Preprints of the 21st IFAC World Congress (Virtual)
Berlin, Germany, July 12-17, 2020

Agile Latency Estimation for a Real-time Service-oriented Software Architecture

Alexandru Kampmann * **Armin Mokhtarian** * **Jan Rogalski** *
Stefan Kowalewski * **Bassam Alrifaee** *

* *RWTH Aachen University - Chair for Embedded Software, Ahornstrasse 55, 52074 Aachen, Germany (e-mail: {kampmann, mokhtarian, rogalski, kowalewski, alrifaee}@embedded.rwth-aachen.de)*

Abstract: This paper presents our testbed and software pipeline for automatic latency estimation for a service-oriented software architecture (SOA). This type of architecture consists of modular services that are dynamically combined at runtime to form a functioning system. As different service combinations become possible at runtime, agile approaches for testing the resulting systems become necessary. Besides other factors, latencies are of particular interest for the implementation of control systems. Our agile approach automatically generates dummy services, including interfaces and tasks for internal processing, based on a service description in a human-readable format. Services are then automatically distributed to the computers of our testbed, which are connected through Ethernet. We empirically obtain latency estimates for processing and communication steps for a given composition of services. In this paper we describe our data format, abstractions about internal run-time behavior of services and the code generation pipeline. The evaluation presents latency estimates that we are able to obtain through our testbed that resembles the sense-plan-act paradigm.

Keywords: Networked systems, Systems with time-delays, Supervision and testing, Distributed control and estimation, Complex systems, Decentralized control, Diagnosis

1. INTRODUCTION

The implementation of complex control systems, such as highly automated vehicles, is based on new technologies, paradigms and architectures. This becomes evident in the automotive domain, where existing communication, compute and software architectures are starting to reach their limits, as described by Broy et al. (2007a) and Farcas et al. (2010). Prevalent communication architectures built on CAN or LIN are not suitable for transporting high bandwidth camera and LIDAR data. The compute platforms, mostly based on microcontrollers, are not suitable for running automation algorithms with high computation demand. Prevalent software architectures are not updatable and do not provide the flexibility required to keep up with the short development- and technology life cycles of aforementioned technologies. As a result, Ethernet, Linux-based Electronic Control Units (ECUs) and Service-oriented Architectures (SOA) become mean for implementation of such systems, e.g. through the introduction of the upcoming AUTOSAR Adaptive Platform. As described by Kugele et al. (2017), SOA architectures are characterized by modular services that are integrated at runtime to achieve a desired functional behavior. In contrast to today's function-oriented architectures, services do have hard-coded assumptions about services they interact with. Instead, they are dynamically assembled at runtime based on the quality of service they provide. This opens the possibility to update the software architecture and to dynamically reconfigure the system for different modes of operation.

Although these trends open up new possibilities, the introduced flexibility also poses a challenge for testing and validation of the resulting architectures. Up to now, systems are developed following the V-model and the architecture is fixed at some point during the engineering process. In many cases, the obtained validation result remains valid as the system is not expected to undergo changes in the future. In SOA approaches, a potentially large set of system variants become possible at runtime and have to be tested and validated. Dynamically adapting the software architecture at runtime impacts many different factors, ranging from memory consumption, computation and communication resources. Latencies, which occur due to internal service processing and communication overhead, also directly impact the performance of control systems. For example, improper consideration of latencies in control schemes can lead to instabilities. Estimating latencies in the system not only helps to improve control quality, but also to achieve a deterministic system behavior, which is especially important in safety-critical applications. In concepts such as Logical Execution Time (LET), latency estimates are systematically considered in order to achieve a predictable and deterministic system behavior (see Farcas et al. (2005)).

This paper presents our framework for agile latency estimation for Ethernet-based SOA. Based on XML descrip-

⋆ 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).

tions of the computation nodes and the services running on them, we automatically generate and execute tests that empirically estimate latencies. With the help of our automated testing pipeline, we can assess latencies for different system instances in an agile manner. Beside communication latencies, we also measure timing of our SOA implementation for assisting the future development process. Our concept is implemented in a testbed that is comprised of multiple computers that can be automatically setup to execute specified test cases.

The remainder of this paper is structured as follows. Section 2 covers existing work on this topic. We provide background to the concept for service-orientation that this work is based upon in Section 3. Our concept for agile latency estimation is presented in Section 4. Results obtained through our work are presented in Section 5. Section 6 concludes the paper and hints at potential future extensions.

2. RELATED WORK

This paper is based on our previous work on automotive SOA as introduced by Kampmann et al. (2019a,b). Although our previous work is focused on the automotive domain, the concepts are also applicable for the implementation of control systems in other domains. SOAs, in particular in the automotive domain, have previously been studied by Kugele et al. (2017), Broy et al. (2007b) and Bocchi et al. (2008), among others. A positive case-study on the implementation of an automotive SOA based on the Data Distribution Service (DDS) has been conducted by Kugele et al. (2018). There is a rich body of work on latency estimation and testbeds for various applications, see Saad et al. (2009) for a vehicle swarm testbed, or Stubbs et al. (2006) for a hovercraft testbed. To the best of our knowledge, no previous work has presented a latency estimation framework for control systems that are implemented using a SOA.

3. BACKGROUND

This paper is based on our previous work, which introduces a concept for SOA (see Kampmann et al. (2019a)). We will now briefly introduce the main building blocks of our architecture. At the core of our concept are modular services $S_i = (R_i, G_i) \in \mathbb{S}$ with typed interfaces, tasks for internal processing and a simple life cycle model. Interfaces required by S_i are requirements R_i, and interfaces provided by S_i to other services are guarantees G_i. Guarantees and requirements are mapped to a unique element in the set of interface types $\mathbb{T}$. Each interface $\tau_i = (I_i, D_i, Q_i, P_i) \in \mathbb{T}$ has a fixed identifier I_i and consists of data type definitions for Data D_i, Quality Q_i and Parameter P_i. Every requirement and guarantee is mapped to an element in the set of interface types by $\mathcal{T}_G$ with $\mathcal{T}_G : \{(i,j) \mid g_i^j \in G_i\} \rightarrow \mathbb{T}$ and $\mathcal{T}_R$ with $\mathcal{T}_R : \{(i,j) \mid r_i^j \in R_i\} \rightarrow \mathbb{T}$.

The connection of interfaces between different services is determined dynamically, and services are designed to make no hard-coded assumptions about the particular service they are connected with during runtime. This enables a run-time integrated, modular architecture. Hence, this requires a designated system component for instantiating the architecture at runtime. For this purpose, we have introduced the Orchestrator as the architecture controller. The Orchestrator $\mathcal{O} = (\mathcal{Q}, \mathcal{E}, \mathcal{A}, \mathcal{R})$ is a state transition system consisting of states $\mathcal{Q}$, a set of events $\mathcal{E}$, a set of actions $\mathcal{A}$ and transition relation $\mathcal{R} \subseteq \mathcal{Q} \times \mathcal{E} \times \mathcal{G} \times \mathcal{A} \times \mathcal{Q}$ with the set of predicate logic guard conditions $\mathcal{G}$. In state $q \in \mathcal{Q}$, the Orchestrator transitions to state q' and executes action a upon event $e \in \mathcal{E}$ if the guard condition g is true, i.e. $(q, e, a, g, q') \in \mathcal{R}$. The actions that the Orchestrator can invoke upon the architecture relevant for this work are the establishment of connections between interfaces and the control of the life cycle states of services.

We use a simple task model for computation within services. Tasks are methods that read an input interface, perform a particular computation, and write an output interface. Each requirement can be read by any task and each guarantee is written by exactly one task. A service may consist of more than one task. We distinguish between periodic and conditional tasks. Periodic tasks are regularly executed according to a user-specified frequency. Additionally, the user can specify the absolute starting point of the first execution of a periodic tasks, and all future task activations will be aligned to the induced time grid. This allows to control the relative phase-shift of multiple periodic tasks, e.g. in order to achieve alternating task activations. We synchronize time across ECUs using the Precision Time Protocol (PTP) introduced by Eidson and Lee (2002). Conditional tasks subscribe to one or more input interfaces and are triggered as long as there is data in any of the subscribed input interfaces.

Services follow a simple life cycle model that is controlled by the Orchestrator. They are either in steady states *stopped* and *started* or transient states *starting* and *stopping*. The tasks of a service are scheduled only in the *started* state.

Our service framework is implemented in C++ and is built upon the Data Distribution Service (DDS) (see Pardo-Castellote (2003)), which is used for data exchange between services and the Orchestrator. DDS is a middleware specification for decentralized, real-time communication in distributed systems and is standardized by the Object Management Group (OMG). It is based upon the User Datagram Protocol (UDP) and supports both publish-subscribe and request-reply communication patterns.

4. AGILE LATENCY ESTIMATION

This chapter presents our approach for latency estimation. We will first describe our approach on a conceptual level and then provide further implementation details.

Our approach for agile latency estimation is depicted in Fig. 1. First, a particular test-case consisting of services, connections between services and their placement on a particular ECU is defined using XML (1). Services are specified by their interfaces and tasks for internal processing. Based on the XML-based test specification, we automatically generate services (2, 3) that exchange dummy data (4) while capturing timestamps for latency estimation at various points in our architecture. Timing information is then automatically collected (5) and various latency measures are automatically computed (6). The flexibility

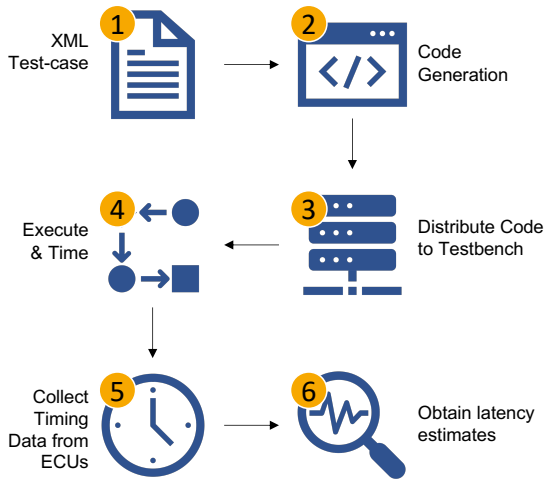

Fig. 1. Our workflow for agile latency estimation.

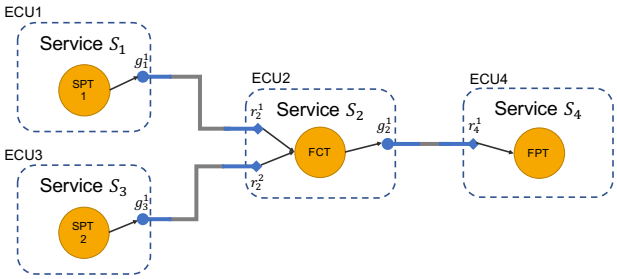

Fig. 2. A particular test case consisting of services S_1, S_2, S_3 and S_4 with interfaces and tasks for internal processing.

of our approach stems from the ability to automatically generate, deploy, execute and evaluate tests based on the aforementioned XML specification.

4.1 Obtained Latency Measures

Various parts that contribute to the overall end-to-end communication latencies are measured and attributed to internal mechanisms of our architecture. This not only allows to estimate overall latencies for a specific system, but also to pinpoint and optimize bottlenecks in our implementation. We will now further detail the approach.

A test definition consists of multiple elements. First, we have ECUs which run one or more services. A service consists of a name, a list of guarantees and requirements, and a list of tasks. Guarantees and requirements are typed interfaces, which also define the size of data being exchanged. As we do not conduct functional tests, we use a dummy interface type which consists of a byte-array with a specific size. Each interface is therefore defined by its name and data size. As explained in Section 3, services consist of two different types of tasks (conditional and periodic tasks) for carrying out computation within a task. These two basic tasks are defined by their activation pattern and not the actual computation carried out. In order to constraint the computation patterns within a task, we assume the following three types of internal task behaviors.

(1) **Source Periodic Task (SPT).** This kind of task is activated with a fixed frequency and writes data to an output interfaces in each activation. The number of packages is later denoted as the sample size, as it defines the number of packages which are sent through the network for testing. An SPT writes data to multiple guarantees but does not read data from requirements. As described in Section 3, an absolute starting time to which all activations will align can be defined for synchronization of multiple periodic tasks. This is especially important when multiple periodic tasks are involved in a computation chain, as offsets between their absolute starting times can influence latencies.

(2) **Forwarding Periodic Task (FPT).** These tasks are also activated periodically. During each activation, SPTs read data from the circular buffers of specific requirements, wait for a configurable amount of time to simulate computation, and forward the data through output interfaces. This type of task is defined by a list of requirements read, guarantees written, the execution frequency and a starting offset.

(3) **Forwarding Conditional Task (FCT).** This conditional task is executed every time all its requirements received data. During execution, for every requirement linked to the task, data is retrieved from the circular buffer and for every guarantee new packages are created and sent to other services. FCTs are similar to SPTs, but follow sporadic instead of periodic activation patterns.

Fig. 2 shows a setup consisting of multiple services, interfaces and tasks. Services S_1 and S_3 each have one SPT that writes to their respective guarantee interfaces. Service S_2 has two requirements r_2^1, r_2^2 and one guarantee g_2^1. The FCT in S_2 is configured to trigger as soon as data for requirements r_2^1 and r_2^2 is available. Upon execution, FCT reads from both requirements and writes g_2^1. Service S_4 has one FPT, which reads data from its only requirement r_4^1. In general, services may consist of multiple tasks, but we constrain our example to one task per service for the sake of simplicity.

Fig. 3 depicts the interaction between services and the latency measures that our framework automatically obtains from the setup above. SPT1, which is executed with 100 Hz, writes one data package to guarantee g_1^1 at the end of each execution. SPT2 is also executed with 100 Hz but its absolute starting point is shifted by 2 ms with respect to SPT1. At the end of each SPT2 execution the guarantee g_3^1 is written. FCT of S_2 is executed as soon as data from both requirements is available. Upon termination, FCT writes data to g_2^1. At the end of the computation chain, FPT in S_4 reads the data in r_4^1 upon activation. The resulting chain of interaction is highlighted in Fig. 3. Note that the particular setup is executed periodically, as hinted in gray.

We take timestamps at different events. First, timestamps are taken upon start of task execution. SPT1 is executed at t_0, SPT2 at t_1, FCT at t_2 and FPT at t_3. SPT1 task has the earliest absolute start reference among all tasks and therefore acts as absolute start reference $t_0 = 0$ s. The second latency measured is the transmission time of the packages exchanged between two services, in our example denoted as d_1, d_2 and d_3. This measure takes into account both the latency stemming from the Ethernet network, the operating system as well as our

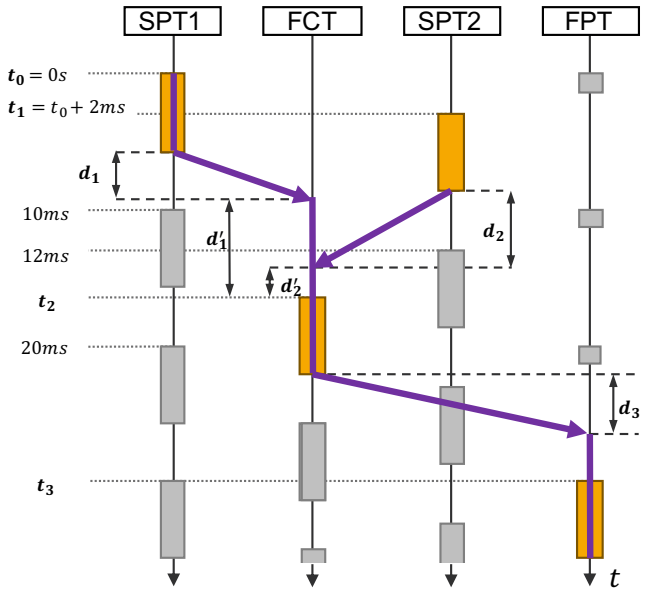

Fig. 3. Sequence diagram displaying the interaction between tasks for the example setup displayed in Fig 2. Measurement points for latency estimation are indicated in bold letters.

service layer. The third measure captures the time between the reception of a package and the actual activation of FCT1, denoted as d_1' and d_2'. A configurable amount of packages are sent through the network and timestamps are obtained for every run. Note that in our concept for service orientation, the interfaces are not matched automatically, but explicitly through the Orchestrator. Therefore, the test specification also needs to contain which interfaces are linked with one another.

4.2 Test description in XML

We will now provide further details on the use of XML files to specify tests. The test specification consists of multiple ECUs and services running on them, which in turn consist of interfaces and tasks. Listing 1 shows the XML definition of the topology in Fig. 2.

In line 1–3, the interface types used in the test are defined. In our example, we define an interface called *LP* which has a 200 bytes payload. The interface name is used for specifying the type of guarantee or requirement (cf. lines 7 and 19).

ECU_1, ECU_2 and ECU_3 are defined in line 5–50. An IP address per ECU is specified for code deployment purposes, although our SOA implementation does not require the user to specify IP addresses.

Service S_1 is placed on ECU_1 (cf. lines 6–14). This service has one guarantee corresponding to g_1^1 (called *gua_s1*) of *LP* type and consists of SPT1 (cf. lines 8–13). Frequency, sample size and the starting offset in microseconds are specified in line 10, 11 and 12. ECU_2 is introduced in line 16–28. Service S_2 has two requirement called *req_s21* and *req_s22*, which correspond to r_1^2 and r_2^2 (cf. lines 19–20) and one guarantee *gua_s2*, which corresponds to g_2^1. The services S_3 and S_4 are placed on ECU_3 (cf. lines 29–39) and ECU_4 (cf. lines 40–49), respectively.

```
1  <latpacks>
2  <latpack size="200">LP</latpack>
3  </latpacks>
4  <ecus>
5  <ecu name="ECU1" ip="137.226.8.72">
6    <service name="S1">
7      <gua name="gua_s1" datatype="LP" />
8      <spt name="SPT1">
9        <gua>gua_s1</gua>
10       <frequency>100</frequency>
11       <samplesize>1000</samplesize>
12       <startref>0</startref>
13     </spt>
14   </service>
15 </ecu>
16 <ecu name="ECU2" ip="137.226.8.78">
17   <username>root</username>
18   <service name="S2">
19     <req name="req_s21" datatype="LP" />
20     <req name="req_s22" datatype="LP" />
21     <gua name="gua_s2" datatype="LP" />
22     <fct name="FCT">
23       <req>req_s21</req>
24       <req>req_s22</req>
25       <gua>gua_s2</gua>
26     </fct>
27   </service>
28 </ecu>
29 <ecu name="ECU3" ip="137.226.8.74">
30   <service name="S3">
31     <gua name="gua_s3" datatype="LP" />
32     <spt name="SPT2">
33       <gua>gua_s3</gua>
34       <frequency>100</frequency>
35       <samplesize>1000</samplesize>
36       <startref>2000</startref>
37     </spt>
38   </service>
39 </ecu>
40 <ecu name="ECU4" ip="137.226.8.76">
41   <service name="S4">
42     <req name="req_s4" datatype="LP" />
43     <fpt name="FPT">
44       <req>req_s4</req>
45       <frequency>100</frequency>
46       <startref>0</startref>
47     </fpt>
48   </service>
49 </ecu>
50 </ecus>
51 <connections>
52   <connection>
53     <gua>gua_s1</gua>
54     <req>req_s21</req>
55   </connection>
56   <connection>
57     <gua>gua_s3</gua>
58     <req>req_s22</req>
59   </connection>
60   <connection>
61     <gua>gua_s2</gua>
62     <req>req_s4</req>
63   </connection>
64 </connections>
```

Listing 1. Specification of the setup depicted in Fig. 2.

The last part of the XML file describes how the interfaces are linked with one another and corresponds to the connection depicted in Fig. 2. Note that it is possible to place multiple services on the same ECU.

4.3 Code Generation and Test Execution

In this section we describe the procedure of how an XML test definition is used to execute the latency measurement

in our testbed. The automated pipeline consists of steps 2–6 as depicted in Fig. 1. The execution of these steps is performed by the designated supervisor host, that is placed in the same Ethernet network as the ECUs to be tested.

(1) **Code Generation.** The code generator produces two outputs. First, the source code for every ECU and second, an Orchestrator script which assembles the system according to the specification. The source code for an ECU consists of the dummy service and package type definitions (cf. lines 1–3 in Listing 1). The Orchestrator script specifies the connection of interfaces and controls the life cycle of each service (cf. lines 51–64 in Listing 1).
(2) **File Distribution and Compilation.** After code generation, the supervisor copies the generated files to every ECU. Services are compiled locally on each ECU. For this purpose, the IP address of each ECU is provided by the XML definition.
(3) **Test Preparation & Start.** After compilation, the service binaries are started by the supervisor. Note that the services at this point are still in stopped state, i. e. the tasks are not executed. Next, the supervisor launches the Orchestrator. The Orchestrator runs on the supervisor and executes the Orchestrator script generated in step 1, which connects the services and activates them. After the activation, the SPTs starts sending the packages, which initiates the test.
(4) **Test End.** Upon sending the amount of specified packages, SPTs indicate the end of a test through a specific message to the supervisor. In order to inform subsequent tasks, the SPTs send a special packet to indicate test finalization. When a FCT/FPT receives such a package, it forwards it and shuts down.
(5) **Data Collection.** The supervisor downloads all generated data from the ECUs. Every task saves a timestamp upon activation. For every package, the timestamp at reception and transmission is saved.
(6) **Evaluation.** For the evaluation of the collected data, various statistics are computed from the raw data obtained in the previous step. For this, our evaluation script establishes the connection between the XML specification and the measured data. Then, it computes various statistics of interest, such as minimum, maximum or mean values.

5. DISCUSSION & EVALUATION

In this section, we present a testbed that implements the approach described in Section 4. Fig. 4 shows the testbed on a conceptual level. We follow the sense-plan-act paradigm and incorporate compute platforms of different characteristics. Three Intel NUCs provide plenty compute power for sensing and trajectory planing services. Xilinx UltraScale+ SoCs and Infineon Aurix provide less computation power but are, due to their embedded nature, more appropriate as a low-level actuator interface. The Intel NUCs run off-the-shelf Ubuntu 18.04 and the Xilinx UltraScale+ runs PetaLinux. For this work, we did not incorporate the Infineon Aurix and plan to do so in future work. The clock on all compute nodes is synchronized using PTP, which is essential for the comparability of timestamps. The supervisor is also running on an Intel

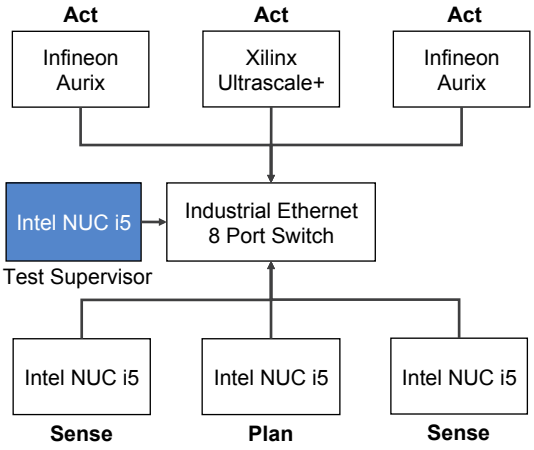

Fig. 4. The computers in our testbed that is used for latency estimation resemble the sense-plan-act paradigm.

Table 1. The starting times in µs of tasks relative to the activation of t_0.

	SPT1 (t_0)	FCT (t_2)	SPT2 (t_1)	FPT (t_3)
mean	0.0	2308.54	1854.84	9824.60
min	0.0	1532.55	1383.78	9410.85
25%	0.0	2126.87	1685.85	9641.40
50%	0.0	2205.25	1744.50	9716.98
75%	0.0	2519.66	2067.32	10028.24
max	0.0	3504.51	2286.91	10267.019

NUC, and all compute nodes are connected via an industrial Ethernet switch.

The purpose of this evaluation to demonstrate and discuss the results obtained from the specified test cases, and not to demonstrate the performance of our service framework. All tests were carried out without a real-time Linux patch, which leaves room for optimization.

For evaluation, we executed the test defined in the XML file displayed in Listing 1. SPT1, SPT2 and FPT run on three Intel NUCs and the FCT is executed on a Xilinx UltraScale+ board. Table 1 shows the results obtained for the start of task execution, which correspond to timestamps t_0, t_1 and t_2 in Fig. 3. Durations are expressed with respect to $t_0 = 0$.

SPT2 starts on average 1.8 ms later than SPT1. This corresponds to the desired activation offset in the provided example of 2 ms. We attribute the deviation to the lacking real-time capabilities of the operating system used. In other experiments running SPTs at 100 Hz, the starting point of two executions of the task deviates up to 1.7 ms. FCT1 is activated on average about 2.2 ms relative to the activation of SPT1. The processing of messages produced by SPT1 and SPT2 in FPT occurs 10 ms later. This corresponds to one activation period of FPT, which is the earliest possible processing time for the given configuration. Assuming that the described service setup implements a control system, the result can be interpreted as follows: a change in a sensor value will take on average 10 ms in order to have an effect on the actuation system.

Table 2 shows the latencies d_1, d_2 and d_3. We can see that the raw communication latencies between Intel NUCs

Table 2. Results of d_1, d_2 in µs.

	req_s21 (d_1)	req_s22 (d_2)	req_s4 (d_3)
mean	302.02	429.84	632.56
min	231.02	129.22	345.23
25%	282.28	409.60	616.72
50%	286.57	413.17	643.01
75%	324.24	451.32	664.94
max	781.05	1484.63	5686.28

Table 3. Results of d'_1, d'_2 regarding FCT in µs.

	$r_2^1 \rightarrow$ FCT (d'_1)	$r_2^2 \rightarrow$ FCT (d'_2)
mean	2004.29	20.15
min	1243.82	12.87
25%	1830.81	19.31
50%	1905.67	19.55
75%	2218.36	20.26
max	2807.14	55.55

are smaller than to the Xilinx UltraScale+, which can be attributed to the reduced computation power. Finally, Table 3 shows d'_1 and d'_2. It can be seen that the values of d'_1 are larger then the ones of d'_2. This is because SPT1 sends packages on average 1.8 ms earlier (according to Table 1). The time between the reception of the packages of SPT1 and the execution of FCT is longer than the time between the reception of packages of SPT2 and FCT's execution. This is expected based on the phase-shift between SPT1 and SPT2.

At the current state of our testbed, only Linux-based systems can be tested, as our code generation, deployment and compilation pipeline relies on Linux tools. Currently, no platforms with scarce resources such as microcontrollers are supported yet.

6. CONCLUSION AND FUTURE WORK

The presented work allows to obtain latency estimates in an agile approach for control systems that are implemented using a SOA approach. We have described the computation model for internal service processing and the data format used for test case definition. We have also provided latency estimates obtained on an exemplary testbed, that resembles the sense-plan-act paradigm. For future work, we plan to be able incorporate the low-level microcontrollers in our testbed in our automated pipeline. This will require adaptations to the code deployment stage, as these controllers need to be flashed using a special programmer. Furthermore, we plan to obtain fine-grained latency estimates for data serialization and other internal processes of our SOA implementation.

REFERENCES

Bocchi, L., Fiadeiro, J.L., and Lopes, A. (2008). Service-Oriented Modelling of Automotive Systems. In *2008 32nd Annual IEEE International Computer Software and Applications Conference*, 1059–1064. doi:10.1109/COMPSAC.2008.228.

Broy, M., Kruger, I.H., Pretschner, A., and Salzmann, C. (2007a). Engineering Automotive Software. *Proceedings of the IEEE*, 95(2), 356–373. doi:10.1109/JPROC.2006.888386.

Broy, M., Krüger, I.H., and Meisinger, M. (2007b). A Formal Model of Services. *ACM Trans. Softw. Eng. Methodol.*, 16(1). doi:10.1145/1189748.1189753. URL http://doi.acm.org/10.1145/1189748.1189753.

Eidson, J. and Lee, K. (2002). IEEE 1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. In *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, volume 10. IEEE.

Farcas, C., Farcas, E., Krueger, I.H., and Menarini, M. (2010). Addressing the Integration Challenge for Avionics and Automotive Systems—From Components to Rich Services. *Proceedings of the IEEE*, 98(4), 562–583. doi:10.1109/JPROC.2009.2039630.

Farcas, E., Farcas, C., Pree, W., and Templ, J. (2005). Transparent Distribution of Real-Time Components based on Logical Execution Time. In *ACM SIGPLAN Notices*, volume 40, 31–39. ACM.

Kampmann, A., Alrifaee, B., Kohout, M., Wüstenberg, A., Woopen, T., Nolte, M., Eckstein, L., and Kowalewski, S. (2019a). A Dynamic Service-Oriented Software Architecture for Highly Automated Vehicles. In *2019 22st International Conference on Intelligent Transportation Systems (ITSC)*, to appear. IEEE.

Kampmann, A., Wüstenberg, A., Alrifaee, B., and Kowalewski, S. (2019b). A Portable Implementation of the Real-Time Publish-Subscribe Protocol for Microcontrollers in Distributed Robotic Applications. In *2019 22st International Conference on Intelligent Transportation Systems (ITSC)*, to appear. IEEE.

Kugele, S., Hettler, D., and Peter, J. (2018). Data-Centric Communication and Containerization for Future Automotive Software Architectures. In *2018 IEEE International Conference on Software Architecture (ICSA)*, 65–6509. doi:10.1109/ICSA.2018.00016.

Kugele, S., Obergfell, P., Broy, M., Creighton, O., Traub, M., and Hopfensitz, W. (2017). On Service-Orientation for Automotive Software. In *2017 IEEE International Conference on Software Architecture (ICSA)*, 193–202. doi:10.1109/ICSA.2017.20.

Pardo-Castellote, G. (2003). OMG Data-Distribution Service: Architectural Overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, 200–206. IEEE.

Saad, E., Vian, J., Clark, G., and Bieniawski, S. (2009). Vehicle swarm rapid prototyping testbed. In *AIAA Infotech@ Aerospace Conference and AIAA Unmanned... Unlimited Conference*, 1824.

Stubbs, A., Vladimerou, V., Fulford, A.T., King, D., Strick, J., and Dullerud, G.E. (2006). Multivehicle Systems Control over Networks: A Hovercraft Testbed for Networked and Decentralized Control. *IEEE Control Systems Magazine*, 26(3), 56–69.