SERVER MESSAGE THROUGHPUT FOR DIFFERENT # OF CNS AND SIZE OF MESSAGE BURSTS (MESSAGE SIZE $<$1KB)

# of CNs	# of messages (burst size) / CN	Stored messages / sec
1	1000	70
1	2000	65
1	10000	44
10	1000	34
500	1	71

V. CONCLUSIONS AND WORK IN PROGRESS

We presented a small scale MQTT-based sensor network implementation to i) explore different setups on the formalization of the protocol data structures. We experimented also ii) on the use of very low-power sensor nodes to handle concurrently encrypted message traffic and sensor data acquisition. More powerful nodes are used to iii) combine locally available data with powerful hardware cloud-based services. Moreover, our work includes iv) experimenting with database setups, using non-SQL and relational characteristics, to efficiently store sensor measurements and expose them as public open data. Working on these four different areas of a sensor network setup, we believe that we have established a standardized way on how to setup a network with the use of the MQTT protocol, based on alert-based sensors and on how to store, expose and combine acquired data with cloud based information and services. Work is in progress to increase the size of our network, using FPGA-based nodes to implement data preprocessing and reduce network traffic and server processing load. Also, work is in progress to finalize the server dashboard, connect it to popular cloud platforms to improve data visualization and alert triggering (emails, sms, etc.), implement big data post-processing algorithms using dedicated server hardware to discover hidden data patterns in the acquired measurements, and optimize the use of the network by small communities in terms of infrastructure and environmental monitoring.

REFERENCES

[1] Alexander Malaver, Nunzio Motta, Peter Corke, and Felipe Gonzalez, "Development and integration of a solar powered unmanned aerial vehicle and a wireless sensor network to monitor greenhouse gases," *Sensors*, vol. 15, no. 2, pp. 4072–4096, 2015.

[2] A. Niruntasukrat, C. Issariyapat, P. Pongpaibool, K. Meesublak, P. Aiumsupucgul, and A. Panya, "Authorization mechanism for MQTT-based Internet of Things," in *2016 IEEE International Conference on Communications Workshops (ICC)*, May 2016, pp. 290–295.

[3] N. Tantitharanukul, K. Osathanunkul, K. Hantrakul, P. Pramokchon, and P. Khoenkaw, "MQTT-topic naming criteria of open data for smart cities," in *2016 International Computer Science and Engineering Conference (ICSEC)*, Dec. 2016, pp. 1–6.

[4] R. Mahmoud, T. Yousuf, F. Aloul, and I. Zualkernan, "Internet of Things (IoT) security: Current status, challenges and prospective measures," in *2015 10th International Conference for Internet Technology and Secured Transactions (ICITST)*, Dec. 2015, pp. 336–341.

[5] R. Brzoza-Woch, M. Konieczny, P. Nawrocki, T. Szydlo, and K. Zielinski, "Embedded systems in the application of fog computing - Levee monitoring use case," in *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, May 2016, pp. 1–6.

[6] Gianluca Barbon, Michael Margolis, Filippo Palumbo, Franco Raimondi, and Nick Weldin, "Taking Arduino to the Internet of Things: The ASIP programming model," *Computer Comm.*, vol. 89-90, pp. 128 – 140, 2016, Internet of Things: Research challenges and Solutions.

[7] A. Azzar, D. Alessandrelli, S. Bocchino, M. Petracca, and P. Pagano, "PyoT, a macroprogramming framework for the Internet of Things," in *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, June 2014, pp. 96–103.

[8] "MQTT Version 3.1.1 - OASIS standard," 2015, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[9] "MicroPython - Python for uCs," 2017, https://micropython.org.

[10] "Git repository," 2018, https://github.com/evlog/mqtt_sensor_network.