Preface

This report examines Adaptive Autosar specification and ROS 2 implementation and compares them side by side. The report is aimed at Apex.AI customers and partners in giving them guidance on how to combine both technologies or select one over the other.

In this report we exclusively examine technical features only and try to propose a solution that technically fits better for the purpose of programming of self-driving cars and robots in safety-critical applications. At this point we do not consider company policies such as “we are a member of Autosar consortium, hence we must use Adaptive Autosar”.

Executive Summary

The key difference for both is in that ROS 2 follows "code first, specification next" approach since ROS 2 is based on ROS which has been used and validated in the robotics field for over 12 years. Adaptive Autosar has been partly written from scratch and partly it has adopted proven technologies such as SOME/IP, DLT, UDS. Adaptive Autosar follows specification first, production code next approach.

ROS 2 excels in terms of large community and tools for e.g. data recording, replay, visualization and debugging. Also binding to other languages such as e.g. Python and Java. ROS 2 leverages common and open-source tools as opposed to tools that are provided by individual Adaptive Autosar vendors and therefore have a smaller user base.

Adaptive Autosar uses SOME/IP which is service oriented, ROS 2 uses DDS which is data oriented. The former can only be switched between UDP and TCP, the latter has QoS. Thanks to DDS, data transport in ROS 2 is extremely efficient and fast due to shared memory (SHM) and Zero-Copy implementations.

1 https://www.autosar.org/standards/adaptive-platform/
2 https://index.ros.org/doc/ros2/
3 http://some-ip.com/
ROS 2 version F will be available in June 2020 as the second long-term release. It is not clear how far individual implementations of Adaptive Autosar are and whether they are based on the latest specification (19.03). Our understanding suggests that most of the implementations are based on the 18.03 specification. Adaptive Autosar specification on the other hand outperforms ROS 2 in the areas of software update and configuration management as well as DEXT diagnostics management.

Both technologies benefit and have challenges with the modern C++. In particular the largest challenges are:

1. Standard exceptions handling
2. Static containers (strings, vectors, maps)
3. Memory management

Currently the best summary would be that Adaptive Autosar excels in terms of compatibility with the existing in-vehicle infrastructure and ROS 2 is outstanding for the application development. Ideally both technologies would co-exist in the production.

What is ROS 2

A robotics framework is a collection of software tools, libraries, conventions, and APIs made to simplify the task of developing software for a complex multi-sensor, multi-actuator system, like a vehicle.

In most cases, the use of robotics framework dictates the general architectural principles of the developed software. For example, if the software is centralized or decentralized, real-time or non-real-time, etc.

A key component is the middleware, which is the glue that holds together the numerous components of a robotics framework. The most basic task of the middleware is to provide the communications infrastructure between software nodes in an autonomous vehicle.

The typical use case for a robotics framework is to provide the essential interfaces between the high level (software) and the low level (hardware) components of the system. These interfaces and components consist of various operating system (OS) specific drivers that would take a single developer a significant amount of time to develop.

The ROS 2 architecture is depicted in figure below:

© 2019 Apex.AI, Inc. Confidential.
ROS 2 is primarily developed by Open Robotics and ROS 2 Technical Steering Committee. It has several 1000 users and it is being used by several 100 companies in autonomous driving as well as robotics.

What is Adaptive Autosar

An official definition puts it as “AUTOSAR Runtime for Adaptive Applications (ARA). Two types of interfaces are available, services and APIs. The platform consists of functional clusters which are grouped in services and the Adaptive AUTOSAR Basis”. In practice it is another robotics framework that consists of the following so-called functional clusters:

---

6 https://www.openrobotics.org/
7 https://discourse.ros.org/t/introducing-the-ros-2-technical-steering-committee/6132
The framework specification is developed by the Autosar consortium\(^8\). An actual implementation is then done by companies such as Elektrobit, Vector, ETAS, Mathworks, Aubass, etc. Development is done by consortium members and ONLY the specification is released in the open. Adaptive Autosar builds on top of classic Autosar which was used for programming of applications on single-core microcontrollers.

It is currently unknown to Apex.AI how many companies are using which Adaptive Autosar implementation.

**Key Differences and Key Similarities between Adaptive Autosar Specification and ROS 2 implementation**

We list key differences and key similarities along the functional clusters list, and add some additional items at the end.

**Execution Management**

**Adaptive Autosar**: Execution Management is responsible for all aspects of system execution management including platform initialization and startup/shutdown of Applications. Execution Management works in conjunction with the Operating System to perform run-time scheduling of Applications. There is a support for platform and applications'...
lifetime management as well as deterministic execution for both data and time determinism.

**ROS 2**: In ROS 2 execution management is done via ROS 2 launch mechanism\(^9\) which is launched by an operating system. Launch system is responsible to launch and monitor ROS 2 applications via e.g. global error reporting system which is aware of both platform as well as applications’ lifecycle management. Using an underlying real-time DDS implementation\(^10\) it aids to write applications that are data deterministic.

**State Management**

**Adaptive Autosar**: State Manager is an ECU-specific feature that is to be developed by a system integrator. It should perform the following functions:

1. FunctionGroups can be requested to be set to a dedicated state
2. (Partial) Networks can be requested to be de- / activated
3. The machine can be requested to be shutdown or restarted
4. Other Adaptive (Platform) Applications can be influenced in their behavior
5. Project-specific actions could be performed

**ROS 2**: In ROS 2 such a state manager is currently unavailable and it would need to be custom written. Components exist that allow for writing of a stateful system: LifeCycleNode, launch, ROS 2 logging mechanism.

**Communication Management**

**Adaptive Autosar**: This so-called *ara::com* module serves two purposes:

1. It provides an API to the developer of an application for autonomous driving
2. It uses service-oriented communication for the applications to communicate between each other. Services consists of events, fields and methods.

*ara::com* provides language bindings (auto generated code for inclusion in the application) as well as network bindings (wire format). Definition of Events, Methods and Fields is done in ARXML\(^11\) format and then code generators are used to generate C++ classes. They state that special priority is given to performance and type safety.

---

\(^9\) [https://index.ros.org/doc/ros2/Tutorials/Launch-system/](https://index.ros.org/doc/ros2/Tutorials/Launch-system/)


Adaptive Autosar applications can be discovered statically, semi-statically (server does not know the clients) or completely dynamically.

**ROS 2:** An alternative to `ara::com` in ROS 2 is `rclcpp`. It is also user facing and provides functionality such as creating nodes, publishers and subscribers. `rclcpp` builds on top of `rcl` and the `rosidl` API, and it is designed to be used with the C++ messages generated by `rosidl_generator_cpp` for language bindings.

ROS2 uses DDS and an RTPS as a wire protocol. DDS takes care of the network binding, that is, how the data is serialized/deserialized into (a CDR format) and bound to a particular transport protocol (e.g. shm, udp, etc.).

`rclcpp` reuses the implementation in `rcl` and it is able to maintain a consistent behavior with the other client libraries that use the `rcl` API (e.g. `rclpy` for Python bindings).

ROS 2 application use DDS discovery mechanism and can be configured for a static or dynamic discovery.

**Key Difference:**
1. Service registry as a central component in Adaptive Autosar could be a single point of failure
2. ROS 2 can have direct DDS access and thus use all its strengths, especially quality of service settings
3. ROS 2 applications also have language bindings to other languages such as Python
4. ROS 2 **could use** real-time DDS implementation RTI Connext Micro which provides 3 key differentiators compared to SOME/IP used in Adaptive Autosar:
   a. Security
   b. Large and efficient data transport by using intra-process transport or SHM or Zero-Copy
   c. Quality of Service settings with which you can assert reliability or build redundant systems (over e.g. Ownership QoS)
5. ROS 2 uses IDL 4.2 for the specification of messages

**RESTful Communication**

**Adaptive Autosar:** `ara::rest` was introduced to be able to communicate with a mobile HTTP/JSON client and vice versa and thus inter-operate with the non-Adaptive Autosar peers.

**ROS 2:** ROS 2 for the same purposes leverages DDS and Connext DDS Connector\(^2\) specifically to communicate with the non-ROS 2 peers. It has support for languages such as Node.js/JavaScript, Python, Go, etc. Furthermore, it is also compatible with the technologies

\(^2\) [https://github.com/rticommunity/rticonnextdds-connector](https://github.com/rticommunity/rticonnextdds-connector)
such as https://github.com/RobotWebTools/ros2-web-bridge which communicate with ROS 2 peers over web sockets.

Diagnostics

**Adaptive Autosar:** The Diagnostic Management realizes the ISO 14229-5 (UDSonIP) which is based on the ISO 14229-1 (UDS) and ISO 13400-2 (DoIP).

The scope is to abstract the diagnostic protocol from Adaptive Applications. The interfaces are harmonized with the Classic Platform (e.g. SetEventStatus) to allow an easy change for Classic Platform developers.

**ROS 2:** In ROS 2 we have a concept for collecting, publishing, aggregating, and visualizing of diagnostic messages [http://wiki.ros.org/diagnostics](http://wiki.ros.org/diagnostics). We believe that to be able to visualize DEXT messages we need to write an application that collects and publishes data in DEXT format as ROS 2 Diagnostic message.

**Key Difference:**
1. ROS 2 currently does not have an application for handling of DEXT messages.

Persistency

**Adaptive Autosar:** Persistency offers mechanisms to applications and other functional clusters of the Adaptive Platform to store information in the non-volatile memory of an Adaptive Machine.

The Persistency APIs take storage location identifiers as parameters from the application to address different storage locations.

The available storage locations fall into two categories:

1. Key-Value Storage
2. File Storage

**ROS 2:** ROS 2 provides a safe configuration management system (called Parameters) that covers the following use cases:

1. Load configuration from a configuration source
2. Process level configuration (node independent configuration)
3. Shared configuration (shared across the system, e.g. vehicle dimensions, other high level configurations, etc.)

Access to above configuration is:
1. Read-only
2. Real-time safe and therefore also does not use dynamic memory

The configuration data can come from different sources:

1. Configuration file (currently implemented)
2. ROS 2 Services
3. Centralized Server (parameter server node)
4. Command-Line Argument
5. Programatically

**Key Difference:**

1. ROS 2 configuration system is implemented as a separation of concern: source of configuration data is separated from the configuration management used to configure ROS 2 applications

**Time Synchronization**

This topic deals with the time synchronization between different applications and/or ECUs.

**Adaptive Autosar:** In Adaptive Autosar a Time Synchronization API was designed that provides a functionality wrapped around the StbM module of the Classic Platform, but with a std::chrono like flavor.

The following functional aspects are considered by the Time Synchronization module:

1. Startup Behavior
2. Constructor Behavior (Initialization)
3. Normal Operation
4. Error Handling

The following functional aspects will be considered in future releases:

1. Shutdown Behavior
2. Error Classification
3. Version Check

**ROS 2:** In ROS 2 we differentiate two fundamental concepts:

1. **Clocks:** Clocks provide the user with a time point (which is mostly represented as some kind of integer or float). There are clocks for system time, steady time, ...
2. **Time tools:** This is the tooling to deal with time. Like calculating the duration, convert seconds to years, ...

After an extensive evaluation we determined that `std::chrono` is the best option for ROS 2 for the following reasons:

1. `std::chrono` uses a different clock type which is encoded in all time types. Therefore it is impossible to mix up times coming from different clocks, for example, calculating the duration between a time point in simulated time and a time point in real time will result in a compilation error
2. `std::chrono` supports customizable time representations. This means `std::chrono` can be protected against time overflow, ...
3. `std::chrono` is not only fully documented (that is easy to accomplish) but there exist a lot of tutorials and books on how to properly use it
4. `std::chrono` can be easily wrapped around the simulated clock

ROS 2 uses `std::chrono` in all of the above listed use cases.

Key Difference: ROS 2 is currently limited to use of `std::chrono` steady_time because `system_time` can travel backwards due to simulated time looping, ntp updates, an administrator changing the time, or leap second correction. Adaptive Autosar supposedly have an implementation for the `system_time` and can therefore safely synchronize time between various ECUs.

Network Management

**Adaptive Autosar:** The AUTOSAR NM is based on a decentralized network management strategy, which means that every network node performs activities independently depending only on the NM messages received and/or transmitted within the communication System.

**ROS 2:** In ROS 2 we use DDS unicast for communication between ROS 2 Nodes. We rely on the system administrator to configure the platform network correctly. In the future we plan to explore Time-Sensitive Networking.

Update and Config Management

**Adaptive Autosar:** One of the declared goals of Adaptive AUTOSAR is the ability to flexibly update the software and its configuration through over-the-air updates. To support changes in the software on an Adaptive Platform, the Update and Configuration Manager (UCM) provides an Adaptive Platform service that handles software update requests.

UCM is responsible for updating, installing, removing and keeping a record of the software on an Adaptive Platform. Its role is similar to known package management systems like
dpkg or YUM in Linux, with additional functionality to ensure a safe and secure way to update or modify software on the Adaptive Platform.

**ROS 2:** ROS 2 is packaged in ROS packages which are described in the following REPs:


From ROS packages we generate platform specific binary artifacts (Debians and docker volumes for Linux, flat files system tarballs for QNX) which are then securely deployed to the target ECUs. The process is as follows:

1. Create a binary artifact following software release process at Open Robotics
2. Copy ROS2-image-vX.Y.Z.tar.gz onto the target
3. Unsign, decrypt, and untar the image
4. Run the installation script based on customer specific env variables (e.g. RTOS type, Kernel version, Kernel settings, package dependency, ...)

**Key Difference:** In ROS 2 we currently do not account for vehicle package manifest.

**Identity and Access Management**

**Adaptive Autosar:** The concept of Identity and Access Management (IAM) is driven by the increasing need for security, as the AUTOSAR Adaptive Platform needs a robust and well-defined trust relationship with its applications.

IAM introduces privilege separation for Adaptive Applications and protection against privilege escalation in case of attacks. In addition, IAM enables integrators to verify access on resources requested by Adaptive Applications in advance during deployment. Identity and Access Management provides a framework for access control for requests from Adaptive Applications on Service Interfaces, Functional Clusters of the Adaptive Platform Foundation and related modeled resources.

**ROS 2:** Security in ROS 2 is implemented on the data flow level, and follows DDS Service Plugin Interface (SPI). In particular there are 3 SPIs:

1. Authentication
2. Access control list (ACL)
3. Encryption

The following occurs when the security features are enabled:

1. Authentication phase
In ROS 2 the entire trust is based on the Certificate Authority (CA), which assumes that nodes authenticate with each other using a TLS handshake (public/private keypair).

This authentication happens at the time when the nodes call the `create_node`, `create_sub`, and `create_pub` functions.

Currently the CA is a mimic using the openssl tool to generate certificate (sha256); however, there are plans to implement and enforce the use of a Hardware Security Module.

2. CA key generation and certificate signing
   - Create root pair
   - Create intermediate pair
   - Create actual key pair
   - Sign certificates by encrypting them with e.g. `aes256`

3. ACL phase
   - Nodes exchange a permissions file, signed with SMIME protocol

As part of the future work we plan to implement process-level security as well as integration with the Hardware Security Modules (HSM).

Cryptography
Analysed as part of Identity and Access Management.

Log and Trace

**Adaptive Autosar:** The logging information can be forwarded to multiple sinks, depending on the configuration, such as the communication bus, a file on the system, and a serial console.

The Log and Trace rely on the LT protocol standardized within the AUTOSAR consortium.

The protocol ensures that the logging information is packed into a standardized delivery and presentation format. Furthermore, the LT protocol (Adopted from GENIVI) can add additional information to the logging messages, such as an ECU ID. This information can be used by a logging client to relate, sort, or filter the received logging frames.

**ROS 2:** See detailed explanation [https://index.ros.org/doc/ros2/Concepts/Logging/](https://index.ros.org/doc/ros2/Concepts/Logging/).

Other Key Advantages of ROS 2

1. Adaptive Autosar component is a process. In ROS 2 a component is a node and there can be multiple nodes per process
2. Adaptive Autosar only uses C++ on the application level. ROS 2 prototypes can also be done in e.g. Python

© 2019 Apex.AI, Inc. Confidential.
3. Adaptive Autosar does not provide tools for data recording, replay, visualization, or debugging. Especially large data recording seems to be completely missing.

4. Adaptive Autosar uses SOME/IP which is service oriented, ROS 2 uses DDS which is data oriented. The former can only be switched between UDP and TCP, the latter has QoS.

5. ROS 2 has only one open-source implementation which several 1000 users use. Adaptive Autosar has several closed-source implementation which are largely incompatible with each other.

6. ROS 2 supports performant data transport protocols such as Shared Memory or Zero-Copy (via RTI Connext Micro).

7. ROS 2 includes reference implementations of device drivers such as e.g. Velodyne VLP-16 HiRes.

8. ROS 2 provides tools such as a static transform library, parts of the certifiable math library, and a time synchronization library for topic synchronization.

Sources

1. BMW: An Open-Source Software Platform for Autonomous Driving Systems
2. ElektroBit: Adaptive AUTOSAR for high-performance in-car computers