RE-Vehicle Aggregator App

System Design Document

(Version 1.1.1)

| S.No | Prepared By | Review By | Version No |
| --- | --- | --- | --- |
| 1 | Ashwath Kumar K | Hareesha/Aparna | 1.0.0 |
| 2 | Lakshmi Patil (RE) |  | 1.0.1 |
| 3 | Ashwath Kumar K | Hareesha/Aparna | 1.1.1 |

**Table of Contents**

[***1.System Overview 2***](#_heading=h.gjdgxs)

[***2.1 High Level Design – Block Diagram 3***](#_heading=h.30j0zll)

[**2.2.1 Hardware Communication Layer (H/W Communication Layer) 4**](#_heading=h.1fob9te)

[**2.2.2 Communication Driver Layer 4**](#_heading=h.3znysh7)

[**2.2.3 Data I/O Interface 4**](#_heading=h.2et92p0)

[**2.2.4 Data Preserve Layer 4**](#_heading=h.tyjcwt)

[**2.2.5 Signal Conversion Layer 5**](#_heading=h.3dy6vkm)

[**2.2.5 Addition Service Layer 5**](#_heading=h.1t3h5sf)

[**2.2.5 Common Data Layer 5**](#_heading=h.4d34og8)

[***3. Classes and its Relation 5***](#_heading=h.2s8eyo1)

[***4. Storage Strategy 5***](#_heading=h.17dp8vu)

[**4.1 Available storage strategy 6**](#_heading=h.26in1rg)

[**4.1.1 Buffering 6**](#_heading=h.lnxbz9)

[**4.1.2 Streaming and Event-Driven Architectures 7**](#_heading=h.35nkun2)

[**4.1.3 Caching 7**](#_heading=h.1ksv4uv)

[**4.2.4 Load Balancing 7**](#_heading=h.44sinio)

[**4.2.5 Asynchronous Processing 7**](#_heading=h.2jxsxqh)

[***5 . Implementation 8***](#_heading=h.z337ya)

[***6 . Data Matrix 8***](#_heading=h.3j2qqm3)

[***7. Test Plan 8***](#_heading=h.1y810tw)

[***8.Bibliography 8***](#_heading=h.4i7ojhp)

# 1.System Overview

Vehicle data aggregator is a service in Android Open Source Project (AOSP) for getting the data from lower level, polishing it and forwarding or exposing interface to other services that require signal or CAN packets. In the automotive industry there is a different flavor of standard. Automotive software AOSP package is available, specifically for four-wheeler. But in case of two-wheelers there is no standard software packet available which contains all the basic necessities which are suitable for two wheelers. Some companies like Qualcomm are building a miniature version of software that behaves like automotive and mimicking it but is not the same. That is why all the companies involved in two-wheeler are generally taking bare AOSP from microcontroller/microprocessor providers that aid the WIFI, Bluetooth and other necessary hardware.

**Purpose**: Aggregate and manage vehicle-related data for applications and services.

**Components**: Vehicle Sensors, Data Aggregator Service, Database, Android Framework.

**Components**:

1. Vehicle Sensors: Accelerometer, GPS, OBD-II (On-Board Diagnostics), speed sensors, etc.

Android-compatible hardware or external devices.

1. Data Aggregator Service:

Data Collection: Collect data from various vehicle sensors. Utilize Android's Sensor APIs for sensor data.

Data Processing: Aggregate and normalize data from different sensors. Handle real-time and batch processing.

Data Storage: Store aggregated data in a dedicated database. Leverage Android's SQLite or consider other databases based on requirements.

1. Database:

Centralized storage for aggregated vehicle data.

Support for efficient querying and indexing.

1. Android Framework:

Integrate the Data Aggregator Service into the Android framework.

Leverage existing Android APIs for sensor integration and data storage.

User Applications:

Allow third-party apps to access aggregated vehicle data.

Develop standard APIs for app integration.

**Data Flow:**

Vehicle sensors generate data.

Data Aggregator Service collects, processes, and stores the data in the database.

Android framework provides APIs for apps to access the aggregated data.

**Security**:

Implement secure communication between sensors and the Data Aggregator Service.

Enforce user consent for accessing vehicle data in third-party applications.

**Permissions**:

Define appropriate permissions for accessing vehicle-related data.

Allow users to control which apps can access specific types of data.

**Integration:**

Integrate the Vehicle Data Aggregator service with Android's existing framework seamlessly.

Provide clear documentation for app developers on how to access and use the aggregated data.

**Testing:**

Implement thorough testing for different vehicle scenarios and sensor inputs.

Consider simulation for testing in non-vehicle environments.

**User Interface:**

Include a user interface for users to manage permissions and view basic vehicle data.

Allow customization for user preferences.

**Updates and Maintenance:**

Plan for regular updates to support new sensors or Android versions.

Provide over-the-air (OTA) update mechanisms.

**Compliance:**

Ensure compliance with legal and regulatory standards related to vehicle data.

The design may evolve based on Android's updates and the specific needs of the project. Always refer to the latest AOSP documentation and Android best practices for implementing such services.

# 2.1 High Level Design – Block Diagram

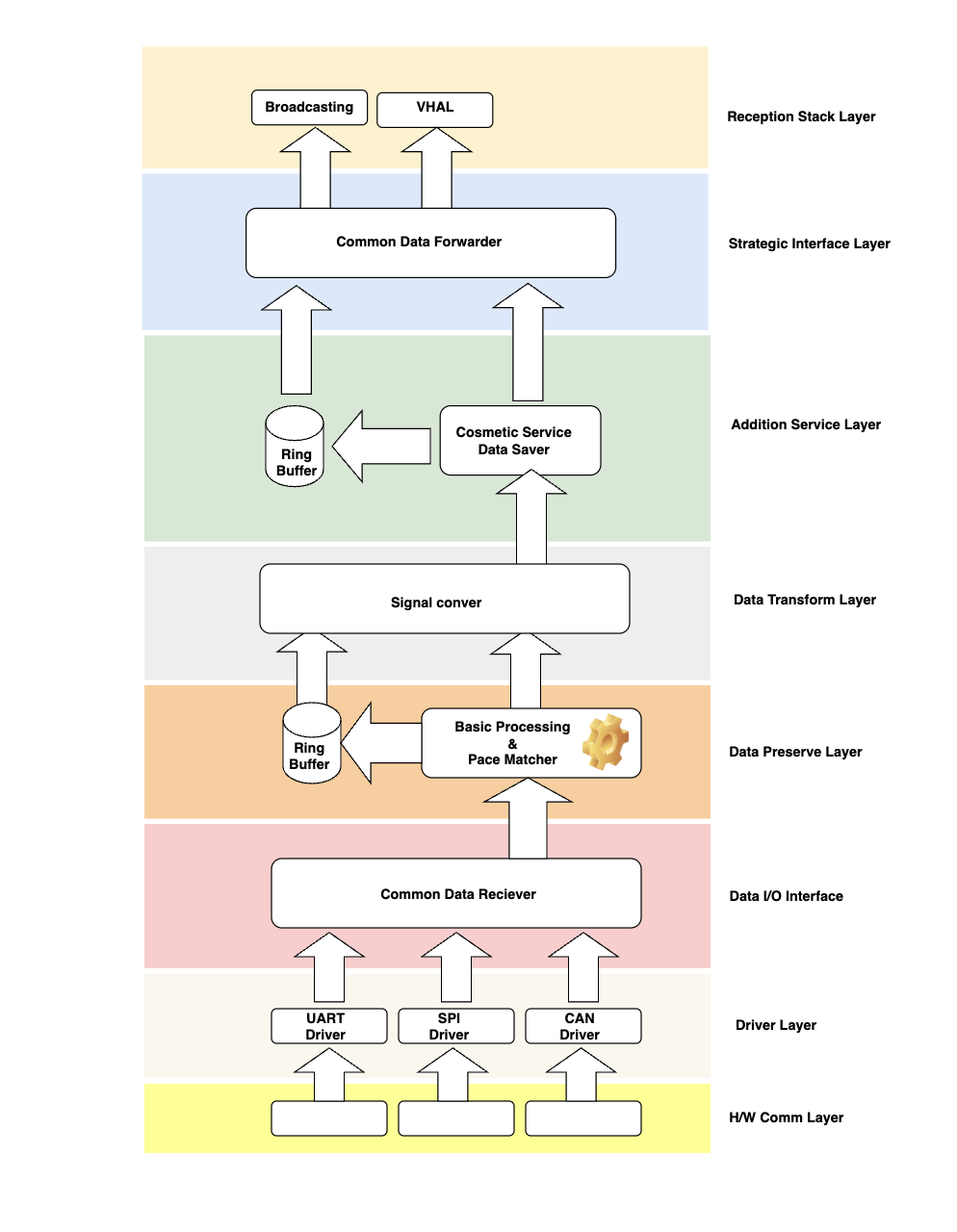
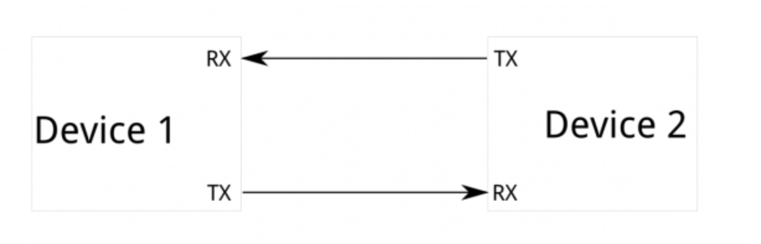


Figure 1 depicts the HLD of the Data Aggregator. Each layer has its own responsibilities and the layer structure of the data aggregator provides the facilities to change anything/adapt anything in the respective layer without touching or disturbing the other layers.

## 2.2.1 Hardware Communication Layer (H/W Communication Layer)

This layer is responsible for the hardware connection between the two devices BCM and SOM that want to communicate with each other. In this case it is UART.



UART

***Figure 2 Communication between Device1 and Device2***

## 2.2.2 Communication Driver Layer

Communication driver layer is responsible for picking the signal from the communication hardware layer, understanding it and making it in digital format and providing the interface to other layers to read it and write it. Generally, there is one to one mapping between the Communication driver layer and communication hardware layer.

## 2.2.3 Data I/O Interface

As we are building the vehicle data aggregator so for this particular service data source might be different, but it should be abstracted in such a way that for all the layers which are responsible for consuming the data pretend the source is single. The work of Data I/O interface is to get the data from different sources but provide the single interface for all above layers which are consumers.

CAN ID/Length/Data and periodicity of the frame to be considered as the data abstraction criteria. <Important: RE to decide the criteria>

Sloki: CAN ID as the arbitrator to be considered

## 2.2.4 Data Preserve Layer

Idea of data preserving layer is to buffer if data is too fast from the sender either from upper layer or lower so by introducing this layer, we will get the data losses issue from consumer issue due to pace difference.

## 2.2.5 Signal Conversion Layer

Objective of the Signal conversion layer is to dedicatedly do conversion/polishing/filtering etc of the signal. Making it separate leads to several benefits like strategically we can plug it and plug out the different conversion methodology/algorithm.

Raw CAN frames into Engineering signals

<RE to decide the criteria for storage of raw CAN frames>

Sloki: 10ms(fastest) , 50ms(moderate) and 500ms(slow) shall be considered for implementation

No separate algorithm shall be followed. We will consider the latest 15mins( configurable: subject to memory availability on the SOM module) of data.

## 2.2.6 Addition Service Layer

In addition to the service layer, we can add the different subservice like security, topic-based protection, signal wise buffering, handshaking, etc.

For this application development, authentication or security levels are not to be considered based on the data from RE.

<RE to confirm, A brief of authentication mechanism was suggested by Sloki in the earlier draft version of this document>

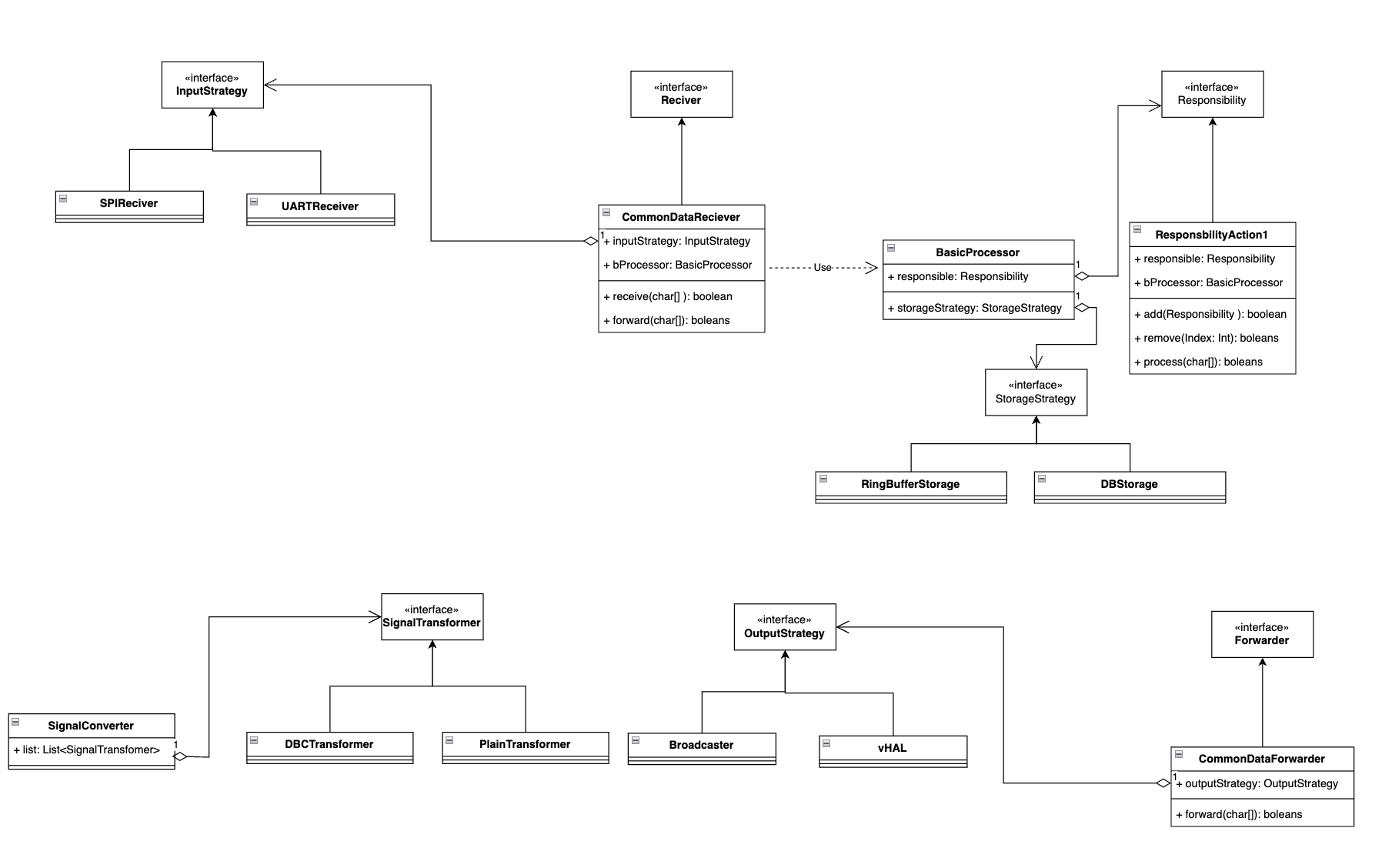
Sloki assumes no authentication provided

Signal wise buffering concept shall be used.

## 2.2.7 Common Data Layer

Common data provider layer acts as the source for the outer layer, It will expose the interface for different strategies for providing the data example, Intent based broadcasting, AIDL(Android Interface Definition Language) based interface etc.

# 3. Classes and its Relation

****

*Figure 3 Classes and their relations Block Diagram*

The above block diagram represents classes and relations among them.

# 4. Storage Strategy

A diagram of a computer program

Description automatically generated with medium confidence

In the data preserve layer and addition service provider layer, we have mentioned the buffer to hold the data in case there is a difference between speed of consumer and sender. Selection of buffer storage is totally dependent upon your requirement and design, to choose the storage type a few things we need to consider like - frequency of data, size of data and type of data. Generally following option are available for buffer storage.

1. RAM buffered space.
2. Database
3. Cached based buffer.
4. Cached based DB.

All the storage strategies that mentioned above are performed optimally based on the use cases. Let application wants driver/user-based document to forward to BMS than index based database is performed optimally similarly if upper application or lower application want sending the command or some critical signal to respective h/w then it has to process before trigger the hardware so we have to buffer to avoid the loss in this case we can choose cached based buffer or ram based buffer.

## 4.1 Available storage strategy

The choice of the best data storage strategy for managing the speed of data consumers and data senders depends on the specific requirements of application, especially in terms of data volume, data velocity, and latency constraints. Here are a few strategies to consider:

## 4.1.1 Buffering

We can use buffers to temporarily store data when there is a mismatch in speed between producers and consumers. This is particularly useful in scenarios where the data rate fluctuates or when consumers need time to process the data.

Buffer size and management play a crucial role in controlling latency. If the buffer becomes too large, it can introduce excessive latency; if it's too small, data may be lost.

Implementing circular buffers or bounded queues can help in managing buffering efficiently.

Since in vehicle aggregator Data going to be huge, we really need to think about buffering carefully.

## 4.1.2 Streaming and Event-Driven Architectures

In scenarios where low latency is a requirement, a streaming architecture might be more appropriate. Data is pushed to consumers as soon as it's produced. Event-driven architectures, using tools like Apache Kafka or RabbitMQ, can be useful for handling high-velocity data streams. Real-time data processing systems can benefit from these architectures to ensure timely delivery.

## 4.1.3 Caching

Caching can be used for both data producers and consumers. Producers can cache data to provide it more quickly to consumers, and consumers can cache data they've already processed for faster retrieval. Caches can help reduce the load on data stores and databases, speeding up access to frequently requested data.

## 4.2.4 Load Balancing

If you have multiple data producers and consumers, load balancing can help distribute the data processing load evenly. Load balancing strategies ensure that data is evenly distributed among consumers, preventing bottlenecks.

## 4.2.5 Asynchronous Processing

For consumers that don't require real-time data, asynchronous processing can be beneficial. Data is collected and processed in the background, allowing consumers to retrieve results at their own pace. This approach can be useful when data consumers are not synchronized with data producers. The choice of the best strategy should consider factors such as the nature of the data, the processing requirements, the volume of data, and the desired level of real-time processing. In many cases, a combination of strategies may be used to manage data efficiently and meet performance requirements. It's essential to assess the trade-offs between latency, throughput, and resource utilization when deciding on the best approach for your specific use case.

< Important RE has to select data storage we are going to use in this app>

Sloki shall use the buffering strategy mentioned below in green.

| Criteria to decide | Strategy | Merits | Demerits |
| --- | --- | --- | --- |
| Data Access Patterns | Buffering | Buffers can facilitate error handling and enhance fault tolerance  Helps create a smoother and more continuous flow of data | Scaling Latency  Complexity for large data |
| Horizontal vs Vertical Scaling | Streaming and Event-Driven Architectures | Scaling,  Low-latency data delivery  Resource-efficient | Limited Fault Tolerance  External data bese event management like Kafka,Rabbit MQ needed Need check their License Availability |
| Real-time vs Batch Processing | Streaming and Event-Driven Architectures | Continuous flow of data from producers to consumers  Reducing the need for intermediate storage | Limited Fault Tolerance |

# 5. Implementation

Application will be implemented in Java 8 coding language along with android specific java classes and complaint libraries.

# 6. Data Matrix

Implementation of the application considers Royal Enfield to provide the CAN data matrix in DBC format.

# 7. Test Plan

For the Vehicle Aggregator Application Separate Test Plan shall be provided with all the test case listed separately. This shall include Unit, Integration, System and Performance Tests

# 8.Bibliography

* SOR\_EV\_SW\_development – Veeraju, Venkat, & Tarun
* Data Aggregator.docx – Mr. Lakshmi Patel
* Data Aggregator - Explanation.pdf – Mr. Lakshmi Patel
* Android Development Guide

<https://developer.android.com/topic/architecture>