Implementation Of MQTT-Geolocation To Wireless Sensor Networks

[1]Suchitra S C, [2]Mrs.Meghashree A C
[1]Mtech student, [2]Assistant professor
[1]VTU PG Centre Mysore,
[2]VTU PG Centre Mysore, Karnataka, India

Abstract: Wireless sensor networks are recently rapidly growing research area in all communication domains such as wireless communications and distributed networks. Here we focus on one specific type of sensor network called MQTT, it stands for Message Queue Transport Telemetry. MQTT is an open source publisher/subscriber protocol standard for Machine to Machine communication. It's well known features like low energy and bandwidth footprint makes it applicable for Internet of Things (IoT) messaging situations where power usage is a main restraint and in mobile devices such as cell phones, embedded devices or microcontrollers. The original version of MQTT is lacking the ability to transmit geolocation with its sensor values as part of the protocol itself. In today's generation in IoT platform, it has become more prominence to have geolocation as a part of the protocol. In the project we add geolocation to the MQTT protocol and offer a new revised version called MQTT-G. Hence explaining the protocol and showing how to embed geolocation successfully**.**

IndexTerms: MQTT, HTTP, IoT, Networks, Protocols, Geolocation.

I. Introduction

Today, due to the increased use of smart phones, wireless communication, the push notification services are now commonly used[1]. Push notifications service keeps the device online for every certain communication cycle, and the server pushes the messages to each client whenever its necessary. Compared to the polling method, push notification method was proved to be more efficient in battery and data consumption. **MQTT** is a client-Server publish/subscribe messaging transport protocol which mainly works well for push notifications. It is light weight, open, simple and designed to be easy to implement by both publishers and subscribers separately. These characteristics make it ideal for use in many situations, including complex environments such as for communication in Machine to Machine (**M2M**) and Internet of Things (**IoT**) contexts where a small code footprint is required and offers lower network bandwidth. As a well-known example, Facebook Messenger and Amazon web services which are based on MQTT [2]. Also, there is no restriction in messaging while using push notification services.

Some of the positive characteristics of MQTT are its light weight nature and binary footprint, which lead it to excel when transferring data over the wire. In comparison to well used transfer protocols like Hypertext Transfer Protocol (**HTTP**) and CoAP, it only has a minimal packet overhead. Another important aspect of MQTT is that it is extremely easy to implement on the subscriber/clients side. Its ease of implementation was one of the goals that was met when MQTT was invented. In this paper, we propose and develop a framework to improve the existing work of protocol MQTT. We establish our new protocol MQTT-g. It is a widely used and well known communication protocol for sharing data exchanged between IoT and wireless devices. MQTT is an extremely simple and lightweight messaging protocol in its original form, with a publish/subscribe architecture. It was designed to be straight forward to deploy, and capable of supporting thousands of clients with a single main server. In addition, MQTT provides reliability, flexibility and efficiency in adverse conditions, which makes it perfect for sensor network use in both wired and wireless scenarios. All these features make this protocol one of the most best used protocols for the communication between smart devices, wireless networks with a high number of applications based on it, increasing rapidly over time [5], [6]. One of the main implementation of MQTT and its deployment as the core protocol for Facebook Messenger [7].

II. LITERATURE SURVEY

This section describes the previous work done on comparison done between different communication protocols.
• High Power consumption by HTTP [8]:–. In the tests done by Hantrakul K et al., the HTTP protocol consumes 10 times higher power than MQTT protocol. It is witnessed that MQTT sends 10 times more messages than HTTP in 1 h of operation.
– Tests done by Upadhyay et al. [9] reveals, power consumption of MQTT Protocol is way more lower and 30% faster performance than CoAP protocol [10].
• High Protocol overheads in HTTP: – IoT applications requires large number of information exchange with tiny packets. Hence the payload is quite less, whereas the overhead caused to transfer the payload is quite high. It is shown that there is elimination of CONNECT/CONNACK flow for MQTT cases, that reduces the overhead and latency, when compared to HTTP, leading faster data transfer as well [10].
• High Bandwidth consumption in HTTP:– From the research done by Yokotani and Sasaki [11] on the comparison of bandwidth usage between HTTP and MQTT on 2 different cases, with payload and without payload (where only topics exist, that is used to decide on the MQTT broker, which client receive which message).
– For MQTT topics cases, where zero payload exists and only the transmission bytes exists reveals, HTTP consumes 300% higher bandwidth .
• Protocol efficiency is a function of the payload size in bytes for the three protocols in a LAN network are compared. The highest efficiency is achieved by CoAP, followed by WebSocket and MQTT QoS0. The reason for this is that CoAP uses UDP as the transport layer protocol. With respect to TCP, UDP has less header and lacks transport layer ACKs, which makes it very efficient. We also observe that CoAP, WebSocket and MQTT with QoS0 achieve very similar RTT, while MQTT with QoS1 has the highest RTT due to the presence of both transport and application layer ACKs. [12].

The rest of the paper is organized as follows. In Section III, we briefly explain the proposed scheme of the project. Lastly explaining results and future work in Section IV and finally with the conclusions in Section V.

III. PROPOSED SCHEME

In this paper, we propose a new framework to improve the protocol MQTT. We call it as a MQTTg. It is a widely used and well known communication protocol for sharing sensor data exchanged between IoT devices. MQTT is a simple and lightweight messaging protocol in its original form, with a publish/subscribe architecture. It was designed to deploy in an environment where it is capable of supporting thousands of clients with just a single server. MQTT provides high reliability and efficiency in adverse conditions, which makes it suitable for sensor network use in both wired and wireless scenarios. All these features make this protocol one of the best used protocols for the communication between wireless devices, smart devices, with a high number of applications based on it, increasing rapidly over time.

Previous work related to the topic is that, we had attempted to tackle MQTTg using the Mosquitto implementation [7] of the protocol. Mosquitto is an open source implementation of MQTT 3.1.1 which was prescribed recently in [6]. Mosquitto provides platform for compliant server and client implementations of the MQTT messaging protocol, however lacked in some code deployment needed to make MQTTg a success. However, our need of the MQTT protocol itself remains strong.

In this paper, we discard using Mosquitto and focus on a combination of the Arduino as one of the MQTTg client and [10] Blynk, the Android OS Application as another client. Arduino provides high performance and includes many in built libraries such as .NET library, GPS library for MQTT based communication. It provides the essentials required for MQTT client (subscriber) and a MQTT server (broker) in a C#/C++ environment. The arduino project has been created to provide scalable open source implementations of open and standard messaging protocols for all MQTT applications and Internet of Things (IoT). Arduino provides best platform to implement all communication protocols project to obtain any sensor values and GPS value. Using Blynk app we can configure the input values and output result using in built add -ons. We specifically focus MQTTg here in three parts, namely

- Arduino Desktop application C++ environment)
- Esp8266 module (Hardware set including GPS)
- An Android OS App- BLYNK app

A. BACKGROUND AND MOTIVATION

MQTT was invented by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom, now Cirrus Link) in 1999. Its initial use was to create a protocol for minimal battery loss and minimal bandwidth connecting oil pipelines over satellite connections [11]. It was then updated to include Wireless Sensor Networks in 2008 [12]. In [11], the following goals were specified:

- Simple to implement
- Provide a Quality of Service Data Delivery Lightweight and Bandwidth Efficient
- Data Agnostic
- Continuous Session Awareness

MQTT uses a client-Server publish/subscribe messaging pattern that enables a coupling between the information provider, known as the publisher, and consumers of information, called subscribers. This quality is achieved by introducing a message broker between the publishers and subscribers.

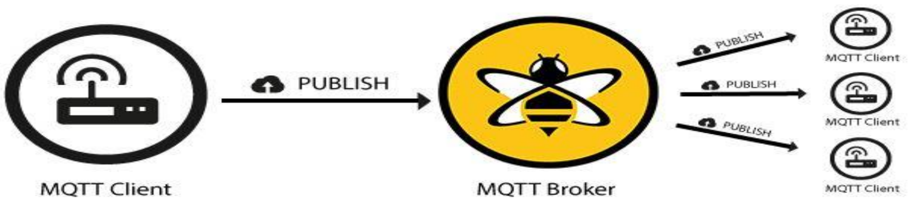

Fig. 1: Publish and Subscribe Model of MQTT

B. Related Work

Nowadays we are emerging with many interesting applications of MQTT recently. First, [12] compared the performance of MQTT and the Constrained Application Protocol (CoAP). CoAP is a specialized web transfer protocol for use with constrained nodes and constrained networks in IoT. The protocol is designed for M2M applications such as smart energy efficient applications and building automation.

We have also seen MQTT used to evaluate MQTT for use in Smart City Services [15]. The authors compare MQTT and HTTP in the context of bandwidth, power efficiency, and smart homes which is currently a hot topic with many large cities wanting to join the digital age and become Smart Cities.

C. Our Contributions

We modify both Arduino and esp8266 module and Blynk app by adding geolocation information into specific MQTT packets and adding gps module in microcontroller such that, for example, client location could be tracked by the broker, and clients can subscribe based on not only topics (sensor values) but also by geolocation. A list of all MQTT packets with details is described in

Table I. This can lead to the client's last known location having a comparison to a polygon geofence. One of the important features of GPS Tracking software using GPS Tracking devices is geofencing and its ability to help keep track of assets by providing location information in terms of latitude and longitude coordinates. Geofencing allows users of a Global Positioning System (GPS) Tracking Solution to draw zones (GeoFence) around places of importance, customer's sites and secure areas anywhere around the world.

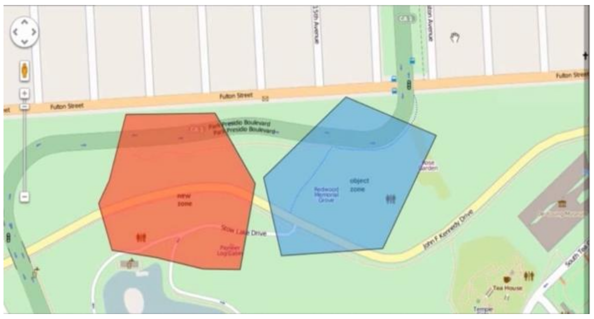

Fig. 2: Polygon Geofence

In MQTTg, by adding geolocation, information reaching subscribers can be filtered out by the broker to only fall within the subscribers geofence. An example of a geofence in shown in Figure 2. As a green IoT example, take a Smart City driving conditions situation. By prescribing a geofence where environment conditions like temperature, humidity, fire alarm and many variety of reasons (weather, construction, accident), can be subscribed through the topic like "temperature and humidity sensor" would receive updates based if there geolocation in real time were to intersect with a polygon geofence where weather conditions may be abnormal. Other subscribers would receive different messages based on their interest in the subscribed topic.

IV. RESULTS AND DISCUSSION

The intention of adding geolocation to the existing MQTT is to compensate unused binary bin data within the protocol itself and optionally embedding geolocation data between the header and payload packet of mqtt. The major change to the packets was the inclusion of the **Geolocation Flag** in the payload. The geo-flag is sent in packets between the clients and broker to notify the broker that a client is sending geolocation data in the packet. The packets that are used to send geolocation information are described in Table I derived from the original MQTT protocol implementation.

TABLE I.PACKETS USED TO SEND GEOLOCATION

Packet	**Details**
CONNECT	client request to connect to Server
PUBLISH	Publish message
PUBACK	Publish acknowledgment
PUBREC	Publish received (assured delivery part 1)
PUBREL	Publish received (assured delivery part 2)
PUBCOMP	Publish received (assured delivery part 3)
SUBSCRIBE	client subscribe request
UNSUBSCRIBE	Unsubscribe request
PINGREQ	PING request
DISCONNECT	client is disconnecting

Geolocation is not sent for **CONNACK, SUBACK, UN-SUBACK, PINGRESP** packets as they are only to transfer information between brokers to client, and thereby perceived unnecessary to contain geolocation information. For all packets mentioned in Table I, with the exception of **PUBLISH**, where the 3rd bit of the publish header is reserved (unused) in the original implementation in [13], so we can easily use it to indicate the presence of geolocation information in the frame packet.

Figures 3 and 4 explain how the location data is sent on the packet. We also use the struct of the code as shown by the struct mosquitto_location.

Listing 1: Struct for Geolocation Data

```
struct mqttGeog {
std::uint8_t version;
double latitude, longitude;
float elevation;}
```

The PUBLISH control packet of mqtt needs a different implementation than other protocol. Because the 3rd bit is already reserved for Quality of Service (**QOS**), and all other packets are also allocated for an existing use, we chose to implement a new version of control packet type: **PUBLISHg** (=0xF0) is used as the flag type for geolocation data when it is to be sent. There are 16 available command packet types within the MQTT standard and 0 through 14 are used.

We presume geolocation data as an optional attribute, as not all clients may wish to publish/provide any geolocation data. For example. the geolocation of manager of the forest fire crews in the aforementioned example does not need to be shared with the crews, it is irrelevant. In our approach, geolocation data is not included in the main packet payload, since not all packet types support a payload, thus rendering payloads not a viable option. Furthermore, we did not wish to require the broker to examine the payload of any packet, thus keeping our processing with less footprint.

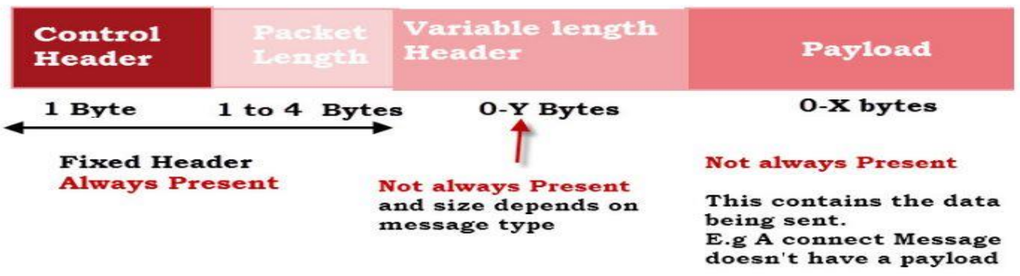

Fig 3: Original MQTT frame format

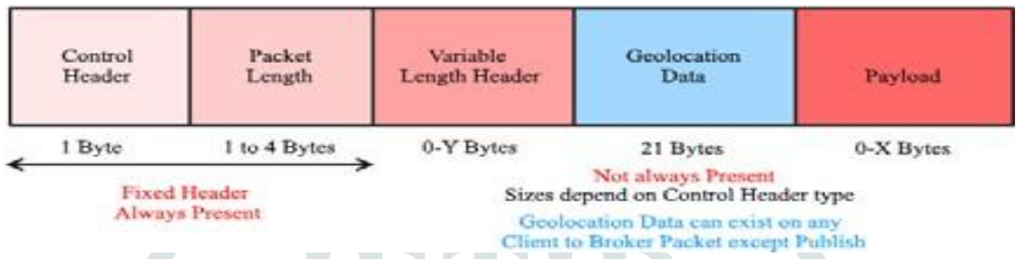

Fig 4: MQTT geolocation frame format

A. Handling of packets

Packets that are received without geolocation are handled via the original C++ code functions, and as such can be left unmodified. Packets that are received with geolocation are handled separately with struct functions but with a call to a **last known location** updating method, which stores the clients unique ID, longitude, latitude values and along with sensor values into a **Hashtable** object designed to be compared against the geofence if and only if they are a subscriber to be sent a PUBLISH. We have elected to attach geolocation data from all control packet types originating from the client to eliminate the need for specific packets carrying only geolocation data, and thus reducing overall network traffic.

B. Geofencing

Creating the geofence code was a critical part in the addition of geolocation to original MQTT. The geofence filtering is only called when a PUBLISH packet reaches the broker, as these packets are forwarded to subscribing clients.

The **float latitude = (gps.location.lat()); float longitude = (gps.location.lng());** returns a boolean value where the clients last known location with latitude and longitudes values. If the point is outside the polygon, it simply aborts forwarding the PUBLISH as the mqtt server is not connected. This condition is tested for many client so that other subscribers may receive packets of their own interest. Thus, we have used our own custom geometry of all client library originally implemented in [12] with features first discussed in [13]. The library is unmodified for the broker implementation, but it is reconfigured for several mobile clients.

Geofence data is presently submitted and cleared by a client to the broker using the topic subscription so that clients may individually submit geofences of their interest. The broker maintains each polygon data separately for each subscribing client. Polygons may be *static* in shape and location may be in *dynamic* but both move with the last known location of the client.

C. Android OS Application

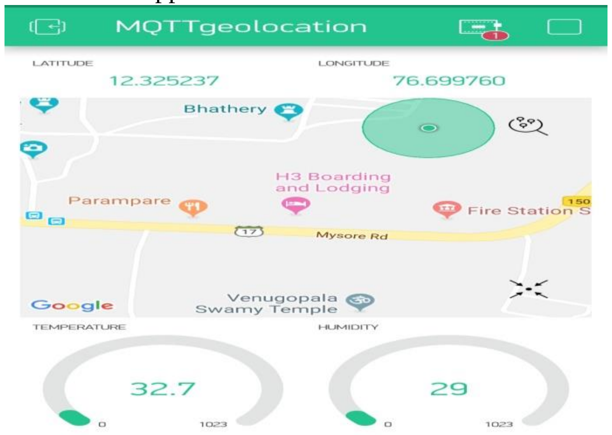

Fig. 5: Android OS BLYNK App

Figure 5 provides some snapshots of the current implementation of the Android OS Application for getting location information through BLYNK app. In Figure 5, a subscriber (client) can identify themselves on the network using authentication key. Pressing the Current Location button will give the broker your current location and access to your present location. By not pressing Current Location, the given client acts in its original MQTT form lacking geolocation information. The topic, say "MQTT geolocation", will subscribe the client to that topic for location updates and sensor values update. If there are more than one client ,an update is provided to the Topic by a publishing client, all other clients within a geofence bounded area of the publisher's creation will receive the message. We are still finalizing the all details of how to define geofences properly by the publishers and subscribers. A client can subscribe to as many topics as they choose based on their interest. In Figure 5, all subscribed topic messages are show here. Topics where geolocation are shared will be specific to a given geofence so only matching geolocation to a given geofence will be shown to a specific client. We expect to add separate sensor values and other network components for say a Publisher scenario versus a Subscriber scenario on the network.

V. CONCLUSION

MQTT is an open source standard for all types Machine to machine communication. Originally developed and designed by IBM, the main use of MQTT is for publish/subscribe protocol. In this paper we are introducing a new version of MQTT called MQTT-G, that include geolocation information to the existing protocol and offers a revised implementation, that can help aid in the vast use of MQTT. We also modernize the protocol to include a somewhat standard feature of most communication protocols in today's IoT technology. The advanced protocol we implement can be used to offer geolocation as part of the publish/subscribe infrastructure, thus aiding in the real time applications that it can be used for. Our implementations offer versions for all environments like C/C++ and java and also for a mobile Android client as well.

VI. ACKNOWLEDGMENT

This project is partly supported by the assistant professors, Digital Electronics and Communication System Department of Visvesvaraya Technological University, Post Graduation Center, Sathagalli Mysuru.

VII. REFERENCES

[1] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on. IEEE, 2014, pp. 1–6.

[2] S. Lee, H. Kim, D.-k. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in Information Networking (ICOIN), 2013 International Conference on. IEEE, 2013, pp. 714–717.

[3] Chuan-Fu Wu and Wen-Shyong Hsieh, "Digital watermarking using zerotree of DCT," IEEE Trans. Consum. Electron., vol. 46, no. 1, pp. 87– 94, Feb. 2000.

[4] Jyotsna Singh and Abhinav Dubey, "MPEG-2 video watermarking using quantization index modulation" 2011 IEEE 978-1-4244-7932-0/10.

[5] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications," IEEE Internet of Things Journal, vol. 4, no. 5,
1125–1142, 2017.

[6] R. A. Light, "Mosquitto: server and client implementation of the mqtt protocol," Journal of Open Source Software, vol. 2, no. 13, 2017.

[7] S. Lee, H. Kim, D.-k. Hong, and H. Ju, "Correlation analysis of mqtt loss and delay according to qos level," in Information Networking (ICOIN), 2013 International Conference on. IEEE, 2013, pp. 714–717.

[8] Hantrakul, K., Sitti, S., Tantitharanukul, N.: Parking lot guidance software based on MQTT protocol. In: 2017 International Conference on Digital Arts, Media and Technology (ICDAMT) (2017)

[9] Upadhyay, Y., Borole, A., Dileepan, D.: MQTT based secured home automation system. In:2016 IEEE Symposium on Colossal Data Analysis and Networking, CDAN 2016 (2016)

[10] Amaran, M.H., Noh, N.A.M., Rohmad, M.S., Hashim, H.: A comparison of lightweight communication protocols in robotic applications. Procedia Comput. Sci. 76, 400–405 (2015)

[11] Yokotani, T., Sasaki, Y.: Comparison with HTTP and MQTT on required network resources for IoT. In: International Conference on Control, Electronics, Renewable Energy, and Communications 2016, Conference Proceedings, CCEREC 2016 (2016).

[12] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, and C. K.-Y. Tan, "Performance evaluation of mqtt and coap via a common middleware," in Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on. IEEE, 2014, pp. 1–6.

[13] A. Stanford-Clark and U. Hunkeler, "Mq telemetry transport (mqtt)," http://mqtt. org. Accessed September, vol. 22, p. 2013, 1999.

[14] R. Bryce, T. Shaw, and G. Srivastava, "Mqtt-g: A publish/subscribe protocol with geolocation," in 2018 41st International Conference on Telecommunications and Signal Processing (TSP). IEEE, 2018, pp. 1–4.