2019 IEEE Intelligent Transportation Systems Conference (ITSC)
Auckland, NZ, October 27-30, 2019

A Portable Implementation of the Real-Time Publish-Subscribe Protocol for Microcontrollers in Distributed Robotic Applications

Alexandru Kampmann, Andreas Wüstenberg, Bassam Alrifaee and Stefan Kowalewski

***Abstract*— This paper presents embeddedRTPS, a portable and open source implementation of the Real-Time Publish-Subscribe Protocol (RTPS). RTPS is the underlying protocol for the Data Distribution Service (DDS), which is a standardized middleware that allows for implementing distributed, loosely-coupled applications. DDS is not only the core protocol for the Robot Operating System (ROS) 2, but is also one of the few protocols that are part of the AUTOSAR Adaptive platform. In contrast to the available open-source RTPS implementations, embeddedRTPS is based on FreeRTOS and lightweightIP and targets resource-constrained embedded platforms. Our contribution allows microcontrollers to become independent, first-class participants in distributed automotive and robotic applications. We benchmark our implementation using a consumer-grade STM32 microprocessor as well as an ASIL-D certified, automotive microcontroller.**

I. INTRODUCTION

Upcoming automotive trends such as connected and self-driving vehicles challenge the traditional electric, electronic and software architectures prevalent in today's vehicles. According to [1] and [2], current automotive architectures can be characterized as Electronic Control Unit (ECU) centric. Each functionality is implemented as one software component that is executed on it's dedicated, purpose-built ECU. The ECUs are connected through a number of heterogeneous bus-systems, such as Controller Area Network (CAN), Local Interconnect Network (LIN) or FlexRay. The emerging technologies mentioned above require strong interconnections across different functional domains, which are hindered by the boundaries imposed by rigid system integration and non-interoperable bus systems. The computing power offered by traditional automotive ECUs is often insufficient for algorithms required for automated driving, e.g. compute intensive perception or control algorithms. Traditional automotive bus systems do not provide the throughput required for the large amounts of data produced by camera or LIDAR sensors [3].

As traditional automotive architectures and technologies are on the verge of disruption, various technologies such as Service-oriented Architectures (SOA) [2] and Ethernet are starting to enter the automotive domain. Ethernet not only provides the bandwidth required for camera and LIDAR sensors but allows software components to transparently communicate and exchange large amounts of data across a scalable vehicular network. SOA originated outside the automotive domain for the development and implementation of large, distributed software systems. This paradigm is centered around services, which are loosely-coupled software components that offer functionalities to other services or respectively consume functionalities of other services [4]. As services are integrated at runtime, this approach facilitates the implementation of modular and updatable software architectures. In order to achieve this, SOA relies upon an appropriate middleware for discovery, binding and data exchange between services.

A strong candidate for this middleware is Data Distribution Service (DDS), an open middleware specification maintained by the Object Management Group (OMG) [5]. DDS is designed for scalable, dependable communication in distributed systems and has been applied in different safety critical systems, such as air-traffic control, transportation systems and medical applications [6]. DDS is starting to enter the automotive domain, as it is one of the middlewares supported in the upcoming AUTOSAR Adaptive platform [7]. This upcoming AUTOSAR extension adds support for full-fledged POSIX-based ECUs and Ethernet communication. Besides being part of AUTOSAR Adaptive, DDS is also the underlying middleware for the Robot Operating System (ROS) 2.

DDS itself is an API specification and the actual underlying protocol is the Real-Time Publish-Subscribe Protocol (RTPS) [8]. Multiple open-source implementations of DDS are available for full-fledged computers running Linux or Windows. These implementations are unsuitable for resource-constrained embedded systems. Although high-level functions such as environment perception or path-planning require rich compute power, embedded platforms still play a vital role for hosting real-time demanding functions or safety-critical supervisory mechanisms.

We address this gap through embeddedRTPS, a portable implementation of the RTPS protocol for embedded platforms with scarce resources[1]. Our implementation assumes only the presence of lightweight IP (lwIP), a widely-used Ethernet network stack for embedded systems [9] and the FreeRTOS microcontroller operating system. We are aware that our implementation is not suitable for consumer-grade products. Nevertheless, we hope to enable researchers to augment their distributed architectures with embedded platforms for prototyping, research and educational purposes.

The remainder of this paper is structured as follows. Section III provides a brief overview of the RTPS protocol. Related work is presented in Section II. We introduce our

Alexandru Kampmann, Andreas Wüstenberg, Bassam Alrifaee and Stefan Kowalewski are with the Chair for Embedded Software, RWTH Aachen University, 52074 Aachen, Germany {kampmann, wuestenberg, alrifaee, kowalewski}@embedded.rwth-aachen.de

[1]https://github.com/embedded-software-laboratory/embeddedRTPS/

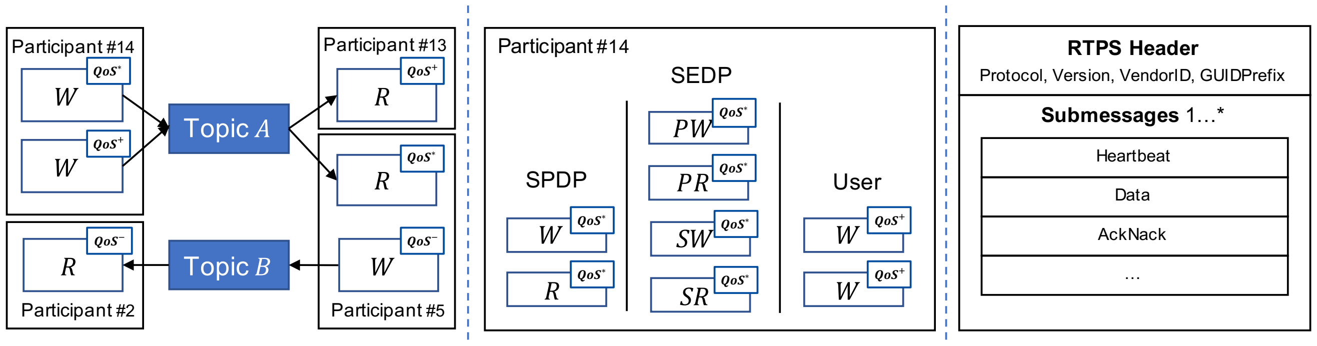

Fig. 1. **Left:** Participants Exchange Messages via User-defined Reader and Writer Endpoints. **Middle:** Built-in RTPS endpoints for Participant (SPDP) and Endpoint Discovery (SEDP) alongside user-defined Endpoints. **Right:** Components of RTPS Messages encapsulated in UDP Packets.

implementation in Section IV. The evaluation in Section V presents benchmarks of embeddedRTPS for two different microcontrollers that have no POSIX-based operating system, i.e. no Linux or Windows. Furthermore, we demonstrate the interoperability to an existing third party RTPS implementation.

II. Related Work

eProsima FastRTPS is a popular RTPS implementation that is also the underlying middleware for ROS 2 [10]. OpenDDS is another widespread implementation that is a DDS implementation on top of the RTPS protocol [11]. Real-Time Innovations (RTI) offers ConnextDDS, a commercial DDS implementation [12]. All of the implementations mentioned above target Linux, Windows or MacOS and do not run on embedded platforms. RTI also offers microDDS, an implementation suitable for microcontrollers. In contrast to embeddedRTPS, microDDS is not open source and only commercially available. Lastly, freeRTPS[2] is an implementation originally developed for microcontrollers in the context of ROS 2. This implementation explicitly targets STM32 microprocessors and is not under active maintenance.

The DDS Extremely Resource Constrained Environments (XRCE) standard is intended for integration of embedded systems into DDS networks [13]. The resource-constrained platform is integrated into the DDS network by a more powerful server, that acts as a gateway. In this architecture, the microcontroller is not an independent, first-class participant in the communication network, as it's ability to communicate depends on server. This also introduces a single point of failure, rendering this architecture potentially unsuitable for safety-critical applications.

III. Background

This section provides a brief overview of the basic concepts behind RTPS. We cover the basic entities, the discovery procedure and data exchange mechanisms. An exhaustive description is provided in the standard maintained by the OMG [8].

[2]https://github.com/ros2/freertps/

A. Entities

As depicted in Fig. 1, the basic actors in RTPS are participants, which in turn have a variable number of readers or writers. Readers or writers (summarized as endpoints in the following) can be identified with an id composed of a unique participant prefix and an entity id which is unique within this participant. They communicate through a publish-subscribe mechanism, a concept also common in other messaging frameworks such as ROS or Message Queuing Telemetry Transport (MQTT). Endpoints are loosely coupled only by a topic name and data type definition. From a user perspective, messages published by writers are addressed to a topic and not to specific recipients. Respectively, readers subscribe to a specific topic and join the network without knowledge about the specific writer(s). For this, participants become aware of each other during runtime in the course of a decentralized discovery process. Thus, readers and writers form a loosely-coupled communication network that is suitable for runtime integrated systems.

Various Quality of Service (QoS) parameters can be used to customize the behavior of endpoints. For example, writers and readers communicate either in a reliable or best-effort mode. Reliable writers keep track of already transmitted packages through sequence numbers and are able to resend messages in case of a transmission failure. Best-effort writers do not provide this capability. The user interacts with endpoints by passing serialized data to writers or receiving data from readers. RTPS is specified for TCP/UDP communication but our implementation currently supports only UDP.

In general, endpoints send UDP packets that are structured as depicted in Fig. 1. Each message starts with a header that contains, among other fields, the RTPS version or vendor identification. The header is followed by a variable amount of submessages that are used for data exchange and discovery. `Data` submessages are used to exchange generic, serialized data between readers and writers. `Heartbeat` and `AckNack` submessages are used to provide a reliable communication. The `Heartbeat` submessage allows writers to announce the available sequence numbers in its storage. Reliable readers send `AckNack` submessages to communicate missing as well as received packets. Based on those acknowledgments, the reliable writer can decide when a packet can be safely removed from its history. While one or multiple participants are addressed by a specific port, the target endpoint is identified by the entity id passed with the aforementioned messages.

B. Discovery Protocols

Entities in RTPS discover each other through a decentralized two-step discovery process, using a set of six built-in endpoints as depicted in Fig. 1. Technically, these are regular endpoints that function just like user-defined endpoints but communicate specific messages for the discovery process and have a fixed entity id. In the first step, the Simple Participant Discovery Protocol (SPDP) uses a reader/writer pair of best-effort endpoints for discovery of participants and their properties in the network. Each participant periodically sends messages through the SPDP-writer containing information about the participant on standardized IP broadcast addresses. The transmitted UDP packets are made up of RTPS packets containing a `Data` submessage and are received by remote SPDP-readers. Discovery of those endpoints is not required because the messages are send on addresses which are relevant for all participants and the destination is known as the built-in endpoints have fixed entity ids. SPDP messages contain, among others, the unicast or multicast locators (i.e. IP-address and port) for meta (e.g. discovery) and user traffic to the sending participant.

Once the participants become aware of each other, they exchange information about user-defined endpoints using the Simple Endpoint Discovery Protocol (SEDP). For this, reliable writers and readers are used. The publication writer PW sends information about user-defined writers to all known participants by addressing the publication readers PR of all known participants. For each user-defined endpoint, information about it's QoS parameters, locators and their entity id are transmitted. Analogously, user-defined readers are announced through the subscription-writer SW and participants receive these respective messages through the subscription-reader SR. After discovering a new endpoint, a participant tries to match it with one if its own endpoints. Endpoints are matched if the topic name, data type name and QoS parameters are compatible.

Once local endpoints have been matched with remote endpoints, every message that is handed to the writer will be send to all known, matching readers in the network. For this purpose, the user hands over a serialized byte array, that is inserted into RTPS packet containing a `Data` submessage. Note that SPDP and SEDP messages are sent periodically, therefore participants and endpoints can dynamically join the RTPS network.

IV. EMBEDDED RTPS

This section introduces our portable implementation of the RTPS protocol. We first discuss general design decisions of our implementation, followed by a brief overview of our software architecture. Afterwards, the mechanisms involved for transmission and reception of user messages is discussed.

A. Design Decisions

We target resource-constrained embedded systems that run with a minimal operating system and are a potential host to applications with real-time constraints. In order to cater to the specific requirements of these platforms and applications, we made various design decision in the implementation of embeddedRTPS.

First, we decided to use C++ for implementation but we refrain from using runtime type information (RTTI) to reduce overhead.

Second, we avoid dynamic memory allocation of variable block sizes. Any memory allocation performed by the C++ runtime only occurs during an initialization phase. Thus, a memory allocation failure will be caught before the actual execution of critical code. We do not use default C++ containers and algorithms which use dynamic memory allocation and use statically allocated data structures only.

Third, we make use of the statically allocated memory pool provided by lwIP for memory management. These memory pools consist of equally-sized memory chunks, which makes them suitable for embedded and real-time applications [14]. In contrast, dynamic memory allocation with variable block sizes can lead to fragmentation, rendering them unsuitable for embedded systems. Additionally, making use of lwIP memory management allows us to avoid necessary memory copies, which will be further elaborated below. This approach allows to determine memory usage at compile time, which is crucial for many embedded systems.

Furthermore, we do not make use of exceptions. According to [15], standard implementations of exception handling can increase the size of the resulting binary by up to 15%. Additionally, exceptions can lead to loss of determinism of execution time and memory consumption [16].

Finally, we ensure that our implementation is testable without introduction of additional overhead. We avoid using runtime polymorphism, that is popular for mechanism for mocking or replacing classes. Instead, we make use of template mechanisms which can be resolved at compile time.

B. Architectural Overview

An overview of our architecture is provided in Fig. 2 and will be elaborated in the following. At the lowest levels, FreeRTOS and lwIP are the basis of our implementation. FreeRTOS provides a real-time scheduler and thread synchronization while lwIP provides the IP stack for transmission of UDP packets. Our implementation assumes the presence of these components as an abstraction to the underlying hardware. More specifically, we only access functions for thread management and synchronization through the lwIP abstraction layer, which will potentially allow for replacement of FreeRTOS with other suitable operating systems. Ports of both lwIP and FreeRTOS are available for a multitude of embedded platforms.

The thread pool decouples both the low-level network stack and user threads from embeddedRTPS. On the one hand, processing messages in the context of lwIP threads or interrupts could block the network stack, thus leading to

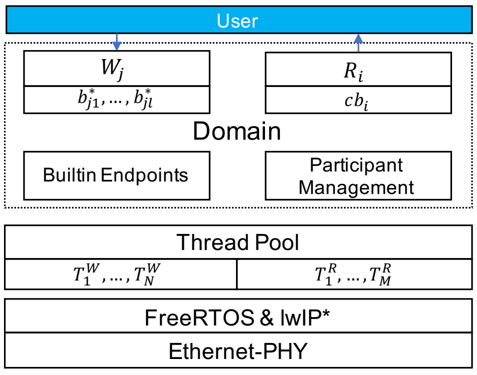

Fig. 2. Software Architecture Overview.

unpredictable timing behavior and potential loss of network packets. This is especially critical if the thread executes user-space functions that can consume unpredictable amounts of time. On the other hand, fully traversing the network stack in the context of user-space threads could impact timing of high-level functionalities. Therefore, the pool consists of two types of worker threads, allowing for asynchronous transmission. The writer pool $P_W = \{T^W_1, \ldots, T^W_N\}$ consists of N threads that handle outgoing user space messages by invoking the low-level API of lwIP for the transmission of UDP packets. The reader pool $P_R = \{T^R_1, \ldots, T^R_M\}$ consist of M threads that handle UDP packets coming from the lwIP layer. The main component of embeddedRTPS is the domain object, which creates and manages all participants and their readers and writers. At the top-level, the user directly interacts with the readers and writers by receiving data through callbacks or passing serialized messages to the writer.

Before setting up readers and writers, the user first requests a participant instance. We have implemented both the SEDP and SPDP discovery protocols mentioned in Section III and each participant is equipped with the built-in endpoints required. Participants are created through the domain object, which serves as a factory using statically allocated memory. The obtained participant is the handle for the user to create further readers and writers. The interaction with a reliable and a best-effort endpoint is identical from user perspective. We will now cover the message processing mechanisms and interactions between the different layers in more detail. In the provided example, we assume the user has created a writer W_j and reader R_j.

C. Processing Incoming & Outgoing Messagesd

For incoming messages, we assume that the user has created a reader R_j with topic and data type specification and also registered a user-space callback cb_i for receiving incoming messages. Furthermore, we assume that this reader has matched up with a remote writer, which has published a `Data` message that is now received as an UDP/IP packet through the Ethernet interface. The reception mechanism of UDP/IP packets at the lowest level is handled by lwIP and the details are specific to the underlying hardware. Upon the reception of a UDP packet, the thread pool receives a callback invoked by the lwIP-layer. This callback passes a reference to a buffer object b^*_i containing the incoming payload to the domain object. In order to avoid lengthy blocking of the lwIP thread, this callback does not carry out further message parsing and processing. Instead, the reference b^*_i is stored in a input ring buffer that is part of the domain. Once the referenced has been stored, we use a semaphore mechanism to notify a reader thread $T^R_i \in P_R$ for further processing and immediately return from the low-level lwIP thread. The thread will wake up and process b^*_i from the input ring buffer. Submessages of type `Data` are passed to the user callback cb_i based on the entity id contained in the message. Once the user-space callback cb_i returns, the memory buffer b^*_i is finally released. Note that we never copy the variable-length payload of incoming `Data` messages, i.e. the b^*_i buffer of lwIP is never copied. Thereby, we reduce memory consumption and increase performance and timing predictability.

Additionally, the reader threads are also responsible for handing `Heartbeat` submessages to readers, which allows reliable readers in our implementation to keep track of potentially missed packets. `ACKNACK` submessages are also delivered by reader pool threads to stateful writers. Requesting or retransmitting missing packets is handled internally in our implementation and does not require user intervention.

For outgoing messages, we assume that the user has created a writer W_j for a specific topic and data type. Users hand a pointer to a `uint8_t` byte buffer which our implementation transmits to all matched remote readers. Internally, the history H_j of writer W_j is implemented as a ring buffer consisting of l low-level lwIP buffer $b_{j1}, \ldots, b_{jl}$. First, we request a data buffer b_{jk} from the lwIP layer for the aforementioned history. Then, the user data is copied into b_{jk}, assigned a sequence number and placed into the history. During this step, the oldest packet of the history will be overwritten. In contrast to incoming messages, we decided against a zero-copy implementation in favor of asynchronous transmission. Alternatively, either the user would have to preserve the byte buffer for unspecific amount of time until lwIP finishes transmission of b_{jk} or the user would be blocked until the transmission is finished. Nevertheless, our implementation can easily be extended for optional synchronous processing in the future. As the final step executed in the context of the user, a writer thread $T^W_i \in P_W$ is signaled to wake up and the call returns to user space. The writer thread $T^W_i \in P_W$ then carries out the transmission of b_{jk} by invoking the respective lwIP APIs. Note that the buffer b_{jk} is not released at this point, in order to handle potential requests for retransmission in case of reliable writers. Instead, b_{jk} is released when the head of the history ring buffer reaches the location of b_{jk} the next time. Reliable writers have an additional thread that periodically transmits `Heartbeat` messages containing the sequence numbers of available packets with a configurable frequency.

D. Testing Concept

Our implementation comes with a suite of unit tests that cover large parts of the system. We deem this helpful for further test-driven development or regression testing. This test suite can be executed on a desktop system with a Linux or Windows, for which lwIP ports exists, rather than on the actual target embedded system, allowing for more convenient and hardware-independent development. We use GoogleTest for implementing the unit tests and currently provide over 100 tests that can be executed automatically.

E. Limitations

There are multiple limitations in embeddedRTPS. First, the standard defines various QoS policies, but we only provide support for reliable and best-effort endpoints yet. Second, our implementation currently does not make use of UDP multicast communication. This can lead to sub-optimal performance when a writer addresses multiple endpoints on different remote machines, as data is sent to each recipient individually. Lastly, the total RTPS message size is limited by the size of individual lwIP buffers. This limitation is due to the fact that we currently do not process messages that are spread across multiple lwIP buffers.

V. EVALUATION & DISCUSSION

This section provides evaluation results for various benchmarks using different embedded platforms. We demonstrate both portability to two different embedded platforms and interoperability with an existing, wide-spread RTPS implementation.

A. Latency Benchmark

This section presents a benchmark of end-to-end latencies in our software stack. Our setup consists of two Infineon Aurix TC277 microcontrollers, which are TriCore-based, ASIL-D certified safety-microcontrollers for automotive applications. Both units are directly connected through Ethernet. We measure the Round-Trip-Time (RTT) of packets with varying payload sizes. Timing is measured with a logic analyzer that is connected to the General Purpose Input Output (GPIO) pins of the microcontrollers. These GPIO pins indicate events in our software stack with minimal implementation overhead. Additionally, the digital analyzer allows for timing measurements with high precision. The process is detailed in Fig. 3. Timing is measured at the measurement unit (MU), which is also the initiator of the communication. The responder unit (RU) returns received packets back to the MU as quick as possible. Both units have one writer and one reader each and we start our latency test after the discovery process has been carried out. The MU raises Pin_{RTT} to high to indicate start of transmission. Then, a data packet from the user level of the MU is handed to an best-effort writer. This packet traverses our implementation stack and is handed to the lwIP layer by a worker thread. The packet is received by the network stack of the RU, traverses our implementation and handed to the reader. This packet is copied to a local buffer in the user level. Finally, the data

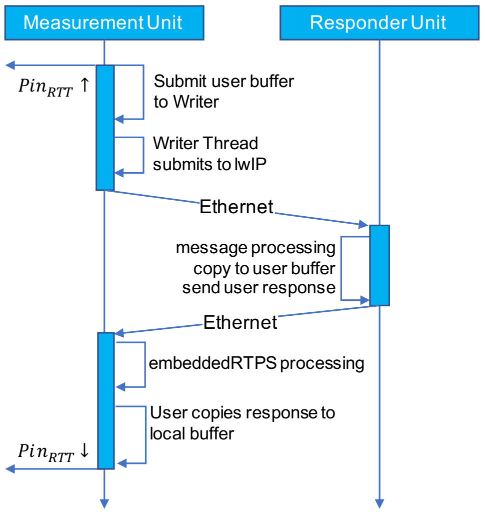

Fig. 3. Latency Measurement Setup

from the local buffer is send back with a best-effort writer to the MU. The packet is received at the MU and passed to the user callback, where it is copied again to a local buffer at the user level. Finally, Pin_{RTT} is pulled back to a low potential to indicate end of measurement. We repeat this procedure 10000 times for varying payload sizes. The largest packet size in our experiment allows to encode 128 single precision IEEE-754 floating point numbers. In order to estimate the overhead added by our RTPS implementation, we also report the RTT of raw data being send via UDP packets as a baseline.

Maximum, minimum and various quantiles of the experiment outcome are displayed in Table I. According to [17], the strictest requirements for end-to-end delays in automotive control applications are $2.5ms$. For the largest packet size in our experiment, the maximum end-to-end delay can be approximated as $2088\mu s \times 0.5 = 1044\mu s$. The worst-case maximum delay measured lies well below the threshold above. It is also worth mentioning that the differences observed between the minimum round-trip-times and the 99%-quantile are only double-digit microseconds, which emphasizes the predictability of our implementation. Comparing to the UDP baseline, our RTPS layer adds an average of $553.5\mu s$ of additional overhead to the RTT.

B. Portability and Interoperability

In this experiment we demonstrate both the portability of our implementation using a different microcontroller as well as the interoperability with an existing implementation of the RTPS protocol for full-fledged computers. Our setup consists of an Intel NUC (NUC7i5BNK) as well as an a STM32-F767ZI board with an Arm Cortex-M7 processor. The Intel NUC runs Ubuntu that has been outfitted with the `PREEMPT_RT` Linux real-time patch. While the microcontroller runs embeddedRTPS, the Intel NUC runs FastRTPS, a popular RTPS implementation for Linux and Windows computers [10]. This RTPS implementation is currently the

TABLE I

V-A: RTTs for Various Packet Sizes with Infineon Aurix

Results for Raw UDP packets (baseline)

Bytes	**Samples**	**Min** [μs]	**50%** [μs]	**99%** [μs]	**Max** [μs]
32	10000	188	189	194	239
64	10000	202	203	208	213
128	10000	250	250	255	298
256	10000	300	301	305	353
512	10000	431	433	436	436
1024	10000	692	695	700	747

Results for RTPS messages

Bytes	**Samples**	**Min** [μs]	**50%** [μs]	**99%** [μs]	**Max** [μs]
32	10000	596	624	655	1181
64	10000	622	649	679	1307
128	10000	677	705	733	1414
256	10000	779	804	840	1496
512	10000	984	1012	1048	1512
1024	10000	1388	1420	1464	2088

TABLE II

V-B RTTs for Various Packet Sizes with STM32 and Intel NUC

Bytes	**Samples**	**Min** [μs]	**50%** [μs]	**99%** [μs]	**Max** [μs]
32	10000	752.00	1046.00	1159.00	1624.00
64	10000	775.00	1067.00	1192.00	1939.00
128	10000	860.00	1259.00	1369.00	2369.00
256	10000	850.00	1172.00	1251.00	2453.00
512	10000	838.00	1270.00	1382.00	2336.00
1024	10000	1191.00	1524.00	1621.00	2720.00

default middleware for ROS 2. The NUC functions as the RU while a Nucleo STM32-F767ZI board is utilized as the MU. We repeat the same procedure as described in the previous benchmark in Section V-A. The results of this experiment are displayed in Table II. The maximum latencies are higher than in the previous experiment but still below the $2.5ms$ threshold cited above. We have not pinpointed the source of increased latencies in comparison to the Infineon Aurix. We speculate that either the Linux network driver adds overhead or the lwIP port for the STM32 is less efficient in interacting with the physical layer. Since the purpose of this experiment is to demonstrate portability and interoperability, we did not further investigate.

C. Memory Consumption

In this section we report the memory consumption for the benchmark presented in Section II. In this setup, we a have two user endpoints as well as six endpoints for discovery (c.f. Section III). The flash memory consumption for lwIP and FreeRTOS alone is $46.5\,\mathrm{kB}$. Including embeddedRTPS adds another $24.5\,\mathrm{kB}$, resulting in a total flash memory consumption of $71\,\mathrm{kB}$. Thus, our implementation consumes only 3.55% of totally available $2\,\mathrm{MB}$ flash memory. Additional heap memory is reserved during runtime for the domain object ($7\,\mathrm{kB}$), the stacks for embeddedRTPS threads ($5.5\,\mathrm{kB}$) as well as for the user thread ($0.8\,\mathrm{kB}$). This results in $13.3\,\mathrm{kB}$ of SRAM consumption attributed to embeddedRTPS. The total SRAM allocation is $\sim 47\,\mathrm{kB}$ and is dependent on the FreeRTOS and lwIP configuration.

VI. Future Work

The work presented in this paper is a rudimentary RTPS implementation and we hope that it can serve as a starting point for future extensions as source code is made available under the MIT license. The RTPS standard offers various QoS policies, of which we currently have only implemented the reliable endpoints. This leaves room for further extensions in the future. Furthermore, we plan to improve memory footprint and overall performance and the introduction of more configuration parameters will allow for further customization. Finally, embeddedRTPS could potentially serve as the basis for making the ROS 2 framework available for microcontrollers.

Acknowledgment

This research is accomplished within the project "UNICAR*agil*" (FKZ EM2ADIS002). We acknowledge the financial support for the project by the Federal Ministry of Education and Research of Germany (BMBF).

References

[1] M. Broy, I. H. Kruger, A. Pretschner, and C. Salzmann, "Engineering Automotive Software," *Proceedings of the IEEE*, vol. 95, pp. 356–373, Feb 2007.

[2] S. Kugele, P. Obergfell, M. Broy, O. Creighton, M. Traub, and W. Hopfensitz, "On Service-Orientation for Automotive Software," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 193–202, April 2017.

[3] A. Camek, C. Buckl, P. S. Correia, and A. Knoll, "An Automotive Side-View System Based on Ethernet and IP," in *2012 26th International Conference on Advanced Information Networking and Applications Workshops*, pp. 238–243, March 2012.

[4] C. Farcas, E. Farcas, I. H. Krueger, and M. Menarini, "Addressing the integration challenge for avionics and automotive systems—from components to rich services," *Proceedings of the IEEE*, vol. 98, pp. 562–583, April 2010.

[5] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pp. 200–206, IEEE, 2003.

[6] Object Management Group, "Who's Using DDS?." `https://www.omgwiki.org/dds/who-is-using-dds-2/`, 2019. [Online].

[7] S. Fürst and M. 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)*, pp. 215–217, IEEE, 2016.

[8] Object Management Group, "DDS Interoperability Wire Protocol." `https://www.omg.org/spec/DDSI-RTPS/2.2/`, 2014. [Online].

[9] A. Dunkels, "Design and Implementation of the lwIP TCP/IP stack," *Swedish Institute of Computer Science*, vol. 2, p. 77, 2001.

[10] eProsima, "FastRTPS." `https://www.eprosima.com`, 2019. [Online].

[11] OpenDDS, "OpenDDS." `https://www.opendds.org`, 2019. [Online].

[12] Real-Time Innovations, "ConnextDDS." `https://www.rti.com/`, 2019. [Online].

[13] Object Management Group, "DDS for Extremely Resource Constrained Environments." `https://www.omg.org/spec/DDS-XRCE/1.0/Beta2`, 2019. [Online].

[14] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*. CRC Press, 2003.

[15] J. Renwick, T. Spink, and B. Franke, "Low-cost Deterministic c++ Exceptions for Embedded Systems," in *Proceedings of the 28th International Conference on Compiler Construction*, pp. 76–86, ACM, 2019.

[16] J. Lang and D. B. Stewart, "A Study of the Applicability of Existing Exception-Handling Techniques to Component-based Real-Time Software Technology," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 20, no. 2, pp. 274–301, 1998.

[17] R. Steffen, R. Bogenberger, J. Hillebrand, W. Hintermaier, A. Winckler, and M. Rahmani, "Design and Realization of an IP-based In-car Network Architecture," in *1st International ICST Symposium on Vehicular Computing Systems*, 2010.