Vehicle Data architecture for Android and CCS

Approaches to end-to-end solutions of vehicle data
v1.1 (WORK-IN-PROGRESS)

Johan Strand, Mitsubishi Electric
Gunnar Andersson, Development Lead, GENIVI Alliance
and Members of AA-SIG

Discussion Material

INTRODUCTION

Foreword

- This is a work in progress and should not be seen as firm proposals. It is for now material for discussion and refinement in the ongoing project(s)
- The intention is to explore improved vehicle data solutions within the Android Automotive SIG. The slide deck may later also discuss complete Vehicle-Data oriented system architectures (Vehicle cloud architecture).
- We do not limit ideas and investigations to meet the *current* Compatibility Test Suite, but ultimately investigations aim to lead to solutions that are compatible, and/or may over time impact the standard Android code base.

Discussion - The reason for doing this

- What are the issue we are trying to solve?
- What are the gains for OEMs?
- Car data needs to be gathered from different nodes and supplied to Android though VHAL, does this need to be standardized?
- If a standard interface and data structure can be used in cars to make car data available then it opens up a world of opportunities for new features and functionality especially for third-party, but what would the key benefits be for OEMs?
    - Data server solution (e.g. VISS or other protocol) would create a standard interface for car data that can be used by OEMs or other parties depending on implementation by the OEM. Properly utilized this can simplify feature development by OEM and increase goodwill by offering customers a car with an secure interface where third-party features can be developed (platform independent).
    - The availability of data can be controlled by OEM and hence done in a secure way

Scope/Goals for our discussion

- Find end-to-end solutions, defining them to at least ~90% completeness
- Decide on approaches, reduce fragmentation, while showing where there still exists flexibility
- Scope: What End-to-end meaning here?
  - Find reasonable cut-off points for how far into details to go
  - E.g. Trace and include solutions for the "source" of data to CAN network or to other ECU, and work up to APIs for end user (Cloud app or on-board app). Leave out some details here (actual sensor measurement)
- Describe everything in between. Sync with all related GENIVI projects (GPRO, CCS, AA-SIG)
- Include **Android platform** as example environment

Background knowledge: Static vs Dynamic approach

- (from to VSS-to-Franca-(to SOME/IP) presentation made previously)

Dynamic “Thin API”

- One/few common access function for **all data**.
- For example a request is made by name or ID:
  getSignal(string signalname)
  - *(leaving aside Create/Update/Modify and so on, for now)*

- Characteristics:
  - Flexible – named data item can exist or not exist.
    Add data dynamically if needed, add new services as needed.
  - Reasonable to divide services up dynamically
    - e.g. some service provides a subset of the whole data tree
  - More WWW-like, Web protocols are often more like this.

  - No “compile-time” checking that signal/API actually exists
  - (Potentially) less efficient to have to send the signal name in, and on the service side “look it up”

Static API

- One/few access function per each data Item
- ~~`getSignal(“LaneDetected”)`~~ → one per item instead →
- → `get`**`LaneDetected`**`()`
  `get`**`OilTemperature`**`()`
  `get`**`VehicleSpeed`**`()`
  … and so on

  (similar for subcribeToXXX())

- Here in CommonAPI C++ style:

```
capi_proxy->getLaneDetectedEvent().subscribe(my_callback_function);
```

Characteristics:

Static and less extensible/flexible
Compile-time checking instead of run-time error
(Usually) less run-time lookup, i.e. efficient

Background knowledge:
Vehicle Signal Specification (VSS)

- You should know, what VSS looks like and what does it define?
- Name, hierarchy, signal data type…
- (→ refer to franca-vss slide deck)

Background knowledge: W3C automotive WG protocols

- You should know the basic content of VISS (v1) and planned content of “Gen2”

W3C VISS (v1)

- Web-oriented access protocol to expose VSS defined data to applications
- Specified by W3C Automotive Working Group around 2016->2018
- WebSocket interface
- Specifying a few functions: get, set, subscribe.
- Simple wildcards for selection of whole subtrees or similar:
  - **Vehicle.Chassis.Body.*.*.Window**
- JSON encoding of RPC methods and returned data

W3C ”Gen2” / VISS v2 / RESTful Service Interface*

- *Naming varies, a common internal working name is simply ”Gen 2”
- Specified by W3C Automotive Working Group, around 2018->2020(?)
- Primary focus is to expose VSS defined data
- Aiming for 2 primary methods *(occasionally discussing bindings to more protocols)*
- 1) RESTful HTTP requests
  - Primarily usable for on-demand (get/set)
  - Service discovery and service capability requests
  - Likely to use standard web-protocols for queries. Possibly GraphQL
- 2) Websocket approach (in principle equal or similar to VISSv1)
  - This provides the publish/subscribe capability
    (and get/set when desired)

Android Vehicle Properties

- Defined as static list of properties, with unique numerical IDs:

```
public static final int INVALID = 0;
public static final int INFO_VIN = 286261504;
public static final int INFO_MAKE = 286261505;
public static final int INFO_MODEL_YEAR = 289407235;
public static final int INFO_FUEL_CAPACITY = 291504388;
public static final int INFO_FUEL_TYPE = 289472773;
public static final int PERF_ODOMETER = 291504644;
public static final int PERF_VEHICLE_SPEED = 291504647;
public static final int ENGINE_OIL_LEVEL = 289407747;
```

etc…

With this comment: /**

/* Copy from android.hardware.automotive.vehicle-V2.0-java_gen_java/gen/android/hardware/automotive/vehicle/V2_0.
Need to update this file when vehicle propertyId is changed in VHAL
Use it as PropertyId in getProperty() and setProperty() in {@link android.car.hardware.property.**CarPropertyManager**}

The IDs change occasionally (reworked). Some IDs appear to follow a structure:

```
+    field public static final int ID_SEAT_HEADREST_FORE_AFT_MOVE = 356518810; //
0x15400b9a
+    field public static final int ID_SEAT_TILT_POS = 356518799; // 0x15400b8f
```

e.g. the hex value of all SEAT* start with 0x1540. Other groups have other common upper 2 bytes.

Android Vehicle Properties

- Named properties and IDs show up in several places and several services:
- 1) car-lib/api/**current.txt** and car-lib/api/**system-current.txt** which are generated(?) text files to document the API
- 2) Vehicle Properties and Sensors. Not the same, but related:
- src/android/car/**VehiclePropertyIds.java**:

```
     public static final int ENGINE_OIL_LEVEL = 289407747;
```
// (note, this is 0x11400303)

- src/android/car/hardware/**CarSensorManager.java**:

```
     public static final int SENSOR_TYPE_ENGINE_OIL_LEVEL = 0x11400303;
```

- 3) There are particular services for particular car subsystems. E.g. HVAC

car-lib/src/android/car/hardware/hvac/**CarHvacManager.java**

- with its own properties:

```
    public static final int ID_ZONED_SEAT_TEMP = 0x1540050b;
    public static final int ID_ZONED_AC_ON = 0x15200505;
```

Android Vehicle Properties

- 4) Permissions are defined in other files
- packages/services/Car/service/src/com/android/car/hal/**PropertyHalServiceIds.java :**

```
/** Helper class to define which property IDs are used by PropertyHalService.
  This class binds the read and write permissions to the property ID.*/
     mProps.put(VehicleProperty.INFO_VIN, new Pair<>(
          Car.PERMISSION_IDENTIFICATION,
          Car.PERMISSION_IDENTIFICATION));
     mProps.put(VehicleProperty.INFO_MAKE, new Pair<>(
          Car.PERMISSION_CAR_INFO,
          Car.PERMISSION_CAR_INFO));
```

In other words, programming the data structure that controls the required permissions.
To access the vehicle property **INFO_VIN** it is required to have the **PERMISSION_IDENTIFICATION** permission.   For **INFO_MAKE → PERMISSION_CAR_INFO** is required.

<u>There is for all properties (quite naturally) a dependency between application service layer</u> (**VehiclePropertyIds.java**) and HAL layer (**PropertyHalServiceIds.java**)

Android Vehicle ~~Properties~~ → Services

- There are now *(an increasing number?)* of car-related services, in addition to VehiclePropertyService
- Some **also** provide specific “convenience” APIs. Probably to make it easier to program with the most important/common car characteristics, compared to accessing everything through the generic Car Property Manager.
- Example: src/android/car/hardware/**CarSensorEvent.java**:

```
public CarEngineOilLevelData getCarEngineOilLevelData(CarEngineOilLevelData data)
{
```

- Note we see here a *specific* function for *this* particular sensor measurement, as opposed to the dynamic/thin API like: get(ID_OF_PROPERTY)

Android Vehicle Properties

- Is there a too large amount of dependencies between parts?
  (that are not automatically code generated from a single source)

- Properties show up in several places and several services.
  Remember the comment from **VehiclePropertyIds.java**:
  Need to update this file when vehicle propertyId is changed in VHAL */

- car-lib/api/**current.txt** and car-lib/api/**system-current.txt.** Are those generated(?) text files to document the API?

- src/android/car/**VehiclePropertyIds.java**:
  `public static final int ENGINE_OIL_LEVEL = 289407747;` // (note, this is 0x11400303)

- src/android/car/hardware/**CarSensorManager.java**:
  `public static final int SENSOR_TYPE_ENGINE_OIL_LEVEL = 0x11400303;`
  (interestingly, same ID is being used...)

Android Vehicle Properties

- Characteristics of Android approach:

- Static/fixed list approach*
- **partly – it is possible to ask for any ID and to check for a property availability during runtime, but with a fixed definition of numerical IDs it appears unlikely to be extended at run-time*
- "Flat" list of vehicle signal properties. No hierarchy.
  example: #define **THIS_IS_THE_COMPLETE_NAME**
- Fixed numerical **ID** for each: final int SIGNAL_NAME = **0x0000234**
- **CarPropertyManager** supports getting properties with different types (bool, float, int and intArray) but it's not well documented which property uses which type(?)
- Extensible: Need to explicitly add new (proprietary) signals to interface (and recompile)
  Q: How do the IDLs play into this and how much is auto-generated?

Comparison (Android vs. VSS-like approach)

- “Flat” list of vehicle signal names seems to be limiting.
  OK for small API, not suited for hundreds or thousands of signals
- Compare: **VEHICLE_LEFT_WINDOW** to **Vehicle.Chassis.Body.Door.Left.Window**
- Conversions (from VSS to flat list) are of course possible but tend to be a bit clumsy:
  → **VEHICLE_CHASSIS_BODY_DOOR_LEFT_WINDOW**
- Lost hierarchy means (among other things) lost ability to filter on sub-trees:
  - e.g. Vehicle.Chassis.Body.Door**.*.**Window
  - → For convenient queries
  - → For security (fine-grained access control)**
- Data types? Are all Android vehicle properties integer only?
- ***Android platform encodes (fixed in source code) relationships between individual signals or groups of signals and predefined permission names.*

Idea 1: Conversion to Android vehicle properties
→ Extend Android signal list from a larger list (VSS)

- Idea: Convert signals from VSS to (extended) vehicle properties on Android
- *(Static/fixed list approach)*
- Generating extended “flat list” from a snapshot of a VSS data model is technically possible, but it loses features as previously shown (Comparison)
- **Name:** Conversion as described **Vehicle.Chassis.Body.Door.Left.Window**
- *becomes* → **VEHICLE_CHASSIS_BODY_DOOR_LEFT_WINDOW**
- **Mapping IDs?**   VSS has Universally Unique Identifiers (UUID). The latest change proposal can also be (re)calculated from the tree path at any time (i.e. both stable and forever unique)
- UUIDs are by definition 128 bits. Android final ints are (presumably?) supposed to be 64 bit. Mapping or generating new/other IDs is theoretically possible.
- Data Types: Uncertain how to manage?
  Seems weakly defined in Android case (what do the rest of you think?)

Idea 2:
Alternative Vehicle Data Server/service

- Could execute in parallel with standard Android services, as a *complement*

  → Should be possible to meet Compatibility Test Suite
- One or several software services in the system provides vehicle data to requesting clients
- Normally a data service implies a dynamic approach (refer to previous slide)
- Implement standard on-demand functions **get**, **set**, and **subscribe** (updates sent automatically when value changes) feature
- Choices: some technical standards:
  - **W3C VISS (version 1)**
  - **W3C “Gen 2” / REST protocol**
  - **SOME/IP data service(s)**

Car data architecture ideas

W3C VISS (v1)

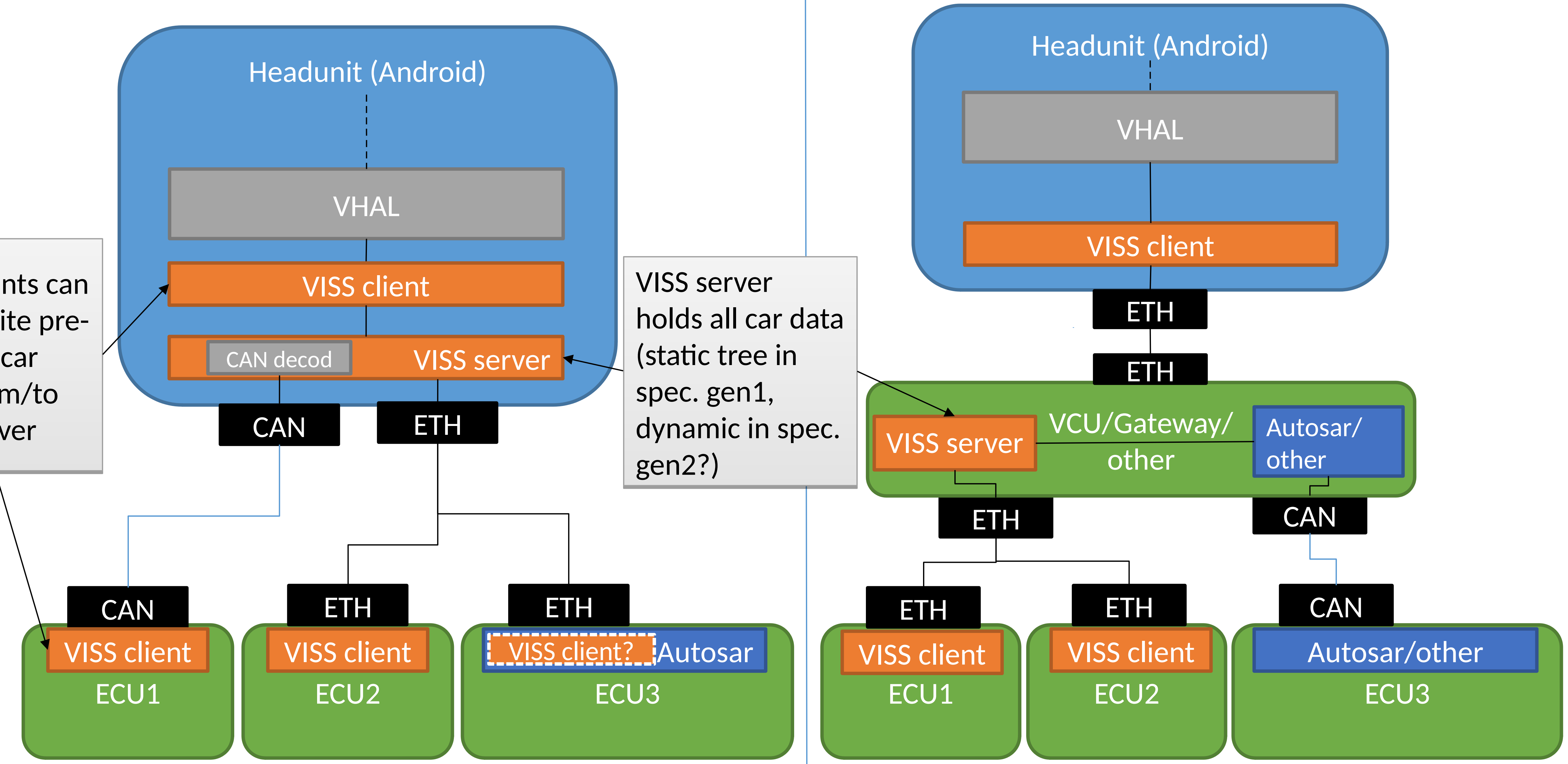
- One exploration idea: Ot would be possible to implement VISSv1 and that this protocol is a viable option

- (Although not shown explicitly in these slides, it should also be said that W3C “Gen2” might be a future option, through REST/HTTP APIs or socket-based that also include subscription)

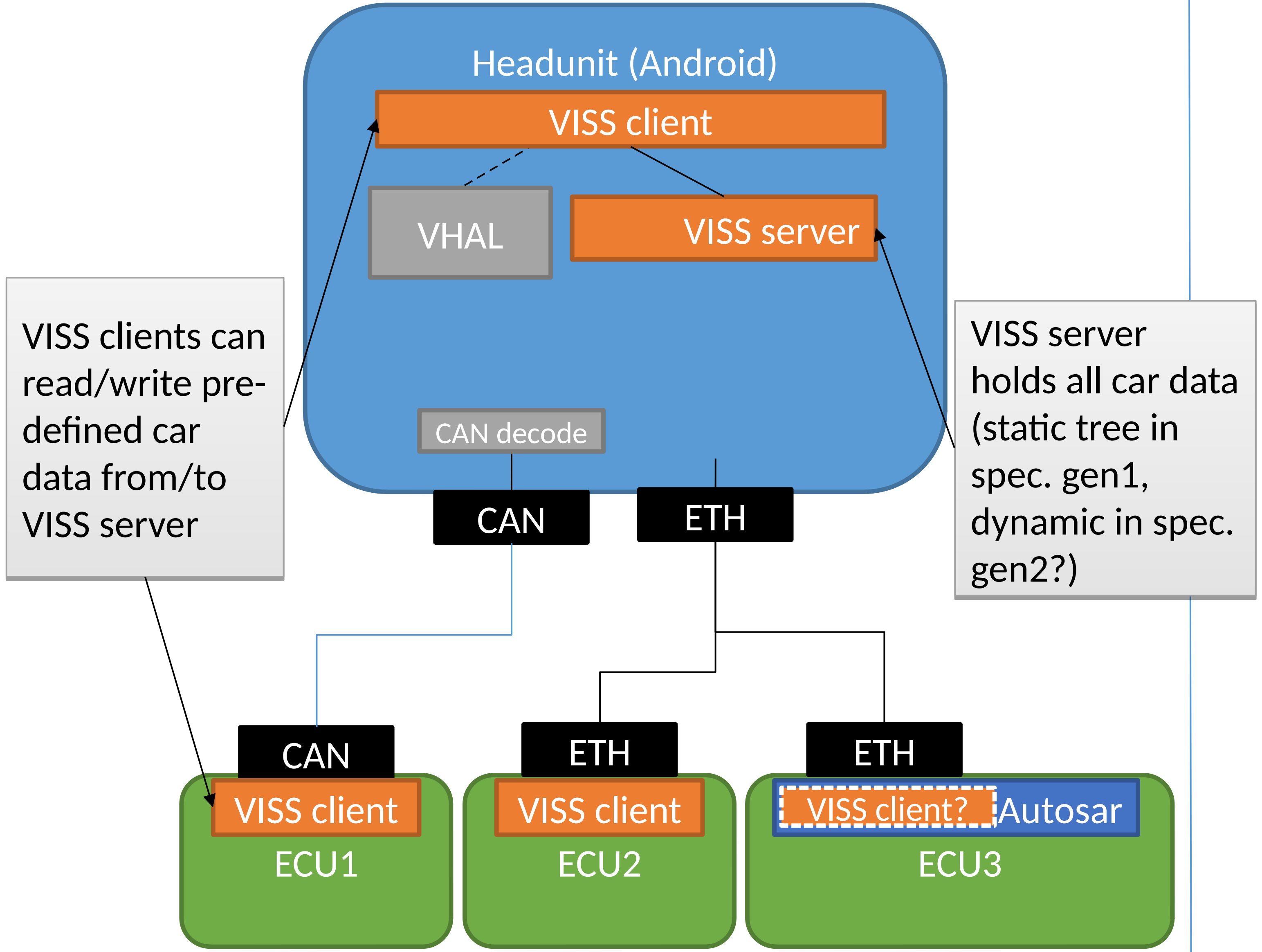
- For VISS (v1), an implementation to start with exists in GitHub (MELCO)

Architecture ideas

VISS websocket solution based on VISS server-client model and using VSS data, JSON webtoken used for access control



Replace or complement car API → **Vehicle Data Server/Service**


Headunit (Android)
VISS client
VHAL
VISS server
CAN decode
CAN
ETH
VISS clients can read/write pre-defined car data from/to VISS server
VISS server holds all car data (static tree in spec. gen1, dynamic in spec. gen2?)
CAN
ETH
ETH
VISS client
VISS client
VISS client?
Autosar
ECU1
ECU2
ECU3

SOME/IP recap

- SOME/IP provides primitives for interaction both using commands and observable data fields
  - **Remote Procedure Call** (invoke method)
  - **Properties** (described as Fields in SOME/IP specification)
  - Property **publish/subscribe** and **on-demand** (getXXX and setXXX)
- Although usage is of course up to the system designer, this principles seem to suggest multiple small services, each with a particular focus.

Alternative: Static “Full API” approach through code generation (1)

- Idea: A chosen (snapshot) of VSS specification can generate required (static) programming interfaces
- Each VSS signal can be an observable ”property”
(in Franca: “attribute”, in SOME/IP: “field”)
- Automatic translation and code generators create the programming interfaces and the necessary configuration of bindings.

Static “Full API” approach through code generation (2)

- Example: Implementation concept was shown in: **vss2franca_attributes** **(see earlier slide deck)**
  - Once VSS signals are converted to Franca Attributes, **Common API C++** code generation can be applied to the Franca Interface
  - This yields **C++ programming APIs** for each of the VSS signals to get, set, subscribe and get notified on value updates.
  - Common API C++ interfaces have backends already for **SOME/IP** (and D-Bus and WAMP).
  - In other words, all these building blocks *exist today*

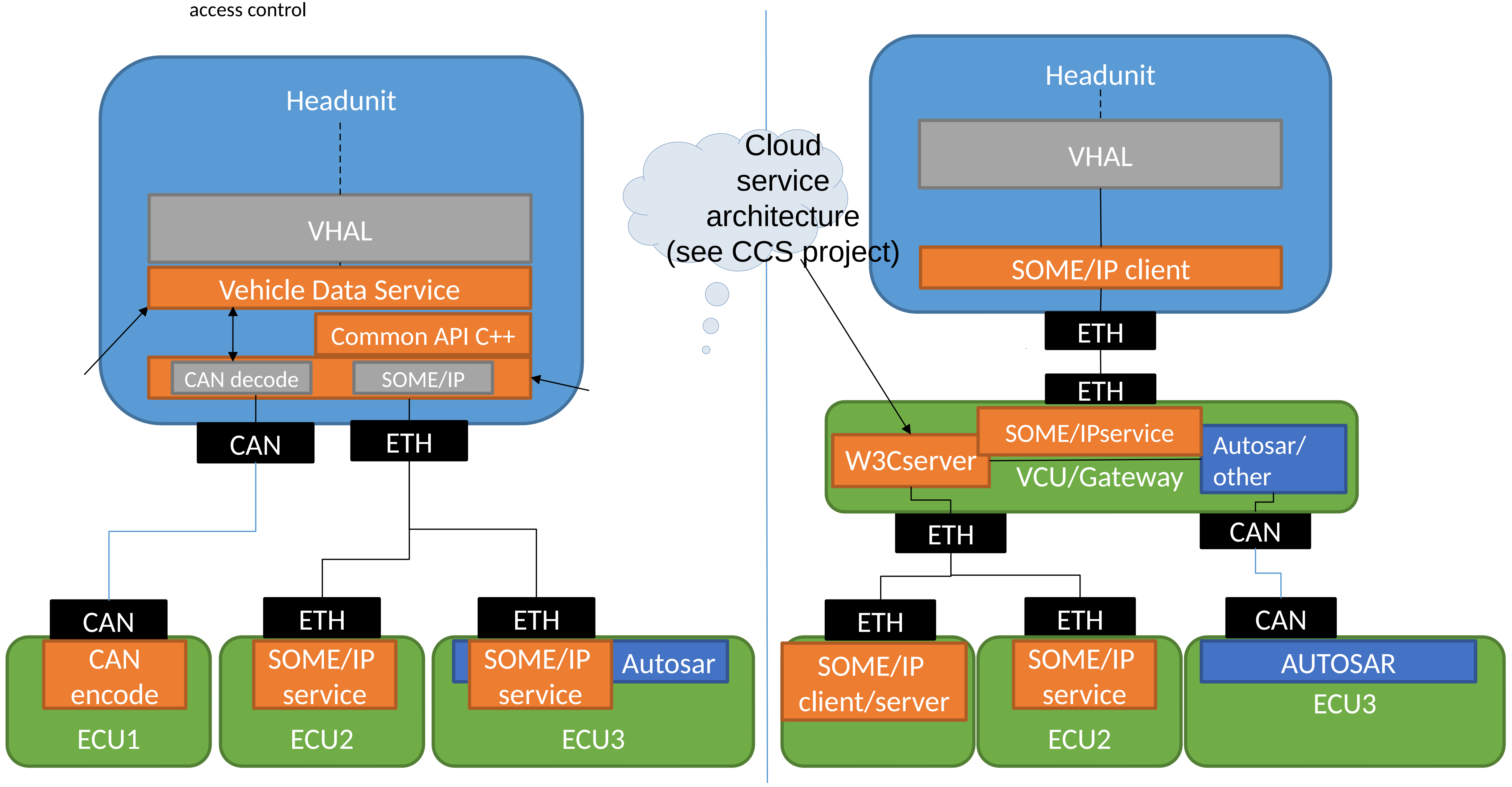
Static approach through code generation (3) AUTOSAR-based ECUs

- Example 1 / AUTOSAR side:
  - In an AUTOSAR context, the generated Franca IDL interface from Example 1 can be translated to AUTOSAR-XML using FARA Tooling
  - AUTOSAR tooling take over to generate the programming bindings

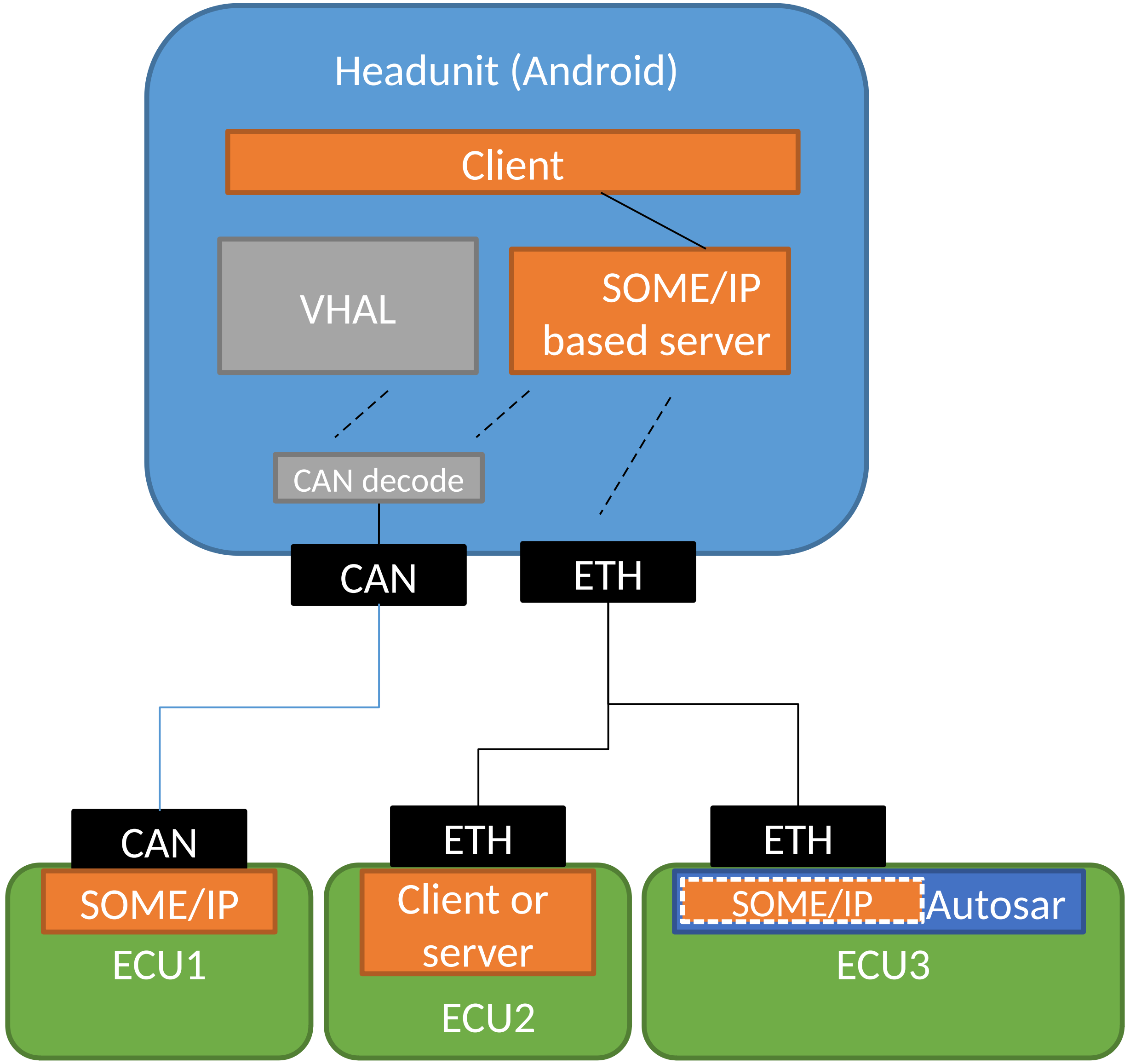
- Example 2:
  - More likely, to avoid the roundtrip via Franca to Franca/ARA translation tool, a direct translation from VSS to AUTOSAR XML will be desired in an AUTOSAR context
  - With compatible tooling, this could still allow for non-AUTOSAR clients using Franca + Common API C++, and a working interaction

SOME-IP Architecture ideas

VISS websocket solution based on VISS server-client model and using VSS data, JSON webtoken used for access control



The side-by-side VHAL + additional service solution seen before also applies if using SOME/IP of course



SOME/IP dynamic approach

- Create an interface using Franca IDL or AUTOSAR XML containing a few VISS-like functions:

  - **getSignal**(string signal_identifier_path)
  - **setSignal**(string signal_identifier_path, value)
  - **subscribe**…
- Convert this to a SOME/IP Vehicle Data Service
- Access control could be applied on service implementation side, including filtering on certain signals / sub-trees / etc.
- (This is in accordance with AUTOSAR identity and access management which prescribes services to implement the enforcement point)