HIVEMQ

Architecture proposal for the VDA 5050

The communication of order and status information between driverless transport vehicles and control system

Technical White Paper

©2020

TABLE OF CONTENTS

Goal of this document **3**
Executive summary **3**
Short Introduction of the VDA 5050 Specification **4**
What are the Challenges and Requirements **4**
Components of the Architecture Proposal Using HiveMQ **4**
Message Broker - HiveMQ 4
HiveMQ Extension System 5
Actors 5
Related Components 5
Message Broker Features 6
Key Features Reliability, Scalability, Performance 6
Monitoring, Logging, MQTT Tracing 7
Security Expectations 9
Security at the transport level 9
Security at application level 9
Enterprise Security Extension 10
Security by Data Integrity 11
Security at Infrastructure Level 12
Order-State Use case 12
Order - State MQTT Message Flow 12
Topic Structure 12
MQTT Client characteristics 13
MQTT Publish - Orders and State 13
MQTT Will, Client offline and online event detection 14
Conclusion **15**
Appendix **16**
Client Implementation with HiveMQ MQTT library 16
AGV Client 16
CS Client 18
HiveMQ Configuration recommendations 19
Glossary **20**

Goal of this document

This document describes a reference architecture for the communication between driverless transport vehicles and a control system. The architecture is based on the interface specification VDA 5050, MQTT as the communication protocol and the HiveMQ Broker. The document focuses on the main use case described in the VDA 5050: The communication of order and status information between AGV and Control System.

Executive summary

This document describes a reference architecture for communication between autonomous transport vehicles (AGVs) and a control system for intralogistics processes. The description of the architecture is based on the specification VDA 5050. According to VDA 5050, the MQTT protocol will be used as the standard communication protocol between the actors.

In addition to the functional requirements, what are the challenges for a production ready deployment? What architecture components are necessary in detail?

1. A key component of the architecture is the message broker. It has to be fail-safe, robust and scalable, as well as providing the necessary performance guarantees. HiveMQ is a powerful, cluster-capable, fail-safe MQTT broker that supports 100% of all functions of the MQTT protocol (all versions MQTT 3 and MQTT 5).

2. The overall correct security implementation for the architecture is critical. Which security mechanisms can be used, are there standard solutions that fit most scenarios, can different mechanisms be used for different clients? HiveMQ provides a freely configurable standard solution that offers various client authentication and authorization mechanisms.

3. How can the overall application be monitored in a simple, efficient way? What metrics are available and how can administrators proactively intervene during operations. What possibilities are given for post-mortem analyses? HiveMQ offers a control center for the monitoring of essential metrics, tracing of individual clients and extensive logging, as well as the possibility of connecting external monitoring systems to visualize a large number of relevant metrics.

4. How can additional business-specific processes be integrated in a simple and efficient way? The HiveMQ Extension SDK offers a flexible API to address almost any conceivable integration challenge.

5. Which properties for the MQTT actors involved are required for a well-designed implementation of the architecture? MQTT 5 tends a lot of new features that would help to implement the scenario described in VDA 5050 elegantly, such as, User properties, payload format indicator or content types or the request response pattern.

This document also contains proposals for broker configurations and examples for the MQTT client setups useful in the Order - State Communication Scenario.

Short Introduction of the VDA 5050 Specification

In the VDA 5050 from August 2019 version 1.0. an interface for communication between driverless transport vehicles (AGV) and a control system is described. The definition of a communication interface for driverless transport systems (AGVs) aims to simplify the connection of new vehicles to an existing control system. A standardized communication interface also allows to operate AGV's from different manufacturers in parallel within the same work environment. VDA5050

The specification includes:

- The definition of a basis for the integration of transport systems in a continuous process automation.
- Increasing flexibility through increased autonomy of vehicles.
- Reduction of implementation time via „Plug & Play" mechanisms.
- The use of a central control system with the corresponding logic for all vehicle types and manufacturers.
- A common interface between vehicle and control system.
- The possibility of integrating proprietary control systems.

To implement the goals, the VDA 5050 specifies

- MQTT as communication protocol between actors
- JSON as data format

What are the Challenges and Requirements

MQTT has become the IoT standard for connecting devices in general and also for AGVS. Now the challenge is to describe and specify what the best solution would look like. Currently, a number of MQTT Implementation solutions are available on the market, providing different features. Beside the support of the MQTT protocol itself, additional key features are necessary to run a secure and stable system.

Challenges are:

- The VDA specification describes a couple of data related traits that could be perfectly described with MQTT 5 features. Therefore, a MQTT solution should support all MQTT 5 features.
- Security is a very important key feature that should be supported both by the client and the broker.
- To support system changes without downtime, the solution needs to be highly available and support rolling upgrades and migration scenarios.
- The MQTT Broker must be scalable and support a growing number of vehicles
- To Integrate other systems or add business functionality via plug & play, the MQTT Broker should be extensible.
- The Data format is specified by the VDA and should be validated before publishing, to make the solution more robust and secure.
- Central monitoring and tracing for specific clients, topics or messages is essential in the production environment.

Components of the Architecture Proposal Using HiveMQ

The following paragraph describes the components required for a robust MQTT-based solution that should be used to implement the VDA specification.

Message Broker - HiveMQ

HiveMQ is an enterprise-ready MQTT broker that is specifically tailored to business needs of IoT use case scenarios.

The HiveMQ enterprise MQTT broker provides fast, efficient, and reliable movement of data to and from connected IoT devices. HiveMQ fully implements the MQTT protocol version 5 and 3 and can support mixed version scenarios.

HiveMQ Extension System

HiveMQ offers a free, open-source extension SDK. The HiveMQ extension framework provides an open API that allows developers to create custom extensions that suit their specific infrastructures. The extension framework can be used to extend HiveMQ with custom business logic or to integrate virtually any system into HiveMQ.

A couple of pre build extensions are available on the HiveMQ Marketplace.

Actors

Clients are all participants that use an interface to communicate via MQTT. In this scenario we have the following clients: the Control System (CS), and any Automated Guided Vehicle (AGV)

A lot of MQTT 5 client implementations are available for different languages, for example Eclipse PAHO – with MQTT 5 support for C and C#, MQTTnet for .net environment and HiveMQ MQTT Client, an open source MQTT Java library that was developed for use in high-performance scenarios and low memory requirements.

The HiveMQ MQTT client library includes following features with:

- Complete implementation of all MQTT 5.0 and 3.1.1 features, including TLS / SSL
- Message persistence, Offline buffering, Automatic Reconnect, Backpressure support,
- Websocket support, HTTP CONNECT support, and many other features

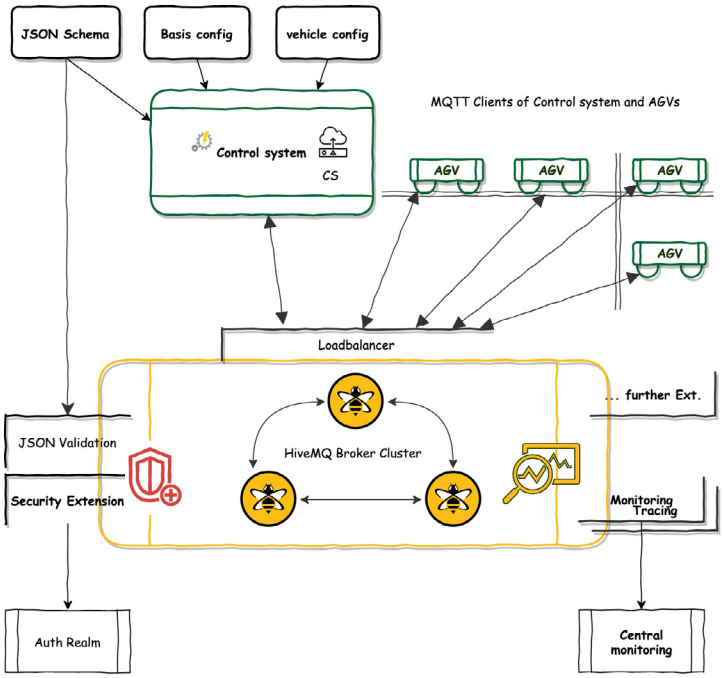
Related Components

The figure beside shows the components of the architecture that are related to each other in the MQTT relevant environment.

From an external vantage point, the HiveMQ message broker operates as one logical unit, internally as a cluster of nodes. The Broker uses the security extension, a standard solution for authentication and authorization, an extension for JSON payload validation, and an extension for tracing and monitoring. Metrics are provided in JMX format that can be monitored centrally. The same applies to the event logs and other logging information.

The actors of the HiveMQ broker are the MQTT clients. As a main use case, the control system (CS) client sends information to the driverless transport vehicles (AGV) and consumes the relevant status or possible error information of the vehicles.

The data for the central control system (CS) is provided in the form of a basic configuration and the device configurations for each vehicle (AGV). The JSON formats to be used are defined in JSON schemes. The schemes are available and can be used by the broker as well as the control system (CS) for validation purposes.



All clients (control center and transport vehicles) communicate with the broker via a load balancer.

Message Broker Features

The central MQTT infrastructure component is the HiveMQ MQTT broker. The MQTT Broker has to be fail-safe, robust and scalable, and providing the necessary performance guarantees. HiveMQ is a MQTT broker supporting 100% of the MQTT protocol in all MQTT versions (MQTT 3 and MQTT 5). HiveMQ is reliable, scalable fail-safe and high performing. The following features should be configured for the Message broker to enable these requirements:

Key Features: Reliability, Scalability, Performance

RELIABILITY

The cluster is set up so that incoming messages and data from the clients in the cluster are distributed evenly across all nodes. The replication of the data in the cluster should be configured relative to the number of nodes, whereby at minimum, the replication of 1 - all data is replicated at least once - is set.

SCALABILITY

A dynamic cluster configuration is recommended in order to be able to carry out migration scenarios during operation without down-time or to optimally process a growing number of clients or messages.

A HiveMQ MQTT broker cluster consists of several individual HiveMQ nodes. Each node can experience a different stress level at any given time due to the number of MQTT PUBLISH messages to process, retained messages, client connect rates, queued messages and other operations that can cause overload for an individual broker.

HiveMQ provides built-in cluster overload protection. Each HiveMQ cluster node is able to reduce the rate of incoming messages from MQTT clients that significantly contribute to the overload of the cluster.

This mechanism improves the resilience of a HiveMQ cluster dramatically as individual MQTT clients can be throttled if the HiveMQ cluster experiences an overload. With this mechanism, HiveMQ is able to recover itself from stress situations (ill-behaved clients or DDOS Attacks) without notable service degradation for all other MQTT clients. A Clustering HiveMQ paper describes this in detail.

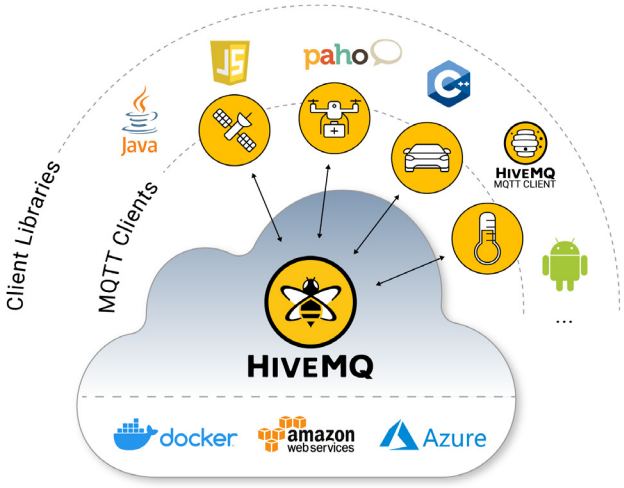
PERFORMANCE

HiveMQ with its extension system, including the provided open source and enterprise extensions, is 100% designed to be performant in every way. HiveMQ is a highly scalable Enterprise MQTT Broker designed for lowest latency and very high throughput.

A lot of performance tests and production systems with million devices and high throughput prove the performance. Benchmarks are available here: HiveMQ Benchmark 10 million and HiveMQ 3 Benchmark on AWS.

RUNS EVERYWHERE

At last, in short - HiveMQ can be setup in any environment, like VMs, Docker on VMs, Cloud Environments like Amazon, Azure or Google Cloud and of course on bare metal with Linux systems. It works perfectly with Cluster Management Systems like Kubernetes or Application Platforms like OpenShift.



HiveMQ runs in many different environments and connects to a wide range of clients and libraries.

Monitoring, Logging, MQTT Tracing

HiveMQ offers a Control Center to monitor the behavior of the message broker and provides a fundamental set of metrics. Operators can also view the status of a specific MQTT client and setup trace recordings of the messages between a specific client and the broker. This allows for more effective troubleshooting of deployed systems.
A couple of pre-build extensions are available to monitor the MQTT Infrastructure by the HiveMQ metrics.

The HiveMQ Logging system uses the standard logging framework. Log files can be configured in a fine-grained fashion. Furthermore, all client related MQTT events are logged in the event log file:

- Client Connection
- Client Disconnection
- Dropping of Messages
- Client Session Expiry

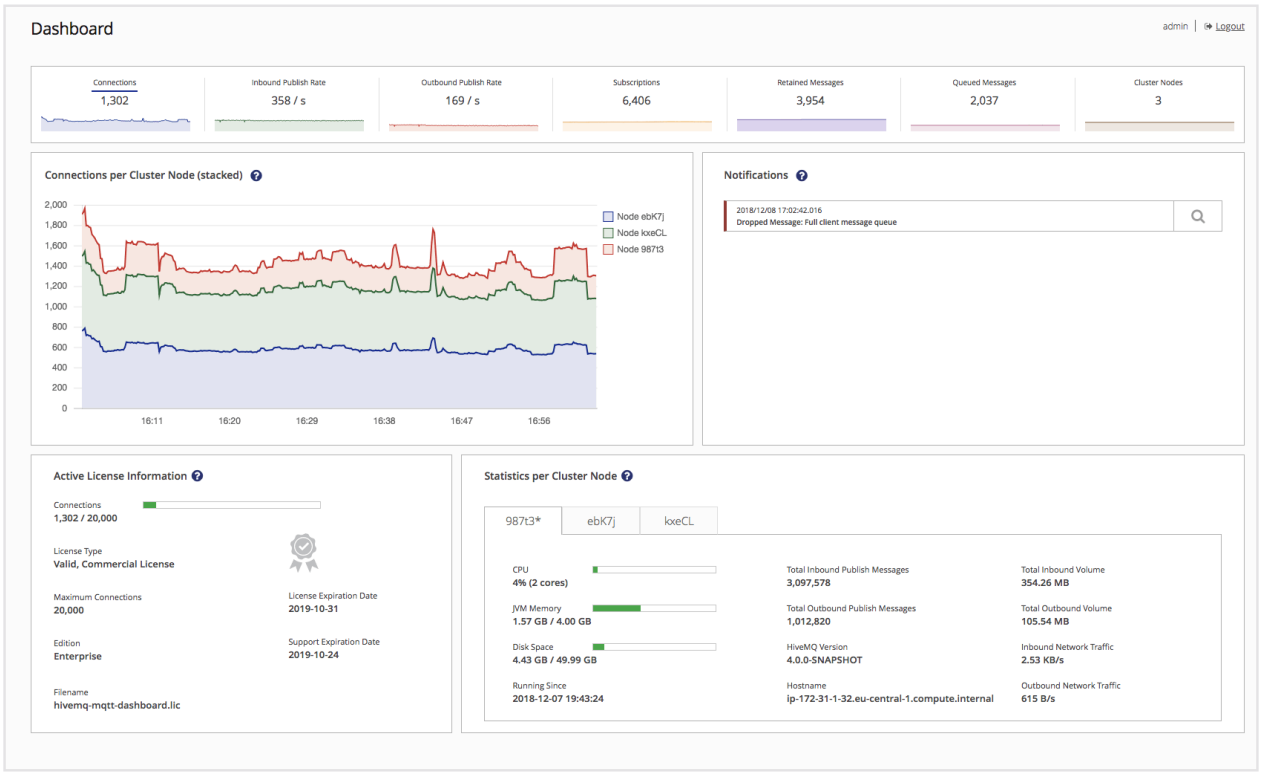
HiveMQ logs can be streamed into a central logging system.

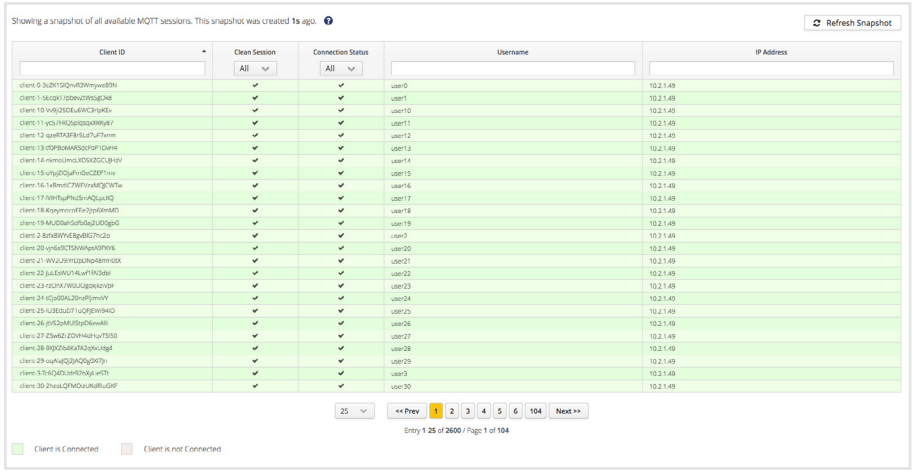
The HiveMQ Control Center should be used for human monitoring the systems, esp. MQTT message throughput, specific kind of errors for MQTT message lost and the trace-log.

The Control Center should only be accessible from trusted IPs. Specific Access Roles can be defined within the usage of the Enterprise Security Extension.

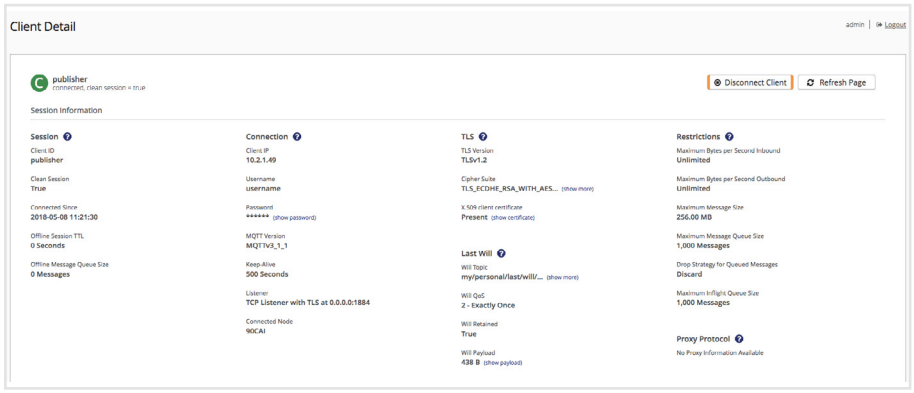
The HiveMQ Control Center provides a couple of useful administration views.

The Dashboard View allows an administrator to monitor the overall health of a HiveMQ deployment. It provides real-time monitoring on the number of client sessions, inbound/outbound publish rate, subscriptions, retained messages and queued messages. Each individual HiveMQ node can also be queried for performance stats pertaining to the node.

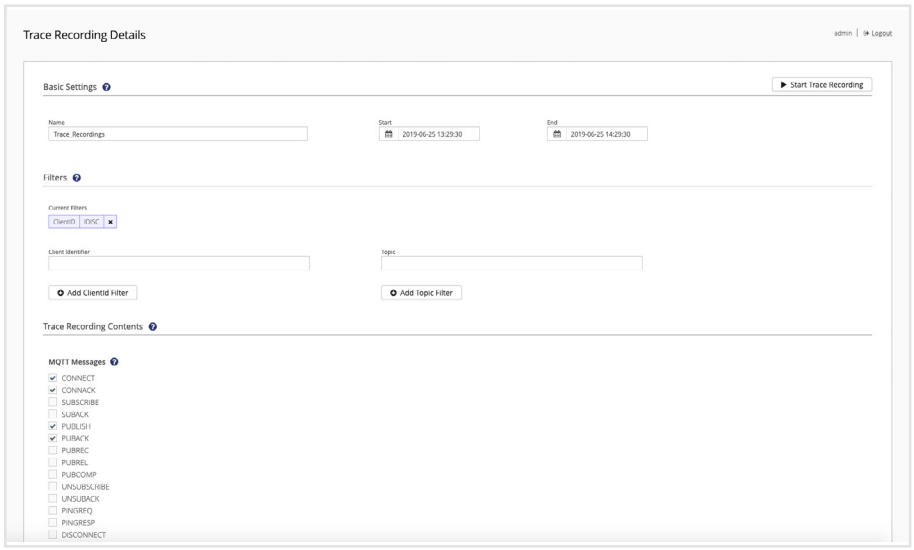




The Client Overview provides a snapshot of all the MQTT client sessions. This overview allows for filtering based on Client ID, Connection Status, Client name and IP address. From the client overview an administrator can drill down into a specific client information.



The client detail view provides all the detailed information about an MQTT client session, including client IP, Keep Alive time period, TLS information, Last Will and Testament, etc. An administrator is also able to disconnect clients or remove a client session and add and remove subscriptions.



At last, HiveMQ CC provides MQTT Trace Recordings. A Trace Recording is a combination of filters which allows you to select messages of specific clients or topics, which are logged to a file in a human readable format. Trace recordings are very useful for diagnostics and debugging irregular behavior of a client.

Beside the core metrics, that are visible in the Control Center, HiveMQ provides several hundred metrics that are very helpful and can be used for monitoring. Monitoring is an important part of operations for any application and HiveMQ is no exception. It is not difficult to create a monitoring setup for HiveMQ for Tools like Prometheus, InfluxDB and Grafana. For these tools there are open source extensions available, so that it is easy to plug in the needed functionality. Also, an initial Grafana dashboard template is available as a starting point.

Security Expectations

Security in general is a major feature in every IoT Application, and should be supported from the beginning.

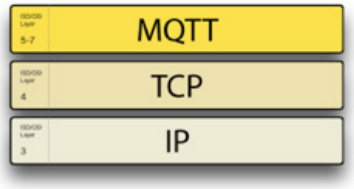
Security can be divided in the areas where it occurs.

Security at the transport level

Since MQTT is based on TCP, all communication can generally be encrypted with TLS. The latest version for TLS should be used if possible.

TLS provides

- Encryption
- Authentication
- Data Integrity



When TLS is used correctly, a third-party observer can only infer the

- Connection endpoints,
- Type of encryption, as well as the frequency and
- an approximate amount of data sent,

but cannot read or modify any of the actual data.

Finally, only those protocols and ports that are actually used, should be allowed in the broker *configuration*.

Security at application level

Each connected client should be authenticated during the connect phase and only authenticated Clients should be able to send messages or subscribe to topics. Authentication information should be stored secure and safe encrypted. Management tools for the authentication infrastructure – if used, should be available.

CLIENT AUTHENTICATION

The way to authenticate is not specified in the VDA5050 specification, therefore we will reference the various ways on how to authenticate MQTT clients as described in MQTT Security Fundamentals.

The HiveMQ Enterprise Security Extension provides a standard solution that supports different authentication methods for the participating MQTT clients, described in the chapter below.

CLIENT AUTHORIZATION

All MQTT clients involved, should only have access to the topics that are relevant. This can be accomplished by applying a role-based model where the permissions for topics can be assigned to specific roles. MQTT clients can be defined as owners of these roles.

Two roles are visible in the scenario described in VDA 5050. (Control system and vehicle). The correct authorization and assignment of these roles before publishing and subscribing to any topic must also be achieved. In general, a Whitelist approach, including pattern replacement at client specific topic filtering should be used. That means, no wildcard subscription or access to any other topic, that is not used in the scenario should be allowed.

The CS-Client subscribes with the “+” wildcard pattern for the placeholders and can publish to the order topic for each client. The AGV Clients should only have pub/sub right for their own topic path, specified by the key information like interface name, version, manufacturer and the serial number.

Enterprise Security Extension

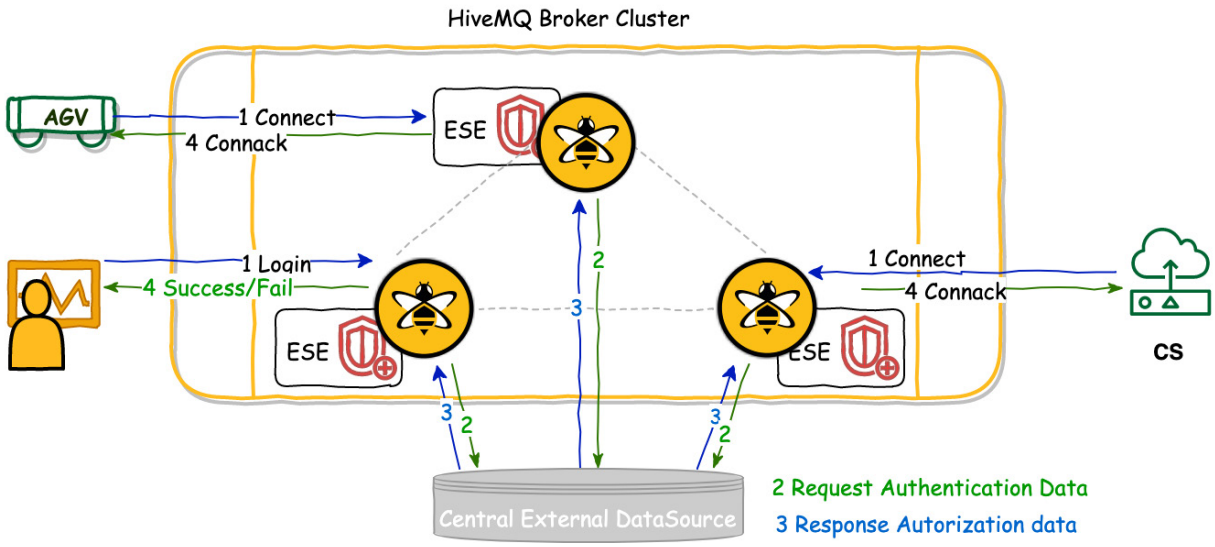
The HiveMQ Enterprise Security Extension can be used for Authentication and Authorization. The HiveMQ ESE expands the role, user, and permission-management capabilities of HiveMQ Enterprise and Professional editions. HiveMQ ESE allows you to use different sources of external authentication and authorization data to authenticate and authorize MQTT clients. In the HiveMQ ESE, you define realms to partition your server into protected areas that can each have their own authentication and / or authorization scheme.

The HiveMQ ESE processes incoming client connections in highly configurable pipelines that offer customizable stages to handle the authentication and authorization of your clients.

The Main Features of the Security Extension

- The Extension provides Security integration patterns for Username & Password, OAUTH 2.0 (JWT), LDAP, SQL Database and x.509 client certificates are available.
- The Extension can also be used for RBAC to the HiveMQ Control Center.
- It is easy to switch Authentication variants, as different variants for the different clients are possible at the same time.
- The Authentication mechanism can differ from authorization mechanism. So as an example, authentication can be done with LDAP and authorization with an external RBAC system.
- There is no further implementation effort for the most common security methods.

A detailed description for different usage possibilities for the security extension can be found in the documentation of the HiveMQ ESE.



Rough description of the mechanism of authentication and authorization with the Security Extension and a central external data source where authentication data are handled.

SECURITY BY DATA INTEGRITY

As specified in VDA 5050 JSON is the preferred payload format. Furthermore, a couple of information about the message content and the client itself could be handled with the usage of MQTT 5 Features. With the usage of these features, it is not necessary to parse the payload, to decide about basic data correctness.

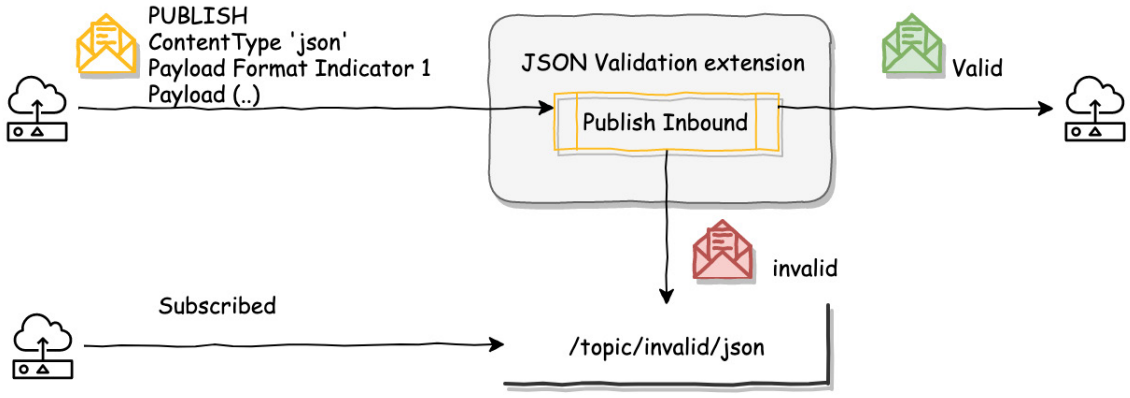
MQTT 5 specific features can be used to specify meta information.

- The Feature 'payload format indicator' can be used to indicate that the payload is UTF-8.
- The Feature 'content-type' can be specified by setting it to 'application/json'.
- User properties can be used to specify client specific header information.

Additionally, data integrity can be ensured by the verification of the payload.

JSON Schema Validation

- Each PUBLISH Message can be parsed against the expected JSON-Schema to verify if the right content is sent.
- Parsing can be executed on the broker side by a generic JSON Validation Extension.
- Errors can be handled on the broker side in a generic manner and published to a pre-configured topic.



PUBLISH Flow with usage of a validation extension.

SECURITY AT INFRASTRUCTURE LEVEL

At the infrastructure level a couple of security actions need to be taken:

- Only expected traffic should be forwarded to downstream systems, so if UDP traffic is not used by MQTT, this should be blocked in general.
- Traffic should only be allowed to ports needed for the MQTT system (1883, 8883).
- The Operating System should be kept updated and libraries should be kept clean.
- Topic namespaces must be separated. Manufacturers may not consume data of other manufacturers.
- The MQTT broker configuration should limit the maximum message size to the maximum needed for the use case.

ORDER-STATE USE CASE

The main use case, described in the VDA 5050 specification describes the necessary communication flow for order and status information exchange between AGV and CS. This use case will be considered in more detail in the following chapter.

ORDER - STATE MQTT MESSAGE FLOW

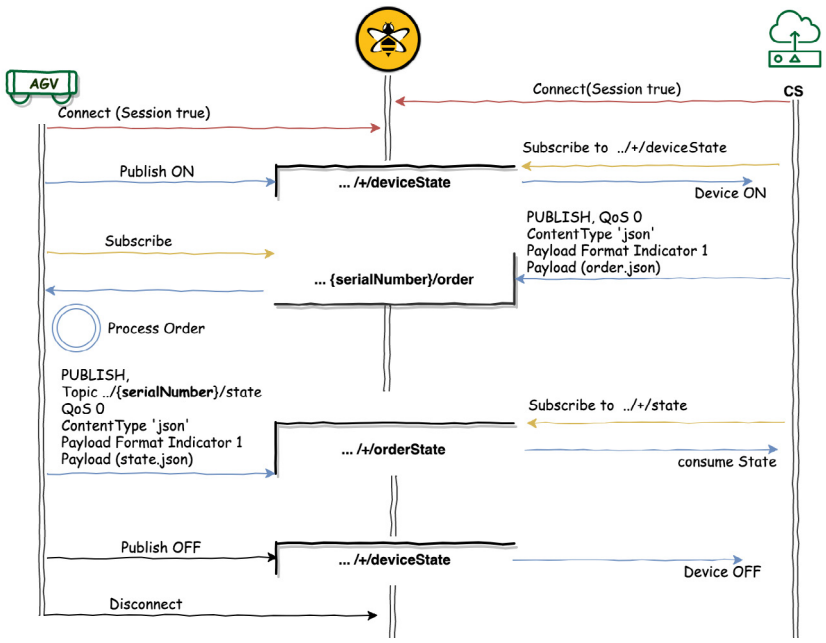
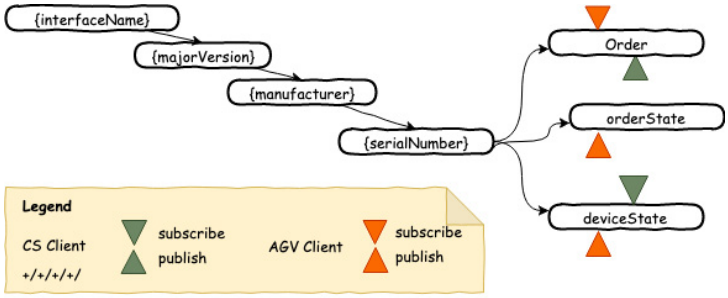


Figure 5: Order Example Flow



AGV-CS Order Information Topic structure

TOPIC STRUCTURE

In the specific scenario for order and state information exchange, the topic structure could be designed as following

The CS-Client subscribes with the "+" wildcard pattern for the placeholders and can publish to the order topic for each client. The AGV Clients should only have pub/sub right for their own topic path, specified by the key information like interface name, version, manufacturer and the serial number.

A set of topics that are used to handle the business logic are described in the VDA 5050. Each topic starts with a subtopic, with dynamic placeholder to separate different communication partners and devices.
Each AGV should use its own Basis topic.
`{interfaceName}/{majorVersion}/{manufacturer}/{serialNumber}/`

Subscriptions and publishes to topics that are not matching the base topic, should be forbidden by default within the authorization process for any participating client.

The subtopics **order** and **orderState** will be used for information exchange.
The topic **deviceState** is useful to get information when a AGVs is going online and offline to handle error cases and initialize devices in specific circumstances.

Each AGV subscribes to their own order topic:
`{interfaceName}/{majorVersion}/{manufacturer}/{serialNumber}/order`
and publishes to the corresponding orderState topic.

The control system subscribes to each state topic of all AGV clients.
`+/+/+/+/ orderState`

and to each deviceState topic of all AGV clients.
`+/+/+/+/ deviceState MQTT Client characteristics`

MQTT Client characteristics

Each AGV Client has a unique Client Identifier – derived from the keys
`{interfaceName}-{majorVersion}-{manufacturer}-{serialNumber}`

The Control System can be designed as one backend client, or if client load balancing is needed, as a group of Control System clients. In this case shared subscriptions must be used, so that incoming messages (order and device States) are distributed over the array of control system clients. Each client should also have a unique identifier like CS-<ID>

All clients should work with sessions, to receive messages that are sent during offline state by the client. Automatic reconnect should also be defined.

Clients should be implemented using MQTT 5 for this use case if possible.

MQTT Publish - Orders and State

The CS Client publishes an Order to each AGV as a MQTT Message with the following characteristics:

- QoS: 0
- Retained
- Identifier (CS-<Number>)
- Payload Format: UTF-8
- Content Type: JSON
- Message Pattern: Request-Response with correlation data (orderId)
- Correlation Data
  - OrderId
- User Properties:
  - JSON Schema / Uri
- Payload: JSON ORDER description

After consuming the Order-Message by the AGV client and processing the Order or parts of it, the corresponding Order State has to be published to the orderState Topic.

The MQTT Publish of the AGV has following characteristics:

- QoS: 0
- Retained Flag
- Identifier (interface name, major Version, manufacturer, serial Number)
- Payload Format: UTF-8
- Content Type: JSON
  - Message Pattern: Request-Response with correlation data (OrderId)
- Correlation Data
  - OrderId
- User Properties:
  - interface name, major Version, manufacturer, serial Number
- Payload: JSON state description

The Order Id will be used as Correlation Data in both Publish Messages, so that the AGV and the CS Client can identify the Order from a message without reading the payload.

MQTT Will, Client offline and online event detection

An AGV client can be disconnected by the broker or a client side action. If the client goes offline by disconnecting itself, this state should also be determined at the control system. In both cases a retained message with state OFF should be sent by an AGV to the deviceState topic.

`{interfaceName}/{majorVersion}/{manufacturer}/{serialNumber}/deviceState`

To get the device state from the clients, each client can define a last will retained message and set up the WILL Publish during CONNECT. If the Client goes offline, the WILL message will be send to the specified topic.

Additionally, each client can publish a message to the deviceState Topic to signal its ONLINE state.

The Will and the device State Message should contain:

- QoS 0
- Retained Flag
- Payload Format: binary
- Payload (ON/OFF – Byte)
- User Properties
- interface name, major Version, manufacturer, serial Number

With MQTT 5, sending the **will** message can be configured with a delay. This could be useful in cases where short interruptions, with no influence. In such cases, the status change should not send.

The Keepalive (heartbeat) can be configured on the broker and client with a common default value of 60 seconds.

Conclusion

The challenge for use cases resulting from the VDA 5050 specification to describe a best solution can be implemented with the means of MQTT 5 and the use of a broker, such as HiveMQ, who offers the necessary key functions to operate a safe and stable system. Data-related features such as specific formats and their validation, as well as the use of metadata can be perfectly described with MQTT 5 functions. There are already implementations for different languages and HiveMQ supports both MQTT 5 and MQTT 3 clients in mixed scenarios.

Security is a very important key feature. The use of standard solutions that support security in different variants is optimal if different manufacturers are to be integrated within one control system.

To support system changes without downtime, the solution must be highly available and support ongoing upgrades and migration scenarios. HiveMQ is highly scalable and can support a growing number of AGVs without problems.

HiveMQ offers a standard solution, the Control Center, for monitoring and tracking for specific clients, topics or messages in the production environment.

In order to integrate other systems such as a central log system or add any business functionality, HiveMQ can be expanded via the extension system. A large number of open source solutions and standard solutions are already available. With the help of the Open HiveMQ Extension API, quite each business-specific solution can be implemented.

The combination of HiveMQ and MQTT 5 to implement the VDA 5050 use cases, would be a perfect solution.

Appendix

Client Implementation with HiveMQ MQTT library

Use case Communication of order and status information
The MQTT Clients (AGV and CS) examples are using the Java HiveMQ MQTT 5 Client library in the example code snippets.

AGV CLIENT
A Client is defined as MQTT 5 Client. A unique identifier is set. Automatic Retry is configured. The Identifier should be created from the information about interface name, major version, manufacturer and serial number.

```
this.client = MqttClient.builder()
        .serverHost(BROKER_HIVEMQ_ADR)
        .serverPort(BROKER_HIVEMQ_PORT)
        .useMqttVersion5()
        .identifier(uniqueIdentifier)
        .automaticReconnectWithDefaultConfig()
        .buildBlocking();
```

The user properties, containing client information needed during information exchange. A retain will message is created, to leave the device state at the device state topic if the client lost connection.

```
this.will = Mqtt5Publish.builder()
        .topic(deviceStateTopic)
        .qos(MqttQos.AT_MOST_ONCE)
        .userProperties(clientProperties)
        .payload(OFF)
        .retain(true)
        .build();
```

At Start the MQTT client will be connected with clean Start, setting the Session expiration interval, the user properties and the will message.

```
Mqtt5ConnAck ack = client.connectWith()
        .cleanStart(cleanStart)
        .sessionExpiryInterval(expiryInterval)
        .userProperties(clientProperties)
        .willPublish(will)
        .send();
```

In success case, that can be figured out by the CONACK Message the AGV client has to subscribe to Order Topic with QoS 0.

For subscribing, the client behavior can be switched to asynchronous client mode, to easily work with acknowledgements, errors and success cases. In case of retrieving a message (an order) on this topic, the Order will be consumed and processed by the callback doConsumeOrder. The SUBACK Message can be evaluated and error handling can be done in this case.

```
final Mqtt5Subscription subscriptionOrder = Mqtt5Subscription.builder()
        .topicFilter(orderTopic)
        .qos(MqttQos.AT_MOST_ONCE).build();

client.toAsync().subscribeWith()
        .addSubscription(subscriptionOrder)
        .callback(this::doConsumeOrder)
        .send()
        .whenComplete((subAck, throwable) -> { /** Error handling **/ });
```

```
private void doConsumeOrder( Mqtt5Publish publish) {
    // JSON is validated by the extension,
    // Can be assumed, that it is valid.
    String jsonOrder = new String(publish.getPayloadAsBytes());
    String jsonState = doOrder(jsonOrder);
    publishState(jsonState);
}
```

Using a validation extension that checks incoming MQTT publish messages directly on the broker, the content can be assumed to be valid and can be processed by the client.

After consuming the Order information, the AGV client has to publish to the orderState Topic. The PUBLISH will be sent with QoS 0 and in addition sets also content-type and payload-format.

```
final Mqtt5Publish state = Mqtt5Publish.builder()
        .topic(orderStateTopic)
        .qos(MqttQos.AT_MOST_ONCE)
        .userProperties(clientProperties)
        .contentType(„application/json“)
    payloadFormatIndicator(Mqtt5PayloadFormatIndicator.UTF_8)
        .payload(jsonState.getBytes())
        .build();

client.toAsync().publish(state)
        .whenComplete((publishResult, throwable) -> { /** Error handling **/ });
```

At least for the device state a message can be sent, that informs the subscribed CS client about the liveness and readiness of the AGV client.

```
final Mqtt5Publish state = Mqtt5Publish.builder()
        .topic(deviceStateTopic)
        .qos(MqttQos.AT_MOST_ONCE)
        .userProperties(clientProperties)
        .payload(ON)
        .retain(true)
        .build();

client.toAsync().publish(state)
        .whenComplete((publishResult, throwable) -> { /** Error handling **/ });
```

If the AGV client has to be disconnected or switched off, the device State OFF can be sent in a similar way. The DISCONNECT can also be sent with user properties as additional information.

```
void stopAGVClient() {
    doSentDeviceState(OFF);
    client.disconnectWith()
            .reasonString(„Device graceful shutdown“)
            .userProperties(clientProperties)
            .send();
}
```

CS CLIENT

Similar to the AGV client, one or more CS Clients connect as MQTT 5 Client. A unique identifier is set. This could be a pattern like CS_<ID>. The MQTT CONNECT initializes the session with a clean start and sets the expiry interval. Automatic retry is configured.

At the start the CS Client could check the state of all vehicles from the information, published to the deviceState Topic, where the message is stored as a retained message. For this the CS client must subscribe to the deviceState.

Based on this information, the CS client could send new Orders only the active AVGs.
The target topic can either be derived from the user properties or from the topic where the messages arrived.
The Payload Format is the marked as UTF-8 and with content type application / json. To have the OrderId available, without parsing the payload, the order Id should be transported as correlation data in the order related MQTT Publishes.

```
void doPublishOrder(Mqtt5UserProperties avgClientProperties) {
    //depending from the content described by the properties,
    //the vehicle can get informed about the next actions
    final String avgClientBaseTopic = getTopicFromProperties(avgClientProperties);

    final Mqtt5Publish state = Mqtt5Publish.builder()
            .topic(avgClientBaseTopic)
            .qos(MqttQos.AT_MOST_ONCE)
            .userProperties(avgClientProperties)
            .payloadFormatIndicator(Mqtt5PayloadFormatIndicator.UTF_8)
            .contentType(„application/json“)
            .payload(JsonOrder)
            .correlationData(orderId)
            .retain(true)
            .build();

    client.toAsync().publish(state)
            .whenComplete((pubResult, throwable) -> { /** Error handling **/ });
}
```

Additionally, at the beginning, the CS Client subscribes to the orderState Topic of all AGV clients by using the wildcard subscription (+) to get information about the order state for all vehicles. Using several CS clients, a shared subscription has to be used here.

```
client.toAsync().subscribeWith()
        .addSubscription('+/+/+/+/orderState')
        .callback(this::doConsumeState)
        .send()
        .whenComplete((subAck, throwable) -> { /** Error handling **/ });
```

By getting a callback, when an AGV client publishes its State to the orderState Topic, this information has to be consumed.

```
private void doConsumeState(Mqtt5Publish publish) {
    final Mqtt5UserProperties clientProperties = publish.getUserProperties();
    //depending from the content of the props (position, client information...)
    doPublishOrder(clientProperties);
}
```

The CS Client can consume the information and react for example by publishing a new Order.

HiveMQ Configuration recommendations

A Message Broker cluster with 3 nodes is a robust and safe installation, able to handle the described scenario during normal operations and peak load situations.

Some general recommendations:

- Cluster discovery can be configured static or dynamically, depending on the underlying infrastructure. The open source Extensions for DNS or S3 Cluster discovery can be used here. HiveMQ Docker files, also with Cluster mode are available on Docker Hub.
- If the Message Broker must be maintained in a public zone, Cluster communication should be secured with TLS.
- MQTT Listeners should be set up with TLS, use 8883 as the standard TLS port.
- Replication count could set up to 2, so that the cluster is fully replicated.
- MQTT Restrictions for Message Expiry, Session Expiry, aligned to the use case.
- Restriction for maximum amount of connections, as expected from the use cases.
- Configure an overload protection with burst rate and normal rate.
- For Authentication the HiveMQ ESE is recommended.
- The Access to the HiveMQ Control Center should be handled via the HiveMQ ESE.
- For Monitoring the open source Prometheus- or InfluxDB extension in combination with a Grafana Dashboard are recommended.

All configurations, including requirements, best practice and default settings can be found in detail described at the HiveMQ Documentation.

Glossary

VDA5050	SCHNITTSTELLE ZUR KOMMUNIKATION ZWISCHEN FAHRERLOSEN TRANSPORTFAHRZEUGEN (FTF) UND EINER LEITSTEUERUNG
AGV	Automated Guided Vehicle
AGVS	Automated Guided Vehicle System
CS	Control System

Ergoldingerstr. 2a
84030 Landshut
Germany

www.hivemq.com
© HiveMQ GmbH 2020