Online latency monitoring of time-sensitive event chains in ROS2

Jonas Peeck, Johannes Schlatow and Rolf Ernst
Institute of Computer and Network Engineering
TU Braunschweig
Braunschweig, Germany
{peeck,schlatow,ernst}@ida.ing.tu-bs.de

***Abstract*—Highly-automated driving involves chains of perception, decision, and control functions. These functions involve data-intensive algorithms that motivate the use of a data-centric middleware and a service-oriented architecture. As an example, we use the open-source project Autoware.Auto, which bases on the Robot Operating System (ROS) 2. Often, function chains define a safety-critical automated control task with weakly-hard real-time constraints. Providing the required assurance by formal analysis, however, is challenged by the complex hardware/software structure of these systems. We therefore propose an approach that combines measurement, suitable distribution of deadlines, and application-level online monitoring that serves to supervise the execution of service-oriented software systems with multiple function chains and weakly-hard real-time constraints. We further evaluate our proof-of-concept implementation in ROS2 on an environment perception use case from Autoware.Auto.**

I. Introduction

The Robot Operating System (ROS) is one of the most popular (open-source) frameworks for building complex software in robotics. In particular, ROS2 (successor of ROS1) is focusing on establishing a service-oriented architecture by centering the framework around existing Data Distribution Service (DDS) middlewares. This concept increases the modularity, reusability and flexibility of the software system.

One of the basic functionalities of a robotic system is to perform environment perception using cameras, lidars or other sensors. This is also a functionality that we observe in the automotive domain more frequently as it is required for achieving highly-automated driving. Interestingly, the automotive domain takes a similar path by standardizing middleware-centric architectures based on DDS and SOME/IP [1]. Furthermore, there are even projects such as Autoware.Auto [2] that build their software stack on ROS2 to tackle autonomous driving use cases. In this paper, we use the perception stack from the Autoware.Auto project as depicted in Fig. 1 as a running example. It shows an environment perception system with two lidars (front and rear). The blue boxes correspond to single-threaded processes that subscribe to DDS topics and publish DDS topics on their own. Here, we also denote these processes as services or ROS nodes. The lidar data is sent periodically as DDS topics to the fusion service on Electronic Control Unit (ECU) 1, which joins the data (based on their timestamps) and publishes a DDS topic comprising a point cloud. The classifier service on ECU 2 subscribes to this

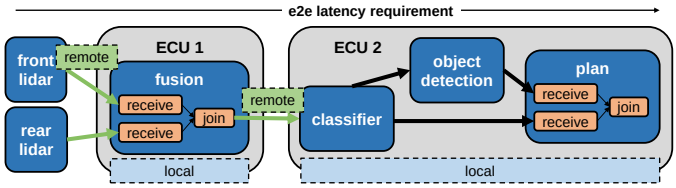

Fig. 1: Autoware.Auto perception stack example.

topic and therefore receives the point-cloud data via the DDS middleware. It classifies the data into ground points and non-ground points, which are both published as separate topics. The non-ground points are used by the object-detection service that uses a clustering algorithm to publish the bounding boxes of detected obstacles. Together with the ground points, the detected objects are received by the plan service that decides on the trajectory to be followed by the vehicle.

In conjunction, all services form a processing chain or event chain for which we must meet certain timing/performance constraints. On the one hand, we can assume a throughput requirement in form of a minimum frames per second that must be achieved. On the other hand, we also have an end-to-end latency requirement. The latency requirement is important because computing a trajectory on the basis of lidar data that was recorded too far back in the past will render the resulting trajectory unusable. Especially when used in the context of highly-automated or autonomous driving, the performance of such event chains become essential for the correct and safe operation. A latency requirement in this scenario is not necessarily a hard requirement but rather a weakly-hard requirement, i.e. occasional violations are tolerable if they do not occur too often. Although the formal analysis of hard and weakly-hard latency requirements is meanwhile well investigated [3], its practical application is challenged by the increasing complexity of software frameworks such as ROS2 and of high-performance architectures, which optimize for average-case throughput rather than worst-case predictability. Furthermore, as the impact of latency violations likely depends on the current driving situation, the severity of a violation cannot be predicted analytically. In this paper, we present a possible solution that consist in runtime mechanisms for detecting latency violations and in involving the application in handling the detected violations appropriately.

https://doi.org/10.24355/dbbs.084-202101271521-0

The paper is structured as follows: We resume related work in Section II. Thereafter, we introduce our monitoring concept in Section III, by defining the system model, its basic mechanisms as well as proposing a trace-based budgeting approach. In order to show the applicability of the monitoring to state of the art systems, we describe a lightweight implementation within the Autoware.Auto platform in Section IV. The evaluation of the implementation is then done in Section V. Last, we draw a conclusion in Section VI.

II. Related Work

Runtime behaviour of computing systems can be analysed by many existing tracing frameworks [4]–[6]. These frameworks preliminary aim at an offline evaluation as they gather a high volume of trace data. In this work, we focus on online monitoring instead, that is capable of timely reactions to latency violations. Online monitoring, also known as runtime monitoring or runtime verification is a long known concept for providing assurance in critical domains. By continuously observing critical properties, protective or reactive countermeasures can be taken. Early approaches aim to validate the correctness of execution of real-time systems at runtime [7] or target to protect critical system functions by early violation detection and corrective action [8]. In the automotive domain, monitoring is applied to limit the execution time of tasks and thereby achieve time protection between different criticality levels [9], [10]. Another approach for isolating tasks or groups of tasks is to monitor the arrival patterns of arbitrary tasks as well as their arrival workload at runtime [11], [12]. Several other framework evolved around runtime monitoring concepts, like [13] or [14]. An overview of different monitoring techniques for safety-critical multicore systems is provided in [15].

Yet, existing monitoring mechanisms are typically restricted to local processing delays or single communication delays. For instance, AUTOSAR Adaptive Platform (AP) supports checkpointing [16] for observing the time between certain progress points within the same application. However, it does not sufficiently include communication latencies between processes. The latter become particularly important in communication-centric architectures as mentioned in the introduction. Therefore communication latencies must not be neglected, especially with the increasing safety requirements involved with the autonomous-driving vision. A possible approach is taken by DDS, which employs a Quality of Service (QoS) mechanism to supervise message deadlines (i.e. their inter-arrival time). The combination of local response-time monitoring and monitoring of message deadlines could, in principle, establish a monitoring of end-to-end latencies by splitting an event chain into separately monitored segments. Yet, special care must be taken to leave no unmonitored gaps, i.e. to ensure that all parts of the event chain are monitored. In this paper, we therefore present a monitoring approach that observes communication events which are already exposed by the API and naturally result in segmentations without gaps. Moreover, by allowing individual segments to cross process boundaries, we can reduce the number of monitored segments and thereby simplify the assignment of local deadlines.

A short version of this paper is published in [17].

III. Monitoring Concept

In the previous sections, we have motivated the need of latency monitoring of end-to-end event chains. We now develop our decentralized monitoring concept that bases on a segmentation of event chains into local and remote segments, which are individually monitored. The segmentation of our example is indicated in Fig. 1 and explained in further detail below. With the decentralized approach, we aim at efficient implementation of monitoring mechanisms without relying on additional, and potentially unreliable, network communication.

For an event chain c, we have several performance requirements: Since all segments execute concurrently, the throughput requirement (minimum frames per second) translates to a maximum latency B^c_{seg} per segment. In consequence of the throughput requirement, it is reasonable to assume that the event chain is triggered periodically with a fixed period. The end-to-end latency requirement is denoted by B^c_{e2e}. We further assume a weakly-hard (m, k) constraint for the latency requirement, which denotes that we can tolerate up to m latency violations of B^c_{e2e} within k consecutive executions [18].

Given a fixed segmentation of an event chain, we now develop a decentralized monitoring concept such that the performance requirements are met or violations are detected and reacted to in time. In the remainder of this section, we first clarify the terminology before we elaborate on how latency violations are handled and how we can determine individual segment deadlines from the given performance requirements.

A. System model and terminology

An event chain c is defined as a sequence of segments $s_i \in S^c$, where i denotes the position of the segment in the chain. A segment s_i starts with a communication event $e^{s_i}_{st}$ and ends with a communication event $e^{s_i}_{e}$. Let $t^{s_i}_{st,n}$ and $t^{s_i}_{e,n}$ denote the timestamps of the n-th start event and end event of s_i respectively. We can further assume an in-order delivery of middleware messages such that the n-th observed start/end event corresponds to the n-th activation/completion of the corresponding segment. The start and end events of a segment must have the same rate. Note, that a system may comprise an arbitrary number of event chains. Moreover, an event (and its corresponding segment) can be involved in multiple event chains.

In Fig. 2, e.g., there are two event chains which share all but the first two segments and are activated synchronously with the same period. Asymptotically, the activation rates of both chains must be the same to avoid an ever increasing event backlog at the join in the fusion service. As we monitor communication events of the middleware, we focus on two observable event types by which the segments are delimited: publication events and receive events. Conceptually, there are no gaps between segments such that $e^{s_i}_{e} = e^{s_{i+1}}_{st}$, which

https://doi.org/10.24355/dbbs.084-202101271521-0

includes $t_{e,n}^{s_i} = t_{st,n}^{s_{i+1}}$. A local segment s^l starts with a receive event and ends with a publication event on the same ECU, whereas a remote segment s^r starts with a publication event and ends with a receive event on another ECU. By maximizing the length of the local segments, an event chain is described by an alternating sequence of remote and local segments.

The latency requirement of an event chain c is expressed by its latency budget B_{e2e}^c, which is allowed to be exceeded up to m out of k consecutive executions. As we follow a decentralized monitoring approach, the budget is divided among the corresponding segments, so that the following equation must hold:

$$B_{e2e}^c \geq \sum_{s_i \in S_c} d^{s_i} \tag{1}$$

Here, d^{s_i} denotes the deadline of segment s_i, i.e. the maximum time between the n-th start event of a segment and the corresponding end event.

B. Temporal exceptions and propagation mechanisms

The basic idea of latency monitoring is to continuously observe communication events at runtime and to raise *temporal exceptions* whenever an end event does not occur in time. These exceptions are then handled by the application itself or by a system-level entity to perform further diagnostics or take appropriate countermeasures. Using exceptions to evoke application-level actions is justified, because it is only possible at that level to decide if a particular temporal exception corresponds to a fault. This approach avoids false positive failure diagnosis as would result from low-level monitoring. In general, an exception handler either recovers from a violation and still provide usable data, or it propagates the violation to the next segment. The propagation of all unrecoverable violations is essential, as it allows us to use the (m, k)-constraint from B_{e2e}^c also for the individual segment deadlines d^{s_i}. Without propagation, the deadline misses would add up, thus leading to undetected violation of the event chain's (m, k)-constraint. In order to enable a recovery, violations must be detected before d^{s_i} and the latency of the exception handler itself must be bounded. The latter is achieved by executing the exception handling on the highest scheduling priorities. We therefore split d^{s_i} into a monitored deadline $d_{\text{mon}}^{s_i}$ and the maximum latency of the exception handling d_{ex}: $d^{s_i} = d_{\text{mon}}^{s_i} + d_{\text{ex}}$.

Without monitoring, the n-th latency $l_n^{s_i}$ of a segment s_i is the difference between its corresponding end and start event timestamps: $l_n^{s_i} = t_{\text{e,n}}^{s_i} - t_{\text{st,n}}^{s_i}$. An end event originates from a regular publication or receive event. With monitoring in place, we define the segment latency as the time between a segment's start event and either the corresponding end event or the end of the temporal exception, whichever occurs first. Monitoring therefore limits this segment latency to d^{s_i} in case the end event does not occur within $d_{\text{mon}}^{s_i}$. Note, that we do not abort the execution of any services within a segment, i.e. despite a temporal exceptions a segment still continues processing and producing its end event. Based on the timestamp of the start event of a segment s_i, the monitor initializes a timer to trigger an exception after expiration of $d_{\text{mon}}^{s_i}$. If the segment terminates in time, an end event $e_e^{s_i}$ is created, which causes the monitor to stop the timer as well as further error handling actions. If the segment does not terminate in time, a timeout is raised which triggers the temporal exception. For local segments, which end with a publication event, the exception is propagated by omitting the publish action that coincides with the next and late end event. As our monitoring mechanism for remote segments is based on the assumption of a lossy transmission channel, it will detect the missing publication event after a certain timeout (cf. Section IV-B). For remote segments, which end with a receive event, the exception is propagated by sending an error propagation event instead of a start event to the monitor of the subsequent local segment.

We visualized the principle of the temporal exceptions and propagation mechanisms along a chain execution based on our use case segmentation in Fig. 3. Here, the first segment, i.e. the remote segment representing the front lidar data, finishes within its budget $d_{mon}^{s_0}$. Next, the local segment s_1 exceeds its deadline $d_{mon}^{s_1}$, however, the error handling decided to recover as the rear lidar is not too critical and the front lidar data is still present. It therefore sends the current point cloud containing only the information from the front lidar. As the following remote segment failed to finish in time too, another exception is raised after $d_{mon}^{s_2}$. In this case, recovery is not possible and the error is explicitly propagated to s_3, which directly goes into error handling in order to react fast to potential safety-critical situations.

Algorithm 1 Remote segment exception handling.

```
1:  procedure HANDLE_REMOTE_EXCEPTION(m)
2:      data = user_exception(m)
3:      if data then                          ▷ recovery case
4:          issue_receive(data)
5:          return true
6:      else                                  ▷ propagation case
7:          propagate_exception()
8:          return false
9:      end if
10: end procedure
```

Algorithm 2 local segment exception handling.

```
1:  procedure HANDLE_LOCAL_EXCEPTION(m)
2:      data = user_exception(m)
3:      if data then                          ▷ recovery case
4:          publish(data)
5:          return true
6:      else                                  ▷ propagation case
7:          return false
8:      end if
9:  end procedure
```

Algorithm 1 and Algorithm 2 show the pseudocode of the exception handling for remote and local segments respectively.

https://doi.org/10.24355/dbbs.084-202101271521-0

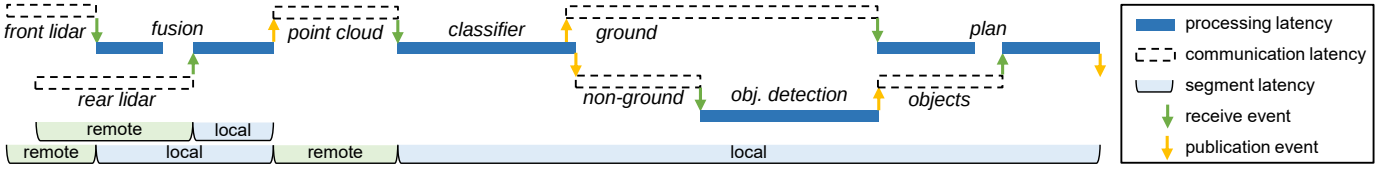

Fig. 2: Sequence of communication events and resulting latencies in our use case.

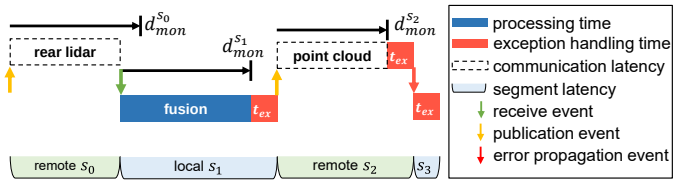

Fig. 3: Example of a chain execution in an error case.

Both procedures are called in case a temporal exception was detected. Their argument m provides the current number of misses within the last k executions. Furthermore, their return value indicates whether it could recover from the current exception (true) or whether it still counts as a deadline miss (false). For both segment types, the application-specific exception handler is called (Line 2 and Line 2). If the handler was able to perform a recovery for a remote segment, it passes the recovered data to the subsequent local segment (Line 4). Otherwise, it propagates the exception by issuing an error propagation event (Line 7). For local segments, the recovery is performed by publishing the recovered data as a regular middleware message (Line 4). The propagation, on the other hand, does not require further actions as the subsequent remote segment will detect a lost message after a certain timeout. Note, that we assume that the monitor logic takes care of discarding end events and their corresponding publication or receive actions if an exception occurred before. Further details are provided in Section IV.

C. Determining segment deadlines

In order to deploy the monitoring concept, we first have to find reasonable segment deadlines. We already pointed out, that formal analyses are not very practical on high-performance architectures, thus we will follow a measurement-based approach. First, we record one or multiple traces (without monitoring) to measure segment latencies $l_n^{s_i} \in L^{s_i}$ according to the aforementioned event definition. Next, we have to add the worst-case response times (WCRTs) of the monitoring exceptions[1] to each of the traced values. This results in the so-called *extended latency* $l_n'^{s_i} = l_n^{s_i} + d_{\text{ex}} \in L'^{s_i} \forall n$ and is a candidate for the deadline d^{s_i}. The set of all extended latencies is then called the extended trace L'^{s_i}.

When determining d^{s_i} values for segments along a chain, we want to fulfill Eq. (1). Yet, if we only took the maximum extended latencies as lower bounds for d^{s_i} we would often not be able to satisfy the end-to-end constraint. Our goal is thus to find the minimum d^{s_i} that can still satisfy the (m,k)-constraint, thereby accepting deadline misses as part of the regular operation. Note, that a violation of the n-th chain execution is defined as an unrecoverable (i.e. propagated) deadline miss of any of its corresponding n-th segment activations within $L_n'^{s_i}, s_i \in S_c$.

We can formulate the following constraint satisfaction problem for determining the minimum possible d^{s_i} while respecting whether deadline misses will be propagated or recovered.

$$\text{find} \quad d^{s_i} \in \mathbb{N} \qquad \forall s_i \in S_c \qquad (2)$$

$$\text{subject to} \quad B_{e2e}^c \geq \sum_{s_i \in S_c} d^{s_i} \qquad (3)$$

$$B_{seg}^c \geq d^{s_i} \qquad (4)$$

$$m \geq \max_n M_i(n) \qquad \forall s_i \in S_c \qquad (5)$$

$$\text{with} \quad m_i(n) = |\{j | n \leq j \leq n+k \wedge l_j'^{s_i} > d^{s_i}\}| \qquad (6)$$

$$M_i(n) = m_i(n) + \sum_{l=0}^{n-1} p_l \cdot m_l(n) \qquad (7)$$

First of all, Eq. (3) represents the event chains' latency budget constraint whereas Eq. (4) represents its throughput requirement. Eq. (5) ensures that at any position n in the trace, there are not more than m misses for every segment including unrecoverable misses of preceding segments. Here, Eq. (6) calculates the number of segment deadline misses that occur between the n-th and $n+k$-th activation of the segment. Eq. (7) takes propagation of deadline misses from preceding segments into account where p_l is the propagation factor of the segment s_l that can assume 0 (in case of perfect recovery) or 1 (in case of propagation). For $p_l = 0$, the constraint satisfaction problem is split into several single-variable problems for each segment. For $p_l = 1$, we refer to heuristic methods or integer linear programming (ILP), which is, due to limited space, out of the scope of this paper. An event chain is then called scheduleable if a solution to the constraint satisfaction problem exists.

IV. Monitoring Implementation

In this section, we present the details of how to implement the monitoring mechanisms for local and remote segments. We implemented the monitoring mechanisims in the libraries of ROS2 Eloquent.

A. Monitoring of local segments

For the latency monitoring of local segments, we leverage the fact that the start event and the end event of a segment occur on the same processing resource. These events correspond to communication events such as the sending of a message by a publisher or the reception of a message

[1] As the exception handling may be safety-critical, a conservative WCRT estimate should be acquired with analytical methods.

https://doi.org/10.24355/dbbs.084-202101271521-0

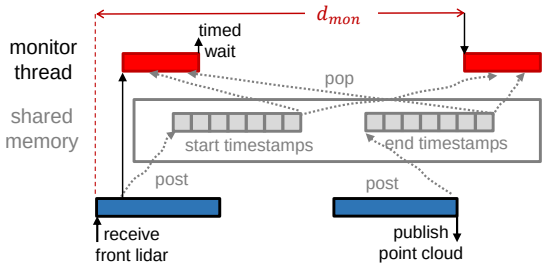

Fig. 4: Local segment monitoring on ECU 1 (cf. Fig. 1).

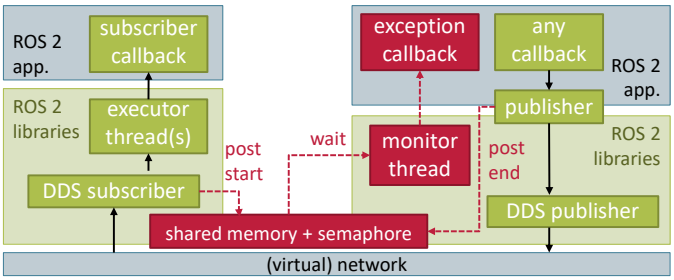

Fig. 5: ROS2 implementation of local segment monitoring (added parts in red).

by a subscriber. In ROS2, the middleware is provided as a library that is dynamically linked into every process, i.e. there is no dedicated middleware process which could observe the start events and end events. Instead, the timestamp of every start event must be made available to the process in which the end event will occur. For this purpose, additional inter-process communication is required that should be faster than the communication primitives provided by the middleware. We therefore leverage shared memory and inter-process semaphores. Figure 4 illustrates our approach on the example of the segment that starts with the reception of the front lidar message and ends with the publication of the combined point cloud. In ROS2, the application logic is implemented as callbacks that are activated by middleware messages or timers and that are dispatched by a single-threaded executor. Our implementation adds a high-priority monitor thread to the process in which the end event occurs. This monitor thread initialises the shared memory sections and semaphores so that they can be used by other processes. For every local segment, there is a distinct shared memory section whereas each monitor thread has only a single semaphore. A monitor thread waits on its semaphore and is resumed either when the semaphore is raised or after a programmed timeout (using `sem_timedwait()` on Linux). Every shared memory holds two separate wait-free ring buffers, one for the start events and one for the end events of the corresponding segment. During initialisation of a ROS node, it connects to the shared memory sections of the segments in which the node is involved. For every start event of the node, it will also connect to the semaphore of the corresponding monitor thread.

Whenever a start event occurs, the modified DDS subscriber code posts the current timestamp into the corresponding ring buffer in shared memory and activates the monitor thread via its semaphore. The monitor thread is immediately resumed as it takes precedence over the executor thread due to its high scheduling priority. When resumed, the monitor thread pops the timestamps from the ring buffers and adds a timeout to an internal timeout queue for every start event according to the defined latency budget d_{mon}. For every end event, the corresponding timeout will be removed from the timeout queue. If a timeout has occurred and the corresponding end event is not in the end-event ring buffer, the monitor thread triggers the application's exception handler.

After an exception has been handled, the next publication event will be skipped. To achieve this, the monitor thread increments a shared counter that will be evaluated by the publisher. In case the registered exception handler acts as a recovery handler, it has already performed the publication. Otherwise, it ensures that the violation is propagated to the next (remote) segment. Whenever an end event occurs, the publisher code posts the current timestamp to the corresponding ring buffer. To save an unnecessary context switch, the monitor thread is not notified since the processing of end events is not time critical.

Figure 5 depicts the necessary modifications of the ROS2 framework by showing the added parts (exception callback, monitor thread and shared memory) as red boxes.

B. *Monitoring of remote segments*

In contrast to the monitoring of local segments, the start and end events of remote segments occur on different processing resources. Thus, this limitation has to be addressed by a special monitoring concept, overcoming lack of information, while still providing reliable in-time deadline violation detection. In the following, we will first introduce our general concept on how to realize decentralized monitoring for a communication between ECUs, before going into details of our specific remote monitoring implementation in ROS2/DDS.

1) Theoretical concept for monitoring distributed communication evens: As stated above, distributed latency monitoring always lacks the ability of making information available to all components at the same time. In addition, the monitor should be placed at the receiver, as latency violations impact the functionalities subscribed to the monitored data. As a basic assumption, we can state that we can not guarantee the exchange of timestamps between the processing resources via Ethernet with a deterministic timing, so that the exact start event timestamp is not available for timer configuration of the monitor at the receiver.

Yet, we can instantiate monitoring on the receiver side without the need of any additional communication. In the following, we discuss the applicability of two approaches in the context of our safety-critical real-time requirements. We start with the *inter-arrival* monitoring as a basic concept in DDS followed by a more sophisticated synchronization-based approach presuming time-synchronization of ECUs.

https://doi.org/10.24355/dbbs.084-202101271521-0

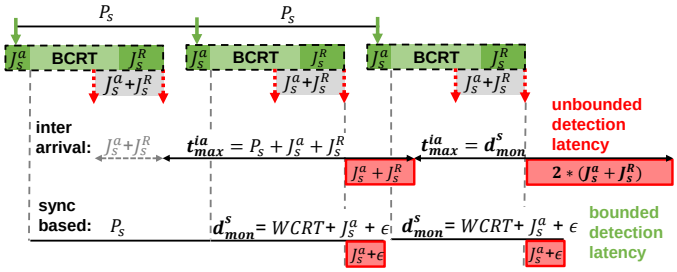

Fig. 6: Inter-arrival monitoring is prone to false positives.

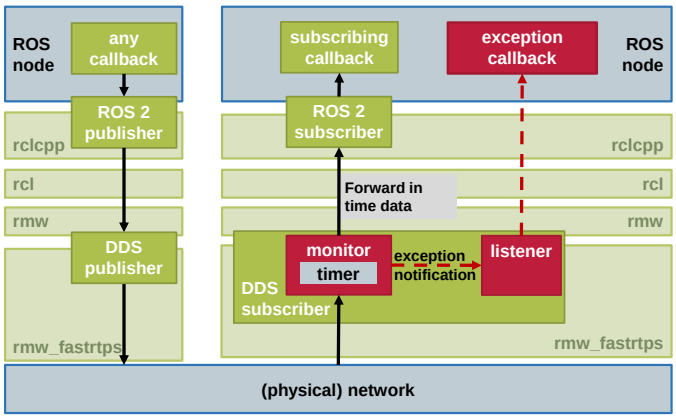

Fig. 7: Remote monitoring integration into the ROS2/DDS middleware.

Inter-arrival monitoring: This common monitoring approach, as used in DDS consists in observing the arrival times on the receiver side. The implications are illustrated in Fig. 6, which shows the periodic activations (green arrows) of a remote segment and the earliest/latest occurrences of its end events (red arrows). As event chains are activated periodically the start events of a remote segment occur with the same period i.e. P_s and are subject to arrival jitter J_s^a. On account of varying execution times, the end events are further affected by the amount of the network response time jitter J_s^R, exceeding the best-case response time (BCRT). In order to apply inter-arrival monitoring, a timer is programmed after each end event (received message) with $t_{\max}^{ia}$, which represents the maximum time between two consecutive end events. The main limitation of this approach is that it does not detect consecutive deadline misses as the timer is only programmed on arrival of a message and thus, without interpretation of any currently available timestamp. Thus, in case of (m, k)-constraints, it is only suitable for $m = 0$. Another drawback with this approach is that consecutive late arrivals can sum up for $t_{\max}^{ia} \geq P_s$ and without the ability of detection. As a result, there is no way to safely detect any concrete d_{mon}^s value modeled by $t_{\max}^{ia}$, without the occurrence of false positives. Such an approach is more suitable for liveliness rather than real-time concerns.

Synchronization-based monitoring: A more sophisticated approach for periodic communication can benefit from time synchronization of control units, which is provided via Precision Time Protocol (PTP) [19] in modern cars. With time synchronization, we can interpret the start event timestamp $t_{\text{st,n}}^s$ that is transmitted together with the data in DDS. The timer for the reception of the $n + i$-th end event can then be programmed with $t_{st,n}^{s_i} = t_{\text{st,n-1}}^s + (i + 1)P_s + d_{\text{mon}}^s$ with $d_{\text{mon}}^s = \text{BCRT} + J_s^R + J_s^a + \epsilon$. Note, that for different (m, k)-constraints, lower $d_{\text{mon}}^{s^r}$ values are possible depending on the number of accepted deadline violation of the event chain, i.e. m, as well as on the possibility to recover appropriately. The principle of the synchronization-based approach is also shown in Fig. 6. In contrast to inter-arrival monitoring, the pessimism is bounded to the arrival jitter J_s^a and the synchronization error ϵ. $J_s^a + \epsilon$ can be determined from the measured trace, based on the received measured start timestamps. Note, that deviations from the measured J_s^a and ϵ do not lead to undetected latency violations: Imagine an activation occurs too late, this would result in a closer deadline for the corresponding transmission, as we programmed the timer based on the previous start event. On the other hand, an unexpected early activation would result in the case, that the transmission could exceed the segment's budget without detection. However, this can only be the case, if the previous segments along the chain finished earlier and thereby left slack to be used by the remote segment. Note, that the monitor works on a high level and is even transparent to retransmissions of (partially) lost data e.g. over DDS, which gives flexibility to lower levels.

2) Synchronization-based monitoring integration into DDS and ROS2: The ROS2 middleware can be set up based on different DDS implementations (different vendors), however, our implementation focuses on the default, which is *eProsima* DDS. As motivated above, the synchronization-based monitor is implemented at the receiver side, in order to achieve short exception handling latencies related to the subscribing application. Moreover, an implementation should cover as much as possible of the communication latency within the communication infrastructure depicted in Fig. 7, to achieve seamless segmentation of the event chain. Thus, it is appropriate in a DDS/ROS2 setup to integrate the monitor directly in the interface to the application (represented by the ROS middleware), which is the DDS subscriber. Higher layers would lead to unnecessary complexity regarding the monitoring integration, as the publishing timestamp is natively passed up to the DDS Subscriber and receive events are directly activating subscription callbacks (i.e. applications). Moreover, the timestamp can directly be used to reconfigure the timer for the next deadline $t_{st,n}^{s_i}$ and exception callback events can be triggered in parallel to receive events. We instantiate one monitor for each subscriber listening to one communication. The concrete monitoring parameters can then be configured via the *eProsima* default XML-configuration. Note, that for multiple communication partners on the same topic, multiple monitors have to be instantiated, and differentiated based on delivered DDS topic keys.

https://doi.org/10.24355/dbbs.084-202101271521-0

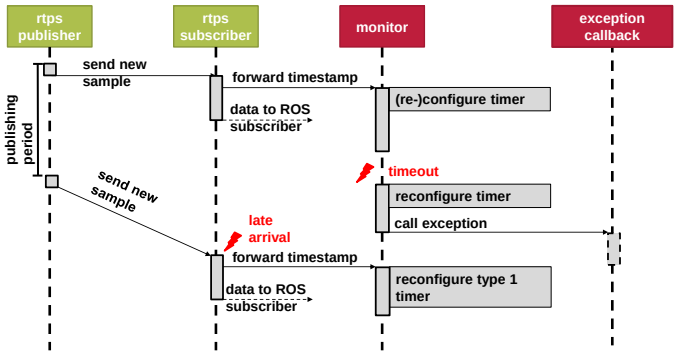

Fig. 8: Behaviour of remote monitoring

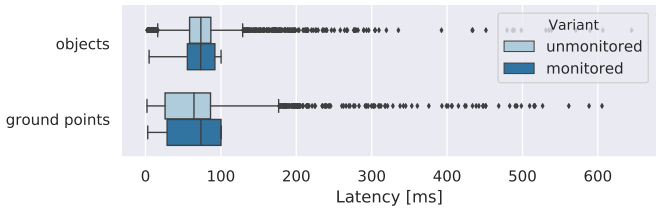

Fig. 9: Segment latencies on ECU2 with and without monitoring.

3) Synchronizatin-based monitoring flow: Fig. 8 illustrates our monitoring implementation in form of a sequence diagram. The monitoring starts with the DDS publisher sending out a new sample, including the send timestamp. After its reception at the subscriber, the monitor initializes the deadline timer based on the synchronization-based approach. Thereafter, it passes the data towards the ROS2 subscriber as part of the regular operation. If the next sample arrives in time, the monitor simply reconfigures the deadline expiration timer based on the currently received timestamp. In case of a deadline violation, an exception is signalled by the subscriber listener to the ROS2 middleware, which causes the execution of the attached callback. Besides this error handling mechanisms, also the next monitoring deadline is configured. In this case we simply add the configured publication rate of the event chain to the last set deadline and restart the timer based on this value. Messages that arrive too late, i.e. after the corresponding exception, will be discarded in order to skip the corresponding receive event. This allows the exception handler to either recover from the violation (and issue the receive event itself) or implicitly propagate the violation to the next (local) segment. Otherwise we would violate the constant rate assumption needed for event chain composability as well as for reliable (m, k)-constraint checks.

V. Evaluation

In order to underline our proposed monitoring concept, we evaluate the involved overheads and the latency for detection of and reaction to temporal exceptions. To achieve a bounded exception latency t_{ex}, that we assumed in Section III, we particularly have a look at the latency for the exception detection and the entry to the application-specific handler. This section is split up into the evaluation of local and remote monitoring respectively.

A. Local monitoring

We evaluated our proof-of-concept (unoptimized) implementation of local segment monitoring on the aforementioned Autoware.Auto use case from Fig. 1. In particular, we used the pcap-data of a front lidar provided by Autoware.Auto and instrumented and traced the software system on ECU2 using LTTng [4]. We executed the software system on a Linux kernel version 5.6.4 with PREEMPT_RT enabled (i.e. full preemptivity). Everything was run on a Core i5-3210M Quad-Core CPU. We assigned distinct real-time priorities to every ROS process in descending order, i.e. the plan process had the lowest priority. The monitor thread was assigned the highest scheduling priority whereas the ksoftirq threads, which handle the interrupts from the network controller, where executing on a priority just below the monitor thread. For representing performance and power optimizations, we allowed thread migration between cores and frequency scaling.

Note, as a compatible trajectory planning service was not available within Autoware.Auto, we replaced it with the visualization service *rviz2* as suggested by the project. This has the implication that we can only monitor the local segments up to reception of each topic within rviz2 since the latter does not publish any topic. We therefore end up with two local segments on ECU2: both starting from the reception of the point cloud by the classifier service and ending at the reception of either the *objects* topic or the *ground points* topic (cf. Fig. 2).

Figure 9 shows a Tukey boxplot for the measured latencies of the two segments on ECU2 with and without monitoring. The plot summarises approx. 4700 data points for each segment. Without monitoring, we get segment latencies up to $600\,ms$. For the monitoring, we used a segment deadline of $100\,ms$ and captured the time up until either the topic was received or the corresponding temporal exception was raised and handled. In consequence, we can guarantee a reaction within $100\,ms$ after the segment's start event. To get a more detailed view on how exactly these $100\,ms$ are enforced, we plotted only the cases in which a temporal exception occurred. The resulting boxplot is depicted in Fig. 10 and bases on 934 data points for the objects segments and 1699 data points for the ground-points segment. What we read from this figures is that the detection and triggering of temporal exceptions can take up to a few hundred microseconds in the worst case. In comparison to the segment latency, we believe this is an acceptable reaction time. Note, that the both segments are monitored by the same monitor thread that processes the buffers in a fixed order so that the ground points segment is delayed by the processing of the objects segment.

In addition to the segment latencies, we also measured the overheads that in consequence of the monitoring logic in the DDS subscriber and the publisher. Figure 11 depicts

https://doi.org/10.24355/dbbs.084-202101271521-0

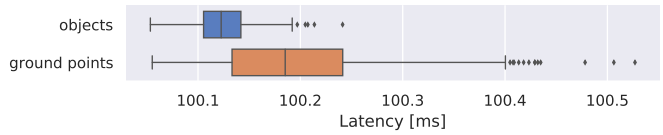

Fig. 10: Segment latencies on ECU2 for the temporal exception cases.

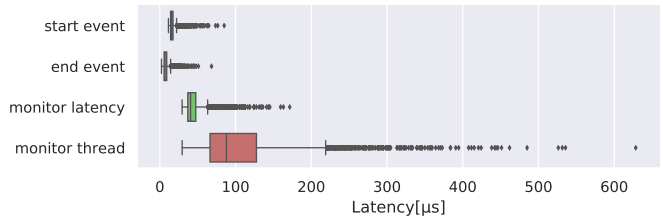

Fig. 11: Measured overheads for local segment monitoring.

the results as Tukey boxplots. The start-event and end-event overhead denote the time it took to post a start event to the ring buffer or to post an end-event into the ring buffer respectively. Both overheads are few 10-th of microseconds on average and below $100\,us$ in the worst case. The monitor latency is the time it took between posting a start event into the ring buffer until the same event was read and processed by the monitor thread. This latency is an indicator for the minimum segment latency budget since the monitor thread must be guaranteed to program the timer before the segment deadline. Last, we also measured the execution time of the monitor thread in total to get a notion of how much time it takes the process and evaluate the incoming start and end events. In conclusion, without any optimizations, we already achieved overheads for the local monitoring that are significantly smaller than the segment deadlines.

B. Remote monitoring

The evaluation of the remote monitoring approach has been implemented in parallel to existing ROS2 deadline and lifespan QoS monitoring mechanisms. Thus, we used the timer connected with an timeout routine, both within the context of the eProsima fastRTPS DDS middleware. The timeout routine would then forward the exception to the high-priority monitor thread, e.g. by using shared memory and a semaphore as we used for local monitoring. As we already know the latency for entering the high-priority monitor after posting an event, which is below $200\,us$, we evaluated the latency to enter the timeout routing after the middleware deadline timer expires. In detail, it is the difference between the entry of the timeout routine and the actual configured monitoring deadline. Figure 12 shows this implication as a Tukey boxplot with 472 points of data: The latencies vary between short $100\,us$ and long outliers up to nearly $2\,ms$. This shows that the latency is much higher compared to the local monitoring reaction time. The problem arises from the fact that we do not run the middleware thread at the highest priority. This would not be practical anyway,

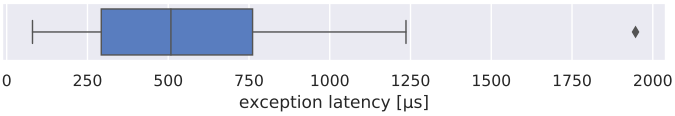

Fig. 12: Exception entry of the remote monitoring within the DDS context.

as the entire network load would interfere with all regular services. As we ran the evaluation with low CPU load, it can be expected that more load will further worsen the results.

We can conclude, that monitoring entirely within the middleware is not sufficient for achieving short and bounded reaction times. Therefore, we propose to forward the timer programming and timeout handling to the high-priority monitor thread instead of programming the timer within the DDS subscriber. As we could use a similar scheme as used for the local monitoring, we can expect similar latencies.

VI. Conlusion

This paper presented an online monitoring concept for safety-critical event chains for middleware-centric architectures with end-to-end weakly-hard real-time constraints. As end-to-end latencies cannot be sufficiently monitored using existing methods, we proposed a new approach that consists of two monitoring mechanisms that complement one another. One mechanism aims at monitoring of latencies between processes on the same processor and leverages shared memory for exchanging timestamps with little overhead. The other mechanisms reuses existing QoS mechanisms in the middleware and allows a supervision of network latencies without requiring additional network communication. Both mechanisms are combined into an end-to-end monitoring scheme by splitting an event chain into local and remote segments that are individually monitored by the aforementioned mechanisms so that a temporal exception is raised to the application in case the segment deadline was exceeded. For this purpose, we argued how the segment deadlines can be determined from execution traces such that weakly-hard (m, k)-constraints can be met and a certain time budget will be reserved for exception handling. In our proof-of-concept implementation in ROS2, we could show that the monitoring logic can be implemented in a way that does not leave any unmonitored gaps between segments. Our evaluation on an Autoware.Auto use case further demonstrated that overheads are comparatively low and well acceptable. Moreover, the latency for exception handling can be bounded if timeouts are programmed and received by a high-priority monitor thread.

References

[1] "AUTOSAR Specification of Communication Management for Adaptive Platform R19-11."
[2] "Autoware.Auto," https://www.autoware.auto/, accessed: 2020-05-28.
[3] Z. A. H. Hammadeh, S. Quinton, R. Henia, L. Rioux, and R. Ernst, "Bounding Deadline Misses in Weakly-Hard Real-Time Systems with Task Dependencies," in *Design Automation and Test in Europe*, Lausanne, Switzerland, Mar. 2017.

https://doi.org/10.24355/dbbs.084-202101271521-0

[4] M. Desnoyers and M. Dagenais, "Lttng: Tracing across execution layers, from the hypervisor to user-space," in *Linux symposium*, vol. 101, 2008.
[5] M. Maggio, J. Lelli, and E. Bini, "rt-muse: measuring real-time characteristics of execution platforms," *Real-Time Systems*, vol. 53, no. 6, pp. 857–885, Nov. 2017.
[6] M. Garca-Gordillo, J. J. Valls, and S. Sez, "Heterogeneous runtime monitoring for real-time systems with art2kitekt," in *International Conf. on Emerging Technologies and Factory Automation (ETFA)*, 2019.
[7] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Euromicro Conference on Real-Time Systems*, 1999.
[8] F. Jahanian, R. Rajkumar, and S. C. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Systems*, vol. 7, no. 3, pp. 247–273, 1994.
[9] D. Bertrand, S. Faucou, and Y. Trinquet, "An analysis of the autosar os timing protection mechanism," in *IEEE Conference on Emerging Technologies Factory Automation (ETFA)*, 2009.
[10] C. Ficek, N. Feiertag, and K. Richter, "Applying the autosar timing protection to build safe and efficient iso 26262 mixed-criticality systems," 2012.
[11] M. Neukirchner, T. Michaels, P. Axer, S. Quinton, and R. Ernst, "Monitoring arbitrary activation patterns in real-time systems," in *Proc. of IEEE Real-Time Systems Symposium (RTSS)*, Dec. 2012.
[12] M. Neukirchner, P. Axer, T. Michaels, and R. Ernst, "Monitoring of workload arrival functions for mixed-criticality systems," in *Proc. of Real-Time Systems Symposium (RTSS)*, Dec. 2013.
[13] S. Kannan, M. Kim, I. Lee, O. Sokolsky, and M. Viswanathan, "A retrospective look at the monitoring and checking (mac) framework," in *International Conference on Runtime Verification*. Springer, 2019.
[14] L. Pike, N. Wegmann, S. Niller, and A. Goodloe, "Copilot: monitoring embedded systems," *Innovations in Systems and Software Engineering*, vol. 9, no. 4, pp. 235–255, 2013.
[15] S. Tobuschat, A. Kostrzewa, F. K. Bapp, and C. Dropmann, "Online monitoring for safety-critical multicore systems," *it - Information Technology*, 2017.
[16] "AUTOSAR Specification of Platform Health Management for Adaptive Platform R19-11."
[17] J. Peeck, J. Schlatow, and R. Ernst, "Online latency monitoring of time-sensitive event chains in safety-critical applications," in *Design Automation and Test in Europe Conference Exhibition (DATE)*, Feb. 2021.
[18] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.
[19] "IEEE 1588-2008 - Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems," IEEE, Standard, Jul. 2008.

https://doi.org/10.24355/dbbs.084-202101271521-0