SOVD Hands On – SOVD in Practice

Whitepaper | V1.0 2022-08

Table of Content

1 Introduction in SOVD

1.1 Service Oriented Vehicle Diagnostics

SOVD [1] stands for Service Oriented Vehicle Diagnostics and is standardized in version 1.0.0 in ASAM e.V. (Association for Standardization of Automation and Measuring Systems). The standard describes a standardized API (Application Programming Interface) for the diagnostics of vehicles with HPC and classic ECUs. The ECUs can be accessed locally at the vehicle ("Proximity"), in the vehicle ("In-Vehicle") or via the cloud ("Remote"). The use cases of diagnostics in the entire vehicle lifecycle are covered: from development over production to operation and after-sales.

SOVD is based on modern web technologies: The API is based on the HTTP protocol [4] and follows REST principles [2]. The data is transferred as a JSON-encoded [5] data structure.

1.2 Motivation for SOVD

1.2.1 What Advantages Does SOVD Offer Today?

If you want to diagnose a vehicle via UDS today, the technology stack typically looks like this: Communication with the ECU is handled by a VCI hardware (Vehicle Communication Interface), that is connected to a diagnostic server via a D-PDU API [9]. The diagnostic server is parameterized via diagnostic databases. The diagnostic server provides a programming interface (API) for the tester application. The following figure illustrates the respective components:

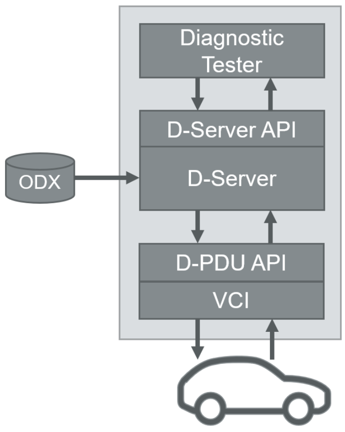

Figure 1: Setup of diagnostic applications today

Most components have been developed for use on desktop operating systems. This is also visible from the standardized programming interfaces, which have been defined as either C, C++ or Java API.

SOVD standardizes a REST API that is independent of the programming language used. This enables new, lightweight application scenarios for diagnostics. Based on SOVD, diagnostic workflows can be written in any modern programming or scripting language such as Python, for example, to read and evaluate data from a vehicle. On the other hand, the REST API enables the development of Enterprise (Cloud) applications for larger use cases such as a next generation workshop or development tester. These can be realized as web applications executable on smartphone, tablet, or laptop or even on the latest data goggles.

The following example shows the communication of a diagnostic application with a SOVD server via REST:

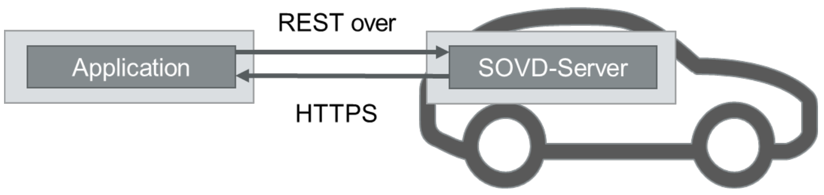

Figure 2: Setup of diagnostic applications using SOVD today

Web standards already offer a very high level of security through standardized and widely used protocols such as OpenID Connect [6] for authentication and OAuth2 [7] for controlling authorizations. It is possible to integrate existing identity providers such as Microsoft Azure AD to reuse identities that are already established in the Enterprise environment.

SOVD's unified interface allows diagnostic access and operations to be performed uniformly in proximity, remote, or in-vehicle scenarios.

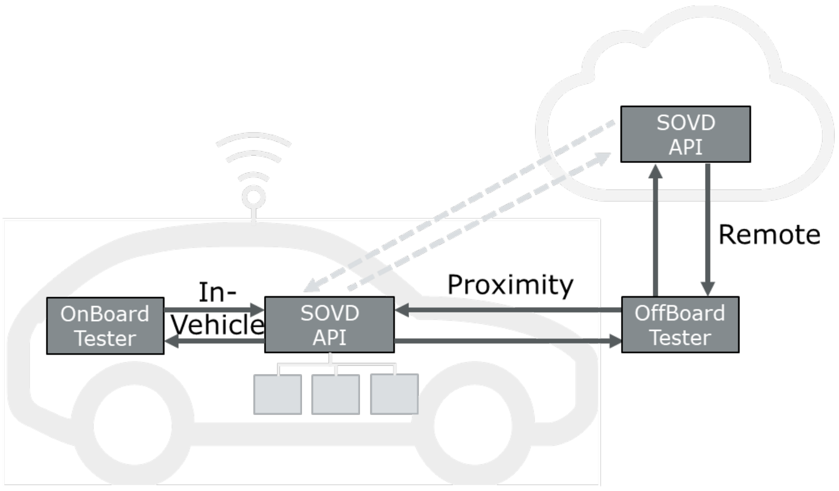

Figure 3: Application scenarios for the SOVD API: In-Vehicle, Remote, Proximity

In addition, the API is able to serve several clients simultaneously (multiclient capability). This paves the way for new application scenarios, e.g., diagnostic access to a vehicle in the workshop and simultaneous remote access from the manufacturer's Technical Assistance Center. This reduces costs and downtime in difficult cases.

1.2.2 What Advantages Does SOVD Offer in the Future?

SOVD was designed with a focus on the diagnostics of software on HPCs and integrates the existing diagnostics of ECUs based on UDS (Unified Diagnostic Services) [8]. Thus, the new standard combines the power and proven real-life experience of UDS diagnostic concepts and enables access with modern technologies. Although HPCs may not be widely used today, SOVD enables a smooth transition from UDS-based diagnostics to the world of diagnostics from software to HPCs.

When the interface uses SOVD on both, the vehicle and the backend, the technology stack for the tester is greatly simplified: no additional specialized hardware (VCI) is required to communicate with the vehicle anymore, as SOVD communicates over Ethernet (WLAN/LAN). SOVD integrates a self-describing approach through which the vehicle's diagnostic capabilities can be read out. ECU-based diagnostics with UDS require a suitable external diagnostic description, without it the tester cannot communicate with the vehicle. In addition, the memory- and computing-time-intensive diagnostic kernel in the diagnostic application is no longer needed to translate the data from the ECU into symbolic values (and back).

The tester of the future focusses on data presentation for the respective use cases - and thus the basis for future innovations.

2 Realization of Today's Diagnostic Use Cases with SOVD

The impact of the new SOVD standard is less on users of diagnostics in workshops, except that the diagnostic tester is possible on new platforms (e.g., web browser instead of desktop application).

Experts in diagnostics who investigate problems on the protocol level and developers of diagnostic testers are affected. For them, the changeover from UDS to SOVD changes significantly more.

This chapter uses a concrete example, the vehicle quick check, to describe what will change for diagnostics experts by highlighting the differences between UDS and SOVD.

2.1 Vehicle Quick Check

The Vehicle Quick Check provides a quick overview of the vehicle's condition and answers the questions:

> Which ECUs are installed in the vehicle?

> Which software versions are available in the ECUs?

> Have trouble codes been filed? If so, which ones?

2.1.1 ECU Overview

A diagnostic description is required for diagnostics of UDS-based ECUs. In addition to the diagnostic scope of the ECUs, this also describes topology information and the specific communication parameters that are required for a tester to be able to communicate with the vehicle.

Management of the variance of vehicles and associated diagnostic descriptions is highly manufacturer-dependent.

SOVD includes API methods for discovery of SOVD servers via well-known mechanisms such as multicast DNS or DNS service discovery, as well as the contained SOVD entities and their resources.

SOVD supports multiple views of the systems:

> Logical view of areas (e.g. powertrain)

> Physical view of components (e.g., specific ECUs) and their apps (e.g., climate control)

> View of cross-vehicle functions (e.g. vehicle identification)

> View of additional SOVD servers (in the vehicle)

Cross-dependencies such as the use of components in individual areas (e.g. ADAS or Powertrain) can be resolved via the REST API.

The following example shows specifically how the overview of installed components (e.g. UDS ECUs) can look like:

Request:
```
HTTP GET {base_uri}/components
```
Response:
```
HTTP 200 OK
{
  "items": [
    {
      "id": "door",
      "name": "Door Unit",
      "href": "{base_uri}/components/door"
    },
    {
      "id": "engine",
      "name": "Engine Controller Unit",
      "href": "{base_uri}/components/engine"
    }
  ]
}
```

The entry with `{base_uri}/components` lists all components (e.g. ECUs) that are installed in the vehicle. In addition to the name of the component, the reference to this resource is also supplied, which can now be used for further actions such as reading out the identification data or fault memory.

2.1.2 Reading out the ECU Identification Data

The UDS standard provides the service 0x22 ReadDataByIdentifier for reading out the identification data. The Data Identifiers (DID) that can be read here are contained in the diagnostic description (e.g. ODX) for the tester. The parameters of the diagnostic services are also described there. Each parameter in turn has, among other things, a name and a conversion (data type) to calculate a hexadecimal value from a human-readable value, which is finally sent to the ECU. The received hexadecimal response from the ECU is then converted back to human-readable values.

Here is an example of reading the VIN via the DID `0xF190`:

UDS Request to Target ECU:
```
[0x] 22 F1 90
```
UDS Response from ECU:
```
[0x] 62 F1 90 56 33 43 54 30 52 56 33 48 31 43 4C 33 31 32 33 34
```

The following figure shows the processing of requests and responses in a current diagnostic system:

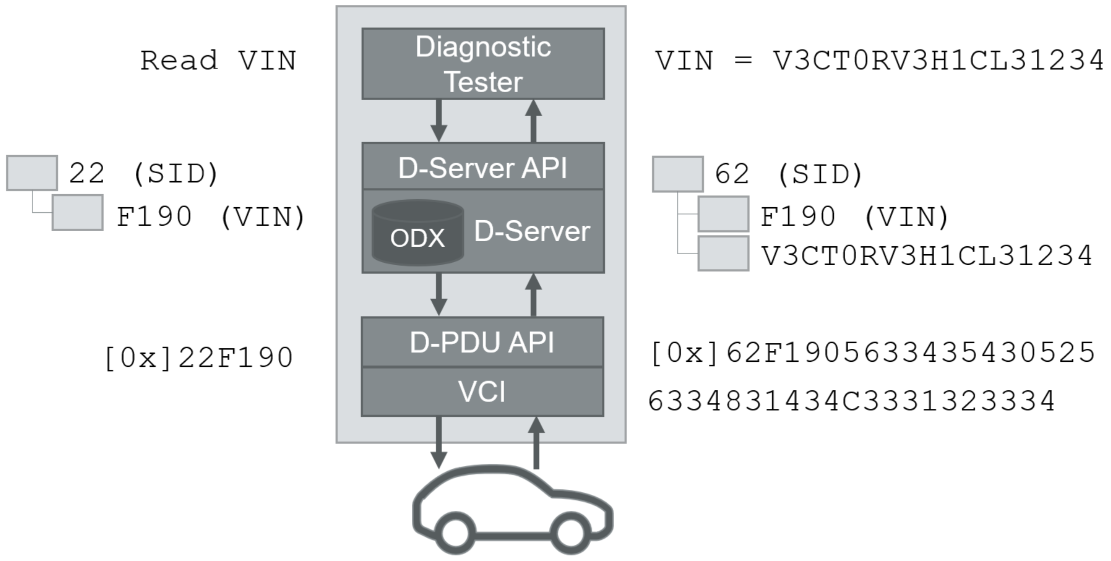

Figure 4: Comparison of data readout via UDS and SOVD

SOVD works fundamentally differently here. All values are already available in symbolic format for the diagnostic application, so that it does not have to perform any time-consuming conversion of the raw data from the ECU.

Each component provides a self-description of its identification data on request. The `/data/` resource offers all parameters that the respective component supports. In SOVD, a so-called query string can be used to filter all data by identification data (identData):

`/data?categories=identData`.

The following example shows how all available identification data can be listed via SOVD:

Request:
```
HTTP GET {base_uri}/components/{component_id}/data?categories=identData
```
Request für das Motorsteuergerät (z.B. „Engine“ aus dem Discovery):
```
HTTP GET {base_uri}/components/engine/data?categories=identData
```
Response:
```
HTTP OK 200
{
  "items": [
    {
      "id":       "vin",
      "name":     "Vehicle Identification Number",
      "category": "identData"
    },
    {
      "id":       "swversion",
      "name":     "Software Version",
      "category": "identData"
    }
  ]
}
```
After all available identification data has been determined, the relevant data and their values can now be queried individually.

The following example shows the readout of the vehicle identification number (VIN) via the SOVD API:

Request for readout of the VIN from the engine ECU:
```
HTTP GET {base_uri}/components/engine/data/vin
```
Response:
```
HTTP OK 200
{
  "id":       "vin",
  "name":     "Vehicle Identification Number",
  "category": "identData",
  "data":     "V3CT0RV3H1CL3123"
}
```

If the selection of identification data is required repeatedly in the diagnostic session, a data group can be created for this purpose. A UUID [11] (Universally Unique IDentifier) is generated for this data group. In this way, many data can be queried at once with one request.

2.1.3 Read out of the Fault Memory

The UDS standard defines a whole range of services that can be used to read out the fault memory. In addition to reading out the number of trouble codes, in UDS it is also possible to query trouble codes with environmental data. This requires a sequence of diagnostic requests that first determines the trouble codes and then uses them as request parameters to query their environment data.

The following example shows the reading of the number of active errors from an ECU:

UDS Request:
```
[0x] 19 01 09
```
UDS Response from the ECU: 1 trouble code stored
```
[0x] 59 01 09 01
```

If the concrete stored trouble codes are needed, the request looks like this:

UDS Request: Reading out the active errors from an ECU
```
[0x] 19 02 09
```
UDS Response: 1 trouble code stored
```
[0x] 59 02 09 FF 12 34 01 08
```

The response from the ECU must finally be interpreted again by the diagnostic kernel and converted into a readable form.

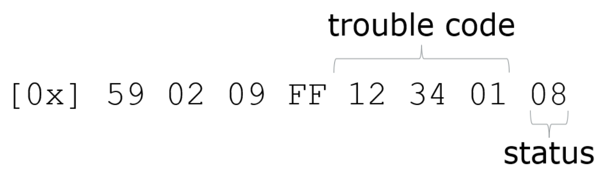

Figure 5: Diagnostic response from the ECU at error request

From the diagnostic description, the diagnostic kernel extracts the concrete fault text and additional fault information such as set and reset conditions. Thus, a `0x123401` becomes an understandable text „`O2 Sensor - Circuit Open`".

SOVD also standardizes access to errors in components and apps via the `/faults/` resource. In comparison to UDS diagnostics, no separate external data description is required here either. In SOVD, no special sequence is required to determine the environment data either. Setting a request option is sufficient.

The query of the same errors as from the previous UDS example looks as follows in SOVD

Request for reading out the active errors:
```
HTTP GET {base_uri}/components/engine/faults?status[aggregatedStatus]=active
```
Response:
```
HTTP OK 200
{
  "items": [
    {
      "code": "123401",
      "scope": "Default",
      "display_code": "P123401",
      "fault_name": "O2 Sensor - Circuit Open",
      "fault_translation_id": "ENGINE_123401_tid",
      "severity": 1,
      "status":
        {
          "mask": "08",
          "aggregatedStatus": "active"
        }
    }
  ]}
```

2.2 Stateful UDS Diagnostics and SOVD

One feature of SOVD is its multi-client capability. This leads to new challenges. Some operations in UDS require exclusive access to certain resources such as actuators, which can be controlled via the UDS service 0x2F (InputOutputControlByIdentifier). This is solved in the standard by locking resources. But how does the tester know that a resource needs exclusive access?

The SOVD Capability Description (OpenAPI [3] Spec) contains an additional attribute to indicate resources that require exclusive access: `x-sovd-lock-required`. A concrete example:

```
paths:
  /components/light/operations/HighBeamControl:
    get:
       operationId: "HighBeamControl"
       x-sovd-lock-required: true
```

If the attribute „`x-sovd-lock-required`“ returns the value „`true`” ", exclusive access is required.

The SOVD client assumes a lock on a resource by taking ownership of the lock via its authorization token. If another SOVD client subsequently tries to access a locked resource, the HTTP error code 409 (Conflicted) is returned.

The following is an example of how to create a lock for a resource:

Request:
```
HTTP POST {base_uri}/components/light/locks
{
  "lock_expiration": 3600
}
```
Response:
```
HTTP Created 201
Location: /components/light/locks/460AB8A5-5971-4693-8626-6287960050AF
{
 "id": "460AB8A5-5971-4693-8626-6287960050AF"
}
```

As you can see, the entire resource is now locked by the active SOVD client. This means that actuators can now be controlled exclusively and resource conflicts can be avoided. This also applies to classic UDS entities, which may also require session handling and security access. Other SOVD clients can therefore also not set this resource to a different state in parallel and thus cause problems.

Each lock has a certain expiration time, which must be extended by the SOVD client. When the time expires, the temporary lock is released and other SOVD clients can get access. If the lock is no longer needed, the SOVD client can explicitly release it.

2.3 Diagnostic Sequences

There are various use cases in which sequences of diagnostic services are used - whether for commissioning or troubleshooting components throughout the lifecycle of the ECU or vehicle. With the widespread adoption of web standards such as REST, there are generators and libraries for almost every programming language imaginable that provide very simple access to REST APIs.

SOVD defines an access to online capability descriptions of the REST API via the `/docs/` resource. It can be used to read out the diagnostic capabilities of the vehicle for developing sequences:

Request:
```
HTTP GET {base_uri}/components/docs?include-schema=true
```
Response:
```
HTTP 200 OK
Content-Type: application/json
[OpenAPI Spec]
```

In addition, an offline capability description can also be provided as an OpenAPI Spec. Code generators can generate type-safe libraries from the OpenAPI Spec. These are useful in the development of diagnostic sequences, as they contain type information that can be used for IntelliSense or compile-time checks, for example:

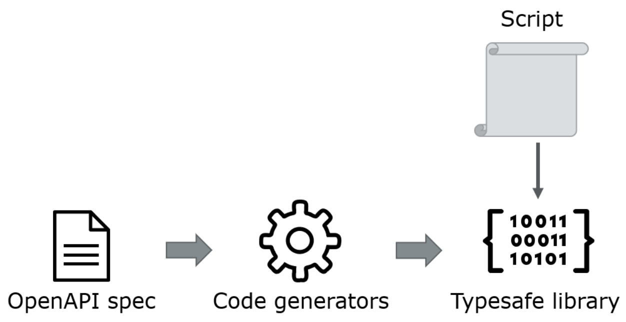

Figure 6: Code generators for programming libraries based on OpenAPI

In addition, almost every programming language provides libraries to interact with REST APIs - even without an OpenAPI specification.

Thus, diagnostic sequences can be developed with common development environments such as Visual Studio (Code). This increases development productivity enormously.

3 Impact of SOVD on Different Target Groups

The introduction of a new technology or a new standard at a central location is not a foregone conclusion. Some framework conditions critical to success in the introduction of SOVD are examined in more detail below.

Depending on the target group of the diagnosis, the impact of the changes brought about by SOVD varies.

3.1 Users of Diagnostics

For the user of diagnostics, SOVD will not change much. He uses a (SOVD-capable) diagnostic tool to read out data from the vehicle, such as fault memory and identification data, or to update the software. Diagnostic expertise is still needed to evaluate the diagnoses and derive actions from them (e.g., troubleshooting or repairing the vehicle).

Only with the spread of HPCs and the diagnostics of software (e.g. apps) with SOVD will there be changes for the user of the diagnostics (e.g. in the workshop). Then, in addition to an understanding of mechatronic systems, there will also be a need for competencies in the analysis of software problems.

3.2 Function Developer

Developers of apps or functions that run on HPCs can use SOVD to access a wider range of functions for troubleshooting. In addition to trouble codes, this also includes extensive log information, which is already known from PC-based application development, in order to analyze faults. To ensure that log information is available in the event of an error, this must already be taken into account during the development of the respective software. In addition, the traditional functional scopes of ECU diagnostics are still available, such as the reading of identification data or the control of functions.

Thanks to the self-describing diagnostic interface in SOVD, developers of diagnostics do not have to wait until their adaptations are made available in diagnostic descriptions for testers. They can access the diagnostic interface directly with a SOVD-enabled tool and run diagnostics like a production or workshop diagnostic user or write tests against it. This increases productivity and both development speed and quality.

For the developer of classic UDS-based ECUs, almost nothing changes. The SOVD server (e.g., HPC) implements the technology conversion from SOVD to UDS and back. Thus SOVD has almost no impact on him.

3.3 Developers of Diagnostic Testers

For developers of diagnostic testers, SOVD has larger impacts. The technology stack of UDS-based diagnostic testers mostly consists of C-/C++ or Java libraries. In some implementations, the interface conforms to the ISO 22900-2 and ISO22900-3 standard ("Road vehicles - Modular vehicle communication interface (MVCI)") [9]. This describes APIs for byte-current-oriented communication with the vehicle and the conversion of these byte currents into human-readable values such as fault codes. Diagnostic descriptions, e.g., in ODX format [10], are used to parameterize these libraries. These memory- and computationally-intensive libraries and the associated expert knowledge for developing diagnostic tools are no longer required when SOVD is used.

The new SOVD standard simplifies access to diagnostic data. The required technology stack is significantly simplified. As a result, new platforms for diagnostic applications can now be addressed that could not be supported previously due to resource-hungry diagnostic libraries.

The SOVD API is self-describing and returns all data as symbolic values. This allows the developer of diagnostic applications to focus on the presentation of the data and to implement new, innovative applications for different platforms (e.g., web applications).

The changeover to SOVD does not have to be a "big bang“ but can also be done step by step. There are various implementation scenarios to make diagnostic applications fit for the future ("SOVD-Ready"), which are listed here as examples:

- Extending the current diagnostic tester to include SOVD functionality.
- Replacing the underlying technology stack in the diagnostic tester to diagnose UDS-capable ECUs via SOVD
- Development of a new diagnostic tester that can diagnose today's vehicles via SOVD.
- Development of a second diagnostic tester that can only diagnose new vehicles via SOVD.

The following figure shows the spectrum of possible introduction scenarios for diagnostic testers.

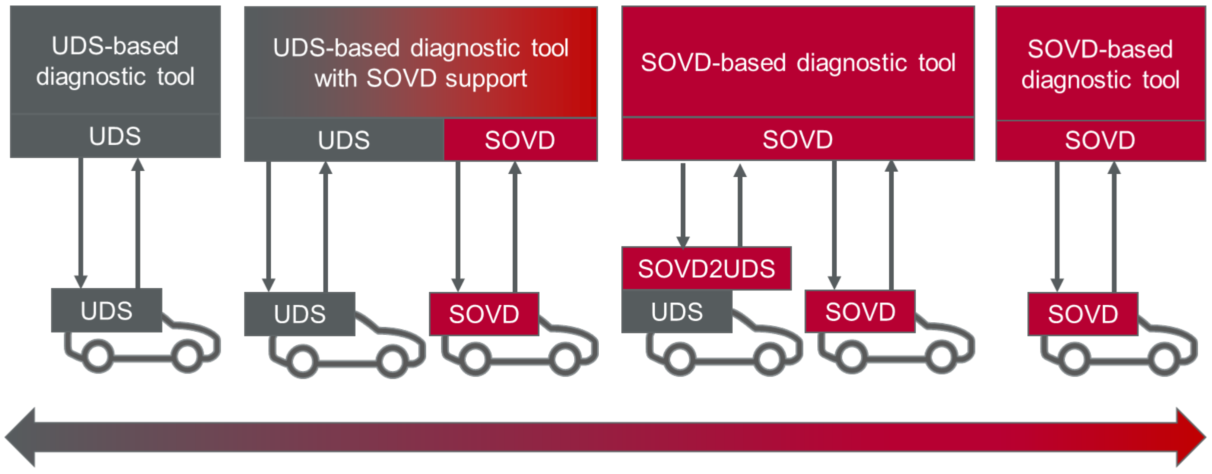
UDS-based diagnostic tool
UDS
UDS
UDS-based diagnostic tool with SOVD support
UDS
SOVD
UDS
SOVD
SOVD-based diagnostic tool
SOVD
SOVD2UDS
UDS
SOVD
SOVD-based diagnostic tool
SOVD
SOVD

Figure 7: Implementation scenarios for SOVD diagnostic testers

For first steps with the new SOVD API, no SOVD ECUs or HPCs with a SOVD API are needed. Instead, simulators that can be generated based on an OpenAPI specification can help. OpenAPI generators [12] are already widely used in web applications and so both clients and servers can be generated for almost any programming language and/or any common development environment.

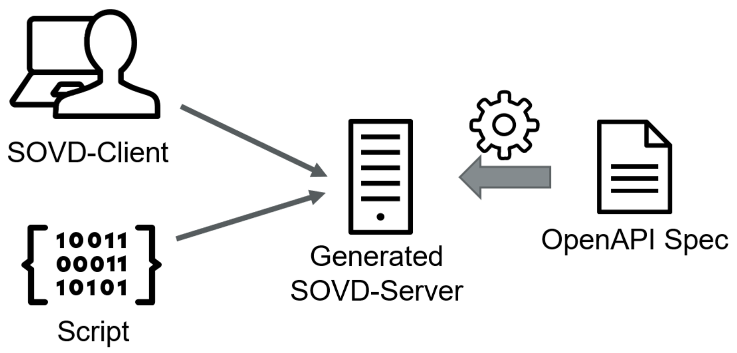

Figure 8: First steps with SOVD: Generated SOVD server

REST APIs can be used in all major programming languages either natively or with the help of libraries. This allows exploratory sample code to be written to gain more experience with SOVD.

In addition to using simulated SOVD servers, connecting existing ECUs and vehicles can provide interesting insights. SOVD2UDS adapters provide the bridge technology to address today's UDS-based vehicles via a SOVD API.

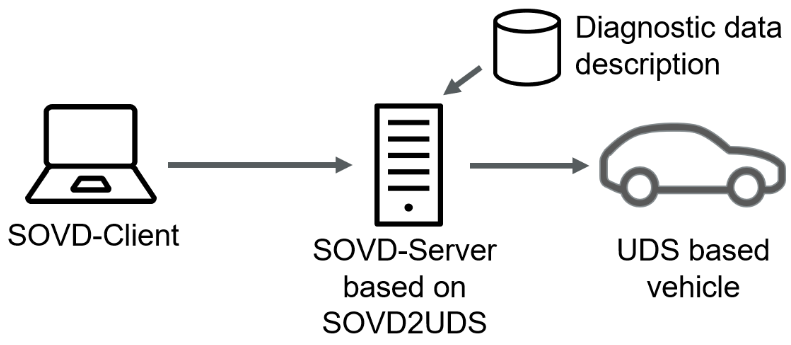

Figure 9: SOVD2UDS: Making UDS-based vehicles in SOVD available

Based on this experience, a migration scenario for diagnostic testers can be designed. In addition to today's desktop-based diagnostic testers, new diagnostic systems can also be implemented on the basis of the web technologies used in SOVD, e.g., as (browser-based) web applications or apps for smartphones and tablets. This opens up new areas of application.

4 Summary

The future of diagnostics has begun! Even if HPC diagnostics is not very common today, SOVD represents an important milestone for the diagnostics of software-based systems.

The uniform interface for in-vehicle, proximity and remote scenarios opens new use cases for the entire vehicle lifecycle. Simultaneous access to the vehicle by technicians and developers, for example, can also help achieve even better service.

Using the Vehicle Quick Check as an example, it was shown that access to diagnostics has been simplified. The self-describing SOVD interface simplifies the handling of variants and eliminates the need to distribute diagnostic descriptions. All diagnostic data queried at the interface can already be retrieved in a readable format. In addition, identification data can be filtered, or many data can be queried at once via data lists. Reading out the fault memory has been simplified and no longer requires special knowledge. The significantly larger amount of data due to the transfer of interpreted data when reading out data with SOVD has little effect on the time required to query the data.

Since SOVD relies on widely used web technologies such as REST, HTTP and JSON, there is a large set of existing tools to deal with these technologies. These include code generators to generate libraries for any major programming language from the OpenAPI specification. This can greatly simplify the development of diagnostic scripts and applications. Code generators can also be useful for testing, as server stubs can be generated, for example.

The introduction of SOVD has different effects on different target groups. For users of diagnostics, there will be little change in the first step - perhaps except for a new diagnostic tester. Diagnostic developers can use the new capabilities of SOVD in software-based systems to provide more information for troubleshooting. For diagnostic experts developing diagnostic testers, SOVD has a greater impact. Small steps can be taken to approach SOVD, REST & Co. and gain initial experience, even if HPCs with SOVD servers are not yet available.

After the content transformation to SOVD and the new technologies, the new application possibilities such as local and remote diagnostics can be evaluated to make the best use of the added value of SOVD.

Ultimately, the experience and knowledge gained from the use of SOVD with the diagnostics of UDS-based ECUs can also be transferred to the diagnostics of purely software-based systems.

Literatur

(1) SOVD – Service Oriented Vehicle Diagnostics (asam.net), Project number P2019-07

(2) R.T. Fielding; “REST: Architectural Styles and the Design of Network-based Software Architectures,” PhD dissertation, University of California, Irvine, 2000

(3) Open API Initiative; OpenAPI Specification v3.1.0; 2021

(4) RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1 (ietf.org); 1999

(5) ECMA-404 – Ecma International (ecma-international.org), 2nd edition, December 2017, https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf

(6) OpenID Connect | OpenID, https://openid.net/connect/

(7) OAuth 2.0, https://oauth.net/2/

(8) ISO 14229-1:2020 Roads vechicles – Unified diagnostics services (UDS) – Part1: Application layer

(9) ISO 22900-2:2009 Road vehicles – Modular vehicle communication interface (MVCI) – Part2: Diagnostic protocol data unit application programming interface (D-PDU API)

(10) ISO 22901-1: 2008 Road vehicles – Open diagnostic data exchange (ODX) – Part1: Data model specification

(11) RFC 4122 – A Universally Unique Identifier (UUID) URN Namespace (ietf.org), https://datatracker.ietf.org/doc/html/rfc4122

(12) API Code & Client Generator | Swagger Codegen, https://swagger.io/tools/swagger-codegen/

Get More Information

Visit our website for:
> News
> Products
> Demo software
> Support
> Training classes
> Addresses

www.vector.com