ACIT'2017
The International Arab Conference on Information Technology
Yassmine Hammamet, Tunisia
December 22-24, 2017

Real Time Hardware System for Synchronizing IoT Device State using the MQTT Protocol

Adnan Shaout and Brennan Crispin

The Electrical and Computer Engineering Department
The University of Michigan -Dearborn

***Abstract*—with the growing number of IoT devices, there is an increasing need for the ability to synchronize state and data across multiple IoT devices. As many IoT devices will be operating in constrained environments with unstable network connections, understanding the tradeoffs of various protocols is critical. The primary objective of this paper is to design and implement an embedded system to connect to a M2M broker, use the cloud server to communicate with other embedded systems, and be configurable from a cloud-based web service. In this paper we explore previous research on machine to machine (M2M) protocols such as AMQP and MQTT, and demonstrate an MQTT based system for synchronizing IoT device state across multiple client nodes. The main goal of the system is for state changes to be registered and distributed throughout the system in under 1 second; and initial registration of a new node should occur in under 30 seconds.**

***Index Terms*—IoT, Machine to Machine, Embedded, MQTT, Cloud Computing, Synchronizing**

I. Introduction

THE Internet of Things (IoT) increasingly covers a wide array of applications and use cases, and is a large current field of study and innovation. One of the major issues for IoT devices is how to efficiently communicate and synchronize between each other, a process referred as Machine to Machine (M2M) communication. Since operating remotely and in low-power devices is critical to the success of IoT devices, nodes must often be able to operate in constrained environments with unreliable network access. Several protocols have been explored for Machine to Machine communication of IoT devices, including Message Queueing Telemetry Transport (MQTT) [1], the Advanced Messaging Queuing Protocol (AMQP) [2], Constrained Application Protocol (CoAP) [3], Data Distribution Service (DDS) [4], Websockets [5], the Extensible Messaging and Presence Protocol (XMPP) [6], and HTTP based protocols [7].

The objective of this paper is to design and implement an IoT system that can detect local state changes and synchronize those state changes with other remote nodes as shown in figure 1. The primary objective of this system will be for an embedded system to connect to a M2M broker, use the cloud server to communicate with other embedded systems, and be configurable from a cloud-based web service. The main goal of the system is for state changes to be registered and distributed throughout the system in under 1 second; and initial registration of a new node should occur in under 30 seconds.

Additionally, no state change data should be lost.For the purposes of this researchin this paper the MQTT protocol was chosen for appearing to best meet the requirements set forward.

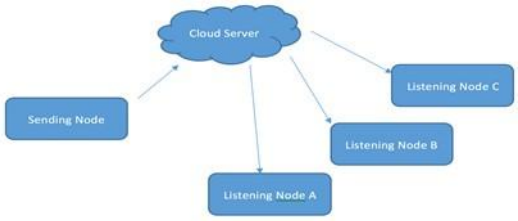
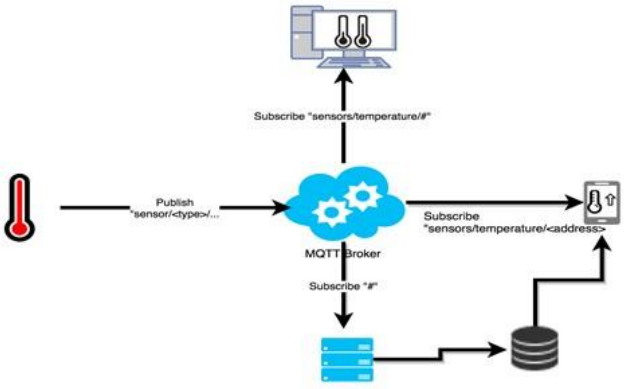


Fig 1. Overview of System Flow – Sending Node sends an update which is received by all nodes registered as listeners.

MQTT is an open-source TCP based protocol based on publish-subscription architecture. In publish-subscribe ("pub-sub"), clients connect to a central broker, and can "subscribe" to interested topics. When a client "publishes" to that topic, any client nodes that have subscribed will receive the published message. Being a TCP based protocol, MQTT has relatively high overhead, but also a high guaranteed quality of service (QoS), and also supports one-to-one and one-to-many messages. MQTT has numerous open source libraries and a robust support community, making it relatively easy to use for applications. A sample MQTT network is shown in Figure 2 [8].



This paper is further organized as follows: Section 2, Prior research on common M2M protocols is summarized and used to provide a basis for using MQTT in this paper. In Section 3, the implementation of the proposed design is discussed, both in terms of design and requirements testing. In Section 4,testing results and how the system did meet the requirements set forward will be discussed; and Section 5, conclusions and future work will be presented.

II. Related Work

In order to determine the best protocol for this project, research on the relative performance of several M2M communication protocols was examined and the results compared to the requirements for the project. A number of studies, [21][22][23][24] have examined M2M protocols for networked devices in the qualitative sense, however, quantitative studies were required to determine which protocol would best meet the requirements for this project.

Yokotani et al [9] compared HTTP and MQTT protocol network requirements in a variety of network conditions, finding that MQTT is a more efficient protocol for connecting IOT devices than HTTP. This is largely due to the additional data overhead involved in HTTP compared to the MQTT protocol.In [10], Daud et al. compared performance for CoAP and HTPP protocols, finding that CoAP has a substantially lighter memory and processing footprint, but that HTTP has substantial security improvements due to build in TLS support. The authors recommended that CoAP be implemented with DTLS to improve communication security. In [12] HTTP and AMQP are compared in the case of RESTful web services; running several tests over a variety of client applications. After comparing the average number of messages sent and received, the authors conclude that AMQP allows a greater number of messages to be supported.

Since many IoT devices will be required to perform in unstable network environments, Luzuriaga et al [11] compared the performance of AMQP and MQTT protocols in poor quality and unstable network environments. They found that AMQP is more reliable and stable, but that MQTT, being the lighter-weight service, is more appropriate for constrained edge nodes.N.De Caro et al. [19] compared CoAP and MQTT bandwidth usage and delay time for varying packet loss constraints. They found that MQTT, being TCP based, had high delivery percentages at high packet loss, but longer delays, while CoAP, being UDP based, lost substantial numbers of packets at a lower delay time. Similarly, Thangval et al [20] ran further comparisons of MQTT and CoAP, finding similar results in data loss and latency.

Due to the frequency of constrained networks in IoT applications, Chen et al [13] compared several protocols under constrained wireless environments: MQTT, CoAP, DDS, and XMPP. Using the example of a medical device transmitting data to a care provider, they examined a number of factors such as latency and packet loss under progressively constrained environments. They found that both MQTT and DDS have zero packet loss in high latency environments, but that DDS is superior in terms of latency – but substantially higher bandwidth requirements. CoAP and XMPP both experienced substantial packet losses and higher bandwidth consumption.

Fig 2. Sample MQTT system. Several clients subscribe for updates from a temperature sensor

From the research that has already been conducted, it was determined that the MQTT protocol would best meet the requirements set out previously for latency, memory and power, publish/subscribe architecture. MQTT has successfully been used to implement a wide variety of data collection services [14][15][16][17][18], although it has not yet been used to synchronize device state across multiple IoT nodes.

III. Implementation

This section discusses the hardware setup and software design and implementation of the system.

A. Hardware Setup

The hardware configuration is shown in Figure 3.

The MQTT client node consists of:

- Arduino Duo Revision 3
- Arduino Ethernet Shield
- LED and button input for Arduino

The MQTT broker and server was hosted on a:

- MacBook Pro 2014
- Linksys network router used for connection between client node and server

Fig 3. Arduino Uno and Ethernet Controller

The LED and button input and output are directly attached to the Arduino‟s Pin 12 and Pin 2, respectably.

B. Software Implementation

For the complete system, three separate subsystems were required to be designed and coded: the embedded client node, the MQTT broker, and the control server.

For the MQTT broker, we took advantage of a widely used MQTT library, Mosquitto MQTT [21], provided by the Eclipse foundation. The broker is run on the MacBook server and was run with TCP and SSL ports open. The client node was implemented using Arduino C, and an open source library, mqttPubSubClient [25], was used to implement connecting the device to the MQTT broker. Finally, the control server was implemented in Angular [26], a Javascript framework, using the built-in Javascript MQTT library to connect to the broker.

C. Client Design and Implementation

The client node was implemented using a foreground-background architecture. During the background loop, the software checks the Ethernet module for any newly received packets. The system parses the packet and takes one of three possible actions depending on the message topicshould any packets be available from the MQTT broker. Figure 4 shows the main background loop. The following are the message possible topics:

1. State

The client node reads the state data and synchronizes its state to that state provided. The client will maintain this state until an interrupt overrides the state with new state information or a new, updated state arrives from the posting node.

2. Subscribe/Unsubscribe

The client node reads the subscription address from the message and subscribes to state updates for the given posting node.

3. Query

The query message comes from the server on initialization, and asks that all nodes update their register information on the server and their state information. The node, upon receiving a query, will re-register with the server and post its current state.

On button press, the controller receives an interrupt to read from the button pin. The client reads the button state, updates its internal and LED state to match, and then publishes its state for any subscribed nodes to synchronize with as shown in Figure 5. However, because the interrupt and the main loop both potentially need to use the Ethernet controller, access to the controller is protected by a semaphore to control sharing the resource(the Ethernet controller). If the background loop is currently publishing, the foreground interrupt will hold until that process has been completed.

D. Server Implementation

The server was implemented as a simple front-end interface to the overarching client nodes and message broker. State and data do not persist from instance to instance, so the server client first collects information on all active nodes by

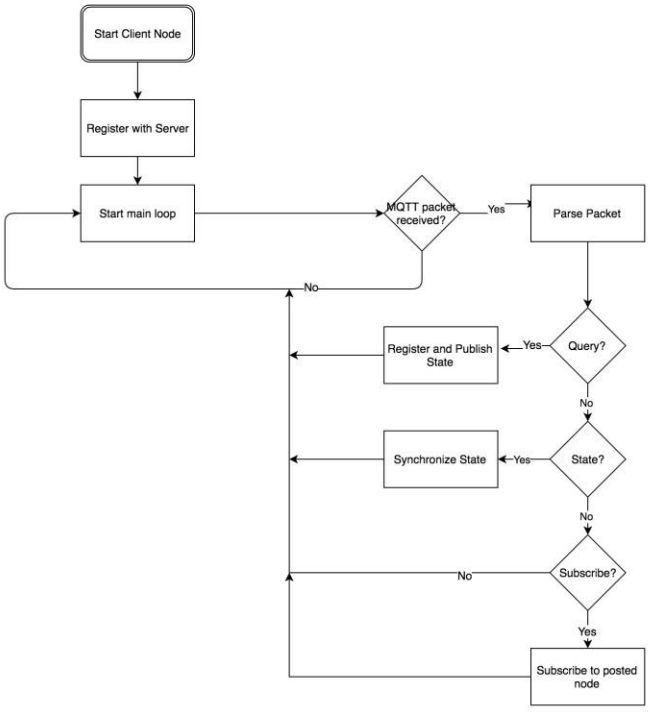


Fig 4. Overview of main background loop

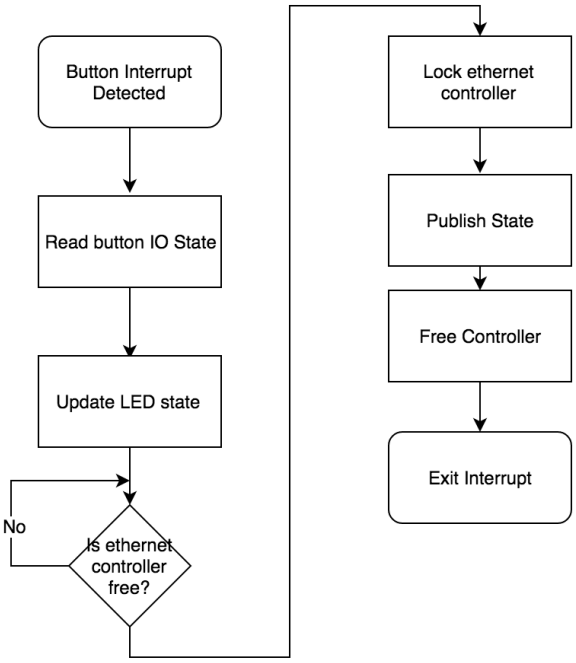


Fig 5. Button interrupt program flow

publishing a „query“ request and then collecting the „register“ posts from each client. When a new client node becomes active, it registers with the server and is added to the control list as shown in Figure 6.

From the registered client nodes, the server creates a list of available nodes, with the option to change any given node“s

state as well as to open a node‟s control page and select which other nodes that node is paired with. Paired nodes will listen for that other nodes state and update their internal state to match on change.

When a user selects an action for a given node, an MQTT publish action occurs that sends a message with the topic <ACTION>/<NODE ID> and the message for the node to follow to the MQTT broker, which then routes the message to the listening node.

Additionally, when the server receives a register message from a node, it subscribes to any state notifications from that node. When a state update is received, the server updates its display to reflect the new state of each node (Figure 6).

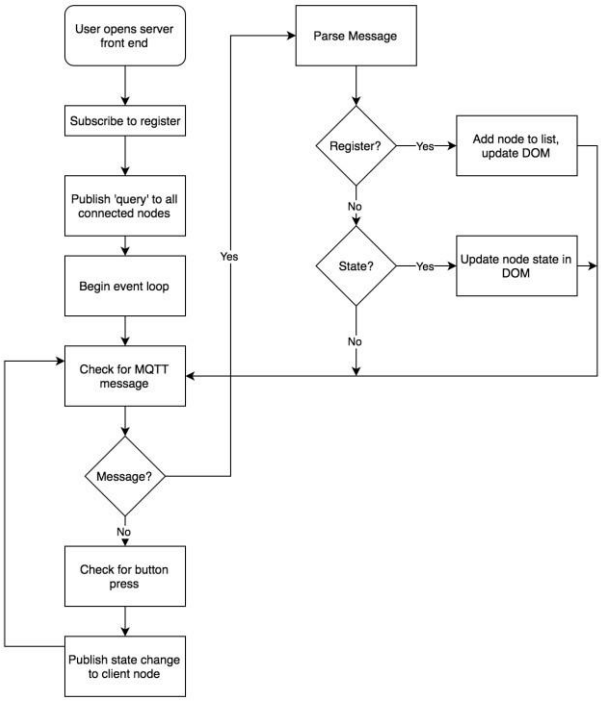


Fig 6. Server controller flow diagram

E. Experimental Setup

An important part of this project was ensuring that the system met certain time bounds. In order to test that these bounds were met, additional tests of the system were implemented.

Since many such IoT devices would need to operate in a constrained environment, NetEM[27] was used to test the system in varying levels of packet loss and latency.

In order to measure latency time from sending to receiving, the client node was modified to subscribe to its own updates, time them from publish to reception, and output the results to an attached serial port. In this manner, a total roundtrip time could be measured from interrupt detection to reception of the state packet. By measuring the response time across varying network and system conditions, a map of system performance in constrained environments would be created.

During latency tests, packet losses were additionally measured. Any state change that featured a posted time of greater than 5 seconds was qualified as a „lost message‟ and counted in the data.Finally, the Arduino IDE provides a useful tracking of CPU utilization during operation. During the above tests, average CPU utilization was tracked. Additionally, a Python script was written to spam the system with state updates, simulating a „high use‟ operation and CPU utilization was further measured.

IV. Results

Three metrics were used to determine whether the design and system meet the requirements established for the project: latency time under various network constraints, lost messages under packet loss, and CPU utilization under expected and extreme loads.

A. Message Latency

Using the NetEM package to vary the loss of different percentages of packets on the network, the resultant latency of packets was measured and is shown in Figure 7. The figure shows the number of packets verses the latency. Since MQTT is a TCP protocol, additional packets are expected to be sent as packet loss increases, leading to an increased latency time. With a requirement that the total routing time for any message be less than 1 second, we can reasonably expect MQTT to be appropriate for network conditions in which less that 15% of the packets are lost. However, at higher loss rates the response time will be greater than 1 second, which is unacceptable for the project requirements in this paper.

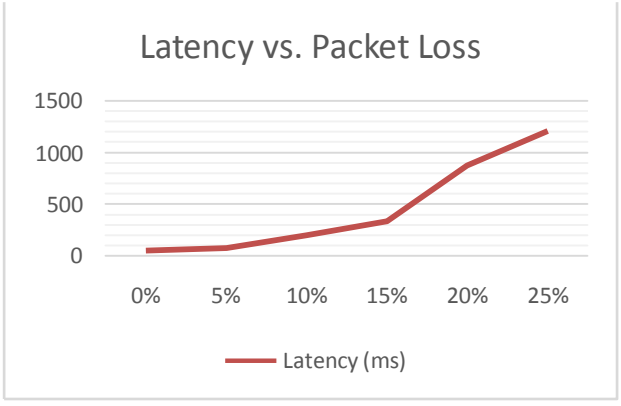


Fig 7. Experienced latency vs. packet loss in network

B. Lost Messages

Again, using the NetEM package, the loss of messages during varying packet loss scenarios was measured. The results are summarized in Figure 8. MQTT, being a TCP based library, experienced zero losses in total messages. Note that zero losses has been experienced because MQTT is a TCP

based algorithm.

Fig 8. Total lost messages vs. packet loss in network

C. CPU Utilization

CPU utilization under various load conditions is shown in Figure 9. In general, the CPU utilization of the Arduino board remained under 50%, even for high load conditions, indicating that MQTT is a suitably lightweight protocol for low power IoT applications and even lower power devices will support its implementation.

Fig 9. CPU Utilization under varying load conditions

CPU Utilization for Various Use Cases	
Use Case	CPU Utilization
0% Data Loss, normal press	26%
25% Data Loss, normal press	28%
0% Data Loss, Spam Press	35%
25% Data Loss, Spam Press	41%
Spammed Notifications	39%

V. Conclusions

There are a few conclusions that can be drawn from this project in this paper. First, given the requirements imposed on the design of the system, MQTT is an appropriate protocol for connecting low power devices in constrained environments.It provides a simple, flexible architecture that allows for the easy synchronization of device states with minimum overhead, and a selection of open source libraries make its implementation comparatively simple.

In terms of performance, analysis of our design has shown that MQTT meets system response time requirements for mildly unstable networks, but will begin to fail as greater packet losses occur. However, MQTT also successfully delivered all packets in constrained environments, which may be more important depending on the exact requirements of a system. Furthermore, MQTT was relatively efficient, seldom using much of the CPU during any operations, indicating that it would be appropriate for even lower power devices. These tradeoffs will need to be considered for anyone attempting to design a practical IoT system with synchronized device states.

Other protocols may meet the requirements for this project in other ways – for example DDS has been shown to have lower latency with still high delivery, and AMQP has a greater flexibility in payload contents. Additionally, protocols that support a secure connection will be important to implement. Further studies exploring more complicated state synchronization and how these protocols meet the requirements for state synchronization are therefore recommended.

References

[1] MQTT protocol specification. (http://mqtt.org)as of July 23, 2017.

[2] AMQP protocol specification. (http://amqp.org)as of July 23, 2017.

[3] CoAP protocol specification. (http://coap.technology)as of July 23, 2017.

[4] DDS protocol specification. (http://www.omg.org/spec/DDS)as of July 23, 2017.

[5] Ian Fette; Alexey Melnikov (December 2011). "WebSocket URIs". *RFC 6455 The WebSocket Protocol*. IETF. sec. 3. RFC 6455.

[6] XMPP protocol specification. (http://xmpp.org)as of July 23, 2017.

[7] HTTP protocol specification. (https://www.w3.org/Protocols/rfc2616/rfc2616.html)as of July 23, 2017.

[8] Monitoring Your Devices with MQTT - Packt Publishing. https://www.packtpub.com/sites/default/files/downloads/1942OS_Chapter_9.pdf - accessed July 2017.

[9] Yokotani, Tetsuya, and Yuya Sasaki. "Comparison with HTTP and MQTT on required network resources for IoT." *Control, Electronics, Renewable Energy and Communications (ICCEREC), 2016 International Conference on*. IEEE, 2016.

[10] Daud M.A., Suhaili W.S.H. (2017) Internet of Things (IoT) with CoAP and HTTP Protocol: A Study on Which Protocol Suits IoT in Terms of Performance. In: Phon-Amnuaisuk S., Au TW., Omar S. (eds) *Computational Intelligence in Information Systems. CIIS 2016. Advances in Intelligent Systems and Computing, vol 532. Springer, Cham.*

[11] Luzuriaga, Jorge E., et al. "A comparative evaluation of AMQP and MQTT protocols over unstable and mobile networks." *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*. IEEE, 2015.

[12] Fernandes, J.L. and Lopes, I.C. and Rodrigues, J.J.P.C. and Ullah, S., Performance evaluation of RESTful web services and AMQP protocol, IEEE ICUFN, pp. 810–815, 2013.

[13] Chen, Yuang, and Thomas Kunz. "Performance evaluation of IoT protocols under a constrained wireless access network." In *Selected Topics in Mobile & Wireless Networking (MoWNeT), 2016 International Conference on*, pp. 1-7. IEEE, 2016.

[14] Kang, Do-Hun, Min-Sung Park, Hyoung-Sub Kim, Da-young Kim, Sang-Hui Kim, Hyeon-Ju Son, and Sang-Gon Lee. "Room Temperature Control and Fire Alarm/Suppression IoT Service Using MQTT on AWS." In *Platform Technology and Service (PlatCon), 2017 International Conference on*, pp. 1-5. IEEE, 2017.

[15] N. D. Caro, W. Colitti, K. Steenhaut, G. Mangino, and G. Reali, "Comparison of two lightweight protocols for smartphone-based sensing", 2013 IEEE 20th Symposium on Communications and Vehicular Technology in Benelux (SCVT), Namur, Belgium, 21Nov -2 Dec 2013.

[16] Dhar, Prarna, and Poonam Gupta. "Intelligent parking Cloud services based on IoT using MQTT protocol." In *Automatic Control and Dynamic Optimization Techniques (ICACDOT), International Conference on*, pp. 30-34. IEEE, 2016.

[17] Yi, Ding, Fan Binwen, Kong Xiaoming, and Ma Qianqian. "Design and implementation of mobile health monitoring system based on MQTT protocol." In *Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC), 2016 IEEE*, pp. 1679-1682. IEEE, 2016.

[18] Hantrakul, Kittikorn, Part Pramokchon, PaweenKhoenkaw, NasiTantitharanukul, and KitisakOsathanunkul. "Automatic faucet with changeable flow based on MQTT protocol." In *Computer Science and Engineering Conference (ICSEC), 2016 International*, pp. 1-5. IEEE, 2016.

[19] D. Thangavel, X. Ma, A. Valera, H. Tan, and C. K. TAN, "Performance evaluation of MQTT and CoAP via a common middleware" 2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, pp.1-6, 21-24 April 2014.

[20] R. Sutaria, and R. Govindachari, "Making sense of interoperability: Protocols and Standardization initiatives in IOT." ComNet 2013, Hyderabad, India, 8-9 Nov 2013.

[21] Z. Sheng, S. Yang, Y. Yu, A. V. Vasilakos, J. A. McCann and K.K. Leung. "A Survey on the IETF Protocol Suite For The Internet of Things: Standards, Challenges, and Opportunities", IEEE Wireless Communications, pp. 91-98, December 2013.

[22] L. Atzori, A. Iera, G. Morabito, "The Internet of Things: A survey", Computer Networks 54 (2010) pp. 2787-2805.

[23] A. Asensio, A. Marco, R. Blasco, and R. Casas, "Protocol and Architecture to Bring Things into Internet of Things", International Journal of Distributed Sensor Networks, Volume 2014, April 2014.

[24] Mosquitto Project (http://mosquitto.org)as of July 23, 2017.

[25] O"Leary, Nick. Arduino Client for MQTT. (http://pubsubclient.knolleary.net/)as of July 23, 2017.

[26] AngularJS Framework. (https://angularjs.org)as of July 23, 2017.

[27] NetEM: Software suite provides Network Emulation functionality (http://www.linuxfoundation.org/collaborate/workgroups/networking/netem).