This full text paper was peer-reviewed at the direction of IEEE Instrumentation and Measurement Society prior to the acceptance and publication.

# IEEE 1451-1-6: Providing Common Network Services over MQTT

Joshua Velez, Russell Trafford, Michael Pierce, Benjamin Thomson, Eric Jastrzebski, Brian Lau
Department of Electrical and Computer Engineering
Rowan University
Glassboro, NJ 08028
Email: velezj05@students.rowan.edu, traffo17@students.rowan.edu, piercem4@students.rowan.edu,
thomsonb7@students.rowan.edu, jastrzebe1@students.rowan.edu, rivera-lb0@students.rowan.edu

***Abstract*—IEEE 1451 has been around for almost 20 years and in that time it has seen many changes in the world of smart sensors. One of the most distinct paradigms to arise was the Internet-of-Things and with it, the popularity of light-weight and simple to implement communication protocols. One of these protocols in particular, MQ Telemetry Transport has become synonymous with large cloud service providers such as Amazon Web Services, IBM Watson, and Microsoft Azure, along with countless other services. While MQTT had be traditionally used in controlled networks within server centers, the simplicity of the protocol has caused it to be utilized on the open internet. Now being called the language of the IoT, it seems obvious that any standard that is aiming to bring a common network service layer to the IoT architecture should be able to utilize MQTT. This paper proposes potential methodologies to extend the Common Architectures and Network services found in the IEEE 1451 Family of Standard into applications which utilize MQTT.**

***Keywords*—MQTT, NCAP, TIM**

## I. Introduction

Since its conception in the 1990's, the IEEE 1451 family of standards was forward thinking in terms of how to facilitate the acquisition and dissemination of sensor/actuator data. The architecture proposed by the original standards committee allowed for systems, which traditionally would have issues with communicating with a large amount of transducers, a method in which each sensor and actuator could be addressed and serviced. By identifying the common functionality that would be required by distributed sensing systems, a common architecture came to fruition. By identifying 3 main entities and defining the methods of communication between them, a system which could be highly adaptive to the number and types of sensors could be implemented. One of the other major functionalities that came out of the initial founding of the standard was the idea of "Plug and Play" sensors where in the network of sensors could adapt and provision new devices being initialized or being taken away, without the need for human interaction. One might think that all of this was aimed at Internet accessible systems, but really the standard at the time proposed methods which were mainly focused on local area networks, mentioning briefly that with protocols such as HTTP, you could access the data from anywhere.

Along with HTTP, many other protocols were created to communicate between devices. These protocols include, but are not limited to, XMPP, SNMP, and MQ Telemetry Transport (MQTT). Each one of these protocols has their own unique features with communications however they all have their limits and downsides. XMPP is secure but also is heavy duty and bulky with its messages. SNMP is open but also requires detailed messages. MQTT is lightweight, but is not completely secure due to the minimal message size. XMPP and SNMP do have similarities though with one standing out more than the others, they are both protocols in the IEEE 1451 family of standards. They have both already been mapped out and have been conformed to their own standards in the family, but as stated before they are both bulky. Bulky meaning that their messages require large amounts of memory and bandwidth to be sent and received, which in result uses more power. This presents a problem as IEEE 1451 systems with memory and power constraints may not be able to support the large memory and bandwidth needed. This is where MQTT comes in as it is lightweight, meaning that it sends messages at minimal sizes and does not require much bandwidth or power. MQTT is not a protocol in the IEEE 1451 Family of Standards just yet, but it is in the midst of being proposed to allow the incorporation of small systems.

The addition of MQTT in the IEEE 1451 Family of standards will open up the doorway to smaller sized systems as well as systems with different functionalities due to its publish/subscribe protocol. With MQTT, low budget systems can be made due to MQTT's lightweight requirements. MQTT requires low power to function. This is due to its minimal message size as well as low required bandwidth. All of these features also contribute to small-scale memory requirements thus meaning cheap systems can be made with just the bare necessities and still completely function. The publish/subscribe messaging protocol opens the door to larger entity systems and easier event driven messages. Larger entity systems correlates to having more NCAPs (Network Capable Application Processor) and TIMs (Transducer Interface Module) present in the system. Being capable of subscribing and publishing to a topic allows for mass messages to be sent to large amounts of entities in a system to be rather simple. Event driven systems include messages that are sent when thresh holds are passed for sensors. If a temperature sensor is reading past a limit set, it can easily publish to every client subscribed to the

978-1-5386-2092-2/18/$31.00 ©2018 IEEE

topic. All of these reasons contribute to why MQTT is being proposed and why it would greatly benefit the IEEE 1451 Family of standards. Along with it being proposed, a working implementation has been designed to prove that MQTT is capable of adhering to the Family of Standards.

## II. Background

### A. MQTT

MQTT was invented in 1999 by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom). Their objective was to create a protocol for minimal battery loss and minimal bandwidth connecting oil pipelines over satellite connection, with a few specified goals. These goals included that the MQTT protocol be simple to implement, provide a Quality of Service Data Delivery, be lightweight and bandwidth efficient, data agnostic, and continuous session awareness. While the focus of MQTT has now shifted to Internet of Things uses, these goals are still the core of MQTT. The MQ in MQTT used to stand for Message Queue, however this is no longer the case as MQTT does not make use of a queue. A traditional message queue works by storing messages until they are consumed by a client and each queue must be named and be created explicitly with a separate command. Instead of this message queue model, MQTT makes use of another model, publish and subscribe.

MQTT uses a publish and subscribe communications model. In traditional client-server models, the client communicates directly with an endpoint. In the publish/subscribe model, there is a client who is sending a message to another topic. The sender is the publisher and the receiver(s) is the subscriber. The publisher and subscriber do not know the existence of one another. A broker, who is known by both the publisher and subscriber, filters all incoming messages from the publisher and distributes them accordingly to the subscribers. A simple diagram is shown below in Figure 1.

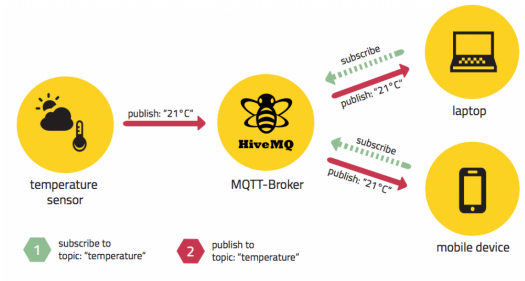


Fig. 1: MQTT Network

In this diagram, the temperature sensor is the publisher which published to the MQTT broker which then sends message to the subscribed clients.

The main aspect in publish/subscribe is the decoupling of publisher and receiver. The two can run independently, and do not need to know each other. They can run at the same time, and operations on either component are not stopped during publishing or receiving. Filtering of the messages allows for the subscriber to receive messages of their interest, while keeping the subscriber and publisher decoupled. In addition, publish/subscribe provides more scalability because the broker operations can be highly parallelized, and processed-event driven. A cluster of brokers can be used to distribute a load over individual servers with load balancers to use publish/subscribe on a larger scale.

As mentioned earlier, the broker can filter messages, so the subscriber only gets the messages it is interested in. There are three main ways to filter the messages, by topic, by content, or by type. The messages filtered by topic include a topic in the sent message. A topic is a general UTF-8 string with hierarchical structure that allow for filtering as can be seen in Fig. 2.


Fig. 2: UTF-8 String with topics

The UTF-8 strings separate each topic with the separation of a forward slash to create the hierarchy. The forward slashes are what the broker looks for to parse through the topics. To filter by content the broker parses the message based on a specific content filter-language. The downside of this is that the content of the message must be known. Lastly, to filter by type means to parse through the message by the type or class of the messages being sent. These filters need to be kept in mind when structuring the network because depending on the application, the data may need to be known beforehand, such as the topics for filtering, or the content itself. It is also important to remember that because this is a publish/subscribe pattern, if a message has no subscribers, that message will not be read. There are, however, ways to specify that a message has been successfully delivered from the client to the broker or vice versa, if there are subscribers.

A client is almost always a publisher or subscriber(s), both label an MQTT client that is only doing publishing or subscribing, but a MQTT client can be both a publisher and subscriber at the same time. A MQTT client is any device from a microcontroller to a server that has a MQTT library running and is connecting to a MQTT broker over any kind of network. The library is straight forward and is available for a variety of languages including Android, Arduino, C, C++, C#, Go, iOS, Java, JavaScript, .NET.

The broker is responsible for receiving all messages, filtering them, deciding what subscribers should get the message, and then sending them the message. The broker holds the sessions of all clients including subscriptions and missed messages. It also manages the authentication and authorization of clients. The broker is easy to integrate, which is important as the broker is what is exposed on the internet and handles many clients and messages that are then passed to downstream analyzing and processing systems. For this to work, the broker

must be easy to monitor, highly scalable, can integrate into back end systems, and fail resistant.

The connection of MQTT itself is based on TCP/IP and both the client and broker need to have a TCP/IP stack. The connection itself is always between one client and the broker, not two clients are connected directly. The connection is initiated through a client sending a CONNECT message to the broker. The broker responds with a CONNACK and a status code. Once the connection is established; the broker will keep it open if the client does not send a disconnect command or losses the connection. The broker will close the connection if the CONNECT message takes too long from opening a network socket to sending it to avoid malicious clients that can slow down the broker. The CONNACK messages only contains two data entries, a session present flag and a connect return code. The flag indicates whether the broker already has a persistent session of the client from a previous interaction, and the code indicates a response. Shown below in Figure 3 is a simple diagram depicting the connection communication.

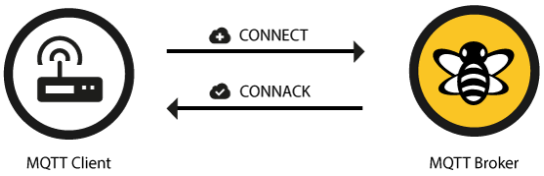


Fig. 3: Connection between client and broker

### B. IEEE 1451 Architecture

The IEEE 1451 family of standards outlines a way to create and support a Smart Transducer Network. To create this network the family of standards defines 3 entities, a client, a Network Capable Application Processor(NCAP), and a Transducer Interface Module(TIM). The family of standards also defines how each of these entities interact. The architecture and connection of these entities can be seen in Fig.4.

### C. Client

The Client is the highest level of the architecture. A Client can be referred to as the end user, which can be seen as a person using a mobile application, or even a website. The Client, whether it be through an application or website, communicates only to the NCAP(s) of the system. The Client sends a message to the NCAP(s) that will provide the specific functions wanted to be performed. This communication between the Client and NCAP happens via internet protocols such as XMPP, HTTP, SNMP, and possibly MQTT in the near future. The specific functions that the message from the Client includes depends completely on what the Client intends to accomplish. The message can include function calls to the type of data wanted, the amount of data wanted, and even the location of where the data is coming from. The type of data completely relies on the sensors and actuators that are present on the TIMs that are in the system. TIMs can be equipped with a variety of sensors including temperature, humidity, and motion sensors.

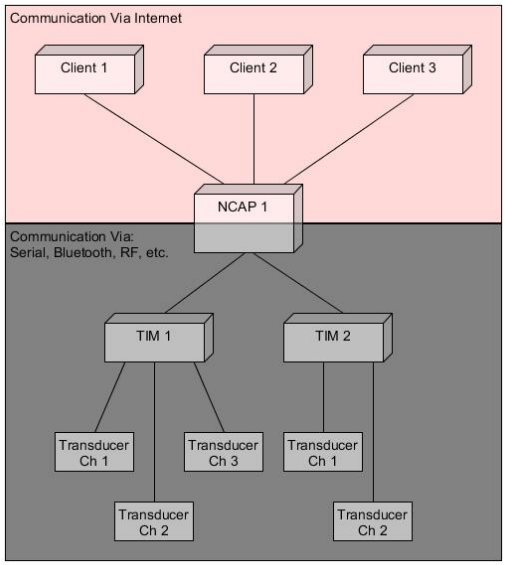


Fig. 4: Basic Architecture of a P21451 Network with multiple Clients talking to a single NCAP.

The location of data represents where the TIMs are stationed. If the system is located in a building then the location of the TIMs would be the specific room or floor that they are placed in. The amount of data can differ as the Client could be asking for one temperature at one given time, or could even be asking for the range of temperatures over a period of time.

### D. NCAP

The middle level of the architecture is the NCAP. The NCAP is the middle man, or ”broker,” of the system and manages the messages coming from the TIMs and Clients. The Clients will send the NCAP the message with its function calls in the form of a structurally defined message. The structure of the message is defined by the standard itself. The NCAP takes this message and parses it to find out exactly what the Client is asking for and then executes those determined functions and talks to the TIMs required to complete those functions. The TIMs then get the data and send a message back to the NCAP so that it can once again parse the message and make it readable to send back to the Client. The connection between the Client and NCAP is as stated before, an open internet protocol. The NCAP and TIM communication is open to the use of various methods. The NCAP could be wired to the TIM with the use of UART, SPI, I2C, but is most commonly wirelessly connected through Wi-fi, Bluetooth, or even ZigBee.

### E. TIM

The last entity of the architecture is the TIM. The TIM is what handles all of the transducers in the system. The TIM provides the software and the hardware to power and drive the transducers so they can operate properly and relay the correct data to the NCAP. There can be many TIMs in one

system and more than one can connect to an NCAP at one time. The transducers on the TIMs can range from various sensors including temperature, humidity, pressure, ultra sonic and even an accelerometer. Transducers do not only have to be sensors, they can be actuators. Actuators include LEDs, motors, servos, and even solenoids.

## III. Implementation

With MQTT not being a communications protocol among the IEEE 1451 Family of standard as of now, the proposal of it being a protocol in the family of standards must include proof that it is capable of functioning as an IEEE 1451 system. The proof comes with a working implementation, as well as proof that MQTT can support all of the IEEE 1451 entities. MQTT will be added into the family of standards as a Network Interface with the likes of XMPP, HTTP, and SNMP. A representation of where it will be added can be seen in the top right of Fig. 5.

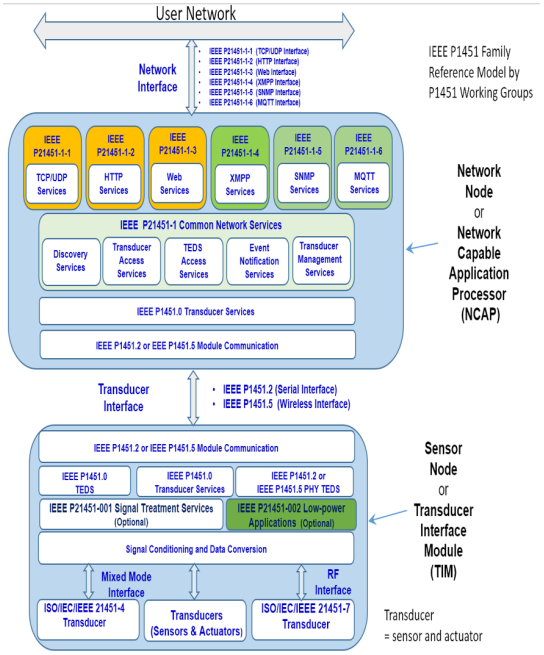


Fig. 5: Addition of MQTT to the IEEE 1451 Family of standards

### A. Integration with P1451

MQTT provides two entities, the Client and the Broker, and the IEEE 1451 Family of standards provides three entities, the Client, NCAP, and TIM. MQTT needs to conform itself to the standard even with the differences in the number of entities. The proposed solution to this is to set the MQTT broker in between the 1451 Client and 1451 NCAP and can be seen in Fig.6 with Amazon Web Services being the MQTT broker By conforming MQTT to the family as so it is forcing the 1451 Client and 1451 NCAP to both be MQTT clients and the TIM would remain the same and only be connected to the NCAP. This creates some issues that must be addressed. One


Fig. 6: Proposed implementation of MQTT system into IEEE 1451 Family of standards

issue is that the 1451 Client and 1451 NCAP both need to be able to subscribe and publish to a MQTT topic. As a 1451 Client it will be easy to subscribe to a new topic as a user will have direct contact with it, however a 1451 NCAP will need to be preset with the topics it is subscribed to so that it can always publish to the required topic. This is an issue as new topics might always be evolving in the system so a solution will need to be developed so that the NCAP will not have to be manually visited and edited, but rather have a command sent to it so it can subscribe to the new topic. Along with the issues comes some positives. With the 1451 NCAP and 1451 Client both being MQTT clients, theoretically an infinite number of 1451 Clients and NCAPs can be subscribed to the same topic. The number of subscribers to the topic completely depends on the size of the broker, but if the the broker is large enough, MQTT can support an abundance of subscribers thus allowing for mass systems to be produced.

### B. Development of MQTT Network

As MQTT is simply a protocol of communication, an integration with devices is necessary to have a functional network. In order to create a network, a Raspberry Pi was used as a server (broker). Using Putty(in a secure shell protocol), a repository package signing key was downloaded into an MQTT directory using the ”wget” command. Once the repository was available, another wget command was used to download the repository list. After updating the list, the mosquitto MQTT broker program file was installed onto a Jessie operated Raspberry Pi model 3. The MQTT broker installation allows the Pi to create a topic that MQTT clients can subscribe and publish to.

One method used to test the functionality of the Rasperry Pi’s broker capabilities was to use an application called ”MQTT Dash”. ”MQTT Dash” is an application designed for a smart phone, which can be found on the Google app store. This allows a smart phone to act as a client and publish/subscribe to topics created by the Raspberry Pi. This program was used to initially test if the topics created by the Raspberry Pi worked properly.

The MQTT client program file was installed onto the Pi, allowing the Pi to be used as either a broker or a client. The MQTT client was installed from the same repository as the broker. With multiple Pis, a network can be established with client Pis connected to a broker Pi. The MQTT client file allows the Pi, or any device that the MQTT client is

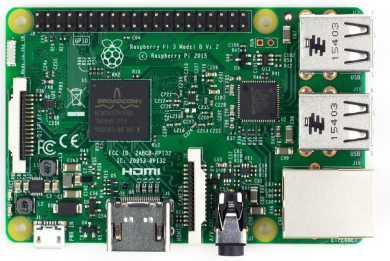

Fig. 7: Raspberry Pi 3

Fig. 8: ESP-32 Board

installed onto, to subscribe and publish to topics created by the broker. To create and connect these clients, the Mosquitto MQTT client can be installed onto ESP-32s. The ESP-32 is a small and inexpensive chip that is rather powerful for wireless communication, coming with both WiFi and Bluetooth communication functionality [8]. These features allow for fast and reliable communication to the Raspberry Pi MQTT broker, as well as allowing multiple client connections to a Raspberry Pi broker. While ESP-32s use Arduino, and Pi model 3s use Linux/Python, they can both communicate over MQTT without the issue of using of a programming language barrier.

As stated before, MQTT has a slightly different communication architecture than the communication protocols already integrated into the P1451 Family of Standards, since everything connected to an MQTT broker is a client. This difference was solved simply by defining MQTT Clients as NCAPs. By redefining what MQTT Clients are, the standard can be correctly applied to the MQTT communication protocol. Having set up a central broker using a Raspberry Pi, NCAPs were then designed using ESP-32 Development Boards. In order to connect the ESP-32 to the broker, it must first be connected to the local WiFi network. WiFi communication is required in order to have access to the broker. To establish a connection to the local WiFi network, several libraries must be called for the ESP-32 code. These libraries provide WiFi functionality for the ESP-32, allowing for a simple code to be written that will establish a connection. The necessary connection information must be inputted into the code, such as the SSID and password. By running the program, a connection from the ESP-32 to the local WiFi network will be established, providing Internet access to the ESP-32 which can now establish a connection to the MQTT broker.

The process of connecting to the broker is similar to connecting to a local WiFi network, however, the broker is private so more securities are involved. These few extra steps are similar to WiFi connection, but are still required for proper connection. The previously written WiFi connection code can be altered for MQTT broker communication. A new library must be added to the code, in order to have proper connection and communication to the broker. The library required, *PubSubClient*, allows for the ESP-32 to subscribe to an MQTT topic on the broker, as well as publish to said topic. In the altered WiFi connection code, broker communication is possible using the same format, by simply adding the broker securities. The MQTT server, MQTT port, MQTT user and MQTT password are the new inputs required for proper connection [8]. Running this new program establishes a secure connection to the broker, as the private information is only known by the user.

The ESP-32 has now been connected to the local WiFi network and to the MQTT broker, making it a fully functional NCAP with subscribe and publish capabilities. By creating topics on the Raspberry Pi broker, the ESP-32 can now subscribe and publish to them. Within the code that connects the ESP-32 to the broker,the topic information can be inputted, which directs where the ESP-32 should subscribe. While subscribed to the MQTT topic, a message can either be published to the topic or read from the topic, demonstrating fully functional NCAP to broker communication on an MQTT network.

### C. Future Development of MQTT Network

MQTT is already on its way to becoming a staple protocol for Internet of Things devices. These devices need low power options for communicating that are low bandwidth and easy to use. However, MQTT has other qualities that will solidify it as a standard protocol, such as not needing to resend application level logic with the use of built-in QoS levels. The MQTT protocol can also communicate multi-directional between the clients and the broker.

Now that MQTT is becoming used for IoT work so often, an updated version, ”MQTT version 5”, includes some performance updates to help further establish itself as the go-to IoT communication protocol. One of these improvements focuses on the number of devices that can connect to a broker, which will bring MQTT’s connectivity capabilities from tens of thousands of connected devices to hundreds of millions. Another feature that MQTT needs in the future is the expiration of messages, so that unused messages don’t build up in the broker.

Although MQTT is an effective communication protocol when it comes to small scale systems, security is a major area for future investigation. One of the great benefits for MQTT is the low bandwidth it requires to transmit data. The bandwidth is low because only the bare minimum of data is communicated, which offers little security. In order to be an effective protocol for Internet of Things applications, security must be addressed. A MQTT security method that still provides a low bandwidth is adding certificates to the publish function of the protocol.

A certificate provides more security as it acts as a key in order to read and write messages to an MQTT Topic. Instead of simply reading and writing the messages, the MQTT broker will ask for a certificate to be satisfied. A certificate is a string of bits that must satisfy the requirements of the broker[9]. More messages are being sent for this measure to be implemented, increasing bandwidth, but the redundancy prevents unwanted users accessing information on the broker. To create an MQTT network that uses certificates would allow for a fast, secure, and effective communication protocol for the standard.

## IV. Conclusion

MQTT is still in the process of being proposed and accepted into the IEEE 1451 Family of standards, but steps towards it being accepted are still being made. A working implementation has been created and proves that it can be functional and beneficial in the 1451 Family of Standards. There are many features and solutions that MQTT can provide to the family to expand its horizons and allow for new opportunities for systems of the family. MQTT provides a whole new messaging protocol in its publish/subscribe messaging protocol and allows for a whole other level of lightweight systems to be adapted. MQTT has its downsides in security, but solutions are being worked on and hope to be overcome in the near future. MQTT is being greatly adapted in the world of IoT today and the hope is extend that adaptation to the IEEE 1451 Family of standards.

## References


[1] "MQTT Essentials: Part 1 Introducing MQTT" Internet: https://www.hivemq.com/blog/mqtt-essentials-part-1-introducing-mqtt.

[2] K. Lee and R. Schneeman, "Distributed Measurement and Control Based on the IEEE 1451 Smart Transducer Interface Standards." *IEEE Transactions on instrumentation and Measurement*, vol. 49, no. 3, pp. 621-627, June 2000.

[3] "Overview of the Raspberry Pi Zero." Internet: https://www.element14.com/community/docs/DOC-79284/l/overview-of-the-raspberry-pi-zero, [June 10, 2016].

[4] "TI LaunchPad Development Ecosystem." Internet: http://www.ti.com/ww/en/launchpad/launchpad.html?DCMP=mcu-launchpad&HQS=launchpad, [June 10, 2016]

[5] A. Kumar and G. Hancke, "An Energy-Efficient Smart comfort Sensing System Based on the IEEE 1451 Standard for Green Buildings." *IEEE Sensors Journal*, vol. 14, no. 12, pp. 4245-4252, Dec. 2014.

[6] J. Schmalzel, F. Figueroa, J. Morris, S. Mandayam, and R. Polikar, "Design and Implementation of a Reliable Message Transmission System Based on MQTT Protocol in IoT" *IEEE Transactions on Instrumentation and Measurement*, vol. 54, no. 4, pp. 1612-1616, Aug. 2005.

[7] H. Hwang, J. Park, J. Shon, "An Architecture for Intelligent Systems Based on Smart Sensors" *Wireless Personal Communications*, vol. 91, no. 4, pp. 17651777, Jun. 2016.

[8] R. Kodali, S. Soratkal, "MQTT based home automation system using ESP8266" *Humanitarian Technology Conference (R10-HTC), 2016 IEEE Region 10*, Dec. 2016.

[9] M. Singh, M. Rajan, V. Shivraj, P. Balamuralidhar, "Secure MQTT for Internet of Things (IoT)" *2015 Fifth International Conference on Communication Systems and Network Technologies*, pp. 746 - 751, Apr, 2015.

[10] "Introduction to MQTT Security Mechanisms" Internet: http://www.steves-internet-guide.com/mqtt-security-mechanisms.

[11] "IoT messaging: The MQTT protocol is stepping up to the plate" Internet: http://internetofthingsagenda.techtarget.com/feature/IoT-messaging-The-MQTT-protocol-is-stepping-up-to-the-plate.