University of Stuttgart
Germany

University of Stuttgart
Institute of Architecture of Application Systems (IAAS)
Chair of Service Computing

Master Thesis

Evaluation and Integration of DDS Middleware for Interconnection between Android Automotive and AUTOSAR

Chandan Girish

Course of Study:	M.Sc. Information Technology Specialization Embedded Systems
Examiner:	Prof. Dr. Marco Aiello
Supervisors:	Andreas Schnell (M.Sc), ITK Engineering GmbH Steffen Rentz (M.Sc), ITK Engineering GmbH
Commenced:	01.07.2023
Completed:	02.01.2024

Acknowledgement

I extend my sincere gratitude and appreciation to all those who have contributed to the completion of my master's thesis.

First and foremost, I would like to thank the University of Stuttgart and ITK Engineering GmbH for giving me the opportunity to pursue my thesis.

I would like to thank my supervisors, Andreas Schnell and Steffen Rentz at ITK Engineering GmbH for their continuous valuable guidance, advice, and encouragement throughout the thesis work. I would like to also thank the colleagues and mentors from Real-time architecture team for their support and encouragement.

My sincere and grateful acknowledgment to Prof. Dr. Marco Aiello for examining my thesis work and reviewing all my steps during this process.

Finally, I would like to thank my family and friends for their constant moral support and encouragement for the entire duration of my master's thesis.

Abstract

Modern day vehicles are transforming into software-defined vehicles, where the functions and features are primarily enabled through software services. The increasing addition of functionalities has led to a rise in system complexity. The automotive system consists of different domains and components, developed simultaneously by different vendors, each serving different purposes and complying with different requirements and technologies. It is essential to establish effective interconnection between these domains for enhanced feature diversity and overall system performance. Advanced Driver Assistance Systems (ADAS) and In-Vehicle Infotainment (IVI) are the two major domains that are rapidly expanding and driven by increasing customer demands. Android Automotive has established itself as a major player in the car IVI landscape due to its extensive features and intuitive user interface. The difference in standards used by the two domains, i.e AUTomotive Open System ARchitecture (AUTOSAR) in ADAS and Android Automotive in IVI, leads to the necessity of a common interface and communication protocol which is supported by both domains.

This thesis presents the solution for intercommunication between AUTOSAR and Android platforms using Data Distribution Service (DDS) middleware. Additionally, the various features and service-oriented communication patterns offered by DDS in automotive applications are discussed along with its distinctive features compared to current protocol such as Scalable Service-Oriented MiddlewarE over IP (SOME/IP). The identification of key components and integration of DDS middleware approaches are central aspects of this thesis. As a proof of concept, the design was implemented with minor modifications, and successfully demonstrated on a hardware setup, exhibiting inter-domain communication using DDS middleware.

Contents

List of Figures

List of Tables

Acronyms

AA Adaptive Application.

AAOS Android Automotive Operating System.

ACC Adaptive Cruise Control.

ACK Android Common Kernel.

ADAS Advanced Driver Assistance Systems.

ADB Android Debug Bridge.

AIDL Android Interface Definition Language.

AOSP Android Open Source Project.

AP Adaptive Platform.

API Application Programming Interface.

APK Android Application Package.

ARA AUTOSAR Runtime for Adaptive Applications.

ART Android RunTime.

ARXML AUTOSAR XML.

AUTOSAR AUTomotive Open System ARchitecture.

AVD Android Virtual Device.

BSD Berkeley Software Distribution.

BSP Board Support Package.

CM Communication Management.

DCPS Data-Centric Publish Subscribe.

DDS	Data Distribution Service.
DDSI	DDS Interoperability Wire Protocol.
DEX	Dalvik EXecutable format.
DTLS	Datagram Transport Layer Security.
ECU	Electronic Control Unit.
HAL	Hardware Abstraction Layer.
HIDL	Hardware Interface Definition Language.
IDL	Interface Definition Language.
IoT	Internet of Things.
IP	Internet Protocol.
IPC	Inter Process Communication.
IVI	In-Vehicle Infotainment.
JNI	Java Native Interface.
LTS	Linux Long Term Supported.
MPC	The Make, Project, and Workspace Creator.
NDK	Native Development kit.
OEM	Original Equipment Manufacturer.
OMG	Object Management Group.
OS	Operating System.
OSI	Operating System Interface.
OTA	Over The Air.
QoS	Quality of Service.
ROS2	Robot Operating System 2.
RPC	Remote Procedure Call.

RTPS Real Time Publish and Subscribe Protocol.

SDK Software Development kit.

SHM SHared Memory.

SOA Service-Oriented Architecture.

SOC Service-Oriented Communication.

SOME/IP Scalable Service-Oriented MiddlewarE over IP.

TCP Transmission Control Protocol.

TLS Transport Layer Security.

UDP User Datagram Protocol.

UI User Interface.

UML Unified Modeling Language.

VHAL Vehicle Hardware Abstraction Layer.

XML Extensible Markup Language.

Chapter 1

Introduction

The automotive industry is undergoing a significant shift towards software-defined platforms, where software plays a central role in driving innovations [7], such as autonomous driving, in-car infotainment, and vehicle integration via cloud and internet platforms. The development of software-defined and the digitization of automotive systems have led to an increased consumer desire for improved interaction and connected experiences in modern vehicles. Modern day vehicles are capable of providing a wide range of software services to the customer requirements. Vehicle infotainment systems have evolved from a basic audio-based configuration to more complex, customer demanding touchscreen-based intuitive user interfaces and displays in head units offering various user interactive services such as navigation, voice assistance, media, and many more.

Android, a predominant Operating System (OS) in the smartphone sector has entered the IVI domain, with Android Automotive becoming a standard feature in many vehicles in recent years. It has found its place in the automotive industry due to its better user experience and feature diversity. Although the primary focus of Android Automotive is only infotainment and not functional safety, the interaction of Android Automotive with other vehicle domains and functions, including ADAS has its benefits. The ADAS domain is responsible for safety-critical features using various sensors to ensure a safe and comfortable driving experience. On the other hand, IVI domain is focuses on passenger entertainment and provides useful information about the driving conditions and the vehicle's state. Although these two domains perform different tasks, there is a set of features that are commonly required in both. This creates an implementation overhead, as similar functionalities need to be implemented in both domains. The exchange of data and resources reduces this redundancy in both domains [8].

For example, a navigation module, which is used by the Android system can also share its data with the ADAS domain. Similarly, Android can directly utilize the data from route planning or traffic signal detection algorithms within ADAS domain to enhance the overall user experience.

AUTomotive Open System ARchitecture (AUTOSAR) is a widely adopted standard in software development for most of the applications in the automotive industry. Within the context of this thesis, we refer to Adaptive AUTOSAR, which follows Service-Oriented Architecture (SOA) as its foundation framework for software architecture. This Service-Oriented Architecture (SOA) approach enables the dynamic integration and execution of software services and components. In the automotive industry, various domains, such as ADAS follow AUTOSAR standards, whereas Android Automotive has a different architecture developed by Google. Although both the domains follow different architectures, we can exchange the data using a suitable middleware protocol such as Data Distribution Service (DDS). Through a communication interface, the data unavailable in one domain can be transferred to the other.

DDS is a middleware protocol and technology specifically designed for real-time and data-centric communication between distributed applications and systems. DDS offers communication Application Programming Interfaces (APIs) and communication semantics that enable interaction between data providers and data consumers. In the latest versions of Adaptive AUTOSAR, DDS is integrated as a network binding within the ara::com module [5], which offers the communication interfaces between different domains using DDS. [7] discusses the comparative analysis of the communication mechanisms adopted for software components in upcoming vehicles, with focus on service-oriented architectures. Several studies have been carried out in the field of inter-domain communication, specifically focusing on two major automotive domains: ADAS and Android Automotive. One such study detailed in [9], discusses the service-oriented communication between ADAS and Android based IVI. Another approach, as detailed in [10], suggests the implementation of an Android native service with the extension of a middleware API. In this context, the Android-based IVI system receives vehicle information from a different domain. Furthermore, [8] explores the development of an inter-domain communication mechanism between ADAS and IVI systems, utilizing model-to-model translation between AUTOSAR XML (ARXML) and Android Interface Definition Language (AIDL) with automated code generation over SOME/IP. However, other than the previously mentioned studies of inter-domain communication between ADAS and IVI, literature explaining a complete solution to the question of inter-domain communication between Adaptive AUTOSAR and Android Automotive using DDS middleware was not available.

1.1 Problem Statement

In the context of this thesis, the IVI domain is based on Android Automotive Operating System (AAOS). Android Automotive, primarily designed for in-vehicle infotainment, poses challenges in intercommunication with other automotive domains such as ADAS due to its distinct standards and focus on user experience rather than functional safety. The Android Automotive standard is defined and developed by Google. The ADAS domain functionality is implemented using the Adaptive AUTOSAR standard. The divergence in standards, with Android Automotive commonly used for user interaction and infotainment and ADAS adhering to the AUTOSAR framework with its proprietary platform, presents a challenge to establish a seamless communication between these two domains.

These two standards were formulated by separate entities with different goals and backgrounds. AUTOSAR concentrates on standardizing software architecture and communication within vehicles, with a focus on functional safety, reliability, and scalability for automotive applications. Its strict safety regulations notably address the demanding needs of the automotive sector, setting it apart from Google's more generalized approach to consumer technology that lacks automotive safety standards. There is a need for a common interface that bridges the communication between the two domains. There are various automotive standard protocols, but in this thesis, we focus on service-oriented communication using DDS middleware protocol. The communication mechanisms available on both platforms differ from one another. Due to the different standards used by Android in IVI and AUTOSAR in ADAS, a common interface that is supported by both domains is necessary for inter-domain communication. The primary objective of this thesis is to propose a solution to address this specific challenge in achieving seamless communication between ADAS and IVI domains.

1.2 Research Methodology

To begin, a systematic study in the field of inter-domain communication between AUTOSAR and IVI domains was conducted. The challenges in the interconnection between different standards, each with different communication mechanisms were examined. A service-oriented approach for data exchange between Android automotive and AUTOSAR platforms using Data Distribution Service (DDS) middleware protocol was conducted. A study of the complete DDS middleware framework was conducted with respect to support the AUTOSAR and Android platforms. Adaptive AUTOSAR already has DDS as its network bindings, providing an inter-domain communication mechanism. Then various approaches in which DDS middleware can be integrated into the Android system were discussed and compared. Further, the appropriate approach suitable for our application was implemented in the Android Automotive environment to demonstrate the communication using DDS middleware. A simple android demo application was implemented incorporating DDS and communication with the remote application running DDS service was demonstrated.

1.3 Document Structure

The content in this thesis is structured into various chapters as follows:

- *Chapter 2* provides the basic knowledge needed to understand the necessary terms and topics covered in this thesis.

- *Chapter 3* discusses the state-of-the-art technologies and existing work related to this thesis. Additionally, based on the literature research four specific research questions are formulated, focusing on the integration of DDS middleware for inter-domain communication between Android Automotive and other automotive domains.
- *Chapter 4* mainly describes the concepts and context of DDS middleware in automotive application. Different approaches to integrate DDS in Android architecture were explained and compared based on ISO25010 analysis to get the pros and cons of the different approaches.
- *Chapter 5* This chapter details the implementation of one of the approaches from the previous chapter. The development of the Android app and including DDS library is explained and the key decision points and modifications are discussed.
- *Chapter 6* evaluates the implementation of the solution and also discusses the timing measurements obtained between the Android application and DDS service on PC.
- *Chapter 7* provides the conclusion of the thesis and a list of possible future works.

Chapter 2

Preliminaries

2.1 DDS

DDS is a middleware protocol and technology defined by the Object Management Group (OMG) as a standard. It is specifically designed for real-time and data-centric communication between distributed applications and systems. DDS offers communication APIs and communication semantics that enable interaction between data providers and data consumers. It provides a high-performance, real-time, interoperable, reliable and scalable framework for sharing data among devices and applications in an efficient manner. It exhibits its importance in various domains such as Internet of Things (IoT), mission-critical applications, defense, air traffic control, robotics, and many more.

2.1.1 DDS Architecture and Features

DDS is a data-centric publish-subscribe communication model, renowned for its numerous advantages including real-time capabilities, dynamic adaptability, reliability, scalability, and flexibility [11]. DDS is well suited for highly dynamic distributed systems, the network topology is dynamically discovered, the connections between nodes are established point-to-point, and there is no central broker as a single point of failure. DDS offers a rich set of QoS policies, making it more suitable for distributed system.

The OMG DDS standard architecture primarily consists of two layers as seen in Figure 2.1, the DDS Interoperability Wire Protocol (DDS) layer, which sits above the network transport layer. And the DDS layer, which defines the DCPS mechanism and the DDS API used by the application [12].

The DCPS layer serves as the fundamental and core component of DDS protocol [13]. This layer provides the basic services of communication and defines how DDS sends data from the publisher to the subscriber. Applications interact in a global data space, which is the top-level abstraction of DDS and contains all the data published within the

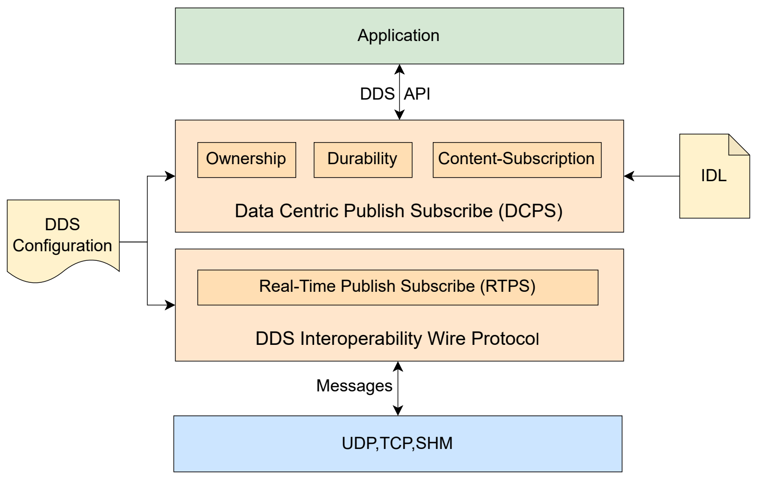

Application
DDS API
Ownership
Durability
Content-Subscription
Data Centric Publish Subscribe (DCPS)
IDL
DDS Configuration
Real-Time Publish Subscribe (RTPS)
DDS Interoperability Wire Protocol
Messages
UDP,TCP,SHM

Figure 2.1: DDS Architecture

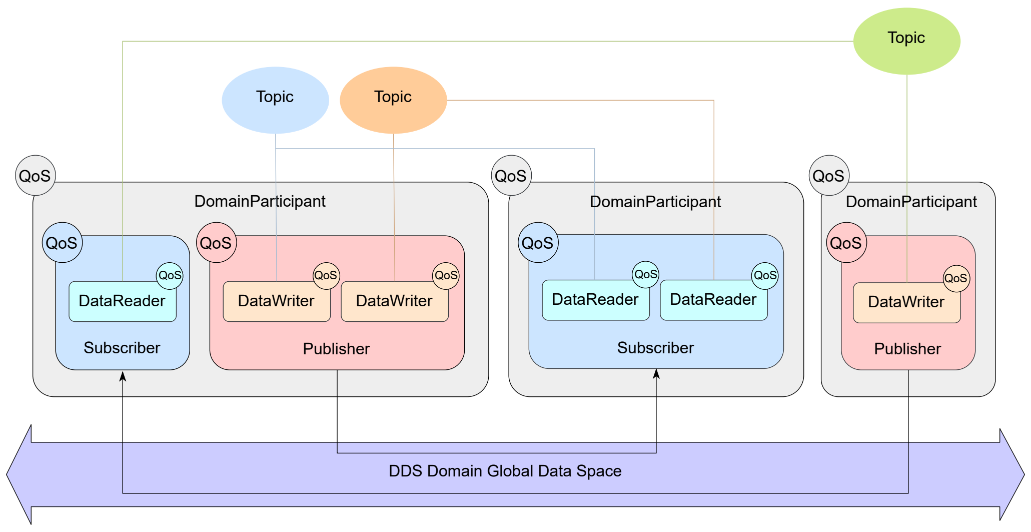


Figure 2.2: DDS domain diagram

domain as depicted in Figure 2.2. The domain is a virtual data space in which DDS client applications publish and subscribe to information given by a topic.

The main entities of DCPS are DomainParticipant, Publisher, DataWriter, Subscriber, DataReader and Topic. These entities work together to accomplish the publish-subscribe mechanism, which will be discussed in Section 2.1.2. Additionally, DCPS also defines QoS policies that specify minimum service levels necessary for communication between DCPS entities. For instance, ownership policy specifies whether multiple entities can simultaneously publish data (shared ownership), or if only one entity can publish data at a time (exclusive ownership). Some other policies that could be included are durability, history, resource limits, liveliness and many more. In Section 2.1.2 QoS policies are explained in detail.

Real Time Publish and Subscribe Protocol (RTPS) is the wire protocol, that enables compatibility between various DDS vendors and thus, helps in interoperability. It can be deployed over multi-cast communication and best-effort transport layer protocols, such as User Datagram Protocol (UDP)/IP [14]. RTPS is specifically designed to meet real-time publish subscribe requirements essential for DCPS application by incorporating timing parameters and properties [15].

DDS offers dynamic discovery of DataWriters and DataReaders, which improves the extensibility of DDS applications. With dynamic discovery, applications don't need to know about communication endpoints. The discovery process occurs at run-time enabling real "plug-and-play" for DDS applications [16]. The discovery mechanism is performed in two phases. First, there's the participant discovery phase, where each DomainParticipant acknowledges the existence of each other by sending announcement messages via multicasting periodically, including Internet Protocol (IP) address and port information. Two DomainParticipants will match when they are in the same DDS Domain. In the subsequent phase, DataWriters and DataReaders acknowledge each other using the communication channel established during the first phase. They exchange data if the topic, data type, and QoS parameters are same.

The Transport layer manages the actual sending and receiving of messages over a physical network and offers communication services for both data and discovery traffic communication between DDS entities. However, the DDS layer itself is independent of transport layer that allows for the customization of transport plugin based on the specific application requirements. Different DDS vendors provide various discovery settings and pluggable transport options like UDP, Transmission Control Protocol (TCP) and SHared Memory (SHM). These settings can be done through a configuration file. Since DDS API makes these features simpler for the application, we do not delve deeply into the transport layer and discovery mechanism.

An essential component of DDS middleware is its code generation capability. This produces source code based on the IDL, which can be utilized by DDS application to publish or subscribe the data. DDS IDL is the key part of DDS as it defines the data structures, topics, interfaces and other elements used to describe the data exchanged between different components of a DDS based system. IDL is a language-neutral specification standardized according to OMG specifications [17], which promotes interoperability across various DDS vendors and enables DDS applications developed in distinct programming languages like C, C++ and Java to collaborate seamlessly.

To build an application, the topic must be defined by means of an IDL file. The data type of a topic is defined in the IDL file.

```
module Temperature {
 @topic
 struct Message
   {
        long  value;
        @key string name;
   };
};
```

Listing 2.1: DDS IDL code example for temperature application [6]

The data type for a topic is described using two classes TopicDataType and TypeSupport [18]. TopicDataType describes the data type exchanged between DataWriter and DataReader, that is data corresponding to a Topic. While TypeSupport encapsulates an instance of TopicDataType providing the functions needed to register the type and facilitate interactions with both the publisher and subscriber components. For example, simple IDL file for Temperature application is explained in Listing 2.1. A data type that can be used as a topic's type is indicated by the `@topic` annotation. This must be a structure or a union. The structure or union may contain basic types, enumerations, strings, sequence, arrays, structures and unions [6]. Here, the IDL defines a structure Message in the Temperature module. The `@key` annotation identifies a field that is used as a key for this topic type. A topic may have zero or many key fields. These keys are used to distinguish various DDS instances within a topic. Each sample published with a unique `@key` name will be defined as belonging to a different DDS instance within the same topic. For instance, we can publish temperature value in Celsius and Fahrenheit just by creating two different topic instances.

2.1.2 DDS Communication Model

DDS middleware is a Data-Centric Publish Subscribe (DCPS) model. The key entities in its implementation are Publisher, Subscriber, DomainParticipant and configuration entities. Data domain consists of domain participants, Publisher and DataWriter for sending the data, Subscriber and DataReader for receiving the data as seen in Figure 2.2. The basic elements of the DCPS model are [18]:

- Publisher: Publishers are responsible for the creation and configuration of the DataWriters it implements. The objects that actually publish data are known as

DataWriters. DataWriters are associated with a single Topic object under which the messages are published.

- Subscriber: These are in charge of receiving the data published under the topics to which it subscribes. It serves one or more DataReader objects, which subscribes to the data under specific topic name and are responsible for communicating the availability of new data to the application. An application typically has a combination of DataWriters and DataReaders.

- Topic: It is the binding entity between publications and subscriptions. It is unique within a DDS Domain. With specific topic name, data type, and QoS policies, Publishers and Subscribers can exchange data unambiguously.

- Domain: DDS Domains provide a logical separation of communication. This concept is used to link all Publishers, Subscribers belonging to one or more applications, which exchange data under different topics. These individual applications that participate in a domain are called DomainParticipant. The DDS Domain is identified by a domain ID.
  An application becomes part of a DDS domain by creating a DomainParticipant associated with a specific domain ID. It is important to note that DomainParticipants in different DDS domains each with a distinct domain ID, do not communicate or exchange messages with one another. However, a single application can engage in multiple DDS domains by creating multiple DomainParticipants, each associated with a different domain ID. Different DDS domain can exist within a system, each isolated from the others. This allows for the creation of multiple independent communication spaces within a larger system.

- Quality of Service (QoS): It is a generic mechanism for the application to control the behavior of the service, such as resource consumption, fault tolerance, communication reliability and thus allowing the user to define how each entity will behave. Each entity (Topic, DataReader, DataWriter, Publisher, Subscriber, and DomainParticipant) has associated QoS that is comprised of individual QoS policies each one configuring different aspects of the entity behavior. DDS specification defines several QoS policies that applications use to specify their requirements to the service. Participants specify the behavior that they require from the service, the service decides how to achieve these behaviors. These policies can be applied to the various DCPS entities. Some of the basic QoS policies are described in Table 2.1.
  For instance, the Reliability QoS policy is used to configure the reliability of the communication between DataWriters and DataReaders. If a DataReader desire to receive data reliably by configuring the Reliability QoS policy, the corresponding DataWriter must be capable of delivering that level of reliability for communication to occur. If the DataWriter only offers a best-effort reliability, then there will be no communication. A detailed explanation of each QoS policy and their compatibility are detailed in [19] [20].

QoS Policies	Description
Liveliness	Controls liveliness checks, to make sure expected entities in the system are still alive
Reliability	Determines whether the service is allowed to drop samples
History	Controls what happens to an instance whose value changes before it is communicated to all Subscribers
Resource Limits	Controls resources that the service can use to meet other QoS requirements

Table 2.1: DDS QoS Policy description [6]

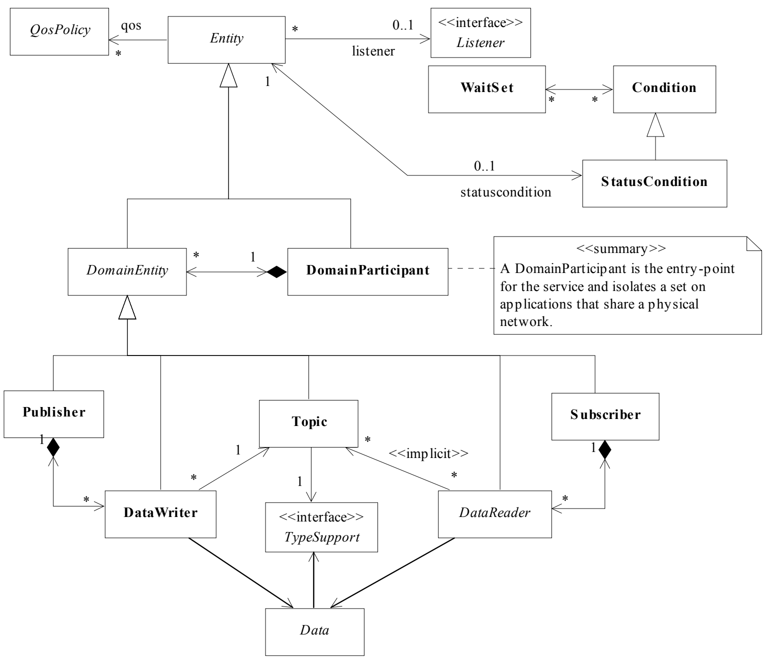


Figure 2.3: DCPS Conceptual Model

DCPS model represented in UML diagram is shown in Figure 2.3, all the main communication objects are attached to a single entity. Entity is the abstract base class for all the DDS entities, it is an object that has a communication status which can be configured by QoS policies. There are common features shared among all types of entities: QoS Policy, Listener and Status. The behavior of each entity can be configured with a set of QoS policies. Each entity is associated with a specific QoS class that groups all the policies customized to meet its requirements. Complete list of QoS policies are described in section 2.2.3 of DDS specification document from OMG organization [20]. Listeners offers a mechanism for the middleware to notify the application about the various asynchronous events such as the arrival of subscription data. Each entity creates an abstract listener interface with function callbacks, such as DataReaderListener, DataWriterListener, DomainParticipantListener, etc. The entity notifies the application of status changes using these callbacks. Listeners are closely connected to changes in status conditions. Each entity has a StatusCondition that acts as an interface with the Wait-set. When the Wait-set is used with conditions, it is possible for the middleware and the application to communicate differently based on Wait-set conditions rather than just notification based. In this process, the application uses Condition objects that are attached to Wait-set to specify the exact data it intends to receive. Once these conditions have been met, the requested information is retrieved. For example, when we subscribe to a topic, DataReaderListener interface has a function on_data_available(), which notifies DataReader on new data available. Once the new data is available, the StatusCondition flag is set to true, which in turn satisfies the Wait-set condition and DataReader can access this new data. All the DCPS entities are attached to a DomainParticipant. The DomainParticipant serves as the entry point and is linked to a single domain from its creation, all the entities are associated to that specific domain. A domain is a collection of entities that share a same communication infrastructure with similar communication objective. The publisher and subscribers may interact with each other, isolating them from entities on different domain. This allows to multiple distributed applications to coexist in the same physical network without causing interference. DomainEntity is an intermediate object whose solely purpose is to specify that a DomainParticipant cannot contain other domain participants.

In Data-centric communication, a unique Topic name is employed to establish the connection. Each Topic corresponds to a single data type. However, several topics may refer to the same data type. The DataWriters and DataReaders interact with the help of topic name. When the application decides to transmit data, a new DataWriter is created in a Publisher. This DataWriter is then associated with the Topic that describes the data type that is being transmitted. To start receiving the transmitted data, application creates a new DataReader in a Subscriber. This DataReader will then start receiving data values that matches the corresponding Topic and QoS policies.

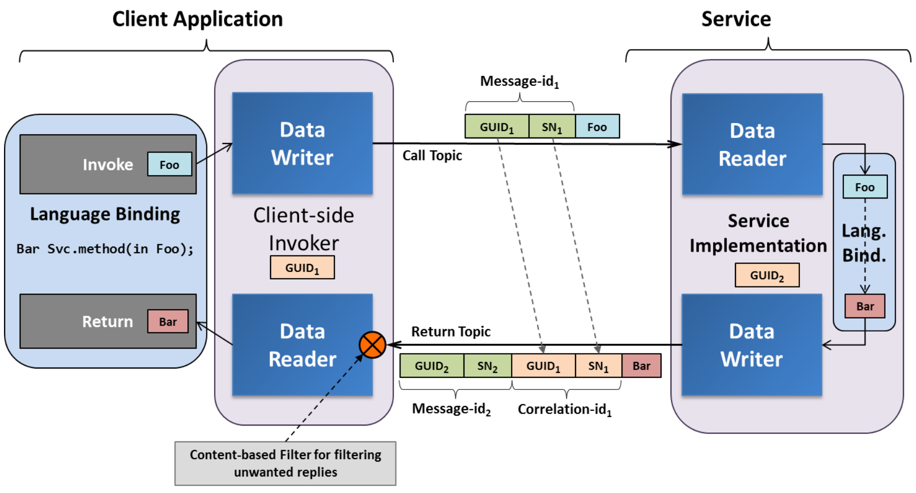


Figure 2.4: RPC over DDS [1]

2.1.3 DDS RPC

DDS emphasizes on data-centric and asynchronous publish-subscribe operations. DDS RPC is a high-performance remote procedure call framework that aims to specify higher-level abstractions built on top of DDS to achieve request-reply communication, promoting portability, interoperability, and a data-centric view for request-reply communication [1]. This enables the architectural benefits of DDS to be leveraged in request-reply communication. It uses the request-reply pattern as its core and main pattern, with the client sending a request message and the server sending a reply message. The framework can be implemented using two publisher-subscriber pairs, handling the request and reply using respective Request and Reply DDS Topics [21].

Remote Procedure Calls involve two participants: a client and a service. The client uses a datawriter to publish a sample representing the remote procedure call, while the service implementation has a datareader to read the method name and parameters. Figure 2.4 shows the high-level architecture of the Remote Procedure Call over DDS. The service computes the return values and sends them back to the client. A content-based filter is used to ensure that the client receives a response to a previous call made by itself. It is crucial to correlate requests with responses, especially when using asynchronous invocations. Requests are identified using a unique SampleIdentity, which is a struct composed of GUID_t and SequenceNumber_t. When a service implementation sends a reply to a specific remote invocation, it is necessary to identify the original request by providing the sample-identity of the request. Refer to the DDS RPC specifications [1] for more information on RPC over and IDL files.

2.2 Android

The Android OS is a popular and open-source mobile operating system developed by Google, primarily designed for touchscreen smartphones and tablets. It is based on a modified version of Linux kernel for its optimized power and process management for mobile devices. Android offers a flexible platform for app development with its extensive ecosystem of apps and service through the Google Play store. It is well known for its user friendly interface and regular updates, making it a dominant player in the mobile device market [22].

2.2.1 Android Platform Architecture

Android adopts a layered architecture based on modified Linux kernel, Figure 2.5 shows the android software stack. In this section we explain the different layers of android architecture.

- Linux Kernel: The Linux kernel, which manages low-level hardware interactions, form the core of the android platform. It is constructed on top of an upstream Linux Long Term Supported (LTS) kernel and is modified for special needs in power management, memory management and the run-time environment to create what's known as Android Common Kernels (ACKs) [23]. Android features a distinct driver known as Binder for facilitating IPC mechanism and remote method invocation system.

- HAL: The HAL serves as a bridge between the Linux kernel and the upper layers. It offers standardized interfaces that expose device hardware capabilities to the higher level Java API framework. It consists of multiple libraries and drivers tailored for specific hardware components such as camera, sensors, and audio. Starting from android 8, the Hardware Interface Definition Language (HIDL) serves as an interface description language used to define the communication interface between a HAL and its users. HALs created with HIDL are reffered to as binerized HALs becuase they can communicate with other architecture layers using binder IPC calls.

- Android RunTime (ART) and Core Libraries: ART is responsible for executing android applications, it compiles byte-code into native machine code. Android applications are written in Java and run in Dalvik virtual machine, a virtual machine featured in android. Each app has its own instance of ART running under its own process. ART is written to run multiple virtual machines on low-memory devices by executing Dalvik EXecutable format (DEX) files, which is a byte-code format developed particularly to run on android and optimized for a minimal memory footprint [2]
  Android core libraries are composed of Java libraries that support application development. These libraries provide functions for creating user interfaces, data storage, and interactions with the android framework

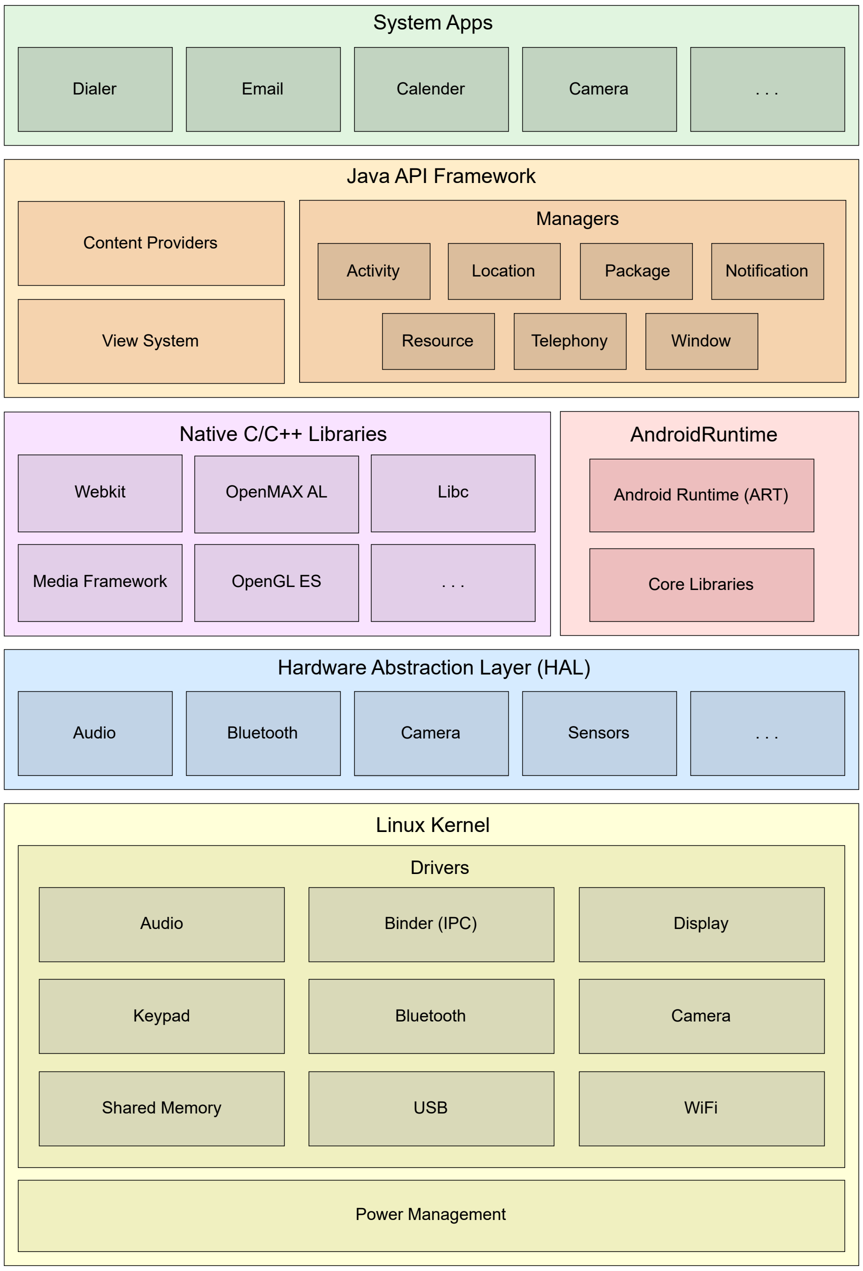


Figure 2.5: The Android Software Stack [2]

- Native C/C++ Libraries: The core system components and services in android, such as ART and HAL, are built using native code, that requires native libraries written in C and C++. Applications can access the services and functionality of these native libraries using Java framework APIs. For example, 'Media Framework' is a multimedia library that supports recording and playback of various commonly used audio and video formats. 'Libc' is a standard C system function library inherited from Berkeley Software Distribution (BSD), specially customized for embedded Linux-based devices. If an app development requires C or C++ code, we can use the Android Native Development kit (NDK) to directly access the native platform libraries [2].
- Java API Framework: The Application framework offers a set of standardized services and API for app development. These APIs simplify the reuse of core, system components and services for app development. It includes components like content providers, view system and managers. For instance manager component consists of activity, that manages the life cycle of each application and notification, which allows the application to display custom prompt information in the status bar and many more.
- Applications: Applications constitute the topmost layer. This layer includes system apps which are preinstalled applications on android devices, such as dialer, mail and calendar. System apps serve as user apps and offer key developer capabilities, for example, if an app needs to send SMS messages, it can leverage existing SMS apps rather than building that functionality from scratch. Applications are typically developed in Java or Kotlin and can include an UI and/or background services. Apps receive software updates without requiring a system restart. Additionally, users can install other third party apps from the Google play store.

2.2.2 Android IPC

Unlike the Linux OS, which employs various techniques for achieving IPC, Android's modified Linux kernel comes with a binder framework which enables RPC mechanism between the client and server processes. Android Binder IPC is a robust, efficient, and secure mechanism used for IPC in the android ecosystem. It allows different processes to communicate with each other, often used for communication between applications and system services.

The Binder IPC follows a client-server model. When a client initiates communication, the Binder takes charge of locating the target service, verifying caller privileges, managing the communication, and message delivery. The service location phase is managed by the service manager, which serves as the endpoint mapper. Service manager maintains a service directory that maps interface names to binder handles. Consequently, when the binder receives a request for a specific service, it queries the servicemanager, which, after permission checks, returns a handle as demonstarted in Figure 2.6. If the client has

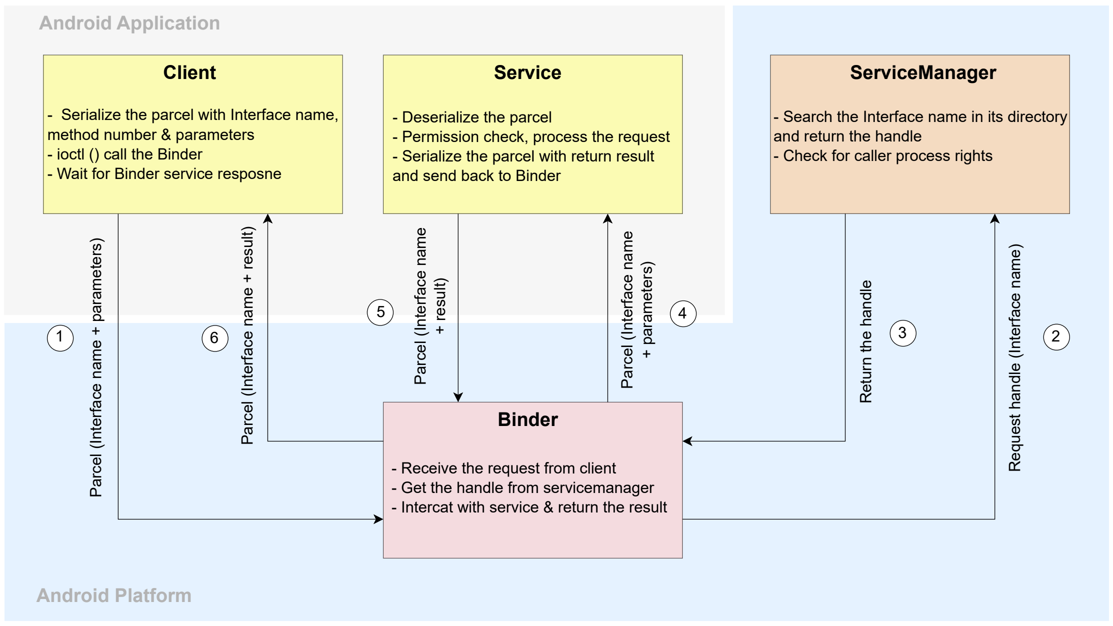

Android Application
Client
- Serialize the parcel with Interface name, method number & parameters
- ioctl () call the Binder
- Wait for Binder service resposne
Service
- Deserialize the parcel
- Permission check, process the request
- Serialize the parcel with return result and send back to Binder
ServiceManager
- Search the Interface name in its directory and return the handle
- Check for caller process rights
1 Parcel (Interface name + parameters)
2 Request handle (Interface name)
3 Return the handle
4 Parcel (Interface name + parameters)
5 Parcel (Interface name + result)
6 Parcel (Interface name + result)
Binder
- Receive the request from client
- Get the handle from servicemanager
- Intercat with service & return the result
Android Platform

Figure 2.6: The Binder IPC Communication [3]

permission to interact with the requested service, the Binder handles communication on the client side by creating a proxy and forwards the message to the server. The server processes the request and sends back the result to the binder. The binder, in turn, delivers it to the client as a 'message', technically referred to as 'Parcels'. These Parcels serve as containers for data, enabling both client and server to communicate through serialization and deserialization.

2.2.3 Android Interface Definition Language (AIDL)

Android OS uses the AIDL to define and communicate interfaces across various processes within an android application. AIDL functions similar to other Interface Definition Language. It allows us to define the programming interface that both the client and service must adhere to when communicating with each other through IPC. AIDL is a tool that simplifies IPC by abstracting the process. By providing an interface definition through a .aidl file, different build systems can use the AIDL binary to generate C++ or Java bindings. This enables the use of the specified interface across processes or components, such as activities, services, or other apps.
AIDL uses the binder kernel driver for Inter Process Communication (IPC). When a method call is made, a method identifier and all related objects are packaged into a buffer and copied to the target process. A binder thread will be awaiting to receive and read this data in the remote process. The binder thread looks for a native stub object in the local process as soon as it receives the data for a transaction. This native stub class opens the data and calls a method on an object that represents a local interface. The

server process creates and registers the local interface object. When a calls are made within the same process the same backend, there is no need for proxy objects, and hence calls are direct without any packaging or unpacking [23].

```
package my.package;

import my.package.Baz; // defined elsewhere

interface IFoo {
    void doFoo(Baz baz);
}
```

Listing 2.2: An example of AIDL interface [23]

Listing 2.2 shows simple AIDL interface example. The object 'Baz' is a parcelable object specified within the AIDL interface. Through AIDl, parcelable objects make it possible to serialize and deserialize of complex custom objects. In order for the 'Baz' object definition to be utilized and referenced within the AIDL definition, it should be created separatly. This separation allows for better organisation and reuse of complex data structures in AIDL interfaces.

2.3 Android Automotive

Android Automotive is an Operating System platform developed by Google for IVI and telematics systems that runs directly on the hardware within the vehicle. It is based on Android, full-stack, open source, highly customizable platform that supports both second and third-party android applications in addition to the pre-installed IVI system applications. Here's an overview of the key components and concepts of Android Automotive architecture.

The Android Automotive is an extension of the Android Open Source Project (AOSP) system architecture. The additional components such as Vehicle Hardware Abstraction Layer (VHAL), Car Service, Car Manager are added to provide functionality that are relevant to an automotive context. Figure 2.7 shows an abstarct layer architecture with division in four layers, namely the Board Support Package (BSP), HAL, Service layer and Application Framework.

- Board Support Package (BSP) is a Linux kernel image with HAL implementation for hardware, part of the 'vendor' partition. It allows Original Equipment Manufacturers (OEMs) to extend source code with self-developed applications and system services, such as head-up display management and tire pressure monitoring. Project Treble from Android 8.0 aims to simplify this process by separating the Android OS framework from device-specific drivers and low-level software, allowing manufacturers to update devices quickly without constant changes to low-level software [24].

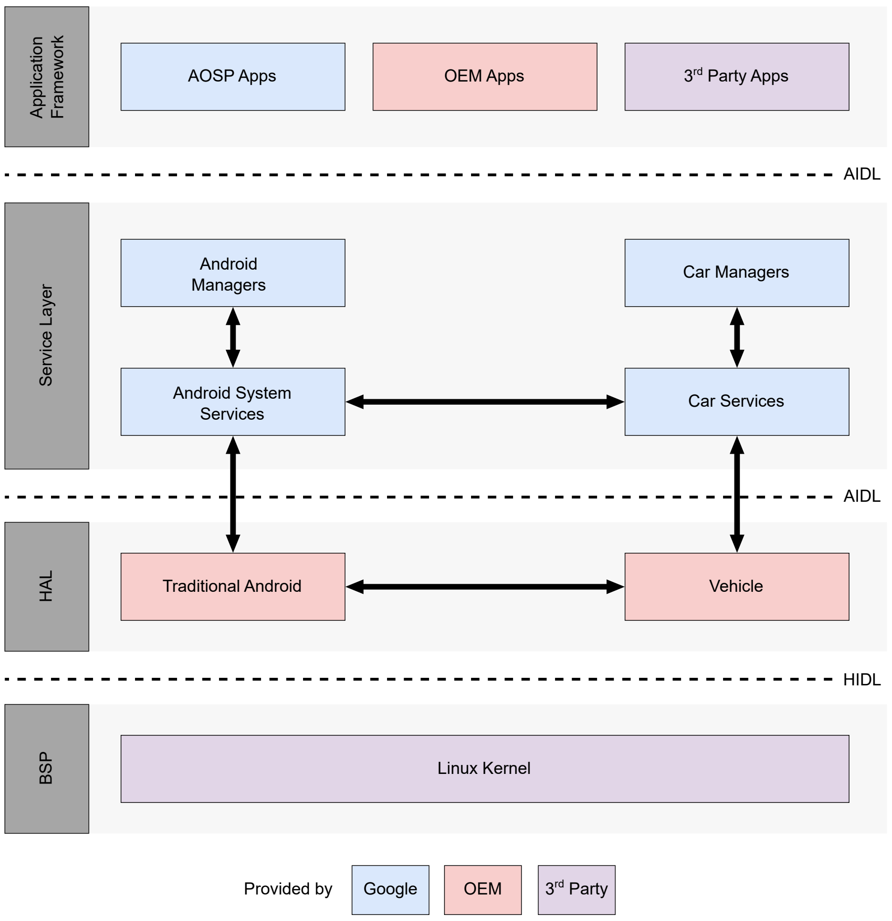


Figure 2.7: Android Automotive Architecture [4]

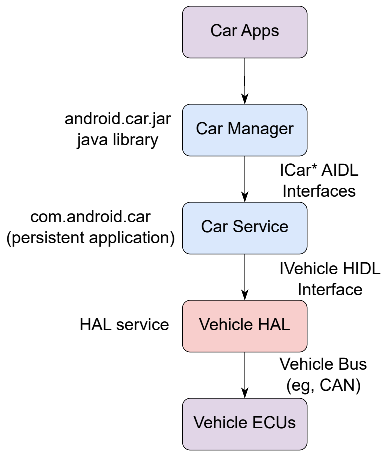
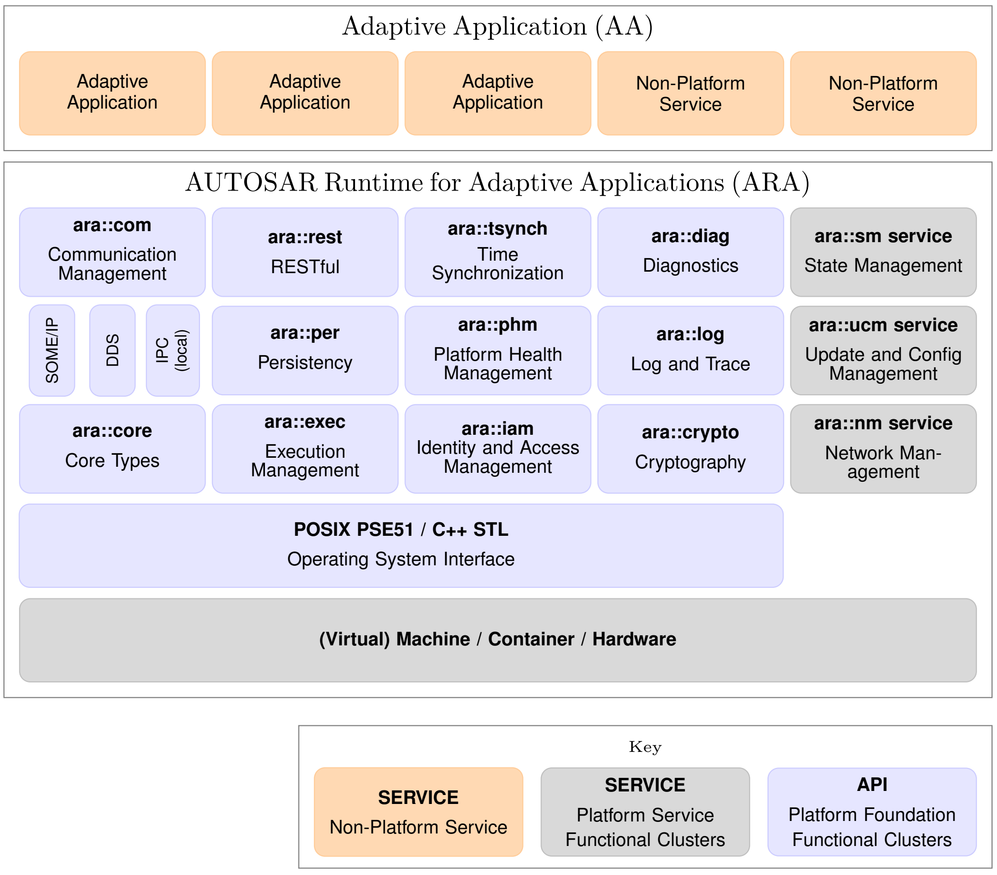


Figure 2.8: Vehicle Data Flow [4]

- Vehicle Hardware Abstraction Layer provides a standard interface to the android framework, regardless of the underlying physical transport layer. It stores the data in the form of vehicle properties. Numerous of these properties are linked to signals that are present on the communication bus of the vehicle, such as the speed of the car or the air conditioner's temperature. Through the VHAL, the data from these signals is obtained from a transport layer (like CAN) and then made accessible to higher layers, in particular the Service Layer and the Application Framework. Figure 2.8 shows the data flow from VHAL to applications. Properties can be changed either automatically, if the signal on the bus changes or programmatically, by an android application, allowing for dynamic control and interaction with the vehicle's data [25].
- Car Service is a part of the Service layer started by the SystemServer. These service run as a system process, providing additional privileges that normal android services do not. The Car Service incorporates vehicle properties and offers a number of useful APIs for applications. The 'com.android.car' persistent system app contains this service as a system service. When the first app tries to connect to the Car Manager, it becomes operational as a service.
- Car Manager facilitates accessing services from car service layer to application layer. Pre-installed on the device, Car Manager is a platform library that contains the android.car.* classes, which together make up the API for interacting with car-related services. To access these services, applications must be given the necessary permissions [26].
- Application Framework contains the system and user applications. OEMs can have their own system specific applications pre installed, that are tailored for in-car use. These apps cover functions like navigation, media playback, phone integration, settings, and more. Just like standard android, android automotive allows developers to create apps specifically designed for in-car use. These apps can interact with vehicle features, access sensors, and provide a more personalized experience for drivers and passengers.

2.4 AUTOSAR Adaptive

Adaptive AUTOSAR is an evolution of the AUTOSAR standard, developed to address the complexity of automotive software systems and the need for advanced, connected, and autonomous capabilities in modern vehicles. It is more adaptable and dynamic than classic AUTOSAR, enabling the development of more complex and adaptive software systems. It focuses on connectivity and communication within the car, enabling features like over-the-air updates, vehicle-to-vehicle and vehicle-to-infrastructure communication. It is compatible with multi-core CPUs and offers real-time capabilities for ADAS and autonomous driving. The platform supports over-the-air software updates, particularly for microprocessor-based Electronic Control Units (ECUs), ensuring reliability and security



Adaptive Application (AA)
Adaptive Application
Adaptive Application
Adaptive Application
Non-Platform Service
Non-Platform Service
AUTOSAR Runtime for Adaptive Applications (ARA)
ara::com Communication Management
ara::rest RESTful
ara::tsynch Time Synchronization
ara::diag Diagnostics
ara::sm service State Management
SOME/IP
DDS
IPC (local)
ara::per Persistency
ara::phm Platform Health Management
ara::log Log and Trace
ara::ucm service Update and Config Management
ara::core Core Types
ara::exec Execution Management
ara::iam Identity and Access Management
ara::crypto Cryptography
ara::nm service Network Management
POSIX PSE51 / C++ STL Operating System Interface
(Virtual) Machine / Container / Hardware
Key
SERVICE Non-Platform Service
SERVICE Platform Service Functional Clusters
API Platform Foundation Functional Clusters

Figure 2.9: Android Automotive Architecture [4]

in the development of advanced vehicle functions [27]. A Service-Oriented Architecture (SOA) serves as the foundation for the software architecture, enabling the dynamic integration and execution of software services and components.

2.4.1 Adaptive AUTOSAR Architecture

The top layer of the AUTOSAR Adaptive Platform (AP) is the AAs, which can be single or multi-threaded processes. Figure 2.9 shows the architecture of adaptive platform. The Adaptive Application (AA) run on top of ARA, which consists of application interfaces provided by functional clusters, either belonging to the Adaptive Platform Foundation or Adaptive Platform Services. The Adaptive Platform Foundation offers the core and essential functionalities of the AUTOSAR AP, and Adaptive Platform Services provide standardized platform services that are part of AP. Any AA can provide services to other AAs, illustrated as Non-Platform Service [27] in Figure 2.9.
Some of the key terms that are necessary to understand the thesis are defined below.

- Machine: The hardware on which the AUTOSAR Adaptive Platform runs is referred to as the 'Machine'. This hardware may be implemented as a physical machine, a fully virtualized machine, an OS-level-virtualized container, a para-virtualized OS, or in any other virtualized environment [27].
- Operating System Interface (OSI): The OSI is compliant with POSIX PSE51 [28]. By providing a standardized architecture for communication between the program and the underlying operating system, it provides the interfaces required for the creation of multi-threaded real-time embedded applications.
- Functional Clusters: The Adaptive Foundation and Adaptive Services software is presented as functional clusters, similar to the classic Platform's basic software (BSW). These clusters offer functionalities as services to the application, but are now processes that can be single-threaded or multi-threaded. To name a few functional clusters in AP are Persistency, Diagnostic Management, Communication Management, Execution Management. In this thesis, we provide a brief overview of the communication management module (ara::com). This module mainly serves as connectivity within the AP components as-well as with external machine.

2.4.2 Communication Management

Communication Management (CM) in adaptive AUTOSAR is a functional cluster within the AUTOSAR Runtime for Adaptive Applications that manages communication channels between local and remote applications. It adheres to a Service-Oriented Architecture approach and facilitates communication between Adaptive AUTOSAR applications and software entities on other machines, including classic AUTOSAR software components and non-AUTOSAR domain. CM includes generic parts for brokering and configuration, as well as specific parts for service providers and service consumers [5].

2.4.3 Communication Paradigms

Service-Oriented Communication (SOC) is the primary communication pattern for adaptive AUTOSAR applications, enabling the establishment of communication paths even at run-time. It is suitable for building dynamic communication networks with unknown participants. The fundamental principle of SOC is depicted in Figure 2.10.
Service Discovery plays a crucial role in deciding whether external and internal service-oriented communication should be established. Communication Management software should offer optimized implementation for both service discovery and communication connection, depending on the service provider's location. The service class is the central element of the service-oriented communication pattern, representing the methods and events provided by applications that implement specific service functionality.

The ara::com API uses the Proxy-Skeleton pattern [5] to generate two distinct code artifacts based on a formal service definition as shown in Figure 2.11.

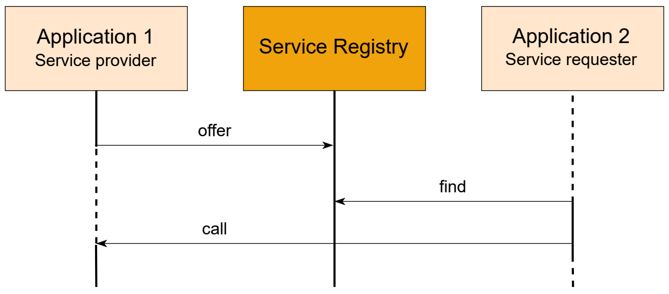


Figure 2.10: Service-Oriented Communication

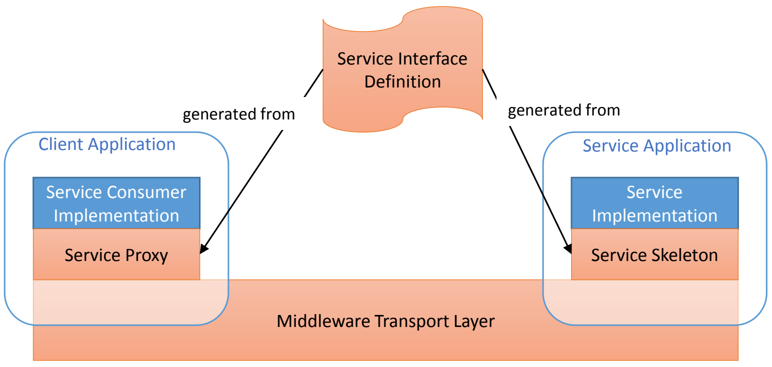


Figure 2.11: Proxy Skeleton Pattern [5]

- Proxy: A proxy acts as a local representation of the service for the consumer at the code level when seen from the perspective of the service consumer attempting to use a remote service. It appears as a C++ class instance that resides inside the program or client utilizing the service. The Proxy class allows the consumer's code to interact with the service as if it were a local component, even though the service might be physically located remotely.
- Skeleton: A Skeleton is a component of a service implementation that connects a user-provided service implementation to the middleware's transport infrastructure. This enables access to and interaction with the user-provided service implementation through the middleware's communication and transport layer, facilitating communication between service functionality and users.
  A service is defined in a service interface description file, where the ServiceInterface is structured with Fields, Methods, or Events. This structure is intended to facilitate communication between the inter- or intra-ECU communication between applications. Further, the various communication patterns used to model a ServiceInterface are presented.

Having discussed the proxy-skeleton pattern. Let us dive into how communication occurs between proxies and skeletons within ara::com framework. Adaptive AUTOSAR follows SOC paradigms to establish communication channels between server and a client, ara::com defines four distinct mechanisms to communicate [5] as shown in Figure 2.12. To use communication mechanisms, a service must be instantiated, defined, and offered to the system through an 'OfferService()' method. The client then connects to the service instance using the Proxy, acting as an intermediary for client-server communication.

- Methods: This mechanism involves method calls to carry out particular tasks or obtain data from the server. For instance, Request-Response methods, where clients send requests to servers, and servers respond to those requests by sending responses back to clients.
- Events: Events are a one-way channel of communication between the server and the client that let the server alert the client to certain occurrences or changes. The server sends interested clients an event message. The client which needs these events are determined via service discovery.
- Fields: clients can read or modify particular data properties by using Fields to access data attributes within the service. A Field can have an optional getter (a request/response method to read out the field's current value), optional setter (a request/response method to update the field), and optional notifier (an event to be dispatched cyclically or on-change).
- Triggers: Triggers are used to start actions or events based on predefined conditions or events in the system. It is used to notify when a specific condition occurs, and it does not transfer any data

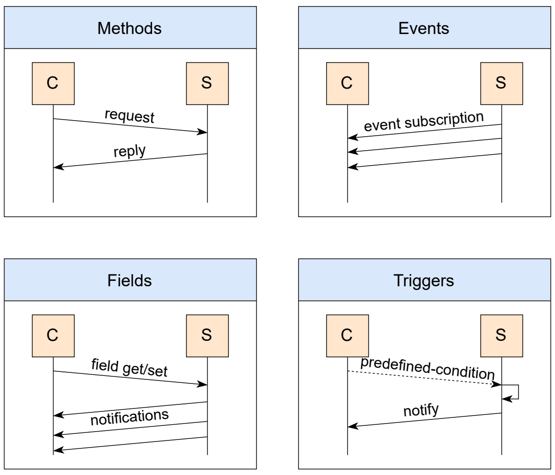


Figure 2.12: Communication Services

Chapter 3

State of the art

This chapter discusses the findings from a literature review focused on the current state of technology for inter-domain communication between Android Automotive and Adaptive AUTOSAR. As software-defined vehicles become more popular, the complexity of vehicle networks rises due to the interaction of several domains, requiring the use of multiple technologies. OEMs are particularly interested in the problem of connecting domains between Adaptive AUTOSAR and Android Automotive. However, many of the most recent developments in this field remain confidential due to the competitive nature of the industry. As a result, the available literature as part of open research is quite limited. Nevertheless, the resources that are openly published and open-source examples were analyzed. Section 3.1 describes the adaptation and use cases of DDS middleware protocol in automotive application. Section 3.1.1 covers the interface definition language of both Android and Adaptive platforms. In Section 3.1.2, the related work that has been carried out on middleware integration for inter domain communication. Finally, in Section 3.1.3 we derived four research objectives that will guide this thesis work based on the overall problem statement discussed in Section TODO«add intro and problem statement section».

3.1 DDS Middleware in Automotive Systems

Modern vehicles are increasingly transforming into software-defined platforms, increasing system complexity [29]. The automotive field has transitioned from signal-based communication to service-oriented communication, where vehicle components are clustered into domains based on specific functions, leading to decoupling of the system [30]. Service-Oriented Architecture (SOA) in automotive software architectures describes functionality through software component interaction.
DDS is a standardized, middleware protocol that is widely used in distributed systems for data-centric communication. It provides a framework for sending and receiving data between different software components. DDS is built on a publish-subscribe commu-

nication model, ensuring efficient data distribution between data producers (publisher) and data consumers (subscriber). DDS implements real-time data sharing, QoS configurability, and support for various data types, ensuring efficient data sharing between publishers and subscribers.

DDS middleware has showcased its effectiveness and capabilities in facilitating real-time communication within a complex distributed network [31]. The significance of DDS middleware in the automotive sector has grown as a result of the complexity of modern vehicles and the demand for sophisticated communication systems. As vehicles evolved into highly advanced and autonomous systems, the need for efficient and reliable data exchange is crucial. DDS middleware is finding its place in automotive systems, as a solution for inter-ECU communication, infotainment systems, Over The Air (OTA) updates, and Advanced Driver Assistance Systems. In [32], authors described how the automotive industry is moving towards Service-Oriented Architecture and benefit from the features offered by DDS middleware.

Due to its capability for real-time communication, event-driven architecture and configurable QoS, DDS was particularly suited for automotive applications. This made it possible for exchange of crucial information between various components in a vehicle. The theoretical basis for DDS functionalities and their key role in future inter-vehicle communication systems are provided in [31], along with simulation results. In [33], the feasibility of DDS for automotive SOA concepts with performance benchmark experiments to assess the benefits and drawbacks of DDS is discussed. DDS also finds application in ADAS components because of its real-time capabilities. By configuring suitable QoS policies that can effectively improve the performance of Adaptive Cruise Control (ACC) systems, article [34] has demonstarted the effectiveness of introducing DDS middleware into ACC systems.

Integration with adaptive AUTOSAR as a network binding in ara::com [5], cloud integration for connected vehicles, and OTA updates are recent developments and trends in the use of DDS middleware in the automotive industry [32].

3.1.1 Android Automotive and AUTOSAR platforms

AAOS is designed specifically for use in vehicles, offering in-vehicle infotainment, connectivity and interaction with external services such as weather updates, traffic information, sunroof control, etc. It allows vehicles to provide functions like navigation, multimedia, and communication tools, similar to those found on smartphones. The AAOS was introduced in March 2017 [35], and runs directly on the car's hardware (head unit) creating a dedicated android environment.

Android Automotive is a variation of the Android Open Source Project (AOSP) that is tailored to the unique requirements of the automotive industry. This benefits OEMs to build their own custom IVI systems from AOSP source-code. Google also offers bundled services that include Google assistant, Play store, and Google maps. Its user-friendly interface, simplified development, and app ecosystem have made it popular in modern vehicles. Through collaborations with third parties, it keeps evolving, adding new apps and services for better in-car experience for both drivers and passengers. Automotive infotainment architecture based on google android is discussed in [36]. Different layers on android automotive architecture are covered in detail in Section 2.3.

AUTomotive Open System ARchitecture (AUTOSAR) is a standardized software architecture and development framework for the automotive components. Traditional classic AUTOSAR was evolved into adaptive AUTOSAR in order to handle the complexities of modern, highly connected vehicles. It provides a structured method for developing and managing complex software components in modern vehicles and focuses on flexible, high-performance computing platforms like domain controllers and central processing units [37]. Section 2.4 provides a detailed explanation of the adaptive AUTOSAR architecture.

An Interface Definition Language (IDL) is a descriptive language used to describe the data types and interfaces of software components in a language-independent and platform-independent manner [17]. IDLs are commonly used in distributed computing and RPC systems to define how different software components or systems can interact with each other.Each of these domains has a unique set of definition languages designed for their particular use. The AUTOSAR framework uses a definition language known as ARXML, which is a file format derived from Extensible Markup Language (XML), a widely used modeling and markup language for automotive application [38]. It is used to build models by defining software components, port interfaces, and data elements. It represents various aspects of an automotive software system, including software components, communication interfaces, and system descriptions. In AUTOSAR development, software components and their interfaces are configured, exchanged, and integrated using ARXML files. For instance, ARXML file format is used to generate proxy and skeleton as shown in Figure 2.11. DDS uses Interface Definition Language as specified by OMG group. DDS is a part of ara::com network binding in latest adaptive AUTOSAR version. DDS types are defined according to OMG IDL version 4.2. AUTOSAR uses equivalent XML syntax for code generation and service interface definition [39].
Android supports its own interface definition language known as Android Interface Definition Language (AIDL) for facilitating Inter Process Communication (IPC) between different android components, such as between applications and services running in separate processes. Compared to ARXML it is a much simpler interface definition language, covering only the communication interfaces [40]. In android, AIDL is mainly used to define a remote service interface. This allows processes to communicate and exchange data through predefined interfaces in a client-server manner. The communication mechanism in android is supported by Binder, and Section 2.2.2 provides a detailed explanation of the IPC Binder mechanism in android. Figure 2.6 illustrates the communication between generated client and server components.

3.1.2 Integration of Middleware in Automotive Systems

Service-Oriented Architecture-based architectures, originating from web technologies, offer low complexity and integration effort, dynamic component addition, and network connectivity, making them ideal for automotive software development due to their reusability and decoupled components [41]. AUTOSAR has successfully integrated SOA into their platform standards. To accommodate future needs and new features AUTOSAR has restructured its portfolio and introduced AUTOSAR Adaptive platform with Ethernet capability and Service-Oriented Communication [42]. In order to facilitate inter- application communication between software components within Service-Oriented Architecture framework, it is essential to employ a Service-Oriented Communication middleware. Although there are a number of middleware solutions available, each designed for specific applications, it is crucial to adopt the necessary middleware according to the automotive domain requirements. DDS is one such middleware protocol that fits in the automotive context with real-time capabilities and configurable QoS [29] [33] [41] [7].

Advanced Driver Assistance Systems (ADAS) and In-Vehicle Infotainment (IVI) systems are two rapidly evolving automotive domains. The fundamental objective of ADAS is to enhance driving safety and experience by providing assistance to drivers in multiple ways. IVI, on the other hand, focuses mostly on providing entertainment and information to passengers in a vehicle. Connecting these systems has many benifits, such as enhanced safety: ADAS system can make use of navigation data from IVI, user experience can be improved by sharing various sensors data readily available from ADAS and also we can make use of surrounding camera sensors from ADAS which will be cost effective.

The main objective of this thesis is to establish a communication between two major domains in automotive system, Adaptive AUTOSAR and Android Automotive using DDS. [9] is one such work, which proposes a service-oriented communication between ADAS and IVI. Implementing an Android native service with the extension of a middleware API is another method suggested in [10]. In this way, the Android-based IVI system receives vehicle information from a different domain. And [8] explores the development of an inter-domain communication mechanism between ADAS and IVI systems, utilizing model-to-model translation between ARXML and AIDL with automated code generation over SOME/IP. All of the mentioned works were implemented using SOME/IP protocol, which is specifically created for automotive applications. Despite theoretical consideration of DDS protocol for serivce-oriented communication between ADAS and IVI systems, it was not practically implemented due to its limited acceptance in the automotive industry, likely due to challenges in aligning with existing automotive standardization efforts. Although the DDS middleware is not yet well suited for the automotive domain, this creates a space for it to apply some of its distinctive features in automotive communication.

3.1.3 Research Questions

It is vital to study on concepts of middleware and inter-process communications in order to facilitate communication between two different automotive domains with different architectures. Based on the understanding of inter-domain connectivity gained in the earlier sections, we have formulated specific research objectives that need to be explored and analyzed in the context of this thesis work.

[Q1.] What is the impact or expressive power of DDS based middleware on the latency, scalability, security, interoperability and reliability for service-oriented approach for interconnection of ADAS and IVI system, and how does it compare to alternative communication frameworks such as SOME/IP?

[Q2.] What are the key challenges and communication requirements when integrating systems with different communication paradigms like adaptive AUTOSAR and android automotive, and how can DDS middleware be effectively utilized to address these challenges?

[Q3.] How can the integration of DDS services in android automotive be achieved, what are the different concepts to integrate DDS protocol considering the software architecture and communication requirements of the system (IPC mechanisms)?

[Q4.] What are the advantages and limitations of using DDS middleware as the communication interface between AUTOSAR and android automotive systems and what is the round-trip time between two systems?

Chapter 4

Concept

This chapter mainly addresses the first and second research questions [Q1,Q2] mentioned in previous section. In Section 4.1, we discuss the key features of DDS in the context of automotive application and also comparing it with SOME/IP protocol, which is well established automotive middleware protocol. Section 4.3, explores the integration of DDS middleware in Android system and the concepts of different approaches for DDS integration. And finally, in Section 4.4 we evaluate the advantages and disadvantages of different approaches using ISO25010 standard.

4.1 Data Distribution Service (DDS) and its Key Features

DDS is a publish-subscribe standardized middleware protocol based on a data-centric approach. Its main purpose is to enable communication among distributed applications. DDS is widely used in high-performance, real-time applications such as aerospace and defense systems, distributed systems, medical and Internet of Things (IoT) applications, as well as in industrial automation.

The foundation of DDS is a 'global data space', a databus responsible for the data exchange between publishers and subscribers. The DDS middleware notifies all the subscribers that are interested in a specific data (Topic), whenever a publisher publishes new data to this global data space. This data-centric communication approach provides the advantage of decoupling publishers and subscribers, which leads to an architecture that is highly scalable and flexible. Section 2.1 explains the DDS architecture and communication model. Figure 4.1 illustrates a simplified representation of the communication between sender and recipient applications operating in the same domain. The sender must first initialize DomainParticipant, Topic, Publisher, and DataWriter, while the receiver must instantiate DomainParticipant, Subscriber, and DataReader respectively, in order to exchange data. The DDS middleware is then in charge of enabling communication between these two endpoints and ensuring the successful delivery of transmitted messages. The Real Time Publish and Subscribe Protocol (RTPS) is the underlying

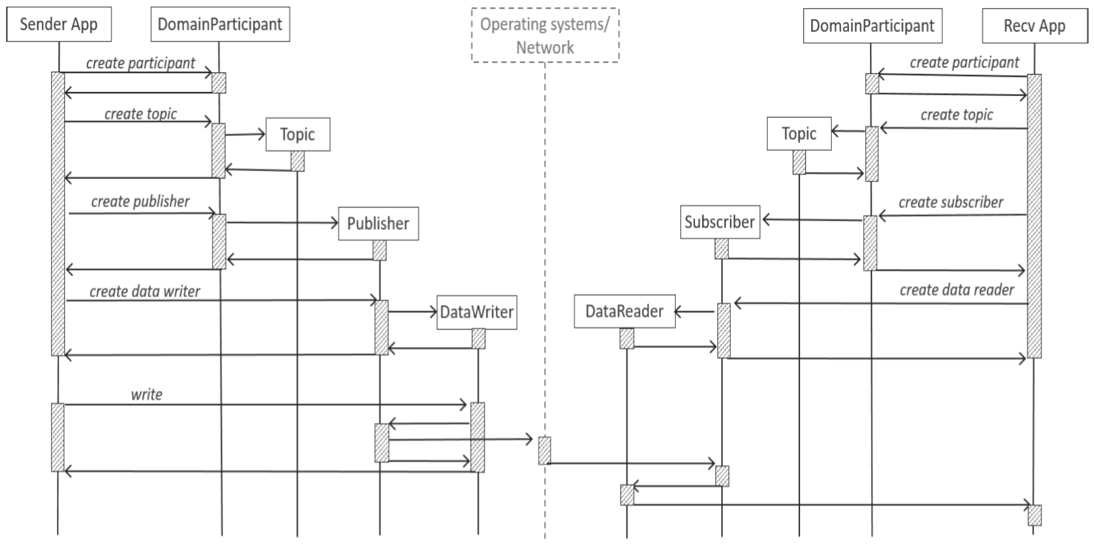


Figure 4.1: Simplified Sequence Diagram of DDS

transport protocol used by DDS. RTPS is specifically designed to meet the demands of data-distribution services. It plays a crucial role in making DDS interoperable across various vendors and additionally provides seamless plug-and-play connectivity for applications [14].

Some of the key features of DDS are,

- data-centric communication - DDS abstracts the complexity of low-level communication protocols (such as TCP/IP or UDP) and network details. It offers a high-level data-centric interface, allowing developers to focus on what data to send or receive, rather than how to send it. This abstraction enables more reliable and efficient data exchange and simplifies software development [16].
- Publish-subscribe model - In this model, data producers (publishers) are decoupled from data consumers (subscribers). Publishers produce data with a specific topic and distribute over a network, and Subscribers interested in that topic subscribes for the data. This allows Publishers and Subscribers to work independently, without direct knowledge of each other. This model is well suited for event driven and asynchronous communication, which promotes flexible and scalable architecture [16].
- Quality of Service (QoS) Configurability - DDS offers a range of Quality of Service (QoS) settings that are configurable to meet specific requirements. QoS parameters include settings for data reliability, delivery timeliness, resource usage, and more. This configurability allows the system to adapt to diverse demands and eventually improving data reliability and delivery.

- Real-time Capabilities - DDS guarantees that data is delivered within specified timeframes and that delivery is predictable and reliable. This determinism is vital for safety-critical applications. DDS is designed to meet real-time requirements by ensuring low-latency communication and deterministic data exchange [43].

Some of the features in the context of automotive applications, which are important requirements in enabling flexible vehicle architectures [44] are:

- Modularity and Reusability - DDS facilitates the creation of modular and reusable software components by abstracting data-sharing and communication features, enabling developers to focus on specific functions or services. This modularity allows for efficient system adaptation, reduced development time, and support for new features integration across various automotive applications.
- Interoperability and Standardization - DDS provides a standardized communication framework, simplifying communication between components and promoting interoperability. Its support for standardized wire protocols, software APIs, and QoS policies, reduces integration complexities and ensures interoperability in the diverse automotive domain. Some commercial DDS support, like RTI-Connext, supports DDS in Robot Operating System 2 (ROS2), AUTOSAR classic, and AUTOSAR adaptive platforms, ensuring seamless data exchange in automotive network [19].
- Flexibility and Scalability - DDS has significance in the automobile industry because of its data-centric and decoupling mechanism, which makes it easier to integrate new services or applications without significant system modifications. DDS's scalability ensures that it can handle the growing number of software components and data sources within vehicles.
- Security and Safety - Security is crucial in the automotive industry, especially for safety-critical functions. DDS offers security features like authentication and authorization, ensuring only authorized services can access critical data, protecting against unauthorized access. DDS standard APIs and wire protocol comply with ISO-26262 automotive functional safety standard [19].

To asses the applicability of the DDS middleware protocol in the automotive context, we compare its features with Scalable Service-Oriented MiddlewarE over IP (SOME/IP) protocol, which is designed specifically for automotive applications. SOME/IP protocol was developed specifically for Ethernet-based communication in the automotive industry. This standard defines the serialization mechanism, service discovery, and integration with the AUTOSAR stack [45]. In contrast, DDS is a full-fledged middleware protocol standardized by the OMG group. DDS is often used as a cross-domain connectivity framework, and finds application not only in aerospace, robotics, and industrial automation but also in automotive.

SOME/IP is a object-based service oriented communication, where a services are mapped to a specific ports on UDP/TCP and specified service IDs during design time. DDS, on

the other hand is more dynamic, and does not require an application to bind to particular service implementations. For example, a subscriber only subscribes to a particular topic, which means there is no static link to server. Service IDs are invoked during dynamic discovery, which is handled automatically by DDS [7].
SOME/IP does not define standard API and is commonly used as a part of AUTOSAR standard API. However, DDS offers standard APIs for multiple programming languages, including C, C++, and Java, enhancing its portability across diverse platforms. SOME/IP provides support for both UDP and TCP for data transmission, but relies on TCP as a fallback in case of reliable communication for larger payloads. In contrast, DDS uses the RTPS wire protocol, which is platform-independent and can be adapted to various network protocols. RTPS supports UDP, TCP and shared memory, which provides reliability and fragmentation capabilities, thus enabling large and reliable data transfer over (multicast) UDP [32].
SOME/IP generally relies on the underlying transport for security, often utilizing Transport Layer Security (TLS) or Datagram Transport Layer Security (DTLS) for secure communication. Instead, DDS defines an additional security standard, offering a more complete solution with finer control, access management, and transport-agnostic capabilities. This makes DDS suitable for deployment with various transport methods, including shared memory, multicast, and custom application-defined transports [7].

The most significant advantage of DDS is its support for wide range of QoS policies. SOME/IP provides a very limited QoS support relying on the capabilities of the underlying transport. DDS, on the other hand, offers a extensive array of QoS policies that are independent of transport method. These policies allow users to specify explicitly how the data is exchanged between publishers and subscribers. Further QoS policies cover various aspects, including the performance, transmission priority, data reliability, resource limits, persistence, latency/deadline monitoring, and many more. Refer [46] for the complete list of QoS policies and their implication.
DDS also introduces several standard built-in features, including content and time-based filtering of data on both publisher and subscriber side. This feature helps to exchange data under specific conditions with a required frequency. But, SOME/IP lacks content and time-based filtering, significantly impacting the automotive domain where data exchange occurs at various frequencies across different applications. As a result, DDS is prepared to can meet complex and flexible data flow requirements.

In Summary, the integration of DDS features into applications can significantly increase efficiency, reliability and real-time performance in the automotive industry. However, customization and modifications may be necessary to align DDS with the specific needs and constraints of the industry [47]. DDS has gained interest in the automotive sector, and its support has been included in adaptive AUTOSAR, and there are developments to incorporate it into AUTOSAR classic [32]. Furthermore, DDS is the middleware adopted in ROS2, which is again a widely used in automotive applications in recent years [48].

4.2 Context

The Adaptive AUTOSAR architecture is based on Service-Oriented Architecture (SOA). A service is defined using a Service Definition, defining events, methods, and fields as discussed in Section 2.4. The service is then modeled using a Service Interface, and both the Service Provider and Service Consumer implement this interface. A code generator utilizes the modeled Service Interface to create boilerplate code for the Skeleton (Service Provider) and Proxy (Service Consumer). The Server application uses the Skeleton to implement the interface and the Client application uses the Proxy as counterpart, effectively transferring communication responsibilities from applications to the ara::com middleware. The Adaptive applications can request data from within the same ECU (intra-domain) or from external components (inter-domain). This configuration can be done through a Service Instance deployment model during system integration, directing ara::com to select a protocol on which it routes traffic through the network protocol. Therefore on Adaptive AUTOSAR we have well defined structure for both intra-domain and inter-domain communication.
In the Android context, inter-process communication is handled by Binder. which is specifically designed for and within Android systems. Binder also provides Service-Oriented Architecture (SOA), but it is confined within Android device. AIDL is used to define a service interface and subsequent code generation is done for Stub (Service Provider) and Proxy (Service Consumer) parts as explained in Section 2.2.2. Apps then use the APIs provided by the generated code to exchange data.
Android adopts a different approach for inter-domain communication, which doesn't match to the service-oriented implementation in Adaptive domain. Google recommends the use of gRPC in Android for inter-domain communication, which is based on HTTP protocol [49]. However, in this context we focus on DDS middleware for inter-domain communication between Adaptive AUTOSAR and Android. Given that the Adaptive platform already incorporates DDS implementation, we focus mainly on integrating DDS into android domain.

4.3 Integration of DDS Middleware

In this section, we address the third research question [Q3] and explore potential approaches for cross-domain interaction between Adaptive AUTOSAR and Android Automotive platforms using DDS middleware. The main objective of this thesis is to establish connectivity without compromising functionalities in both domains. The adaptive platform can access data from the Android domain, such as Google navigation and in-car temperature, while Android Automotive can use engine-related values, tire pressure, and ADAS sensor data. This enhances the overall user experience by leveraging data from AUTOSAR.

The interconnection of Android Automotive and Adaptive AUTOSAR faces challenges due to their different architecture pattern and safety-critical features on AUTOSAR, such as ADAS and autonomous driving which require real-time and reliability data

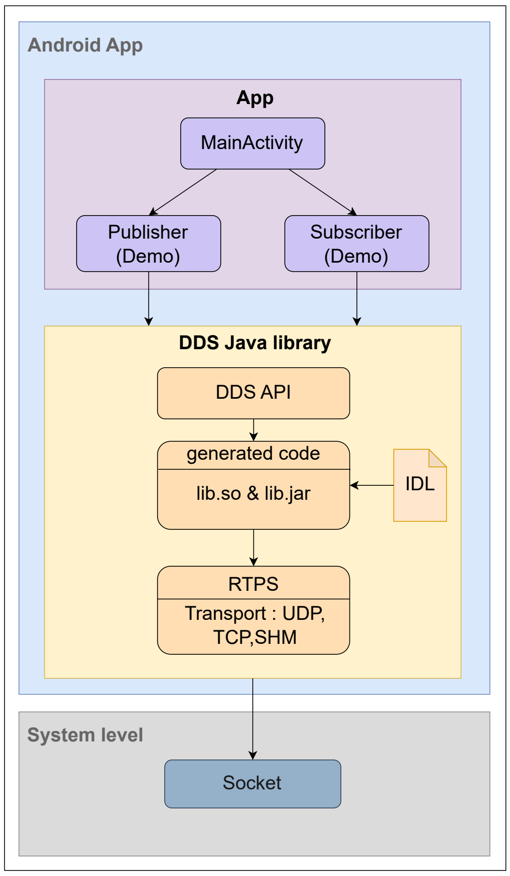


Figure 4.2: Integration of DDS library at application level

transmission. Android Automotive, on the other hand, may not guarantee real-time performance due to its design for general purpose nature and non-deterministic behaviour mainly focusing on better user experience rather than safety critical systems. However, data transfer from AUTOSAR to Android is technically feasible, but limited due to potential safety issues. This thesis, however, emphasizes the functional feasibility of data transfer between Android Automotive and AUTOSAR, rather than addressing safety and security concerns.

4.3.1 Approach 1: Application level

In this section, we explain the concept of integrating DDS middleware in Android at Application level. At this level, DDS library can be integrated as part of individual Android application. It involves incorporating DDS functionality within specific applications to enable data distribution and communication. Android follows a layered architecture where applications run in their own process. The integration of DDS at the application level allows for encapsulating DDS functionality within a specific application, ensuring non-interfere with other applications or the core system. Additionally, It provides more flexibility to choose different DDS implementations. Most of the Android applications are developed in Java or Kotlin, depending on DDS vendors we can get DDS for java applications. The DDS code generator allows us to utilize specific language binding for DDS library. The Java wrapper or API library uses Java Native Interface (JNI) to invoke corresponding C++ API calls. The integration of DDS at application level is illustrated in Figure 4.2. Here the Publisher and Subscriber generated by DDS API interact directly with the main Android UI. DDS library with Java bindings makes it easier for Android development.

We can configure the DDS build for specific language binding and IDL file, which defines the communication interface and also we can configure the transport layer which we want to use, since DDS allows pluggable transport layer.

Configuration options for DDS builds include specifying language bindings and defining the transport layer. Since Android targets multiple architectures and has various versions, we can define the target android API version at the build stage. DDS at it's core, is a C++ library. It uses Android NDK to build for Android systems. Android NDK is a tool-set that allows to cross-compile native C/C++ code for Android systems. As we buildDDS library for Java, the creation of a JNI interface, which acts as a bridge between Java and native C++, is not necessary. DDS library with Java wrapper creates JNI through which Java function calls invokes corresponding C++ API calls. This JNI interface allows us to access the C++ code from our Java Application. To achieve this, cross compile the DDS core libraries for the Android platform, by configuring at the build time. Then the generated shared library (.so files) and corresponding java library (.jar files) are used to build Android application. [50] discuses about porting RTPS middleware to Android using Android Software Development kit (SDK) and NDK. The system consists of mainly two modules, the App module and the DDS Java library module. Within the App module, there are three components, The MainActivity is used to display the UI to the user. And the two components, Publisher and Subscriber are the demo applications relevant for communication functionality utilizing DDS API. These components initialize the DDS middleware, creates DDS entities (DomainParticipant,Publishers,Subscribers), and handles data publication and subscription. [50] discusses about porting RTPS to Android and Android application for robot control. Additionally, there are few open-source DDS implementations that support cross-platform builds for Android applications [6].

Typically, Android uses Intents, method calls or Binder mechanism for communication. However, at application level DDS uses its own mechanism of data-centric to share data between participants using Topic. We need to ensure that the application has necessary network permission, since DDS relies on network communication to distribute data. Integrating at application level challenges interoperability between different applications and limits system-wide data sharing. It leads to multiple applications incorporating multiple DDS instances leading to data redundancy.

4.3.2 Approach 2: System level

In the previous approach, we discussed the implementation of DDS middleware as a part of Android application. However, we did not explain about the mapping of services in AUTOSAR (events, methods and fields) to DDS. In this section, we look into implementing DDS at system level and integrate the concept of service-oriented communication with AUTOSAR domain using DDS. DDS can be integrated into the Android system level, providing DDS functionality as a system-level service. This involves extending the Android system service to include DDS capabilities, allowing standardized DDS service that is readily available to all applications and system components and seamless data sharing between applications.

To establish connectivity between two different domains using DDS middleware,it is essential to examine the communication patterns in each architecture. As we discussed earlier, Adaptive AUTOSAR follows a service-oriented approach, defining services in the form of events, methods and fields. Whereas, DDS middleware is a data-centric primarily designed for publish/subscribe model. And Android follows server-client architecture for its IPC mechanism within the system. Therefore, it is necessary to map services from AUTOSAR domain to Android domain for seamless interaction. To map events, methods and fields to DDS concepts, we make use of mechanisms outlined in the OMG dds and RPC over dds specification. DDS RPC has service mechanism for remote method calls with pair of request topic and reply topic as discussed in Section 2.1.3. Events can be mapped to regular Topics in DDS, simply subscribing to the corresponding DDS Topics. Methods map to DDS Service provided by DDS RPC. Similarly, fields can be mapped to combination of DDS service and topic [51]. Fields like getter and setter methods are mapped to DDS service, and field notifications map to regular DDS Topics that provide current field value. DDS offers disticnt APIs for publish/subscribe model and for RPC. In Adaptive AUTOSAR ara::com specification, mapping of services in AUTOSAR domain to DDS services using DDS RPC is explained in section 7.5.3 [39]. In Android domain, IPC mechanism are handled by the Binder. Android system services are native services running at system level in the background, facilitating applications to access these service functionality using AIDL. Here native service acts as DDS DomainParticipants, enabling DDS functionality as Android system service.AIDL is then used for IPC communication between apps and the native service.

The implementation of methods and events in android are done differently. The method implementation is comparable to AIDL method call and it is more straight forward. The native service does the DDS method call using DDS RPC mechanism and returns the result obtained back to the app. However, handling events, and notifiers in Android services involves two AIDL methods [8]. The first AIDL method is called from the user application and its argument must be the Binder object of the second AIDL method interface, indicated in TODO:fig as 'subscriber(AidlInterfaceListener)'. This registers the interface of the second AIDL method in our Android service. The second AIDL method is invoked when an event or notifier is captured within the subscribed 'EventCallbackFunction(Data)' function from the DDS Participant. The AIDL method

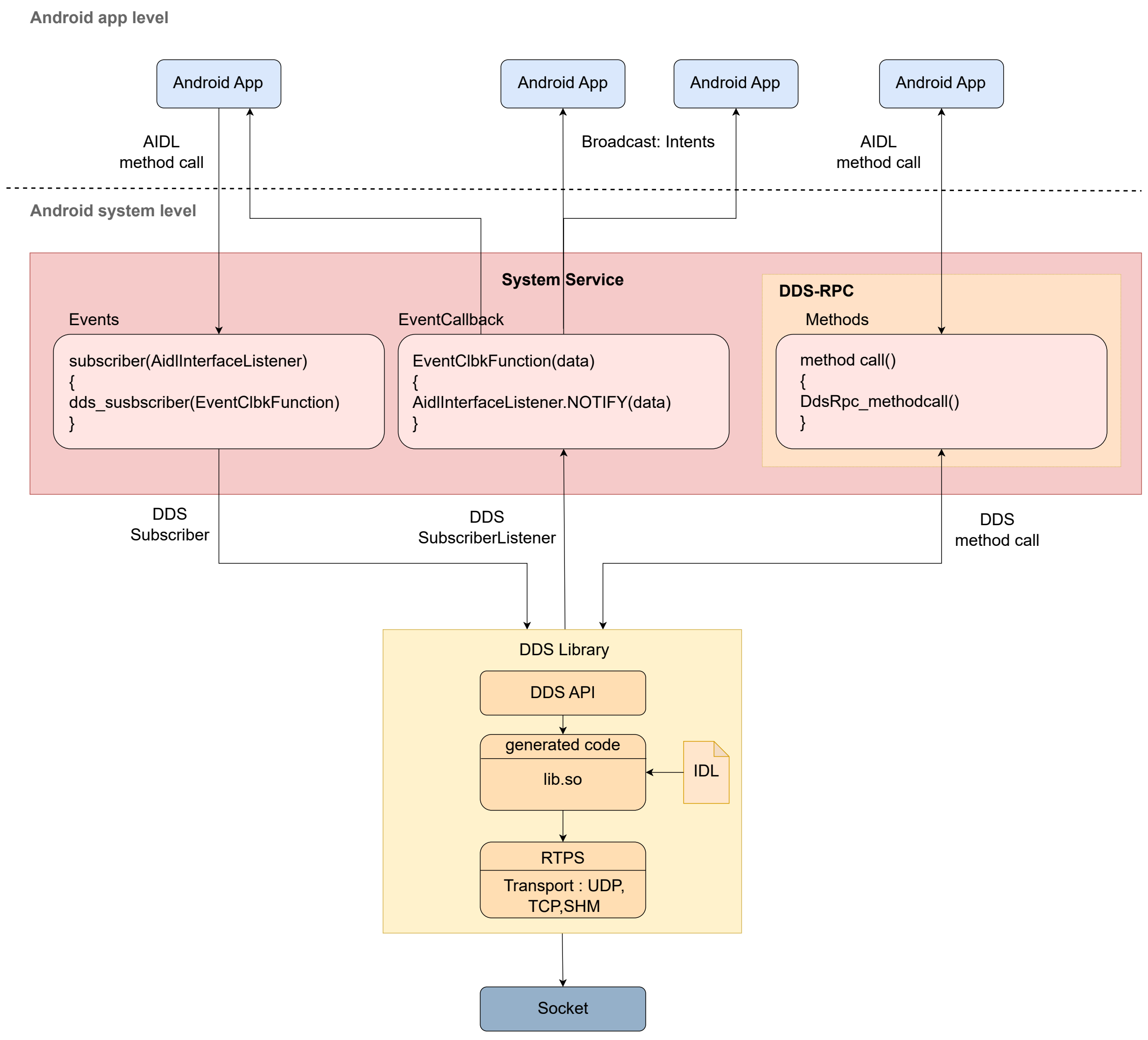


Figure 4.3: Integration of DDS library at system level

of the registered interface can be called during this process, depicted as 'AidlIFaceListener.NOTIFY(Data)' to notify the user application. A Java Helper service can be utilized to broadcast an intent to multiple apps. We need to modify the DDS API according to our requirements and support for both publish/subscribe and RPC communication mechanism.[8] proposes a solution with model-based code generation approach, to generate Stub and Proxy components in Android. This can be employed in the future to simplify the usage of DDS middleware for app developers.

4.3.3 Approach 3: Middleware level

In this section we discuss about the integration of DDS as a separate middleware layer within the android architecture. As discussed earlier in Section 2.3, Android Automotive is an extension of Android system architecture with additional components like VHAL, Car Service and Car Manager that are specific to automotive application. Similarly we can adopt or integrate DDS as a middleware alongside of other component in VHAL. This level of integration allows for a modular and scalable architecture, where DDS can be used in conjunction with other components to meet the specific communication requirements of the Android system. Unlike system level integration, DDS can be integrated in vendor service rather than Android system service, which will again be challenging for supporting android updates and future releases. Instead, by integrating as a separate middleware layer allows us to update our library independent of android versions. In this concept we propose implementing DDS in VHAL layer, which can be exposed to application level through service layer or directly using AIDL. Applications can use data from AUTOSAR/ADAS domain by simply using the HAL modules. In Android Automotive the VHAL layer must be implemented by the OEMs to fetch the data or car properties from an available vehicle bus (typically CAN bus). We can implement DDS middleware and its RTPS protocol similar to VHAL implementation. This allows to fetch data from the DDS layer. To achieve this, we need to modify VHAL to fetch data from DDS in vendor service and applications can then use this data by simply using the HAL modules in the higher layers. We can make use of DDS API with the modification to our requirements so that it won't be overhead in communication. Additionally, we can also combine with the previous approach and implement DDS functionalities in both Android system service as well as vendor service. This allows the service to communicate with both applications and the VHAL modules directly [8]. [10] also discusses the implementation of middleware services in vendor service and distributing data from ADAS to android apps. In this approach we implement DDS as part of VHAL as shown in Figure 4.4. This approach provides as integration into the existing Android Automotive Car API based architecture. The VHAL stores the data in the form of vehicle properties which can be used by app developers as part of car API to extract information about the car to use in their apps. Figure 4.5 shows the sequence of operations for the get operation from an Android app. vehicle properties can be vehicle speed, engine temperature, etc. When a getProperty() is invoked, VHAL module request for the value from DDS service.

Vendor specific apps can also access the DDS data using Java SDK, which provides

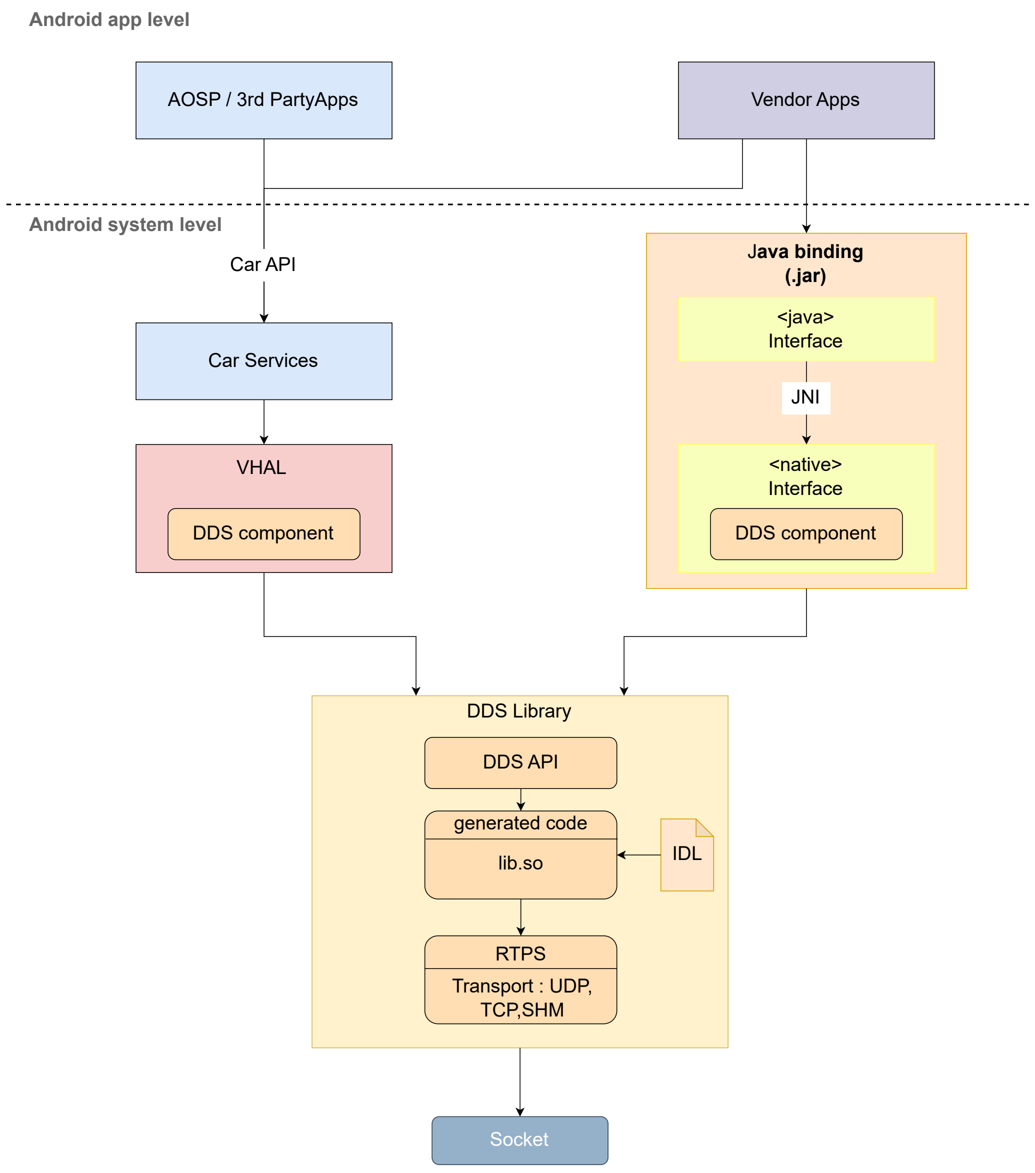


Figure 4.4: Integration of DDS library on Vehicle HAL

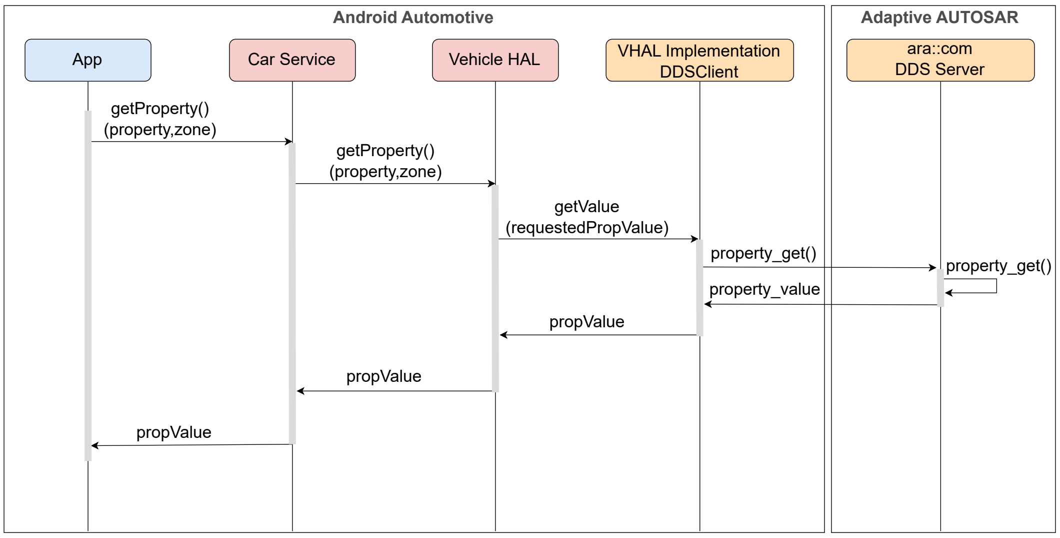


Figure 4.5: Sequence of get() operation from Android app

Java interfaces to Android vendor apps to communicate with their Adaptive platform counterparts.This allows OEMs to have access to privileged data from DDS to vendor specific apps. The DDS middleware would handle the data distribution aspects within the Android system, providing a standardized interface for applications to publish and subscribe to data. Integrating and managing an additional middleware layer requires additional development effort and expertise. Adding a middleware layer introduces additional complexity to the system architecture, requiring careful design and integration.

4.4 ISO 25010 based Analysis

4.4.1 Weighting of the Quality characteristics

All the quality criteria do not have same level of importance. A weightage of each quality criteria was considered based on the relevance and maturity of the solution.

Quality characteristics	**Sub-characteristics**	**Comments**	**Weight**

Compatibility	- Co-existence - Interoperability	The system should perform its necessary functions and be compatible with both the existing Android framework and future Android versions. Furthermore, it should be compatible with Adaptive AUTOSAR standards to exchange information.	5
Maintainability	- Modularity - Reusability - Modifiability - Testability	The system should be discreet, ensuring minimal impact on modifications for ease of maintenance with updates. It should also be modifiable without affecting functionality and easily testable.	5
Functional Suitability	- Functional completeness - Functional correctness - Functional appropriateness	The system should provide basic functions that meet stated requirements	4
Performance Efficiency	- Time behavior - Resource utilization - Capacity	The system should have good timing behavior and throughput rates with less resource utilization	4
Usability	- Operability - Learnability - User error protection	The system should be easily adaptable on evolving hardware or software and should be easy to install	3
Portability	- Adaptability - Installability	The system should be easy to operate. It should provide a good usability to the system integrator as well as the app developer with minimal effort.	3

Reliability	- Maturity - Availability - Fault tolerance - Recoverability	Although system should be reliable, functional safety is not considered to be relevant criteria.	3
Security	- Authentication - Confidentiality - Integrity - Accountability	The system is security relevant, but this is not the main priority initially.	2

Table 4.1: Weighting of the Quality Criteria

4.4.2 Analysis

Compatibility:		
Approach 1: Application level		Approach 3: Middleware level
+ DDS protocol remains compatible – DDS API should be modified to support DDS RPC required to communicate with AUTOSAR – On AAOS side, proper handling of data must be implemented for fields and methods, without which it would remain incompatible		+ However, if the ara::com component or modified DDS API is used, then compatibility issues could be resolved. – The DDS API would result in compatibility issues. The complete feature set needs to be aligned with AUTOSAR – From the System integrator perspective, the two models must be maintained. The services from Adaptive side should be properly mapped to DDS and DDS API (Including DDS RPC) to support request-response mechanism
Score	2	3
Weighted Score	10	15

Maintainability:	
Approach 1: Application level	Approach 3: Middleware level

+ Software updates are easier to support on app level and can be modified easily + Apps uses JNI APIs, which are mostly backward compatible, meaning the code written using an older version should continue to function properly with newer versions of the JDK + All components, including the DDS API and transport layer, are bundled within the app. Tools like Android Studio makes development, debugging, and testing processes easier – The creation of a new app necessitates the packaging of the DDS API and transport layer once again	+ However, if the ara::com component or modified DDS API is used, then compatibility issues could be resolved. – The DDS API would result in compatibility issues. The complete feature set needs to be aligned with AUTOSAR – From the System integrator perspective, the two models must be maintained. The services from Adaptive side should be properly mapped to DDS and DDS API (Including DDS RPC) to support request-response mechanism
Score 4	3
Weighted Score 20	15

Functional Suitability:	
Approach 1: Application level	Approach 3: Middleware level
– The app has limited access to SOA functionality, and developers need to implement it specifically within the app – The application is confined to certain system-level permissions, potentially affecting the functionality	+ The complete SOA functionality to support communication with AUTOSAR can be implemented and directly available to apps + The customized DDS API to support service-oriented communication is more suitable – There might be unused features of DDS API
Score 3	4
Weighted Score 12	16

Performance Efficiency:	
Approach 1: Application level	Approach 3: Middleware level

+ Startup time is minimized as all the components are integrated within the apps + The run-time behavior is faster due to the absence of any middleware layers, allowing direct communication from the app using the protocol layer + Less resources are required because there are no middleware layers and less boilerplate code − Multiple apps each with its own DDS implementation could lead to overhead	+ Multiple apps can access the DDS Service at the system level − During startup phase, system components are initiated earlier, followed by app components. This implies that overall time taken for the entire startup process would be extended − Runtime: Since there are multiple layers ($App \rightarrow AIDL \rightarrow SystemService \rightarrow DDSAPI \rightarrow RTPStransport$), the data must traverse through all the layers resulting in higher timing − RAM usage will be higher due to presence of separate middleware service
Score 4 Weighted Score 16	3 12

Usability:	
Approach 1: Application level	Approach 3: Middleware level
+ System integrator is not necessary, and all the responsibilities are transferred to the App developer. It requires expertise in Android app development − Expertise knowledge in service-oriented communication and DDS middleware	+ From an app developer's standpoint, operability is simplified because of the availability of readily accessible APIs − System complexity and needs careful design and integration − Expertise in Android architecture and system level knowledge
Score 3 Weighted Score 12	2 8

Portability:	
Approach 1: Application level	Approach 3: Middleware level

+ Developing and adapting an Android app to different API versions is made easy by specifying the minimum API and target versions during the build process in Android Studio + The installation of the APK build file on Android system is straightforward − Expertise knowledge in service-oriented communication and DDS middleware	− At system level we need to comply with evolving Android standards. − Building an AOSP project and debugging errors during the build process can be challenging
Score 3	2
Weighted Score 9	6

Reliability:	
Approach 1: Application level	Approach 3: Middleware level
+ The failure of an app component does not impact the functionality of the Android OS, making the recovery from such failures easier + The possibility of errors is lesser due to simple app level implementation	− The failure of a system component can impact functionality and may be challenging to recover, often requiring a restart of the entire operating system − A larger code stack and dependencies may introduce more sources of errors
Score 3	2
Weighted Score 9	6

Security:	
Approach 1: Application level	Approach 3: Middleware level

+ Android apps make use of the built-in security features of the Android operating system, such as process isolation, sandboxing, and permission-based access controls − An authenticating mechanism for an app that can communicate over a network does not exist within the Android system, instead must use DDS's inbuilt security feature.	+ An App component's authenticity can be verified by the System Service component before allowing it to communicate over the network − A system component having privileged authentication may result in data manipulation of other components
Score 3	2
Weighted Score 6	4
Overall Score 25	21
Overall Weighted Score 94	82

Table 4.2: ISO 25010 Analysis

In summary, both the approaches have their advantages and disadvantages. It is important to analyze our specific requirements, system architecture, use cases and trade-offs to determine the most suitable approach. Application-level integration offers flexibility and simplicity, while system-level integration is initially more complex but provides a standardized approach for easier future development.

Chapter 5

Implementation

This chapter explains the implementation of our solution. Approach 1 explained in Section 4.3 is implemented. We focus on a publish-subscribe model of the standard DDS API in this solution. OpenDDS is utilized in our implementation, which is open-source and offers build support for Android systems as well as Java language bindings. This helps with Android application development. DDS RPC will not be part of the implementation, since currently there are no open-source DDS API libraries for RPC that provides build support for Android. Nevertheless, we also demonstrate a simple method using publish-subscribe model later in this chapter. While other approaches such as system-level integration can also be implemented, but that would require proper designing of API and requires larger efforts. The app level approach was implemented as a proof of concept to demonstrate the feasibility of our solution. The first section of this chapter discusses the design modifications made to our solution. We then give an overview of the DDS application, covering about the domain participants creation, followed by the configuration and build process of openDDS for Android architecture. Later, the implementation of Android DemoAPP is explained, and finally development of openDDS application in a Linux environment.

5.1 Design Modification

In the previous sections, we explained various approaches and also their architectural impact on various characteristics according to the ISO 25010 standard. We are making some decisions and modifications to our approaches to implement the solution.

Despite the advantages of the system-level approach, which offers a standardized architecture for future development and integration, we decided not to adopt this approach due to various reasons. Different vendors offer DDS middleware solutions that support various Operating System and offer capabilities with different language bindings. We used an open-source DDS in our thesis implementation. Currently, Android does not rely on a single, but rather a combination of multiple stages of building and file gener-

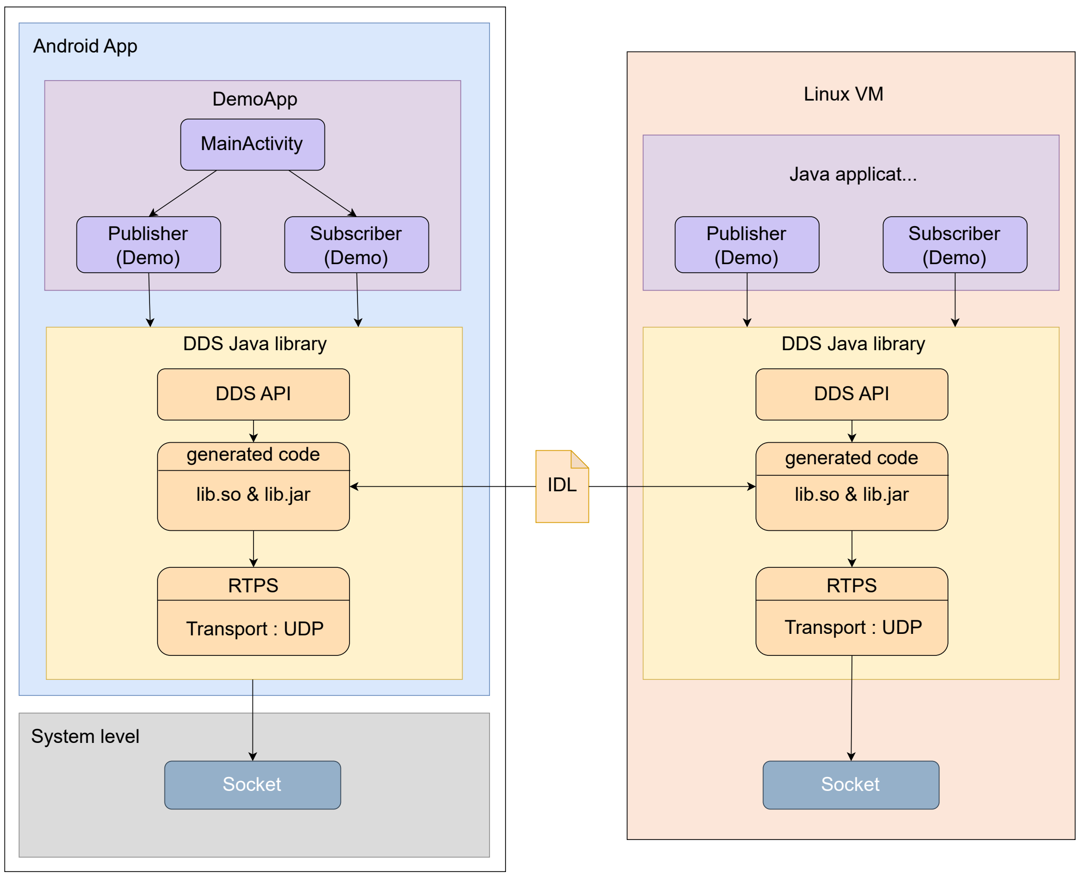


Figure 5.1: Modified design

ation. Android consists of modules, a component of AOSP that can be independently built and compiled. At its core, Android relies on the Ninja build system. However, we won't get into the details of the Android build system. Integrating the DDS library at the Android system level would require configuring and modifying DDS library and building with the Android build system, requiring larger efforts and in-depth knowledge of both DDS and the Android system.

And the middleware-level approach also provides a better integration for automotive applications. Integrating DDS as part of VHAL implementations allows easy access to current HAL modules in AAOS and vendor-specific applications using the Java SDK library by defining interfaces. However, as previously mentioned, creating a module at the system level involves significant development effort in addition to in-depth understanding of both systems. The OEMs are responsible for developing the VHAL implementation so that it interacts with kernel modules and device drivers sufficiently. An actual VHAL implementation would require additional development efforts and proprietary source code from chip vendors based on the underlying hardware that is used. As a result, the implementation of DDS at the VHAL layer, with Car API integration is excluded.

To develop and implement the proof of concept we follow the application level approach described in Section 4.3.1, where we implement the DDS library at application level with Java bindings. DDS with JNI bindings enables the interaction with the C++ based DDS libraries. As a result, we don't need to focus on the Android build system, instead, we cross-compile the required DDS libraries for the Android platform and use them for our application development.

The initial idea was to run the demo application in AAOS and a remote application in an Adaptive AUTOSAR environment. The fundamental concept was to establish communication between Android Automotive and Adaptive AUTOSAR using DDS middleware. However, developing applications in the Adaptive environment would demand additional efforts, as it requires configuring the entire machine, setting up service interfaces, and subsequently developing and deploying the application. Additionally, this would also require access to a proprietary source code of an Adaptive AUTOSAR stack. Since the development in Adaptive Platform is not a primary focus of the thesis, the Adaptive AUTOSAR part will not be presented and implemented.

The design modification we implemented in this thesis shows the communication between two applications running on separate machines. The Android Automotive OS on a Raspberry Pi provides the demo application and the Linux virtual machine executes the remote application. The design modification involves the implementation of a remote DDS application on Linux machines, utilizing the standard DDS API, thereby simplifying the demonstration process. Adaptive AUTOSAR incorporates DDS as its network binding within ara::com module. Due to DDS's wire protocol RTPS, it is possible to interact with any other DDS systems. So, the implementation of Adaptive application with DDS as network binding is similar to standard DDS API with few modifications to support client-server communication patterns with service interfaces between applications. Figure 5.1 shows the modifications to our approach.

There are two modules in the Android app. One is the App module which has the Main Activity for UI related tasks, and Demo publisher and subscriber application. These publisher and subscriber applications interact with DDS API to publish and subscribe the data from a remote application.

5.2 DDS Application

In this section, we explain the simple DDS application with the publishing and subscribing process. Given the various DDS vendors available in the market, we chose openDDS for our implementation since it is an open-source that offers Java bindings for the DDS library and also provides support for Android. We explain the code in subsections for better understanding.

5.2.1 Initializing the Participant

The first part of DDS application is initializing and creating the Domain Participant. DDS applications begin initialization with the initial reference to the Participant Factory. The Participant Factory is a fundamental element that manages the creation and configuration of Domain Participants within the DDS system. *TheParticipantFactory.WithArgs()* returns a Factory reference, we can then create Participants for specific domains. To configure the application, a configuration file must be provided as an argument, allowing developers to define and configure various parameters related to the transport protocol, service discovery, and other settings. DDS API offers flexibility in selecting the transport protocol like TCP, UDP or shared memory. The DomianParticipant is created using the *create_participant()*, where we can pass DomaiID as a parameter, which is unique and separates different domains in the network.

```java
public static void main(String[] args) {

    //Get DomainParticipant Factory
    DomainParticipantFactory dpf =
        TheParticipantFactory.WithArgs(new StringSeqHolder(args));
    if (dpf == null) {
      System.err.println ("Domain Participant Factory not found");
      return;
    }
    //Domain ID
    final int DOMAIN_ID = 4;

    //Create DomainParticipant
    DomainParticipant dp = dpf.create_participant(DOMAIN_ID,
      PARTICIPANT_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
    if (dp == null) {
      System.err.println ("Domain Participant creation failed");
      return;
    }
```

5.2.2 Registering the Data Type and Creating a Topic

In the next step, we register our data type with the DomainParticipant, which defines the structure of the data and data type to be exchanged between Publisher and Subscriber. We need to specify the data structure by defining a data type in IDL file, this data type is then registered with the DDS middleware. *register_type()* operation is used to create a data type, we can explicitly specify a type name or pass an empty string as shown here. If an empty string is passed, then the middleware uses the identifier generated by the IDL compile for the type. The process involves defining the data type in an IDL file and then registering this data type with the DomainParticipant. In our implementation, we have created an IDL file for our demo application, which will be explained later in the next sections. In this IDL file we have defined a structure *PubMessage* with one string type and three long types.

```
//Create Datatype, type support
PubMessageTypeSupportImpl ts = new PubMessageTypeSupportImpl();
if (ts.register_type(dp, "") != RETCODE_OK.value) {
    System.err.println("ERROR: register_type failed");
    return;
}
```

Now, we create a topic using the type support. The topic name is 'Topic', with the registered data type *PubMessage* and default QoS policy is created and associated with the DomainParticipant (dp). While defining data types in IDL file we annotate with `@topic`, which defines the data type for creating topic as explained in Section 2.1.1.

```

//Create Topic
Topic top = dp.create_topic("Topic",
                            ts.get_type_name(),
                            TOPIC_QOS_DEFAULT.get(), null,
                            DEFAULT_STATUS_MASK.value);


if (top == null) {
      System.err.println("ERROR: Topic creation failed");
      return;
    }
```

5.2.3 Creating Publisher and DataWriter

The Publisher and DatWriter are responsible for publishing the data to DDS domain. We create Publisher using *create_publisher()* operation associated with the DomainParticipant. In our demo application, we create Publisher *pub* with default QoS. As we can see, we pass null as an argument, typically refers to a listener that can be attached to the Publisher for monitoring various events.

```
//Create Publisher
Publisher pub = dp.create_publisher(
  PUBLISHER_QOS_DEFAULT.get(),
  null,
  DEFAULT_STATUS_MASK.value);

if (pub == null) {
      System.err.println("ERROR: Publisher creation failed");
      return;
    }
```

Following the creation of the publisher, we create DataWriter. The DataWriter is for a specific topic. In our demo application, we use the default QoS policies and a null DataWriterListener. While it is also possible to specify the QoS like Reliability, and durability for DataWriter, but the DataReaders must also match and be compatible with the corresponding DataWriter's QoS policy. Next, we should narrow the generic DataWriter to a type-specific DataWriter and register the instance we want to publish. As explained earlier in Section 2.1.1, the use of the `@key` annotation allows us to specify the instance. In our demo application, we did not set the key. If the key field is specified then the respective data should be specified. The example code for a simple Java application can be found in java/tests/messenger in [52]. After narrowing down the DatWriter, we can load the data to the message and *write()* operation is used to write to specific DataWriter with message data and handle instance as arguments.

```
//create DataWriter specific to topic
DataWriter dw = pub.create_datawriter(
  top, DATAWRITER_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);

PubMessageDataWriter mdw = PubMessageDataWriterHelper.narrow(dw);

//create message
PubMessage msg= new PubMessage();
msg.text= "Hello";
```

```
//write message to DataWriter
int ret = mdw.write(msg, HANDLE_NIL.value);

/** To register instace**/
// @key annotated in IDl
// msg.<@keyfield> = value;
// int handle = mdw.register(msg);
// int ret = mwd.write(msg, handle);
/** To register instance**/
```

5.2.4 Creating Subscriber and DataReader

In this section, we explain the Subscriber part of the DDS application. We create a Subscriber similar to a Publisher associated with DomainParticipant(dp). Since we do not use SubscriberListener, we pass null and we use default QoS policy. Status mask can be used to check the status condition, but it's not in the scope of this thesis.

```

//Create Subscriber
Subscriber sub = dp.create_subscriber(
  SUBSCRIBER_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
```

Once the Subscriber is created, we create DataReader to subscribe the data. Within DataReader, we implement a DataReaderListener, a component responsible for notifying the middleware upon reception of data and providing access to the received data. To achieve this, an instance of a Listener is created using *DataReaderListnerImpl()*. This is then as a parameter to create DataReader.

```

//Create Listener for DataReader
DataReaderListenerImpl listener = new DataReaderListenerImpl();

//Create DataReader with specific Topic
DataReader dr = sub.create_datareader(
   top, DATAREADER_QOS_DEFAULT.get(), listener,
   DEFAULT_STATUS_MASK.value);
```

The incoming messages will be received by the Listener in the middleware's thread. when new data arrives Listener's methods, such as 'on_data_available()', 'on_subscription_matched()', etc., are invoked within the context of the middleware's thread. Meanwhile, the application thread performs other tasks.

DataReaderListener class is implemented by extending 'DDS._DataReaderListenerLocalBase', which is an abstract class provided by the DDS library. This abstract class serves as a base for creating custom DataReaderListener in our application. In our application, we only use on_data_available method callback from the middleware.

```
public class DataReaderListenerImpl extends DDS._DataReaderListenerLocalBase {

    //on_data_available callback
    public synchronized void on_data_available(DDS.DataReader reader) {

        //narrow down the generic DataReader to type specific
        PubMessageDataReader mdr = PubMessageDataReaderHelper.narrow(reader);
        if (mdr == null) {
          System.err.println ("read: narrow failed.");
          return;
        }
```

Further, we create a MessageHolder object for actual Message and SampleInfoHolder to manage the incoming messages. SampleInfo includes meta information about the message such as the message validity, instance state, source timestamp, etc. Subsequently, we take the next sample, which is then removed from the DataReader's available sample pool.

```
//Message Holder
PubMessageHolder mh = new PubMessageHolder(new PubMessage());
//SampleInfo Holder
SampleInfoHolder sih = new SampleInfoHolder(new SampleInfo());
//source timestamp
sih.value.source_timestamp = new DDS.Time_t();
//take next sample
int status  = mdr.take_next_sample(mh, sih);
```

After successfully retrieving the next sample, if the data is valid in SampleInfoHolder, then it prints the details from the PubMessage. Depending on the instance state, error messages are printed. If status is false, it indicates that *take_next_sample()* resulted in an error.

```
if (status == RETCODE_OK.value) {
    if (sih.value.valid_data) {
    //print the message details
        System.out.println("Received: " + mh.value.text + " " + mh.value.num1);
    }
} else if (status == RETCODE_NO_DATA.value) {
    System.err.println ("ERROR: reader received DDS::RETCODE_NO_DATA!");

} else {

    System.err.println("ERROR: take_next_sample_: " + status);

}
```

5.3 Configuration

The essential library for the project is openDDS library. The first step is to clone the openDDS master repository [52]. We used the openDDS 3.26 release for this project. After cloning, the next steps include configuring the build and building for the Android target. OpenDDS developer's guide provides the details of the configuration steps to build on the Android architecture. In our case, we built openDDS for Android by utilizing the Android NDK, which is bundled with Android Studio. It is necessary to specify the minimum API version and target API version for Android during the configuration process. For this particular project, the configuration of the build is illustrated as follows:

```
OpenDDS configure script (in root directory):
./configure --doc-group3 --target=android --macros=android_abi=arm64-v8a --macros=
    android_api=28 --macros=android_ndk=/home/user/Android/Sdk/ndk/25.1.8937393 --
    macros=android_sdk=/home/user/Android/Sdk --macros=android_target_api=32 --java
```

We built openDDS specifically for Raspberry Pi running on the arm64-v8a architecture. Additionally, openDDS provides the option to build with Java bindings during configuration. OpenDDS library was cross-compiled on a Ubuntu 22.04 LTS virtual machine. Once the openDDS is cross-compiled, it generates two copies of openDDS, one for the Android target in 'build/target' directory and 'build/host' directory. The host directory holds the static library built for the host platform and used to build the target directory. All the Java examples can be found in 'java/tests' directory, which serves as the basis for our project.

5.4 OpenDDS in Android application

In this section, we explain the cross-compiling IDL libraries and developing the Android application.

5.4.1 Cross-Compiling IDL Libraries

After cross-compiling the openDDS library for the Android target, all the required libraries can be found in '/lib' directory. This directory also contains the required jar files for the Java application development. The initial step for openDDS Android application development is defining IDL file. In this IDL file we define the structure and the data types that will be used to communicate between the publisher and the subscriber. For our demo application, we defined IDL file as follows:

```
//module name
module Demo {

  @topic // type support for simple publish-susbscribe
  struct PubMessage {
    string text;
    long num1;
    long num2;
    long num3;
  };

  @topic // type support for method call
  struct PubMessage1 {
    string text;
    long num1;
    long num2;
    long num3;
  };


  @topic // type support for acknowledge message
  struct AckMessage {
    long num;
    string text;
  };
};
```

For our Android application, we created a Demo.idl file with the module name 'Demo'. Within this module, we define three data types or type support PubMessage, PubMessage1, and AckMessage. PubMessage consists of one string and three long values and is used for simple publish-subscribe application. And the other PubMessage1 is similar to PubMessage, used to demonstrate the method call using two topics- 'RequestTopic' and 'ReplyTopic', which will be discussed in later sections.

OpenDDS uses The Make, Project, and Workspace Creator (MPC) build system. Next, we need to define Demo.mpc project file to cross-compile the IDL file created earlier. In this file, Java support for IDL can be incorporated by inheriting 'dcps_java' (assuming that the OpenDDS library was initially built with Java). The necessary examples for IDL and code generation for java IDL can be found in the '/java' sub-directory. The inclusion of Java support for IDL in the Demo.mpc project ensures that it is built alongside the native IDL libraries.

```
source $DDS_ROOT/setenv.sh

opendds_mwc.pl && PATH=$PATH:$TOOLCHAIN/bin make
```

once the IDL MPC project is ready, it can be cross-compiled using the above command. It is crucial to set the proper environment variables before building the project. All the necessary variables are configured using 'setenv.sh' script. During cross-compiling the IDL, we need to specify the compiler path for the target architecture. Given that 'build/target' has already been built for the Android architecture, 'setenv.sh' script sets all the required environment variables. If we would like to cross-compile independently, then we have to make sure the appropriate environment variables are set. After the compilation of java IDL, two components are generated - a Java .jar file and a supporting native shared library .so file. These files are then included in the Android application development.

5.4.2 Android App

In this section, we discuss the development of an Android demo application. To make use of the openDDS library in an Android application, it is essential to include all the dependent native libraries and the java .jar files of the app's build.gradle file. In Android.Manifest.xml file, it is necessary to add network permissions for DDS to access WiFi. See [53] in the openDDS developer's guide for detailed instructions on integrating openDDS libraries into an Android app. OpenDDS uses a configuration file, where the transport protocol and openDDS service-discovery related configurations are defined. This configuration file is included under resource or assets files so that the Android app can access the file stream.

Figure 5.2 shows the detailed class diagram of the demo application. Within the App module, the components include LoginActivity, MainActivity, and DemoApplication. Each of these components runs in separate threads. OpenDDS is implemented in DemoApplication, LoginActivity, and Mainactivity constituting the user interface and user interaction that runs in a separate thread.

In Android development, an Activity serves as the entry point for user interaction in an application. Every activity is associated with a layout, which represents the UI for interacting with the user. This layout is typically designed using XML file format. in

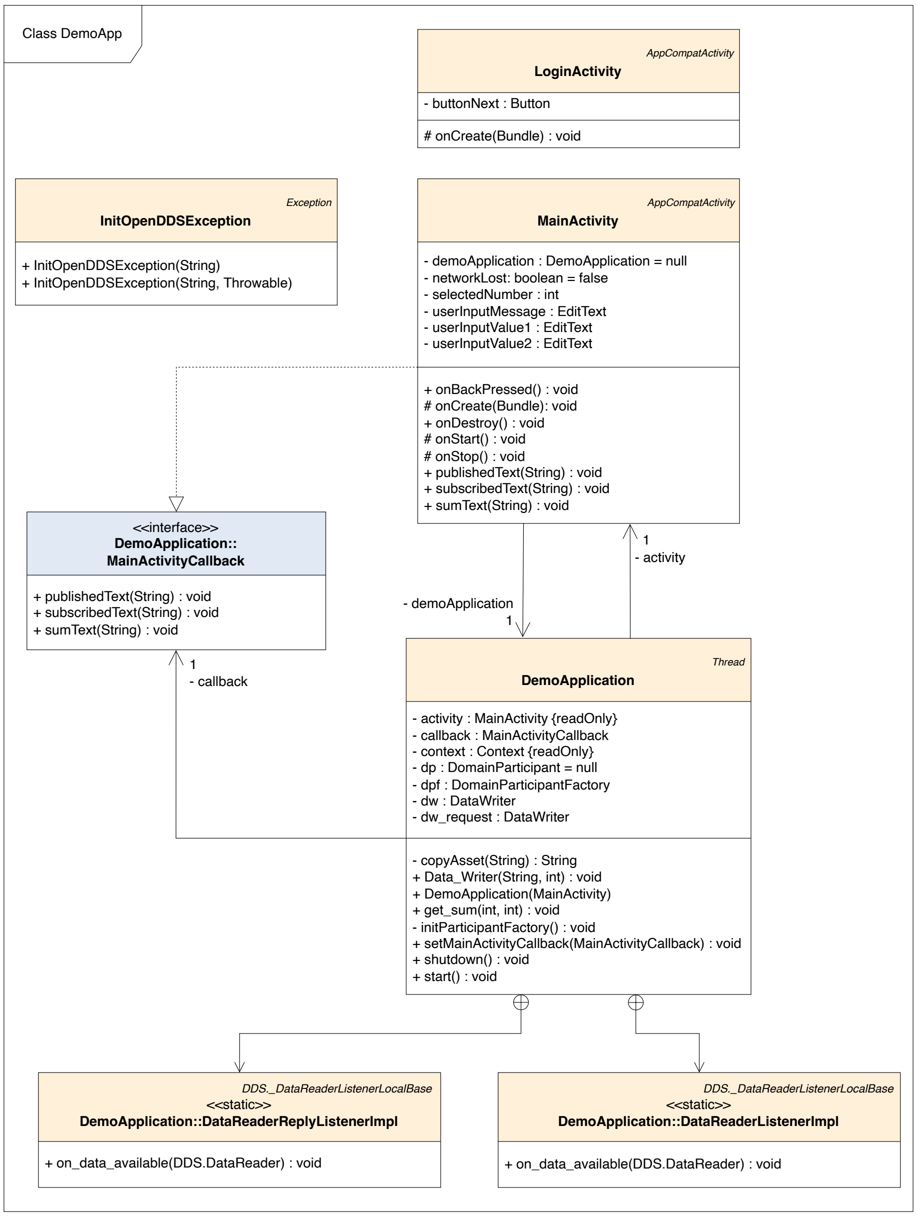

Class DemoApp
AppCompatActivity
LoginActivity
- buttonNext : Button
# onCreate(Bundle) : void
Exception
InitOpenDDSException
+ InitOpenDDSException(String)
+ InitOpenDDSException(String, Throwable)
AppCompatActivity
MainActivity
- demoApplication : DemoApplication = null
- networkLost: boolean = false
- selectedNumber : int
- userInputMessage : EditText
- userInputValue1 : EditText
- userInputValue2 : EditText
+ onBackPressed() : void
# onCreate(Bundle): void
+ onDestroy() : void
# onStart() : void
# onStop() : void
+ publishedText(String) : void
+ subscribedText(String) : void
+ sumText(String) : void
<<interface>>
DemoApplication::
MainActivityCallback
+ publishedText(String) : void
+ subscribedText(String) : void
+ sumText(String) : void
1
- activity
- demoApplication
1
1
- callback
Thread
DemoApplication
- activity : MainActivity {readOnly}
- callback : MainActivityCallback
- context : Context {readOnly}
- dp : DomainParticipant = null
- dpf : DomainParticipantFactory
- dw : DataWriter
- dw_request : DataWriter
- copyAsset(String) : String
+ Data_Writer(String, int) : void
+ DemoApplication(MainActivity)
+ get_sum(int, int) : void
- initParticipantFactory() : void
+ setMainActivityCallback(MainActivityCallback) : void
+ shutdown() : void
+ start() : void
DDS._DataReaderListenerLocalBase
<<static>>
DemoApplication::DataReaderReplyListenerImpl
+ on_data_available(DDS.DataReader) : void
DDS._DataReaderListenerLocalBase
<<static>>
DemoApplication::DataReaderListenerImpl
+ on_data_available(DDS.DataReader) : void

Figure 5.2: Class diagram of the DemoApp

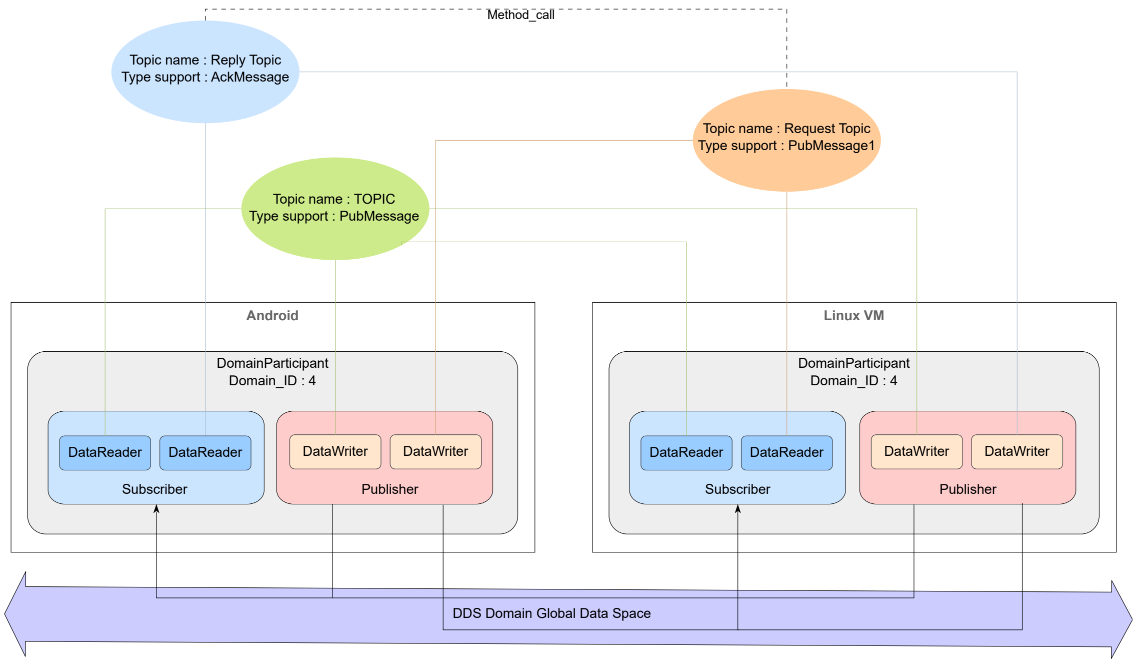


Figure 5.3: DDS domain diagram of the application

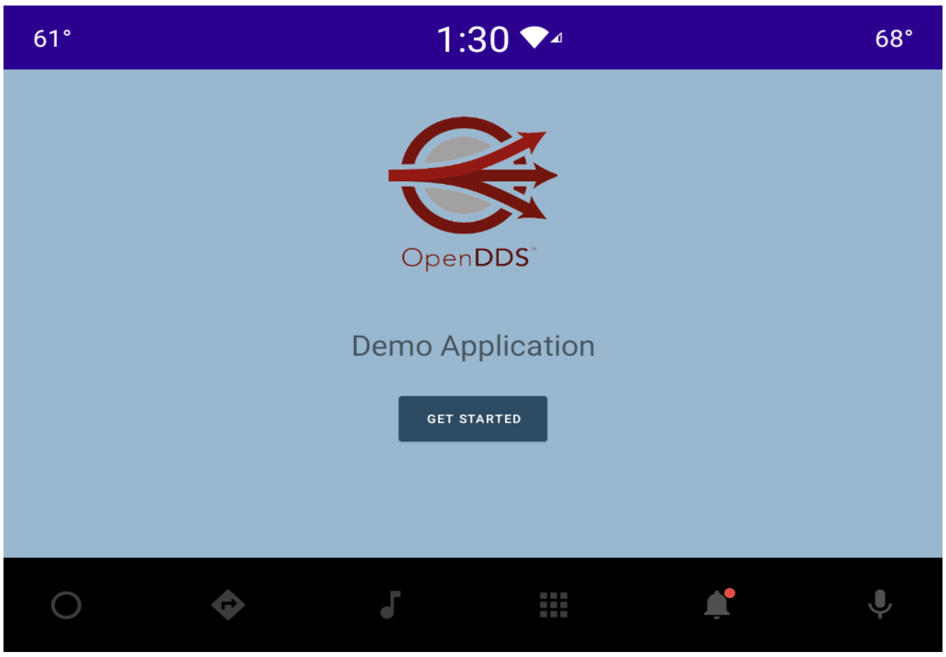


Figure 5.4: DemoApp login screen

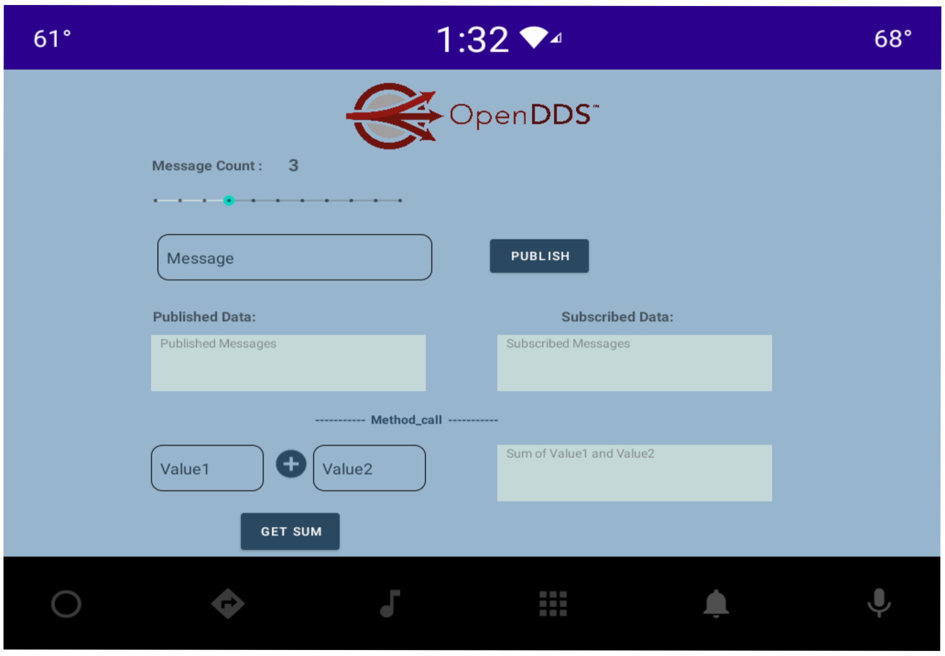


Figure 5.5: DemoApp main UI screen

our app, the LoginActivity is the initial activity that starts up when the application is launched. The LoginActivity is the first screen or interface a user sees when the app opens. On startup, as shown in Figure 5.4, the user has the option to initiate the demo application by clicking 'Get Started' button. This triggers the MainActivity class, where an instance of DemoApplication is created and OpenDDS participants are initialized.

Figure 5.5 shows the main UI screen of the App. The upper part allows the user to input a string or message intended for publishing on DDS domain. Additionally, the user can also specify the message count using seek bar to determine the number of messages to publish. Once the user inputs the message and message count, the message gets published by clicking the 'Publish button'. In the DemoApplication class, the configuration files are read and all the DomainParticipants are initialized accordingly. Upon clicking 'Publish' button, the Data_writer operation is invoked. this operation takes the user input and writes them to the DataWriter. During OpenDDS initialization, the DataReaderListnerImpl class is initialized for DataReader. If data is available, the listener notifies the middleware. OpenDDS has a separate worker thread for handling these listener callbacks. However, once the data is available these callbacks can't directly make changes to the Android UI as it's not the main thread. To achieve this, the MainActivityCallback interface is utilized. When the data is received in the listener class, the MainActivityCallback is invoked to notify the user about the subscribed data on UI text view.

This thesis focused on using the standard open-source DDS API from openDDS to integrate the middleware into the Android architecture. Since there are no open-source DDS RPC implementations on Android at the moment, we decided not to implement the DDS RPC. But as a workaround, we implemented a simple method call using two pairs of Topics, one for the Request topic and another for the reply topic. Since RPC requires a synchronization process, we need a modified DDS API for server-client communication with Adaptive AUTOSAR counterpart.

The lower section of the main user screen as shown in Figure 5.5, illustrates a simple method call for calculating the sum of two values. To facilitate this method call process, we use the PubMessage1 data type which was defined in Demo.idl file. AckMessage data type is used as type support for the reply of a method call. Users can input integer values in the text box and click 'Get Sum' button. This triggers the 'get_sum' operation in the DemoApplication, which writes the input values to a new DataWriter associated with the 'RequestTopic' similar to explained earlier. A separate DataReader is created with 'RequestTopic' and a listener 'DataReaderReplyListenerImpl' class, which notifies when data is received. The server-side implementation of this method call is executed in a virtual Linux machine. The DDS domain diagram for our implementation is as depicted in Figure 5.3.

5.5 Linux Application

As motioned earlier in the design modification, we are not implementing the Adaptive AUTOSAR application. Instead, we are implementing an openDDS application in a Linux virtual machine. In this section, we discuss how to build an openDDS library and implement an openDDS Java application to communicate with the Android application implemented in the previous section.

The first is to clone the master branch of openDDS git repository [52]. Following, we configure the project for Java bindings using './configure –java', and then 'make -j$(nproc)' to build the openDDS Java library. Once the library is successfully built, all the shared library files (.so) and .jar files will be in '/lib' subdirectory. The next step is to define an IDL file and a MPC project file to compile the IDL file and generate the required code for application development. We use the same IDL file used before in the Android application. In the project MPC file, we include 'opendds_java' project to build IDL libraries for the Java application. A simple example of a Message.idl file and the process of building a simple Java application is outlined in [54].

Once the IDL libraries and .jar files are generated, we include them in our application. We create DemoPublisher class, with participant initialization, Topic and Publisher creation, and finally, DataWriter creation as discussed in Section 5.2. Similarly, we establish a DemoSubscriber class with a DataReaderListener class. After implementing these Java classes we proceed to build them using openJDK, specifically using Java 11 for our project. It is crucial to ensure that the DomainID and Topic name matches with the Android application. Additionally, it is also necessary to match the DataWriter and

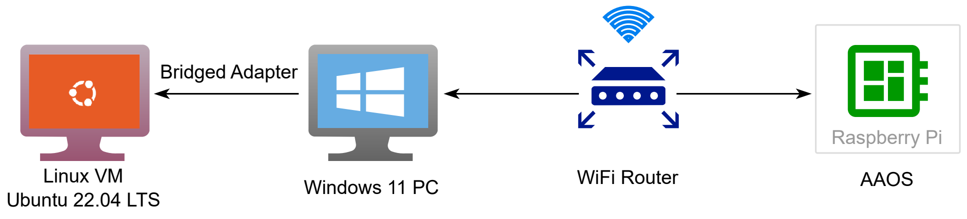


Figure 5.6: Hardware setup

DataReader QoS. Most often the default QoS is used. Although we can modify them according to the application's requirements and resource availability.

5.6 Hardware Setup

Figure 5.6 depicts the hardware setup used for our project. The Windows PC serves as the Android app's development environment and is then used to deploy the app to the AAOS running on a Raspberry Pi 4 model B with 2GB RAM. The AAOS image for the Raspberry Pi is based on Android 13 [55]. In Windows PC, a virtual machine running Ubuntu 22.04 LTS with bridge adapter network configuration is used for the openDDS application. In this way, we can assign a unique IP address in the network for the virtual machine.

The AAOS on the Raspberry PI boots up via an SD card that has the AAOS image flashed onto it. Once the Android app is developed on the Windows PC, it can deployed to the Raspberry Pi over WiFi using the Android Debug Bridge (ADB). Alternatively, we can build the Android Application Package (APK) file for our Android app in Android studio and copy the APK file in USB and then install on AAOS. All the components are connected through a WiFi router.

Chapter 6

Evaluation

In this chapter, we evaluate the approach designed in Chapter 4 and implemented in Chapter 5. This chapter also addresses our last research question [Q4] defined in Section 3.1.3. In section Section 6.1, we address the issues and challenges faced during our implementation, and DDS as a middleware package. Followed by observations of data communication between DDS applications using Wireshark. In the last section Section 6.3, the latency performance test was conducted.

6.1 Issues and Challenges

During the implementation phase, we faced various issues and challenges. Initially, the goal was to deploy an Android app using Android Virtual Device (AVD) in Android Studio as a proof of concept. However, we found out that the android emulator or AVD runs behind a virtual router or firewall service, that isolates it from the network interfaces of the development machine. Each emulator instance runs behind a virtual router or firewall service and lacks support for multicast traffic [56]. Due to these limitations, the openDDS service participants were unable to discover each other, as it relies on multicast messages by default for participant discovery. Therefore, to overcome these limitations in the Android virtual emulator, the development was shifted to physical hardware, particularly the Android Automotive Operating System image running on a Raspberry Pi.

The Android Automotive Operating System Raspberry Pi image was based on AOSP 13 [55]. This image was initially selected as a proof of concept demonstration and it was successfully installed and the Demo APP was tested. However, it was unstable and had issues such as random reboots and crashes, making it unsuitable for evaluating the latency performance tests. To address this, testing was performed on an Android smartphone running the standard Android 13. The main difference between the standard Android and AAOS image is that the Android Automotive extensions specified in Section 2.3 are not available in standard Android. However, this will not affect the scope of our implementation and evaluation, since our proposed solution operates at the application level, and the required components from the operating system remain the same for both the standard Android and AAOS.

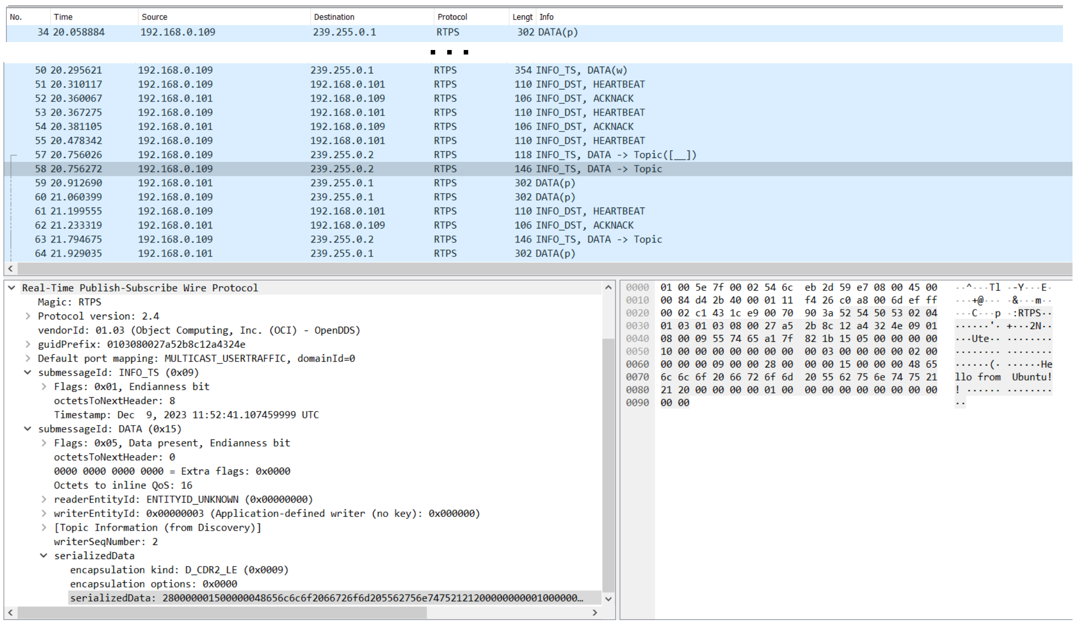
6.2 Observations and Results

After development, the Demo app was deployed to an Android smartphone using Android Studio. For the observation, we published a message from a virtual Linux machine and subscribed to this message in Android application. We used Wireshark to capture the packets from the network connection between the publisher application (in Linux VM) and the subscriber application (Android smartphone). The Android application is started as detailed in Section 5.4.2, which initializes the Subscriber and the DataReader. Once the Android application is running, we start the Linux virtual machine. After setting the required environmental variables, the publisher application is executed. This initializes the Publisher and the DataWriter.

Participant announcements are the initial phase of the DDS. All the participants, which transmit the user data using the RTPS are announced in the network using a built-in writer identified by DATA(p) in the packet list as shown in Figure 6.1. This participant writer records all the essential information about the participant it belongs to. We can also observe that the destination IP address is different from the source address. This destination is a multicast address, because RTPS discovery uses multicast to automatically discover other participants on the network. We can also configure this to use unicast.

The next packet is for Writer announcement, this is also announced using a built-in RTPS writer called the publication writer, which is indicated by DATA(w). The publication writer has the information about the actual DataWriters so that DataReaders can get ready to receive messages if they match. This information includes the topic name, topic type name, QoS policy, etc. In our application, we configured our DataWriter with the Reliability QoS policy. As you can see once the DataWriter announcement is made, it periodically sends the HEARTBEAT sub-message. As seen in Figure 6.1 if the matching reader is in the network, then it will respond with an ACKNACK sub-message.

Following the DataWriter announcement, the DDS topic instance is registered, refer to the topic instance explained in Section 2.1.1. The serialized data-carrying samples or RTPS messages, are then written to each matching reader. The serialized message can be seen in the highlighted area on the right side of Figure 6.1. It shows two ”Hello from Ubuntu!” string messages that are sent every second. Figure 6.2 shows the messages that are published from a Linux virtual machine and subscribed by the DemoApp on an Android device. Participants are periodically announced to the network to let know about their existence until they are destroyed. The frequency of announcing participants can be configured, in our case, we set this Resendperiod = 1 in the configuration file Listing 6.1.



```
No.  Time       Source          Destination     Protocol  Lengt Info
  34 20.058884  192.168.0.109   239.255.0.1     RTPS        302 DATA(p)
                                   . . .
  50 20.295621  192.168.0.109   239.255.0.1     RTPS        354 INFO_TS, DATA(w)
  51 20.310117  192.168.0.109   192.168.0.101   RTPS        110 INFO_DST, HEARTBEAT
  52 20.360067  192.168.0.101   192.168.0.109   RTPS        106 INFO_DST, ACKNACK
  53 20.367275  192.168.0.109   192.168.0.101   RTPS        110 INFO_DST, HEARTBEAT
  54 20.381105  192.168.0.101   192.168.0.109   RTPS        106 INFO_DST, ACKNACK
  55 20.478342  192.168.0.109   192.168.0.101   RTPS        110 INFO_DST, HEARTBEAT
  57 20.756026  192.168.0.109   239.255.0.2     RTPS        118 INFO_TS, DATA -> Topic([__])
  58 20.756272  192.168.0.109   239.255.0.2     RTPS        146 INFO_TS, DATA -> Topic
  59 20.912690  192.168.0.101   239.255.0.1     RTPS        302 DATA(p)
  60 21.060399  192.168.0.109   239.255.0.1     RTPS        302 DATA(p)
  61 21.199555  192.168.0.109   192.168.0.101   RTPS        110 INFO_DST, HEARTBEAT
  62 21.233319  192.168.0.101   192.168.0.109   RTPS        106 INFO_DST, ACKNACK
  63 21.794675  192.168.0.109   239.255.0.2     RTPS        146 INFO_TS, DATA -> Topic
  64 21.929035  192.168.0.101   239.255.0.1     RTPS        302 DATA(p)
```

```
Real-Time Publish-Subscribe Wire Protocol
  Magic: RTPS
  Protocol version: 2.4
  vendorId: 01.03 (Object Computing, Inc. (OCI) - OpenDDS)
  guidPrefix: 0103080027a52b8c12a4324e
  Default port mapping: MULTICAST_USERTRAFFIC, domainId=0
  submessageId: INFO_TS (0x09)
    Flags: 0x01, Endianness bit
    octetsToNextHeader: 8
    Timestamp: Dec  9, 2023 11:52:41.107459999 UTC
  submessageId: DATA (0x15)
    Flags: 0x05, Data present, Endianness bit
    octetsToNextHeader: 0
    0000 0000 0000 0000 = Extra flags: 0x0000
    Octets to inline QoS: 16
    readerEntityId: ENTITYID_UNKNOWN (0x00000000)
    writerEntityId: 0x00000003 (Application-defined writer (no key): 0x000000)
    [Topic Information (from Discovery)]
    writerSeqNumber: 2
    serializedData
      encapsulation kind: D_CDR2_LE (0x0009)
      encapsulation options: 0x0000
      serializedData: 28000000150000048656c6c6f2066726f6d205562756e747521212000000000001000000…
```

```
0000  01 00 5e 7f 00 02 54 6c  eb 2d 59 e7 08 00 45 00   ··^···Tl ·-Y···E·
0010  00 84 d4 2b 40 00 01 11  f4 26 c0 a8 00 6d ef ff   ···+@··· ·&···m··
0020  00 02 c1 43 1c e9 00 70  90 3a 52 54 50 53 02 04   ···C···p ·:RTPS··
0030  01 03 01 03 08 00 27 a5  2b 8c 12 a4 32 4e 09 01   ······'· +···2N··
0040  08 00 09 55 74 65 a1 7f  82 1b 15 05 00 00 00 00   ···Ute·· ········
0050  10 00 00 00 00 00 00 00  00 03 00 00 00 00 02 00   ········ ········
0060  00 00 00 09 00 00 28 00  00 00 15 00 00 00 48 65   ······(· ······He
0070  6c 6c 6f 20 66 72 6f 6d  20 55 62 75 6e 74 75 21   llo from  Ubuntu!
0080  21 20 00 00 00 00 01 00  00 00 00 00 00 00 00 00   ! ······ ········
0090  00 00                                              ··
```

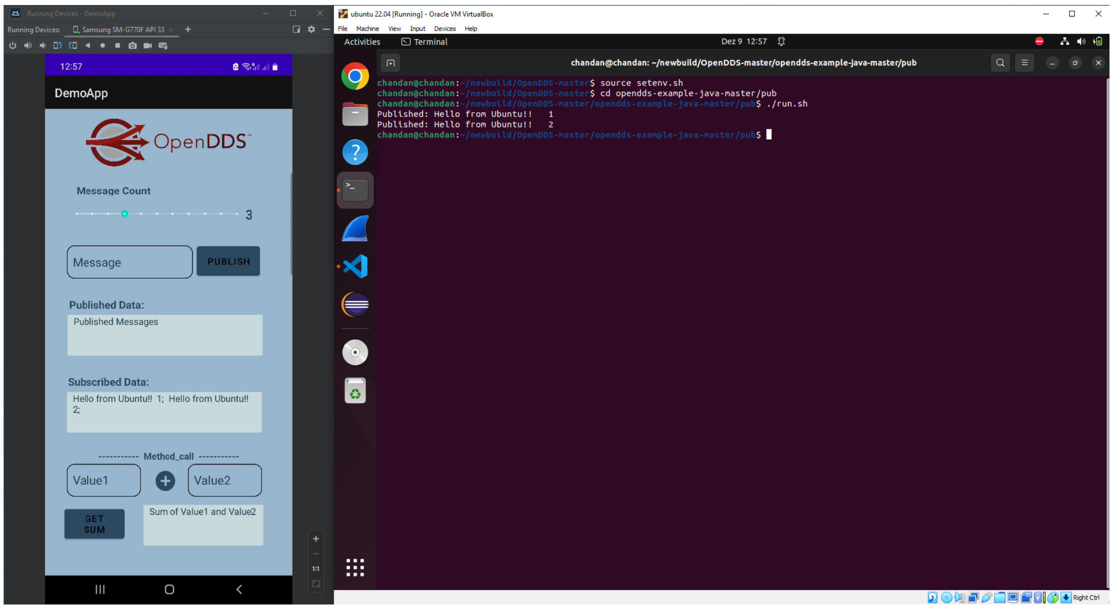
Figure 6.1: Message packets in Wireshark

```
[common]
DCPSGlobalTransportConfig=$file
DCPSDefaultDiscovery=fast_rtps

[transport/the_rtps_transport]
transport_type=rtps_udp

[rtps_discovery/fast_rtps]
ResendPeriod=1;
};
```

Listing 6.1: DDS configuration file for the demo application

6.3 Performance Test

In this section, we perform the latency test for the openDDS Android application. The objective is to measure the end-to-end time between the data being written and the acknowledgment being received. To achieve this, we use two pairs of DDS topics, similar to the implementation of method call in Section 5.4.2 and in Section 2.1.3. The 'pubmessage_topic' is used to publish the specified size of test data and the 'ackmessage_topic' is to receive the acknowledgment. This performance test involves a synchronous handshake operation, where a publisher sends test data of specified size with a sequence num-


Running Devices - DemoApp
Running Devices: Samsung SM-G770F API 33
12:57
DemoApp
OpenDDS
Message Count
3
Message
PUBLISH
Published Data:
Published Messages
Subscribed Data:
Hello from Ubuntu!! 1; Hello from Ubuntu!! 2;
----------- Method_call -----------
Value1
Value2
GET SUM
Sum of Value1 and Value2
ubuntu 22.04 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Activities
Terminal
Dez 9 12:57
chandan@chandan: ~/newbuild/OpenDDS-master/opendds-example-java-master/pub

```
chandan@chandan:~/newbuild/OpenDDS-master$ source setenv.sh
chandan@chandan:~/newbuild/OpenDDS-master$ cd opendds-example-java-master/pub
chandan@chandan:~/newbuild/OpenDDS-master/opendds-example-java-master/pub$ ./run.sh
Published: Hello from Ubuntu!!   1
Published: Hello from Ubuntu!!   2
chandan@chandan:~/newbuild/OpenDDS-master/opendds-example-java-master/pub$ 
```

Figure 6.2: DDS messages from Linux machine to Android application

ber and a subscriber responds with the same sequence number as an acknowledgment. The time measurements are obtained using the monotonic clock offered by openDDS, it has a separate class for handling time measurements 'ACE_Time_Value' with a resolution of one nanosecond.

Test Setup

As shown in Listing 6.2, we created a new IDL file to perform the latency test. The latency test was performed on the data sizes ranging from 64 bytes to 512 bytes. For our message, we used the primitive long data type, and the module 'DDSPerfTest' includes four respective topic types to publish data of various sizes, along with one 'AckMessage' type for acknowledgment. The IDL file is compiled for both Linux machines and cross-compiled for Android architecture. For the test, we developed a simple Android application similar to Section 5.4.2, where on button click, it initializes all the participants and waits for 'pubmessage_topic'. Once the data is received on this topic, it publishes the data to 'ackmessage_topic' as an acknowledgment. The total duration is then calculated on the Linux machine upon receiving the data on 'ackmessage_topic'. On Linux application, We start the timer once the message is being written on 'pubmessage_topic' and stop it once the acknowledge message is received. The end-to-end latency is then calculated by subtracting the start time from the stop time. We evaluate the performance with 5000 data samples. We calculate the mean latency, minimum and maximum latency in the test duration, and also the jitter is calculated as the standard deviation from the expected or mean latency.

Results

Figure 6.3 shows the mean latency across various data sizes ranging from 64 bytes to

512 bytes. For latency comparison, we also tested end-to-end latency on openDDS C++ applications running on the same machine. Table 6.1 shows the latency metric comparison between openDDS C++ applications running on a Linux virtual machine and communication between Linux to Android applications, with Android app utilizing the openDDS Java library. From the results, it is observable that the latency between Linux to Android app is higher compared to C++ applications running on a virtual machine. The reason for the higher latency in Android app is due to its Java application that runs on a virtual Java machine, which can introduce some overhead compared to native C++ applications. Also, the network capability of hardware drivers in Android devices is lower compared to PC hardware, which again limits the bandwidth. And it is important to note that the tests were performed on wireless connections or WiFi on the Android device.

```
module DDSPerfTest {
  @topic
  struct Pt64 {
    long seqnum;
    long value1;
        ...
    long value15;
  };
  @topic
  struct Pt128 {
    long seqnum;
        ...
  };
  @topic
  struct Pt256 {
    long seqnum;
        ...
  };
  @topic
  struct Pt512 {
    long seqnum;
        ...
  };
  @topic
  struct AckMessage {
    long seqnum;
  };
};
```

Listing 6.2: DDSPerfTest.idl for latency test

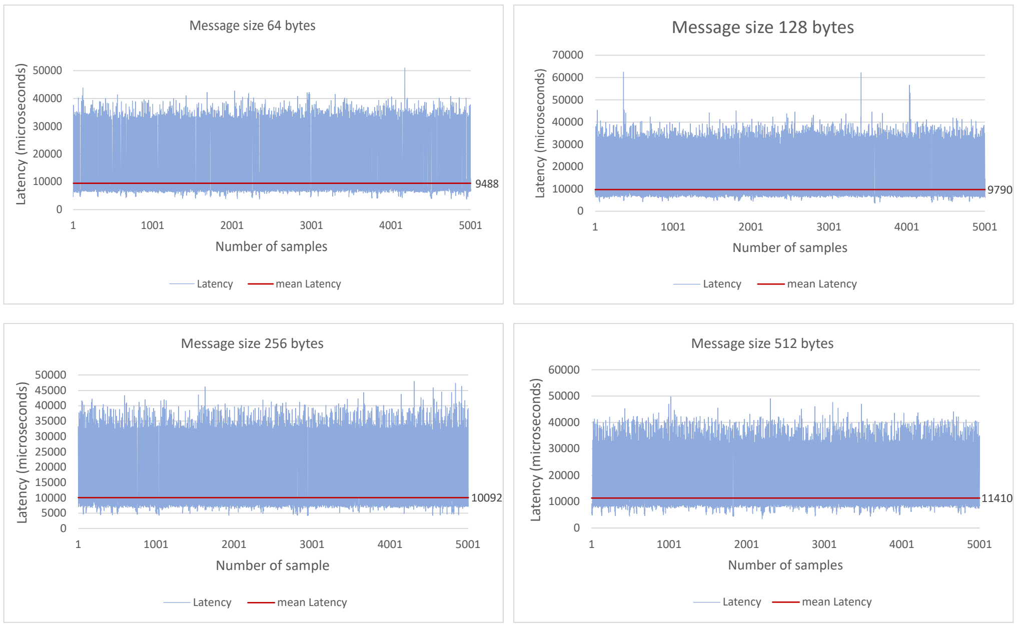


Figure 6.3: Latency measurements for various data sizes

Latency Metric (ms)	64 bytes		128 bytes		256 bytes		512 bytes	
	L-L	L-A	L-L	L-A	L-L	L-A	L-L	L-A
Mean	0.276	9.488	0.282	9.790	0.287	10.092	0.288	11.410
Min	0.259	3.669	0.263	3.734	0.262	4.226	0.261	3.466
Max	10.752	51.000	12.000	62.470	8.584	47.903	11.868	49.709
Std_dev (Jitter)	0.423	8.829	0.520	9.039	0.593	9.096	0.526	9.603

Table 6.1: Latency metrics
L-L : C++ applications on Linux VM
L-A: C++ application on Linux VM & Java application on Android 13

Chapter 7

Conclusion

This chapter provides a summary of the contributions and conclusions from the thesis work. It also suggests ideas for future work.

7.1 Summary

This thesis examined the possibility of establishing the inter-domain communication between Android Automotive and Adaptive AUTOSAR domains using DDS middleware protocol. DDS functionalities were analyzed to achieve service-oriented communication between these two domains. In this thesis work, we formulated the four research questions, which mainly addressed the capabilities of DDS as a middleware, and the concepts of integrating DDS library at different architecture levels for inter-domain communication.

The first research question [Q1] addresses the key features offered by DDS for service-oriented communication and its significance in automotive applications. In the second research question [Q2], the key challenges and communication requirements between different architectures such as Android Automotive and Adaptive platforms are discussed. This also addresses how these communication requirements can be established using DDS. In the third question [Q3], communication patterns and requirements in the Android system are examined and different approaches for integrating DDS library into the Android architecture are discussed. These approaches were further analyzed using the ISO 25010 standard. OpenDDS library was implemented with Java bindings. The choice of DDS vendor is more up to the solution provider and system requirements. The RTPS protocol makes it easier for intercommunication between different DDS vendors and applications with different language bindings. As a proof of concept openDDS was implemented at the Android application level, where the cross-compiled openDDS Java library is included in the Android app. Some modifications were made to the implemented solution and testing was performed on physical hardware, with Android

Automotive running on Raspberry Pi as the client and the Server application running on a Ubuntu virtual machine. It was observed that the openDDS Android application successfully discovers the service from the virtual machine. Both the publish-subscribe communication pattern and a simple method call pattern using DDS API were successfully demonstrated between the server and client applications.

Finally, in the fourth question [Q4], timing measurements were conducted to calculate end-to-end latency between DDS applications. Latency measurements between DDS applications for various data sizes were measured. From the results, it was observed that the processing time in the Android system is higher compared to applications on the Ubuntu virtual machine on a PC.

7.2 Future Work

Modified DDS API
As discussed in Section 4.3 we used opensource standard DDS API for our Android app development. To have seamless integration of DDS middleware for inter-domain communication it is necessary to have a modified DDS API that supports the communications patterns and service features in Adaptive AUTOSAR. Current open-source DDS follow OMG standard for distributed systems. Therefore, we need a modified API specifically designed for automotive In-vehicle communication, so that we can get rid of unnecessary features and reduce the overhead.

VHAL and system level integration
Integrating DDS middleware at a system level and VHAL layer provides a more standardized approach for future development. The AOSP source code allows implementation of custom native system services, by which we can implement DDS library as a system service and offer various services to the applications. Additionally,AOSP has an emulated VHAL layer used for building the AAOS emulator image, providing a reference for VHAL implementation. This solution can be extended to include the integration of DDS library to incorporate service-oriented communication from the Adaptive domain into the existing AAOS-Car API based architecture, as explained in Section 4.3.3.

Model-based code generation
The Adaptive platform incorporates model-based code generation for generating all the components necessary for data exchange between applications based on a ARXML file. Similarly, a software tool can be developed that generates all the components required for communication between AAOS and the Adaptive platform from a DDS IDL file. This tool would allow developers to model the service interface and automate all the steps on the Android side. As a result, app developers would only need to use the APIs provided by the code generator in the apps, reducing the development process.

Bibliography

[1] Object Management Group (OMG). Data distribution service for real-time systems (dds) - rpc over dds version 1.0. Online, 2017.

[2] Android Developers. Android platform guide. Online, 2023.

[3] Author's Name. Android binder framework. Online, 2019.

[4] Peter Gessler. Android automotive embedded os whitepaper. Online, 2020.

[5] AUTOSAR. Autosar ara comapi. Online, 2022.

[6] OpenDDS. Introduction to opendds. Online, Year.

[7] Marcel Rumez, Daniel Grimm, Reiner Kriesten, and Eric Sax. An overview of automotive service-oriented architectures and implications for security countermeasures. *IEEE Access*, 8:221852–221870, 2020.

[8] Dušan Kenjić, Dušan Živkov, and Marija Antić. Automated data transfer from adas to android-based ivi domain over some/ip. *IEEE Transactions on Intelligent Vehicles*, 8(4):3166–3177, 2023.

[9] Dušan Kenjić, Marija Antić, and Dušan Živkov. Service-oriented communication between adas and ivi domains in automotive solutions. 2022.

[10] Luka Bilac, Dusan Stanisic, Dusan Kenjic, and Marija Antic. One solution of an android in-vehicle infotainment service for communication with advanced driver assistance system. In *2022 45th Jubilee International Convention on Information, Communication and Electronic Technology (MIPRO)*, pages 1420–1425, 2022.

[11] Wan-Hua Cao, Bei Me, Hai-Xin Wu, and Xiong Cheng. Design of publish/subscribe middleware based on dds. *Jisuanji Gongcheng/ Computer Engineering*, 33(18):78–80, 2007.

[12] Michael James Michaud, Thomas R. Dean, and Sylvain P. Leblanc. Attacking omg data distribution service (dds) based real-time mission critical distributed systems. *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 68–77, 2018.

[13] Paulo Neves, Rodrigo Santos, and Hugo Oliveira. Towards real-time big data stream analytics. *Future Internet*, 15:24, 2021.

[14] Object Management Group (OMG). Data distribution service for real-time systems (dds) - interoperability wire protocol specification (rtps) version 2.3. Online, Year.

[15] Nanbor Wang, Douglas C. Schmidt, Hans van't Hag, and Angelo Corsaro. Toward an adaptive data distribution service for dynamic large-scale network-centric operation and warfare (ncow) systems. In *MILCOM 2008 - 2008 IEEE Military Communications Conference*, pages 1–7, 2008.

[16] The DDS Foundation. What is dds? Online, Year.

[17] Object Management Group (OMG). Corba interface definition language (idl) version 4.2. Online, Year.

[18] eProsima. Fast dds documentation. Online, 2019.

[19] RTI Real-Time Innovations. Getting started with rti connext dds. Online, 2020.

[20] Object Management Group (OMG). Dds-dlrl version 1.4. Online, 2015.

[21] eProsima. eprosima fast rtps user manual. Online, 2014.

[22] Wikipedia. Android (operating system). Online, 2023.

[23] Android Open Source Project. Android source code documentation: Kernel. Online, 2023.

[24] google blog. Treble plus one equals four. Online, 2020.

[25] Android Open Source Project. Android source code documentation: Vehicle system isolation. Online, 2023.

[26] Chris Simmonds. Introduction to aaos. Online, 2021.

[27] AUTOSAR. Autosar platform design. Online, 2022.

[28] AUTOSAR. Autosar software architecture. Online, 2022.

[29] Mehmet Çakir, Timo Häckel, Sandra Reider, Philipp Meyer, Franz Korf, and Thomas C. Schmidt. A qos aware approach to service-oriented communication in future automotive networks. *CoRR*, abs/1911.01805, 2019.

[30] Philipp Obergfell, Stefan Kugele, and Eric Sax. Model-based resource analysis and synthesis of service-oriented automotive software architectures. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 128–138, 2019.

[31] Fetiha Ben Cheikh, Mohamed Anis Mastouri, and Salem Hasnaoui. Implementing a real-time middleware based on dds for the cooperative vehicle infrastructure systems. In *2010 6th International Conference on Wireless and Mobile Communications*, pages 492–497, 2010.

[32] Claudio Scordino, Angela Gonzalez Mariño, and Francesc Fons. Hardware acceleration of data distribution service (dds) for automotive communication and computing. *IEEE Access*, 10:109626–109651, 2022.

[33] Stefan Kugele, David Hettler, and Jan Peter. Data-centric communication and containerization for future automotive software architectures. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 65–6509, 2018.

[34] Basem Almadani, Najood Alshammari, and Anas Al-Roubaiey. Adaptive cruise control based on real-time dds middleware. *IEEE Access*, 11:75407–75423, 2023.

[35] Wikipedia. Android automotive. Online, 2023. 10.26.2023.

[36] Gianpaolo Macario, Marco Torchiano, and Massimo Violante. An in-vehicle infotainment software architecture based on google android. In *2009 IEEE International Symposium on Industrial Embedded Systems*, pages 257–260, 2009.

[37] Simon Fürst and Markus Bechter. Autosar for connected and autonomous vehicles: The autosar adaptive platform. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 215–217, 2016.

[38] Dušan Kenjić, Marija Antić, and Tihomir Anđelić. Theoretical aspects of automatically generated service-oriented communication between adas and ivi domains. In *2022 IEEE Zooming Innovation in Consumer Technologies Conference (ZINC)*, pages 87–92, 2022.

[39] AUTOSAR. Autosar software specification communication management. Online, 2022.

[40] Dušan Kenjić, Dušan Živkov, and Marija Antić. Automated data transfer from adas to android-based ivi domain over some/ip. *IEEE Transactions on Intelligent Vehicles*, 8(4):3166–3177, 2023.

[41] Author's Name (if available). Title of the document. Online, Year.

[42] AUTOSAR. Autosar adaptive platform standards. Online, 2023.

[43] eProsima. Fast dds real-time use cases. Online, 2023. 11/23.

[44] RTI ITK Engineering GmbH. Enabling flexible vehicle architectures with autosar and dds. Online, Year. 10/23.

[45] AUTOSAR. Autosar prs: Some/ip protocol. Online, 2017. 11/2023.

[46] Object Management Group (OMG). Data distribution service (dds) version 1.4. Online, 2015. 11/2023.

[47] Markus Helmling Alexander Mayr. Middleware protocols in the automobile: Service-oriented, data-centric or restful? Online, 2020. 11/2023.

[48] Michael Pöhnl, Alban Tamisier, and Tobias Blass. A middleware journey from microcontrollers to microprocessors. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 282–286, 2022.

[49] gRPC. grpc: A high-performance, open-source universal rpc framework. Online, Year. 11/23.

[50] Pavel P´ı˘sa Martin Vajnar, Michal Sojka. Porting of real-time publish-subscribe middleware to android. Online, Year. 11/23.

[51] Fernando Garcia-Aranda. Integrating dds into the autosar adaptive platform. Blog Post, 2023. 11/23.

[52] OpenDDS. Opendds github repository. Online, 2023. 06/23.

[53] OpenDDS. Opendds android build guide. Online, 2023. 11/23.

[54] Adam Mitz. Opendds java example github repository. Online, 2023. 08/23.

[55] Damian Petrecki. Android automotive os 13 on raspberry pi 4b. Online, 2023. 10/23.

[56] Android Developers. Emulator networking. Online, 2023. 11/23.