DE GRUYTER OPEN

Logistics & Sustainable Transport
Vol. 7, No. 1, October 2016, 41–50
doi: 10.1515/jlst-2016-0004

For Safety and Security Reasons: The Cost of Component-Isolation in IoT

Alexander ZUEPKE, Kai BECKMANN, Andreas ZOOR and Reinhold KROEGER
RheinMain University of Applied Sciences, Wiesbaden, Germany

***Abstract*— The current development trend of Internet of Things (IoT) aims for a tighter integration of mobile and stationary devices via various networks. This includes communication of vehicles to roadside infrastructure (V2I), as well as intelligent sensors / actors in Logistics and smart home environments.**

Compared to isolated traditional embedded systems, the exposure to open networks increases the attack surface, and errors in the networking components could compromise the safety and security of the embedded application or the whole network. But often current system architectures for mass-market IoT devices lack the required isolation concepts.

Using a partitioning microkernel and enforcing the use of a microcontroller's memory protection unit (MPU) facilities, we compare different isolation concepts for a publish/subscribe middleware implementing OMG's Data Distribution Service (DDS) standard and we evaluate our results on an STM32F4 microcontroller. The results of this case study show moderate costs for increased memory usage and additional context switches.

***Key words*— Component-Isolation, Microkernel, Partitioning, IoT.**

I. MOTIVATION

Today, low-cost microcontrollers are quite powerful and offer wireless networking facilities. Based on a network of connected sensors and/or actors, this allows the design of new embedded applications. The umbrella terms Cyber-Physical Systems (CPS) and Internet of Things (IoT) cover a wide range of applications in domains like Ambient Living, Smart Cities, and Intelligent Transportation Systems & Logistics. Thus, formerly isolated and domain specific networks now become connected to the Internet.

But the first generation of networking-capable microcontrollers lacked support for isolation concepts. Any potential error in the application code or the protocol stacks could compromise the whole embedded system or open a door into a protected network. This makes early controllers unsuitable for safety- or security-critical applications, such as Vehicular Ad-hoc Networks (VANETs) communication to roadside infrastructures or critical transportation systems.

Recently, low-cost 32-bit microcontrollers featuring memory protection units (MPU) became available. They enable isolation of application code and protocol stacks, not only from each other, but also from the rest of the system, for example from uninvolved real-time tasks.

Unfortunately, a lot of current software platforms for IoT devices, like FreeRTOS, Contiki, or RIOT-OS, originate from real-time operating systems (RTOS) without or only limited MPU support. On the other end of the spectrum, operating systems explicitly designed for safety-critical applications, like VxWorks, Integrity, or PikeOS, as well as general-purpose operating systems like Linux and QNX, often require a full virtual memory management unit (MMU) in the processor and therefore target higher-priced platforms.

Additional safety and security do not come for free: isolation concepts usually add costs by increased memory usage due to storing multiple copies of data. Also, isolation can impose a significant performance overhead by multiple data copy operations between isolated software components or additional context switches and related reprogramming of memory protection hardware.

This work analyses these costs based on a case study of porting a data-centric middleware onto a microkernel platform for safety-critical applications. The main safety objective here is to allow multiple applications to communicate over the network via the middleware, while keeping potential errors isolated to the erroneous software components. And for security reasons, software components must be able to prevent other unrelated components from accessing their internal data, e.g. encryption

keys or personal data. This is especially important when software components of different vendors need to be integrated on a single platform.

Applied to logistics, an example would be an intelligent sensor integrating two different applications on a single platform for cost reasons: one application (provided by the logistics company) tracks the shipment, while a second application (provided by the goods producer) tracks the integrity of sensitive goods, for example if a medical product was transported at its required temperature. Isolating the applications increases trust and keeps business data separated.

In this paper, the system's software components are decomposed into isolated containers. The system comprises (1) two synthetic test applications sending and receiving data, (2) sDDS [1], a publish/subscribe middleware based on the Data Distribution Service (DDS) standard [2], (3) the lightweight TCP/IP stack (lwIP) [3], (4) an Ethernet driver for the STM32F4 microcontroller, and (5) AUTOBEST [4], a small partitioning microkernel developed for safety-critical automotive use cases. These software components form a vertically layered system model, where higher level components rely on services of the lower level components. This allows applying and evaluating isolation concepts at each of the layer interfaces.

The proposed system is comparable to AUTOSAR [5], the established software architecture for automotive electronic control units (ECUs). AUTOSAR uses a three layered software-model comprising an application layer, the Runtime Environment (RTE) providing data exchange, and the Basis Software (BSW) hosting device drivers and software stacks. However, in contrast to AUTOSAR, our setup is simpler and easier to analyse due to the larger size of its building blocks, but the general results are applicable to AUTOSAR as well.

The rest of this paper is organized as follows: Sec. II introduces the software components of this study in detail. In Sec. III, we discuss different concepts where to apply isolation techniques between the component interfaces and select one approach for evaluation. This approach is compared to a baseline without isolation in Sec. IV. Sec. V summarizes and discusses the results and compares the approach to related work. Finally, Sec. VI concludes the paper.

II. Software Architecture

In this section we introduce the used software components in detail and discuss their interfaces. All components are customizable to a high degree regarding their resource usage, making them suitable for IoT systems. The hardware platform is a Texas Instruments STM32F4 microcontroller which includes a Cortex-M4-based ARM core clocked at 168 MHz, 112 KiB SRAM, 1 MiB flash memory, and 100 MBit/s Ethernet.

A. AUTOBEST

As operating system, we use AUTOBEST [4], which was developed and implemented at the authors' faculty in a research project together with Easycore GmbH in Erlangen, Germany. Application areas are safety-critical automotive use cases following the standard ISO26262 [6], which requires "freedom of interference" between independent software components. The kernel targets embedded microcontrollers with memory protection (MPU) support and is fully statically configurable.

AUTOBEST is based on a small microkernel, which isolates different applications into so-called partitions. A partition entails an isolation boundary in (scheduling) time and (address) space and comprises a set of executable tasks (threads) plus necessary system resources for synchronization inside the partition. All communication and synchronization mechanisms across partitions are statically configured, allowing fine grained access control at system compile time.

AUTOBEST is not limited to automotive use cases: The kernel implements an abstract programming model supporting multiple operating system APIs implemented in domain specific libraries inside the partitions, as Fig. 1 shows. Currently, AUTOBEST support AUTOSAR for automotive use cases, as well as an ARINC 653 [7] application executive for avionics.

Logistics & Sustainable Transport
Vol. 7, No. 1, October 2016, 41–50
doi: 10.1515/jlst-2016-0004

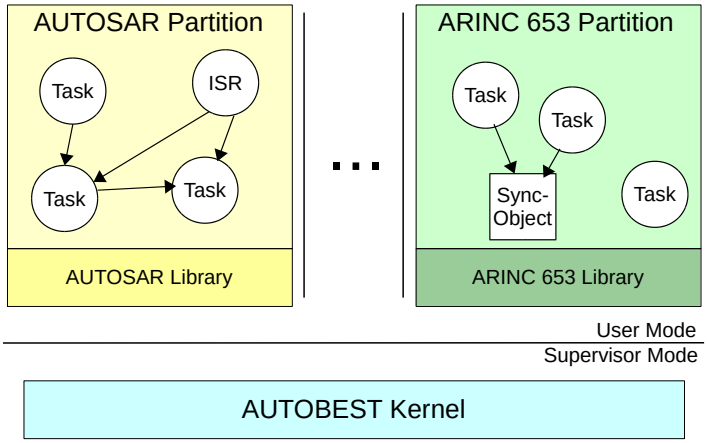

Figure 1: AUTOBEST system architecture with two different partition types. Libraries in user space implement the differences of the domain specific APIs on an abstract microkernel interface.

Data exchange between partitions is realized by dedicated shared memory segments (SHMs) between communicating partitions. Code generators provide the structure of the SHMs at compile time. The system supports linking of arbitrary variables in SHM with kernel wait queues. This allows the construction of blocking synchronization means, like the producer-consumer pattern, by updating the variables atomically in the fast path, or calling into the kernel to wait or wake up tasks on the other side. Also, tasks in one partition can notify other partitions asynchronously via inter-partition events. Lastly, AUTOBEST provides a synchronous Remote Procedure Call (RPC) to call functions in other partitions.

B. sDDS

The Object Management Group's (OMG) standard Data Distribution Service (DDS) [2] specifies a vendor- and platform-neutral data-centric middleware following a publish-subscribe paradigm. The standard defines a set of configurable Quality-of-Service (QoS) properties regarding real-time message delivery, redundancy, persistence, and resource constraints. The architecture of DDS is based on the concept of a global data space (domain), in which participating nodes share data over a network as publishers and subscribers. The exchanged data is structured in application-specific data types. A topic links such a data type and its QoS properties with a globally unique name. The application interface to the data space comprises DataWriter and DataReader classes. To get access to meta-information, like the available nodes and topics in the system, DDS provides so-called built-in topics. Originating from military applications, the development of the DDS standard today is mainly driven by industrial applications with a promising future in application areas like *Internet of Things* and *Industry 4.0*.

Although developed for distributed embedded systems, implementations of the DDS standard do not support resource constrained hardware platforms like those used in wireless sensor networks and IoT environments. Therefore, for this work, sDDS (sensor network DDS) [1] is used. Applications for sensor networks or IoT typically allow for statically configured nodes and only use a limited subset of the DDS middleware functionality. sDDS is based on a model-driven software development process, which collects the application requirements regarding its specific computing and sensor capabilities and generates an individually tailored middleware implementation for each node. This allows for deploying DDS on heterogeneous platforms, ranging from 8-bit microcontrollers to standard PCs, and simplifies both horizontal and vertical integration.

Currently, sDDS supports a configurable subset of the DDS standard. Besides simple data exchange with callbacks and polling, sDDS supports some QoS features important for sensor networks. Communication between application components can be static, dynamic, or a combination of both. For dynamic communication relations, sDDS uses a discovery mechanism based on built-in topics.

Focusing on platform independence, sDDS is implemented in C, and its API is conforming to the DDS standard. In Fig. 2 an overall architecture of sDDS is outlined. Operating system and platform dependent functionality is abstracted via internal modules: sDDS requires dynamic heap memory for initialization, cyclic activation of internal callbacks, and mutual exclusion. On the networking side, sDDS depends on a datagram delivery service with routing and broad- and multicast functionality,

like UDP/IP, by the underlying operating system. The discovery mechanism requires multicast groups. Due to the focus on IoT use cases, sDDS uses IPv6.

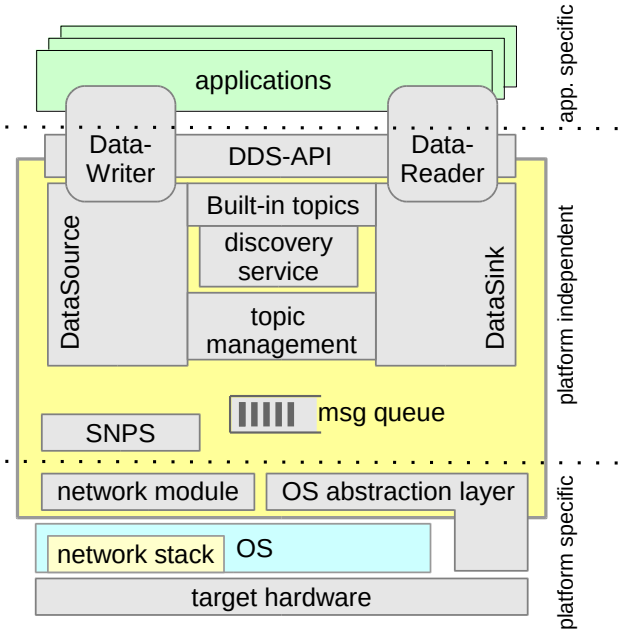

Figure 2: Overall architecture of sDDS. Divided in three layers, platform-specific, platform-independent, and application-specific.

C. lwIP

For this work, we integrated the highly customizable open source lightweight TCP/IP stack (lwIP) [3] to connect sDDS to an Ethernet driver. sDDS only uses the UDPv6 functionality of lwIP and its IPv6 Stateless Address Auto Configuration (SLAAC) mechanism. LwIP provides two component interfaces: the interface towards a network driver sends and receives single Ethernet frames, and the interface towards the application side (in our case sDDS) handles datagrams via UDP. Both interfaces use decoupled buffers following the producer-consumer pattern. LwIP internally manages the protocols (Ethernet, IP, UDP) fully transparently for the application.

LwIP also has built-in dynamic memory management for data and Ethernet frames, so called pbufs. By reserving spare space at the front and at the end of a pbuf, lwIP can add the necessary protocol headers for outgoing data. Likewise, lwIP strips off the headers from incoming frames without copying overhead. Frames larger than 536 bytes of user data (the default TCP maximum segment size) can be split into multiple linked pbufs.

D. Test Applications

In the presented scenario, two applications run on the embedded target on AUTOBEST and communicate with a third application on a Linux host. An sDDS specific IDL compiler generates the necessary middleware components, application-specific initialization, and application stubs according to the system model.

The first application cyclically publishes data for the topic *Alpha* using the synchronous DDS method call `DDS_AlphaDataWriter_write()`, while the second application receives data from the topic *Beta*. To keep latency low, the application registers a *listener* callback at the sDDS middleware. This callback is called when new data becomes available. Then the application reads the topic-specific data (samples) using the topic-specific method `DDS_BetaDataReader_take_next_sample()`. This non-blocking call fetches the data out of the input queue and acknowledges the arrival to sDDS. Both topics have the same data structure with a size of only 2 bytes to keep the overhead small for copy operations during measurements.

The third application on the Linux system is the counterpart for the two applications on the embedded system. It implements the roles of subscriber and publisher for the topic *Alpha* and *Beta*, respectively. Besides that, the realisation is mostly identical.

Logistics & Sustainable Transport
Vol. 7, No. 1, October 2016, 41–50
doi: 10.1515/jlst-2016-0004

III. ISOLATION CONCEPTS

In a traditional monolithic system, the software components discussed in Sec. II would be placed in a vertically layered model, as Fig. 3 shows on the left. But here, on the microkernel AUTOBEST, the components can be *isolated* into horizontally ordered partitions. The primary goal is to have multiple isolated applications communicating over the network via sDDS. Isolation enables fault containment and increases information security, but induces communication overhead (in terms of memory usage and performance impact). Therefore, the secondary goal is to keep the communication overhead between partitions low.

The obvious places for such *splits* into dedicated partitions are the component interfaces, as Fig. 3 shows. At such a split, the component interface is mapped onto the microkernel's communication mechanisms and a shared memory interface. In the following sections, we discuss multiple scenarios where to apply a split.

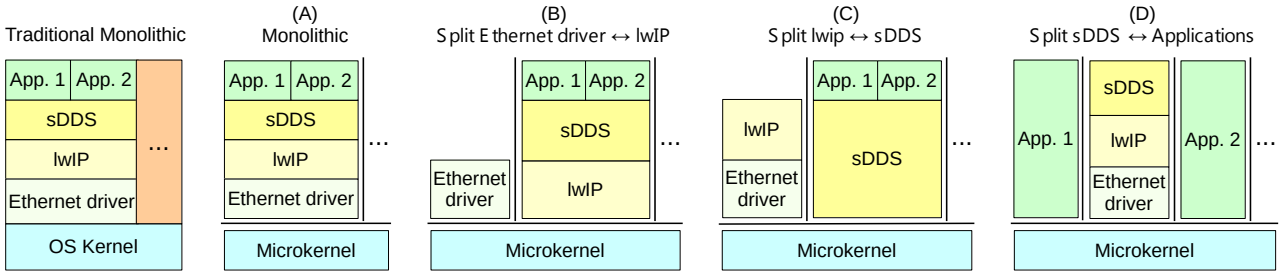

Figure 3: Variants. From left to right: monolithic approach without separation, (A) monolithic approach with isolation of networking components, (B) split between Ethernet driver and lwIP, (C) split between lwIP and sDDS, and (D) split between sDDS and the applications. Adjacent boxes represent a partition. Dots refer to other, unrelated partitions.

A. *Isolation of Networking Components from Rest of System*

The simplest variant of isolation is to keep all networking-related software components in a single partition. This allows isolating the applications, sDDS, lwIP, and the Ethernet driver from the rest of the system. Such an approach is acceptable for scenarios where unrelated parts of the system must not be affected by faults in the network stack, for example if the network application implements remote monitoring of a real-time control application.

We use this scenario as baseline for comparison, because here all component interfaces can be implemented by function calls, and data can be passed indirectly via pointers instead of costly copy operations. This variant is also comparable to implementations in other IoT operating systems without isolation concepts.

B. *Isolation between Ethernet Driver and lwIP*

Starting at the bottom of the network stack, the first split can be applied between the Ethernet driver and lwIP. Besides the initial setup of Ethernet addresses and the network's link status, the component interface between the Ethernet driver and lwIP mainly comprises the exchange of Ethernet frames. Both incoming and outgoing traffic may happen in bursts. Therefore, for a robust decoupling of both components, two ring buffers in a shared memory segment can be used. This avoids unnecessary context switches on bursts, and both components can notify each other asynchronously on new incoming or outgoing frames. Also, memory management in lwIP supports fine grained control of memory locations of the pbufs.

Additionally, the network chip could directly access the Ethernet frames in the shared memory via DMA. In case the hardware supports DMA on fragmented frames, the ring buffers could consist of pbufs of a smaller size than the maximum Ethernet frame size of 1518 bytes. As lwIP can handle incoming packets in arbitrary order, the assignment of frames becomes more complex.

Summarizing, this interface offers a good opportunity to decouple the software components and offers options for optimization when using DMA. On the other hand, memory usage can be higher.

C. Isolation between lwIP and sDDS

The next split is possible between lwIP and sDDS. For network connection, sDDS uses an internal interface of lwIP, which is similar to the BSD socket interface. However, incoming and outgoing datagrams are directly managed in pbufs and not copied into internal buffers.

The interface to the network side of sDDS opens a multicast and a unicast server connection for UDPv6 to handle all data exchange and service discovery. The job of lwIP is therefore limited to a pass-through of data and the resolution of IPv6 addresses to Ethernet MAC addresses. Like the split between Ethernet driver and lwIP, datagrams can be kept in ring buffers between the partitions; the internal memory management of lwIP would allow this, as long as only one client application (like sDDS) uses the IP stack.

Otherwise lwIP could asynchronously notify sDDS on datagram reception, while sDDS uses synchronous RPC for sending. This asymmetry is necessary, because sDDS (client) can trust the lwIP stack (server), but in reverse lwIP should not trust sDDS, because an error in a client could then affect the functionality of the whole IP stack. Compared to split (B) in Fig. 3 between Ethernet driver and lwIP, the implementation effort and runtime overhead of a split between lwIP and sDDS are higher.

D. Isolation between sDDS and Application

The third option is a split between sDDS and its applications. This isolates the applications from each other and from the rest of the system. Further, this allows the integration of applications of different origin without compromising safety and security of the rest of the system.

As described in Sec. II, the first application (publisher) uses a potentially blocking call to send its data, thus simplifying the design of sDDS when passing data down the stack to the network component, as calls to the network stack may block themselves. Because publishing of data may block, an isolating approach should also use a synchronous communication mechanism that supports blocking semantics.

The second application (subscriber) registers a callback, which sDDS notifies on incoming data. On notification, the subscriber calls sDDS synchronously and without blocking to remove the new sample from the topic's input queue. Depending on the depth of this queue, incoming data of a topic could be kept in a shared memory segment, so the application can read a new sample directly without any context switch. However, the application would then need to have write access to the input queue to mark the samples as read and acknowledge the reception.

A split between applications and sDDS therefore requires a mapping of the function calls of the DDS API to a synchronous communication mechanism and an asynchronous feedback channel to notify the application on new data. The amount of transferred data only depends on the size of the topics and is usually small, e.g. updates of sensor values.

E. Selected Isolation Approach

Comparing the isolation concepts presented in Sec. III (A) – (D) shows that typically each publish operation of the application leads to a frame for transmission in the Ethernet driver (this neglects the cases where sDDS publishes data to other applications on the same node or aggregates data for multiple topics into a single datagram) . However, incoming Ethernet frames do not necessarily contain data which finally reaches the application, because the frames could be used for discovery purposes in lwIP or for internal management of sDDS. Likewise, the size of exchanged data (respectively the required memory space) increases from application to network driver. On the other hand, the interfaces at the lower levels are more decoupled and allow the use of asynchronous notifications together with ring buffers.

With regard to the typically scarce availability of RAM on embedded microcontrollers, variant (D) with isolation between sDDS and the applications was selected (Fig. 4). This approach promises the greatest flexibility towards application development, as it splits base software (kernel, software stacks) from applications, and keeps the applications isolated from each other, allowing the integration of software applications of different vendors. But this comes at the price of implementing the more complex API of DDS. The selected approach combines Ethernet driver, lwIP, and sDDS into a single partition. The Ethernet driver uses the pbuf memory management of lwIP for its Ethernet frames. The Ethernet driver comprises an ISR task, which passes received frames via a message queue to lwIP. lwIP uses a single task that dispatches incoming messages from either the Ethernet or the sDDS side. sDDS consists of multiple tasks as Fig. 4 shows: one task that manages all incoming data and asynchronously notifies the application; and dedicated tasks handle each application's synchronous RPC calls.

Logistics & Sustainable Transport
Vol. 7, No. 1, October 2016, 41–50
doi: 10.1515/jlst-2016-0004

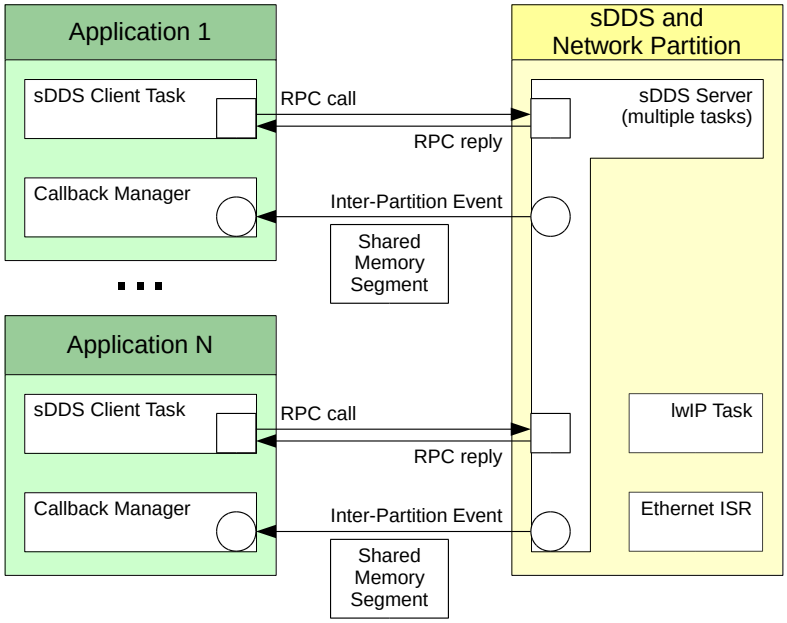

Figure 4: Communication between application partitions and the combined sDDS and network partition. Function calls of the DDS API are mapped to synchronous RPC calls. sDDS asynchronously signals the availability of new data samples to the applications.

The application partitions contain two tasks each: a callback manager task waiting for events from sDDS and running the registered callbacks in its context, and the client task executing the generated application code. A dedicated shared memory segment is used for data exchange between an application partition and sDDS. The shared memory segment contains the topics, the notifications, and the RPC call and reply arguments. A code generator sets up the structure of the shared memory segment, as well as necessary stubs for both client and server side (application and sDDS).

From both application and sDDS points of view the component split is completely encapsulated by generated code. This protects both sides from errors due to the potentially insecure communication via a shared memory segment. The code generator also knows the specific roles of the applications and creates the corresponding DataReader and DataWriter classes. These are used for the communication to other nodes.

IV. Evaluation

For evaluating the costs of the selected isolation approach (D), both memory usage and the time for transmission and reception of a topic's sample are compared to the monolithic approach without isolation (A).

In order to measure the performance overhead, trace points were added into the send and receive paths of all component interfaces. These trace points toggle GPIO pins of the microcontroller. Changes of the pins' voltage levels are recorded externally by a logic analyser. This kind of measurement adds additional costs for a system call into the kernel to toggle a pin (interference). However, as the microcontroller has no caches, this execution time shows a constant overhead of measured meantime 0.920 µs. The only sources of variance are timer interrupts and interrupts from the Ethernet controller.

For data transmission, the trace points mark the different phases starting at the application, via sDDS and lwIP, down to the Ethernet driver. Comparably, data reception includes trace points from the incoming interrupt of the arriving Ethernet frame up to the visibility of the data in the application. In splitting at the interface between sDDS and the applications, additional trace points were added to the kernel's event and RPC calls. The experiment consists of publishing data for a topic repeatedly, either from the microcontroller to a Linux host system, or the other way around.

Table 1 shows the results for the data reception path with minimum and maximum values (MIN, MAX), arithmetic mean (AVG), and standard deviation (STD) in microseconds for an experiment with 16500 recorded trace events.

Table 1: Measurement Data Reception (values in microseconds)

Phase	without isolation				with isolation			
	MIN	AVG	MAX	STD	MIN	AVG	MAX	STD
Ethernet IRQ in kernel	6.040	6.762	12.120	0.074	6.010	6.707	12.140	0.097
Ethernet ISR	8.240	8.276	18.380	0.428	8.240	8.291	18.380	0.573
lwIP stack	29.310	36.457	41.890	0.178	31.220	36.501	43.670	0.195
sDDS UDP module	17.490	17.505	23.380	0.125	17.410	17.426	20.320	0.073
sDDS DataSink_processFrame()	8.080	9.001	25.710	1.194	7.750	8.680	25.580	1.204
Data available callback	5.630	5.639	10.940	0.081	5.460	5.471	8.490	0.054
sDDS sends event to cb manager					1.140	1.147	3.070	0.023
Activation of callback manager					20.380	20.415	27.990	0.367
DataReader_take_next_sample; on isolation: RPC to sDDS					1.160	1.167	4.490	0.044
Reception of RPC in sDDS					8.240	8.252	13.750	0.107
sDDS sends RPC reply					3.390	3.396	6.960	0.048
Data available in application	3.340	3.352	8.820	0.075	1.160	1.167	4.490	0.044

Up to the point of the split, the data reception process shows similar values in the first phases for both approaches. Afterwards, the measured timing deviates: in the variant without split, the application can call `DataReader_take_next_sample()` directly from the callback, while the split variant sends an event to the callback manager first. The activation of the callback manager requires 20.415 µs on average, because the tasks of the application have a lower priority than the tasks in the networking partition and the network stack prepares internally for reception of the next frame. The callback manager calls the registered callback, which in turn calls `DataReader_take_next_sample()` to read the data. This is realized by a synchronous RPC back into sDDS and requires two additional context switches until the data arrives in the application.

Table 2 shows the steps for publishing data. Here, both variants already differentiate in the first steps: The variant without split can call `DataWriter_write()` directly, while the splitting approach sends an RPC to the sDDS partition. Afterwards, both variants show similar timing. However, the final step of the send phase, until control returns to the application, shows a difference again. Internal work of the lwIP state machine leads to visible delays, again caused by the different priorities of the tasks.

Table 2: Measurement Data Sending (values in microseconds)

Phase	without isolation				with isolation			
	MIN	AVG	MAX	STD	MIN	AVG	MAX	STD
Call to DataWriter_write; on isolation: RPC to sDDS					6.630	8.272	8.300	0.168
Reception of RPC in sDDS					1.180	1.390	18.070	1.093
Execution of DataWriter_write()	6.010	13.909	14.030	0.864	8.910	13.957	14.000	0.392
Generate SNPS packet	4.090	5.482	18.060	0.736	4.130	5.409	6.230	0.103
sDDS UDP module	7.130	7.145	7.390	0.020	7.130	7.142	7.390	0.020
lwIP stack	23.140	23.175	23.570	0.031	23.210	23.240	23.610	0.030
Ethernet driver	25.550	25.567	25.580	0.005	16.270	16.284	16.300	0.005
Back in application					13.170	13.178	13.190	0.004

For the whole project of all discussed software components including the kernel, RAM usage increases from 60,592 bytes to 65,792 bytes, i.e. by 8.6%. The costs are mainly caused by the stacks for the additional tasks on application and sDDS side. Likewise, the overall size of the program code increases from 83,944 bytes to 90,608 bytes by 7.9%. This is caused by the additional stubs, additional tasks, and two additional partitions.

V. Discussion and Related Work

The measured average execution time of data reception takes 87.0 µs in the monolithic case and 127.0 µs in the case including the selected split between sDDS and its applications. The execution times for sending are closer together: 75.3 µs for the variant without split and 88.9 µs with split. However, these values still include the overhead for the trace points of 0.920 µs each. The adjusted values after removing the constant overhead show a difference of 80 µs to 115 µs (44%) for data reception. Also, the time for sending data increases from 70 µs to 81 µs, i.e. an increase of 16%. The additional context switches between the tasks added for the split are the main cause of performance overhead here. These tasks also drive the costs for RAM usage due to their stacks.

Similar results were also observed in related work on decomposition of monolithic systems into multiple servers: SawMill [8] implements a file system as server on top of the L4 micro kernel [9]. For reading a 4K block of cached file data, the microkernel approach shows a 500 cycle overhead compared to a native Linux implementation with 3000 cycles [8]. The SawMill approach is comparable to the selected split on the application side.

Comparable approaches for splits on lower levels are more often found in hypervisors: XEN [10] virtualizes disk and network accesses at the level of disk blocks and Ethernet frames. This fits well for server virtualization, because here full operating systems are virtualized.

A comparable middleware approach like sDDS is offered by AUTOSAR with its *Runtime Environment* (RTE) [5]. The RTE layer comprises generated code and handles the data exchange between applications on different ECUs independently of the used network technology, e.g., CAN, FlexRay, or Ethernet. Optionally, AUTOSAR supports the spatial separation of applications from each other and the rest of the system, the *Basis Software* (BSW). However, AUTOSAR is currently restricted to applications in the automotive domain only.

VI. Conclusion

Summarizing, the results show that isolation concepts (and with it increased safety and security) has its price. The largest impact is on the performance side and results in performance losses due to additional context switches.

In this work, we presented an approach that splits a software architecture at the component interface between applications and the overall network system with the primary goal of isolating the applications from each other and from the rest of the system. A secondary goal is to keep extra RAM usage low. Still, the presented approach requires additional RAM for the stacks of the added tasks. Therefore, the implementation of isolation concepts is always a trade-off between RAM usage (for buffers and stack), ROM usage (shared code, generated stubs), and performance overhead by additional context switches instead of direct function calls.

In our opinion, the presented approach can help to increase safety and security of IoT devces by isolating application code, software stacks, and other software components from each other. From a safety point of view, this limits the impact of software errors to affected components. And from a security point of view, isolation prevents leakage of sensible data by unrelated applications. Also, the presented approach demonstrates a reasonable solution for cost sensitive markets like logistics in order to combine different applications on a single microcontroller.

In our future work, we plan to optimize the presented approach regarding memory usage and performance. Furthermore, we intend to implement and analyse potential splits at the other discussed boundaries with the final goal of decomposing a full AUTOSAR system.

References

1. Beckmann, K. & Dedi, O. (2015). sDDS: A portable data distribution service implementation for WSN and IoT platforms. In *12th International Workshop on Intelligent Solutions in Embedded Systems (WISES), 29-30 October 2015* (pp. 115-120). Ancona, Italy: IEEE.J. Clerk Maxwell, *A Treatise on Electricity and Magnetism*, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp. 68–73.
2. Object Management Group. (2015). *Data Distribution Service.* DDS Version 1.4. Retrieved June 6, 2016, from http://www.omg.org/spec/DDS/1.4/.
3. Various. (2016, June). *Lightweight TCP/IP Stack* [computer software]. Retrieved June 6, 2016, from http://savannah.nongnu.org/projects/lwip/.
4. Züpke, A., Bommert, M. & Lohmann, D. (2015). AUTOBEST: A United AUTOSAR-OS and ARINC 653 Kernel. In *21th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS), 13-16 April 2015* (pp. 133-144). Los Alamitos, CA: IEEE.
5. AUTOSAR Consortium. (2016). *AUTomotive Open System ARchitecture*. Version 4.1. Retrieved June 6, 2016, from http://www.autosar.org/.
6. International Organization for Standardization. (2011). *Road vehicles - Functional safety*. ISO 26262:2011.

7. Airlines Electronic Engineering Committee. (2010). *Avionics Application Software Standard Interface*. ARINC Specification 653.
8. Gefflaut, A., Jaeger, T., Park, Y., Liedtke, J., Elphinstone, K., Uhlig, V., Tidswell, J. E., Deller, L. & Reuther, L. (2000). The SawMill Multiserver Approach. In *9th Workshop on ACM SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, 17-20 September 2000* (pp. 109-114). New York, NY: ACM.
9. Liedtke J. (1993). Improving IPC by Kernel Design. In *14th ACM Symposium on Operating Systems Principles (SOSP), 5-8 December 1993* (pp. 175-188), New York, NY: ACM.
10. Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P. & Neugebauer, R.. (2003). Xen and the Art of Virtualization. In *19th ACM Symposium on Operating Systems Principles (SOSP), 19-22 October 2003* (pp. 164-177), New York, NY: ACM.

AUTHORS

A. Zuepke is with the Faculty of Design - Computer Science - Media, RheinMain University of Applied Sciences, Wiesbaden, Germany (e-mail: alexander.zuepke@hs-rm.de).

K. Beckmann is with the Distributed Systems Lab, Faculty of Design - Computer Science - Media, RheinMain University of Applied Sciences, Wiesbaden, Germany (e-mail: kai.beckmann@hs-rm.de).

A. Zoor is with the Distributed Systems Lab, Faculty of Design - Computer Science - Media, RheinMain University of Applied Sciences, Wiesbaden, Germany (e-mail: andreas.b.zoor@student.hs-rm.de).

R. Kroeger, PhD, is Professor at the Faculty of Design - Computer Science - Media, RheinMain University of Applied Sciences, Wiesbaden, Germany and leads the Distributed Systems Lab (e-mail: reinhold.kroeger@hs-rm.de).