COVESA VISS version 3.0-Transport Examples

COVESA

07 January 2025

▾ **More details about this document**

Latest published version:
none

Latest editor's draft:
https://github.com/COVESA/vehicle-information-service-specification/blob/gh-pages/spec/VISSv3.0_TransportExamples.html

History:
Commit history

Editors:
Ulf Bjorkengren (Ford Motor Company)
이원석(Wonsuk Lee) (한국전자통신연구원(ETRI))

Feedback:
GitHub COVESA/vehicle-information-service-specification (pull requests, new issue, open issues)

Abstract

The Vehicle Information Service Specification (VISS) is a service for accessing vehicle information, signals from sensors on control units within a vehicle's network. It exposes this information using a hierarchical tree like taxonomy defined in COVESA Vehicle Signal Specification (VSS). The service provides this information in JSON format. The service may reside in the vehicle, or on servers in the internet with information already brought off the vehicle.

This specification describes a third version of VISS which has been implemented and deployed on production vehicles. The first version of VISS only supported WebSocket as a transport protocol, the second version is generalized to work across different protocols as some are better suited for different use cases. The second version added support for the HTTP and MQTT transport protocols, subscription capabilities was improved and an access control solution was added.

There are three parts to this specification, [viss3-core], [viss3-payload-encoding], and TRANSPORT-EXAMPLES. This document, the VISS version 3.0 TRANSPORT-EXAMPLES specification, describes the VISSv3 transport protocols, and the mapping of the message layer on these transports. The companion specification [viss3-core] describes the messaging layer, and [viss3-payload-encoding] describes payload encodings to/from the primary JSON payload format.

Table of Contents

§ 1. Introduction

This document provides examples of how the message payloads defined in [viss3-core] are used together with different transport protocols. The Websocket ([RFC6455]) transport protocol is used to give an example of how the JSON primary payload format is directly used. This is followed by

examples of other transport protocols where exceptions from the primary payload format are defined and exemplified.
The Vehicle Information Service Specification, version 3.0 permits any transport protocol to be used that can transport the unaltered primary payload format. Transport protocols that require modifications or transformations of the primary payload format *MUST* define this in either this document or in the [viss3-payload-encoding] document. The transport protocols that define deviations in this document are listed below.

Protocol name	Reference
HTTP	[RFC9112]
MQTT	[MQTT]
gRPC	[gRPC]

The Websocket protocol is used to provied an example for protocols that does not introduce any deviations so therefore it is not shown on the list above.

§ 2. Conformance

As well as sections marked as non-normative, all authoring guidelines, diagrams, examples, and notes in this specification are non-normative. Everything else in this specification is normative.

The key words *MAY*, *MUST*, *SHALL*, and *SHOULD* in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

§ 3. Terminology

The acronym 'VISSv3' is used to refer to this document, the VISS version 3 specification. The acronym 'VSS' is used to refer to the 'Vehicle Signal Specification' which is defined by the COVESA Alliance. The term 'WebSocket' when used in this document, is as defined in the W3C WebSocket API and the WebSocket Protocol.

§ 4. Transport Common Definitions

This chapter defines features that *SHALL* be common for all transport protocols.

§ 4.1 Status Codes

A server implementing this specification *SHALL* support the error codes, error reasons and error messages shown in the table below, for all supported transport protocols. The server may choose to dynamically replace the error message as described in the sub-chapters
The client *MAY* support any status code defined in [RFC2616].

Error Number (Code)	Error Reason	Error Message
400 (Bad Request)	bad_request	The request is malformed
400 (Bad Request)	invalid_data	Data in the request is invalid
401 (Unauthorized)	invalid_token	Access token is invalid
403 (Forbidden)	forbidden_request	The server refuses to carry out the request
404 (Not Found)	unavailable_data	The requested data was not found
408 (Request Timeout)	request_timeout	Subscribe duration limit exceeded
429 (Too Many Requests)	too_many_requests	Rate-limiting due to too many requests
502 (Bad Gateway)	bad_gateway	The upsteam server response was invalid
503 (Service Unavailable)	service_unavailable	The server is temporarily unable to handle the request
504 (Gateway Timeout)	gateway_timeout	The upsteam server took too long to respond

would it be possible to introduce some guidance or any further information
on when/after what delay these errors(service_unavailable, gateway_timeout) are supposed to occur?

§ 4.1.1 400 Bad Request Error Messages

what is the intention why this is not made mandatory?

This error code and reason shall be used for JSON schema related errors. The default error message is shown in the table above. The server may dynamically replace this by any of the error messages in the list below, or by any other relevant error message.

- Missing or invalid action
- Missing or invalid path
- Missing or invalid filter
- Missing or invalid value

§ 4.1.2 400 Invalid Data Error Messages

This error code and reason shall be used for errors that are not covered by the JSON schema but e. g. breaks a rule set by a VSS property. The default error message is shown in the table above. The server may dynamically replace this by any of the error messages in the list below, or by any other relevant error message.

- Update of a sensor is not supported
- Requested action on a branch is not supported
- Data value outside limit
- Incorrect data type

§ 4.1.3 401 Unauthorized Error Messages

This error code and reason shall be used for errors related to access control validation. The default error message is shown in the table above. The server may dynamically replace this by any of the error messages in the list below, or by any other relevant error message.

- Access token has expired
- Access token is missing

§ 4.1.4 404 Not Found Error Messages

This error code and reason shall be used when the server do not have access to the requested data. The default error message is shown in the table above. The server may dynamically replace this by any of the error messages in the list below, or by any other relevant error message.

- Data temporarily unaccessible
- Data is unknown

§ 4.2 Transport Payload

The payload *SHALL* have JSON format. See the JSON Schema chapter in the CORE document for the primary payload format.

§ 4.3 Authorization

If authorization is enabled on a signal requested by the client, it *MUST* provide a token to the server in order to verify that it is correctly authorized for the service it requests (see [viss3-core] document).Tokens are integrated in HTTP requests in the `Authorization` header. For WebSocket and MQTT requests an optional `authorization` property in the payload can be used.

§ 5. Transport Protocols

The transport protocols supported are the secure versions of HTTP, WebSocket, and MQTT, the latter on which a thin application layer protocol is applied.
The server *MUST* support the HTTP and WebSocket protocols, other protocols are optional.
Further transport protocols may be supported in future versions of this specification.

§ 5.1 Secure WebSocket

The WebSocket protocol is used in this document to provide examples of a protocol that does not apply any deviations to the primary payload format. As it does not implicitly provide a logical association between the request and response messages a key-value pair with the keyname "requestId" is added to the data components as described in the [viss3-core] document.
As the WebSocket protocol neither specifies a set of explicit methods, another key-value pair with the keyname "action" is also added. See 6.2 Action Definitions for the declaration of these key-value

pairs. All data components are mapped to the payload.

§ 5.1.1 Session Life Time Management

§ *5.1.1.1 Initialization*

If the client application is an HTML Application running in a web runtime or is a web page running in a browser, the WebSocket instance may either be instantiated natively or be created using a 'standards compliant' WebSocket JavaScript library.

A WebSocket can also be initiated from a native (e.g. C++) Application or from an Application written using a 'Managed Runtime' language like Java or C#. It is assumed that native and managed clients use a suitable standards compliant WebSocket library to request that a WebSocket connection is opened on the server.

Implementations that support additional devices or multiple VISSv3 services should provide discovery. Alternatively, the location of a particular VISSv3 Server instance on the local vehicle network may be handled by configuration, either as part of a package manifest or by consulting a registry on application install. The 'wwwVISSv3' hostname in this specification is used an example.

A client running on the vehicle is able to connect to the VISSv3 Server instance using the hostname e.g. 'wwwVISSv3' and uses the default port 443. The hostname 'wwwVISSv3' may locally be mapped to the localhost IP address 127.0.0.1 e.g. by adding an entry to the /etc/hosts file.

The sub-protocol name *SHALL* be 'VISSv3' with the digit $\overset{3}{2}$ being the version number. The sub-protocol version will be associated with exactly one VISS Server Specification version so that the client and server can correctly validate and parse request and response message packets.

```
var vehicle  = new WebSocket("wss://wwwVISSv3:443", "VISSv3");
```

The client *SHALL* connect to the server over HTTPS and request that the server opens a WebSocket. All WebSocket communications between the client and server *MUST* be over ‘wss’. Non encrypted communication is not supported, hence the server *MUST* refuse ‘ws’ connection requests.

This specification assumes that a single WebSocket is used to enable communication between a client application and the server. The client *MAY* open more than one websocket. However, the server *MAY* refuse to open a subsequent WebSocket connection and the client is responsible for handling this gracefully.

If more than one WebSocket connection is established between a client application and the server then each connection *MUST* be managed independently. For example, subscriptions created using a particular WebSocket connection shall only trigger event messages via that connection and the client *MUST* use that WebSocket connection to unsubscribe.

If more than one WebSocket connection has been established between one or more clients and a particular server instance, there is a risk that race conditions and concurrency issues could occur. An example of this would be where two or more WebSocket connections are used to update a particular setting at the same time.

Unless explicitly stated otherwise, the client *MAY* only assume that the server implements a simple concurrency model where lost updates and dirty reads could potentially occur if the server has more than one WebSocket connection open.

§ *5.1.1.2 Closure*

The WebSocket may be closed by either the client or the server by invoking the ‘close()’ method on the WebSocket instance.

The following example shows the lifetime of a WebSocket on the client:

```
// Open the WebSocket
var vehicle  = new WebSocket("wss://wwwVISSv3:443", "VISSv3");
…
// Close the WebSocket
vehicle.close();
```

The VISSv3 Server may terminate the WebSocket connection if it has not received a request for a period determined by the server. It is the client’s responsibility to handle this gracefully and to recover and request new subscriptions, where required.

§ 5.1.2 Transport Messages

§ *5.1.2.1 Read*

The client *MAY* send a getRequest message to the server to get the value of one or more vehicle signals. If the server is able to satisfy the request it *SHALL* return a getSuccessResponse message. If the server is unable to fulfil the request, e.g. because the client is not authorized to retrieve one or more of the signals, then the server *SHALL* return a getErrorResponse message. The structure of these message objects is defined below.

Object Name	Attribute	Type	Required
getRequest	action	Action	Yes
	path	string	Yes
	filter	string	Optional
	authorization	string	Optional
	requestId	string	Yes

Object Name	Attribute	Type	Required
getSuccessResponse	action	Action	Yes
	requestId	string	Yes
	data	object	Yes

In the table above the "data" attribute is either an object containing "value" and "ts" name/value pairs, or an array of such objects.

Object Name	Attribute	Type	Required
getErrorResponse	action	Action	Yes
	requestId	string	Yes
	error	Error	Yes
	ts	string	Yes

Example:
Request:

```
{
  "action": "get",
  "path": "Vehicle.Drivetrain.InternalCombustionEngine.RPM",
  "requestId": "8756"
}
```

Successful response:

```
{
  "action": "get",
  "requestId": "8756",
  "data":{"path":"Vehicle.Drivetrain.InternalCombustionEngine.RPM",
          "dp":{"value":"2372", "ts":"2020-04-15T13:37:00Z"}
         },
         "ts":"2020-04-15T13:37:05Z"
}
```

Error response:

```
{
  "action": "get",
  "requestId": "8756",
  "error": {"number": 404, "reason": "unavailable_data", "message":
"The requested data was not found."},
  "ts": "2020-04-15T13:37:00Z"
}
```

§ 5.1.2.1.1 Authorized Read

If the operation on the VSS node that is addressed requires authorization, then the request must contain the field "authorization" with its value being a JWT token. The token validation must be successful for a getSuccessResponse to be returned, else a getErrorResponse is returned. A token can be combined with all types of read requests.

Example:
Request:

```
                {
                  "action": "get",
                  "path": "Vehicle.Drivetrain.InternalCombustionEngine.RPM",
                  "authorization":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1...Zw_KSsds",
                  "requestId": "8657"
                }
```

Response:

```
                {
                  "action": "get",
                  "requestId": "8657",
                  "data":{"path":"Vehicle.Drivetrain.InternalCombustionEngine.RPM",
                          "dp":{"value":"2372", "ts":"2020-04-15T13:37:00Z"}
                         },
                  "ts":"2020-04-15T13:37:01Z"
                }
```

§ 5.1.2.1.2 Search Read

A client may issue a search read request to access multiple values in one request message. This is realized by adding a "filter" object following the paths filter operation described in the [viss3-core].

Example:
Request:

```
                {
                  "action": "get",
                  "path": "Vehicle.Cabin",
                  "filter": {"variant":"paths", "parameter":["Door.*.*.IsOpen",
 "DriverPosition"]},
                  "requestId": "5688"
                }
```

Response:

```
                {
                  "action": "get",
                  "data":[{"path":"Vehicle.Cabin.Door.Row1.Left.IsOpen", "dp":
 {"value":"false", "ts":"2020-04-15T13:37:00Z"}},
                          {...},…
                          {"path":"Vehicle.Cabin.Door.Row4.Right.IsOpen", "dp":
 {"value":"true", "ts":"2020-04-15T13:37:01Z"}},
                          {"path":"Vehicle.Cabin.DriverPosition", "dp":
 {"value":"1", "ts":"2020-04-15T07:00:01Z"}}
                         ],
                  "requestId": "5688",
                  "ts":"2020-04-15T07:00:02Z"
                }
```

§ 5.1.2.1.3 History Read

A client may issue a history read request to access recorded data points. This is realized by adding a "filter" object following the history filter operation described in the [viss3-core].

Example:
Request:

```
                {
                  "action": "get",
                  "path": "Vehicle.Acceleration.Longitudinal",
                  "filter": {"variant":"history", "parameter":"P2DT12H"},
                  "requestId": "5688"
                }
```

Response:

```
                {
                  "action": "get",
                  "data": {"path": "Vehicle.Acceleration.Longitudinal", "dp":
 [{"value": "0.123", "ts": "2020-04-15T13:00:00Z"}, {"value": "0.125", "ts":
 "2020-04-15T13:37:02Z"}]},
                  "requestId": "5688",
                  "ts": "2020-04-15T13:37:02Z"
                }
```

§ 5.1.2.1.4 Signal Discovery Read

that seems to have been removed

A client may issue a signal discovery read request to access dynamic metadata. A successful response will contain the requested metadata from all nodes of the subtree defined by the subtree root node that is addressed by the path. The static metadata, i. e. the metadata in the VSS tree, is retrieved by the setting the "type" to "static-metadata", and the parameter object to relevant static metadata.

Example:
Request:

```
                {
                  "action": "get",
                  "path": "Vehicle.Drivetrain.FuelSystem",
                  "filter":{"variant":"dynamic-metadata", "parameter":
 ["availability", "validate"]},
                  "requestId": "5687"
                }
```

Response:

```
                {
                  "action": "get",
                  "requestId": "5687",
                  "metadata": {"FuelSystem":
  {"availability":"available","validate":"read-write","children":["HybridType", ...
  ]}},
                  "ts": "2020-04-15T13:37:00Z"
                }
```

§ 5.1.2.2 Update

how does that work? setting multiple signals at once?

The client may request that the server sets the value of one or more signals e.g. to lock one or more doors or open a window by sending a setRequest message to the server. In the case of several signals being set, they *MUST* all be of the same data type, and be set to the same value. If the server is able to satisfy the request it *SHALL* return a setSuccessResponse message. If an error occurs e.g. because the client is not authorized to set the requested value, or the value is read-only, the server *SHALL* return a setErrorResponse message.

Object Name	Attribute	Type	Required
	action	Action	Yes
	path	string	Yes
setRequest	value	string/array/object what about number and boolean?	Yes
	authorization	string	Optional
	requestId	string	Yes

Object Name	Attribute	Type	Required
	action	Action	Yes
setSuccessResponse	requestId	string	Yes
	ts	string	Yes

Object Name	Attribute	Type	Required
setErrorResponse	action	Action	Yes
	requestId	string	Yes
	error	Error	Yes

Object Name	Attribute	Type	Required
	ts	string	Yes

Example:

Request:

```
{
  "action": "set",
  "path": "Vehicle.Drivetrain.Transmission.PerformanceMode",
  "value": "sport",
  "requestId": "5687"
}
```

Successful response:

```
{
  "action": "set",
  "requestId": "5687",
  "ts": "2020-04-15T13:37:00Z"
}
```

Error response:

```
{
  "action": "set",
  "requestId": "5687",
  "error": {"number": 404, "reason": "unavailable_data", "message": "The requested data was not found."},
  "ts": "2020-04-15T13:37:00Z"
}
```

§ 5.1.2.2.1 Authorized Update

If the operation on the VSS node that is addressed requires authorization, then the request must contain the field "authorization" with its value being a JWT token. The token validation must be successful for a setSuccessResponse to be returned, else a setErrorResponse is returned. A token can be combined with all types of update requests.

Example:
Request:

```
                {
                  "action": "set",
                  "path": "Vehicle.Drivetrain.Transmission.PerformanceMode",
                  "value": "sport",
                  "authorization":
 "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1...Zw_KSsds",
                  "requestId": "5687"
                }
```

Response:

```
                {
                  "action": "set",
                  "requestId": "5687",
                  "ts": "2020-04-15T13:37:00Z"
                }
```

§ *5.1.2.3 Subscribe*

The client may send a subscribeRequest message to request a subscription to one or more signals, thereby requesting the server to repeatedly return subscription event messages, as specified by the filter request described in the [viss3-core]. The server *MAY* reduce the number of subcriptionEvent messages sent to the client in order to reduce processing demands.
If the server is able to satisfy the request it *SHALL* return a subscribeSuccessResponse message. If an error occurs e.g. because the client is not authorized to read the requested value, the server *SHALL* return a subscribeErrorResponse message. If an error occurs during the subscription session, the server

shall return an subscriptionErrorEvent message.
The subscription variants are, as described in the [viss3-core] document:

- timebased: event messages are issued at a regular time interval,
- change: event messages are issued when the value has changed as specified,
- range: event messages are issued when the value is in the specified range,
- curvelog: event messages are issued when the buffer is full, and then processed according to the curve logging algorithm.

contradicts core which state in 5.1.3 that filter is mandatory. JSON schema does not allow empty filter

If none of the above trigger condition variants is specified, then an event message will be issued whenever the underlying vehicle system supplies a new data point to the server.

Object Name	Attribute	Type	Required
subscribeRequest	action	Action	Yes
	path	string	Yes
	filter	string	Optional
	authorization	string	Optional
	requestId	string	Yes

Object Name	Attribute	Type	Required
subscribeSuccessResponse	action	Action	Yes
	requestId	string	Yes
	subscriptionId	string	Yes
	ts	string	Yes

Object Name	Attribute	Type	Required
subscribeErrorResponse	action	Action	Yes
	requestId	string	Yes
	error	Error	Yes
	ts	string	Yes

Object Name	Attribute	Type	Required
subscriptionEvent	action	Action	Yes
	subscriptionId	string	Yes
	data	object	Yes

Object Name	Attribute	Type	Required
ts	string	Yes	

Object Name	Attribute	Type	Required
subscriptionErrorEvent	action	Action	Yes
	subscriptionId	string	Yes
	error	Error	Yes
	ts	string	Yes

Example:

Request:

```
{
  "action": "subscribe",
  "path": "Vehicle.Drivetrain.FuelSystem.Level",
  "filter": {"variant":"timebased", "parameter":{"period":"500"}},
  "requestId": "6578"
}
```

Successful response:

```
{
  "action": "subscribe",
  "subscriptionId": "12345",
  "requestId": "6578",
  "ts": "2020-04-15T13:37:00Z"
}
```

Error response:

```
{
  "action": "subscribe",
  "requestId": "6578",
  "error": {"number": 404, "reason": "unavailable_data", "message":
"The requested data was not found."},
  "ts": "2020-04-15T13:37:00Z"
}
```

Event:

```
{
  "action": "subscription",
  "subscriptionId": "12345",
  "data": {"path": "Vehicle.Drivetrain.FuelSystem.Level",
          "dp": {"value": "50", "ts": "2020-04-15T13:37:00Z"}
  },
  "ts": "2020-04-15T13:37:00Z"
}
```

Error event:

```
{
  "action": "subscription",
  "subscriptionId": "12345",
  "error": {"number": 401, "reason": "expired_token", "message": "Access token has expired."},
  "ts": "2020-04-15T13:37:00Z"
}
```

§ 5.1.2.3.1 Authorized Subscribe

If the operation on the VSS node that is addressed requires authorization, then the request must contain the field "authorization" with its value being a JWT token. The token validation must be successful for a subscribeSuccessResponse to be returned, else a subscribeErrorResponse is returned. An "authorization" key-value pair can be combined with all types of subscription requests.

Example:
Request:

```
{
  "action": "subscribe",
  "path": "Vehicle.Drivetrain.FuelSystem.Level",
  "filter": {"variant":"range", "parameter":[{"boundary-op":"gt",
"boundary":"49"}, {"boundary-op":"lt", "boundary":"51"}]},
  "authorization":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1...Zw_KSsds",
  "requestId": "6578"
}
```

Successful response:

```
{
  "action": "subscribe",
  "subscriptionId": "12345",
  "requestId": "6578",
  "ts": "2020-04-15T13:37:00Z"
}
```

Event:

```
{
  "action": "subscription",
  "subscriptionId": "12345",
  "data": {"path": "Vehicle.Drivetrain.FuelSystem.Level",
          "dp": {"value": "50", "ts": "2020-04-15T13:37:00Z"}
  },
  "ts": "2020-04-15T13:37:00Z"
}
```

§ 5.1.2.3.2 Curve Logging Subscribe

Curve logging data compression by eliminating data points that are within a set error margin is activated via a subscription request. Event messages will be issued when the buffer becomes full, after insignificant data points have been eliminated, refer the Curve logging Filter Operation chapter in the [viss3-core] documentation.

Example:
Request:

```
                {
                  "action": "subscribe",
                  "path": "Vehicle.Drivetrain.FuelSystem.Level",
                  "filter": {"variant":"curvelog", "parameter":{"maxerr":"0.5",
 "bufsize":"100"}},
                  "requestId": "6578"
                }
```

Successful response:

```
                {
                  "action": "subscribe",
                  "subscriptionId": "12345",
                  "requestId": "6578",
                  "ts": "2020-04-15T13:37:00Z"
                }
```

Event:

```
                {
                  "action": "subscription",
                  "subscriptionId": "12345",
                  "data":{"path": "Vehicle.Drivetrain.FuelSystem.Level",
                          "dp":[{"value": "50", "ts": "2020-04-15T13:38:00Z"}, ...,
 {"value": "25", "ts": "2020-04-15T13:39:30Z"}]
                  },
                  "ts": "2020-04-15T13:37:00Z"
                }
```

§ 5.1.2.3.3 Range Subscribe

Subscription to a range of values, that can have either a single boundary, or multipe boundaries as in the example below. For a more information how to use range of values, refer the Range Filter Operation chapter in the [viss3-core] documentation.

Example:
Request:

```
                {
                  "action": "subscribe",
                  "path": "Vehicle.Drivetrain.FuelSystem.Level",
                  "filter": "filter":{"variant":"range","parameter":[{"boundary-
 op":"lt","boundary":"50","combination-op":"OR"},{"boundary-
 op":"gt","boundary":"55"}]},
                  "requestId": "6578"
                }
```

Successful response:

```
                {
                  "action": "subscribe",
                  "subscriptionId": "12345",
                  "requestId": "6578",
                  "ts": "2020-04-15T13:37:00Z"
                }
```

Event:

```
                {
                  "action": "subscription",
                  "subscriptionId": "12345",
                  "data":{"path": "Vehicle.Drivetrain.FuelSystem.Level",
                          "dp":{"value": "51", "ts": "2020-04-15T14:00:00Z"}},
                  "ts": "2020-04-15T14:00:00Z"
                }
```

§ 5.1.2.3.4 Change Subscribe

Subscription to when a signal has changed between two sequential captures. For a more information how to use change of values, refer the Change Filter Operation chapter in the [viss3-core] documentation.

Example:
Request:

```
                {
                  "action": "subscribe",
                  "path": "Vehicle.Drivetrain.FuelSystem.Level",
                  "filter":{"variant":"change","parameter":{"logic-
op":"gt","diff":"10"}},
                  "requestId": "6578"
                }
```

Successful response:

```
                {
                  "action": "subscribe",
                  "subscriptionId": "12345",
                  "requestId": "6578",
                  "ts": "2020-04-15T13:37:00Z"
                }
```

Event:

```
                {
                  "action": "subscription",
                  "subscriptionId": "12345",
                  "data":{"path": "Vehicle.Drivetrain.FuelSystem.Level",
                          "dp":{"value": "101", "ts": "2020-04-15T14:00:00Z"}},
                  "ts": "2020-04-15T14:00:00Z"
                }
```

§ 5.1.2.4 Unsubscribe

To unsubscribe from a subscription, the client *SHALL* send an unsubscribeRequest message to the server. If the server is able to satisfy the request it returns an unsubscribeSuccessResponse message. If an error occurs, for example because an invalid subscriptionId is passed to the server, an unsubscribeErrorResponse message is returned.
If the client has created more than one WebSocket instance, it *MUST* always unsubscribe using the same WebSocket instance that was originally used to create the subscription.

Object Name	Attribute	Type	Required
unsubscribeRequest	action	Action	Yes
	subscriptionId	string	Yes
	requestId	string	Yes

Object Name	Attribute	Type	Required
unsubscribeSuccessResponse	action	Action	Yes
	subscriptionId	string	Yes
	requestId	string	Yes
	ts	string	Yes

Object Name	Attribute	Type	Required
unsubscribeErrorResponse	action	Action	Yes
	subscriptionId	string	Yes
	requestId	string	Yes
	error	Error	Yes
	ts	string	Yes

Example:
Request:

```
{
  "action": "unsubscribe",
  "subscriptionId": "12345",
  "requestId": "5786"
}
```

Successful response:

```
{
  "action": "unsubscribe",
  "requestId": "5786",
  "ts": "2020-04-15T13:37:00Z"
}
```

Error response:

```
{
  "action": "unsubscribe",
  "requestId": "6578",
  "error": {"number": 400, "reason": "invalid_data", "message":
"Data present in the request is invalid."},
  "ts": "2020-04-15T13:37:00Z"
}
```

§ 5.2 HTTPS

The message data components described in the [viss3-core] document are in the first hand mapped to required HTTP parameters, and only when there is no appropriate mapping it is mapped to the payload. The most significant deviations are:

- The path is part of the URL.
- A filter expression is appended to the URL as a query.

- The HTTP methods GET, POST replaces the use of "action".

The subscribe/unsubscribe messages are not supported by this transport protocol.

§ 5.2.1 Session Life Time Management

§ *5.2.1.1 Initialization*

Initialization involves setting up a secure HTTPS session between the client and the server. This ensures encrypted communication for data transmission. To initialize a secure session, the client sends a request to the server using the HTTPS protocol. This is achieved by connecting to the server's designated URL using the 'https://' scheme. The client can use a web browser, a native application, or a suitable library in the case of programmatically managed sessions.
While the client typically connects to the server using the specified hostname, which often includes the "www" prefix, it's important to note that this convention may not apply in situations where VISS operates within a local, in-vehicle network or if remote vehicle connections are allowed. The communication typically takes place over port 443, the default port for secure HTTPS connections. The hostname resolution can be done via DNS or configured through local settings.

§ *5.2.1.2 Closure*

Closure entails ending the established HTTPS session when the communication is complete or when the client no longer requires the connection. Either the client or the server can initiate the session closure. The client can signal the end of the session by sending an appropriate request to the server, indicating the intent to close the connection.
Upon session closure, any allocated resources, such as server-side threads or memory, are released, improving overall system efficiency.

§ 5.2.2 Transport Messages

§ *5.2.2.1 Read*

The client *MAY* send a HTTPS GET request message to the server to get one or more value(s) of one or more vehicle signal(s). If the server is able to satisfy the request it *SHALL* return a response containing the requested value(s). If the server is unable to fulfil the request, e.g. because the client is not authorized to retrieve one or more of the signals, then the server response *SHALL* have the status code set to indicate error.

Example: Request:

```
GET /Vehicle/Cabin/SeatPosCount   HTTP/1.1
Host: 127.0.0.1:1337
Accept: application/json
...
```

Successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
...
{
  "data":{"path":"Vehicle.Cabin.SeatPosCount",
          "dp":{"value":["2", "3", "2"],
"ts":"2020-04-15T13:37:00Z"}
         }
}
```

Error response:

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
...
{
  "error": {"number": 404, "reason": "unavailable_data", "message":
"The requested data was not found."},
  "ts": "2020-04-15T13:37:00Z"
```

```
                }
```

§ 5.2.2.1.1 Authorized Read

JWT tokens will be sent in the `Authorization` header, following with term `Bearer` and a space character.

The following example assumes `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSm9obiBEb2UifQ.xuEv8qrfXu424LZk8bVgr9MQJUIrp1rHcPyZw_KSsds` is the actual token. A token header can be combined with all types of read requests.

Example: Request:

```
            GET /Vehicle/Drivetrain/InternalCombustionEngine/RPM   HTTP/1.1
            Host:127.0.0.1:1337
            Authorization:Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSm9obiBEb2UifQ.xuEv8qrfXu424LZk8
bVgr9MQJUIrp1rHcPyZw_KSsds
```

Successful response:

```
            HTTP/1.1 200 OK
            Content-Type: application/json; charset=utf-8
            ...
            {
              "data":{"path":"Vehicle.Drivetrain.InternalCombustionEngine.RPM",
                      "dp":{"value":"2372", "ts":"2020-04-15T13:37:00Z"}
                     }
            }
```

Error response:

```
            HTTP/1.1 401 Unauthorized
            WWW-Authenticate: Bearer realm="127.0.0.1:1337",
                                     error="invalid_token",
                                     error_description="The access token is
invalid or expired"
            Content-Type: application/json; charset=utf-8
            ...
            {
              "error": {"number": 401, "reason": "invalid_token", "message":
"Access token is invalid."},
              "ts": "2020-04-15T13:37:00Z"
            }
```

§ 5.2.2.1.2 Search Read

The search read request uses the paths filter operation described in the [viss3-core] document to provide one or more path expressions, relative to the path in the GET URL.
Example: Request:

```
              GET /Vehicle/Cabin/Door?filter={"variant":"paths", "parameter":"*/
*/IsOpen"}   HTTP/1.1
              Host: 127.0.0.1:1337
              Accept: application/json
              ...
```

Response:

```
              HTTP/1.1 200 OK
              Content-Type: application/json; charset=utf-8
              ...
              {
                "data":[{"path":"Vehicle.Cabin.Door.Row1.Left.IsOpen", "dp":
{"value":"false", "ts":"2020-04-15T13:37:00Z"}},
                        {...},…
                        {"path":"Vehicle.Cabin.Door.Row4.Right.IsOpen", "dp":
{"value":"true", "ts":"2020-04-15T13:37:00Z"}}
```

```
                 ]
        }
```

Error response:

```
        HTTP/1.1 404 Not Found
        Content-Type: application/json; charset=utf-8
        ...
        {
                  "error": {"number": 404, "reason": "unavailable_data",
 "message": "The requested data was not found."},
                  "ts": "2020-04-15T13:37:00Z"
        }
```

§ 5.2.2.1.3 History Read

The history read request uses the history filter operation described in the [viss3-core] document to read recorded values for a given period backwards in time.
Example: Request:

```
        GET /Vehicle.Acceleration.Longitudinal?filter={"variant":"history",
 "parameter":"P2DT12H"}   HTTP/1.1
        Host: 127.0.0.1:1337
        Accept: application/json
        ...
```

Response:

```
        HTTP/1.1 200 OK
        Content-Type: application/json; charset=utf-8
        ...
        {
          "data":{"path":"Vehicle.Acceleration.Longitudinal", "dp":
```

```
[{"value":"0.123", "ts":"2020-04-15T13:00:00Z"}, ..., {"value":"0.125",
"ts":"2020-04-15T13:37:00Z"}]}
              }
```

§ 5.2.2.1.4 Signal Discovery Read

The signal discovery request uses the static metadata filtering type as described in the [viss3-core] document to retrieve metadata in the VSS tree. If the parameter object is set to an empty string, then all metadata in the addressed VSS nodes is returned. If only specific metadata is desired, the value can be set to this, e. g. "parameter":["type", "datatype"]. The values must be the names as defined in the VSS specification.

Example: Request:

```
              GET /Vehicle/Drivetrain/FuelSystem?filter={"variant":"static-
metadata", "parameter":""}   HTTP/1.1
              Host: 127.0.0.1:1337
              Accept: application/json
              ...
```

Response:

```
              HTTP/1.1 200 OK
              Content-Type: application/json; charset=utf-8
              ...
              {
                "metadata": {"FuelSystem":{"type":"branch","description":"Fuel
system data.","children":{"HybridType, ... }}},
                "ts": "2020-04-15T13:37:00Z"
              }
```

§ 5.2.2.1.5 Dynamic Metadata Read

The dynamic metadata, i. e. any other metadata kept by the vehicle system, is retrieved by the setting the "type" to "dynamic-metadata". The value *MUST* be set to one of the domain names as specified in the [viss3-core], Dynamic Metadata Filter Operation chapter.

Example: Request:

```
               GET /Vehicle?filter={"variant":"dynamic-metadata",
 "parameter":"server_capabilities"}   HTTP/1.1
               Host: 127.0.0.1:1337
               Accept: application/json
               ...
```

Response:

```
               HTTP/1.1 200 OK
               Content-Type: application/json; charset=utf-8
               ...
               {
                 "metadata": {"filter":["timebased", "change", "paths",
 "curvelog", "dynamic-metadata"], "access_ctrl": ["short_term",
 "signalset_claim"], "transport_protocol": "https", "wss"},
                 "ts": "2020-04-15T13:37:00Z"
               }
```

§ *5.2.2.2 Update*

The client may request that the server sets the value of one or more signals e.g. to lock one or more doors or open a window by sending an HTTPS POST request to the server. In the case of several signals being set, they *MUST* all be of the same data type, and be set to the same value. If the server is able to satisfy the request its response *SHALL* have a 200 OK status code set. If an error occurs e.g. because the client is not authorized to set the requested value, or the value is read-only, the server response *SHALL* have the status code set to indicate error.

Example:

```
POST /Vehicle/Drivetrain/Transmission/PerformanceMode   HTTP/1.1
Host: 127.0.0.1:1337
Accept: application/json
...
{
  "value": "sport"
}
```

Successful response:

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
...
{
  "ts": "2020-04-15T13:37:00Z"
}
```

Error response:

```
HTTP/1.1 404 Not Found
Content-Type: application/json; charset=utf-8
...
{
  "error": {"number": 404, "reason": "unavailable_data", "message": "The requested data was not found."},
  "ts": "2020-04-15T13:37:00Z"
}
```

§ 5.2.2.2.1 Authorized Update

JWT tokens will be sent in the `Authorization` header, following with term `Bearer` and a space character.

The following example assumes `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSm9obiBEb2UifQ.xuEv8qrfXu424LZk8bVgr9MQJUIrp1rHcPyZw_KSsds` is the actual token. A token header can be combined with all types of update requests.

```
            POST /Vehicle/Drivetrain/Transmission/PerformanceMode   HTTP/1.1
            Host:127.0.0.1:1337
            Authorization:Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJuYW1lIjoiSm9obiBEb2UifQ.xuEv8qrfXu424LZk8
bVgr9MQJUIrp1rHcPyZw_KSsds
            {
              "value": "sport"
            }
```

§ 5.3 MQTT

As the MQTT protocol is based on a pub-sub model, while VISS is based on a client-server model, it is necessary to apply a thin application level protocol on top of MQTT to abstract the client-server specific behavior. This is done by embedding the VISS messages in he MQTT messages, together with supplementary information that makes this abstraction possible, see the next chapter.
The VISS messages are adhering to the primary payload format without any deviations.

§ 5.3.1 Application Level Protocol

For MQTT to support the complete VISSv3 interface, as decribed in the interface chapter of the [viss3-core] specification, an application level protocol that runs on top of MQTT is added. It is described in the following, please also see the sequence diagram below. To emulate the client-server pattern that is described in the [viss3-core] specification, the vehicle server, via its vehicle client, issues a subscribe request to the broker on a topic named VID/Vehicle, where VID is an identity that uniquely links to the vehicle in the access control ecosystem. This vehicle identity is not necessarily the manufacturer's Vehicle Identification Number (VIN).
The client on the "cloud side" of the broker is expected to have access to this vehicle identity. How it

obtains it is out of scope for this specification. When the cloud client wants to issue a request to the vehicle server it first generates a unique topic name, which it subscribes to at the broker. It then generates a JSON formatted payload with the general structure
{"topic":"aUniqueTopic", "request":"VISSv3Request"}
where "aUniqueTopic" is the uniques topic name it just subscribed to, and "VISSv3Request" is the request for the vehicle server. This request *MUST* follow the payload format that is specified in the Websocket chapter of this specification. This JSON message is then issued to the broker, associated to the topic VID/Vehicle. This message will then be forwarded by the broker to the vehicle client, which forwards the string being the value of the "request" key in the message to the vehicle server. When the vehicle client receives the response to this request, it publishes it to the broker associated with the topic name that was the string value of the "topic" key name in the message it previously received from the broker.
The broker will then forward this message to the cloud side client that earlier subscribed to this topic name, which concludes the client-server based request-response as described in the [viss3-core] specification.
In the case of subscription requests the vehicle client needs to save the subscriptionId found in the subscribe response, together with the topic name associated to the subscribe request. When the vehicle server later issues event messages, the vehicle client must parse the subscriptionId from it, and retrieve the topic name associated to it. The vehicle client shall delete the saved topic name and subscriptionId when it receives an unsubscribe request in a message from the broker.
In following requests from the cloud side client, the unique topic name may be reused from the previous request-response cycle, or a new unique topic name may be generated. If a new topic name is generated, an unsubscribe should be issued on the old topic name. The vehicle client can continue to use the topic name it subscribes to.
The payload format of the response/event messages *SHALL* follow the payload format that is specified in the Websocket chapter of this specification. The access control model is applicable also over this transport alternative. The Access Token server should then implement its own version of the application level protocol described here, using the topic name "VID/ATS". The Access Grant Token server may also do the same, with the topic name "VID/AGTS", or if it is deployed in the cloud it may expose the HTTP interface that is defined in this specification.

VISSv3 over MQTT

Figure 1 Message flow of VISSv3 over MQTT

§ 5.3.2 Security Aspects

The MQTT architecture mandates a "broker" that acts as a middleman in between the client and server endpoints (the subscriber and the publisher in MQTT terminology). This broker has full access to the plaintext communication between the two endpoints as each of the endpoint's TLS channel terminates at the broker. This aspect should be considered when selecting to use the MQTT protocol.

§ 5.3.3 Transport Messages

As mentioned in the "Application Level Protocol" chapter, the "request" messages issued to the broker contains two JSON formatted key-value pairs, where the value of the "request" key is a string that contains the request the vehicle server will respond to. The format of this request *MUST* follow the payload format that is specified in the Websocket chapter of this specification.

§ 5.4 gRPC

The gRPC protocol uses protobuf for the serialization. The definition of protobuf messages shall be expressed in a .proto-file. The proto file defining the encoding of the VISS primary payload format is found in the [viss3-payload-encoding] document. A protobuf compiler is used to generate code from the .proto-file that can then be called by the code executing the actual encoding/decoding between the two payload formats. This code can be implemented in different languages, and is out-of-scope for VISS standardization. An example of it in Go language can be found here.

§ 6. Definitions

§ 6.1 Term Definitions

Attribute	Type	Description
action	Action	The type of action requested by the client or delivered by the server.
path	String	The path to a node in the VSS tree, as defined by the Vehicle Signal Specification (VSS).

requestId	String	Unique id value specified by the client. Returned by the server in the response and used by the client to link the request and response messages. The value *MAY* be an integer or a Universally Unique Identifier (UUID).
subscriptionId	String	Value returned by the server to uniquely identify each subscription.
authorization	string	A JWT formatted security token.
data	object	Contains a path and one or more data points.
dp	object	The data point contains a value and a timestamp.
ts	string	The Coordinated Universal Time (UTC) time stamp that represents the capture of the value.
value	string	The data value associated with the path.
filter	string	Provides a filtering mechanism to reduce the demands of a subscription on the server. Query format, see [viss3-core], Filter Request chapter.
metadata	object	Metadata describing the potentially available signal (sub)tree.
error	Error	Returns an error code, reason and message.

§ 6.2 Action Definitions

The Action enumeration is used to define the type of action requested by the client. All client messages *MUST* contain a JSON structure that has an action name/value pair and the value of the action property *MUST* be one of the values specified in the enumeration:

get
Enables the client to read one or more values.

set
Enables the client to update one value.

subscribe
Enables the client to request event messages containing a JSON data structure with values for one or more vehicle signal.

unsubscribe
Enables the client to request that it should no longer receive event messages based on that subscription.

subscription
Enables the server to send event messages to the client containing a JSON data structure with

values for one or more vehicle signals.

§ 6.3 Error Definitions

The error number *SHOULD* be a status code defined in [RFC2616], c. f. chapter "Status codes". The error reason *SHOULD* be short. two or three words connected by underscor. It *SHOULD* relate to the reason-phrase from [RFC2616] for the corresponding status code. The error message is meant to give a more precise description of the error.

Object Name	Attribute	Type	Required
	number	integer	Yes
Error	reason	string	Yes
	message	string	Yes

§ List of Figures

§ A. References

§ A.1 Normative references

[MQTT]
Message Queuing Telemetry Transport (MQTT). OASIS. March 2019. 5.0. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. IETF. March 1997. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc2119

[RFC2616]
Hypertext Transfer Protocol -- HTTP/1.1. R. Fielding; J. Gettys; J. Mogul; H. Frystyk; L. Masinter; P. Leach; T. Berners-Lee. IETF. June 1999. Draft Standard. URL: https://www.rfc-

editor.org/rfc/rfc2616

[RFC6455]
The WebSocket Protocol. I. Fette; A. Melnikov. IETF. December 2011. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6455

[RFC8174]
Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words. B. Leiba. IETF. May 2017. Best Current Practice. URL: https://www.rfc-editor.org/rfc/rfc8174

[RFC9112]
HTTP/1.1. R. Fielding, Ed.; M. Nottingham, Ed.; J. Reschke, Ed. IETF. June 2022. Internet Standard. URL: https://httpwg.org/specs/rfc9112.html

[viss3-core]
COVESA VISS version 3.0-Core. Ulf Bjorkengren. URL: https://raw.githack.com/COVESA/vehicle-information-service-specification/main/spec/VISSv3.0_Core.html

[viss3-payload-encoding]
COVESA VISS version 3.0-Payload Encoding. Ulf Bjorkengren. URL: https://raw.githack.com/COVESA/vehicle-information-service-specification/main/spec/VISSv3.0_PayloadEncoding.html

↑