Measuring Latencies of IEEE 11073 Compliant Service-Oriented Medical Device Stacks

Martin Kasparick*, Benjamin Beichler*, Björn Konieczek*, Andreas Besting†, Michael Rethfeldt*
Frank Golatowski*, and Dirk Timmermann*
*Institute of Applied Microelectronics and Computer Engineering
University of Rostock, Germany
firstname.lastname@uni-rostock.de
†SurgiTAIX AG, Herzogenrath, Germany
lastname@surgitaix.de

***Abstract*—Vendor-independent interoperability is one of the key-enablers for medical devices in future operating rooms, intensive care units, and medical care in general. Using the paradigm of a Service-Oriented Architecture (SOA) is a promising approach realized by the new IEEE 11073 SDC family of standards. Standard compliant communication stacks will be used to build up systems of networked medical devices. The performance of the stack implementation is crucial for the usability in real-world medical environments. Therefore, we investigate the latency of currently available middleware stacks: SoftICE, openSDC, and OSCLib. The aim is to evaluate the suitability of the underlying concepts, understanding the communication behavior using different hard- and software platforms, and finding problems to support future development. For the latency measurements we build up a use-case independent testbed and instrument the libraries to get more information. On the one hand, our investigations substantiate the suitability of the underlying concept and the available middleware stack implementations. On the other hand, unexpected results occurred, like a strong dependency of communication latency on the combination of hardware platform, Java Virtual Machine (JVM), and JVM configuration and even a strong dependency on the intensity of exchanged data when using Java middleware implementations.**

I. Introduction

The IEEE 11073 SDC standard family [1] supports a promising new way for the development of standardized vendor-independent interoperability of medical devices based on the known principle of a Service-Oriented Architecture (SOA). The ability to exchange information (fundamental interoperability) is realized by the Medical Devices Communication Profile for Web Services (MDPWS, IEEE 11073-20702) [2] that is derived from the well-established standard Devices Profile for Web Services (DPWS). The Domain Information and Service Model (IEEE 11073-10207) [3] ensures the structural interoperability. It defines the way medical devices describe their capabilities and their state as well as the services that can be used for remote interaction, like get-, set-, or event-service. IEEE 11073-20701 describes the overall Service-Oriented Medical Device Architecture (SOMDA) and defines the binding between the previous two standards. Fig. 2 shows the interaction of the new standards and the integration into the whole communication stack. While the focus of the established IEEE 11073 standard family is on interconnection of two medical devices, IEEE 11073 SDC focuses on systems of multiple networked devices, using a SOA-based approach.

Although the standardization process is not completely finished, there were already several efforts on implementations of IEEE 11073 SDC compliant middleware stacks. Currently, three implementations are available as open-source projects: SoftICE, openSDC, and OSCLib. The IEEE 11073 SDC bases on DPWS and the usage of existing DPWS frameworks as, e. g. the widely-used JMEDS framework [4] is reasonable. JMEDS as well as many web services frameworks are implemented in Java with the focus on productivity. Therefore, SoftICE and openSDC are developed in Java and make use of JMEDS. The third library, OSCLib, is implemented in C++. As such frameworks are complex software projects, this paper omits a detailed description, but can be found in [5] focused on OSCLib.

The requirements for the middleware stacks in terms of latency, reliability, determinism, etc. depend on the actual medical use case. They vary from no over soft real-time up to hard real-time requirements and timing constraints from seconds down to sub-milliseconds. Thus, in this paper we investigate the latency of the available service-oriented medical communication stacks. These implementations are currently not intended to be used for approved medical devices. Therefore, the focus of this paper is a general evaluation of the suitability of the underlying concepts by examining the frameworks, investigating the communication behavior, and identifying problems that will support future developments. We conduct a use case independent communication stack evaluation. As the available libraries are prototypes without quality assurance, we do not take other important characteristics into account like usability, documentation, support, or release stability.

II. Evaluation Setup

A. Testbed

We perform the measurements in an isolated testbed containing one service provider and one service consumer, as displayed in Fig. 1. Both are connected using a D-Link DES-1008D 100 Mbit/s Fast Ethernet Switch and 1 m CAT5 Ethernet cables. Additionally, a control PC is used to start the measuring procedure and collect the measured timestamps

TABLE I: Overview about used evaluation platforms. Timer Res.: Minimal period between two timestamps in C++ and Java. *64 MByte reserved for GPU; **due to Raspbian limitations it behaves like 32 bit ARMv7

PLATFORM	ARCHITECTURE	CPU	CLOCK RATE	TIMER RES.	RAM	DISTRIBUTION	JVMS
Intel Galileo Board (GBoard)	32 bit x86 (Intel Pentium-class)	Intel ® Quark SoC X1000	400 MHz	$5 \cdot 10^{-3}$ ms	256 MByte	Debian GNU/Linux 8.4 Jessie (Kernel 3.8.7)	Oracle JVM 1.8.0_131 Oracle JVM 1.7.0_80 OpenJDK 1.8.0_131 OpenJDK 1.7.0_101
Raspberry Pi 1 Model B (RPi 1)	32 bit ARMv6	ARM1176JZF-S	700 MHz	$1 \cdot 10^{-3}$ ms	512 MByte*	Raspbian GNU/Linux 8.0 Jessie Lite (Kernel 4.4.50-v7)	Oracle JVM 1.8.0_131 Oracle JVM 1.7.0_75 OpenJDK 1.8.0_40 OpenJDK 1.7.0_131
Raspberry Pi 2 Model B V1.1 (RPi 2)	32 bit ARMv7	ARM Cortex-A7	900 MHz	$6 \cdot 10^{-4}$ ms	1024 MByte*		
Raspberry Pi 3 Model B V1.2 (RPi 3)	64 bit ARMv8**	ARM Cortex-A53	1200 MHz	$5 \cdot 10^{-4}$ ms			

after the data exchange has finished. During the measurements, the control PC is physically separated from the actual testbed by unplugging the connecting Ethernet cable. The isolation from other network participants has the advantage that middleware related latencies can be measured without dealing with problems like network overload, packet loss, etc. We use static IPs for the devices hosting the service consumer and provider.

B. Evaluation Platforms

The measurements are performed on different hardware platforms, shown in Tab. I. We have chosen a set of hardware, covering two different instruction set architectures (ARM and x86) and different classes of computation power and memory resources. These platforms represent the heterogeneous devices within operating rooms (ORs) and intensive care units (ICUs). In the focused Point of Care (PoC) medical device domain, the constraints in terms of computation power, memory, power consumption, etc. are not as high as in other domains. Thus, we state that Intel Galileo Board (GBoard) and Raspberry Pi 1 Model B (RPi 1) are suitable representatives.

For a comparison of the available IEEE 11073 SDC implementations (see Section II-C) it is necessary to provide suitable Java Virtual Machines (JVMs). As the Oracle JVM and the OpenJDK have the highest market share, we investigate both JVMs for Java 7 and Java 8. Due to security reasons, Java should be up-to-date. Thus, we use the currently newest stable release version of each JVM. For details see Tab. I. Note, during our evaluation, we noticed that performance disparities can occur even between minor JVM updates providing only bug fixes and security updates.

Unless stated otherwise, we used default platform settings for our investigations, e.g., JVM-mode (*-client*) or default settings of the CPU governor (*ondemand*) for RPis (frequency scaling of GBoard is not provided).

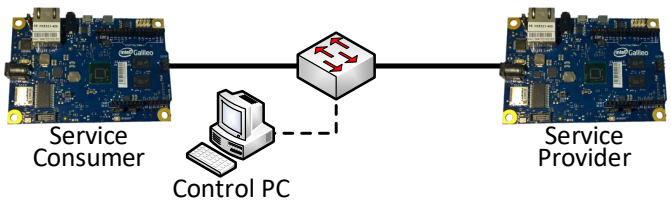

Fig. 1: Physical testbed setup.

C. IEEE 11073 SDC Implementations

For the evaluation within this paper three different implementations of the new IEEE 11073 SDC standard family come into question. These software libraries are made available as open source projects: *openSDC* (Java), *SoftICE* (Java, requires Java 8), and *OSCLib* (C++). Commercial implementations are currently not available on the market. Note, as IEEE 11073 SDC uses a new communication paradigm, other IEEE 11073 frameworks, like Antidote using IEEE 11073-20601, are not compatible.

We evaluate the following library versions: OSCLib 2.0 [6], openSDC version "OR.NET-BETA_06-SNAPSHOT-conhit16" [7] (based on [8]), and SoftICE 0.96b [9]. Note, these are not necessarily the latest available versions. As the IEEE 11073 SDC standard family is currently in the process of standardization, it is not fully stable yet. Therefore, we use the last versions of the libraries being fully compatible among each other. Based on these versions, comprehensive demonstrators have been shown with the complexity of modern ORs, for example at the conhIT exhibitions 2016 and 2017 in Berlin, Germany.

III. MEASUREMENT METHODOLOGY

A. Communication Round Trip Time (RTT)

In IEEE 11073 SDC, and in SOAs in general, there are two different communication patterns: request-response and publish-subscribe. In the request-response pattern, the service consumer invokes a provided service operation of the service provider (request) and gets the corresponding answer from the service provider (response). The get-service works according to this pattern. If the service consumer wants to get informed about state changes of the service provider periodically or episodically (whenever a state changes), the service consumer subscribes to events provided by the service provider. The latter will publish the corresponding events to the subscribers when its state has changed or after a certain period. IEEE 11073 SDC uses this publish-subscribe pattern for events and alerts. As remote control operations in the medical domain

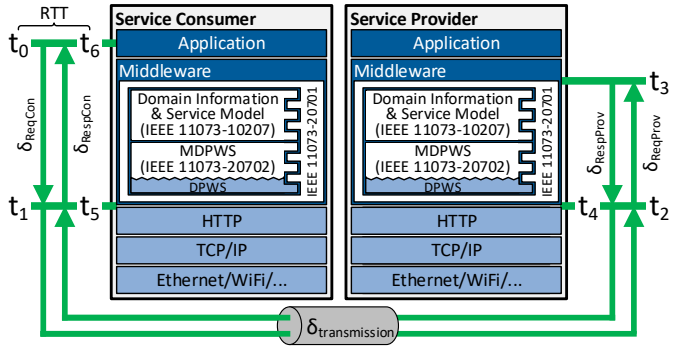

Fig. 2: Timestamps to instrument the libraries.

are not performed as "fire and forget", set-operations are handled as a combination of both mechanisms: Simplified, after the service consumer has invoked a remote control command, it gets informed about the result of the invocation by asynchronous event mechanisms.

In this paper, we investigate the latency of get-operations, by measuring the Round Trip Time (RTT) from invoking a get-operation, until the response is available. More precisely, according to Fig. 2 the RTT is defined as follows:

$$RTT = t_6 - t_0 \tag{1}$$

The first timestamp t_0 is taken when the application software of the service consumer calls the corresponding invoke method of the middleware Application Programming Interface (API). The measurement stops when the responded value is available at the application software of the service consumer to be displayed or processed, represented as timestamp t_6.

Since measuring the RTT takes place only at the service consumer, we do not have to care about time synchronization of service consumer and provider. As described above, set-operations behave quite similar.

Measuring the latency for asynchronous and uni-directional event-based communication requires either time synchronization between service provider and consumer or using the HTTP status response of the client. The latter would be equivalent to investigate get-operations. As events use the same technical mechanisms as get-operations and the behavior can be derived from investigating get operations, we omit events in this work.

B. Middleware Instrumentation

Measuring the RTT provides information about the overall performance of the investigated system. In order to get a deeper understanding of the elapsed times in all parts of the communication we instrumented the different middleware libraries by taking additional timestamps: t_1 to t_5. Fig. 2 visualizes the simplified instrumented communication stack of service provider and consumer. Using external tools like *wireshark/tshark* is not suitable, as the computation overhead on resource-constrained systems has a high performance effect, e.g., SoftICE library on RPi 1 shows an increased RTT of ~ 17 %.

Timestamp t_0 is taken directly before the vendor/device specific application of the service consumer invokes a request by calling a corresponding middleware API function. Afterwards, the middleware software creates the request message, including serialization, and sends the message to the lower layer. Directly before the whole message is sent, timestamp t_1 is taken. Taking additional timestamps in the lower layers (TCP/IP and below) is not possible using the software under our control and would cost much effort. The message can be transmitted using any IP-based network. In our case, we use a 100 Mbit/s switched Ethernet connection. The next timestamp is taken at the service provider at the border between the underlying layer and the middleware software. As the header has to be read to decide whether the incoming message has to be handled, we define timestamp t_2 as the point of time, when the middleware has received the header of the request message, but has not started to handle the message body. The middleware deserializes and interprets the message. All IEEE 11073 SDC middleware implementations under investigation do the metric value provisioning on their own, i.e., the application pushes new values to the middleware but is not called during requests. Therefore, the provider application is not touched. Timestamp t_3 is taken, when the requested metric state has been found. This is the point, where request handling is finished and the response creation starts. Corresponding to timestamp t_1, t_4 is taken immediately before the middleware commits the complete response message to the lower layer. Afterwards, timestamp t_5 is taken when the header of the response message has been received by the middleware software of the service consumer, analogous to t_2. The measurement is finished when the requested value becomes available at the application on service consumer side, represented by timestamp t_6.

1) Delay Calculation: In addition to the RTT calculation given in Eq. 1, the following sub-delays can be defined: service consumer middleware request delay (δ_{ReqCon}), service consumer middleware response delay ($\delta_{RespCon}$), service provider middleware request delay ($\delta_{ReqProv}$), service provider middleware response delay ($\delta_{RespProv}$), as well as the summed service consumer middleware delay ($\delta_{ConMiddle}$), and service provider middleware delay ($\delta_{ProvMiddle}$).

$$\begin{aligned} \delta_{ReqCon} &= t_1 - t_0 & \delta_{ReqProv} &= t_3 - t_2 \\ \delta_{RespCon} &= t_6 - t_5 & \delta_{RespProv} &= t_4 - t_3 \\ \delta_{Stack} &= RTT - \delta_{ReqCon} - \delta_{RespCon} \\ &\quad - \delta_{ReqProv} - \delta_{RespProv} \end{aligned} \tag{2}$$

The fraction δ_{stack} expresses the delay of the transmission of request and response messages between the two middleware instances of service consumer and provider and its calculation is presented in Eq. 2.

2) Accuracy and Comparability: The timestamps t_1 to t_5 are taken within the IEEE 11073 SDC middleware implementations. These implementations are quite complex and use other third-party libraries. Therefore, it is not possible to determine exactly the same defined measurement probe points

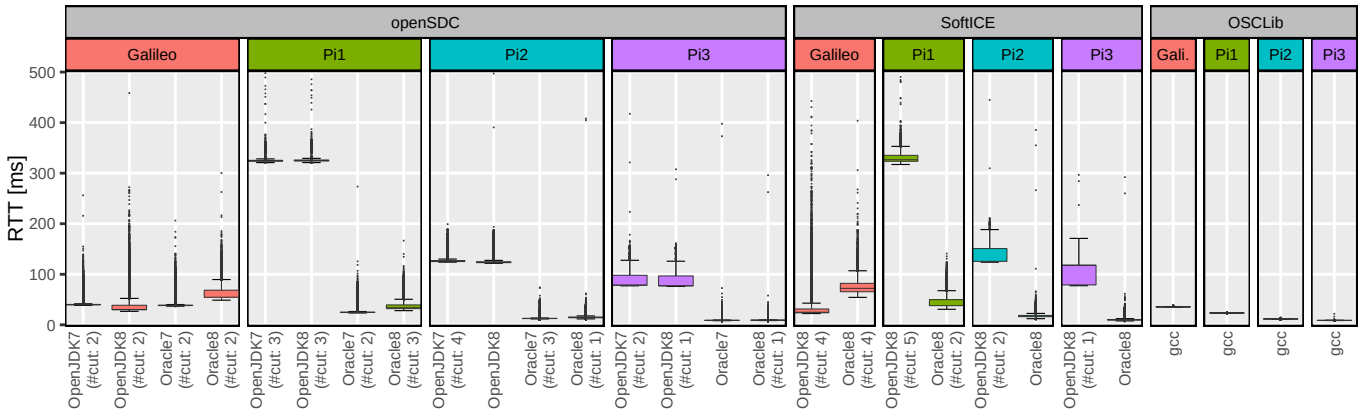

Fig. 3: "Tukey boxplots" (box represents range between 25%-quartile and 75%-quartile; median marked as horizontal line within the box; whiskers (vertical line above and below the box with horizontal line at its end) represent lowest/highest value still within the 1.5 interquartile range (IQR)) of the RTT measurements for combinations of libraries, hardware platforms, and JVMs. Y-axis is cut to 500 ms; number of not displayed values is denoted as #cut. Each box represents 40,000 measurements (aside from omitted values above 500 ms) including all aspects described in Sec. IV. For uncut figure see [10].

in each middleware library. Nevertheless, the probing seems to show decent timings of the examined parts to get a well-informed estimate about the delay portions in all middlewares.

C. Measurement Procedure

For our latency evaluation we performed 40,000 measurements for each combination of middleware library, hardware platform, and JVM, i. e., each calculated median, standard deviation, etc. is based on 40,000 values. In our standard setup we implemented an inter-measurement delay of 20 ms, i. e., a sleeping period between timestamp t_6 of run n and timestamp t_0 of run $n+1$. For the basic evaluations this results into 1,120,000 measurements, each producing seven timestamps. Additional measurement series were performed, e. g., to investigate the effect of XML schema validation (Section IV-A) or the effect of Java just-in-time (JIT) compiling (Section IV-D), resulting in 400,000 additional measurements.

D. Resolution and Technical Realization of Timestamps

Taking the timestamps described previously in this section is done using the mechanisms provided by the implementation languages of the IEEE 11073 SDC middleware libraries. We use the method *java.lang.System.nanoTime()* to take the timestamps of openSDC and SoftICE library and *std::chrono::steady_clock* class for the C++ library OSCLib. Regardless of the formal definition of corresponding timer methods, we experimentally determined the minimal timer resolutions for all platforms as shown in Table I. The worst resolution is 0.005 ms. Therefore this resolution is suitable as the measured RTTs are in the range of tenth of milliseconds. Consequently, we present milliseconds measurements with a resolution of one fractional digit in this paper.

E. Traceability

All measured raw data, the instrumented middleware libraries, and the used implementations of service consumer and provider can be found at [10].

IV. Evaluation

A. Evaluation of RTT Measurements

An overview about all measured RTT values in milliseconds is given in the Fig. 3. Note, we cut the y-axis at 500 ms and omit RTT values above to ensure readability. The number of omitted values which is only a marginal fraction of all measurements, is annotated at the x-axis labels. The complete figure can be found at [10]. We group the measurements by the different IEEE 11073 SDC communication libraries and further divide them into sub-groups by the different hardware platforms. A "Tukey boxplot" is shown for each combination of library, hardware platform and JVM.

The (sometimes huge) differences of RTT latencies between the different JVMs running on the same hardware platform hosting the same middleware implementation can be seen clearly. For example, comparing the median RTTs of openSDC, running on a RPi 1, using the JVMs OpenJDK8 (325.0 ms) and Oracle7 (24.7 ms) shows a difference of factor greater than 13. For the RPis, having an ARM architecture, it can generally be seen that the Oracle JVMs perform significantly better than the OpenJDK JVMs. This behavior shows that Oracle has been optimizing the ARM implementation of its closed source JVMs in comparison to the open source reference OpenJDK. One of the main differences is the absence of a highly optimized JIT compiler for the ARM architecture in OpenJDK. There are ambitions to port a JIT compiler equivalent to Oracle HotSpot, which is maintained in [11]. However, the OpenJDK version available in Raspbian does not provide this JIT. In its standard options, an interpreting mode JVM called *zero* is used.

Comparing RPi 1, 2, and 3 using openSDC and SoftICE the percentage difference between the JVMs decreases with increasing system capabilities. While the median RTTs do not vary much between Oracle 7 and Oracle 8 JVM, for the openSDC library running on the different RPis, the Oracle 7 JVM reveals the best performance. As SoftICE requires Java 8, Oracle 8 JVM has the lowest RTTs on the ARM architectures.

TABLE II: RTT measurement results for the standard schema validation configuration. Values in parenthesis are the results with schema validation for both incoming and outgoing messages. Results using the JVM with best performance for each system are displayed: [1]OpenJDK8; [2]Oracle 7; [3]Oracle 8

	Galileo			**Pi1**			**Pi2**			**Pi3**		
	openSDC[1]	**SoftICE**[1]	**OSCLib**	**openSDC**[2]	**SoftICE**[3]	**OSCLib**	**openSDC**[2]	**SoftICE**[3]	**OSCLib**	**openSDC**[2]	**SoftICE**[3]	**OSCLib**
Median [ms]	30.3 (39.2)	24.9 (46.2)	35.4	24.7 (33.7)	42.1 (86.1)	23.5	12.6 (17.0)	17.4 (30.5)	11.7	8.8 (12.2)	9.8 (21.6)	8.8
Std.Dev. [ms]	20.4 (27.4)	25.5 (37.0)	0.3	10.5 (12.8)	11.8 (15.7)	0.3	5.9 (7.0)	4.9 (6.8)	0.3	4.4 (4.7)	4.1 (6.0)	0.1
Max [ms]	1784.9 (1458.2)	1194.4 (1339.8)	38.7	1228.9 (1313.4)	859.5 (1001.6)	25.3	630.4 (676.8)	385.5 (854.9)	14.4	397.6 (419.8)	292.0 (803.9)	21.9

As the GBoard has an x86 CPU, we get a different picture. Although the differences are not as obvious as for the RPis, it can be seen that the OpenJDK 8 has the best RTT performance on this platform. As the OpenJDK includes a JIT implementation, which was derived from earlier Oracle JVM versions, small differences seem reasonable. Furthermore, it seems that Oracle's optimizations focus rather on high performance x86 platforms and therefore may be less effective for the resource constrained GBoard.

For the OSCLib we used only the standard C++ environment based on the default gcc versions available in both distributions. According to the increasing CPU clock rate, the measured RTTs decrease from GBoard, over RPi 1, 2, and 3.

In Table II we present median, standard deviation (Std.Dev.), and maximum value for the different RTT measurements. In contrast to Fig. 3, we grouped this table by hardware platform. To reduce the amount of data displayed, we just present the measurements taken with the best performing JVM and omitted the values of the other JVMs. There are different standard configurations concerning XML schema validation within the three libraries: The OSCLib does a schema validation for every incoming and outgoing message. There are no options implemented to change this behavior. The openSDC library performs a schema validation for every incoming message, but not for outgoing messages by default. This behavior can be changed by the usage of a JVM flag. On the contrary, applications using the SoftICE library typically do not validate any message schema, but an API call is provided to activate full schema validation. To ensure a fair comparison, we additionally performed the measurements for openSDC and SoftICE with schema validation for each incoming and outgoing message, as it is done by OSCLib. These results are displayed in parentheses in Fig. 3.

As measured by the median RTT (with schema validation), the C++ implementation OSCLib has the best performance, closely followed by openSDC. While there is a clear difference to SoftICE on the RPi 1, the gap decreases with increasing resources on RPi 2 and 3.

Unexpectedly, on the GBoard, the difference between the libraries is much smaller, e. g., factor ~1.2 between SoftICE and openSDC, compared to factor ~2.6 at RPi 1 or ~1.8 at RPi 2 and RPi 3. For the measurements with standard schema validation the rank even changes and SoftICE has the best performance although openSDC and OSCLib have a better performance with the same configuration on the RPis. This shows that the software latency on the platforms with more resources (like the RPi 3) tends to be more dominated by other factors than the algorithmic complexity, i.e., the latency of I/O operations caused by the JVM implementation or operating system. Since the GBoard has a lower computation power than the other platforms, the algorithmic complexity has a greater impact and the OSCLib with schema validation for all messages reveals the highest latency.

Schema validation has a significant impact on the performance. For SoftICE the delay increases by factor ~1.8 – 2.2. In the case of openSDC the growth is lower (factor ~1.3 – 1.4) as schema validation is only added for outgoing messages. Whether it is necessary to perform schema validation for all messages, incoming messages, or no messages, has to be discussed according to the use case and risk management of the manufacturer that is necessary for the different applications.

B. Standard Deviation and Outliers

The latency evaluation of communication middleware implementations cannot be reduced to the median RTT. Depending on the application, variance/standard deviation and maximum latency are important additional parameters to evaluate whether a system performs well or not. Therefore, we present the standard deviation as well as maximum RTT values in Tab. II. Additionally, the boxplots of Fig. 3 illustrate the main aspects of the measurement value deviation.

1) C++ Implementation: The C++ library OSCLib has a small standard deviation. Except for the RPi 3, the maximum RTT value is less than 3.5 ms above the median. The clear outliers (in terms of percentage) might be results of non-deterministic delays caused by the operating system. This could be fixed by using a real-time capable operating system.

2) Java Implementations: Due to garbage collection (GC) of JVMs, the Java implementations have a much higher standard deviation than the C++ implementation, especially when running on the more resource-constrained GBoard and RPi 1. The effect of GC is traced in Fig. 4. To provide a deeper analysis of the results, we choose the openSDC library, running on a RPi 3, using Oracle 7 JVM. The discussed aspects can be transferred to all Java-based measurements.

In Fig. 4 we present a comparison of a small subset of RTT measurements. Firstly, we will discuss the measurement series represented by green dots, each representing one RTT measurement using standard GC mechanisms. The GC can be investigated by starting service consumer and provider using JVM parameters to display GC information, like *-verbose:gc* (for basic information). This additional output shows, that the

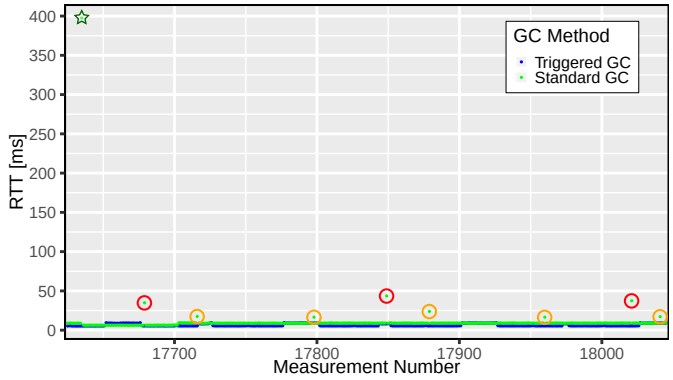

Fig. 4: Latency characteristics of different GC strategies. Subset of measurements is shown, excluding aspects discussed in Sec. IV-D.

GC is started on service consumer and provider with different but nearly constant periods. As Java GC is mostly triggered by memory pressure, the periods can change unpredictably. Nevertheless, these investigations give us strong evidence that the measurements marked with orange circles are influenced by GC at the service consumer and the measurements marked with red circles are influenced by GC at the service provider. These highlighted measurements are moderate outliers. The two frequencies of GC at service consumer and provider can be followed in the chart. All of the JVMs under investigation use a two phase GC when started with standard options. In addition to the "normal" GC, which mostly collects short-living objects, a so-called *Full GC* takes place and tries to clean objects in the whole heap memory. Also the point of time where a *Full GC* occurs is hardly predictable. As it suspends the execution significantly longer, it causes massive outliers as marked with the dark green star (top left of Fig. 4). In this case, measurement number 17,635 has the maximum RTT value of 397.6 ms.

Standard JVMs provide only very limited possibilities to cope with the non-deterministic character of the GC. According to the documentation, calling the method *System.gc()* is a "suggestion" to the JVM to perform a (full) GC. There is neither a guaranty that the GC is performed at the moment the method is called, nor that it is performed at all. Nevertheless, we implemented an extension to openSDC service consumer and provider, breaking the measurements and triggering the GC manually. After triggering the GC there is a waiting period of one second before the next measurement is performed. For this investigation, we suggest performing an GC every 25 measurements. The result is visualized by the blue dots in Fig. 4. As there are no blue outliers, it can be recognized that no GCs take place, beside the ones explicitly triggered. This series of 40,000 measurements has the following characteristics: median RTT: 5.5 ms; standard deviation: 1.8 ms; maximum value: 34.5 ms (this high maximum value is caused by the JIT compiler, as described in Section IV-D).

Having a look at the whole amount of measurements it can be stated that no *Full GC* has taken place and that no or only a very limited number of unplanned GCs occurred. (Due to limited space in this paper, the whole deviation of RTT values is not displayed and, to ensure readability, Fig. 4 shows only a subset of all measurements.) In another run we have seen a single outlier within a measurement using the described mechanism of periodic GC triggering. This underlines the non-deterministic character, as this outlier can be caused either by an additionally performed *Full GC* by the JVM or by a GC taking a longer time than the configured sleeping period between the measurements after the GC has been triggered. This small experiment shows that triggering the GC from application code can only give hint that it is possible to improve predictability of execution times of Java applications. The initial memory management can also have a positive effect. For this paper, we only used the JVM standard configuration concerning initial memory size. As broadly discussed in the Java developer community, triggered GC, as we have experimentally performed, could have serious disadvantages.

Additionally, there is a need of a predictable amount of time depending on the activity of the software running on the JVM, while no other task needs to be run. Otherwise, the manually triggered GC will needlessly suspend the application, as a *Full GC* takes longer as a normal GC and is usually not needed as often. Unfortunately, the standard Java API only provides a trigger to run *Full GC*. Whether reserving time for GC is possible or not depends on the application scenario and cannot be decided in general.

To really cope with this issue, a dedicated real-time JVM has to be used and the implementation should follow the Real-Time Specification for Java (RTSJ). Konieczek et al. [12] have shown a real-time capable implementation of the Constrained Application Protocol (CoAP) based on Java. CoAP is another Internet of Things (IoT) middleware specification roughly comparable to DPWS that is the base technology of IEEE 11073 SDC. Additionally, a real-time capable operating system has to be used to ensure deterministic delays. As mentioned earlier, this is a general requirement, independent from the way the middleware is implemented (Java, C++, etc.). Concluding the work in [12] and our experimental results, we state that a deterministic middleware implementation is possible using Java.

C. Breakdown of RTT

As described in Section III-B the RTT is composed of different sub-latencies. Fig. 5 presents the percentage deviation of the delays taking place during the different phases of the communication: Request creation at service consumer middleware (δ_{ReqCon}), request handling at service provider middleware ($\delta_{ReqProv}$), response creation at service provider middleware ($\delta_{RespProv}$), and response handling at service consumer middleware ($\delta_{RespCons}$). Additionally, there is the phase between the middleware (δ_{Stack}), containing the HTTP implementations and TCP/IP stacks of both participants as well as Ethernet transmission time, including switch delay. As we use a Fast Ethernet switch in store and forward mode and an upper bound of 30 μs for the switch delay as measured

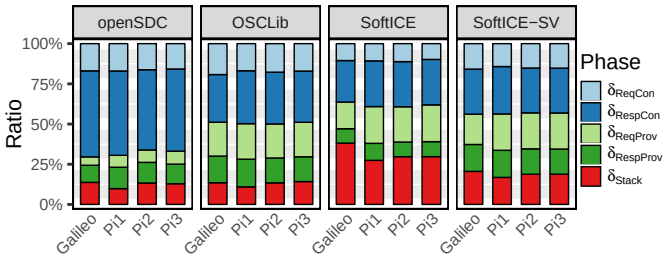

Fig. 5: Measured fractions of RTT as defined in Eq. 2.

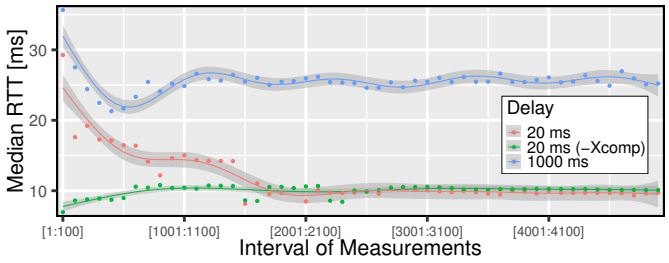

Fig. 6: Time dependent latency progression. Each dot: median of 100 values. First 5,000 of 40,000 measurements shown.

in a previous work at our institute [13], we roughly determine an upper bound of 300 μs for plain Ethernet transmission. As shown in Table II, this relatively small, mostly constant delay ranges from 0.3 % to 3.4 %. All other δ_{Stack} components are highly platform dependent.

The different parts of the bars within the chart are grouped by phases taking place at the service consumer (dark and light blue) and at the service provider (dark and light green) and they are not displayed in the order they have taken place during the communication. We present the results for all libraries and all hardware platforms with each using the JVM with the best performance. As already discussed, for a detailed comparison the different configurations concerning schema validation have to be taken into account. Therefore, we also added the measurements with enabled schema validation for SoftICE into Fig. 5, labeled as SoftICE-SV.

It can be recognized that the time elapsing during the processing of the middleware is much higher than the time elapsing in the lower communication stack layers and the network transmission (δ_{Stack}). The percentage of δ_{Stack} in case of openSDC and OSCLib is less than 15 %. For SoftICE the portion is higher. The ration of processing time defers with schema validation, as the schema validation takes a significant period of time (see also Tab. II.) When using SoftICE with schema validation, the percentage is less than approximately 20 %. We draw the following conclusion from these results: For optimizing the latency of the communication system, the main focus should be on optimizing the middleware, as the main period of time elapses here. Looking at the difference between SoftICE and the other two libraries concerning the percentage of δ_{Stack}, the performance of the used HTTP implementations should be investigated.

It can be seen that the latency caused by service consumer and provider is approximately equal for SoftICE and OSCLib. In the case of openSDC, it can be recognized that the delay caused by the service provider is much lower than the delay caused by the service consumer: approximately a factor of 3 to 4.5. As service providers are typically more resource-constrained than the service consumer, care was taken during the development of IEEE 11073 SDC that the service consumer can be as simple as possible with respect to medical safety issues. On the one hand, it could be inferred that therefore greater efforts were made on service provider side by the openSDC developers. On the other hand, there seems to be space for optimization at the service consumer side.

D. Analysis of RTT Values Over Time

To investigate the performance of the middleware implementations we also analyzed the change of measured latency values over time. For the C++ implementation OSCLib the values just vary around the median, as expected and already discussed. Having a deeper look into the Java implementations openSDC and SoftICE, a huge influence of the Java just-in-time (JIT) compiler can be seen. As a representative Fig. 6 shows the change of RTT values over time for the SoftICE library running on a RPi 3 using Oracle 8 JVM. This analysis is transferable to the other platforms, if also a sophisticated JIT is used, as generally in the Oracle JVM and in the x86 implementation of OpenJDK.

Each dot in the chart represents the median RTT of an interval of 100 measurements. On the x-axis the intervals [1:100] to [4901:5000] are plotted, which represent measurements in the warm-up phase. Afterwards, there are no significant changes in the behavior. The red dots represent the standard configuration of our measurement setup: 20 ms delay between the single measurements and standard configuration of the JVM. To get a better overview, we added a smooth line to the chart. A significant reduction of the median RTT values of the intervals can be seen from the start of measurements approximately until 2,500 measurements. During this time the values decrease by factor ~2.5. We investigated the memory management of the JVM. As the used functional parts stay the same over time and the GC will mostly have no decreasing time effort, the reason for this behavior is the JIT compiler and its optimizations. JIT compiling means that the JVM starts running applications by interpreting the bytecode. During runtime critical parts of the code will be identified and will be compiled. Running compiled code is typically faster than running interpreted code. The observed issue is that the JIT compiling causes a “warm-up phase”, which is a well-known behavior of Java applications. A significant long period of ~2,500 exchanged request/response messages for the optimization is surprisingly long and has to be taken into account for real-world applications. A reason for this behavior could be the complexity of IEEE 11073 SDC middlewares and their complex optimization process. Another explanation could be, that it takes fairly long for the JIT to identify all routines in the critical path of the middleware to trigger the compilation.

Our analysis that the increasing performance is caused by the JIT compiler is supported by the green dots and smooth

line in Fig. 6. The green part presents the median RTT values of the intervals of a length of 100 using the same measurement setup with 20 ms delay between the single measurements but starting the JVM using the flag "*-Xcomp*". This flag forces the JVM to compile the whole code during startup of the application. No JIT compiling is done during runtime. Nearly constant median RTT values can be seen over time. Typically, it is not recommended to use the *-Xcomp* flag as the startup process becomes longer (in this case from ~6 s to ~13 s) and no further optimization happens during runtime. This can be seen on the long run: Comparing the red and the green graph it can be seen that RTT is slightly higher using the *-Xcomp* flag than using the JIT compiler. For the whole 40,000 measurements the median RTT using JIT compiler is 9.8 ms, for *-Xcomp* flag usage it is 10.1 ms. Whether precompilation should be used or not has to be decided according to the concrete application and its requirements.

Additionally, we performed a measurement using a longer delay between the single measurements of 1,000 ms. The results are shown by the blue dots in Fig. 6. It can be seen that the JIT compiling works significantly worse compared to an inter-measurement delay of 20 ms. The median RTT values are ~2.5 times higher. For the whole 40,000 measurements the median RTT is 25.0 ms. This shows that the JIT does not perform as well as it does for a higher communication intensity. Thus, we draw the conclusion that the behavior and efficiency of the JIT compiler unexpectedly has a strong dependence on the amount and frequency of exchanged data. This has to be taken into account for real-world medical applications, as the traffic patterns may vary heavily.

V. Related Work

There exist studies like [14], which support our methodology regarding differentiation between JVMs, JIT compiling, and GC tuning, but this meta study focuses on x86 architectures and synthetic benchmarks on older JVMs. Also many non-scientific reports reflect performance dependencies also for ARM architectures as, e. g., Oracle's comparison in [15]. However, it is not as systematic as the presented measurements and it does not include Java 8 virtual machines. Since the fairly recent nature of the implementations of IEEE 11073 SDC middlewares, to the best of our knowledge, there exist no other performance studies focused on actual implementations of a middleware in a similar setup.

VI. Conclusion and Future Work

In this paper we presented latency measurements for service-oriented medical device communication middleware implementations being compliant to the new IEEE 11073 SDC family of standards. Therefore, we investigated the three currently available middleware stacks: openSDC, SoftICE, and OSCLib. We like to conclude the paper by deriving the following theses from our investigations:

Conclusion 1: All investigated IEEE 11073 SDC compliant communication stacks substantiate both their applicability and the adequacy of the new communication concept.

Conclusion 2: The performance of one and the same Java-based middleware implementation highly depends on the combination of hardware platform, used JVMs, JVM configuration, and traffic pattern. For each particular medical device and its application, an adequate system configuration has to be profiled. General recommendations are not possible.

Conclusion 3: For Java libraries, the traffic pattern and communication intensity have a high influence on the JIT compiling and therefore on the communication latency.

Conclusion 4: Validation of the complex IEEE 11073-10207 XML schema has a strong impact on the middleware latency.

Conclusion 5: Relating to the overall communication latency, the main field for improvements is the concept and implementation of the middleware (mainly on application layer), as this part causes the main portion of latency.

As we performed a generic evaluation, the results have to be judged related to a particular medical application scenario. In the future, traffic patterns should be derived from real medical device ensembles for deeper performance analyses based on realistic communication scenarios. Additionally, the available libraries can be enhanced based on the presented results, like reducing latency and improving real-time capability.

References

[1] S. Schlichting and S. Pöhlsen, "An architecture for distributed systems of medical devices in high acuity environments - A Proposal for Standards Adoption," in *11073/HL7 Standards Week*, San Antonio, USA, 2014.

[2] M. Kasparick, S. Schlichting, F. Golatowski, and D. Timmermann, "Medical DPWS: New IEEE 11073 Standard for Safe and Interoperable Medical Device Communication," in *IEEE Conference on Standards for Communications and Networking (CSCN)*, Tokyo, Japan, oct 2015.

[3] ——, "New IEEE 11073 Standards for Interoperable, Networked Point-of-Care Medical Devices," in *Int. Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*, Milan, Italy, 2015.

[4] "JMEDS (Java Multi Edition DPWS Stack)." [Online]. Available: https://sourceforge.net/projects/ws4d-javame/

[5] A. Besting, D. Stegemann, S. Bürger, M. Kasparick, B. Strathen, and F. Portheine, "Concepts for Developing Interoperable Software Frameworks Implementing the New IEEE 11073 SDC Standard Family," in *17th Annual Meeting of the International Society for Computer Assisted Orthopaedic Surgery (CAOS)*, Aachen, Germany, 2017, pp. 258–263.

[6] SurgiTAIX AG, "Open Surgical Communication Library (OSCLib)," https://github.com/surgitaix/osclib/commit/df860802c19522bc59af bd9007e9d06d6c7b174b.

[7] "openSDC Source," https://gitlab.amd.e-technik.uni-rostock.de/opensdc/opensdc-main/commits/conhIT16_17.

[8] "openSDC," https://sourceforge.net/projects/opensdc/.

[9] SurgiTAIX AG, "SoftICE - Software for the Integrated Clinical Environment (ICE)," https://bitbucket.org/surgitaix/softice/commits/529a75b558a6dd402fafe4fa0132fb08f5461ca1?at=master.

[10] "Git Repository with all modified Libraries and Measurements," https://gitlab.amd.e-technik.uni-rostock.de/middleware_latency_measurements/mod_libs_data/commits/SOCNE2017.

[11] "openJDK ARM32 port," http://openjdk.java.net/projects/aarch32-port/.

[12] B. Konieczek, M. Rethfeldt, F. Golatowski, and D. Timmermann, "Real-Time Communication for the Internet of Things using jCoAP," in *18th IEEE Symposium on Real-Time Computing (ISORC)*, 2015, pp. 134–141.

[13] E. B. Schweißguth, P. Danielis, C. Niemann, and D. Timmermann, "Application-Aware Industrial Ethernet Based on an SDN-Supported TDMA Approach ," in *Proceedings of the 12th IEEE World Conference on Factory Communication Systems (WFCS)*, 5 2016, pp. 1–8.

[14] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, no. 10, pp. 57–76, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1297105.1297033

[15] J. Connors, "Comparing Linux/Arm JVMs Revisited," https://blogs.oracle.com/jtc/comparing-linuxarm-jvms-revisited.