A Middleware Journey from Microcontrollers to Microprocessors

Michael Pöhnl, Alban Tamisier, and Tobias Blass
Apex.AI
{*michael.poehnl, alban.tamisier, tobias.blass*}*@apex.ai*

***Abstract*—This paper discusses some of the challenges we encountered when developing Apex.OS, an automotive grade version of the Robot Operating System (ROS) 2. To better understand these challenges, we look back at the best practices used for data communication and software execution in OSEK-based systems. Finally we describe the extensions made in ROS 2, Apex.OS and Apex.Middleware to meet the real-time constraints of the targeted automotive systems.**

I. Introduction

Automotive software has usually been distributed across numerous microcontrollers throughout the car. However, modern functionality such as assisted and autonomous driving requires more processing power and tighter integration of the various components of a car, bringing this traditional architecture to its limits. Automotive manufactures therefore increasingly deploy centralized vehicle computers, which combine the functionality of multiple microcontrollers in a single, more powerful microprocessor.

This transition requires major changes to the underlying software. In the past, microcontrollers usually ran the OSEK operating system, e.g. as part of an AUTOSAR Classic deployment. On microprocessors, manufacturers usually prefer to use a POSIX operating system like Linux or QNX. Compared to OSEK, POSIX systems have a variety of advantages such as memory isolation, better developer tooling, and more advanced thread schedulers.

Unfortunately, many of those features come at a performance cost. Although these costs are usually negligible compared to the increased performance of microprocessors, some common software architectures from the AUTOSAR Classic world may incur large overheads on POSIX systems. We have observed that such overheads can be a significant fraction of the actual computational load, negating much of the promised performance benefits of microprocessors and threatening the deployment's viability.

In this paper we identify a few core assumptions in software architectures designed for AUTOSAR Classic that lead to poor performance on POSIX systems. Specifically, architectures derived from AUTOSAR Classic often assume that communication has negligible cost and that context switches are rare. We then present how ROS 2 and our products Apex.OS and Apex.Middleware address those problems and enable well-established software architectures to run efficiently on modern POSIX systems. Specifically, communication costs and context switches are avoided through the following four mechanisms:

a) Leveraging zero-copy communication: Some middlewares supported by ROS 2, including Apex.Middleware, provide *zero-copy communication*, which allows applications to cheaply transfer large messages such as video frames via shared memory.

b) Avoiding context switches in the middleware: Apex.Middleware avoids separate communication threads in the communication path. Compared to many DDS implementations, it therefore reduces the number of context switches required for message transmission.

c) Executing multiple nodes in a single thread: ROS 2 and Apex.OS can multiplex multiple independent tasks (called *nodes*) onto a smaller set of *executor threads*, thereby reducing context switching overhead. In addition, Apex.OS allows developers to run entire chains and graphs in a predefined order, which reduces interaction with the middleware and ensures that the nodes run in a predictable order.

d) Identifying non-triggering topics: Apex.OS allows developers to specify which topics are *triggering* and *non-triggering*, which avoids needless wake-ups for messages on non-triggering topics and leaves more of the message queuing and retention to the middleware.

The remainder of this paper is structured as follows: we first provide a brief overview over the relevant frameworks (section II). We then discuss the performance issues mentioned above in more detail, using a real-world system as illustration (section III). Finally, we discuss the mechanisms provided by Apex.OS and Apex.Middleware that address these performance issues and empirically evaluate them (section IV).

II. Background

We briefly review the traditional platforms for automotive development, OSEK and AUTOSAR Classic, and then discuss ROS, a popular robotics framework that is commonly used to prototype autonomous-driving systems.

A. OSEK / AUTOSAR Classic

In the last decades, automotive electronic control units (ECU) have usually been implemented using an OSEK/VDX operating system [1], usually as part of an *AUTOSAR Classic* platform [2]. The OSEK/VDX standard (or OSEK for short) dates back to 1993 and describes among other things a standardized real-time operating system for microcontrollers without a memory-management unit.

An OSEK system comprises multiple *tasks*, each of which represents an independent thread of execution. Each task is

978-3-9819263-6-1/DATE22/©2022 EDAA

either triggered by a time schedule, from an interrupt handler or an event sent by another task and then scheduled with a fixed-priority scheduler. A per-task configuration parameter determines whether the scheduler may preempt a task. The number of tasks, their priorities and core affinities are configured statically and cannot be changed at runtime.

Usually an additional execution abstraction in the form of C functions or C++ methods is used that is more fine-grained than an OSEK task. These so called *runnables* are mapped to tasks, such that each task may run one or more runnables in a predefined order every time it gets activated. For example, it is good practice to group all runnables with a 10 ms period into a single 10 ms OSEK task, or to execute all runnables of a processing pipeline one after the other in a single task. Multiplexing runnables this way not only reduces context switching and task management overhead but also economizes on the limited number of tasks that an OSEK operating system provides.

Runnables communicate through messages with predefined types, senders, and receivers. Since all OSEK tasks share a single address space, message passing is usually implemented through global variables and therefore incurs *at most* the cost of a `memcpy` under a mutex. In many cases even that cost can be avoided by using multiple message buffers and passing only a pointer instead of the whole message.

B. ROS 2

Much of robotics development, including the prototyping stage of autonomous-driving software, is done using the ROS 2 framework [3]. Community metrics indicate tens of thousands of developers worldwide [4] and the initial paper has been cited over 8,850 times according to Google Scholar. Within the automotive domain, we estimate that ROS 2 (or its predecessors ROS 1) is used by about 80% of all automotive OEMs and tier-1 suppliers working on autonomous vehicles.

The main advantage of ROS 2 is its ease of use and its extensive ecosystem. By implementing against the common ROS 2 API, developers get access to a plethora of tools and packages that make it easy to develop, simulate, and test complex systems containing multiple actuators and sensors.

To achieve this code reuse, ROS 2 enforces a modular system structure. A ROS 2 system consists of multiple *nodes*, which communicate through a *publish-subscribe* middleware (usually an implementation of DDS [5]). A publish-subscribe middleware allows a node to publish messages on one or more *topics*, which may be created dynamically at runtime. Other nodes may *subscribe* to such a topic to receive all messages published to that topic. Crucially, this design decouples sender and receivers, such that senders do not know the number or identity of the receivers and vice versa. In ROS 2, receivers register a *callback function*, which is triggered every time a new message arrives.

ROS 2 runs these callbacks through dedicated *executor threads*. Each executor thread is assigned one or more nodes by the system developer. The thread is then responsible for monitoring the relevant ROS 2 topics and running the appropriate callbacks whenever a message arrives. We refer to Blass et al. [6] for further details on the executor.

III. Challenges in Applying ROS 2 to Automotive Real-Time Systems

At Apex.AI we develop Apex.OS, a fork of ROS 2 that aims to turn ROS 2 from a prototyping system into an automotive grade framework. Besides the refactoring and extensions that enabled safety certification according to the ISO 26262 standard, one of the main challenges is to meet the performance constraints of automotive real-time systems. Despite the ever-increasing processing power of automotive ECUs, these performance constraints can be demanding as computation demands keep increasing as well. At least on the dual-core or quad-core processors common today, an increased middleware overhead therefore cannot be ignored.

In a typical deployment, only one CPU core would be available for a perception pipeline. A ROS 2 implementation of the pipeline would then consist of a graph of nodes. The first node could be triggered by a sensor at a rate of 30 Hz; each node performs operations in the message callbacks and forwards the results to one or more subsequent nodes. These pipelines can consist of over 50 publishers and 100 subscribers, each of which processes and transfers messages ranging from a few bytes to several megabytes.

Implementing such an architecture in frameworks like ROS 2 encounters two main performance pitfalls:

a) Pitfall 1: copy overhead: As discussed in section II-A, many existing automotive software architectures come from an environment where ECU internal communication costs are negligible. This is no longer the case on POSIX systems. When nodes are distributed across multiple processes, the per-process address spaces make it impossible to share data through global variables or by passing pointers. Instead, messages are sent through the loopback network interface or other POSIX IPC methods, which requires the message to be copied multiple times: from the sender to the middleware buffer, from the middleware buffer to the kernel's networking buffers, and finally from the receiver's networking buffers to the middleware buffers. If the APIs are not carefully designed, they may require copies because the lifetime of message references is not guaranteed beyond the scope of a method call.

All those copies require significant amounts of execution time, particularly if each iteration of the pipeline comprises over 100 message transmissions. As the time it takes to copy messages increases the overall latency, this becomes a severe problem in autonomous driving pipelines that have to deal with large messages like images or point clouds.

b) Pitfall 2: context switching overhead: To fully exploit memory isolation and the decoupling provided by the middleware, it is tempting to separate each node into its own process. This isolates the components from each other and allows them to be updated independently. Unfortunately, this also maximizes the number of messages that have to cross process borders. In the worst case, each message will wake up a thread in another process to do the processing. Although a single operating system context switch usually has a negligible

cost, the combined delay of all context switches with the additional logic in the framework and middleware layers can reach an appreciable fraction of the total pipeline cost.

To make matters worse, many DDS implementations spawn additional communication threads for each process. As a consequence at least two threads are involved for processing a message in the receiving node, a DDS middleware thread and the executor thread that runs the callback. The perception pipeline above with 100 subscribers thus incurs over 200 context switches in each iteration of the pipeline.

The problem is exacerbated further by the design decision in ROS 2 to handle *all* incoming messages with callback functions, each of which is triggered as soon as a message arrives. This is convenient if the message needs to be processed immediately. However, if the message needs to wait for a later event, for example another message, the only way to save the incoming message for later is to store it within the ROS 2 node. Besides creating a second ad-hoc message cache on top of the optimized caches already provided by the middleware, the consequence is that these messages induce the full context switching and callback management overhead. In the worst case, such expensive processed messages are no longer relevant if the actual execution condition for the node is met.

In the following section, we will demonstrate the performance impact of the discussed pitfalls on a set of microbenchmarks and describe the extensions in ROS 2, Apex.OS and Apex.Middleware to reduce the described overheads.

IV. Extensions made in ROS 2, Apex.OS and Apex.Middleware

The following measurements were carried out with Apex.OS and Apex.Middleware on a Renesas R-Car V3H board running the QNX real-time operating system. The average latencies from sending a message via a publisher in one node to processing the message with a subscriber in another node were measured for different message sizes. The two nodes run either in two different processes (inter-process) or after another in one thread (intra-thread). We compare three different versions of Apex.Middleware which is based on Eclipse Cyclone DDS™ [7] and Eclipse iceoryx™ [8].

1) *No optimizations:* Eclipse Cyclone DDS is used without a shared memory transport which means that messages are transferred via the loopback interface.
2) *Zero-copy transport:* Eclipse Cyclone DDS is used together with Eclipse iceoryx as shared memory transport.
3) *Zero-copy transport and reduced number of threads:* Apex.Middleware version optimized for intra-host communication. In addition to using a shared memory transport, additional threads and context switches in the data path are avoided.

A. Leveraging zero-copy communication

ROS 2 provides an efficient intra-process communication mechanism that enables zero-copy communication when the API is used with C++ unique pointers [9]. Since this functionality is limited to a single subscriber that is within the same process, additionally the so-called *loaned-message API* [10] was introduced to enable zero-copy communication for many subscribers and across process borders. The loaned-message API allows applications to write directly into the middleware buffer, saving the copy between application and middleware. In combination with a middleware implementation that uses a shared-memory transport for intra-host communication, messages can be transmitted without any copy whatsoever, leading to constant messaging overhead irrespective of the message size.

In Apex.Middleware, iceoryx was integrated as shared memory data transport which implements an end-to-end zero-copy approach from publishers to subscribers. POSIX shared memory segments are used that can be configured in terms of size and access rights. iceoryx provides APIs in various programming languages that can be used for polling or event-driven interaction. The publishers and subscribers can be dynamically connected and their queuing behavior can be configured. iceoryx thereby combines the efficiency of OSEK-style shared memory communication with the flexibility and convenience of POSIX systems.

As shown in *figure 1*, the inter-process communication latency could be reduced significantly with the introduction of a zero-copy transport. For small message sizes the copy overhead is negligible but when messages are in the megabytes range, the resulting latencies of several milliseconds when sending messages over the loopback interface would make it impossible to run a perception pipeline, as described in the previous section.

B. Avoiding context switches in the middleware

ROS 2 comes with a layered software architecture that clearly separates concerns and has interchangeable layers. This allows to develop parts like the middleware and the executor independently and to combine different executors with different middleware implementations. Unfortunately, this also leads to a software stack in which several threads are involved when a message is passed from the network interface to the user callback. Even when publisher and subscriber run in the same thread, additional middleware threads are involved in the data transfer. When optimizing Apex.Middleware for automotive real-time systems, we reduced the number of threads in the middleware and ensured that only the thread that is waiting or polling for data is involved in the processing of incoming messages. This was one of the most important measures to further reduce the communication latencies for all types of intra-host communication. (see *figure 1* and *figure 2*)

C. Executing multiple nodes in a single thread

On the framework side, both ROS 2 and Apex.OS reduce context switches by allowing users to multiplex multiple nodes onto a single executor thread. If, for example, two consecutive parts of a pipeline run in the same thread, the sender writes the data into a middleware buffer, hands it over to the middleware and returns to the main executor loop. The executor then notices that a message for the receiver arrived and invokes the receiver within the same thread context. With the before mentioned

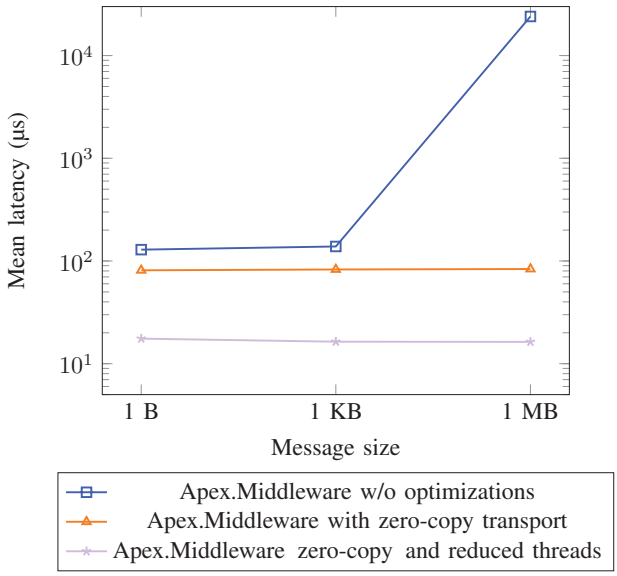

Fig. 1. Inter-process communication latency

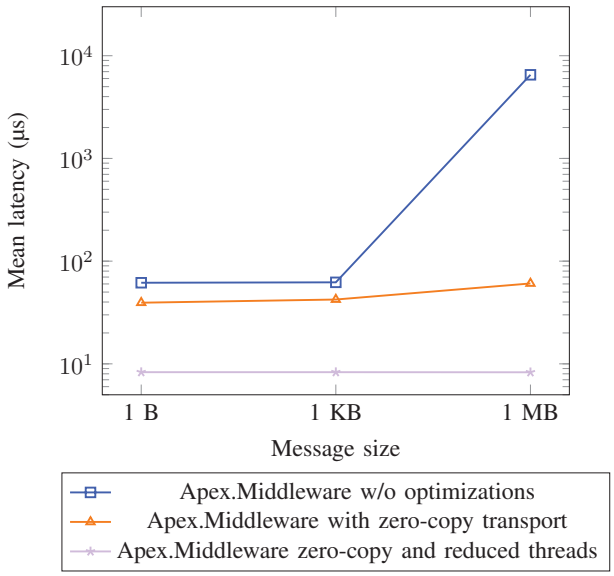

Fig. 2. Intra-thread communication latency

optimizations in Apex.Middleware, the entire transaction runs without any context switches or message copies.

While running all nodes of a pipeline on the same executor thread already improves efficiency over running each node in a separate thread, it still requires the middleware to detect the message arrival and to interact with the executor. To reduce the overhead of such transactions even further, the Apex.OS executor allows users to statically configure node chains at initialization time. When a node in the chain completes, the executor directly invokes the next node in the chain without even consulting the middleware. Using this technique, the switch between consecutive nodes runs again at the speed of a function call. The latencies in *figure 2* were measured with a setup of the Apex.OS executor that runs the nodes with the publisher and subscriber in a chain in a single thread. When comparing *figure 1* and *figure 2*, it becomes clear that executing as many nodes as possible in one thread and the associated avoidance of context switches is one way to reduce communication latencies.

For more complex pipelines, the Apex.OS executor also supports node graphs instead of just chains. This allows for more parallelization with a thread pool, while still retaining the benefits of a pre-defined and static execution order.

D. Identifying non-triggering topics

A node in a perception pipeline typically generates output by processing the input data received from multiple subscribers. The condition for starting the processing of new data is often the arrival of a new message at one specific subscriber of the node. For the other subscribers, it is sufficient to use only the latest available message or all messages that have been queued since the last node execution. The DDS middleware was designed for such use cases and provides caching of messages as well as the flexibility to decide on which events the user wants to react. With the ROS 2 default behavior, each subscriber comes with an individual message callback and even if the user doesn't have to react immediately to an incoming message, the arrival of each message triggers the callback and all of the discussed context switches.

Apex.OS avoids this problem by allowing the node to explicitly take messages out of the middleware queue when needed *but no earlier*. Instead of being triggered every time a message arrives, the nodes can distinguish between *triggering* and *non-triggering* topics. Only a message on a triggering topic activates the node; messages from non-triggering topics remain in the middleware buffers until they are explicitly requested by the node. This means that the number of context switches can be further reduced by only reacting on messages from triggering topics and not having callbacks for all incoming messages. When it comes to cyclic node execution, none of the received messages needs to be processed with an individual callback. The Apex.OS executor also allows optionally to define execution conditions that are evaluated whenever a message is received on a triggering topic. With the execution condition, the content of the message caches can be evaluated for all subscribers and a decision can be made as to whether the node should be executed or not.

V. Conclusions

Modern automotive systems increasingly move to powerful, centralized POSIX systems instead of the AUTOSAR Classic-based microcontrollers. Translating established software architectures directly to such systems often suffers from poor performance. We identified two major reasons for these performance issues: the increased cost of communication and the increased number of context switches.

To address these issues, our products Apex.OS and Apex.Middleware offer four techniques to avoid these pitfalls.

First, Apex.Middleware offers *zero-copy communication*. In combination with the ROS 2 loaned-message API, it allows applications to communicate even large messages at little cost.

Second, Apex.Middleware reduces the number of threads involved in communication, reducing the number of context switches during message transmission.

Third, Apex.OS provides an executor API that allows developers to run multiple related nodes in a predefined order within a single thread or thread pool. This allows Apex.OS to run complex pipelines of multiple related components in a defined order and with little context switching overhead.

Finally, Apex.OS distinguishes between triggering and non-triggering topics, and allows users to control the activation of nodes. This avoids needless context switches when nodes are activated before all relevant data has arrived and makes it easier to leave message caching entirely to the middleware.

REFERENCES

[1] "OSEK/VDX Version 2.2.3, Specification of OSEK OS," http://portal.osek-vdx.org/files/pdf/specs/os223.pdf, 2005.

[2] "AUTOSAR Release 4.2, Specification of Operating System," http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/system-services/standard/AUTOSAR SWS OS.pdf, 2014.

[3] "Robot Operating System," https://www.ros.org/, 2021.

[4] T. Foote, "ROS Community Metrics Report," http://download.ros.org/downloads/metrics/metrics-report-2020-07.pdf, 2020.

[5] "Data Distribution Service," https://www.dds-foundation.org, 2021.

[6] T. Blass, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ROS 2 Response-Time Analysis Exploiting Starvation Freedom and Execution-Time Variance," in *Proceedings of the 42nd Real-time Systems Symposium (RTSS)*, 2021.

[7] "Eclipse Cyclone DDS," Project website: https://cyclonedds.io/, 2021.

[8] "Eclipse iceoryx," Project website: https://iceoryx.io, 2021.

[9] "ROS 2 intra-process communication," https://docs.ros.org/en/galactic/Tutorials/Intra-Process-Communication.html, 2021.

[10] K. Knese, W. Woodall, and M. Carrol, "Zero copy via loaned messages," ROS design document, https://design.ros2.org/articles/zero_copy.html, 2020.