WEBINAR

Building Scalable & Reliable MQTT Clients for Enterprise Computing

HIVEMQ

WELCOME

Silvio Giebl

- Software Developer @HiveMQ
- Developer and Maintainer of the HiveMQ MQTT Client
- Distributed & scalable systems
- High performance and reactive applications

Clive Jevons

- Independent Consultant @Jevons IT
- Used the HiveMQ MQTT Client to integrate it in a connected car platform
- Was involved in the development of the HiveMQ MQTT Client

What we will talk about ...

- What is an MQTT client?
  → You will learn how flexible MQTT can be used for a variety of use cases
- Why HiveMQ MQTT Client?
  → You will learn the features and the advantages
- Real world Enterprise use case of the HiveMQ MQTT Client in a connected car platform
  → You will learn from the experience of an actual user

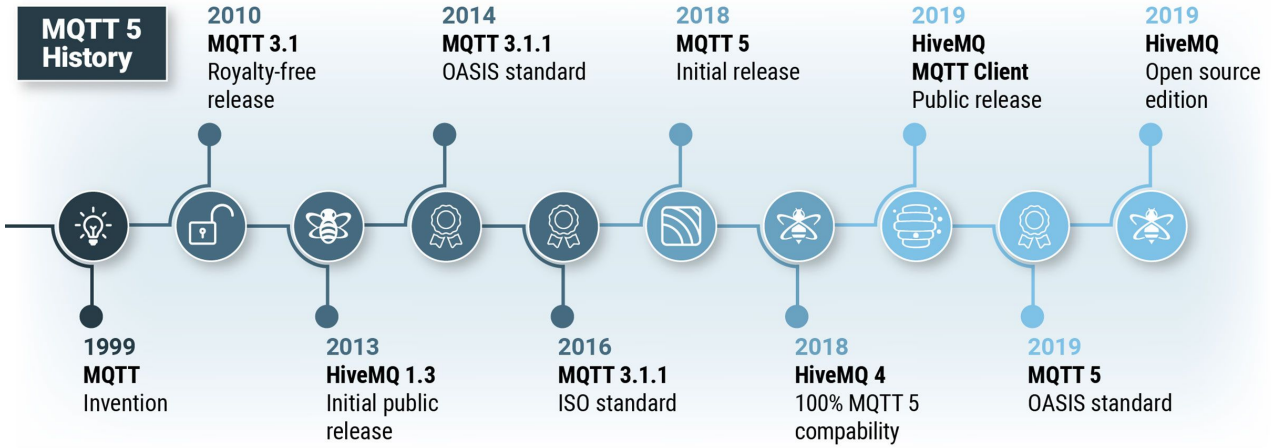
What is MQTT?

What is MQTT?

- Communication protocol
- Publish/Subscribe pattern
- OASIS and ISO Standard (ISO/IEC PRF 20922)
  → interoperability
- Decoupling of sender and receiver in space, in time
  → more robust and scalable
- QoS levels
  → reliable communication over unreliable networks
- Flexible, lightweight, dynamic topics, data agnostic
- Use cases: IoT, IIoT, Industry 4.0, Logistics, Connected Cars, ...
  → everything that links a lot of devices

What is MQTT 5?

- A lot of additional features while keeping MQTT lightweight and flexible
- Many improvements making MQTT an even more versatile protocol



HIVEMQ

What is an MQTT Client?

What is an MQTT Client?

- MQTT is known to be used for small devices
- Actually it is used for a variety of systems
- MQTT is lightweight and does not put restrictions on applications

→ Almost everything can be an MQTT client

- **Embedded**: sensors, control units, ...
- **Mobile/desktop**: apps, browser applications, ...
- **Backend**: integration with other systems, databases, microservices, ...
- Different use cases, different requirements
- But all have in common that they need to communicate in a reliable way

MQTT Clients for Embedded

Requirements

- Low computing power
- Low bandwidth
- High latency
- Unstable network
- Huge amount of devices, little data per device

→ All covered by MQTT

- MQTT clients are lightweight
- Minimal network overhead
- QoS guarantees
- MQTT Broker removes complexity from clients, ensures scalability

MQTT Clients for Mobile

Requirements

- Platform independent
- Responsiveness, reactiveness
- Unstable network

→ All covered by MQTT

- MQTT is a wire protocol standard, so interoperable
- MQTT is push based
- QoS guarantees

MQTT Clients for Backends

- Usually MQTT is used for a huge amount of clients, each handling a small portion of the data
- Backend systems are often used for ingestion of all data for monitoring, analytics and control

Requirements

- Scalability
- High throughput per service
- Reliability, no message loss, no overload, backpressure

→ All covered by MQTT

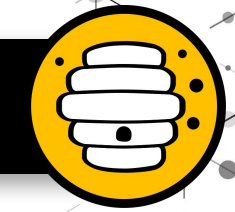
- Scalability is ensured by the broker
- Shared subscriptions for scaling out/load balancing MQTT clients

MQTT Clients for Backends

- Shared subscriptions are especially useful for microservice like systems
- Subscribers can join/leave the shared subscription group dynamically
- Shared subscriptions are standardized with MQTT 5
  (they can be supported for MQTT 3 as well)

HiveMQ MQTT Client

HiveMQ MQTT Client

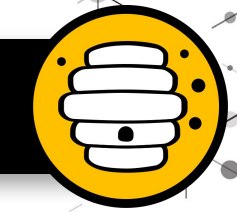


- MQTT Client Java Library
- All MQTT 3.1.1 and MQTT 5 features (including all optional features)
- Open Source
- Different API flavors: Reactive, Asynchronous, Blocking

Key Benefits

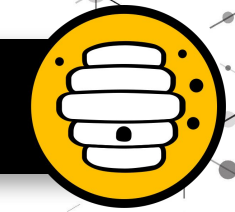
- Reactive
- Backpressure, Stability, Reliability
- Resource efficiency, low overhead, high throughput

MQTT Features



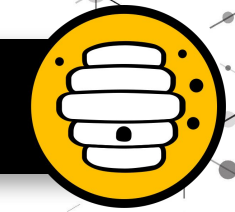
- MQTT 3 and 5: all QoS levels, retained messages, Will/LWT, ...
- MQTT 5
  - Session expiry
  - Message expiry
  - Flow Control → better backpressure handling
  - Shared subscriptions (also supported for MQTT 3)
  - Payload Format Indicator and Content Type
  - User properties
  - Negative acknowledgements and reason strings
  - Request/Response
  - Topic Aliases (automatically)
  - Subscription Identifiers (automatically)
  - Enhanced Auth

Features on top of MQTT



- TLS/SSL
- Websocket, Secure Websocket
- Automatic reconnect (automatic & configurable)
- Offline message buffering
- Thread management (automatic & configurable)
- Thread safety
- Pluggable Enhanced Auth support
- Automatic topic alias tracing and mapping
- Backpressure handling (deep integration with the reactive API)

Open Source

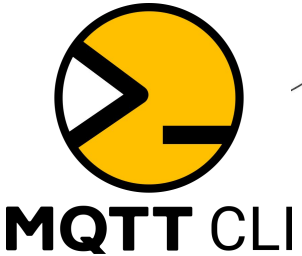


- Source code on GitHub: https://github.com/hivemq/hivemq-mqtt-client
- Apache 2 license
- Free to use
- Actively maintained by HiveMQ
- Transparent development
  - Issues and PRs on GitHub
  - Feedback and contributions are welcome
- Why Open Source?
  - MQTT is the standard IoT protocol
  - Everybody should be able to use MQTT

Example Uses

MQTT CLI

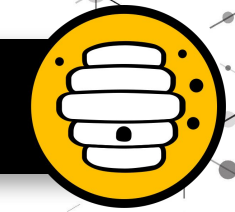
- Command Line Tool
- Debugging
- Simulating MQTT Clients



Internal: HiveMQ Device Simulator

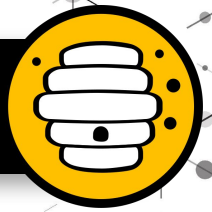
- Simulating millions of MQTT clients with few machines
- Used to reproduce customer scenarios
- Used as a benchmark tool

Why Different API Flavors?



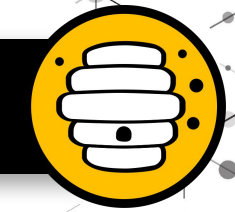
- Enables fast prototyping
  - All APIs are as simple as possible
  - But starting with the blocking API is often simpler
  - Only a few lines of code for MQTT communication
- Allows evolution of applications
  - Asynchronous API is often enough
  - Parts can be switched to reactive when scaling and more precise backpressure control is needed
  - Different API styles can be used simultaneously
- Fluent Builders also help
  - Only use the features you need
  - When you need more features, no need to rewrite your whole code

What Does Lightweight Mean?

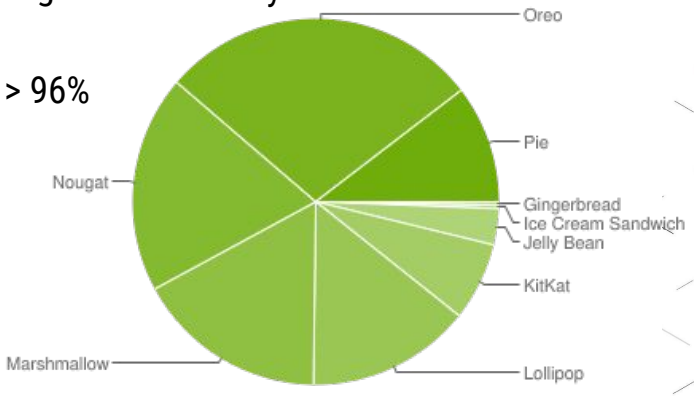


- Communication should not use a major part of the computing time
- Low memory usage → around 5KB per client instance
- Many clients possible
  - Intelligent thread pooling
  - Overhead per client is minimal
- Also possible to use 1 client by many threads → flexibility
  - Recommendation: if different parts of a service are independent, use more clients instead of sharing 1 client to avoid unnecessary coupling
- Application messages and computations are important

Embedded, Mobile, Backend

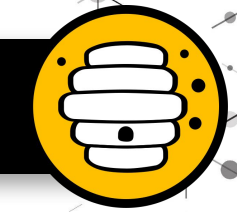


- Resource efficiency helps all use cases
  - Embedded/mobile → hardware/battery restrictions
  - Backend → enables higher throughput for actual processing
- Backend:
  - Scaling with shared subscriptions
  - Backpressure helps building more robust systems
- Mobile: Support for Android
  - API 19/KitKat and up → > 96%
- Reactive API
  - Mobile: responsiveness, often used on Android
  - Backend: resilience

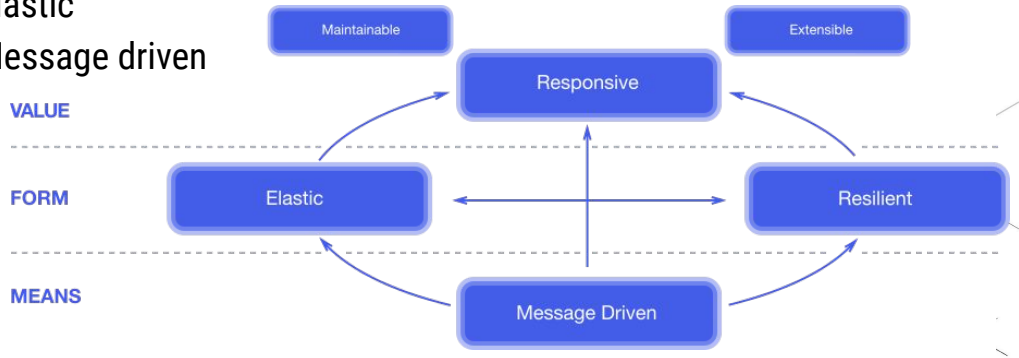


https://developer.android.com/about/dashboards

Why Reactive?

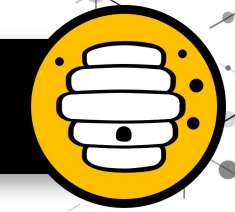


- Reactive Manifesto (https://www.reactivemanifesto.org/):
  - Responsive
  - Resilient
  - Elastic
  - Message driven



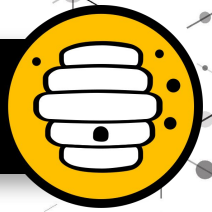
- Reactive is the solution for high scale applications
- Perfect fit for MQTT

Why Reactive?



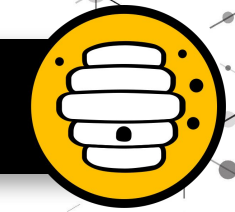
- HiveMQ MQTT Client is reactive
  - In its core
  - Has a reactive API using RxJava which follows the reactive streams specification
  - Interoperable with other reactive libraries (interoperability is not only important for the MQTT protocol, but also the libraries)
- Barrier of entry:
  - You have to learn new concepts, think differently
  - Good news: you can start with the asynchronous API and move to reactive later

What is Backpressure?



- Mechanism to adapt message rates in an asynchronous system
- If an application is overwhelmed by too many messages
  - It might crash
  - It might drop important messages (without other applications/the broker even knowing)
  - A lot of unnecessary work is done when dropping messages
- Backpressure lets the application that produces too many messages know, that they can not be handled → appropriate and early actions can be taken
- When using shared subscriptions the load can be better balanced between all clients in the group
- → Backpressure improves resilience and robustness
- MQTT 5 Flow Control limits concurrent unacknowledged messages

API Design

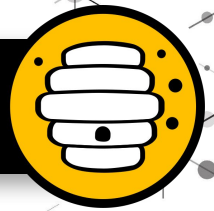


"APIs should be easy to use and hard to misuse. It should be easy to do simple things; possible to do complex things; and impossible, or at least difficult, to do wrong things."
(Joshua Bloch)

- The HiveMQ MQTT Client gives you full control over all MQTT features
- It is not a restrictive framework
- But using sensible defaults, you do not have to configure everything
- The context sensitive fluent builder pattern used throughout the library enables short concise code but highly customizable

Code examples

Setup

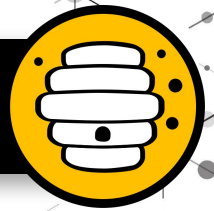


```
dependencies {
    implementation group: 'com.hivemq', name: 'hivemq-mqtt-client', version: '1.1.3'
}
```

```
<dependencies>
    <dependency>
        <groupId>com.hivemq</groupId>
        <artifactId>hivemq-mqtt-client</artifactId>
        <version>1.1.3</version>
    </dependency>
</dependencies>
```

Maven Central, JCenter, JitPack

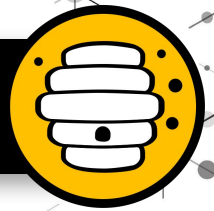
Client configuration



```
Mqtt5Client client1 = Mqtt5Client.builder().build();
```

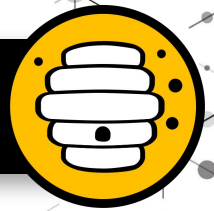
```
Mqtt5Client client2 = Mqtt5Client.builder()
        .identifier("client2")
        .serverHost("broker.hivemq.com")
        .serverPort(1234)
        .sslWithDefaultConfig()
        .webSocketWithDefaultConfig()
        .automaticReconnectWithDefaultConfig()
        .addConnectedListener(context -> System.out.println("connected"))
        .addDisconnectedListener(context -> System.out.println("disconnected"))
        .build();
```

Client configuration



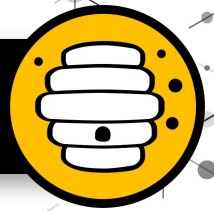
```
Mqtt5Client client3 = Mqtt5Client.builder()
        .identifier("client3")
        .transportConfig()
            .serverHost("broker.hivemq.com")
            .serverPort(1234)
            .sslConfig()
                .protocols(Arrays.asList("TLSv1.3"))
                .cipherSuites(Arrays.asList("TLS_AES_128_GCM_SHA256"))
                .trustManagerFactory(myTrustManager)
                .keyManagerFactory(myKeyManager)
                .applySslConfig()
            .webSocketConfig()
                .serverPath("mqtt")
                .subprotocol("mqtt")
                .applyWebSocketConfig()
            .applyTransportConfig()
        ...
```

Client configuration



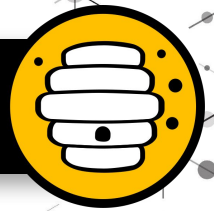
```
...
.automaticReconnect()
    .initialDelay(100, TimeUnit.MILLISECONDS)
    .maxDelay(10, TimeUnit.SECONDS)
    .applyAutomaticReconnect()
.addDisconnectedListener(context -> {
    context.getReconnector().reconnectWhen(
            getOAuthToken(),
            (token, throwable) -> {
                ((Mqtt5ClientDisconnectedContext) context).getReconnector()
                        .connectWith()
                        .simpleAuth().password(token).applySimpleAuth()
                        .applyConnect();
            });
})
...
```

Client configuration



```
...
.simpleAuth()
    .username("username")
    .password("password".getBytes())
    .applySimpleAuth()
.willPublish()
    .topic("will")
    .qos(MqttQos.AT_LEAST_ONCE)
    .payload("hello world".getBytes())
    .messageExpiryInterval(10)
    .payloadFormatIndicator(Mqtt5PayloadFormatIndicator.UTF_8)
    .contentType("text/plain")
    .userProperties()
        .add("time", System.currentTimeMillis() + "ms")
        .add("sender", "client3")
        .applyUserProperties()
    .applyWillPublish()
.build();
```

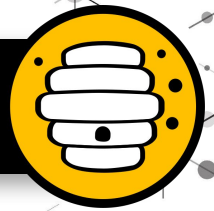
Simple Publish & Subscribe



```
client.connect();
client.publishWith()
        .topic("demo/topic")
        .qos(MqttQos.EXACTLY_ONCE)
        .payload("hello world".getBytes())
        .send();
client.disconnect();
```

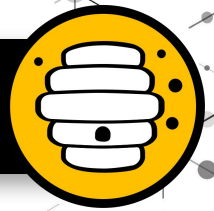
```
client.connect();
client.toAsync().subscribeWith()
        .topicFilter("demo/#")
        .callback(System.out::println)
        .send();
client.disconnect();
```

Async Publish



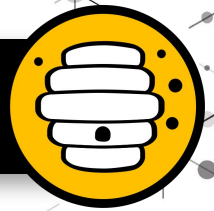
```
Mqtt5AsyncClient async = client.toAsync();

async.connect()
        .thenCompose(connAck -> async.publishWith()
                .topic("demo/topic")
                .qos(MqttQos.EXACTLY_ONCE)
                .send())
        .thenCompose(publishResult -> async.disconnect());
```

MQTT 5 Features



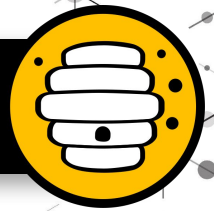
```
client.connectWith()
        .cleanStart(false)          // resume a previous session
        .sessionExpiryInterval(30)  // keep session state for 30s
        .restrictions()
            .receiveMaximum(10)             // receive max. 10 concurrent messages
            .sendMaximum(10)                // send max. 10 concurrent messages
            .maximumPacketSize(10_240)      // receive messages with max size of 10KB
            .sendMaximumPacketSize(10_240)  // send messages with max size of 10KB
            .topicAliasMaximum(0)           // the server should not use topic aliases
            .sendTopicAliasMaximum(8)       // use up to 8 aliases for the most used topics
            .applyRestrictions()
        .send();
```

MQTT 5 Features



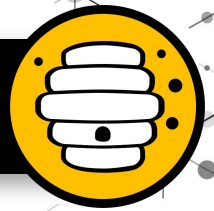
```
client.publishWith()
        .topic("demo/topic")
        .qos(MqttQos.EXACTLY_ONCE)
        .payload("hello world".getBytes())
        .retain(true)
        .payloadFormatIndicator(Mqtt5PayloadFormatIndicator.UTF_8)
        .contentType("text/plain")  // our payload is text
        .messageExpiryInterval(120) // not so important, expire after 2min if can not be delivered
        .responseTopic("demo/response")
        .correlationData("1234".getBytes())
        .userProperties()           // add some user properties to the message
            .add("sender", "client1")
            .add("receiver", "you")
            .applyUserProperties()
        .send();
```

Reactive Request/Response



```
Flowable<Mqtt5Publish> requestStream = client.toRx()
        .subscribeStreamWith()
        .topicFilter("request/topic")
        .applySubscribe();

Flowable<Mqtt5PublishResult> responseStream = client.toRx()
        .publish(requestStream
                .filter(requestPublish -> checkIfResponsible(requestPublish))
                .observeOn(Schedulers.computation())
                .map(requestPublish -> Mqtt5Publish.builder()
                        .topic(requestPublish.getResponseTopic().get())
                        .qos(requestPublish.getQos())
                        .payload(performComputation(requestPublish.getPayload()))
                        .correlationData(requestPublish.getCorrelationData().orElse(null))
                        .build()));

responseStream.subscribe();
```

Reactive Android Example



```
client1.toRx()
       .subscribeStreamWith()
       .topicFilter("chat1/messages/#")
       .applySubscribe()
       .observeOn(AndroidSchedulers.mainThread())
       .doOnNext(message -> addMessageToUi(message))
       .observeOn(AndroidSchedulers.from(backgroundLooper))
       .filter(message -> isImportant(message))
       .doOnNext(message -> createNotification(message))
       .subscribe();
```

Use of MQTT Clients in a Connected Car Platform

Use Cases

1) Mirroring fleet data between clusters

2) Integrating HiveMQ Client into communication middleware joynr

3) Using HiveMQ Client for processing data from a production plant

Resources

[Get Started with MQTT]

[MQTT Essentials Series]

MQTT [MQTT at OASIS]

HIVEMQ [Evaluate HiveMQ Broker]

HIVEMQ CLOUD [Try HiveMQ Cloud for Free]

ANY QUESTIONS?

Reach out to community.hivemq.com

THANK YOU

Stay updated on upcoming webinars

Subscribe to our Newsletter!