Green Communication Protocol with Geolocation

Gautam Srivastava*‡, Andrew Fisher*†, and Robert Bryce†

* Department of Mathematics and Computer Science, Brandon University, Brandon, Manitoba, Canada
† Heartland Software Solutions Inc., Ardmore, Alberta, Canada
‡ Research Center for Interneural Computing, China Medical University, Taichung, Taiwan, Republic of China

***Abstract*—Green communications is the practice of selecting energy efficient communications, networking technologies and products. This process is followed by minimizing resource use whenever possible in all branches of communications. In this day and age, green communication is vital to the footprint we leave on this planet as we move into a completely digital age. One such communication tool is Message Queue Transport Telemetry or MQTT which is an open source publisher/subscriber standard for M2M (Machine to Machine) communication. It is well known for its low energy and bandwidth footprint and thus makes it highly suitable for Green Internet of Things (IoT) messaging situations where power usage is at a premium or in mobile devices such as phones, embedded computers or microcontrollers. It is a perfect tool for the green communication age upon us and more specifically Green IoT. One problem however with the original MQTT protocol is that it is lacking the ability to broadcast geolocation. In today's age of IoT however, it has become more pertinent to have geolocation as part of the protocol. In this paper, we add geolocation to the MQTT protocol and offer a revised version, which we call MQTTg. We describe the protocol here and show where we are able to embed geolocation successfully. We also offer a early glimpse into an Android OS application we are developing for Open Source use.**

***Keywords*—Green communication, MQTT, IoT, networks, protocols, geolocation, broker**

I. Introduction

Today, the world is at a crossroads when it comes to many complex issues. Some include sustainable cities, pollution, safety, and most importantly energy consumption [1]. Internet of Things (IoT) is considered the core technology for building Smart Cities that will solve some of these problems. The core component of most IoT solutions is being connected to the internet. The use of more efficient sensor networks, adoption of cloud based services, and a lower energy footprint will all improve the quality of life in these Smart Cities. We can see IoT support the building of Smart Cities through:

- Lowering the consumption of water
- Medics with access to medical data in real time
- Real time energy consumption sensors
- smart street lights that can detect traffic

At the core of IoT is the idea that technology devices located in different places will communicate with each other and generate large amounts of data [2], [3]. Moreover, some of these devices and sensors maybe movable or in motion, fueling the need for geolocation to be part of the data collected or used. Consequently, there is a need to implement geolocation in the network protocols used. Moreover, taking into account what was mentioned earlier, these protocols need to be light in nature, where both bandwidth, energy consumption, and carbon footprint need to be taken into account when selecting the right protocol for the applications that require them. Thus the need for green communication and in turn green IoT is what Smart cities are in need of the most.

In this paper, we propose and develop a framework to improve the protocol MQTT. We call our new protocol MQTTg. It is a widely used and well known protocol for sharing data exchanged between IoT 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 just a single server. In addition, MQTT provides reliability 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 used protocols for the communication between smart devices, with a high number of applications based on it, increasing rapidly over time [4], [5]. A claim to fame for MQTT was its deployment as the core protocol for Facebook Messenger [6].

Previous to the work here, we had attempted to tackle MQTTg using the Mosquitto implementation [7], [8] of the protocol. Mosquitto is an open source implementation of **MQTT 3.1.1** which was prescribed recently in [5]. Mosquitto provides standard compliant server and client implementations of the MQTT messaging protocol, however lacked some in code deployment needed to make MQTTg a success. More specifcally, due to the way Mosquitto implemented MQTT, having it synchronize all the geolocation changes made to packets became infeasable. However, our praise of the MQTT protocol itself remains strong. We were initally drawn to MQTT due to its envisioned future in Green IoT and Green communications. To quote [5],

> MQTT uses a publish/subscribe model, has low network overhead and can be implemented on low power devices such micro-controllers that might be used in remote Internet of Things sensors. As such....is intended for use in all situations where there is a need for lightweight messaging, particularly on constrained devices with limited resources.

In our current project, we move away from Mosquitto and focus on a combination of the MQTTnet [9] for the main deployment of MQTTg and Paho [10] for the Android OS Application. MQTTnet is a high performance .NET library for MQTT based communication. It provides the essential MQTT client (subscriber) and a MQTT server (broker) in a C# environment. The Paho project has been created to provide scalable open source implementations of open and standard messaging protocols aimed at emerging MQTT applications for Machine-to-Machine (M2M) and Internet of Things (IoT). Paho reflects the inherent physical and cost constraints of device connectivity. Paho initially started with MQTT publish/subscribe client implementations for use on embedded platforms. Using Paho and porting MQTTg from MQTTnet to a Java based implementation will make it more accessible to multiple Operating Systems. We specifically focus MQTTg here in three parts, namely

- MQTTnet C# Desktop (Server and Client)
- Paho Java Desktop
- An Android OS App using Paho (Java)

The rest of the paper is organized as follows. In Section II, we briefly explain the basic concepts needed to understand the work presented here. We follow that with our main results in Section III. We end the paper first with a look at future work in Section IV and finally with the conclusions in Section V.

II. 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.

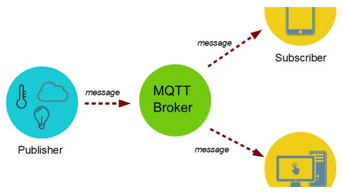


Fig. 1: Publish and Subscribe Model of MQTT

Compared with the traditional point-to-point protocols, the advantage of MQTT is that the publishing device does not need to know anything about the subscribing device, and vice versa. We can distinguish three MQTT essential concepts that will remain present throughout the paper.

1) **Topics**: The publishers are responsible for cataloguing the messages they send in topics. A topic defines the content of a message or a category in which the message can be classified. Topics are important because while in the point-to-point protocols messages are sent to a specific address, in a publish/subscribe protocol, messages are distributed based on the selected topics by the subscriber. By subscribing to a particular topic, the subscriber will receive all messages sent with that topic by any publisher.
2) **Client**: MQTT clients connect to a broker to exchange messages. They must subscribe to topics and can publish information to other entities connected to the same broker by providing a topic.
3) **Broker**: MQTT brokers are servers acting as intermediaries for the messages. MQTT protocol messages' format consists of three parts: a fixed header, a variable header; and a payload, all shown in Figure 2;

MQTT is one of the most used protocols in the world. We summarize many of those uses in Table I.

TABLE I: Use Cases for MQTT

Brokers	**Clients**	**Smart Applications**
SurgeMQ	hbmqtt	FHEM
hrotti	rumqtt	pimatic
VerneMQ	Paho	Home Assistant
Moquette	mqtt_cpp	aqara-mqtt
HiveMQ	M2Mqtt	cul2mqtt
Azure	MQTT Rx	HA4IoT
Moquitto	CocaoMQTT	Homegear
MQTTnet	emqttc	Domoticz

A. *Related Work*

We have seen some very interesting applications of MQTT recently. First, [13] 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 and building automation. In [14], the authors investigated the use of **OAuth** in MQTT. OAuth is an open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications.

We have also seen MQTT used to evaluate MQTT for use in Smart City Services [15]. The authors compare MQTT and **CUPUS** in the context of smart city application scenarios, which is currently a hot topic with many large cities wanting to join the digital age and become Smart Cities. Furthermore, it has been used in the development of an environmental monitoring system [16]. MQTT has also been used to support research less directly as part of a scheme for remote control of an experiment [17].

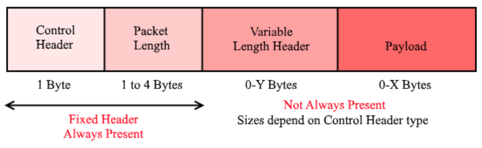


Fig. 2: Original MQTT Packet

B. Our Contributions

We modify both MQTTnet and Paho by adding geolocation information into specific MQTT packets such that, for example, client location could be tracked by the broker, and clients can subscribe based on not only topics but also by geolocation. A list of all MQTT packets is given in Table II. 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. 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.

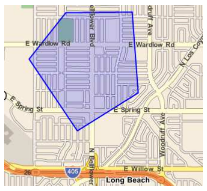


Fig. 3: Polygon Geofence

In MQTTg, by adding geolocation, information reaching subscribers can be filtered out by the broker to only fall within the subscribers geofence. We can see an example of a geofence in Figure 3. As a green IoT example, take a Smart City driving conditions situation. By prescribing a geofence where driving conditions may not be adequate for a variety of reasons (weather, construction, accident), specific subscribers on a Smart City topic like "driving conditions" would receive updates based if there geolocation in real time were to intersect with a polygon geofence where driving conditions may be abnormal. Other subscribers would receive different messages based on their driving routes in the city. The applications for this are plenty and include:

- Field team coordination
- Search and rescue improvements
- Advertising notifications to customers within specific ranges
- Emergency notifications, such as inclement weather or road closures.

III. Results

The basis of adding geolocation to MQTT is to leverage unused binary bin data within the protocol definition itself and optionally embedding geolocation data between the header and payload, as shown in Figure 4. We show the 21 bytes of Geolocation data as indicated in Figure 4 in Listing 1.

Listing 1: Struct for Geolocation Data

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

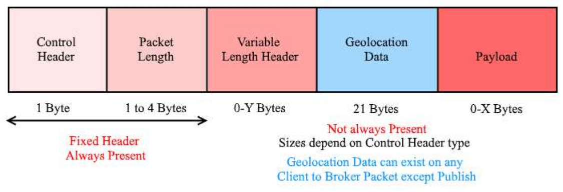
The major change to the packets themselves was the inclusion of the **Geolocation Flag**. The flag is sent in packets between the client (subscriber) to broker to notify the broker that a client is sending geolocation data in the packet. The packets that are used to send geolocation information are given in Table II derived from the original protocol implementation. In Listing 2, we see the updated C# code for MQTTnet packet deserializer for the Publish/PublishG packet. The **isGeog** Boolean passed is based on the packet type identified by the calling method. Based on this geolocation flag we treat the Publish/PublishG packets differently.

TABLE II: Types of MQTT Packets used for Geolocation

Packet	**Description**
CONNECT	client request to connect to Server
PUBLISH	Publish message
PUBACK	Publish acknowledgement
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

Listing 2: C# Code from the MQTTnet Packet De-Serializer

```
private static MqttBasePacket DeserializePublish
    (MqttPacketReader reader, MqttPacketHeader
    mqttPacketHeader, bool isGeog)
        {
            var fixedHeader = new ByteReader(
                mqttPacketHeader.FixedHeader);
            var retain = fixedHeader.Read();
            var qualityOfServiceLevel = (
                MqttQualityOfServiceLevel)
                fixedHeader.Read(2);
            var dup = fixedHeader.Read();
```



Control Header
Packet Length
Variable Length Header
Geolocation Data
Payload
1 Byte
1 to 4 Bytes
0-Y Bytes
21 Bytes
0-X Bytes
Fixed Header Always Present
Not always Present
Sizes depend on Control Header type
Geolocation Data can exist on any Client to Broker Packet except Publish

Fig. 4: MQTT Geolocation Packet

```

            var topic = reader.ReadStringWithLengthPrefix();

            ushort packetIdentifier = 0;
            if (qualityOfServiceLevel > MqttQualityOfServiceLevel.AtMostOnce)
            {
                packetIdentifier = reader.ReadUInt16();
            }

            MqttGeog GeoLocation = null;
            if (isGeog)
            {
                GeoLocation = new MqttGeog();
                GeoLocation.version = reader.ReadByte();
                GeoLocation.latitude = reader.ReadDouble();
                GeoLocation.longitude = reader.ReadDouble();
                GeoLocation.elevation = reader.ReadSingle();
            }

            var packet = new MqttPublishPacket
            {
                Retain = retain,
                QualityOfServiceLevel = qualityOfServiceLevel,
                Dup = dup,
                Topic = topic,
                Payload = reader.ReadRemainingData(),
                PacketIdentifier = packetIdentifier
            };

            packet.GeoLocation = GeoLocation;

            return packet;
        }
```

From the Paho MQTTg implementation, Listing 3 gives the updated Java implementation for de-serializing MQTT packets which they call **WireMessage**. For a Publish packet, the Java client is normally setup to determine the topic when creating a new **MqttPublish** object. For a PublishG packet, it is setup to have the topic before the geolocation data. So, the code is modified to do as such if one is received. The Java client also expects to be able to read the geolocation data in big endian the way that their **getDouble** and **getFloat** methods are setup. The geolocation data is, however, encoded in little endian so the bytes need to be reversed to get the correct output for this data. To be consistent, we are adhering to the IEEE floating point representations everywhere.

Listing 3: Poha Java Code Packet De-Serializer

```java
private static MqttWireMessage createWireMessage(InputStream inputStream) throws MqttException {
 try {
  CountingInputStream counter = new CountingInputStream(inputStream);
  DataInputStream in = new DataInputStream(counter);
  int first = in.readUnsignedByte();
  byte type = (byte) (first >> 4);
  byte info = (byte) (first &= 0x0f);

  long remLen = readMBI(in).getValue();
  long totalToRead = counter.getCounter() + remLen;

  MqttWireMessage result;

  MqttGeog geoLoc = null;
  String topic = null;

  if (type == MqttWireMessage.MESSAGE_TYPE_PUBLISHG) {
   geoLoc = new MqttGeog();
   //The C# implementation reads the topic before the GeoLocation

   int len = in.readUnsignedShort();

   byte[] encodedString = new byte[len];
   in.readFully(encodedString);

   topic = new String(encodedString, "UTF-8");

   geoLoc.version = in.readByte();

   int i = 1;
   byte[] lat = new byte[8];
   i = 0;
   while(i < 8) {
    lat[7 - i] = in.readByte();
    i++;
   }
   geoLoc.latitude = ByteBuffer.wrap(lat).getDouble();

   byte[] lon = new byte[8];
   i = 0;
   while(i < 8) {
    lon[7 - i] = in.readByte();
    i++;
   }
   geoLoc.longitude = ByteBuffer.wrap(lon).getDouble();

 byte[] elev = new byte[4];
 i = 0;
 while(i < 4) {
  elev[3 - i] = in.readByte();
  i++;
 }
 geoLoc.elevation = ByteBuffer.wrap(elev).
     getFloat();
}

long remainder = totalToRead - counter.
    getCounter();
byte[] data = new byte[0];

// The remaining bytes must be the payload...
if (remainder > 0) {
 data = new byte[(int) remainder];
 in.readFully(data, 0, data.length);
}
```

Geolocation is not sent for **CONNACK, SUBACK, UNSUBACK, PINGRESP** packets as they are only for information passed from broker to client, and thereby deemed unnecesary to contain geolocation information. For all packets mentioned in Table I, with the exception of **PUBLISH**, the 3rd bit of the fixed header is unused (reserved) in the original implementation in [11], so we can easily use it to indicate the presence of geolocation information. Figures 2 shown earlier and 4 explain where the location data is on the packet.

The PUBLISH control packet needs a different implementation. Because the 3rd bit is already allocated for Quality of Service (**QOS**), and all other packets are also reserved for an existing use, we chose to implement a new 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 deem geolocation data as an optional attribute, as not all clients may wish to publish their geolocation data. In our approach, geolocation data is not included in the packet payload, since not all packet types support a payload, thus rendering payloads not a viable option, especially for Green IoT. Furthermore, we did not wish to require the broker to examine the payload of any packet, thus keeping our processing footprint low.

A. Handling of packets

Packets that are received without geolocation are handled via the original MQTTnet and Paho functions respectively, and as such can be left unmodified. Packets that are received with geolocation are handled similarly but with a call to a **last known location** updating method, which stores the client's unique ID and the location data 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 packet types originating from the client to eliminate the need for specific packets carrying only geolocation data, and thus reducing overall network traffic as well.

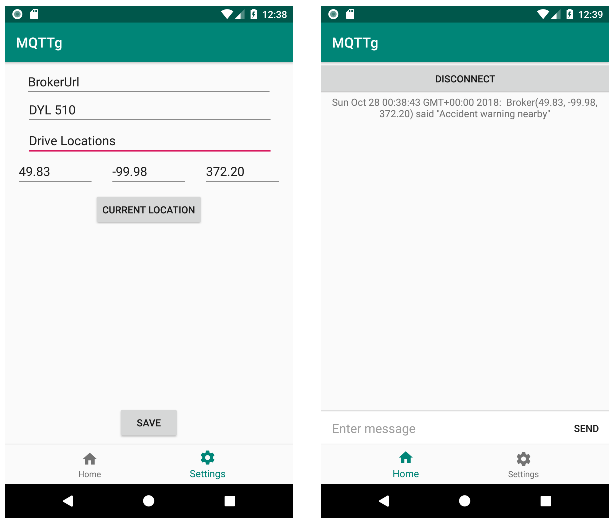
B. Geofencing

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

The **mosquitto_check_polygon** mentioned in [7] is a crucial routine that is currently being re-tested for the C# and Java code respectively. It returns a boolean value indicating whether the client's last known location is within the polygon. If the point is outside the polygon, it simply aborts forwarding the PUBLISH as the client has indicated it is not interested in the message. This condition is tested for each client so that other subscribers may receive packets of interest. Thus, we have used our own custom geometry library originally implemented in [18] with features first discussed in [19]. The library is unmodified for the broker implementation, but it is reconfigured for mobile clients.

Geofence data is presently submitted and cleared by a client to the broker using the **$SYS** MQTT topic convention so that clients may individually submit geofences of interest. The broker maintains polygon data for each subscribing client. Polygons may be *static* in shape and location or *dynamic* and move with the last known location of the client. We are currently navigating over a few options to provide configuration data from the client to the broker in a manner consistent with retrieving status information from brokers, we have aptly named this **$SYSg**.

C. Android OS Application



BrokerUrl
DYL 510
Drive Locations
49.83 -99.98 372.20

Sun Oct 28 00:38:43 GMT+00:00 2018: Broker(49.83, -99.98, 372.20) said "Accident warning nearby"

Enter message

(a) OS Subscriber ID Page (b) Subscriber Feed Page

Fig. 5: Android OS App

Figure 5 provides some snapshots of the current implementation of the Android OS Application for MQTTg. In Figure 5a, a subscriber (client) can identify themselves on the network. Pressing the **Current Location** button will give the broker your current location and access to your geolocation. By not pressing **Current Location**, the given client acts in original MQTT form lacking geolocation. The topic, say "Driving Conditions", will subscribe the client to that topic for future updates, which will show in Figure 5b. If 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 details of how to define geofences properly by the publishers. A client can subscribe to as many topics as they choose. In Figure 5b, all subscribed topic messages will show here. Topics where geolocation are shared will be specific to a given geofence so only matching geolocation to a given geofence will show. We expect to add separate layouts for say a Publisher scenario versus a Subscriber scenario on the network.

IV. Future Work

We are still finishing the final testing of the application of MQTTg for the Android OS. Once finished, the implementation will pull geolocation directly from the Android OS and allow publishing clients to quickly and easily create and destroy polygon geofences on the go. Applications of the Android client are plentiful but have some key uses in green IoT, natural disaster containment and safety in this age of mobile devices. We can also see some direct applications for visually impaired individuals trying to navigate smart cities [20]. There is also room to make the Android OS app more visually appealing.

We have also yet to deal with the Quality of Service (QoS) level and how it will relate to the MQTTg implementation. The QoS level is an agreement between the sender of a message and the receiver of a message that defines the guarantee of delivery for a specific message. There are 3 QoS levels in MQTT. QoS is a key feature of the MQTT protocol. QoS gives the client the power to choose a level of service that matches its network reliability and application logic. Therefore, there still needs to be some connection between QoS and MQTTg. For example, only allowing geolocation packets to be shared when QoS is level 2 or 3, but not for level 1, as a possible outcome. There is room and viability to use the 21 bytes of information as shown in Figure 4 to help manage the QoS levels as they relate to geolocation.

V. Conclusion

MQTT is an open source standard for M2M communication. Originally designed by IBM, the main use for MQTT is as a publisher/subscriber protocol. In previous works, MQTT has shown to be very viable in green communications and more specifically in green IoT. In this paper, we have introduced a new version, called MQTTg, that adds geolocation information to the protocol and offers a revised implementation, that can help aid in the breadth of uses for MQTT in Smart cities and energy efficient applications. We feel MQTTg modernizes the protocol to include a somewhat standard feature of most protocols in today's IoT age. 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 both C# and Java environments and adds a mobile Android OS application as well.

References

[1] M. Maksimovic, "Greening the future: green internet of things (g-iot) as a key technological enabler of sustainable development," in *Internet of Things and Big Data Analytics Toward Next-Generation Intelligence*. Springer, 2018, pp. 283–313.

[2] A. D. Dwivedi, G. Srivastava, S. Dhar, and R. Singh, "A decentralized privacy-preserving healthcare blockchain for iot," *Sensors*, vol. 19, no. 2, p. 326, 2019. [Online]. Available: https://doi.org/10.3390/s19020326

[3] A. D. Dwivedi, P. Morawiecki, and G. Srivastava, "Differential cryptanalysis of round-reduced speck suitable for internet of things devices," *IEEE Access*, vol. 7, pp. 16 476–16 486, 2019.

[4] 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, pp. 1125–1142, 2017.

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

[6] 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.

[7] 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.

[8] R. Bryce and G. Srivastava, "The addition of geolocation to sensor networks," in *ICSOFT*. SciTePress, 2018, pp. 796–802.

[9] "Mqttnet," https://github.com/chkr1011/MQTTnet, accessed: 2018-10-01.

[10] "The eclipse paho project," https://www.eclipse.org/paho/, accessed: 2018-10-01.

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

[12] U. Hunkeler, H. L. Truong, and A. Stanford-Clark, "Mqtt-s—a publish/subscribe protocol for wireless sensor networks," in *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 2008, pp. 791–798.

[13] 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.

[14] P. Fremantle, B. Aziz, J. Kopecký, and P. Scott, "Federated identity and access management for the internet of things," in *Secure Internet of Things (SIoT), 2014 International Workshop on*. IEEE, 2014, pp. 10–17.

[15] A. Antonić, M. Marjanović, P. Skočir, and I. P. Žarko, "Comparison of the cupus middleware and mqtt protocol for smart city services," in *Telecommunications (ConTEL), 2015 13th international conference on*. IEEE, 2015, pp. 1–8.

[16] P. Bellavista, C. Giannelli, and R. Zamagna, "The pervasive environment sensing and sharing solution," *Sustainability*, vol. 9, no. 4, p. 585, 2017.

[17] M. Schulz, F. Chen, and L. Payne, "Real-time animation of equipment in a remote laboratory," in *Remote Engineering and Virtual Instrumentation (REV), 2014 11th International Conference on*. IEEE, 2014, pp. 172–176.

[18] C. Tymstra, R. Bryce, B. Wotton, S. Taylor, O. Armitage *et al.*, "Development and structure of prometheus: the canadian wildland fire growth simulation model," *Natural Resources Canada, Canadian Forest Service, Northern Forestry Centre, Information Report NOR-X-417.(Edmonton, AB)*, 2010.

[19] C. Bose, R. Bryce, and G. Dueck, "Untangling the prometheus nightmare," in *Proc. 18th IMACS World Congress MODSIM09 and International Congress on Modelling and Simulation, Cairns, Australia*, 2009, pp. 13–17.

[20] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 2347–2376, 2015.