SCTE CABLE-TEC EXPO FAST FORWARD 2021
VIRTUAL EXPERIENCE
OCTOBER 11-14

2021 Fall Technical Forum
SCTE • NCTA • CABLELABS

Message Queuing Telemetry Transport (MQTT) For IoT Devices: Less is More

A Technical Paper prepared for SCTE by

Sweety Bertilla
Senior Android Engineer
Comcast Cable
518 Highland Ave, Cherry Hill, NJ, 08002, USA
+1-267-785-3798
sweetybertilla_francisxavier@cable.comcast.com

Kristopher Linquist
Principal II Engineer
Comcast Cable
1050 Enterprise Way #100, Sunnyvale CA, 94089
+1-408-940-5747
kris_linquist@comcast.com

Robert Farnum
Principal II Engineer
Comcast Cable
13029 Titus Court, Austin, TX, 78732, USA
+1-512-860-6166
robert_farnum@comcast.com

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

Table of Contents

Title **Page Number**

List of Figures

Title **Page Number**

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

List of Tables

Title **Page Number**

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

1. Introduction

Message Queuing Telemetry Transport (MQTT) is a well-designed, lightweight messaging protocol that can be used for communication between mobile clients, microservices, and IoT devices. Unlike HTTP (Hypertext Transfer Protocol) and other messaging protocols, MQTT is a low bandwidth, low latency alternative for IoT device transmissions, which is far more suitable because these devices may operate within tiny bandwidth, power, and transmission footprints. MQTT uses publish/subscribe operations to exchange data between client and server -- meaning an IoT device (or any other client) "subscribes" to a topic and asynchronously receives messages when data is published on that topic.

Also, unlike HTTP, this method saves a substantial amount of time previously spent on polling, which makes updates occur more quickly and smoothly. The lightweight nature of MQTT helps the end user – or customer - receive messages even when they are in low bandwidth situations, such as when traveling in areas with limited connectivity. Quality of Service (QoS) features supported by MQTT help clients opt into the level of service based on network reliability.

In this paper, attendees will learn how and why Comcast opted to adopt the MQTT protocol in customer-facing applications such as the Xfinity Application when connecting and bridging communications with IoT devices such as Philips Hue, LIFX, Ecobee, August door looks, and other Zigbee devices (via a Residential Gateway) using the AWS IoT Core as the MQTT message broker.

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

2. History of MQTT

MQTT was created in 1999 by Andy Standford-Clark (IBM) and Alen Nipper (Arcom, now Eurotech) as part of the IBM MQ series of products. The goal was to invent a new protocol for connecting oil pipelines over unreliable satellite networks. In 2011, IBM and Eurotech donated MQTT to the proposed Eclipse project called Paho. In 2013, MQTT version 3.1 was submitted to OASIS (Organization for the Advancement of Structured Information Standards). On October 29, 2014, MQTT 3.1 was approved. Since then, MQTT 3.1.1, currently the most supported version, was ratified in 2016, and MQTT 5.0, which supersedes MQTT 3.1.1, was ratified in 2019. MQTT is also published as an ISO/IEC 20922 standard reference.

A frequent question concerning MQTT that arises is, "What does the acronym stand for?" The answer is debatable. Given its source, the "MQ" in "MQTT" is historically based on the name of the IBM "MQ" Series product line. Others say "MQ" means message queueing - but this is a misnomer, as MQTT is not a message queue, but *can* queue messages for clients. Unfortunately, the OASIS technical committee named itself "OASIS Message Queuing Telemetry Transport Technical Committee". This is likely the source of the acronym's more common definition. In any case, the messaging protocol is no longer really an acronym, but is simply recognized by the letters "MQTT."

Given the maturity and design of MQTT as a lightweight messaging protocol, it has been widely adopted for MTM (machine to machine) communications for industrial, messenger, and IoT applications. MQTT's ability to keep bandwidth requirements to a minimum -- while dealing with high latency, unreliable networks, small footprint devices, and low power consumption -- has made it an excellent choice for communications over a variety of networks between clients and/or devices with high cardinality to cloud-based platforms.

Other protocols, such as, HyperText Transfer Protocol (HTTP), Advance Message Queuing Protocol (AMQP), eXtensible Messaging and Presence Protocol (XMPP), or Websockets, while often considered as potential solutions, have fallen short of meeting some of the basic selection criteria. MQTT continues to edge out other options because of its nature as a lightweight, asynchronous, bi-directional, secure, efficient, reliable, publish/subscribe, and data-agnostic messaging protocol.

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

3. Advantages of MQTT

3.1. Asynchronous bi-directional communication

Asynchronous APIs matter in that they can provide significant performance and responsiveness in client-server communications. This mechanism is inherently *non-blocking,* which means that an application does not need to wait for a response in order to proceed. In the context of an application showing the status of IoT devices, a list of devices can be shown – and updates to their status will be received as devices report them, rather than pausing to wait for all devices to respond.

Messages can also be pushed directly to the client without an explicit request. Downstream microservices can choose to directly send updates to a client application – an ability they may not have had if they communicate only to an upstream synchronous API. This is useful for status reporting, user messaging, and more.

For asynchronous APIs to work, the client must maintain a persistent connection with the server.

3.2. MQTT Features that make it an ideal protocol for async messaging

3.2.1. Publish/Subscribe

MQTT uses a publisher / subscriber messaging pattern to provide a framework for exchanging messages between publishers (producers) and subscribers (consumer clients). A publisher can send a message and it will automatically be routed – via the MQTT message broker – to any client subscribed to that topic.

3.2.2. Topic structure

MQTT uses a topic hierarchy to send and receive messages. This can be thought of as the equivalent of a path or route in a HTTP API, except that they do not have to be pre-defined.

MQTT topics can consist of wildcards and topic separators. An example topic may be:

platform / customerId4 / house1 / lights / lamp4

MQTT topic levels are separated by a forward slash (/) and subscribers can use two different types of wildcards when subscribing:

+ A plus represents a single topic level wildcard.

A hash is a multi-level wildcard that can only be at the end of a topic subscription.

If a client wants to subscribe to all of customerId4's devices located in house1, they will subscribe to:

platform / customerId4 / house1 / #

If a client wants to subscribe to all of customerId4's lights in ANY of their houses, they will subscribe to the topic:

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

platform / customerId4 / + / lights / #

These patterns allow for significant flexibility across many types of services.

The MQTT protocol has no limits on the number of subscriptions for a given connection.

3.2.3. QoS

Quality of Service (QoS) is a key feature of MQTT. An MQTT Client gets an option to select the level of service that works for their application, based on network reliability. MQTT manages the re-transmission of messages and guarantees delivery, even when the underlying transport is not reliable. QoS makes communication in unreliable networks a lot easier. There are 3 different levels of QoS:

QoS 0 is also known as "Fire and Forget service". An MQTT client sends the message to an MQTT broker and doesn't wait for an acknowledgement, so there is no confirmation if the message is received by the end client. However, this is considered the fastest message delivery service.

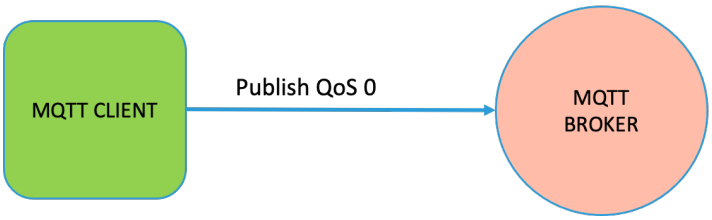

Figure 1 – Quality of Service 0(QoS 0)

QoS 1 makes sure that the MQTT message is delivered at least once to the end client. The publisher sends the message to MQTT broker and waits for an acknowledgement. If the publisher doesn't receive the acknowledgement within the given time, it publishes the message again with a duplicate flag (DUP). The MQTT broker sends the message to end clients and responds back with PUBACK packet to the publisher. The only drawback of this service is the possibility of end clients receiving the same message more than once.

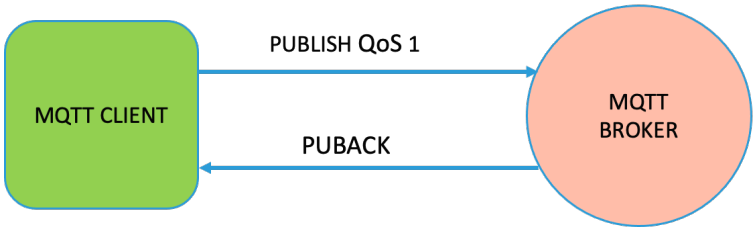

Figure 2 – Quality of Service 1 (QoS 1)

QoS 2 is also known as the safest and slowest message service. The publisher (MQTT Client) sends a QoS2 message to an MQTT broker and waits for acknowledgment. Once the publisher receives an acknowledgement (PUBREC) from the MQTT broker, it sends a PUBREL message

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

to the MQTT broker. The MQTT broker only sends the message to the end clients after it receives the acknowledgment (PUBREL) from publisher and sends back PUBCOMP to the publisher. Simply put, there is a four-way handshake that happens between a publisher and an MQTT broker to make sure the end client receives the message only once.

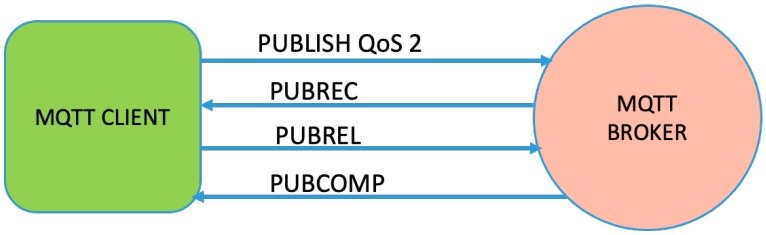

Figure 3 – Quality of Service 2(QoS 2)

The AWS IoT Core (MQTT broker) only supports QoS0 and QoS1 but not QoS2.

3.2.4. Last Will and Testament

Last Will and Testament (LWT) is a feature of MQTT that can notify other clients about a client that is disconnected without notice, often due to a lost network connection.

During the connect phase, a client registers the LWT with the broker. The broker then stores the LWT and publishes it if the client disconnects ungracefully.

LWT contains several parameters:

1. Last Will Topic (topic that the LWT message will be published to)
2. Last Will QoS (QoS of the LWT message)
3. Last Will Message (LWT message itself)
4. Last Will Retained (boolean) (whether the last lost connection information will be retained)

One example of usage in IoT devices is a *connected* flag. Immediately after connecting to a broker, an IoT device could send a *connected: true* message **and** set a LWT message that contains *connected: false*. If the IoT device's network connection is lost, the MQTT broker would send *connected: fals*e on the device's behalf, giving other clients real-time information as to whether it can expect a device to receive a command or update its status.

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

4. How to use MQTT for mobile or web application client

The MQTT message broker acts as medium for communication between the clients. Clients can be mobile or web application or an IoT device or microservice. Let's take a look at how a mobile application client can use MQTT for communication with IoT devices. Key actions include connect, disconnect, subscribe, unsubscribe, and publish. Key components include the MQTT broker, client(s) and topic(s).

Clients connect to the IoT services and devices through the MQTT broker using the connect method. Next, the client subscribes to topic(s) to receive messages. When a message is published to a topic, all clients that are subscribed to the topic will receive the message asynchronously, as depicted in Figure 4, below:

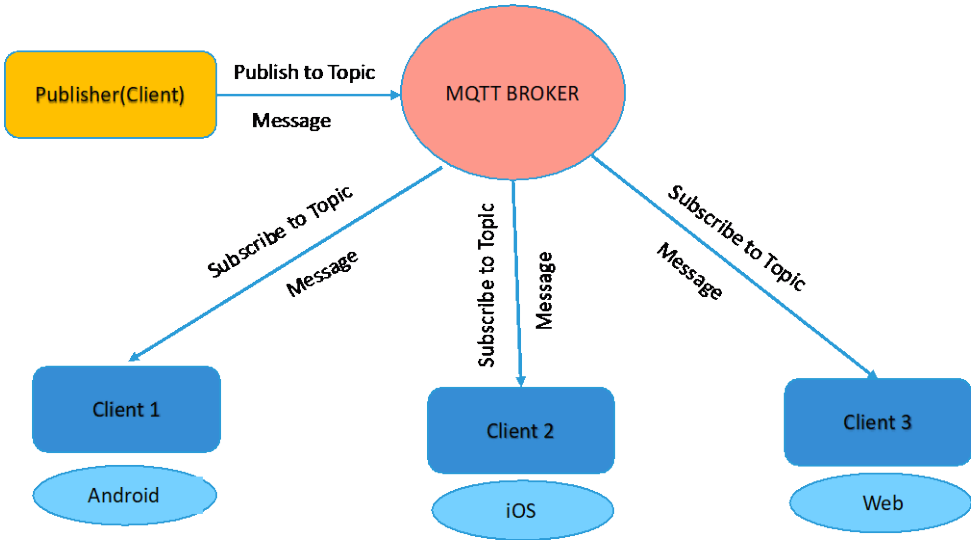

Figure 4 – High level diagram of MQTT

4.1. MQTT Connect

Application clients can establish a secure, persistent connection with an MQTT broker. Let's take a look at one of the MQTT brokers (used in Xfinity app) within AWS's IoT Core. IoT Core provides an open source SDK which can be used by the mobile client to integrate and consume the MQTT broker. AWSIotMqttManager, which is part of the SDK, provides an interface to make an MQTT connection. The AWSIotMqttManager requires the region, ClientID, and account endpoint, to which it uniquely establishes an MQTT connection.

The client ID provided by the application client is a unique identifier that represents the connection (one example could be a concatenation of a client account ID and app installation ID). "Region" is the AWS region identifier to which to connect (such as us-east-1 or us-west-2). The topic namespace is limited to an AWS account and namespace. The string accountEndpointPrefix indicates the specific customer endpoint. Example:
[prefix].iot.[region].amazonaws.com

AWS IoT Core allows a client to connect to the MQTT broker in four different ways:

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

1. Connect with keystore (secure location for storing cryptographic keys) and port number
2. Connect with AWS credentials provider
3. Connect with proxy host and proxy port.
4. Connect using custom authorizer. The IoT client of Xfinity Application uses custom authorizer to connect with MQTT broker. This custom authorizer is used to validate the customer against an internal identity provider.

The AWSIotMqttClientStatusCallback provides the status of the connection. Here is a sample of status call back details from a mobile client:

Connecting -> Trying to establish the connection Connected -> Successfully connection is established Reconnecting -> Trying to reconnect after a connection error ConnectionLost -> No longer client is connected to MQTT broker ConnectionError -> Error in the connection due to network or attempting to connect to a invalid topic

The attributes set as defaults can be altered by the application client. By default, the autoReconnect attribute is set to TRUE, which indicates that a reconnect attempt will be established when there is a connection error reported by the MQTT broker. A client can set the minimum and maximum reconnect time in seconds and maximum reconnect attempts. Another key attribute used is *clean session*. By default, clean session is set to TRUE, which means a non-persistent connection is established, which will not store any subscription information or undelivered messages for the client. By setting clean session to FALSE, a client can establish a persistent connection which will store subscription information and undelivered messages are delivered with QoS 1.

Table 1 – Clean Session with QoS

CleanSession Flag	QoS	Message received after Reconnect
TRUE	0	NO
TRUE	1	NO
FALSE	0	NO
FALSE	1	YES

4.2. Publish and Subscribe to Topic

Clients who subscribe to a topic will receive messages published to the topic, after a secure connection is established through MQTT broker. QoS level can be set to subscribe and publish topics. A client can also unsubscribe to topics before they disconnect. Using the AWS IoT Core, a client can publish messages as a JSON string or simple string.

4.3. MQTT and AWS Shadow State

The AWS Shadow State service, provided by the AWS IoT Core, is a persistent cache that can be used to store IoT device state. Reserved MQTT topics are provided to update this cache and to receive notification of create, update, and delete events. The two types of state in shadow state messages are *desired state* and *reported state*. Desired state lets the client send a request through an MQTT broker to the Shadow State service, where the Shadow State service calculates a *delta state*

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

(difference between desired and reported) and forwards the request to a device via an AWS IoT Core Rule. The device then attempts to honor the request, resets the desired state, and updates the reported state accordingly. The reported state is in turn returned via the device's MQTT topic to which the client is subscribed.

Here is a sample json payload that demonstrates desired and reported shadow state messages:

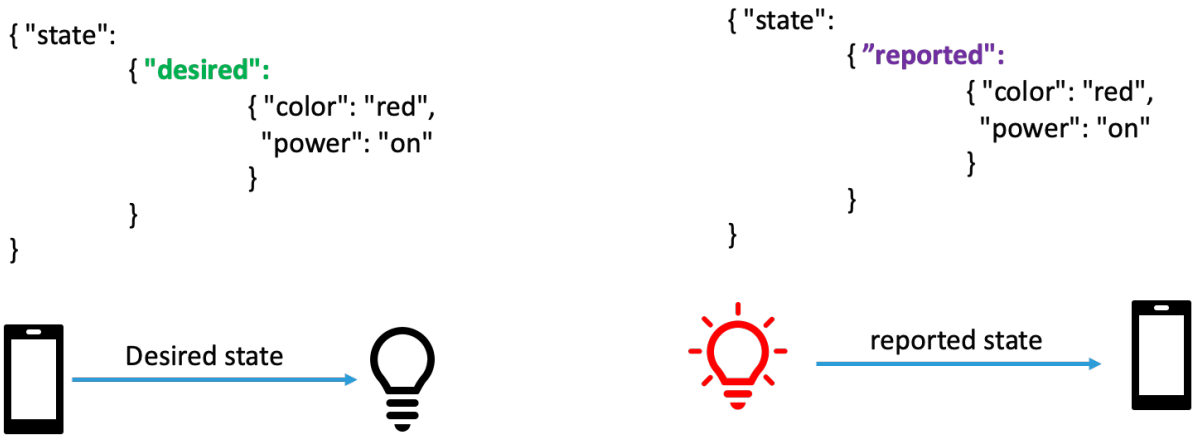

Figure 5 – desired and reported state messages

4.4. Message payload

Since MQTT is a lightweight protocol, the message payloads cannot extend in size. AWS's IoT Core supports a maximum payload of 128kb. In the Xfinity Application, the client receives the payload of one device at a time. MQTT messages also have no guaranteed ordering. Due to these limitations, we included s*equence number* and *total messages expected* to our MQTT message payloads to ensure all messages are received and ordered properly.

Shadow Reported state message payloads contain a property named *version*, which gets incremented with each change in shadow state. If the client receives a payload with version 5 and then a payload with version 4, the client can ignore the payload with version 4 as it is considered stale. Also, the desired state payload request sent by the client either needs to be same or higher version than the reported state, otherwise the AWS IoT Core MQTT broker rejects the desired state message with a *409 version conflict*. This feature, called optimistic locking, ensures that the client has the latest device status prior to trying to update the state.

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

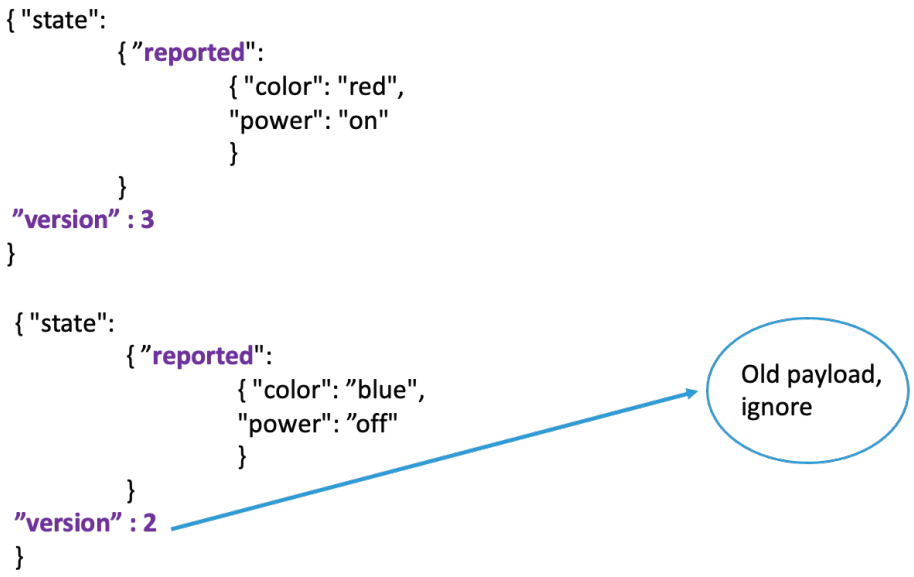

Figure 6 – reported state with old payload

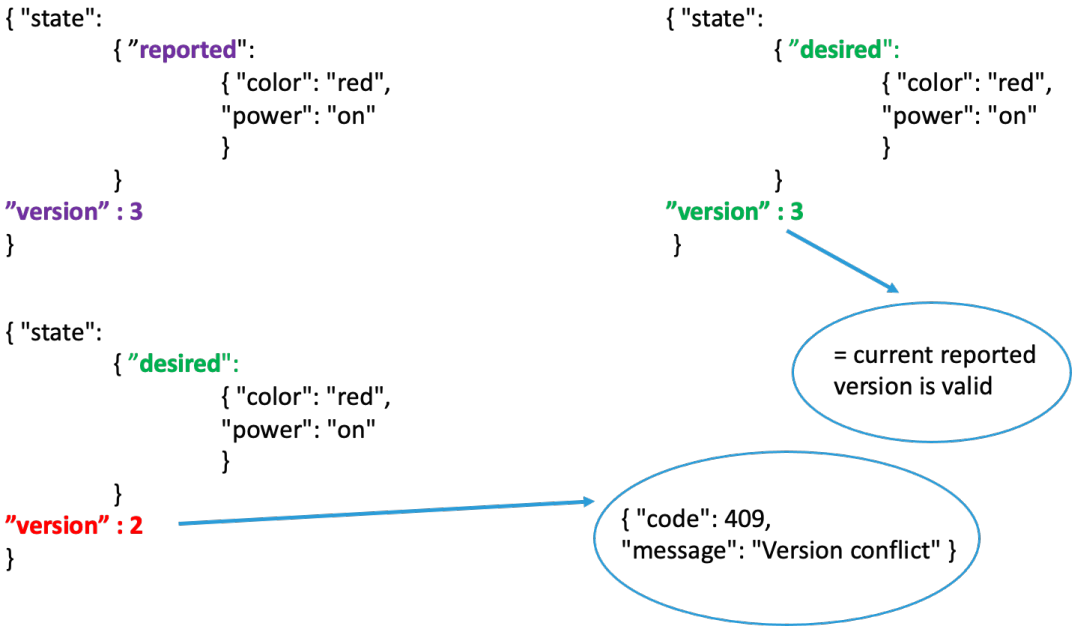

Figure 7 – desired state with previous version number

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

5. Security

The security of MQTT connections for a given subscriber (using mobile/web clients and residential gateways) is of the highest importance. To achieve fine-grained security controls and minimize blast radius (total impact of a security event), Xfinity leverages the capabilities provided by the AWS IoT Core MQTT Message Broker and its support for custom authorization using Java Web Token (JWT) and Just in Time Registration (JITR) with x509 Certificates. The authorizers generate AWS IoT Core Policies that allow MQTT connect, publish, subscribe and receive actions at the finest level of granularity - the MQTT topic.

5.1. Mobile Client

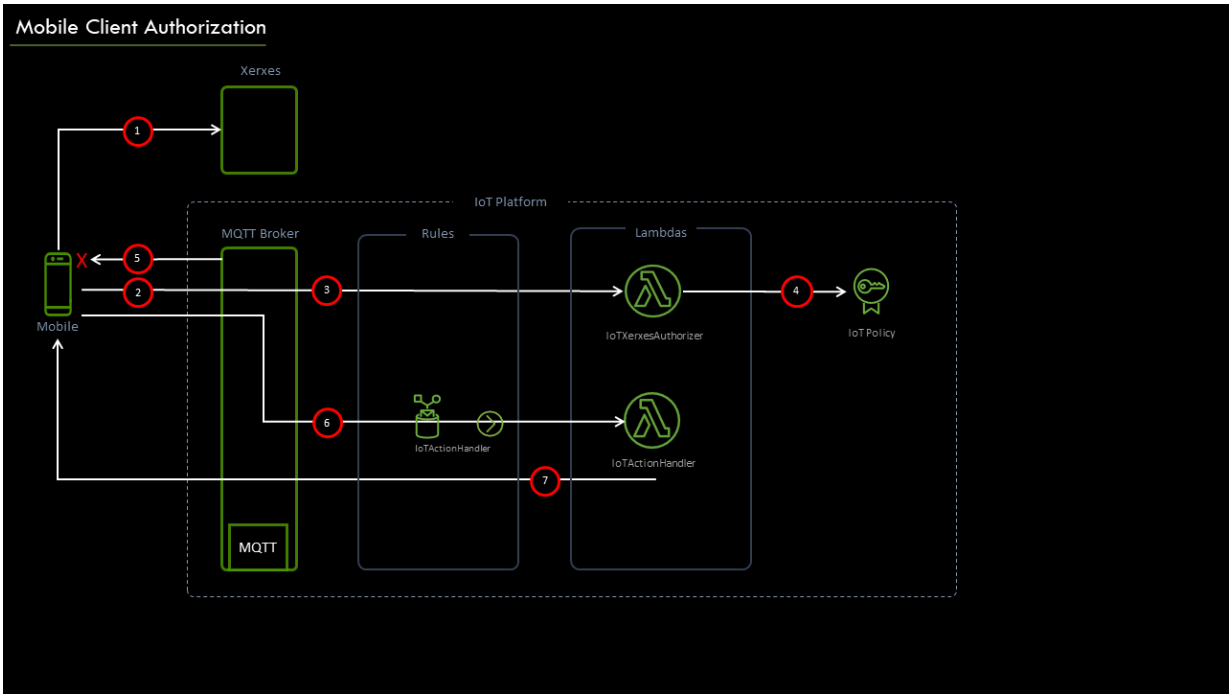

Figure 8 – Mobile Client Authorization Flow Diagram

The Xfinity Application uses an Enterprise Federated Identity Management system, known as Xerxes, to integrate with syndicated partner IdPs (Identity Providers). This system provides authentication and authorization via a Java Web Token (JWT) based Identity Token. Within the JWT there are principal attributes critical in supporting a fine-grained security model. These attributes are:

- Partner Identifier (PID) – Syndicated partner identifier for the Multi System Operator (MSO).
- Account Identifier (AID) – Identifies the subscriber's account id
- JTI – Java Token Identifier (JTI) attribute

The AWS IoT Core MQTT message broker provides a custom authorization capability which is used to verify the JWT and authorize the Mobile Client MQTT connect operation. The JWT is verified via Signature, audience attribute (aud), and issuer attribute (iss) match. Once

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

verified, the JWT principal identity attributes are used to generate an AWS IoT Policy that allows the MQTT Mobile Client to:

- Connect – allow the MQTT connection with a unique MQTT Client ID (CID) generated from the principle information within the JWT. This information includes the PID, AID, and the Java Token Identifier (JTI). An MQTT Client ID must be unique amongst all connected MQTT clients or the prior MQTT Client is disconnected. An example MQTT Mobile Client ID is as follows:

xfi:[PID]_[AID]:clt:[JTI]

  This authorizes only the MQTT client matching the validated JWT to establish a connection to the MQTT Message Broker. An MQTT client with a JWT that is expired or is invalid is denied connection.

- Publish – allows the MQTT client to publish messages to MQTT topics which are name spaced based on the PID, AID and JTI. An example service request message topic is as follows:

c/xfi/[PID]_[Account ID]/[SID]/[CID] /[RID]

  where,

  c is the root of the MQTT topic

  xfi represents the Xfinity Application part of the MQTT topic

  [PID] represents the Syndicated Partner

  [SID] represents a unique Service Identifier as the destination of the request

  [CID] represents the MQTT Client Identifier as the source of the request

  [RID] represent the type of request. This limits the ability of the MQTT Client to publish only to topics containing its principle identity and to services to which it has been granted permission to send action request messages.

- Subscribe/receive – allows the MQTT client to subscribe/receive messages to/from MQTT topic(s) which are name spaced based on the PID, AID and JTI. An example service request response topic is as follows:

c/xfi/[PID]_[AID]/[CID]/[SID]/[RID]/[success|failure]

  where,

  c is the root of the MQTT topic

  xfi represents the Xfinity Application part of the MQTT topic

  [PID] represents the Syndicated Partner

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

[SID] represents a unique Service Identifier as the destination of the request

[CID] represents the MQTT Client Identifier as the source of the request

[RID] represent the Response Identifier or type of response. This limits the ability of the MQTT Client to subscribe and receive action response messages only on topics containing its principle identity and from services to which it has been granted permission to receive action response messages.

5.2. Residential Gateway

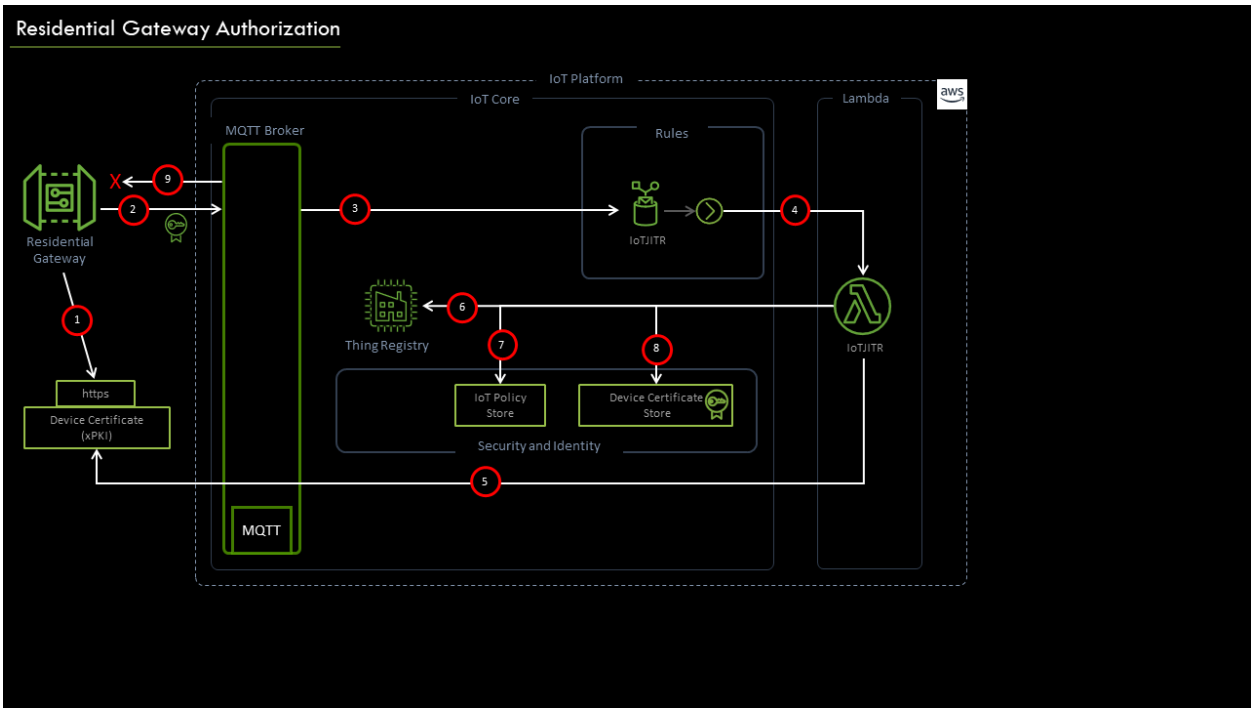

Figure 9 – Residential Gateway Authorization Flow Diagram

The Xfinity Residential Gateway enables advanced IoT Bridge capabilities for Zigbee, Thread, and WiFi-based devices. To provide a secure MQTT connection to the IoT Platform, the Residential Gateway performs a CSR (Certificate Signing Request) and is issued a device certificate by Xfinity Public Key Infrastructure (xPKI). Within the Certificate there are critical principle attributes that provide for a fine-grained security model. These attributes are:

- Partner IDentifier - syndicated partner identifier for the Multi System Operator (MSO). Stored in the SAN (Subject Alternative Name) -> Dir Name (DirName) -> Org Name (O) attribute.
- Account Identifier (AID) - identifies the subscriber's opaque account id; Stored in the SAN (Subject Alternative Name) -> Dir Name (DirName) -> Common Name (CN) attribute.
- Installation Identifier – the device identifier as a UUID (Universal Unique Identifier); Stored in the Subject -> UID attribute as the fourth part of the form [Model]:[Serial Number]:[AID]:[Installation ID]

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

- Mac Address – Device CMAC address. Stored in the SAN -> URI of form *urn:dev:mac:[CMAC]*. CMAC is of valid EUI-64 format.

The MQTT message broker provides certificate-based authorization support. The AWS IoT Core exposes this capability as part of the Just In Time Registration (JITR) feature, which is used to verify a pending device certificate and authorize the MQTT connect operation. The certificate issuer chain is verified and an OCSP check is performed. Once verified, the certificate's principal identity attributes are used to generate an AWS IoT Policy that allows the MQTT Residential Gateway to:

- Connect – allow the MQTT client to connect with a unique MQTT Client ID (CID) generated from the principle information within the Certificate. This information includes the PID, AID, and IID. An MQTT Client ID must be unique amongst all connected MQTT clients. An example MQTT Residential Gateway Client ID is as follows:

iot:[PID]_[AID]:adp:xhf:[IID]

  This authorizes only the Residential Gateway MQTT client matching the validated Certificate to establish a connection to the MQTT Message Broker. An MQTT client with a certificate that has expired or has been revoked is denied the connection.

- Publish– allow the MQTT client to publish messages to MQTT topics which are namespaced based on the PID, AID and IID. An example service request message topic is a follows:

c/iot/[PID]_[Account ID]/[SID]/[CID] /[RID]

  where,

  c is the root of the MQTT topic

  iot represents the IoT Device part of the MQTT topic

  [PID] represents the Syndicated Partner

  [SID] represents a unique Service Identifier as the destination of the request

  [CID] represents the MQTT Client Identifier as the source of the request

  [RID] represents the type of request. This limits the ability of the MQTT Client to publish action request messages only on topics containing its principle identity and to services to which it has been granted permission to send action response messages.

  This limits the ability of the MQTT Client to publish only to topics containing its principle identity and to services to which it has been granted permission to send action request messages.

- Subscribe/receive – allow the MQTT client to subscribe/receive actions to/from MQTT topics which are name spaced based on the PID, AID and CID. An example service request response topic is as follows:

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

c/iot/[PID]_[AID]/[CID]/[SID]/[RID]/[success|failure]

where,

c is the root of the MQTT topic

iot represents the IoT Device part of the MQTT topic

[PID] represents the Syndicated Partner

[SID] represents a unique Service Identifier as the destination of the request

[CID] represents the MQTT Client Identifier as the source of the request

[RID] represents the type of response. This limits the ability of the MQTT Client to subscribe and receive action response messages only on topics conatining its principle identity and from services to which it has been granted permission to receive action response messages.

This limits the ability of the MQTT Client to subscribe and receive action response messages only on topics that contain its principle identity and from services to which it has been granted permission to subscribe/receive action response messages.

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

6. MQTT versus other messaging protocols for client-platform communication

How does MQTT stack up against other messaging protocols? This section seeks to answer that question by first defining the key criterion upon which the messaging protocols can be fairly evaluated. This criteria can then be weighed upon each messaging protocol. These messaging protocols include HTTP, Websocket, AMQP, XMPP, and MQTT. The pros and cons of each messaging protocol is then considered, and finally this evidence can be used to inform a decision.

6.1. Evaluation Criteria

It is important to first define the evaluation criteria used to determine how the messaging protocols fit the needs of an IoT-based messaging system.

Asynchronous bi-directional messaging is necessary to provide support for enhanced Message Exchange Patterns (MEP). This includes asynchronous request/response messages and events using a publish/subscribe messaging pattern. Support for either UDP/IP or TCP/IP bi-directional communication is required.

The ability to support different type of data models for message exchange, whether binary, XML, JSON, etc, provides the flexibility to choose the data type most appropriate to publish a message specification, and enable message validation using client libraries and tooling that is widely available and adopted by the industry. JSON Schema support is required given its wide adoption and support within our organization.

Given the heterogenous nature of program languages and platforms across mobile clients, web clients, devices and services, wide support for client libraries must be available. At a minimum, client libraries must be supported from the likes ofAndroid (Kotlin), IOS (Swift), web clients (Javascript), devices (C/C++), and services (Java/Golang) ..

The security of an IoT system is of upmost importance. It is absolutely necessary to provide a secure communication channel between the mobile/web application clients, devices, and services of the IoT Platform. To this end, TLS 1.2 is required for inflight encryption, support for port 443 to avoid router/firewall traversal issues, and support for token-based auth for mobile/web clients and certificatebased auth for devices.

The performance and efficiency of the messaging protocol and overhead of the client/device libraries must be considered. The solution must offer minimum resource overhead, given factors including the negotiation of the initial connection and connection keep-alive, and the need for efficient messaging (small size on the wire), reduced power consumption for all devices and lightweight implementation (i.e code footprint, memory, etc). The solution must minimize latency related to network protocol negotiation and message broker and router traversal.

The reliability of the communication protocol is important as it offers a way to mitigate errors that may arise from unreliable mobile and/or home networks. The ability for the messaging protocol to deliver in-order messages, to acknowledge receipt of messages, and retransmit messages, if necessary, is required to build a reliable IoT system.

The solution should be cost-sensitive. Some of the key factors to consider that drive cost are the number of mobile clients and devices connected over a period of time (i.e. one day), the size and frequency of the messaging payloads, and the cost of vertical/horizontal scalability of the solution. Note that the extent to which the mobile and web clients are typically connected is a fraction of the time that the gateways are connected, which is to say 24x7.

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

6.2. HTTP (HyperText Transfer Protocol)

HTTP semantics has dealt quite well with brokering synchronous request/response messaging between the client and backend services, via URL paths. But HTTP has fallen short of satisfying the need to support asynchronous notifications across the convergence of application clients, gateways, and backend service(s). The use of multiple HTTPS long or short polling mechanisms, which mimic support for asynchronous responses, requires the mobile and web clients to connect and poll numerous backend services simultaneously. A better mechanism is required, and hence the need to extend the HTTP API semantics to include asynchronous notifications, using a publish/subscribe MEP. HTTP simply does not support publish/subscribe.

Also, it is not always easy to provide a single secure HTTP API connection endpoint for HTTPS client(s). Different services, within a vast organization, typically expose their own HTTPS endpoints. This necessitates a common strategy to expose a single endpoint across an HTTP API Gateway deployment. and in turn. fan-out the requests across the backend services. Horizontal scaling and high availability of this solution is required. Sections 6.2.1 and 6.2.2 detail the pros and cons of HTTP.

6.2.1. Pros

- Supports primary request/response but can lend itself to pub/sub with some work using PUSH_PROMISE vs. Long-polling, but please no short-polling!
- Reasonably small with HPACK for header compression required
- IETF standards are preferred by some vendors
- An HTTP API gateway supports millions of connected clients
- Rsed in mobile applications and most web applications that do not require pub/sub
- Support for many programming languages

6.2.2. Cons

- No guaranteed delivery (retry required
- No “last will & testament” feature for IoT devices that suddenly lose connectivity
- Slightly larger message size due to headers
- Primarily a pull technology:
    - short polling is not real-time (request timer driven) and if the time between requests is short, it can be server resource-intensive; most recommend never to use
    - long polling is immediate but is more complex and server resource-intensive; typically used for responses to requests for information in cases where a server is doing work in the background
- Server push requires work to implement pub/sub and asynchronous responses via PUSH_PROMISE

6.3. Websockets

Websockets provide an extension to application clients and web browser support of HTTPS. Where HTTPS offers only a request/response message pattern, websockets offer a very raw form of bi-

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

directional packet-based communication. It is the lack of built-in MEP and the number of potential connections that is the primary distractor of using websockets directly. Sections 6.3.1 and 6.3.2 detail the pros and cons of websockets.

6.3.1. Pros

- Works over port 443
- Supported by numerous mobile clients and web browsers (implementation on modern browsers is less of a concern)
- Supported by many web servers such as NGINX and Apache

6.3.2. Cons

- Client reconnection implementation on top of websockets is required
- A bit too raw even though it supports two-way asynchronous client/server communication. There is no built-in support for asynchronous request/response and pub/sub message exchange patterns. This would require additional work.
- Using raw websockets would still suffer from having to maintain a connection per client and potentially backend services. This approach would be very resource inefficient from both a client and backend perspective. It would lead to having to scale the web servers to meet the needs of all client/service connection needs, especially when websocket endpoints are not shared across services.

6.4. AMQP (Advanced Message Queueing Protocol)

AMQP is a great protocol for communicating between applications. It meets many of the needs of an asynchronous publish/subscribe and guaranteed message delivery system which supports numerous programming languages. But, as a message-oriented, middleware-based solution, it has not really found its home in the IoT space. Sections 6.4.1 and 6.4.2 detail the pros and cons of AMQP.

6.4.1. Pros

- Richest set of message scenarios (patterns)
- Asynchronous
- Supports queuing
- Mobile client support; web browser support via web sockets

6.4.2. Cons

- Uses port 5672 (not 443), which would lead to potential router and firewall issues
- Requires RabbitMQ on AWS EC2 instances; not a managed AWS service
- Larger protocol
- Used in IoT space by very few vendors

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

6.5. XMPP (eXtensible Messaging and Presence Protocol)

Extensible Messaging and Presence Protocol is an open communication protocol designed for instant messaging, presence information, and contact list maintenance. Based on XML, it enables the near-real-time exchange of structured data between two or more network entities.

6.5.1. Pros

- Support for message read, write, and consume
- Extended messaging and presence protocols
- Used for two-way chat and push notifications

6.5.2. Cons

- Uses port 5222 (not 443) which would lead to potential router and firewall issues
- Requires an XMPP/Jabber server on EC2 instances; not a managed AWS service
- It's XML-based, not JSON

6.6. MQTT

MQTT, as described in this paper, is a mature, stable, and feature rich messaging transport system. It delivers on all the key evaluation criteria. Sections 6.6.1 and 6.6.2 detail the pros and cons of MQTT.

6.6.1. Pros

- Provides a backend service the ability to publish notifications without the added complexity of implementing a long polling or websocket mechanism, as well as message routing within the context of each individual service. Aggregating and pushing the responsibility to the MQTT Message Broker decouples this problem from the client and underlying services. Note: MQTT also handles session persistence by delivering missed messages to the client.
- Provides a means to multiplex notifications across 1 or more topics grouped by backend service responsibility; the MQTT Message broker inherently supports message routing via topics.
- Only one secure client connection to the MQTT Message broker is required, allowing for simpler configurations for clients, load balancers, security policies, etc
- JWT and X.509 certificate-based authorization support
- Support for port 443
- Supports asynchronous message patterns (i.e. pub/sub)
- Assured delivery (3 QoS levels) and retained messages which provide flexible options for Client/Server.
- Supports the “last will & testament” feature that notifies other clients of an ungraceful disconnected of gateway-based devices
- Multiple subscriptions are ‘multiplexed’ over one connection
- Smaller size on the wire - minimum compressed header
- Great for a resource-constrained device, low bandwidth conditions, and high latency networks
- Brokers support millions of connected devices
- Easy to route messages to topic subscriber(s)
- Used in mobile applications, web clients, and devices for numerous IoT-based applications

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

- Lightweight implementation for embedded clients
- Power-efficient due to no-polling, shorter messages
- Less resources consumed compared to long polls on server.
- Support for many programming languages

6.6.2. Cons

- MQTT is based on the OASIS standard (OASIS-open.org), which is not preferred by some vendors
- Fixed headers and options apply if extensibility is required

6.7. Decision

In the end, after numerous discussions, careful consideration, and weighing the options based on the selection criteria, MQTT was the clear choice. Our Mobile/Web Client and Gateway teams decided that MQTT was the preferred communication protocol for messaging to/from the IoT Platform.

7. Conclusion

The Xfinity Mobile and Platform teams worked together to implement data models and topic structures for MQTT communications that are standardized and extensible across any device type and payload size. Its robust security, enabled by custom authorizers, ensures that mobile/web clients and devices can only publish and subscribe to topics relevant for the operator and subscriber's account.

By contrast, API semantics based on HTTP and other communication protocols have fallen short when it comes to satisfying the need to support secure, asynchronous notifications across the convergence of application clients, gateways, and backend service(s). The use of MQTT, in addition to the use of HTTP API semantics, fulfills the need to extend the API semantics to include asynchronous notifications using a Publish/Subscribe MEP.

We chose MQTT as a protocol for mobile-to-platform communications as well as Residential Gateway-to-platform communications. Its maturity as a lightweight, scalable, and robust protocol has proven itself in our production deployment of millions of clients.

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.

8. Abbreviations

AID	Account IDentifier
AMQP	Advance Message Queuing Protocol
AWS	Amazon web services
CSR	Certificate Signing Request
HTTP	HyperText Transfer Protocol
IdP	Identity Provider
JTI	Java Web Token Identifier
JWT	Java Web Token
KB	Kilo Byte
MSO	Multiple System Operator
MQTT	Message Queuing Telemetry Transport
PID	Partner IDentifier
PUBACK	Publish acknowledgement
PUBREC	Publish received
PUBREL	Publish release
PUBCOMP	Publish complete
QoS	Quality of Service
RID	Request/Response IDentifier
SCTE	Society of Cable Telecommunications Engineers
SID	Service IDentifier
XMPP	eXtensible Messaging and Presence Protocol

9. References

MQTT client programming concepts

https://www.ibm.com/docs/en/ibm-mq/9.0?topic=telemetry-mqtt-client-programming-concepts

IoT Core Developer's Guide: MQTT

https://docs.aws.amazon.com/iot/latest/developerguide/mqtt.html

© 2021, SCTE® CableLabs® and NCTA. All rights reserved.