Features of the MQTT protocol

1. Connection-oriented

   A client requesting a connection with an MQTT server is designed to establish a TCP/IP socket and then stay connected and send messages until it explicitly disconnects or is disconnected due to network issues.

2. Publish-Subscribe

   In the publish-subscribe model of topic-based messages, there is no direct transmission or reception of messages between clients, and all messages can only be linked through a server (or broker).

3. Quality of Service (QoS)

   When sending a message to a target topic, you can set the QoS to 0, 1, or 2.

   - 0 : Maximum one transmission, no guarantee of message delivery.

   - 1 : Guarantees at least one delivery, but the same message may be sent multiple times.

   - 2 : Guarantees that the message is sent exactly once with no duplicates.

4. Types of messages

   The following messages are defined in the MQTT specification.

   - CONNECT, CONNACK

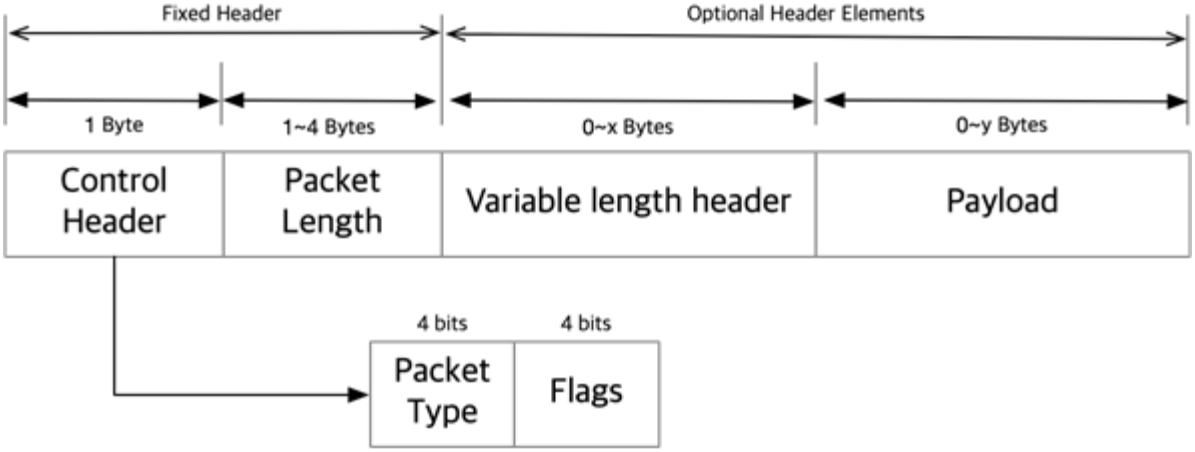
   - DISCONNECT

- PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP
- SUBSCRIBE, SUBACK
- UNSUBSCRIBE, UNSUBACK
- PINGREQ, PINGRESP

MQTT messages are used in a wide variety of applications, from enterprise applications to gaming and entertainment, because they have little overhead (two-byte fixed headers and variable-length headers, except for the topic path), support QoS, and provide flexibility for a wide range of applications. In particular, it is becoming the standard for telemetry transmission in the IoT.

Structure of MQTT packets

A packet consists of a fixed 2-byte header that must always exist, a header that can vary in size, followed by the payload as shown below.



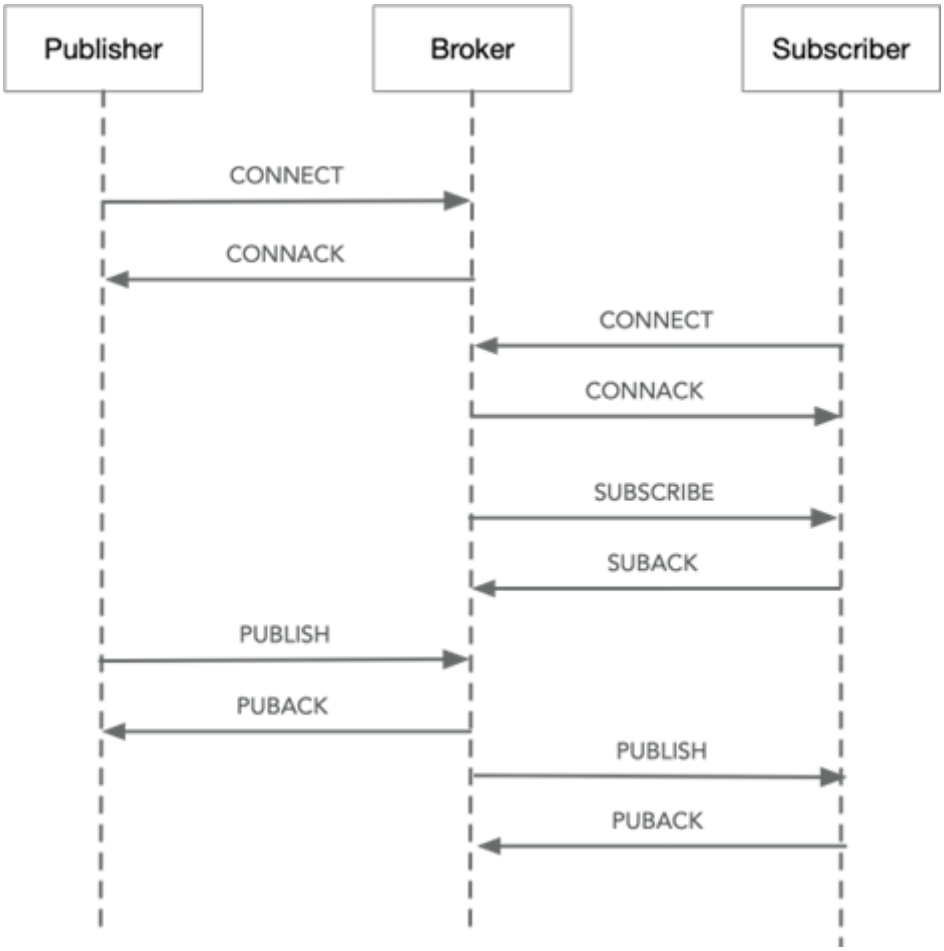
The first four bits indicate the type of packet. They are defined as follows.

Name	Value	Direction	Description
Reserved	0	Disable	-
CONNECT	1	Client → Server	Connection request
CONNACK	2	Server → Client	Connect ACK
PUBLISH	3	Client ←→ Server	Publishing messages
PUBACK	4	Client ←→ Server	Issuing an ACK (QoS 1)
PUBREC	5	Client ←→ Server	Receive publications (QoS 2)
PUBREL	6	Client ←→ Server	Receive publications (QoS 2)
PUBCOMP	7	Client ←→ Server	Receive publications (QoS 2)
SUBSCRIBE	8	Client → Server	Subscription requests
SUBACK	9	Server → Client	Subscription requests ACK
UNSUBSCRIBE	10	Client → Server	Unsubscribe
UNSUBACK	11	Server → Client	Unframing ACK
PINGREQ	12	Client → Server	PING request
PINGRESP	13	Server → Client	PING response
DISCONNECT	14	Client → Server	Disconnect
Reserved	15	Disable	-

The lower four bits then display additional flags in the control command. Only the defined values should be used in messages other than the PUBLISH message.

Control Packets	Flag	Bit 3	Bit 2	Bit 1	Bit 0
CONNECT	Reserved	0	0	0	0
CONNACK	Reserved	0	0	0	0
PUBLISH	MQTT 3.1.1	DUP	QoS	QoS	RETAIN
PUBACK	Reserved	0	0	0	0
PUBREC	Reserved	0	0	0	0
PUBREL	Reserved	0	0	1	0
PUBCOMP	Reserved	0	0	0	0
SUBSCRIBE	Reserved	0	0	1	0
SUBACK	Reserved	0	0	0	0
UNSUBSCRIBE	Reserved	0	0	1	0
UNSUBACK	Reserved	0	0	0	0
PINGREQ	Reserved	0	0	0	0
PINGRESP	Reserved	0	0	0	0
DISCONNECT	Reserved	0	0	0	0

The basic message sequence for an application using MQTT is shown in the figure below. The Clients, Publisher, and Subscriber, send a CONNECT message to the server, Broker, and Broker returns CONNECT in response. The Subscriber makes a SUBSCRIBE request to the topic of interest and the response is a SUBACK. PUBLISH is a message sent by the publisher to the target topic or by the broker to the subscriber, and if QoS 1, the receiver acknowledges receipt with PUBACK.



The following fields may be included in a CONNECT message.

Field	Required	Description
clientID	Required	Customer identifier, length 1-23
cleanSesison	Select	0: The server resumes the client's session. The server must save the session state after the client disconnects. 1: The client and server should discard the old session and start a new one.
username	Select	Name for authentication
password	Select	Binary password
will	Select	lastWillTopic, lastWillQoS, lastWillMessage, lastWillRetain
keepAlvie	Select	Specifies a time interval in seconds. The client must send a mes sage or PINGREQ before the keepalive timer expires. The server is forced to terminate the connection with the client after 1.5 ti mes the keepalive time. Specifying 0 disables the keepalive mec hanism.

When the server responds to a client's CONNECT request with CONNACK, the response code has the following meaning

Response code	Description
0	Connection success
1	Reject - Unacceptable MQTT protocol version
2	Reject - Client ID is valid, but server does not allow connection
3	Reject - The server is unavailable
4	Reject - Invalid username or password
5	Reject - the client connection is denied

Sending and receiving server-to-client messages is done via PUBLISH messages. The following fields can be set.

Field	Required	Description
packetID	Required	A unique identifier for the packet. The client library is responsible for this.
topicName	Required	Topic path for the message.
qos	Required	Specifies the QoS level of the message. (0, 1, 2)
retainFlag	Required	Retain message flag
dupFlag	Required	Displays a retransmission message.
payload	Select	A byte array for the message body.

MQTT and IoT applications

At a simplified level, most applications of MQTT in the IoT space look like the image below.

Telemetry data collected from sensors is sent to a pre-arranged MQTT broker topic in a format such as JSON or CSV, and the server-side application subscribing to the topic receives the message from the broker and stores it in a database, and the application accesses the database to retrieve or process the data as needed.

This is probably the most common and familiar architecture and may seem obvious. However, over the course of many IoT projects, both large and small, we've seen the problems that this structure brings, and they usually boil down to the following.

Issue 1) Availability of MQTT Broker

There are several quality MQTT brokers out there, including open source. These brokers are mature and work well on their own. Having implemented these brokers in various types of IoT projects, I have found that the availability of the MQTT broker is directly dependent on the availability of the entire IoT service. Since all data is communicated through the broker, the availability of the overall service can never exceed the availability of the MQTT broker used, which means that the development and operational know-how of the chosen broker is the most important factor in determining the quality of the overall service. Accumulating sufficient technical skills on an easily accessible open-source broker has become as much or more of an effort than making

the application you're trying to develop stable, and what was once a secondary component has become a key one.

Issue 2) Managing "redundant" storage

The purpose of an MQTT Broker is to ensure that messages sent by sensors (Publishers) are delivered correctly and without loss to ingesting applications that are subscribing to the topic. In this context, "lossless delivery" means that the broker itself must first store the messages it receives in some form of memory. The store and forward strategy is a common design strategy for all message brokers, not just MQTT. The ingestion application, which receives the messages forwarded by the broker, transforms them into a serviceable structure and stores them (in a database such as an RDBMS or NoSQL).
Once the system is deployed, two types of storage (MQTT's storage and the application's storage) must be managed to keep the service running reliably and to cope with failures.
And when new types of sensors are added or data types are added, the collection application must be further developed or modified.

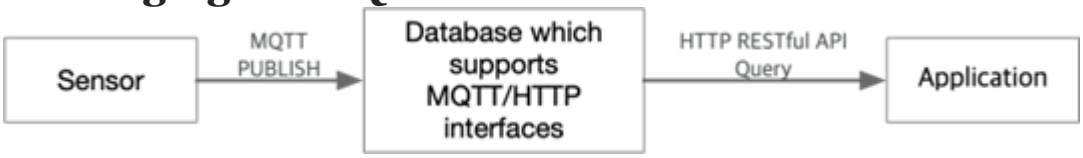
As a result, every IoT service developer or system architect starts with the question: Do I always have to bear the burden of managing "redundant" storage with the operation of an MQTT broker? and ends up with the question: How can I efficiently process large amounts of IoT data? to the question of how to efficiently process large amounts of IoT data.

Machbase and Machbase Neo

What if the sensor sends telemetry data directly to the database via MQTT? It would look something like this.

Traditional architecture:

Leveraging the MQTT interface as Machbase sees it:



The existence of this type of database solves the problem of managing MQTT broker operations and managing redundant storage operations to prevent message loss. Additionally, the need for a collection application could be eliminated. Sensors would continue to send telemetry data via MQTT and applications could retrieve the required data via the HTTP Restful API provided by the database or via the standard database interface of the programming language (JDBC, ODBC, Go Driver...).

Machbase Neo is the result of this approach.(https://machbase.com/neo)

Machbase is the world's №1 performance database, certified by tpc.org (see TPCx-IoT V2, https://www.tpc.org/tpcx-

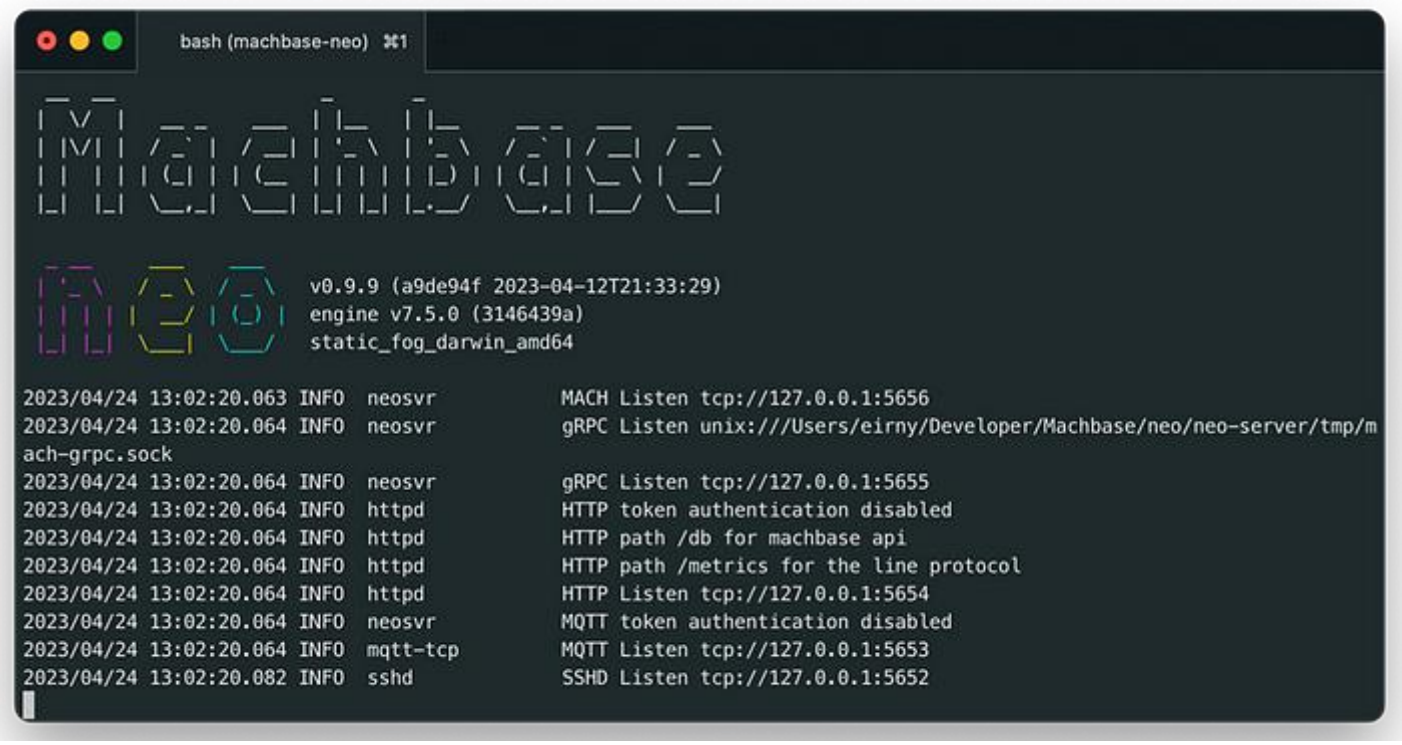
iot/results/tpcxiot_perf_results5.asp?version=2), and machbase-neo combines Machbase with IoT-oriented convenience features such as an MQTT server. All features are distributed in a single executable file, so it can be installed by simply copying the file, and the UX is implemented to suit the development workflow of IoT developers.

For the sake of brevity, we'll take a quick look at Machbase-neo's MQTT interface for storing data from sensors. For more details, you can refer to the documentation and tutorials on the official site(https://machbase.com/neo)
For the sake of demonstration, we'll be testing MQTT with the mosquito_pub command, so you'll need to have the mosquito-client tool installed first. And machbase-neo can be downloaded from the site or by running the command below. (Note: Windows users should download it directly from the release page on the site).

```
$ sh -c "$(curl -fsSL https://neo.machbase.com/install.sh)"
```

Unzip the downloaded archive and copy the "machbase-neo" executable to your desired location. Complete the installation.

When you run machbase-neo serve, it creates a database and runs as shown below. As shown on the screen, the MQTT server is opened on port 5653 and the HTTP server on port 5654.



Then use the command below to create the tables we need for the demo.

```
$ curl -o - http://127.0.0.1:5654/db/query \
    --data-urlencode \
    "q=create tag table EXAMPLE (name varchar(40) primary key, time datetime basetime, value double)"
```

Now we can mimic the sensor sending data to MQTT with mosquito_pub as shown below. If you look at the topic path, you can see that the table name EXAMPLE is used.

```
$ mosquitto_pub -h 127.0.0.1 -p 5653 \
    -t db/append/EXAMPLE                    \
    -m '[ "my-car", 1670380342000000000, 32.1 ]'
```

Applications can retrieve data entered via HTTP.

```
$ curl -o - http://127.0.0.1:5654/db/query \
    --data-urlencode "q=select * from EXAMPLE"
```

Conclusion

I believe Machbase Neo(https://machbase.com/neo) is a good example of how the technology accumulated in the development of time series databases can help solve the problems of developers in the IoT space. I hope this article will be useful for many developers and I invite you to visit our GitHub(https://github.com/machbase/neo-server) for more information and to help and contribute to the community.

by Eirny Kwon