Technology Research Paper

April 27, 2021

IoT Protocol Evaluation

CU Boulder Senior Capstone Project

Table of Contents

By Jacob Hans, Chris Kardaras, and Nick Wentzel

Introduction

Delivery of data is at the foundation of the Internet of Things (IoT). Open source data protocol solutions have been at the forefront of the market due to the wide acceptance and support from many vendors. This makes it easier for companies to collaborate and be interoperable with other solutions and applications[1]. As the industry has developed over the past few years, certain protocols have become clear leaders in performing this task within the industry. This paper will delve into 4 of the most prevailing protocols that make up a good portion of the IoT landscape. Accounts are geared toward both the tech-savvy, and the non technical layman. This paper will give in-depth accounts of Message Queuing Telemetry Transport (MQTT), Advanced Message Queueing Protocol (AMQP), Constrained Application Protocol (CoAP), and Lightweight Machine to Machine (LwM2M), through both analysis of testing performed by the team and explanations of how the protocols operate. The goal of this paper is not only for a recommendation to be made on what protocol BI Incorporated should use in their technology, but to provide enough information on the protocols, so that engineers within the company can pick up any necessary information regarding their implementation.

The Protocols In-Depth

MQTT

Background

MQTT was originally developed in 1999 and became an OASIS standard in 2013. Since MQTT was developed earlier than most other IoT protocols, it has become a forerunner and quite a popular choice in IoT solutions. MQTT is intended to be used in lightweight and constrained device applications that are designed to run over low bandwidth networks[2]. This protocol is structured to operate for sending streams of data over a TCP connection which allows for reliable network transmission. MQTT was also designed to be "data agnostic" which is beneficial in a wide range of applications. Standard communication is done over port 1883 and the encrypted TLS port for MQTT is 8883.

The Communication Model

MQTT is structured as a publisher/subscriber (pub/sub) model where publishers send data to a central server and subscribers are sent data from the server. In this model, a single node can be both a subscriber and a publisher on one-to-many topics. The overall model centralizes around a server or "broker" which handles the facilitation of messages over different topics from publishers to subscribers. The broker does not handle the storing of messages that are sent from publishers, but rather sends messages to all available subscribers that are subscribed to the topic.

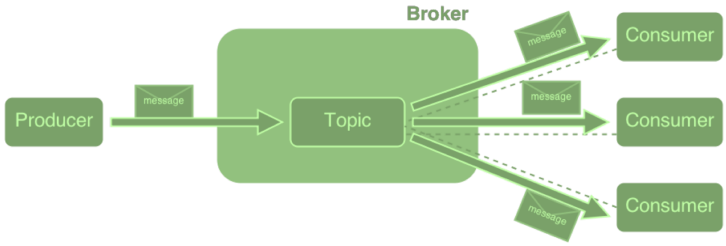


Figure 1.1 Flow of messages through broker from Producers to Consumers (Publishers to Subscribers)[3]

[1] IoT Use case study https://iot.eclipse.org/community/resources/case-studies/iot-on-railway-systems-db/

[2] Get to Know MQTT https://thenewstack.io/mqtt-protocol-iot/#:~:text=MQTT%20was%20created%20way%20back,pipelines%20over%20unreliable%2C%20satellite%20networks

[3] Image from: https://thenewstack.io/mqtt-protocol-iot/

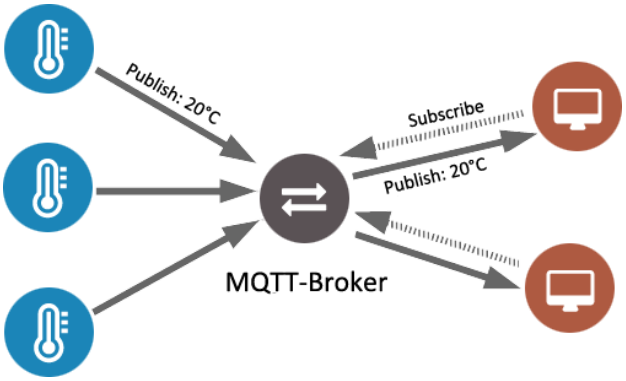


Figure 1.2 MQTT Message flows through topics[4]

As mentioned above, communication flow in the MQTT protocol is based on topics. Topics specify how data should flow through the broker to the current subscribers and publishers.

MQTT Topics

Topics specify channels in which data flows from publishers to subscribers. Topic names are "case sensitive, use UTF-8 strings, and must have at least 1 character to be valid". A topic is created when a publisher publishes data over the topic or a subscriber subscribes to a topic. Topics are temporary and only exist if a subscriber is subscribed to it except if the topic is using the retained feature discussed in a later section. Subscribers can subscribe to multiple topics and there are wildcards that can be used to subscribe to many/all topics at once. Topics are also '/' delimited so a valid topic could be '/sensors/sensor1'. There are also wildcards: '#' (multi-level) and '+' (single-level). So a subscriber could subscribe to all topics by subscribing to '#'. Similarly, a subscriber can subscribe to the topic '/sensors/#' which would cover all topics and levels stemming from the '/sensors/' node, which might include something like '/sensors/sensor1' and '/sensors/sensor3/temperature'. Whereas the '+' wildcard is single-level which means that subscribing to '/sensors/+/temperature' would subscribe to '/sensors/sensor1/temperature' and '/sensors/sensor2/temperature' but would not subscribe to '/sensors/sensor1/humidity'. Topics and wildcards are very useful in an "administrator" program that monitors all of the topics or subsets of topics to be displayed or managed from a central program. Information from this section taken from Steve's Internet Guide[5].

Packet Structure

The MQTT standard utilizes a fairly lightweight packet header of 8 bytes with a predefined set of packet types such as a subscribe versus a publish event. Since MQTT utilizes a TCP connection, there is a greater overhead than with UDP message flow due to the connection-oriented nature of TCP. However, MQTT does get the added benefit of the reliability of TCP to more or less guarantee transport of MQTT packets to the broker. Thanks to Steve's Internet Guide, a nice packet structure visualisation is shown below[6].

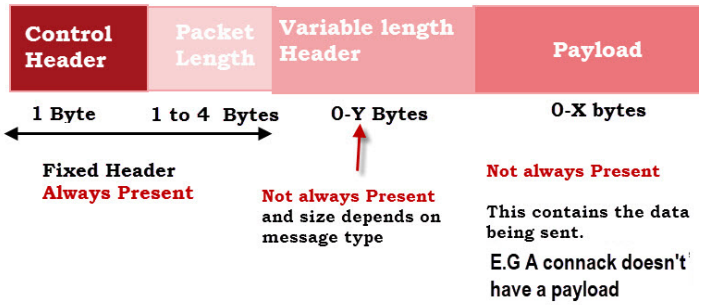


Figure 2.1 MQTT Packet Structure[7]

The control header of the packet is fixed at 2 bytes which helps limit the overhead. The fixed header format is shown in the image below.

[4] Image from: https://www.paessler.com/it-explained/mqtt

[5] MQTT Topics http://www.steves-internet-guide.com/understanding-mqtt-topics/

[6] Steve's Internet Guide MQTT packet structure http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/

[7] Image from Steve's Internet Guide: http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/

Bit	7	6	5	4	3	2	1	0
Byte 1	MQTT Control Packet Type				Flags specific for each MQTT Control Packet Type			
Byte 2	Remaining Length							

Figure 2.2 MQTT Fixed Byte Header[8]

Due to the limit of 4 bits in the "MQTT control packet type" field, there is a limit of 16 possible packet types which are shown in figure 2.3 below. The flags field is only used for the PUBLISH control packet which specifies quality of service levels and whether to retain the message which is explored further in the features section below. The remaining length field in the 2 byte header is also calculated based on the control packet and how much extra data exists within the packet.

Packet Type	Description	Value	Direction of flow
CONNECT	Client requests a connection to a Server	1	Client to Server
CONNACK	Acknowledge connection request	2	Server to Client
PUBLISH	Publish message	3	Either
PUBACK	Publish acknowledgment	4	Either
SUBSCRIBE	Subscribe to topics	8	Client to Server
SUBACK	Subscribe acknowledgment	9	Server to Client
UNSUBSCRIBE	Unsubscribe from topics	10	Client to Server
UNSUBACK	Unsubscribe acknowledgment	11	Server to Client
PINGREQ	Ping request	12	Client to Server
PINGRESP	Ping response	13	Server to Client
DISCONNECT	Disconnect notification	14	Client to Server

Figure 2.3 MQTT Control Packets[9]

As seen in the above discussion, the overall MQTT packet structure was designed to be incredibly simple to keep the protocol as lightweight as possible and reduce the amount of overhead needed to implement.

Features

MQTT has a few special features that help expand the protocol to meet other applications. These features include Quality of Service (QoS), clean sessions, last will messages, and the retain option. QoS levels define how "guaranteed" it is for the message to make it to the interested subscribers. QoS level 0 is a "fire and forget" where the server does its best to send the message out to the subscribers but there is no guarantee of delivery. QoS level 1 is an "at least once delivery" where the message is guaranteed to arrive to the subscribers, but it may send more than once. QoS level 2 is an "at most once delivery" where the message will be sent exactly one time to the subscribers.

Clean sessions are an interesting aspect to MQTT. When the clean session flag is set to true, this means a non-persistent connection is established and the server will not store any information regarding subscriptions or undelivered messages. When the clean session flag is set to false, a persistent connection is established and the server will store subscription information as well as hold any undelivered messages for the client which is useful for when processes go down. The use of a persistent connection requires a client id so the server can maintain which client needs to get messages and once a message has been delivered to the client, it is immediately deleted from the server[10]. A good table that can be used for reference from Steve's internet guide is shown in figure 2.4 below.

[8] MQTT Packet Header Information: https://docs.solace.com/MQTT-311-Prtl-Conformance-Spec/MQTT%20Control%20Packet%20format.htm

[9] MQTT Control Packets https://docs.devicewise.com/Content/Products/IoT_Portal_API_Reference_Guide/MQTT_Interface/MQTT-Control-Packets.htm

[10] MQTT clean sessions http://www.steves-internet-guide.com/mqtt-clean-sessions-example/

Retain Message, Clean Session and QOS Table

Clean Session Flag	Retain Flag	Subsribe QOS	Publish QOS	Published Message Always Reeeived
True	False	0	0	No
True	False	0	1	No
True	False	1	0	No
True	False	1	1	No
False	False	0	0	No
False	False	0	1	No
False	False	1	0	No
False	False	1	1	Yes – All messages
True	True	0	0	Yes –Last Message only
True	True	0	1	Yes –Last Message only
True	True	1	0	Yes –Last Message only
True	True	1	1	Yes –Last Message only
False	True	0	0	Yes –Last Message only
False	True	0	1	Yes –Last Message only
False	True	1	0	Yes –Last Message only
False	True	1	1	Yes – All messages

Note: QOS 1 and QOS 2 produce same result. Therefore for QOS 1 in the table read 1 or 2.

Figure 2.4 MQTT Clean Sessions[11]

Last will messages are interesting in that they are sent upon a network drop or ungraceful disconnect of some kind. This feature is implemented to inform other clients about when another client goes down and can no longer receive messages. The last will message is discarded when the client sends a graceful DISCONNECT packet. The last will message gets sent out to the other clients upon an ungraceful disconnect[12]. This feature can be useful when developing applications so that the other nodes can accordingly handle the dropped node.

The broker will retain a message when the retain flag is set to true in the PUBLISH packet. It is important to note that the broker will only retain the last message that had retain set to true for a particular topic[13]. When a new subscriber subscribes to the topic, it will receive the retained message. Typically that is the purpose for using the retained messages, for when you want clients to receive a particular message upon initially subscribing to a topic. Retained messages can also be used in conjunction with last will messages to help clients restore a dropped connection by retrieving the last message before a connection drop.

Security and Reliability

Since MQTT runs over TCP, the transport security of TLS/SSL can be utilized to protect the transmission of the data being sent over MQTT. Setting up TLS support is fairly straightforward and involves generating certificate files much like setting up TLS for HTTP would be performed.

On the server side, user accounts can and are encouraged to be created to prevent random connections from anonymous users. Logging in to the server is done through the initial connect packet where the username and password are included as data. Along with the creation of users, access control lists can also be implemented to keep stricter security to prevent nonusers or unprivileged users from viewing potentially sensitive information.

MQTT as a protocol has reliability built in with the QoS levels described in the features section. Quality of service helps to define, at the protocol level, some more confirmation that a client has received a message intended for them. Using QoS levels in conjunction with last will testaments and retained messages helps to maintain reliability and accountability between all the clients connected to the system.

Industry users and their applications

MQTT is well designed to work for stream-based applications. Facebook messenger is one of the largest MQTT applications currently on the market[14]. Facebook uses text messages as data being sent between clients that are subscribed to a topic that

[11] MQTT clean sessions http://www.steves-internet-guide.com/mqtt-clean-sessions-example/

[12] MQTT Last Will and Testament https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament/

[13] MQTT Retained Messages https://www.hivemq.com/blog/mqtt-essentials-part-8-retained-messages/

[14] Facebook using MQTT https://engineering.fb.com/2011/08/12/android/building-facebook-messenger/

links two or more users. Amazon Web Services and the IBM Watson IoT programs both utilize and offer support for MQTT applications on their platforms. Other notable companies that utilize MQTT are: Eurotech, Cisco, Red Hat, Software AG, Tibco, ITSO, M2Mi, Indusoft, and Fiorano.

AMQP

Background

AMQP originated in 2003, and was created by JPMorgan Chase. In 2011, AMQP was adopted as an OASIS standard. AMQP is an application layer protocol for message-oriented middleware. It typically runs over TCP, and contains basic authorization implementations for security. AMQP is also designed to be data agnostic as was MQTT so any data format can be sent. Standard unencrypted traffic is done over port 5672 and port 5671 for encrypted TLS traffic (AMQPS).

Communication Model

AMQP follows the same pub/sub model that MQTT uses. From a very high level, AMQP's architecture contains producers which push messages to a broker which manages exchanges and queues. An enqueued message is sorted by exchanges to a proper queue, which appropriate consumers then access to dequeue a message. In this model, unlike MQTT, the central broker holds onto the messages until there are subscribers to pull messages from the queues instead of immediately routing them to subscribers.

Messages are sorted via bindings and routing keys. A binding is a 'link' that is setup to connect a queue to an exchange. A routing key is a message attribute the exchange looks at to decide which (if any) of it's bound queues to route a message to. Messages that cannot be routed to any queues are either dropped or returned to the publisher.

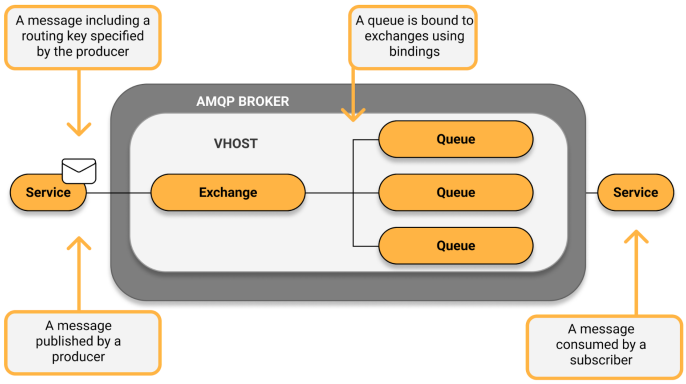


Figure 3.1 AMQP server and communication structure[15]

Packet Structure

Data in AMQP is sent in what is called frames. Much like with HTTP, AMQP has a myriad of frame options which impacts the size of the packets. Frames contain a frame header that contains such information as type and size of the payload. Then there is the actual payload, which is then followed with an end-byte marker which is a single byte to determine the end of the frame.

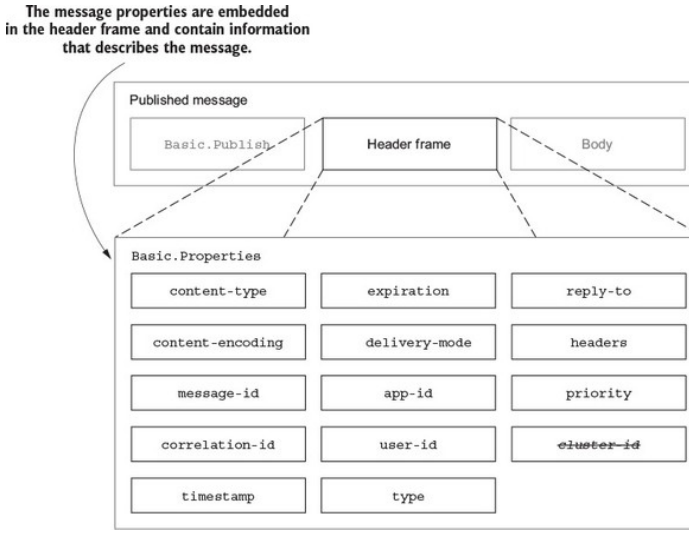


Figure 3.2 AMQP packet structure header options[16]

[15] Image from: https://www.cloudamqp.com/blog/what-is-amqp-and-why-is-it-used-in-rabbitmq.html

[16] Image From: https://livebook.manning.com/book/rabbitmq-in-depth/chapter-3/11

There are five types of frames that can be sent, each one serving a valuable purpose. A protocol header is a frame sent to establish a connection between broker and client, and is used only once. A method frame carries rpc requests and responses (example: used in Basic.Publish). A content header frame is used to send properties of content. For example the content header frame may have the content-type of a message that is going to be published, as well as a timestamp. A body frame is the actual content of the message being sent, and it can be split into multiple frames if the message surpasses the limit (the default frame size limit is 131 kilobytes). Lastly is the Heartbeat Frame, which is used to confirm that a client is still alive.

There are quite a few notable properties in the 8 byte header frame. One such property is delivery-mode, which marks a message as persistent or transient (2 is persistent, anything else is not). Content-type is another such property, which is used to describe media type and its encoding. Reply-to is yet another property, which is commonly used to name a callback queue. While all the properties have their own value, the last one worth mentioning is correlation-id, which is used to correlate responses with requests.

Features

AMQP's main features revolve around its queues and exchanges.

Queues store messages that are consumed by a consumer. Queues are named, this is so that exchanges can choose proper queues to send messages to, and that consumers can receive messages from proper queues. Queues can also be set to be durable. This means that the queue will be able to survive a broker restarting. Queues can also be set to be exclusive, this means that they are only used by one connection and that they will be deleted once that connection closes. Queues can also be set to be automatically deleted when they no longer have any subscribers. The last major feature of queues which is worth mentioning is that they can accept arguments, which are used by plugins and broker specific features.

Many of the features of AMQP come from what type of exchange is being used. Exchanges define the relationship between a message's routing key, and the bindings that connect queues to the exchange. In AMQP's architecture there are 4 types of exchanges. Direct exchanges are the most simple and straightforward types of exchanges, where a message goes to the queue(s) with the binding key that exactly matches the routing key of the message. The binding key either matches with a routing key or it doesn't.

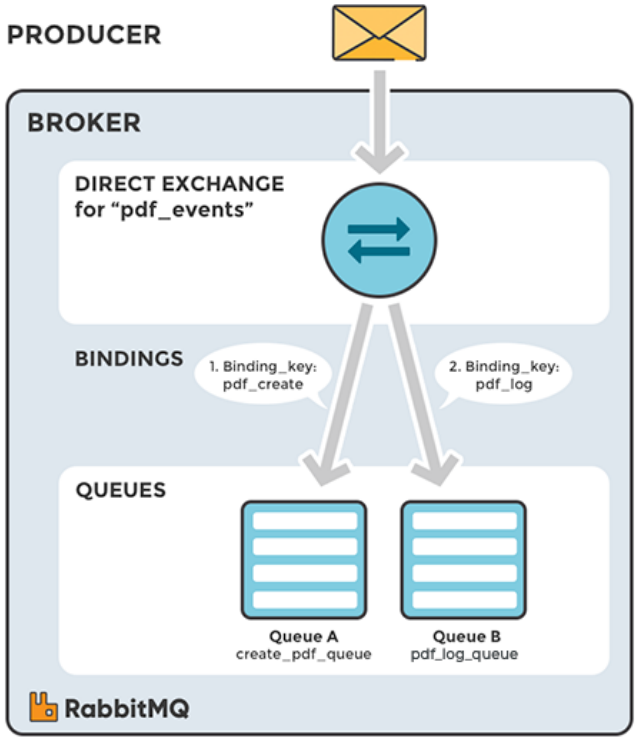


Figure 3.3 AMQP Direct Exchange[17]

The next type of exchange is a topic exchange, which routes messages to queues based on wildcard matches between the routing key and the routing pattern, which is specified by the queue binding. Messages are

[17] Image from: https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html#direct-exchange

routed to one or many queues based on a matching between a message routing key and a pattern. The routing key must be a list of words delimited by a period such as: 'word1.word2'. For example, a binding of: *.*.Boulder would pattern match with messages that have routing keys such as: 'John.Smith.Boulder'. Each topic must have a unique name/path. There are also wildcards such as '*' which can substitute for 1 word and '#' which can substitute for zero or more words.

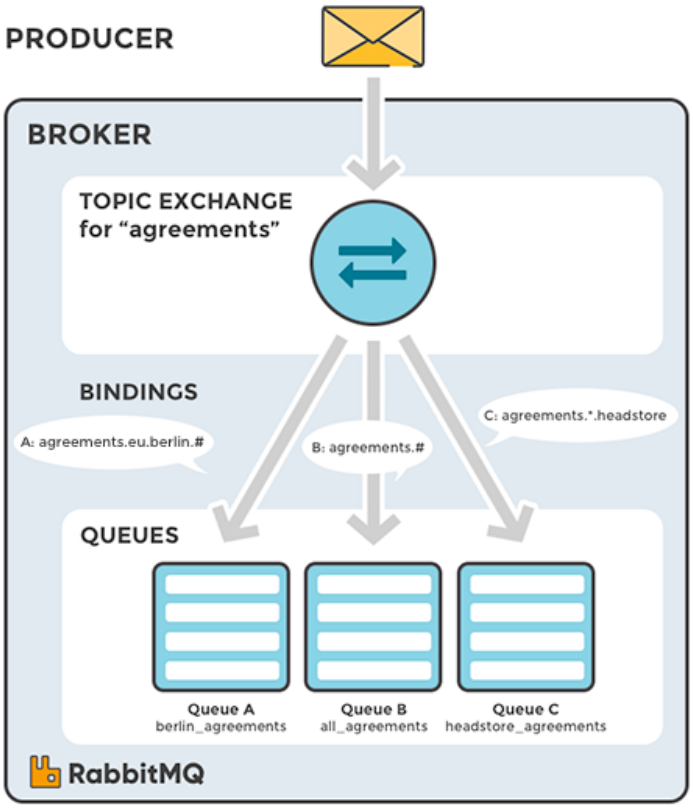


Figure 3.4 AMQP Topic Exchange[18]

The third type of exchange is a fanout exchange. A fanout exchange copies and routes a received message to all queues that are bound to it regardless of routing keys or pattern matching.

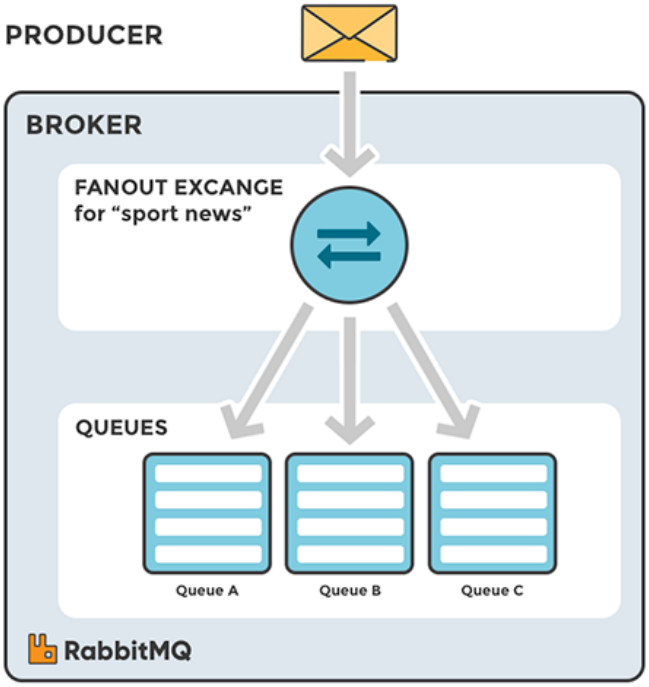


Figure 3.5 AMQP Fanout Exchange[19]

The last type of exchange is a headers exchange, which routes messages based on arguments containing headers and optional values. Headers exchanges are very similar to topic exchanges, but the server routes messages based on header values instead of routing keys. A message matches if the value of the header equals the value specified upon binding.

[18] Image from: https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html#topic-exchange

[19] Image from: https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html#fanout-exchange

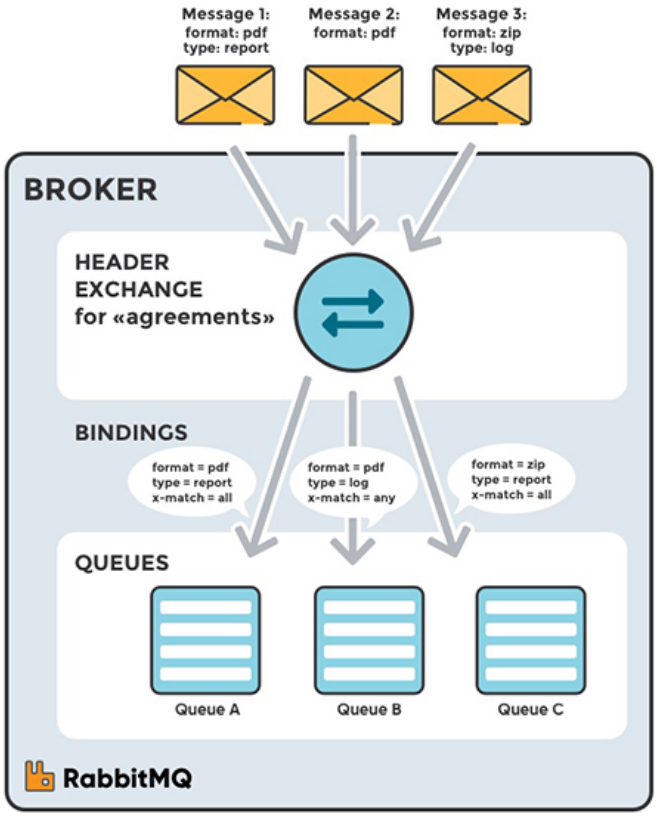


Figure 3.6 AMQP Header Exchange[20]

Security and Reliability

In order to ensure data safety in AMQP, consumer acknowledgement can be implemented. This is a confirmation mechanism that allows consumers to acknowledge deliveries to the node they are connected to. There are 3 types of consumer acknowledgements: Basic.ack which are positive acknowledgements, Basic.nack which are negative acknowledgements that can reject or re-enqueue multiple messages at once, and Basic.reject, which is a negative acknowledgement that can not be used on multiple messages.

Since all messages from queues are pushed to clients asynchronously, implementing Channel Prefetch Settings (QoS) can allow for some load balancing between the clients. The last feature worth discussing in AMQP in regards to reliability is Publisher Confirms which can be implemented as an extension of the protocol.

In regards to transport layer security, AMQP can implement TLS/SSL support much like MQTT due to the nature of running over TCP. Setting up a TLS connection is similar to that of HTTP and MQTT where client/server certificates and keys need to be generated and distributed to perform proper encryption.

Industry users and their applications

AMQP is a popular middleware in applications not dealing with power/bandwidth constrained devices. AMQP offers a lot of flexibility in implementations and can be adapted for cloud computing solutions and telecommunications. Notable users of AMQP in their solutions includes JPMorgan, NASA, RedHat, AT&T, Google, and VMWare[21]. These applications are more designed for devices/nodes in the network that are hardwired to a power connection and bulkiness of communication is not an issue. RabbitMQ is a widely used AMQP service that can be utilized/adapted in a variety of applications such as handling transactions in vendor machines. Other notable companies that utilize AMQP include: Microsoft, Bank of America, Barclays, Goldman Sachs, and Credit Suisse.

CoAP

Background

Added as an IETF standard in June of 2014, CoAP is a specialized web transfer protocol for IoT devices, that runs in the service layer primarily over UDP (although TCP implementations are available). It is designed to run on resource constrained networks (such as lossy networks), for use by power-constrained devices. Standard traffic is done over port 5683 and DTLS encrypted traffic (coaps://) is done over port 5684.

[20] Image from: https://www.cloudamqp.com/blog/part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html#headers-exchange

[21] AMQP users and applications https://www.amqp.org/about/examples

Communication Model

CoAP uses a request/response interaction model between endpoints which is designed to easily interface with HTTP[22]. Data sent over CoAP is designed to easily translate to HTTP for simplified integration with the web. For the sake of simplified translation, a CoAP application often contains a proxy as part of its infrastructure. The CoAP architecture allows for implementations to have a client node command another node by sending a packet and having the CoAP server extract, interpret and decide what to do based on its logic. A CoAP network tends to be made up of a bunch of CoAP devices (such as sensors), all communicating over CoAP with a server, which in turn is communicating with a REST-CoAP Proxy, which in turn is using HTTP communication to communicate with the REST internet. Because CoAP is restful, it's clients support the use of GET, PUT, POST, and DELETE methods when communicating with a CoAP server.

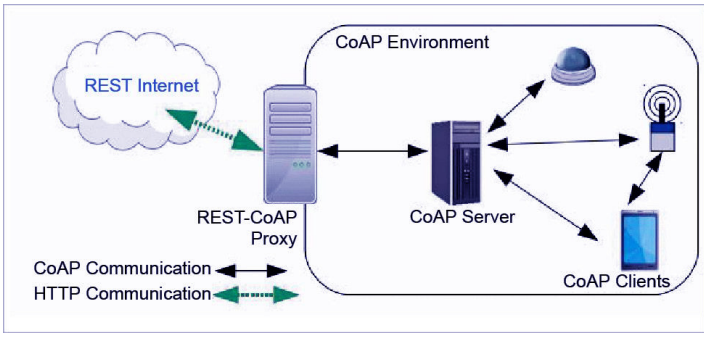


Figure 4.1 CoAP communication and server structure[23]

Packet Structure

Beyond its reduced power requirements, CoAP also offers very small packet sizes. In fact, the packet size can get down to only 4 bytes if token options (and of course a payload) are omitted. The CoAP packet consists of: The Version, Message Type, Token Length, Code, the Message ID, the token, options if they exist, and then the payload.

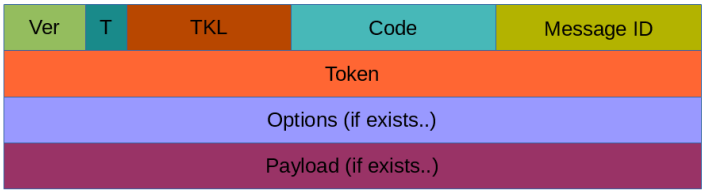


Figure 4.2 CoAP packet structure[24]

Features

Another advantage is regarding security, where Datagram Transport Layer Security (DTLS) can be applied to the UDP the protocol uses. The ability to have Asynchronous Message Communication is also an advantage, where clients can request to observe a device by setting a flag, and as a result the server can then stream the state of changes to the client as they happen, with either side being able to cancel the 'observe' request (this then operates much like a pub/sub model where a resource is streamed to the 'observer' as it gets updated). Lastly, because the client is built to support IPv6, so it also supports multicasting. This protocol also supports resource discovery, but firewalls and network address translation ("CoAP can have problems communicating with devices behind a NAT since the IP can be dynamic over time") can cause problems with resource discovery operating properly.

CoAP has an existing extension to its protocol that allows CoAP clients to "observe" resources. The resource observation extension (https://tools.ietf.org/html/rfc7641) allows a client to retrieve a representation of a resource, and keep this representation up to date by a server over the length of a certain period of time. This is a very useful protocol extension to enact, if you expect the state of resources on a server to change over time.

[22] CoAP Abstract https://tools.ietf.org/html/rfc7252
[23] Image From: https://www.linkedin.com/pulse/4-major-iot-protocols-mqtt-coap-amqp-dds-malini-shukla/
[24] Image From: https://dzone.com/articles/coap-protocol-step-by-step-guide

Security and Reliability

Since the original implementation of CoAP runs over UDP, its primary security feature is DTLS, allowing messages to be sent in a way that prevents eavesdropping, tampering, or message forgery.

CoAP supports 4 different types of messages. The types of messages are defined in the T field of the CoAP header. The first 2 message types are Confirmable (CON), and Acknowledgement (ACK). An ACK is used to confirm that a CON (identified by its message ID) arrived. ACK must echo the message ID of the CON, and must carry a response or be empty. The 3rd type of message is the non-confirmable (NON) message, which when sent does not require an ACK. The last of the 4 types of messages is the Reset (RST) message, which when sent indicates that a specific message (either CON or NON) arrived, but some context is missing to properly process it. Usually encountered when a receiving node has rebooted and has forgotten some state that would have been required to interpret a message. RST messages must echo the message ID of the NON or CON, and must be empty.

The most reliable message type is CON, as the receiving node must retransmit back an ACK (or if context is lacking, reject with a RST). Retransmission is controlled by 3 main variables that the sender must keep track of. The 3 variables are RESPONSE_TIMEOUT, RESPONSE_RANDOM_FACTOR, and MAX_RETRANSMIT. These 3 variables are responsible for both choosing how long a CON message should wait for a response before attempting to resend the message, as well as how many times the sender should attempt to resend a message[25]. The time limit for resending a message is determined by some random number between RESPONSE_TIMEOUT and RESPONSE_TIMEOUT * RESPONSE_RANDOM_FACTOR.*($2^{MAX\_RETRANSMIT}$-1). The same message ID must not be reused during the retransmission window. The MAX_RETRANSMIT variable is used to determine how many times a message should attempt to be retransmitted.

Additional Information

CoAP is designed with REST in mind and is meant to be used in a constrained RESTful environment. Since the protocol is similar to HTTP, programmers familiar with programming RESTful web applications will have no problems picking up and using this protocol. Obtaining a value from a sensor is not much different than obtaining a value from a web API. Because it is RESTful, resources have a URL, and many familiar verbs like GET, PUT, POST, and DELETE are still used. Going off these familiar verbs, we can look at exactly how a resource is accessed in CoAP. Resource placement follows a directory structure where the URI looks like: coap://heater.net/room1/temperature, with temperature being the resource we are aiming to get. The response to our get would have the resource we are seeking (in this case the temperature). IoT devices (endpoints) can find the resource directory by utilizing IP multicasting.

Industry users and their applications

CoAP is incredibly popular in IoT devices due to its ability to function in low power environments and can even be found in some of IKEA's smart lights (Tradfri). RIIM (Radio Crafts Industrial IP Mesh), which is an embedded easy to use RF system with direct IP addressing also uses CoAP. Other notable companies that utilize CoAP include: Cisco, Contiki, Erika, and IoTivity.

[25] CoAP Timeout Extension https://tools.ietf.org/id/draft-li-core-coap-request-timeout-option-00.html

LwM2M

High level overview

LwM2M is a machine to machine protocol that was first approved and published by the Open Mobile Alliance (OMA) in December of 2013[26]. LwM2M is still a fairly up-and-coming protocol with new specifications for an LwM2M 1.0 being updated and defined by OMASpecworks in 2017[27]. The protocol is designed with constrained devices in mind, particularly ones with memory and battery limitations. To achieve high performance on such devices the protocol runs on top of CoAP. The main intention for the protocol was to provide device management such as configuration and firmware updates, as opposed to strictly message sending which was the intention of the other protocols mentioned in this paper. The operating ports are the same as CoAP (5683/5684) since LwM2M is built on top of CoAP.

Communication model

Since LwM2M utilizes CoAP for the underlying communication we see the familiar GET, PUT, POST, and DELETE methods of communication at the low level. LwM2M however, has four interfaces between the client and server:

1. Bootstrapping

The bootstrap interface allows for provisioning of a client as to allow the client to register with one or more LwM2M servers.

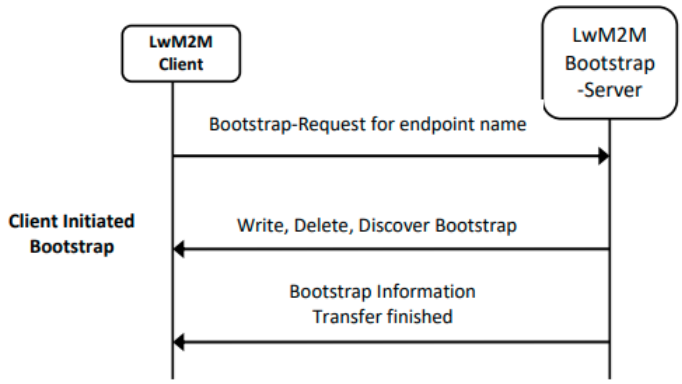


Figure 5.1 LwM2M bootstrap flow[28]

2. Device registration

Registration provides the server with the supported objects as well as the existing object instances belonging to the client.

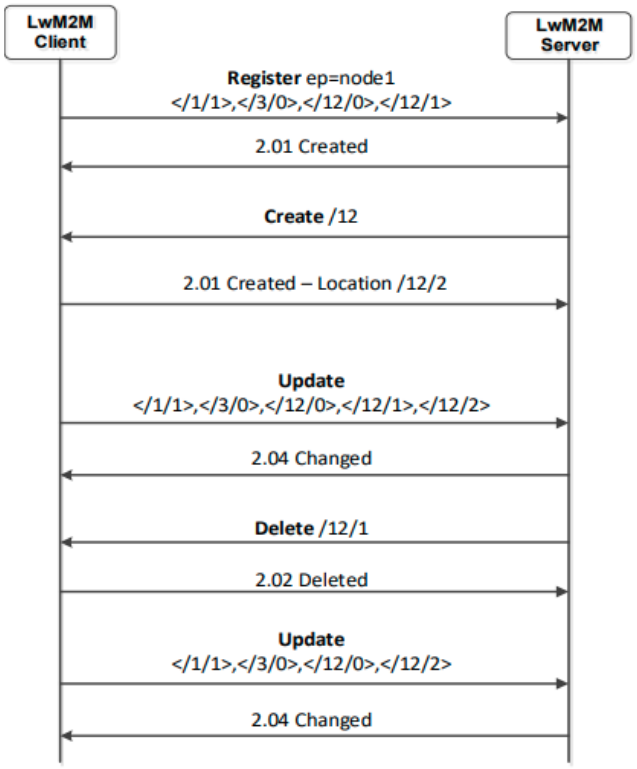


Figure 5.2 LwM2M Device Registration Flow[29]

[26] LwM2M tutorial slides by Zach Shelby https://www.slideshare.net/zdshelby/coap-tutorial

[27] OMASpecworks blog post on LwM2M https://omaspecworks.org/what-is-next-for-lightweightm2m-lwm2m/

[28] LwM2M Core Documentation (Page 32) http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf

[29] LwM2M Core Documentation (Page 53) http://www.openmobilealliance.org/release/LightweightM2M/V

3. Device management and service enablement

This interface is used by the server to access instances of objects on a registered client. This interface supports "Create", "Read", "Write", "Delete", "Execute", "Write-Attributes", and "Discover" operations.

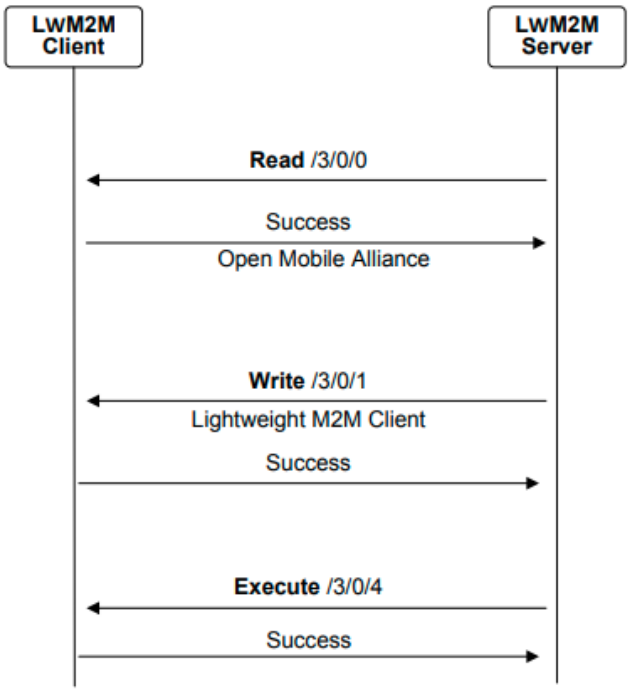


Figure 5.3 LwM2M flow of managing resources[30]

4. Information Reporting interface

This interface provides the server with the ability to "observe" changes in resources on the client by using the observe feature built into CoAP. The server sends an observe operation and then the client will send notifications whenever a change occurs on that resource.

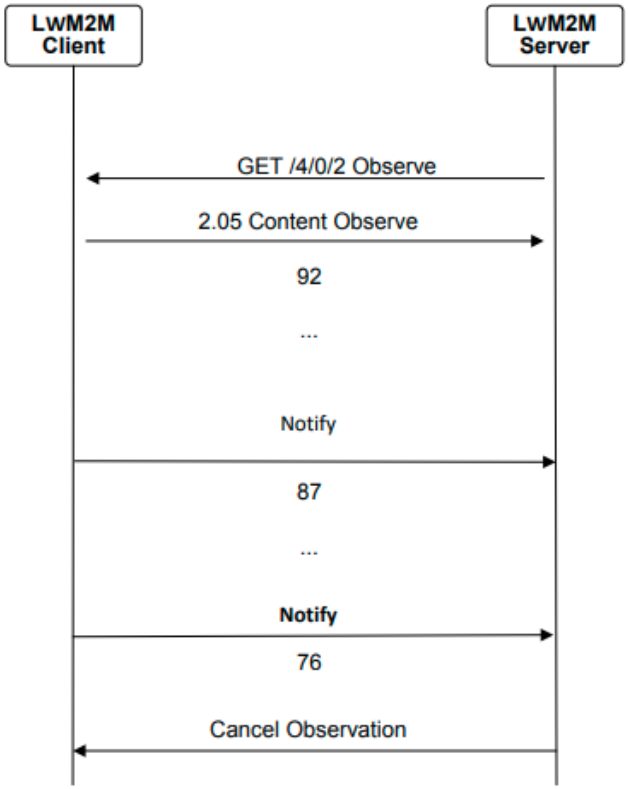


Figure 5.4 LwM2M Observe Feature Flow[31]

Data Model

LwM2M uses an object based resource model with many objects already having definitions as a part of the protocol. The objects are identified by 16 bit integers in order to save on bandwidth. Objects have multiple resources associated with them, for instance the "device" object has a "battery" resource. Depending on the definition, an object may have multiple instances. Objects are referenced in the protocol using a uri in the form of:

/{object ID}/{object instance}/{resource ID}

1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf

[30] LwM2M Core Documentation (Page 54) http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf

[31] LwM2M Core Documentation (Page 62) http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf

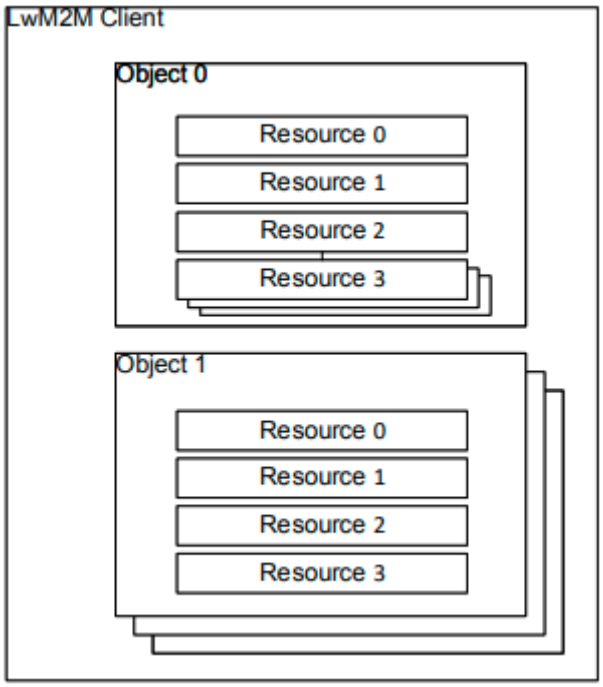


Figure 5.5 LwM2M Object Model[32]

Security

LwM2M can utilize the same transport layer security of the DTLS protocol offered by CoAP to secure communication. In addition, the protocol requires that client and server authenticate each other prior to sending any information. This is facilitated by the bootstrapping server which provides keys for a client to authenticate with a server. Authentication is paramount to ensure that devices are receiving the correct information and unauthorized access is prohibited by unknown clients. A device management framework such as LwM2M is critical to ensuring devices are up-to-date with security patches and feature updates to code.

Protocol Proof of Concept

For the proof of concept in evaluating the protocols in depth, the SARA R500S Eval board[33] was used and 4 of these kits were provided by U-Blox for this project. This board comes configured with a cellular module and antenna to connect to cellular networks. This project also utilized an AT&T sim card/plan for setting up IP connections and sending data via the cell networks for each of the protocols. In order to run each of these protocols in an automated manner, a python framework was developed that can send AT commands to the SARA R5 board via a serial connection (see appendix A.1 for notes on python libraries available for serial connections)[34]. The following section will discuss the different AT commands that were used with the U-Blox board, but a full list of AT commands can be found at https://www.u-blox.com/sites/default/files/SARA-R5_ATCommands_%28UBX-19047455%29.pdf which entails a full explanation of the AT commands (refer to this document for further questions regarding implementations on the SARA R5 board).

In order to show the proof of concept for each of the protocols on the SARA R5 board, some sample json data was provided as exemplary data to get a feel for how these protocols will perform in an application for BI's products. The testing json data consisted of five different events: a "single event", a "single event with data", a "location event", an "update with multiple events event", and lastly, a "four hours of GPS data event". These events were used to simulate the different applications and scenarios that might exist in which these protocols would be used at BI. Each event consists of an event code to signify what information is being sent along with a timestamp. For the events that also contain data, the data field is supplied and in

[32] LwM2M Core Documentation (Page 68) http://www.openmobilealliance.org/release/LightweightM2M/V1_2-20201110-A/OMA-TS-LightweightM2M_Core-V1_2-20201110-A.pdf

[33] SARA R500S Eval Board Site https://www.u-blox.com/en/product/sara-r5-series

[34] Python Framework for Sending AT commands for the protocols https://github.com/ckardaras/iot/tree/master/Python_Testing

the case of the GPS location events, fields such as lat and long are supplied. The same json file was used and sent via MQTT and CoAP on the board, and through AMQP run with python on a computer client (due to the inability to implement on the board in time). Packet captures were then conducted on the server side and current consumption measurements were taken for the MQTT and CoAP scenarios, which are evaluated in greater detail in the current consumption analysis section of this paper.

Implementing the Protocols on the SARA R500S from U-Blox

Overview

The protocols used on the board require an internet connection and therefore a Packet Data Protocol (PDP) context is required to be set up before any of the protocols can be utilized. This process is fairly straightforward and only requires a few AT commands. The setup of the UPSD profile to be used on the AT&T network is run with the command AT+CGDCONT=1,"IPV4V6","m2m.com.attz" which will point to the APN m2m.com.attz. The next commands to run is AT+UPSD=0,100,1 (maps the UPSD profile specified by cid 1 which was setup in the previous command) followed by AT+UPSD=0,0,2 (specifies use of ipv4 with the default username) and then AT+UPSDA=0,3 (activates the PDP context with the given parameters) which properly sets up the PDP context with regards to the AT&T network. All of the required AT commands to send data via MQTT and CoAP are in the python files on the project github repository found in Appendix A.6. The python files are more in depth and follow the exact AT command execution required to set up these connections, but a summary of the AT commands are also explained in detail in the following sections.

MQTT Implementation

MQTT is fairly straightforward to implement on the SARA R5 board and only requires a few AT commands to publish and subscribe to an established broker. To specify the server to connect to, the command AT+UMQTT=2,"<server_name>",<port> is used which will store the parameters specified as the connection parameters. Then to specify the secure mqtt option, use the command AT+UMQTT=11,<0/1> use 0 for no tls encryption, use 1 for tls encryption. Setting up certificates to be used for the TLS encryption option can be followed from the AWS IoT setup document[35]. To specify a username and password with the mqtt client, use the command AT+UMQTT=4,"<username>","<password>" which will use those parameters to login to the MQTT server (the credentials must be setup first on the server side). After setting up the overall configuration parameters, logging into the MQTT server is as easy as running the command, AT+UMQTTC=1 (AT+UMQTTC=0 is to gracefully logout of the server). Once connected, the board will maintain a tcp connection and subscribing to a topic can be done with the command AT+UMQTTC=4,0,"<topic>" and publishing can be done with the command AT+UMQTTC=2,<QoS_Level>,<Retain>,<hex_mode>,"<topic>","<data>" (hex_mode 0 = interpret data payload as ASCII, hex_mode 1 = interpret data as raw hex). If subscribed to a topic and data is published to that topic, the board will asynchronously receive that data and an AT+UMQTTC=6,<message_num> will come in denoting that a message has been received. To read the message, use the command AT+UMQTTC=6,<0/1> (0 to get all messages, 1 to get the next message) to retrieve the received message. Those are the essential commands for using the MQTT features on the board.

There are limits imposed on the actual AT command with sending MQTT data via the AT+UMQTTC=2,...

[35] MQTT setup for AWS IoT Core
https://www.u-blox.com/en/docs/UBX-20044809

command. If in ASCII mode (<hex_mode>=0), there is a limit of 1024 characters and if in hex mode (<hex_mode>=1), there is a limit of 512 octets to be sent from that specific command. For larger data packets that exceed these limits, fragmentation can be gracefully handled by utilizing a client specific format for sending data such as appending a field to a json packet denoting how many packets are associated with this specific payload. The client code would then have to impose checks on how large the data is and handle it accordingly when the data payload is too large. Alternatively, the AT command AT+UMQTTC=3,... can be used which denotes publishing a file to a topic. The actual MQTT topic message limit in the specification is around 260 Mb[36]. Utilizing the filesystem on the SARA R5 board could then be used to get around the message limits associated with the AT+UMQTTC=2 command which takes in raw data as a parameter to send as the MQTT message. The AT commands manual mentions, however, that there is a theoretical limit of around 1100 files that can be stored depending on the average file size. This could make the file method unreasonable as a file would have to be created and then deleted after sending the data which incurs more operations by the board to send the message, which would increase the current consumption.

AMQP Implementation

It is important to note that AMQP does not come default on the SARA R5 board and thus requires an immense amount of effort to implement. As discussed in the AMQP overview discussion, this protocol is a lot more intensive than MQTT and CoAP. This therefore brings a lot more baggage when implementing the protocol from scratch due to the lengthy amount of options to specify in the packets and the involved connection/message communication. AMQP can be implemented using a manual TCP connection setup through AT commands. It is important to note that the AT+USOWR command, which is used to send data via a TCP connection, has a limit of 1024 bytes in ASCII mode and 512 octets in hex mode. Some of the packets received by the computer AMQP client by the server were up to about a max of 530 bytes in length. This means that some of the AMQP control packets may need to be edited for brevity for cases such as string names. Due to the time constraint of this project, this method of implementing AMQP on the board was not verified, however the theory of setting up a manual TCP connection supports its use on the board. AMQP would absolutely be the most involved and heavy protocol to implement with the SARA R5 board.

CoAP Implementation

Setting up the board for CoAP using AT commands are as follows. Set the board to verbose by running: “AT+CMEE=2”. Then set the current profile as valid by running: “AT+UCOAP-4,1”. Then run: "AT+UCOAP=2,0,1", "AT+UCOAP=2,1,1", "AT+UCOAP=2,2,1" and "AT+UCOAP=2,3,1", to enable automatic recognition of URI_HOST, URI_PORT, URI_PATH, and URI_QUERY respectively directly from the URI. The next command is to store the current profile as profile number 0: "AT+UCOAP=6,0". Lastly, the command: “AT+UCOAP=5,0”, restores profile 0 as the current profile.

Once a CoAP profile is set up, GET, PUT, and POST commands all start the same by establishing a connection to the server and resource by running the AT command: 'AT+UCOAP=1, server uri + resource uri’. Then if you want the message to be confirmable run the AT command: "AT+UCOAP=2,5,1", otherwise run "AT+UCOAP=2,5,0" for non-confirmable. The last AT command sent to the boards, are the actual AT verb

[36] Steve’s Internet Guide to MQTT http://www.steves-internet-guide.com/mqtt-broker-message-restrictions/#:~:text=The%20MQTT%20protocol%20allows%20messages,especially%20public%20brokers%2C%20will%20restrict

commands. GET requests require the command "AT+UCOAPC=1". PUT requests require "AT+UCOAPC=3,hex_data_of_message,0". Lastly, POST requests require "AT+UCOAPC=4,hex_data_of_message,0". Regardless of what verb message you send, it is best to wait for the board to respond to your AT command to confirm the message has been sent.

Like MQTT, running CoAP on the SARA R5 board has some limitations in how many bytes can be sent via the AT commands. From the AT commands manual, the "AT+UCOAPC=" command can only take in a max of 512 octets and the data must be sent in hex and not ascii. Data received by the get request is also received in hex and must be decoded if the desired output is in ascii or another data format.

LwM2M Implementation

While the SARA R500S board has built in AT commands to run the LwM2M protocol, there appears to be some functionality that is lacking in terms of implementing this protocol with personally built infrastructure. After looking into the documentation and attempting to connect to a Leshan server (a primary LwM2M server) setup on an AWS instance, the commands came back with an error in which the commands could not be run. The LwM2M section of the AT commands manual also mentions a connection to the MNO profile which is related to the cell network that the board is connected to. It is our best assumption that this suggests the LwM2M functionality is linked to firmware updates for the board from the actual cell companies. A support request was sent to U-Blox for help in implementing the LwM2M AT commands on the board, but no concrete assistance was given. It would be best to get direct assistance from U-Blox in order to try and implement LwM2M for the business driven server for handling device management operations on the boards.

Packet Capture Analysis

The Setup

To handle a centralized server system to facilitate traffic for each of the protocol evaluations, Amazon Web Services (AWS) was used to set up an EC2 instance. After setting up an AWS EC2 instance along with servers to handle each of the different protocols (See Appendix A for a list of servers used), the related ports were opened to receive data from the different protocols. Using the tcpdump command line tool, packet captures were then collected by running the following command: tcpdump -i eth0 port <port_num> -w <protocol_outfile>. This then generates a packet capture file which can be analyzed in the Wireshark tool to see how many bytes and packets are being sent for the same amount of application data across the different protocols being examined. It is important to try and minimize the amount of traffic being sent to the server over the different protocol ports in order to keep clarity. This was done with this analysis section and only a single device was sending packets during the collected packet capture, which makes the Wireshark analysis more succinct and easier to interpret. Find the full packet captures on the project Github page described in Appendix A.6.

Analysis

As mentioned in the protocol specific overview sections, each of the protocols have their own nuances for sending data which can incur penalties of needing to send more bytes. For AMQP and MQTT which rely on a TCP connection, there will be more overhead in the setup and tear down of the connection channel. The packet captures show the differences in how much data is being sent and how much overhead cost there is in order to send the same amount of data. There is a table at the end of this section that outlines the overall bytes sent for each of the different "events" that was outlined in the protocol proof of concept section. This also includes a table of the amount of bytes sent when

using the encryption schemes of TLS and DTLS for the different protocols.

MQTT Capture

Figure 6.1 shows the packet capture from the connect command for the MQTT protocol. Since this is the first message sent to establish the connection, there are the initial TCP [SYN], [SYN,ACK], [ACK] to start up the connection. Then the remaining ACK packets are from the connect command.

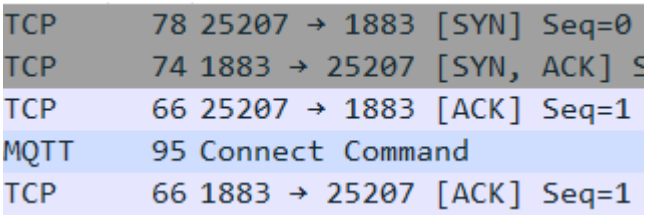


Figure 6.1 MQTT Connect Packet Capture

The overall data sent via MQTT is reasonably small, but incurs the extra bytes from setting up and maintaining the TCP connection. The connect command will also have extra bytes when features such as authentication are used. The byte overhead for these features depends on the amount of characters used in the username/password since these are sent as ASCII data. Due to the small 2 byte overhead, the overall bytes sent is effectively dependent on the amount of data being sent and how long the topic names are. To keep the size of the data transfer low, best practices suggest keeping topic names as small as possible. Figure 6.2 shows an example of a publish command.

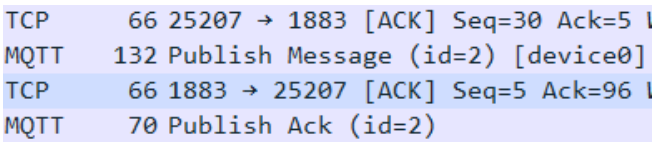


Figure 6.2 MQTT Publish Packet Capture

The publish message here was an example of a "single event" being published to the topic "device0." Overall, there is a fairly consistent overhead of the 66 bytes for TCP added when sending a message to the server. In this case, the number of bytes sent is directly dependent on the amount of data that is desired to be sent to the server, in this case there was about 52 bytes of data sent, and the rest is overhead/topic name ('device0'). The final packet capture event that was captured was a disconnect from the server which is shown in figure 6.3.

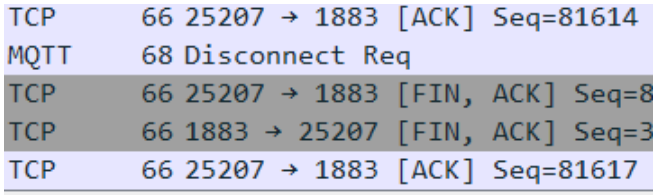


Figure 6.3 MQTT Disconnect Packet Capture

The teardown of the TCP connection is the other overhead event that MQTT incurs. In this case, if a device wanted to wake-up from a sleep state and send data, it would need to send the connect/disconnect events on top of sending the publish event.

Other than the overhead of connecting to the broker and TCP, the use of TLS/SSL to provide transport security also adds a significant amount of overhead to the transmission of data. Figure 6.4 below shows an example of the overhead added when MQTT is used to send a single event payload.

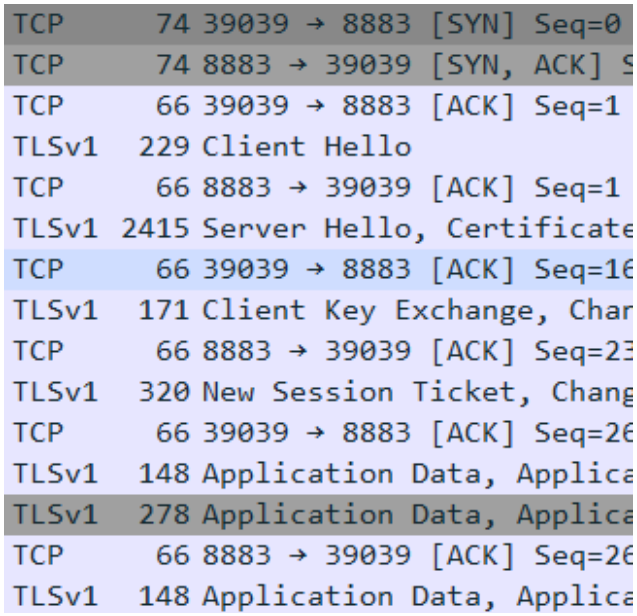


Figure 6.4 MQTT TLS Addition

The addition of TLS adds approximately 3200 extra bytes for the same small transmission of data. In this case, the size of the overhead depends on the size of the certificate being used as it has to retrieve that certificate from the server for the encryption scheme. This would get costly if many transmissions entail only sending a small amount of data and it may be more worthwhile to wait and send larger payloads if possible to reduce the amount of cost in the overhead of establishing the connection to the server via TCP and utilizing TLS support.

CoAP Packet Capture

Since CoAP is a RESTful protocol and operates over UDP, the amount of overhead is greatly reduced. Figure 6.5 below shows an example of sending a "single event." In this case, there is no TCP overhead and focuses more on just sending the desired data. The size of the payload relies on the amount of ASCII data being sent and the length of the resource. Therefore, in order to keep the packet sizes down, the length of resources should be conventionally designed to be as small as possible, much like the MQTT topic names used.

```
CoAP    132 CON, MID:24654, POST, coap://4
CoAP     62 ACK, MID:24654, 2.01 Created
```

Figure 6.5 CoAP POST to a resource

Since there is no extra transport overhead such as establishing a connection to the server, the above packet capture is representative of the entire protocol, the size of the packets changes when more or less data is sent to modify a resource on the server. Changing the verb from POST to PUT or GET is negligible with regards to the packet header as described in the CoAP overview discussion.

The addition of DTLS causes the CoAP protocol to add additional overhead which is expected. An example of the DTLS addition to the protocol is shown in figure 6.6.

```
DTLS…  194 Client Hello
DTLS…  102 Hello Verify Request
DTLS…  226 Client Hello
DTLS…  517 Server Hello, Certificate, Server Key Exchan
DTLS   434 Certificate, Client Key Exchange, Certificat
DTLS…  117 Change Cipher Spec, Encrypted Handshake Mess
DTLS   208 Continuation Data
DTLS…   82 Application Data
```

Figure 6.6 CoAP DTLS Addition

Compared to the unencrypted version of the protocol, adding on transport security adds around a constant 1800 bytes of overhead. However, this is a necessary addition to the protocol and should be used to protect the transmission of the data being transmitted by the devices.

AMQP Packet Capture

As with MQTT, AMQP operates over TCP and is also designed with far more options, which causes the packet sizes and number of messages to increase. As AMQP is designed more for non low-power devices, there are extra messages to establish connection options which is seen in figure 6.7.

```
TCP      66 53709 → 5672 [SYN] Seq=0 Win=64240
TCP      66 5672 → 53709 [SYN, ACK] Seq=0 Ack=
TCP      54 53709 → 5672 [ACK] Seq=1 Ack=1 Win
AMQP     62 Protocol-Header 0-9-1
TCP      54 5672 → 53709 [ACK] Seq=1 Ack=9 Win
AMQP    578 Connection.Start
AMQP    371 Connection.Start-Ok
TCP      54 5672 → 53709 [ACK] Seq=525 Ack=326
AMQP     74 Connection.Tune
AMQP     74 Connection.Tune-Ok
AMQP     70 Connection.Open vhost=/
TCP      54 5672 → 53709 [ACK] Seq=545 Ack=346
TCP      54 5672 → 53709 [ACK] Seq=545 Ack=362
AMQP     67 Connection.Open-Ok
AMQP     67 Channel.Open
TCP      54 5672 → 53709 [ACK] Seq=558 Ack=375
AMQP     70 Channel.Open-Ok
```

Figure 6.7 AMQP Establish Connection

Establishing a connection is quite extensive with all of the possible options that AMQP offers. Once the connection is established, sending data to the queues

is straightforward. The extent of publishing data is shown in figure 6.8 below.

```
AMQP     77 Basic.Publish x= rk=events
AMQP     76 Content-Header
AMQP    131 Content-Body
TCP      54 5672 → 53709 [ACK] Seq=601
```

Figure 6.8 AMQP Publish Data to a Queue

After sending data, there must also be a graceful teardown much like MQTT. This is shown in figure 6.9. The teardown of the AMQP session is similar in terms of number of messages sent and the size of the messages when the connection was established.

```
TCP      54 5672 → 53709 [ACK] Seq=6
TCP      54 5672 → 53709 [ACK] Seq=6
AMQP     88 Channel.Close reply=Norm
TCP      54 5672 → 53709 [ACK] Seq=6
AMQP     66 Channel.Close-Ok
AMQP     88 Connection.Close reply=N
TCP      54 5672 → 53709 [ACK] Seq=6
AMQP     66 Connection.Close-Ok
TCP      54 53709 → 5672 [FIN, ACK]
```

Figure 6.9 AMQP Disconnect

Overall, AMQP adds a lot more overhead to the actual protocol when the connection is set up and data is being sent when compared to MQTT and CoAP. This is especially true when TLS support is enabled since there is the extra cost of sending the certificate and providing encryption as was seen in the MQTT packet capture discussion. An example of the encrypted AMQP packet capture is shown in figure 6.10 for the case of a "single event" transmission.

```
TCP      74 33474 → 5671 [SYN] Seq=0 Win=6424
TCP      74 5671 → 33474 [SYN, ACK] Seq=0 Ack
TCP      66 33474 → 5671 [ACK] Seq=1 Ack=1 Wi
TLSv…   583 Client Hello
TCP      66 5671 → 33474 [ACK] Seq=1 Ack=518
TLSv… 1514 Server Hello
TCP      66 33474 → 5671 [ACK] Seq=518 Ack=14
TLSv… 1020 Certificate, Server Key Exchange,
TCP      66 33474 → 5671 [ACK] Seq=518 Ack=24
TLSv…   192 Client Key Exchange, Change Ciphe
TCP      92 5671 → 33474 [ACK] Seq=2403 Ack=6
TLSv…   117 Change Cipher Spec, Encrypted Har
TCP      66 33474 → 5671 [ACK] Seq=644 Ack=24
TLSv…   103 Application Data
TCP      92 5671 → 33474 [ACK] Seq=2454 Ack=6
TLSv…   619 Application Data
TCP      66 33474 → 5671 [ACK] Seq=681 Ack=30
TLSv…   412 Application Data
TCP      92 5671 → 33474 [ACK] Seq=3007 Ack=1
TLSv…   115 Application Data
TLSv…   115 Application Data
TLSv…   111 Application Data
TCP      92 5671 → 33474 [ACK] Seq=3056 Ack=1
TLSv…   108 Application Data
TLSv…   108 Application Data
TLSv…   111 Application Data
TLSv…   121 Application Data
TLSv…   122 Application Data
TLSv…   118 Application Data
TLSv…   162 Application Data, Application Dat
TLSv…   129 Application Data
TCP      92 5671 → 33474 [ACK] Seq=3199 Ack=1
TLSv…   107 Application Data
TLSv…   129 Application Data
TLSv…   107 Application Data
TCP      66 33474 → 5671 [FIN, ACK] Seq=1492
TCP      66 5671 → 33474 [FIN, ACK] Seq=3281
TCP      66 33474 → 5671 [ACK] Seq=1493 Ack=3
```

Figure 6.10 AMQP TLSv1 Encryption Example

As can be seen in the above figure, TLS adds about 4700 bytes of overhead making AMQP the most costly in regards to overall bytes sent for sending the same amount of application level data.

Packet Capture Comparison Summary

The following table shows the total number of bytes and number of messages sent for each of the events described in the protocol proof of concept section. The encrypted total bytes and encrypted number of messages rows refers to sending the same data via TLS/DTLS so that overhead can be examined.

Event Name	MQTT	CoAP	AMQP
Single Event			
total bytes	769	196	3385
number of messages	10	2	30
total bytes (encrypted)	4634	1880	7427
# messages (encrypted)	18	8	37
Single Event with Data			
total bytes	795	222	3411
number of messages	10	2	30
total bytes (encrypted)	4666	1912	7453
# messages (encrypted)	18	8	37
Location Event			
total bytes	1020	446	3727
number of messages	10	2	31
total bytes (encrypted)	4890	2264	7809
# messages (encrypted)	18	8	39
Multiple Events			
total bytes	1163	589	3778
number of messages	10	2	30
total bytes (encrypted)	5126	2532	7728
# messages (encrypted)	19	8	36
4 Hours GPS Data			
total bytes	104568	119382	111374
number of messages	59	204	79
total bytes (encrypted)	107665	121165	115248
# messages (encrypted)	61	212	82

Current Consumption Analysis

The Setup

Since the SARA R500S board does not come with a current consumption evaluation module, an external oscilloscope had to be attached across the J109 pin on the cellular board module. The J109 pin comes equipped with a jumper that must be removed in order to gain access to the pins directly as is described in the EVK-R5 user guide[37]. The Analog Discovery 2 (AD2)[38] by Digilent was used to measure the voltage drop across the J109 pin. A shunt resistor of 1.5 Ohms was then placed across this pin with the AD2 scope probes attached to the sides of the resistor. A sufficiently small resistor had to be used since the cellular module can operate down to 3.3V but 3.8V is typical[39]. See Appendix B for the resistor setup on the board. Within the Waveforms software[40], which takes in the AD2 scope input, Ohm's law provides us with the current calculation using the formula

$$\frac{(V_2 - V_1)}{1.5\Omega}.$$

The AD2 can sample up to 100MHz, however 16kHz was used to cut down on the amount of samples taken to keep the file size smaller. However, 16kHz was a large enough sample rate so as to not miss any important events. See figures B.1 - B.3 in Appendix B for images describing the setup for current consumption. Only CoAP and MQTT were able to have current consumption measurements taken from the SARA R500S board since an AMQP implementation

[37] EVK-R5 User Guide (Page 20) https://www.u-blox.com/sites/default/files/EVK-R5_UserGuide_%28UBX-19042592%29.pdf

[38] Analog Discovery 2 reference page https://reference.digilentinc.com/reference/instrumentation/analog-discovery-2/start

[39] Sara R5 datasheet (page 25) https://www.u-blox.com/sites/default/files/SARA-R5_DataSheet_%28UBX-19016638%29.pdf

[40] Waveforms Software https://store.digilentinc.com/digilent-waveforms/

was unable to be developed in time and the LwM2M implementation was not feasible on the board. It is also important to note that both the MQTT and CoAP current consumption implementations were tested with unencrypted data being sent and it is assumed that the addition of transport encryption would equivalently impact both MQTT and CoAP on the board implementation.

In the analysis section below, there are many graphs that compare the current consumption of MQTT (drawn with a blue line in all the graphs) and CoAP (drawn with an orange line) on the SARA R500S board. For a full size look at all of the graphs, see Appendix C.

Analysis

Overall, CoAP and MQTT had similar current consumption readings for sending the same amount of data. Due to the TCP-oriented nature of MQTT, there were additional packets of data that needed to be sent out to connect and disconnect from the server as well as maintaining the TCP connection.

The interesting thing to note is that, when the cell modem is powered on and connected to the network, there is a recurring pulse around every 1.2 seconds which may indicate the modem is pinging the nearby cell tower to maintain a connection. This pulse lasts for around 72 ms and averages around 80 mA during this time. A scope shot of this event is shown in the figure below. Due to this event, it may be worthwhile to turn off the modem when data is not being sent to prevent this constant current consumption event from occurring.

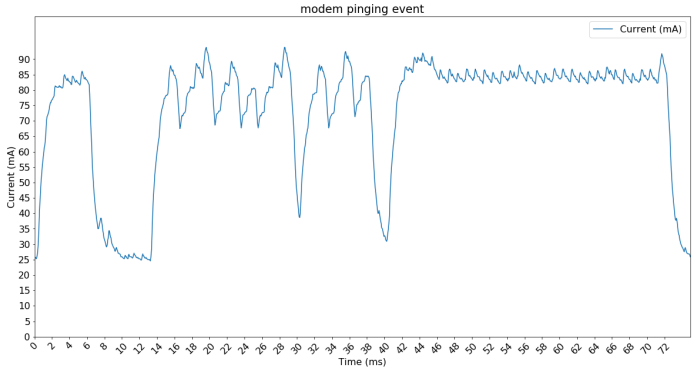


Figure 7.1 Current consumption trace of the modem pinging cell towers

As discussed in the Protocol Proof of Concept section, there were 5 major events of data scenarios that were sent. To effectively compare the current consumption of the different protocols, these events will be compared between CoAP and MQTT. Since MQTT must first send a connect request, this event is an additional consequence of running the MQTT protocol and thus this event is shown on its own below. This event lasted for around 5300 ms and peaked at 412.35 mA with an average current consumption of 80.01 mA during this time period. Along with the connection event, the MQTT implementation also has a corresponding disconnect event at the end of the final transmission which is effectively the same as the current trace shown in figure 7.2 below.

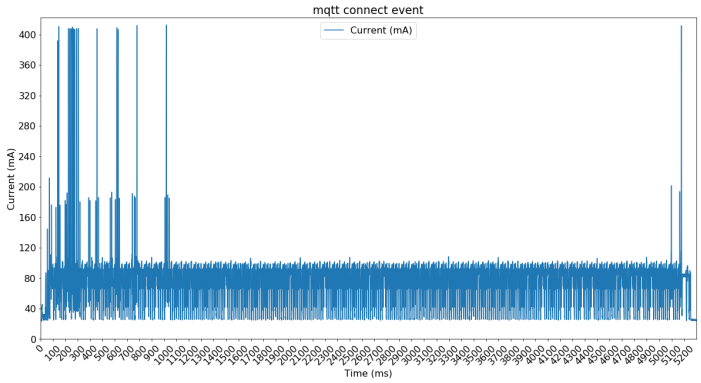


Figure 7.2 Current consumption trace of MQTT connecting to the server

The first comparison event is the "single event" which contained around 80 characters to be sent via MQTT and CoAP. The graph in figure 7.3 shows the difference in current consumption between sending this initial basic event. It is clear that the coap event is shorter and thus consumes less current during the sending of the 80 bytes to the main server. Overall, the current spikes to around 400 mA for both protocols and both protocols create a similar pattern in terms of a current spike at the beginning of the transmission and then a consistent leveling off at around 100 mA for the remaining byte transmissions. It is likely that the MQTT event lasts longer due to the TCP connection that has to be set up and torn down whereas CoAP is running over a UDP connection. The MQTT event lasted about 4950 ms and the CoAP event lasted about 4470 ms which is not a large difference but could be impactful in the long run for power considerations. The MQTT event peaked at 412.35 mA and averaged 80.8 mA while the CoAP event peaked at 410.30 mA and averaged 79.9 mA during the single event.

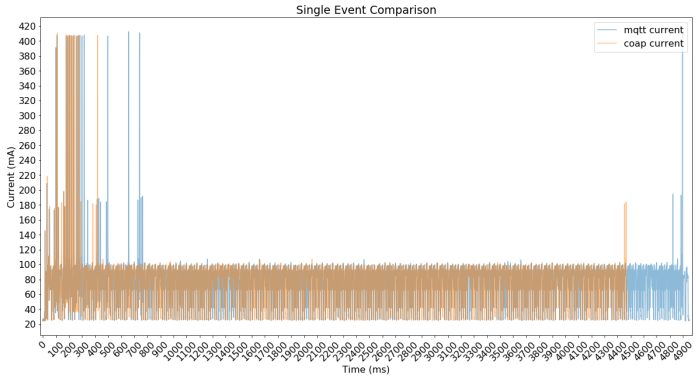


Figure 7.3 MQTT vs CoAP comparison for a single event data transfer

The next event was a "single event with data" which was around 93 total data bytes transmitted. This event was similar to the previous event, but just adds a few more bytes in transmission. We can see that the current comparison in the graph in figure 7.4 is quite similar to the figure 7.3 in that the MQTT event lasts slightly longer than the CoAP event due to the TCP connection. The MQTT event lasted for about 5000 ms, had a peak current value at 411.67 mA, and averaged 80.40 mA. The CoAP event lasted for about 4650 ms, had a peak current value at 412.8 mA, and averaged 81.81 mA.

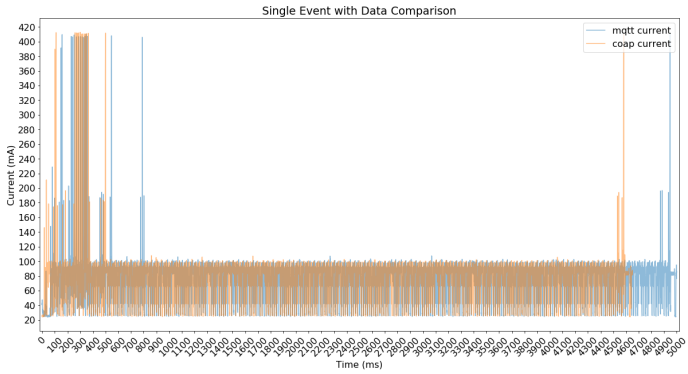


Figure 7.4 MQTT vs CoAP comparison for a single event with data transfer

The next event was a "single location event". This event was around 320 bytes, so this event will be slightly longer due to the extra bytes that have to be sent. The MQTT event again extends a little longer which is the common theme between each of these events. The MQTT event lasted about 5300 ms, peaked at 411.21 mA, and averaged about 85.48 mA during this transmission. The CoAP event lasted for about 4750 ms, peaked at 413.25 mA, and averaged 85.76 mA.

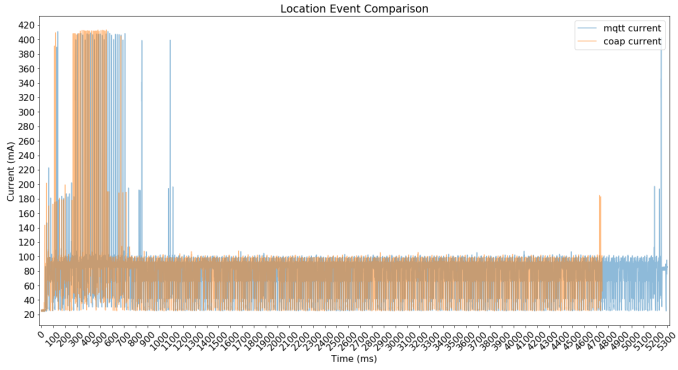


Figure 7.5 MQTT vs CoAP comparison for location event data transfer

The next event to be sent was a multiple events event which was intended to simulate the situation in which the board has buffered a few events to be sent

collectively in one transmission. The size of this transmission was around 550 bytes which is reflected in the CoAP transmission since there are the AT limitations of 512 octets that can be used with the CoAP command. Therefore, the CoAP transmission of this event was longer since it had to send the overall message as two separate messages whereas the MQTT command was able to handle it all in one transmission.

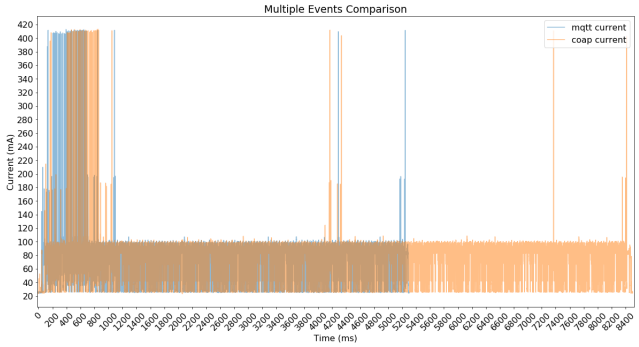


Figure 7.6 MQTT vs CoAP comparison for sending multiple events

The final event that was used to evaluate the protocols was a four hours of GPS data dump from the board. This was significantly larger than the other events, due to the nature of this data payload, at around 100 kB of data to be sent. Therefore, CoAP performed significantly worse than MQTT since it had to chunk up the data into smaller pieces (512 octets) which takes the board longer to send whereas the MQTT board implementation was able to handle larger payload sizes (1024 octets).

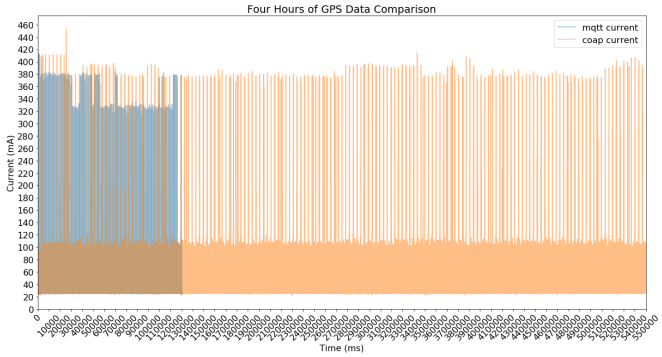


Figure 7.7 MQTT vs CoAP comparison of four hours of gps data transmission

Current Consumption Metrics Summary

The following table shows a breakdown of the valuable metrics taken from the current consumption analysis section. This is intended as a quick breakdown to give a good view on how heavy MQTT and CoAP are when implemented on the SARA R500S board. The average current value was calculated using the root mean squared (RMS) calculation during the event.

Event Name	MQTT	CoAP
MQTT Connect Event		
average current value	80.02 mA	n/a
peak current	412.34 mA	n/a
event length	5300 ms	n/a
Single Event		
average current value	80.81 mA	79.93 mA
peak current	412.35 mA	410.30 mA
event length	4950 ms	4650 ms
Single Event with Data		
average current value	80.40 mA	81.81 mA
peak current	411.67 mA	412.80 mA
event length	5000 ms	4650 ms
Location Event		
average current value	85.48 mA	85.76 mA
peak current	411.21 mA	413.25 mA
event length	5300 ms	4750 ms
Multiple Events		
average current value	87.98 mA	83.86 mA
peak current	412.80 mA	413.26 mA
event length	5300 ms	8500 ms

4 Hours of GPS Data		
average current value	83.99 mA	80.57 mA
peak current	413.48 mA	452.76 mA
event length	130700 ms	550002 ms
MQTT Disconnect		
average current value	87.18 mA	n/a
peak current	471.38 mA	n/a
event length	4680 ms	n/a

Recommendations and Next Steps

The next steps for BI Inc. is to evaluate the needs of their applications to determine which protocol would be best for their products. Due to our limited knowledge of the products/applications that BI develops, we can only provide situational recommendations on which protocol would be best to utilize.

Overall, the protocols analyzed have different use cases and applications that are best suited depending on the needs of the program/products. Since the products that BI develops are targeting constrained low-power devices, it is evident that AMQP is not a strong choice to use. This recommendation comes from the difficult nature to implement/integrate the protocol on the U-Blox boards as well as the heavy overhead that is built into the protocol specification. While AMQP has a substantial amount of use in many applications, it falls short on low-power situations due to it being quite heavy on the number of messages/size of packets required to implement which was seen in the AMQP overview as well as in the packet capture analysis section.

MQTT and CoAP are both strong choices for targeting low-power constrained devices. As seen in the above analysis, CoAP performs better than MQTT on the SARA R500S board for messages that are fairly small (< 512 octets) and do not need to be chunked into separate payloads. From the packet capture analysis, we also see that CoAP consistently uses less bytes to send the same amount of data due to the low overhead nature of UDP.

The other consideration to keep in mind is the server side implementation for maintaining the data that is being sent by the devices. For MQTT, there would need to be a separate client application running that is subscribed to all of the topics, or different distributed processes set up that would handle being subscribed to each topic to retrieve all of the data coming in from devices. Persistent connections would need to be used in this setup to ensure that data is not lost from devices sending data to the server. On the other hand, the CoAP server implementation is REST based and contains resources which are retained and handled on the server end. This means that when devices send data to update a new resource, as long as the server is up, that data will be accepted and stored. Therefore, only the server needs to be robust instead of maintaining processes to subscribe to topics. This is one advantage of using CoAP, where the server will handle and store all of the incoming data from the devices.

In this sense, CoAP becomes quite an attractive choice for implementing on the U-Blox boards. The downside of utilizing CoAP is that it is a much newer protocol specification when compared to MQTT which may incur some penalties for less support/documentation since it is not dominating the market as much as MQTT is. CoAP also does not have a dominant server implementation like MQTT has the mosquitto broker framework which may cause the server-side implementation to be more difficult to work with for CoAP.

Of course, this does not need to be a single choice selection. Both MQTT and CoAP can be combined in a

joint effort for different data/events. In a possible case, MQTT could be used to send large payloads and CoAP could be used when the payload is reasonably small (meeting the byte limits for a single coap command on the board). This would give a sort of "best of both worlds" scenario from a current consumption perspective. However, this combination of protocols would make an administration application a bit more complicated as it would need to acquire data from both a CoAP server as well as from the MQTT subscribers.

LwM2M is also a strong consideration for a device management protocol. Due to the different nature of LwM2M being primarily a device management protocol and not designed to be more of a data transfer protocol (like MQTT and CoAP), this protocol would assist in firmware updates on the device side. A device management system is likely vital in the pipeline rather than requiring people to come in with their devices to update them manually. Therefore, it is highly recommended to look further into LwM2M development on the U-Blox board to assist in the device management process. It is possible that MQTT and CoAP could be used to develop a business based framework for updating boards over those protocols. However, LwM2M is a specification already created over CoAP to facilitate these needs.

It is recommended to take the research conducted in this paper and conduct an internal BI discussion to make the best possible choice based on the needs of the products. A blended approach utilizing both CoAP and MQTT could be the most viable option for the applications since it would get the best of both protocols where their use would depend on the data sizes that need to be sent.

Appendices

Appendix A (Libraries and Software Implementations)

A.1 (Python Serial Libraries)

Python was used to create a serial connection to the SARA R500S board and run AT commands. The library package pyserial was used to make these serial connections and the documentation can be found here: https://pythonhosted.org/pyserial/. An example of how to use this library package can be found in the python scripts listed under our github repository here: https://github.com/ckardaras/iot/tree/master/Python_Testing.

A.2 (MQTT Libraries)

The most common MQTT server is mosquitto[41] which comes fully equipped to handle the latest MQTT standard. For client based operations, the Paho library was developed which has language support in python as well as for embedded application in c/embedded c. See a full list of client MQTT libraries here: https://github.com/mqtt/mqtt.org/wiki/libraries.

A.3 (AMQP Libraries)

The most common package that is used for an AMQP server implementation is RabbitMQ[42]. There are a plethora of tutorials that can be used to set up the RabbitMQ server and utilize some clients for connecting and sending data to the server. It is important to mention that a user needs to be configured if a client outside of the localhost domain is to properly access the RabbitMQ server. Pika is the primary python library developed by the RabbitMQ team to set up a computer client to connect to a server and tutorials for this can be found here: https://www.rabbitmq.com/getstarted.html.

A.4 (CoAP Libraries)

There are many libraries available for CoAP however, we did our testing using the CoAPthon library to build our server. CoAPthon is a python implementation of CoAP. The library includes code for: running a CoAP server, running a CoAP client, implementing a CoAP-to-CoAP Forward Proxy, implementing a CoAP-to-CoAP Reverse Proxy, implementing an HTTP to CoAP Forward Proxy, Using the Caching Feature, Using the Observe Feature, CoRE Link Format parsing: “Web Linking using a link format for use by constrained web servers to describe hosted resources” (RFC 6690), Multicast Server Discovery, and doing the Block-wise transfers.

This implementation does not have DTLS, for an example of a server that uses DTLS see https://github.com/Tanganelli/CoAPthon/blob/master/test_secure.py. Coap UDP typically runs on port 5683, with DTLS it runs on port 5684. There are 5 classes of libraries, libraries for constrained devices, server-side libraries, browser based implementations, libraries for smartphones (both iOS and Android), and commercial implementations. There are so many options for libraries, it comes down to exactly what extensions you want, or what languages you wish to write your implementations in. For a full list of available libraries, see https://coap.technology/impls.html.

[41] Mosquitto Broker Main Page: https://mosquitto.org/
[42] RabbitMQ main page: https://www.rabbitmq.com/

A.5 (LwM2M Libraries)

Eclipse Leshan https://github.com/eclipse/leshan is a very popular and widely used OMA Lightweight M2M library, with server and client implementations in Java.

A.6 (Github Repository for this Project)

The github repository can be found here: https://github.com/ckardaras/iot which contains all of the testing scripts and python files created to evaluate the different IoT protocols with the SARA R5 board by U-Blox. Under the Python_Testing folder, there are plenty of examples and classes that can be used for sending AT commands to the SARA-R5 board for MQTT and CoAP and sending general AT commands. Packet capture files gathered from the protocol proof of concept section can be found/analyzed further here: https://github.com/ckardaras/iot/tree/master/Testing_Results/Packet_Captures. Analysis for the current consumption measurements was performed in a jupyter notebook and the related files can be found here: https://github.com/ckardaras/iot/tree/master/Testing_Results/Current_Consumption_Analysis.

Appendix B (Current Consumption Evaluation Setup)

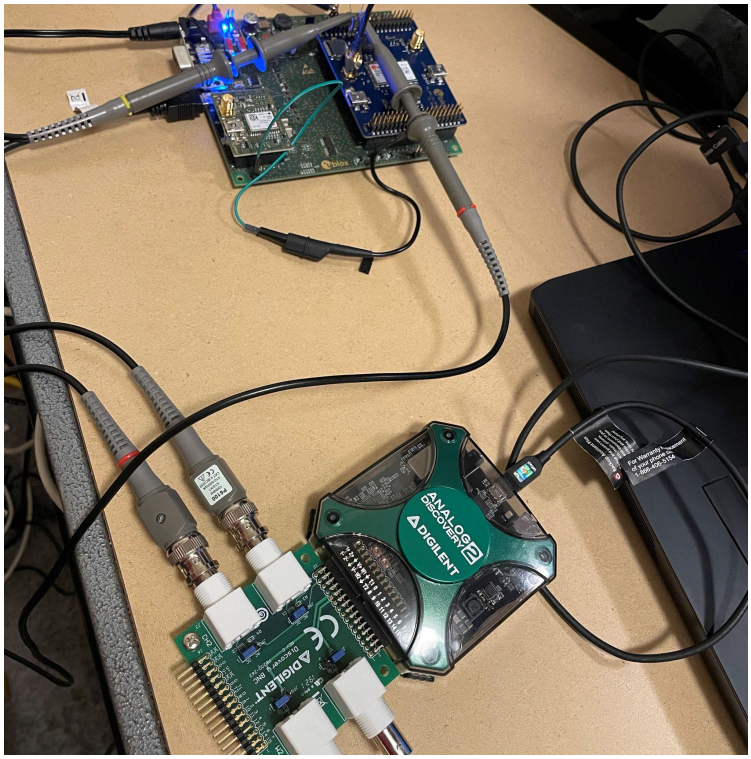


Figure B.1 *AD2 setup with probes connected across pin J109 on SARA R500S*

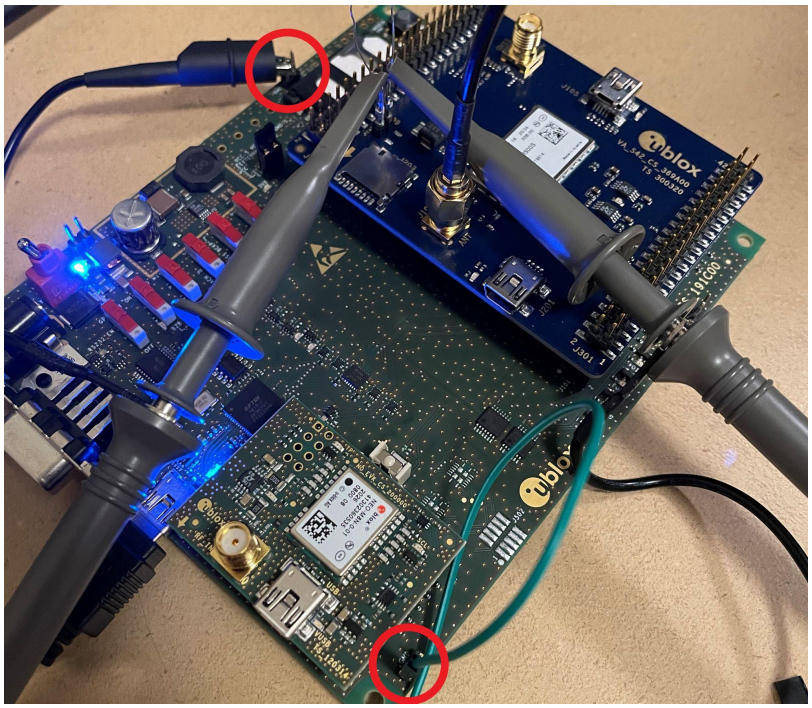


Figure B.2 *Utilizing the ground pins on the edges of the boards for the signal probes*

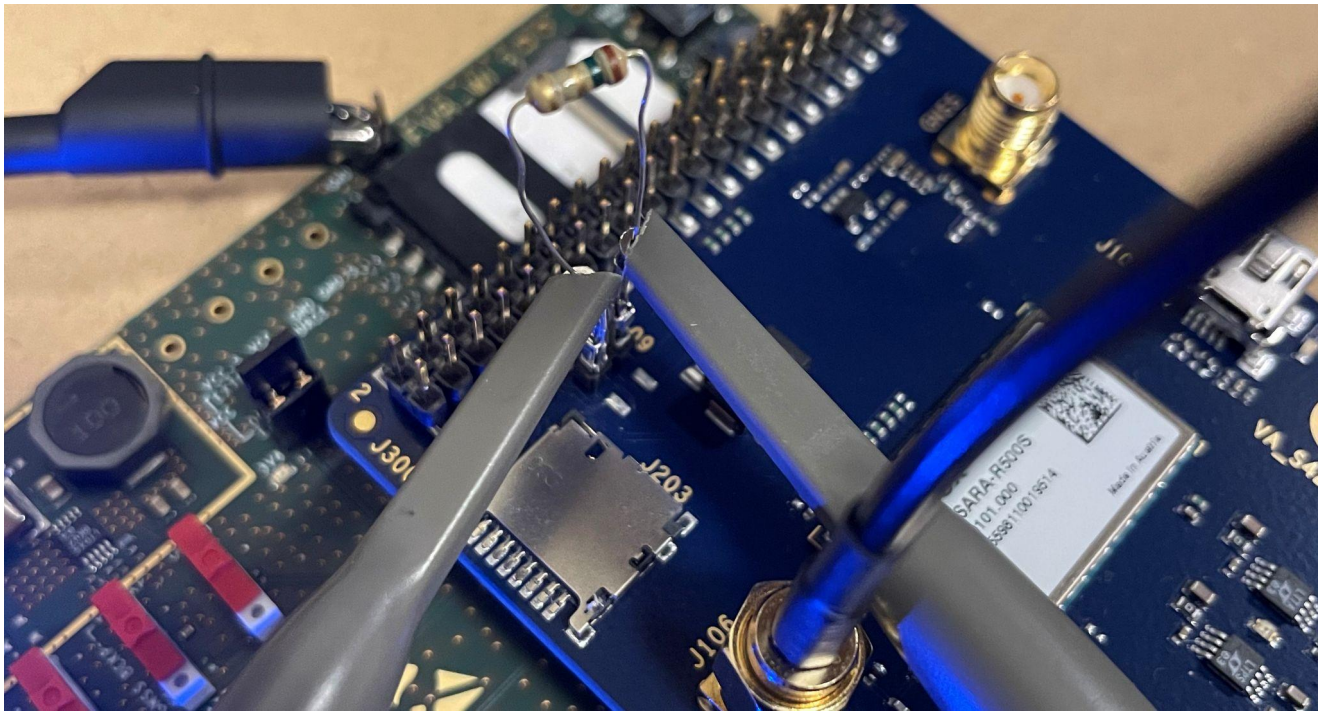
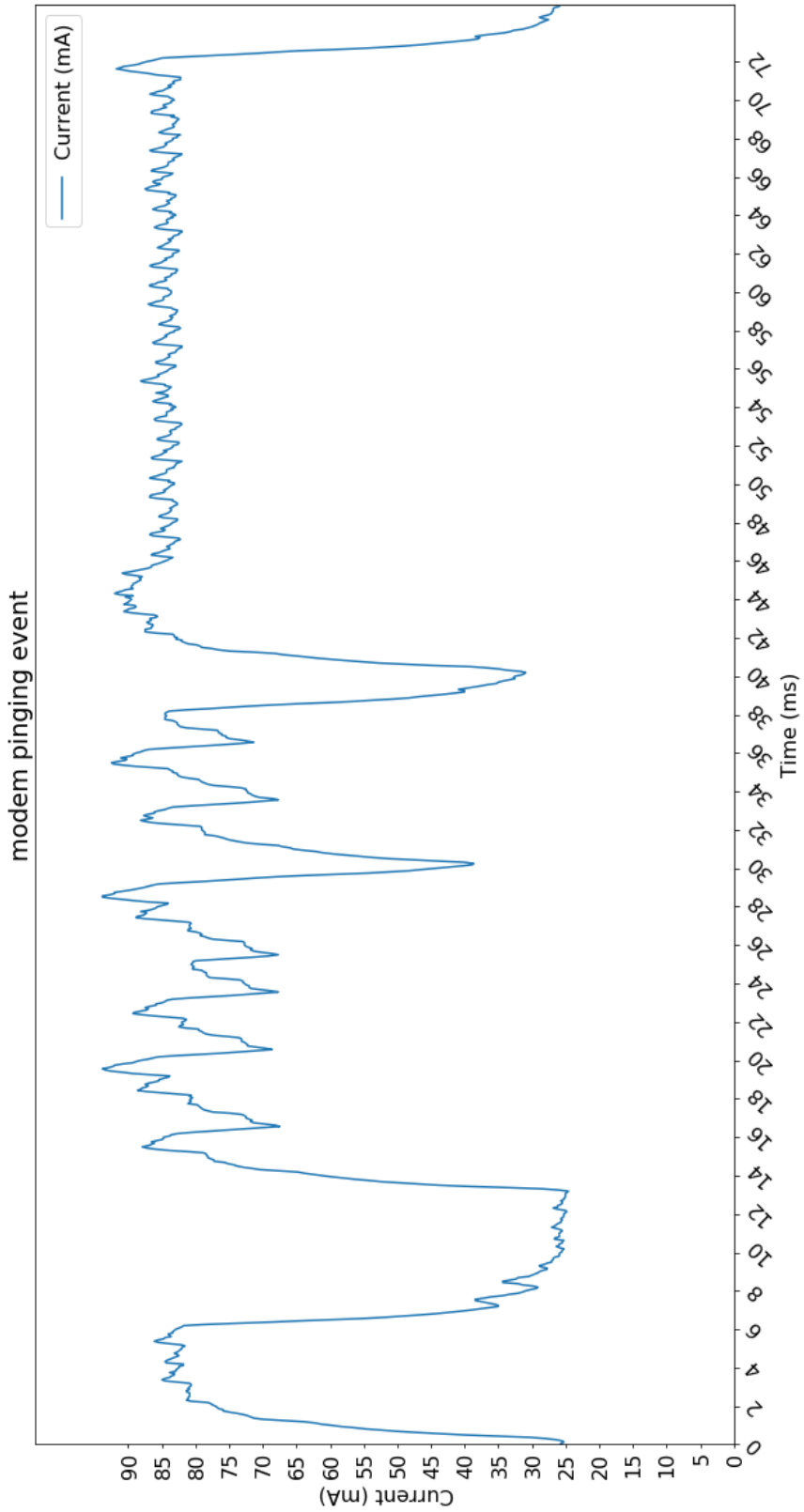
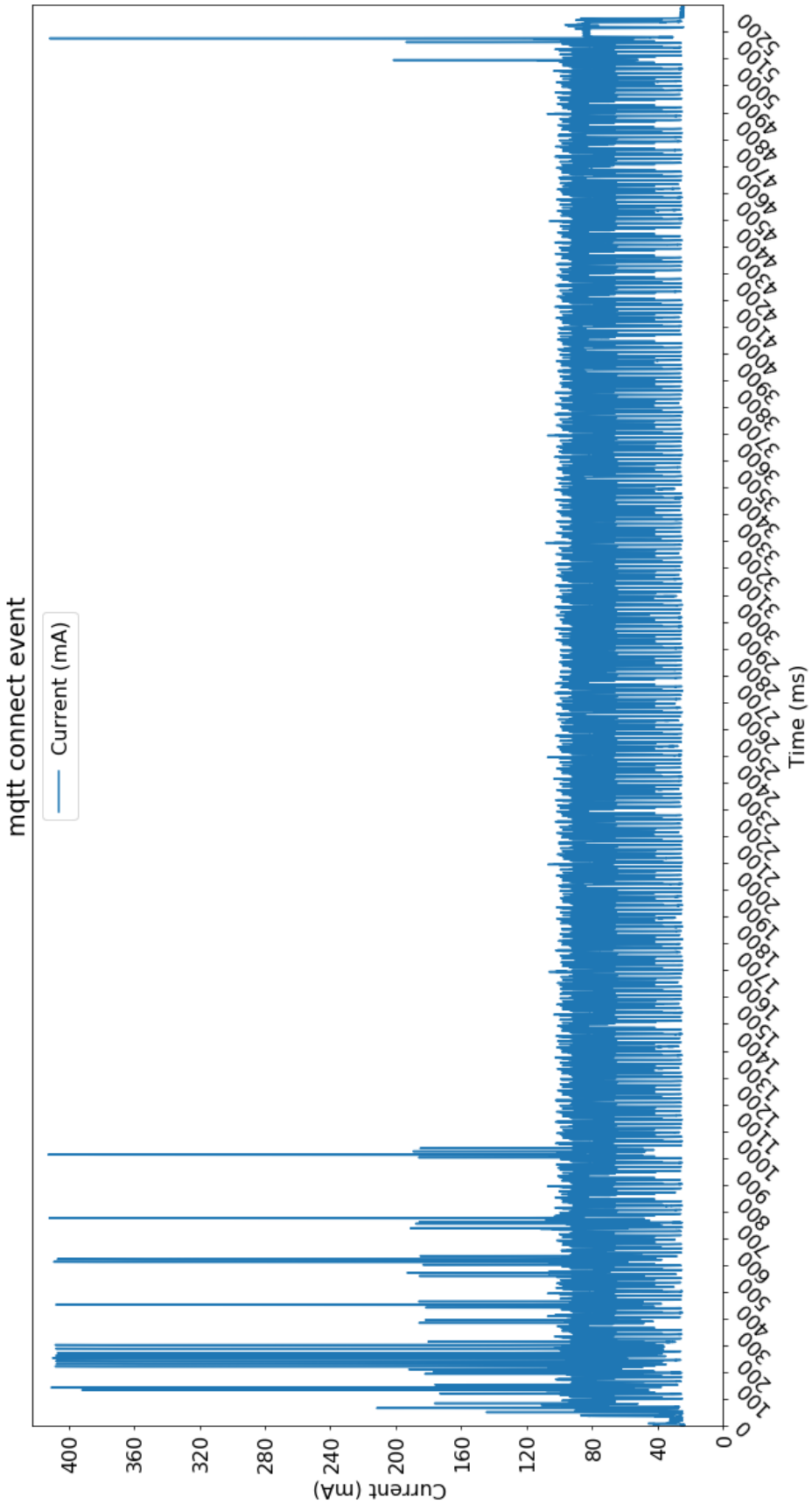
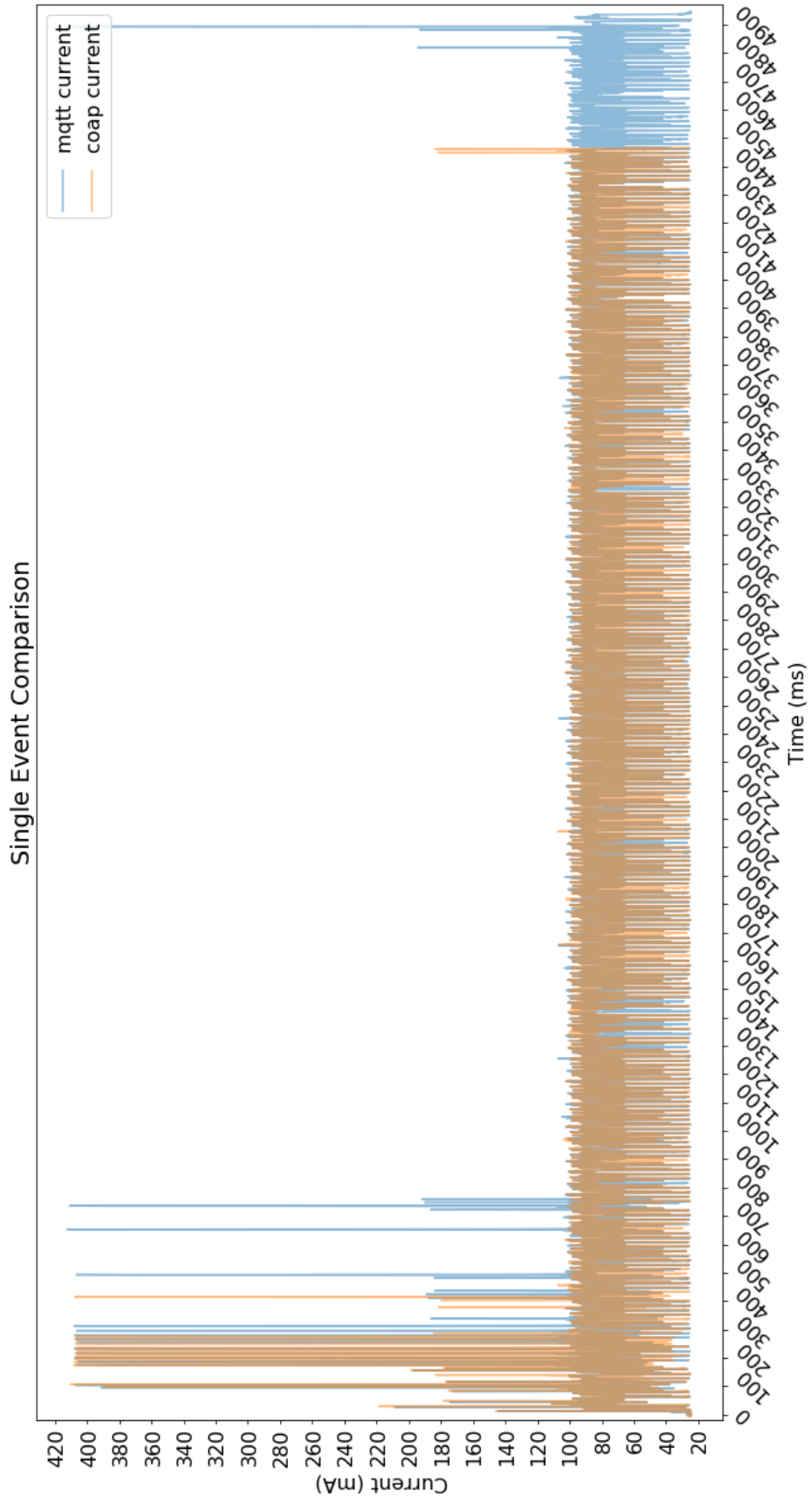


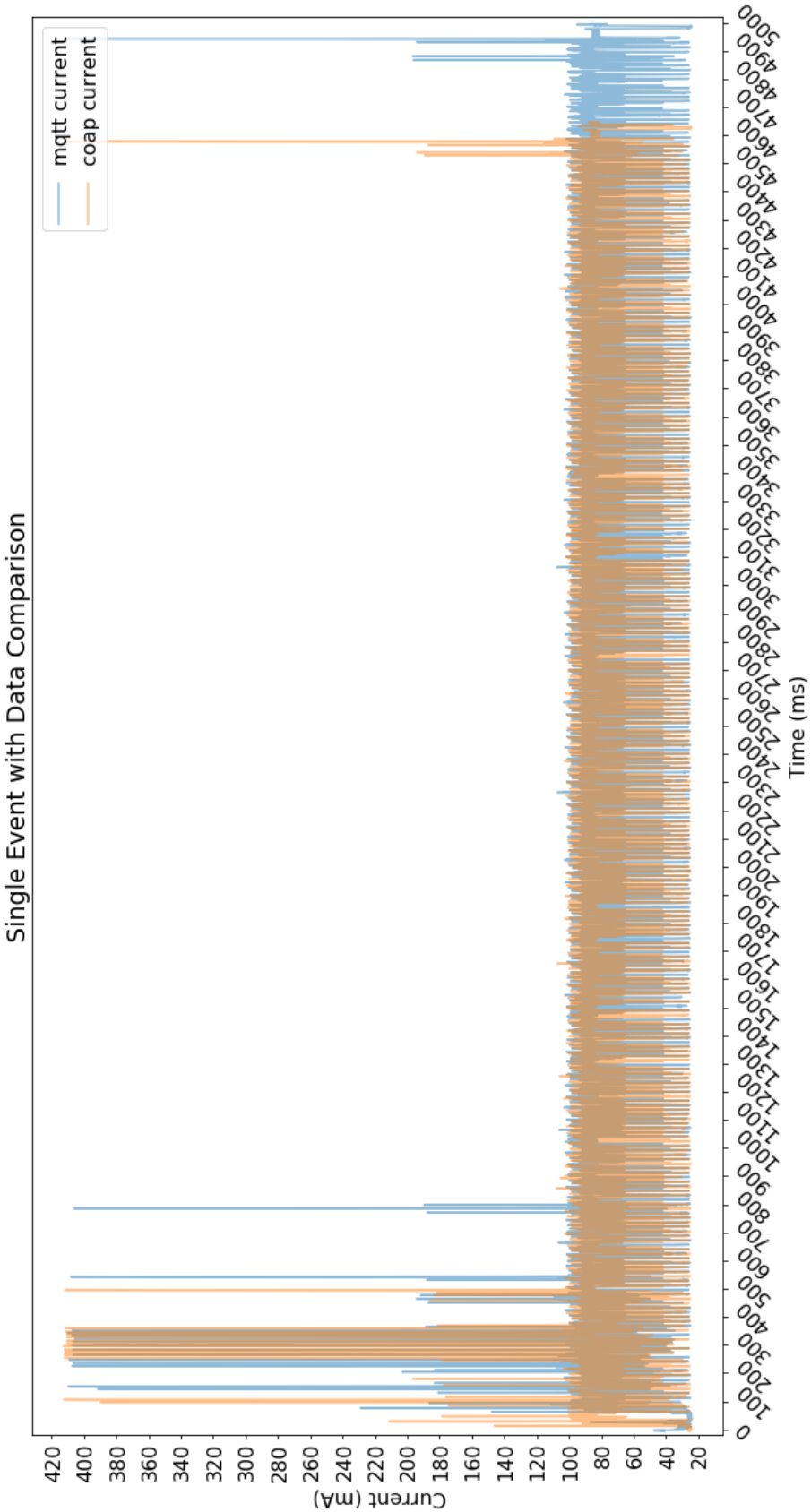
Figure B.3 *1.5 Ω shunt resistor on J109 pin and signal probes across the resistor to read voltage drop*

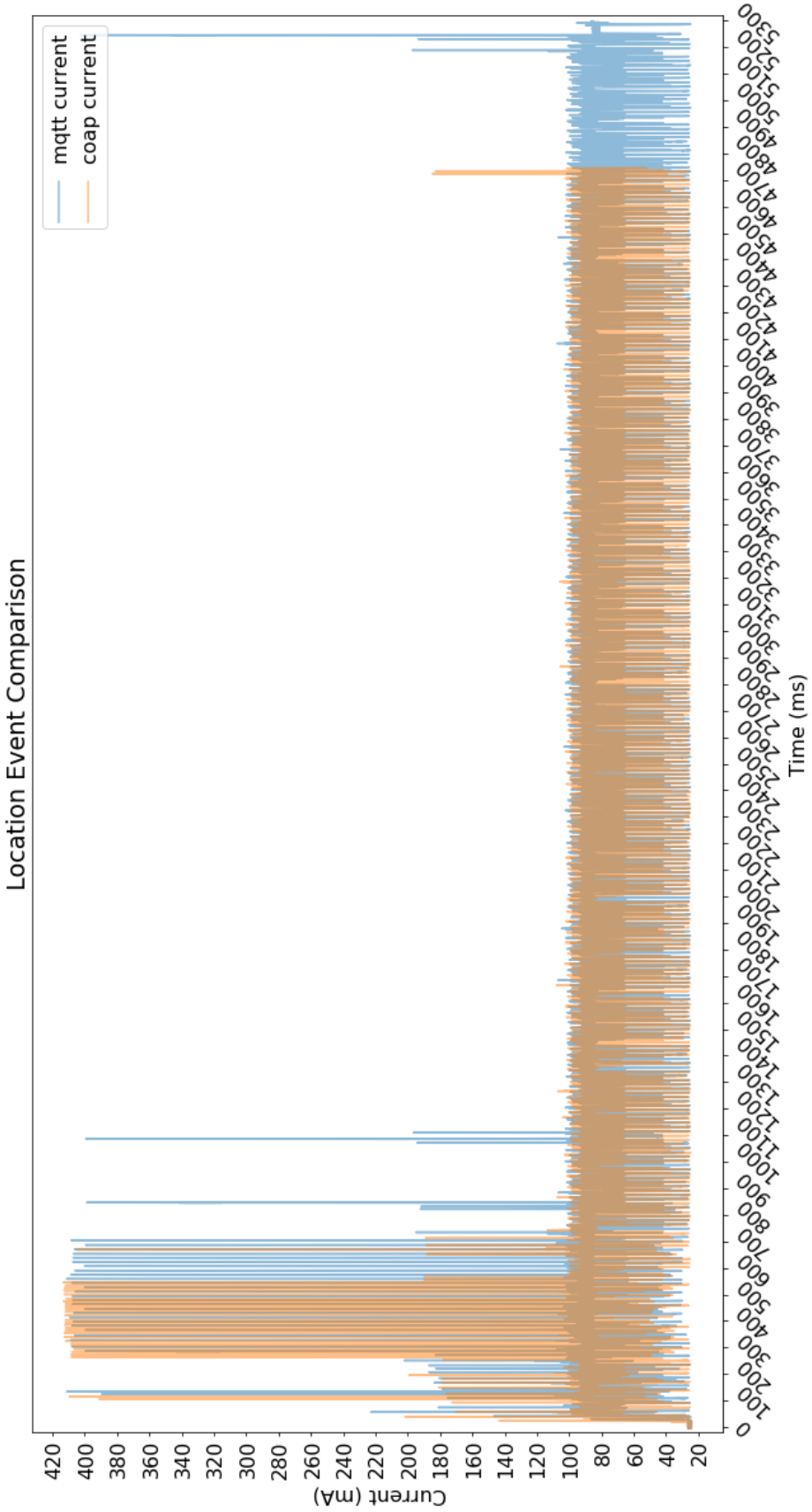
Appendix C (Full Sized Current Consumption Comparison Graphs)

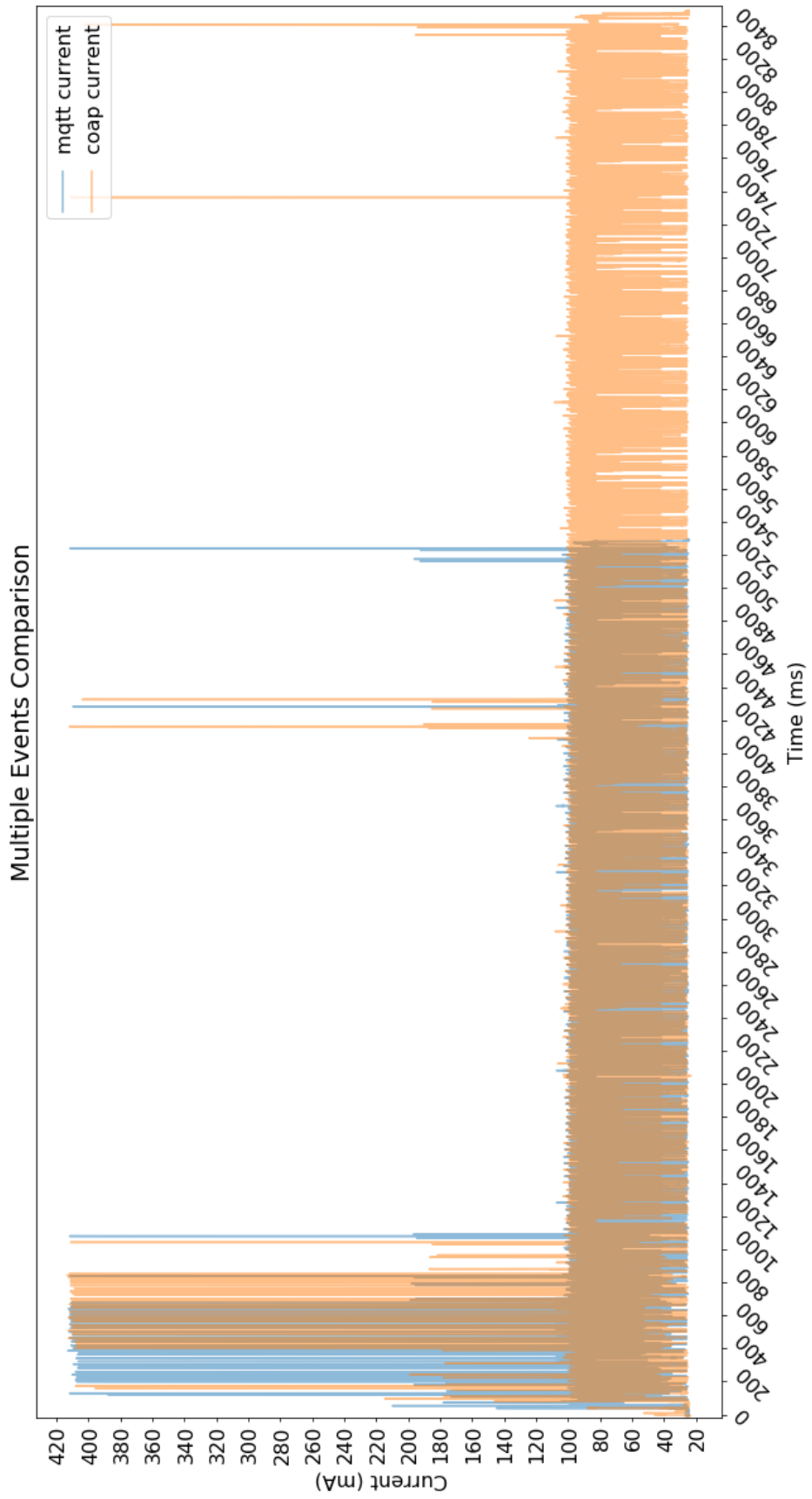


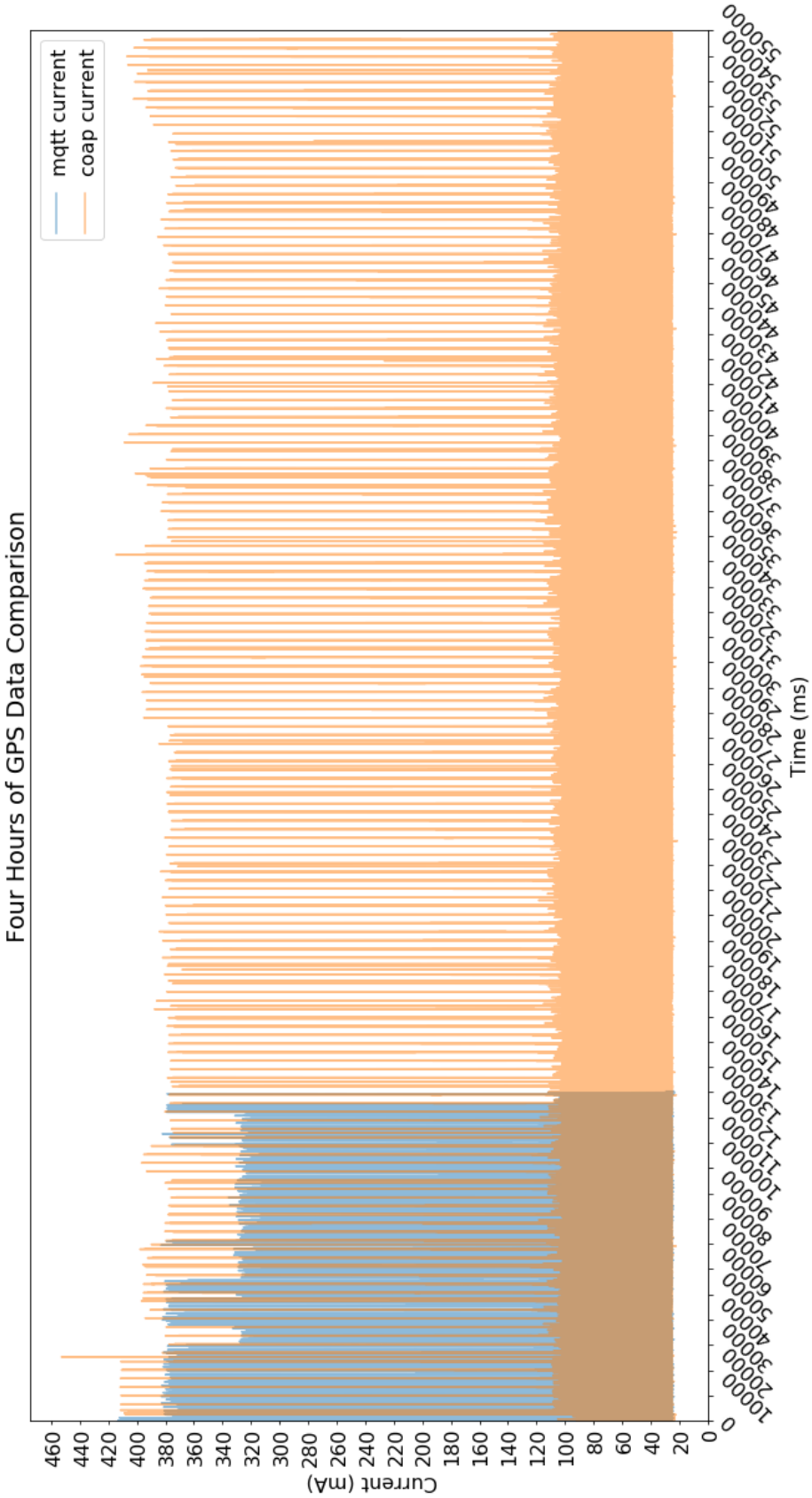

mqtt connect event
Current (mA)
Current (mA)
Time (ms)
0
40
80
120
160
200
240
280
320
360
400
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
3000
3100
3200
3300
3400
3500
3600
3700
3800
3900
4000
4100
4200
4300
4400
4500
4600
4700
4800
4900
5000
5100
5200

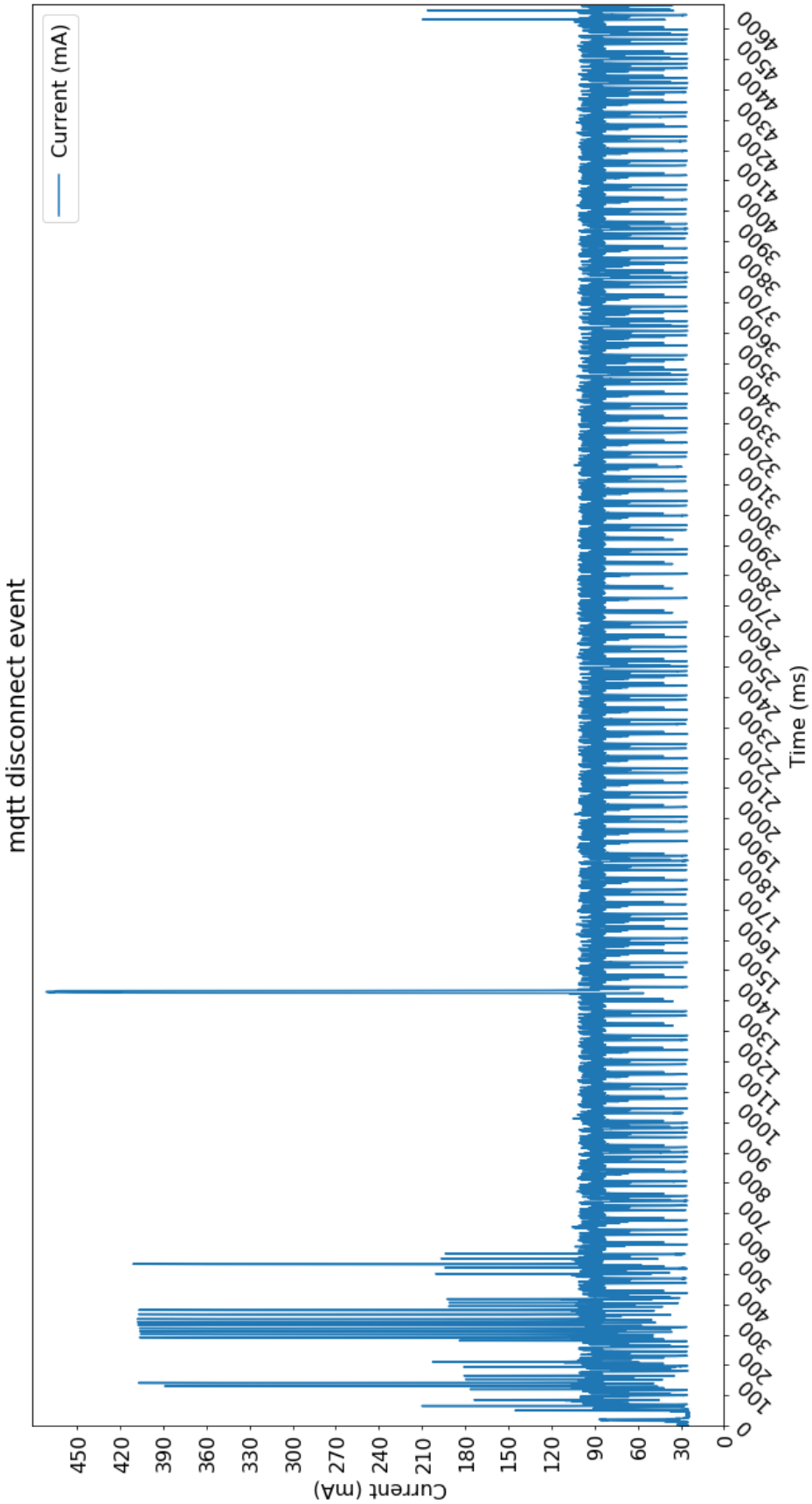

Single Event Comparison
mqtt current
coap current
Current (mA)
Time (ms)


Single Event with Data Comparison
mqtt current
coap current
Current (mA)
Time (ms)


Location Event Comparison
mqtt current
coap current
Current (mA)
Time (ms)


Multiple Events Comparison
mqtt current
coap current
Current (mA)
Time (ms)


Four Hours of GPS Data Comparison
mqtt current
coap current
Current (mA)
Time (ms)


mqtt disconnect event
Current (mA)
Current (mA)
Time (ms)
0
30
60
90
120
150
180
210
240
270
300
330
360
390
420
450
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900
2000
2100
2200
2300
2400
2500
2600
2700
2800
2900
3000
3100
3200
3300
3400
3500
3600
3700
3800
3900
4000
4100
4200
4300
4400
4500
4600