Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Design and Implementation of a Container-based Architecture for Real-Time Control Applications

Jan Melcher

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dr. rer. nat. Asim Abdulkhaleq, M.Sc. Timur Tasci
Commenced:	September 29, 2017
Completed:	March 29, 2018

Abstract

The fourth industrial revolution and the advent of cyber-physical systems increase the flexibility and effectiveness in production, but they also change the role of software. Traditional monolithic systems need to split up in order to increase flexibility, maintainability and performance. There are existing approaches transforming traditional software towards a cloud-based infrastructure, but little work is done in applying this to real-time applications. This work proposes an architecture that uses containers to modularize real-time control applications, messaging for communication and a hardware abstraction layer to improve maintainability, reusability and flexibility. Using a prototypical implementation of the architecture, we validate the feasibility of this approach through a benchmark.

Kurzfassung

Die vierte industrielle Revolution und die aufkommende Verbreitung von cyber-physikalischen Systemen (CPS) erhöht die Fliexibilität und Effektivität von Produktionsanlagen, ändert jedoch auch die Rolle der Software. Traditionelle monolitische Systeme müssen aufgesplittet werden, um die Flexibilität, Wartbarkeit und Performanz zu erhöhen. Es gibt bereits Ansätze, traditionelle Software in eine Cloud-basierte Infrastruktur zu transformieren, aber bisher gibt es wenige Arbeiten darüber, wie dies auf Echtzeitanwendungen übertragen werden kann. Diese Arbeit stellt eine Architektur vor, die Container verwendet, um Echtzeit-Steueranwendungen zu modularisieren, und außerdem Messaging zur Kommunikation und eine Hardware-Abstraktions-Schicht einsetzt, um Wartbarkeit, Wiederverwendbarkeit und Flexibilität verbessert. Mit einer prototypischen Implementierung der Architektur wird der Ansatz mit einem Benchmark evaluiert.

Contents

List of Figures

1 Introduction

The field of industrial production is currently undergoing the fourth revolution, entering the era of *Industry 4.0*. Techonlogical advances in fields like artifical intelligence, robotics, and the Internet of Things enable new possibilities in the way goods are manufactured. The creation of cyber-physical systems plays a major role in this. Cyber-physical systems comprise of physical components coupled with software that coordinates, monitors and controls them [Raj+10]. In comparison to traditional automated systems, more complex software runs closer to the actual physical components and the interconnectivity between components increases.

As the importance and complexity of control software increases, so does the frequency it needs to be updated. Mass customization requires manufacturers to incorporate customer-specific changes into prodution processes [Gil+97]. Technological advancements yield newer and better algorithms, e.g. for CNC path planning. Monitoring and machine learning tools can predict that a process would be imporoved if a software component is changed. If every software update requires to stop a machine completely, they can impose a significant loss of revenue while the update is taking place. Thus, there is an incentive to localize downtimes to specific modules or to avoid them completely if possible.

Productions usually follow a hierarchical structure, illustrated by the automation pyramid (DIN EN 62264 [DIN14] or Siepmann [Rot16], see the left side of Figure 1.1). The bottom layer comprises of systems that directly communicate with hardware. The upper layers send commands to lower layers, which in turn report data up the hierarchy. Traditionally, this hierarchy is not only used to logically structure the components, but it also manifests itself in the physical deployment. Components of all levels are usually deployed on-premise in the manufacturer's IT infrastructure, and network infrastructure enforces data flow as defined by the pyramid.

With the advent of cloud computing and cyber-physical systems, this strict hierarchy is softened [BK13] (see the right part of Figure 1.1). Components can still be categorized into the levels of the pyramid (indicated by the colors), but they are deployed more flexibly, e.g. in a cloud. Interaction between the components is also loosened so that data can be transferred as required.

However, real-time components are usually left out in these approaches, and are still developed on proprietary and monolithic platforms. On the one hand, this allows to focus on the components that can be more easily transformed to a cloud-based architecture, and keeps the reliability of traditional control systems where safety is needed. On the other hand, it creates a gap between an increasing flexible and easy-to-deploy-to world of high-level applications and the traditional monoliths for real-time control software.

For applications running in the cloud, flexibility can be achieved by splitting monolithic applications into smaller services and running them independently. Using operating system containers, these services can be isolated without significant overhead. For non-real-time applications, this is state of the art and widely used. This work investigates how a container-based architecture can be used for real-time applications.

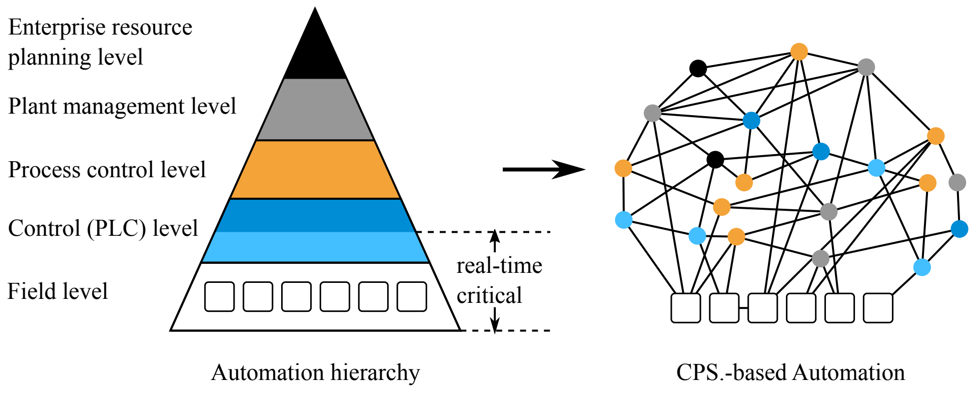


Figure 1.1: The automation pyramid and its decomposition with the adoption of cyber-physical systems [BK13] (translation by [LM13])

2 Related Work

For running industrial applications in the cloud, there are already several architectures. Gievhchi et al. give an overview over the existing architectures [GTJ13]. For example, Goldschmidt et al. propose a cloud-based architecture for PLC Software that offers multi-tenancy support and is horizontally scalable [Gol+15]. Within their own classification of real-time software into *soft*, *firm*, and *hard*, where only the latter considers a deadline miss a total system failure, they recognize that a cloud-based system can never support hard-realtime requirements. Benchmarks of a prototype of their architecture yield round-trip times below 1000 ms in 99.72% of cases for a system with 30 tenants on one instance.

Goldschmidt et al. evaluate the use of containers for industrial applications [GHS16]. Their results are promising in that they show a very low overhead of using containers in comparison to executing code natively. In their proposed architecture, they also adress the issue of running legacy software on different architectures using emulation.

In conclusion, there is existing work on running industrial software in the cloud with latencies of hundreds of milliseconds. There is however not much research done on using containers for real-time control software. Goldschmidt et al. show the feasability of this approach, but there are many research questions left open. This work focuses on how to use containers to split a real-time control application into modules, how these modules can communicate with each other and how they can interact with hardware.

3 Basics

3.1 Real-time Operating Systems

A real-time system needs to achieve reliable response times even in the worst case. In a control application, this means reading input values, computing a response and writing this into output values needs to complete within a specified time span. For some devices, it is safety relevant that this time never exceeded. Software that runs on bare metal – i.e., without an operating system – can reach this goal without complications if the computation time is smaller than the available time, e.g. a cycle time. If however additional software is run on the same CPU, it may delay execution of the critical real-time code, so that it may miss its deadline. Multi-processing operating systems incorporate interrupt management and a scheduler to control when which application can run. In a complex system with many applications and drivers, it can be hard to guarantee that none of them delays the critical real-time code so that it misses deadlines. Modern operating systems usually support task priorities, but internal routines in the operating system kernel can still block a user task from running for a non-deterministic time. Therefore, a real-time operating system is needed.

3.1.1 Approaches

There are different ways to enhance a modern operating system so that it can fulfill real-time requirements. In the following, the four most common ones are briefly explained.

RT-Enhanced Kernel

A standard operating system kernel can be enhanced to meet real-time requirements. First, it needs to be able to prioritize tasks that are marked as real-time tasks over other, non-real-time tasks. As soon as a real-time task is runnable, the scheduler should interrupt (preempt) other running task and switch to the real-time task. This feature is already available as a scheduling policy in many standard operating systems. For example, in Linux, `SCHED_FIFO`, `SCHED_RR`, and `SCHED_DEADLINE` all provide a deterministic scheduling policy with a higher priority than other tasks.

Secondly, real-time tasks also need to be able to interrupt the kernel itself. For Linux, a patch called `PREEMPT_RT` is available that allows all kernel tasks to be preempted. It is actively maintained and available for recent versions of Linux. Many of its changes are already merged *upstream*, i.e., into the standard Linux kernel, and there is ongoing effort to continue this practice.

The purpose of this patch is to minimize the time the kernel can not be preempted, that is, not be interrupted by tasks of higher priority. The fewer and shorter such non-preemptible phases occur, the better is the worst-case delay until a critical real-time task can be run. The patch does this by replacing many instances of spinlocks by preemptible synchronization mechanisms like semaphorse and by executing interrupt handlers within kernel threads [McK05].

Asymmetric Multiprocessing

A second approach is to dedicate a processor core to real-time tasks. They can run directly on bare metal without a kernel; then, no real-time code can be prevented from being run by kernel routines. Alternatively, a small real-time kernel can be used to provide basic operating system features such as multi tasking. In contrast to a rt-enhanced kernel this one can be simpler so that it is easier to verify that its maximum latencies are within bounds.

Co-Kernel

The co-kernel approach also employs a separate real-time kernel (called *co-kernel*), but it also runs the regular kernel. The co-kernel's scheduler decides when a real-time task should be run and when there is time to run the regular kernel. Similar to the previous approach, this co-kernel is relatively small compared to a full Linux kernel and thus validation is easier.

Separation Kernel

A separation kernel is located in a hypervisor. Both real-time and non-real-time code only see a virtual processor, and the separation kernel decides when each of them can be run. This separation is transparent to the guest systems. Thus, a regular operating system can be run on a separation kernel.

3.2 Messaging

3.2.1 Message Brokers

A message broker is a component that receives messages and routes them to their intended destinations [Hoh03]. In a system with a message broker, the individual components do not need to know each other. Instead, they only know how to connect to the broker and retrieve messages from and send outgoing messages to it.

Messages are usually categorized in *channels* or *topics*. Components only need to know which category of messages they are interested in and categorize their outgoing messages. The broker can use these categories to route the messages, but it can also be configured with advanced routing instructions or even transform messages in transit. As all messages pass through the broker, it can also implement management and monitoring functionality, such as logging messages of a certain kind or auditing the message flow against security guidelines. To cope with the high throughput and to provide high availability, brokers should be replicated with multiple instances.

However, brokers also introduce a significant overhead, both regarding network usage and latency [Zer17]. As the components can not communicate directly with each other, all messages need to be routed through the message broker. In a system where each message is routed to exactly one recipient component, this doubles the network usage because each message needs to pass twice through the network. It also doubles the latency introduced by the network alone. Also, the broker itself adds to the latency because of the processing time needed to determine the recipient component.

Brokers do not need to be centralized [Rab10]. A system can consist of multiple brokers that in turn communicate with each other. E-mail is an example for such a system: An SMTP server acts as a message broker; clients only need to contact their server to send an email. To deliver a message, the SMTP servers communicate with each other.

A system can also combine brokers with brokerless message flow [Rab10]. To a client, a broker is just one peer it can send messages to and receive messages from. To a message broker, it does not matter if one of its clients also communicates with other peers directly. Message brokers can also be used as a buffer between two components [Zer17] in case one of the components is not available and messages need to be queued until it is ready to receive messages again.

3.2.2 Messaging Frameworks

Advanced Message Queuing Protocol

Advanced Message Queuing Protocol (AMQP) is an open standard for a language-independent messaging protocol. It describes how messaging clients can communicate with message brokers. There are many message brokers that implement this protocol, e.g. RabbitMQ[1] or JBoss AMQ[2].

[1]https://www.rabbitmq.com/

[2]https://www.redhat.com/en/technologies/jboss-middleware/amq

ZeroMQ

ZeroMQ is a framework for developing brokerless messaging systems. It is written in C, but bindings for many other languages exists. It provides basic functions to create sockets and send messages consisting of byte sequences. There are multiple transport implementations such as TCP, UDP or IPC that all are accessible through the same API.

3.3 Scheduling

The scheduler is a component of an operating system kernel that determines which task is to be run at a specific time. It operates on a list of tasks that *can* be run (and are not, for example., waiting for another task or a system resource). Scheduling can be *preemptive* or *cooperative*; in the former case, the scheduler can interrupt already running tasks in order to prioritize other tasks, while in the latter case, tasks need to yield to the scheduler themselves when they want other tasks to be able to run. Waiting on I/O such as disk or network operations also implicitly yields control to the scheduler and thus to a different runnable process. Each task switch takes time because the state (mainly the registers) of one process needs to be saved and the state of the to-be-run task is to be restored. Cooperative scheduling avoids unnecessary task switches and therefore improves the throughput of a system. On the other hand, one misbehaving task can effectively freeze the whole system by performing continuous CPU operations and never yielding. This is especially bad for interactive systems where certain tasks need to react to user input or other triggers within a specific time. Therefore, depending on the nature of the system, an appropriate scheduler needs to be selected.

Real-time tasks can be classified as interactive tasks as their quality of operation is defined by the latency of their responses and not by the average throughput. If both real-time and non-real-time tasks run on the same system, non-real-time tasks need to be interrupted if a a deadline of a real-time task would be missed otherwise.

Linux offers three real-time scheduling policies [Schb]. The simplest one is `SCHED_FIFO` (first-in-first-out). It instructs the scheduler to choose the task that was enqueued last. Tasks can be assigned a priority between 0 and 99; tasks of higher priority are always preferred over those with lower priority and can preempt them if they become runnable. `SCHED_RR` (round-robin) is similar but restricts the maximum execution time of a task until it is preempted.

`SCHED_DEADLINE` is a policy for sporadic tasks, i.e. one with a sequence of jobs that are executed at most once per cycle [Schb]. It implements the global earliest deadline first algorithm with constant bandwidth server. Tasks with this policy are configured with a triplet of values: period, deadline and runtime. The policy guarantees that the task is scheduled periodically for a given runtime, before the configured deadline is exceeded. To ensure this, the kernel keeps track of the sum of runtime-period ratios and disallows further task creations if a predefined limit exceeded. Tasks may still be preempted during this runtime, e.g. because of a task with an earlier deadline, but can continue their execution afterwards with the remaining amount of their runtime. [Scha] When the runtime quota is exceeded, the task will not be scheduled until the next period begins. If a task blocks, e.g. because it waits on the result of an I/O operation, its runtime is still being reduced, so it can only execute a second time if it unblocks before the remaining runtime has reached zero. This way, the scheduler can meet the runtime guarantees for all running processes.

4 Architecture

In this chapter, an architecture for a modular real-time control software system will be presented. Meeting the real-time requirement is critical for this architecture, so we will start with a basic system that meets this requirement. Throughout this chapter, when new features of the architecture are proposed, the real-time aspect will be reconsidered to make sure it is met within the whole architecture.

4.1 Modularization

One way of developing a module is to implement its functionality for one specific machine and run this software directly on the machine without an operating system ("bare metal"). This would give the developer full control on how the software is run, as there would be no third-party code involved. As long as the module is implemented correctly, real-time properties are guaranteed to be satisfied.

However, using one physical machine per module does not scale well and puts highly modularized systems in a very disadvantageous position. Therefore, the architecture needs to be able to run multiple real-time modules on the same machine. Also, running the module software on bare metal would require the developers of a module to bring all utilities and drivers they need with them, as there would be no support by an operating system.

The base for this architecture will be a real-time operating system. Section 3.1 introduced multiple approaches. In the following, we will evaluate approaches to modularization in the context of the initial requirements. After having decided on a modularization approach, we will revisit the options for real-time operating systems and evaluate their fitness for the modularization approach.

4.1.1 Source Code Modularization

One way of running multiple modules on the same system is to combine their source codes and compile the resulting monolith into one binary. This allows to reevaluate realtime properties of the combined system by inspecting the source code, but this would be a time consuming and a difficult manual task. There are several disadvantages to this approach:

- The source code needs to be available, which is not always the case for third-party code.

- All modules need to be written in the same programming language
- Dependencies to libraries need to be found and version conflicts resolved
- To add or update a module, the whole monolith needs to be compiled and deployed again

An advantage is that modules can synchronously communicate with each other by calling functions. Regarding latency, this is the fastest and most predictable way of communication, so it would be a good fit for hard real-time constraints.

4.1.2 Static or Dynamic Linking

Instead of integrating at the source code level, each module could also be compiled into one library. The whole system then would consist of an executable that loads each module library and calls them all. In contrast to the previous approach, no source code would need to be available, and different programming languages could be used. However, not all kinds of languages could be combined as e.g. the calling convention needs to be the same.

Libraries can be linked either statically or dynamically. Statically linked libraries are added to the executable. To add or update such a library, the executable thus has to be relinked and redeployed. Using dynamic linking, libraries can be added or removed at runtime. That way, this approach enables it to update libraries in a running system without affecting existing modules.

Like the previous approach, modules can also communicate via function calls. However, when modules are updated in a running system using dynamic linking, still being able to call functions is not a trivial task.

4.1.3 Multiple Processes

The most prevalent method to run multiple services on an operating system is to run them in separate processes. If each module runs in its own process, they can be updated and restarted independently from each other. Modules are also more independent in how they are developed: They can be written in different programming languages and use their own versions of third-party libraries. For communication between modules, function calls are no longer an option. Instead, operating systems offer methods for inter-process communication. This can involve shared memory regions, pipes or sockets. It is also possible to communicate via network protocols through a loopback device.

Modules that are running as processes are mostly independent from each other, but they are still coupled to the host system. For example, a module targeted for Linux might assume that specific versions of system tools or binaries exist in certain locations, which might differ between distributions. Also, all modules share the system resources and might interfere with each other. Deploying a new module into a running system might jeopardize real-time guarantees that have been met before, just before it uses more system resources than anticipated.

4.1.4 Virtualization

To isolate modules even more while still running them on one machine, virtualization technology can be used to run multiple virtual operating systems at the same time. They are mostly independent from each other, so if one virtual machine is used per module, it can be configured exactly as needed for the module. Also, resources can be dedicated to specific virtual machines, so that their runtime behavior is isolated from other modules.

However, virtual machines impose a significant overhead on the system, as each module needs to bring its own operating system. Also, using a standard virtualization system makes it impossible to implement real-time software within virtual machines. Instead, a real-time hypervisor needs to be used which is aware of which machine needs to execute real-time tasks.

4.1.5 Containerization

Containers are a lightweight alternative to virtual machines. They allow programs to run in their own private space without the need to run the operating system multiple times. Usually, only the target program is run inside a container, and no other services like logging, remote access, or user management. They are either provided by the container host system or not needed to just one application. That way, running a container needs significantly less memory and CPU than a full operating system.

For the host system, a process or thread running in a container is not conceptually different from a task that runs directly on the host system. Scheduling options like real-time properties do not get lost, and so the kernel can treat them correctly.

Docker provides many options to configure a container. If containers need to communicate, they can be assigned a shared namespace for inter-process communication. Tooling exists to easily create and assign networks. By setting capabilities, modules can be allowed to set their scheduling options. With resource limits, CPU and memory quotas can be set.

4.1.6 Conslusion

Source code modularization and static linking do not fulfill the requirement to add or update modules at run-time, and even using dynamic linking, this is not easy to accomplish. Also, for all of them, the coupling between the modules is too high. Therefore, these options are no longer considered.

Virtualization offers good isolation at the cost of a high overhead. The architecture targets control devices which usually have limited power. If only few modules could actually run simultaneously, designing a distributed system would be difficult to accomplish. Probably, one would look for alternative ways to do a more lightweight modularization within this system which defeats the purpose of this architecture. Additionally, the overhead of using a real-time hypervisor in conjunction with the real-time scheduler of the guest system would impose an overhead in latency.

Using processes or containers is very similar. Containers have a low overhead compared to regular processes. The benefits it offers like resource limits help to achieve the requiements of this architecture. Therefore, this architecture will use containerization as modularization technique.

Table 4.1: Comparison of modularization options

	Static Linking	Dynamic Linking	Processes	Containers	Virtual Machines
Language-independent			✓	✓	✓
Independent permissions			✓	✓	✓
Independently deployable		✓	✓	✓	✓
Independent system utilities				✓	✓
Architecture-independent					✓
Small memory overhead	✓	✓	✓	✓	

4.2 Real-time Operating Systems

Section 3.1.1 outlines four basic approaches on how a real-time operating system can be designed. In this section, they will be evaluated in the context of Docker.

4.2.1 RT-Enhanced Kernel

A rt-enhanced kernel is a regular kernel augmented with features to support running real-time tasks. This capability is orthogonal to supporting containers, so if the base kernel supports containers, the rt-enhanced kernel is likely to support containers that can run real-time tasks.

Linux supports containers, and the PREEMPT_RT patch enhances Linux for real-time use. The main changes introduced in the patch affect locking and hardware interrupts. Containerization features like process namespacing are sufficiently encapsulated so that the patch does not break them.

Standard linux already supports real-time scheduling policies and supports limiting real-time execution time via cgroups (`cpu.rt_runtime.us` with `CONFIG_RT_ROUP_SCHED`), which is used by Docker. This allows containers to rely on a certain CPU bandwidth which they can in turn use to guarantee their real-time properties.

4.2.2 Co-Kernel

A co-kernel is a second, specialized, kernel – in our case, a real-time kernel. It runs real-time tasks as well as the main kernel which in turn runs regular tasks. The co-kernel needs to be able to schedule tasks independently from the main kernel because otherwise, it would need to synchronize with it and thus lose its real-time property. Therefore, it needs to maintain a separate task structure and run queue. Unless specially being accounted for, namespacing and process isolation features that make up the foundation of containers are therefore not respected in the co-kernel. However, the feature set of the co-kernel is reduced compared to the main kernel. For example, file system access is not handled by a co-kernel. Therefore, file system namespacing does not need to implemented by the co-kernel anyway.

Xenomai's co-kernel, Cobalt, is integrated into the Linux kernel. Task to be scheduled by Cobalt can be created by regular Linux tasks, and they are known to the Linux kernel, too. To access non-realtime resources like the file system, they are run as a regular Linux task and can make system calls to the Linux kernel. For these actions, the Linux kernel takes care of namespace restrictions.

However, Cobalt itself does not support containers. This is most apparent in that it does not resolve namespaced process ids correctly. This bug can however be fixed with few changes to the Cobalt source code. Then, Cobalt can be used from within Docker containers. To do this, a special device file that is used to communicate with the Cobalt kernel needs to be mapped into the Docker container. Also, Cobalt needs to be configured to grant access to the kernel not only to the root user, but also to users of a certain group, and this group has to be configured for the container.

Resource limits like the CPU quota set on the container via cgroup is are not respected by Cobalt. Also, the Xenomai developers explicitly state that once granted access to the Cobalt APIs, a process should be considered to have unlimited access over the system. The APIs are not security-reviewed and allow direct access to the hardware, anyway.

In conclusion, a co-kernel approach using Cobalt can be integrated into a container architecture but, with the current state of the art, does not provide security or isolation.

4.2.3 Asymmetric Multiprocessing

In the asymmetric multiprocessing approach, some CPU cores are dedicated to real-time tasks while other execute non-real-time tasks. If multiple real-time tasks are to be run on one CPU core, a special real-time scheduler is needed. In our architecture, this would likely be the case, because otherwise, the number of modules would be limited to the number of CPU cores.

The compatibility of this approach with containers depends on the design of the scheduler for real-time tasks. If it maintains task structure totally separate from the main kernel, containerization features implemented in the main kernel do not apply to real-time tasks. If both kernels are interconnected and allow a task to switch between real-time and non-real-time mode, this approach is comparable to the co-kernel approach.

4.2.4 Separation Kernel

A separation kernel resides in the hypervisor of a virtualized system. This is on a different level than containers and the separation kernel does not see the containers at all. Thus, this approach is not compatible with containerization.

4.2.5 Conclusion

A rt-enhanced kernel is the approach most compatible with containerization. The architecture will therefore be based on a Linux kernel with the PREEEMPT_RT patch.

However, for safety-critical features, the predictability of the co-kernel approach might be needed. In this case, both approaches can be combined, using Xenomai and Cobalt in combination with a Linux kernel with the PREEEMPT_RT patch. Cobalt is linked into the kernel via a set of patches. These patches need to be merged with the patches of PREEEMPT_RT. This results in some file conflicts. Once they are resolved, a functional kernel can be compiled which passes basic tests regarding real-time properties and Cobalt APIs. Further evaluation is needed to determine if this works in all edge cases.

4.3 Messaging

The advantages of the container architecture are most apparent when a control system is split into multiple modules. These modules likely need a way to synchronize and send data to each other. In a conventional, monolithic application, modules can communicate using simple function calls and shared memory. When modules run in different containers, they can no longer call each other, and they can not necessarily access shared memory. Instead, the modules can communicate over network protocol.s In case two modules run on the same host system, inter-process communication techniques are also an option

Modules are most reusable if they do not make assumptions as to whether they run on the same host or on different hosts. Therefore, it is advisable to not base their design on the availability of shared memory to communicate. If it is, this may be an efficient way to exchange data. However, the modules should still work when being distributed across different hosts, then exchanging the data using a networking protocol. This suggest the usage of the messaging design pattern. It provides an abstraction on the physical way of transport and notification. Messages can be exchanged using various protocols, and the applications that are sending and receiving the messages do not need to know the implementation details.

Another advantages of messaging is that it generally leads to more loosely-coupled systems. If the format of messages are formally specified, one module can be exchanged for a different one without the need to change connected modules, as long as the message format and semantics stay the same.

This enables several use cases, for example:

- The behavior of a system can be modified without changing existing modules. For example, to debounce input values, a simple module can be plugged between the input device and the modules that access it.
- The algorithm of one aspect can be swapped by replacing a module with a compatible one. In the controller of a CND machine, the interpolation code could be swapped this way.
- Cross-concern aspects like logging or monitoring can be added to an existing system by connecting the modules that generate these values to logging and monitoring modules. Existing connections that are required for the function of the system do not need to be changed.

4.3.1 Real-time messaging

Messaging can be used to decouple components regarding time. In many situations, it is not important that a messages is processed in the instant it arrives. In case the recipient is currently not available, busy with other tasks, or the network connection is down, an incoming message can be queued and processed later. Such a system is considered robust because the functional correctness is not affected by load peaks.

However, in a real-time context, there is a hard limit on the acceptable time it takes to process a message. If a message could not be delivered in a certain time, there is no point in reliably storing it. This suggests there are special aspects to consider when designing a real-time messaging system.

- Messages of unbound lengths are not allowed in a real-time system because it would not be possible to calculate a worst-case transmission or processing time.
- There can be message queues, but they also need to be limited in length. This is relevant when messages can be processed faster than they are created, and a certain processing delay is acceptable. The system has to be configured so that the message queue does not grow beyond a certain length.

- The timing characteristics of message transport affect the real-time property of the whole system. Thus, in order to evaluate the real-time behavior of a system, the transport methods of all message channels need to be selected first.
- The ability of a module to meet its deadline requirements depends on the responsiveness of other modules. The runtime behavior thus becomes part of the contract of a message channel, much like the message format.

In summary, although messaging allows to functionally decouple modules from each other and form the technical communication implementations, a system still needs to be evaluated as a whole in order to determine its real-time behavior.

Obviously, the whole code that implements the messaging framework needs to be real-time capable. This means its runtime may not exceed a specific limit, even if other, non-real-time, tasks are running on the same system. For example, it can not use a standard network stack that is shared with non-real-time applications. If kernel features are used, they need to be aware of the priority of the calling process.

However, the restrictions also grant an assumption to be made by the messaging framework: As both message length and queue size is limited for a given message channel, the whole queue size is limited. Thus, a ring buffer of fixed length can be used for the queue. This property will be used in one implementation that is described later.

4.3.2 Message Brokers

As described in Section 3.2.1, message brokers are components that receive messages and route them to their destination. Employing a message broker in an architecture has many benefits as it brings features that are likely to be required, from service discovery over management to monitoring. However, its main disadvantage – the introduced latency and overhead – is to be considered, especially for a real-time application.

If a control application with a cycle time below one millisecond is to be split up into several modules, a low-latency way to communicate becomes crucial. One cycle can involve several modules that each need to receive a message, process it and pass the result to the next module. Depending on the module granularity, the actual work done by a module can be as small as comparing a value to a threshold, doing a conversion or an interpolation. In this scenario, the extra overhead of passing each message *twice* – once to a broker, and once from the broker to the next module – is significant. Therefore, the disadvantage of the extra overhead introduced by a broker is emphasized in our architecture.

If a broker is used for communication between real-time modules, the broker itself needs to be real-time capable, too. First, this means that all procedures involved in receiving, routing and sending messages, need to be time bounded. Even algorithms of logarithmic time complexity regarding an unbounded variable might violate this property. All threads running these procedures need to use real-time scheduling policies. Secondly, if real-time and non-real-time messages are processed by the same broker, it has to support priorities on messages.

In a real-time system, having message queues that are available even if modules are unavailable, is generally less important than in many other systems. Messages describing the current state of the system are obsolete rather quickly and replaced by the next one. If a module is not available at a given time, queuing the message may not be of any use because by the time the module is restarted, the deadline for the response might have already elapsed. In a safety-critical control application, the correct way to handle a faulty situation is usually to switch to a safe state, e.g. by switching off power of actuators. This is not to say that there is no use for message queues in control applications, but there are many situations where this is not needed.

Similarly, persistent message storage is less important than in other contexts. The guarantee of a messaging broker to store messages persistently allows to develop transactionally safe components. A component might for example receive a message, process it by performing a modification in a data base, and then acknowledge the successful processing of the message. If the system crashes before the data modification is acknowledged and persisted in the data base, the message is still held persistently in the message broker, and after a restart, can be processed again. This only works if the message broker guarantees to store messages persistently.

Recovering from a crash of a control system is not this simple. With cycle times in the range of milliseconds, recovering from a crash takes longer than a cycle and thus deadlines are missed either way. As physical state of the machine might have changed in between, it is generally safer to recover from a crash with a defined protocol instead of continuing to process old messages as if nothing happened. Besides, any state in the modules would need to be persisted in each cycle in real time.

On the other hand, some features of message brokers are required by this architecture. Even though modules should be able to communicate directly with each other, the configuration of which modules communicates with which should not be hard-coded in the modules themselves. Ideally, it should even be possible ot update this configuration in a running system. In this case, a message broker can be used to distribute the configuration to the modules. However, this approach is limited because it might be necessary to reconfigure the containers if they need to be able to communicate to a new module. In Docker, some reconfigurations can only be done by removing and recreating a container.

4.3.3 Messaging Framework

In a messaging system without a broker, the whole messaging functionality needs to be implemented in the communicating modules themselves. In Section 3.2.2, the brokerless messaging framework ZeroMQ has been introduced. It supports a variety of transport mechanisms from in-process to network protocols and offers a clean uniform API. The developers of ZeroMQ performed latency tests on a Linux system with a rt-enhanced kernel [Sus08] and measured maximum latencies below 100 μs. Therefore, this library qualifies in principle for usage in real-time systems, although more specific tests obviously need to be performed.

ZeroMQ provides the generic functions `zmq_bind` and `zmq_connect` which accept a string that specifies the transport protocol and protocol-specific address information. After this configuration, the functions to send and receive messages work independent from the protocol. This is important because the transport protocol might not be known at compile time. The API of ZeroMQ allows to write the code transport-independent and pass the correct connection string at runtime, e.g. via environment variables, config files or arguments.

4.3.4 Messaging Patterns

The simplest form of messaging uses unidirectional channels which transport arbitrary messages from one module to another. Most use cases can be implemented using this simple design.

Request-reply

A synchronous request-reply pattern (where a client module sends a message to a server module and then waits for the reply) can be implemented with two channels, one from the client module to the server module for the requests and one opposite for the replies. As there is at most one request under way, it is clear to which request a reply relates. If the client module does not synchronously wait for the reply but continues executing, it might happen that it sends a second request while the first one has not been answered yet. As long as the server module processes incoming requests in order, replies can still be related properly on the client side. If, however, processing of requests can be parellelized and replies are sent in any order, there needs to be an explicit mechanism to correlate replies to requests. In this case, the request message should include a request identifier. For convenience for the developers and to reduce the chance of flawed implementations, this functionality can be provided by a library.

ZeroMQ has special socket types for the synchronous request-reply pattern. The user is requred to send and receive messages alternately. Both requests and replies are sent through one socket. This simplifies set-up for this use case but it requires that sockets can be bidirectional. In contrast to two unidirectional channels, this reduces the flexibility as both requests and replies now need to follow the same transport path. There might be transport protocols that only support one direction (e.g. UDP multicast) or it might be appropiate to route requests or replies, respectively, through an additional module. In any case, the synchronous request-reply pattern is simple enough with unidirectional channels so that the complexity of bidirectional channels will be avoided in the architecture.

Publish-subscribe

Publish-subscribe is a messaging pattern where one participant sends (*publishes*) messages and others can *subscribe* to them. The sender usually does not know who will receive the messages or how many recipients there are. This is useful to inform others about an event if no direct response is required.

Some of the requirements for this architecture require the availability of the publish-subscribe pattern. For example, modules that provide sensor data should make them available for any interested module. This data providing module does not require any feedback from the recipients. Thus, it is a perfect fit for a publisher. Monitoring modules also make use of this pattern as they are linked to certain output parameters in order to monitor their value. The modules that output the values do not care if there is a monitor attached or if they are just sending them through the normal path to the next processing module. Here, the monitoring modules are subscribers.

Messaging architectures with brokers easily support this pattern because the broker already provides an abstraction to who sends and receives messages; publish-subscribe is just one way to route messages. Partial subscriptions, e.g. based on hierarchical topics, can be implemented in the broker. Also, it is possible for multiple publishers to send message into the same topic, which then get merged and delivered to all receivers of this topic.

In a brokerless environment, where messages only flow between receivers and publishers, more configuration effort is needed for the publish-subscribe pattern. In the simplest case, the publishers know all recipients and send the messages directly to them. This can be abstracted in a framework so that the application code only needs to call the API for sending the message once. The list of recipients then can be configured at deploy time or at runtime.

If connection-based message protocols are used (e.g. TCP or unix sockets), it is also possible that the subscribers open connections to the publisher. The publisher then sends outgoing messages through all open connections and does not otherwise need to know the subscribers.

There are also scenarios where neither recipients nor publishers directly know each other but are connected through a common handle like a topic. In this case, a common component needs to find out who is subscribed to a topic and who publishes messages to it. Then, it can for example instruct subscribers to connect to all the publishers or it can instruct publishers to connect to all the subscribers. These might even be combined, i.e. a publisher sends its messages to some recipients, but other recipients initiate the connection to the publisher. Which is best depends on the transport protocol.

In conclusion, publish-subscribe is a pattern required for our architecture. Without a broker, support by the framework is needed in order to set up the message flow correctly. Therefore, the architecture needs to include way to configure a message flow and inject the resulting connection configuration into the modules.

Proxy

As noted above, any change in the system can affect the real-time properties. Message flow is no exception. In particular, if a core real-time module needs to send outgoing messages to many other modules, potentially over the Internet, this can impact its performance.

To reduce the impact on the real-time module, the proxy pattern can be used. This is a simple module which only passes messages through, from potentially many incoming connections to potentially many outgoing connections. The original module only needs to send its messages to the proxy module. If the proxy module is located on the same machine as the real-time module, an efficient inter-process transport can be used. The proxy module then takes care of sending the message to a larger number of recipients. If this transmission takes longer than anticipated, the real-time properties of the original module are not impacted.

4.3.5 Ring Buffer Transport

If a monolithic application is split into several modules, it is likely that they still need to exchange a lot of data. To reduce the overhead of the containerized architecture to such software, we propose a transport implementation based on ring buffers. This transport assumes the availability of shared memory between both connected modules. This can be achieved by assigning both containers to the same IPC namespace.

The algorithm is based on an open source implementation by Frederick M. Proctor [Pro99]. It uses a queue data structure which resides in memory shared between the receiver and the sender. It consists of a fixed-length array of messages, a read position and a write position. A message is an arbitrary fixed-length data structure, e.g. a byte array and a length information.

Upon initialization, read and write position are set to equal values. To send a value, the write position is determined by incrementing the current write position, and wrapping it around to zero if it exceeded the queue length. the message is copied into the queue at the correct position and then, the write position is updated in the data structure. The receiver waits until the write position changes and then reads the next message off the queue. It also updates its read position so that the sender can check if it would override messages that have not yet been read.

Instead synchronizing using write and read positions, semaphores can also be used. A write semaphore is initialized with zero; a read semaphore with the queue length. the write semaphore is incremented each time a message is sent and decremented when it has been read. for the read semaphore, it is the opposite. That way, the operating system takes care of synchronizing both processes.

Limitations

Using Docker, only one IPC namespace can be assigned to each container. This is either a private namespace, the host namespace, or it is shared with other containers. If many containers should communicate via shared memory, they all need to share one IPC namespace. Any of these modules can then potentially intercept the communication of any other module in the same namespace. In contrast to networking, the channels can not be locked down further using routing rules or a firewall.

The suggested solution for this case is to identify sets of modules of the same trust level and assign a IPC namespace to each of these sets. The communication between the namespaces then needs to use a different transport, e.g. named pipes or a network protocol.

4.3.6 Conclusion

We use the messaging pattern for communication because it decouples the modules in several ways. Once the message formats are defined, modules can be developed independently from each other can even switched for different implementations without impacting the interoperability. The technical transportation of messages can be abstracted with a simple API so that for each communication channel, the best transport can be chosen without having to change the module implementation. It is necessary to support more than one transport depending on latency requirements, physical and logical location of modules and available hardware resources.

A brokerless solution is preferred because it minimizes latency, which is especially important in loops with small cycle period lengths. The publish-subscribe messaging pattern should be supported explicitly because it is needed to implement the requirements and it can not easily be implemented without framework support. Other patterns like request-reply do not need special support because it can be implemented within the modules.me

4.4 Drivers

Control applications usually need to interact with specialized hardware like sensors or motors. To improve the portability of an application, it should not be assumed that the required drivers for the hardware it accesses, is installed on the target environment. Instead, it would be beneficial to leverage the advantages gained by containerization for drivers, too. This can be achieved with user-mode drivers [Nak02]. There is still some support needed in the kernel, but this support can be generic enough to support various use cases, while the actual drivers are implemented within a Docker container running in user space. These containers need special permissions to access memory mapped directly to hardware, so it is advisable to keep them small. The driver modules then can communicate with other modules via messaging.

It is also possible to combine regular kernel-space drivers with Docker containers: If a device is connected to the host via a generic bus interface, it can be advisable to use a regular driver for the bus, and a container that uses this bus driver to communicate with the remote device. The bus driver would then be mapped as device file within the container.

4.5 Scheduling

The architecture involves running many modules, with mixed real-time and non-real-time tasks, on the same system. The system's scheduler and the task's scheduling policies need to configured properly to ensure all real-time guarantees are met. Whether this is the case can be answered by induction (looking at modules individually and deducing the overall performance) or by tests of the whole system.

Looked at in isolation, a module fulfills it requirements if it performs correctly, i.e., it delivers correct responses in time. To prove this, one can assume that other modules and the operating system (most importantly, the scheduler) behave as specified. Then, the whole system is correct if all its modules are implemented correctly, their assumptions do not contradict, and the system is able to meet the total requirements.

As an example, imagine a system consisting of three modules that each require 400 μs of CPU time each millisecond, and that needs to read inputs at the beginning of each cycle of one millisecond and respond before the end of a cycle. On a dual-core processor, the system is able to guarantee the required execution time for each module with some fraction to spare for non-real-time tasks. If module three depends on the results of both modules one and two, the scheduler is able to execute the first two modules in parallel, and then the third one after both of them have finished. If however the second module depends on a result of the first module, and the third module depends on a result of the first module, then this configuration can not be scheduled in a way that still fulfills the response time of one millisecond. This shows that adding up the CPU fractions of all individual modules is not sufficient, and the interaction between modules needs to be taken into account. In the general case, the interaction can be complex and may even be different for each cycle. Deciding if the system can be scheduled properly, just by looking at the source code of each module is undecidable, because it is a non-trivial property according to Rice's theorem [Ric53]. Therefore, it is necessary to specify the module's behavior in a simpler way. One possibility is to use *execution profiles* that describe how a module reacts to incoming messages, how and when it sends messages itself and when it requires what system resources like the CPU.

Examples of execution profiles are:

1. A *periodic source* that is activated periodically, performs a computation takes up to t CPU ticks (e.g. to read a value from a hardware port), and then sends an outgoing message

2. A *consumer* that waits for an incoming message and then performs a computation that takes up to *t* CPU ticks (e.g. to write a value to a hardware port).
3. A *pipeline module* that waits for an incoming message, then performs a computation that takes up to *t* CPU ticks, and then responds with an outgoing message.

Real-world modules can be more complicated than this, and listing all possible execution profiles would not be possible. The point is to bring the module's source code to an abstraction level where only external events such as incoming and outgoing messages are represented, and everything in between is summarized in something like *t CPU ticks*. An UML interaction diagram could be used to visualize an execution profile, but it would be advisable to use a format that is specifically tailored for this use case. One possibility is the following definition:

1. A *module* consists of several *tasks*.
2. A task consists of a sequence of *phases*, that is repeated infinitely.
3. A phase consists by a *blocking state* where the module waits for an external event and a *running state* with a solid block of execution on the CPU.
4. A phase defines a trigger that ends the blocking state and starts the running state, e.g. a timer or an incoming message.
5. A phase also defines the worst-case CPU time for the running state, and which messages will have been sent after it executed.
6. A phase may define a deadline for its completion.

This definition can be used to describe all of the examples above and more, but it can not describe all possible cases. To determine if a system consisting of several modules can be scheduled, all execution profiles need to be created. Then, a dependency diagram for phases needs to be constructed. Each phase has a dependency on the previous phase, and for each message it waits for a dependency to the phase of a module that sends this message is added. In order to reflect the latency of message passing, an intermediate node needs to be inserted with the worst-case latency as its time. Once the dependency diagram is created, a topological sorting algorithm can be used to determine an optimal scheduling plan. If all phases that define deadlines are completed before their deadline, the system can be scheduled correctly.

This approach assumes the scheduler always chooses the optimal schedule. None of the policies explained in Section 3.3 are able to do this because they miss cricical information about the dependency between modules or about deadlines. *SCHED_DEADLINE* is the closest one because it respects deadlines, but it is not suitable for tasks that wait multiple times, e.g., once for a timer and then for an incoming message. Also, multi-core systems are not properly represented by a topological sort but need a more advanced algorithm. Another approach is to use the execution profiles and calculating all possible outcomes with a specific scheduler, such as *SCHED_RR*, and then determining if the worst case of them respects all deadlines.

If all modules are already implemented, it is also possible to test the behavior by running the complete system.

4.6 Complete Architecture

Figure 4.1 illustrates the architecture from a functional view. An application consists of several *modules* (*M1*, *M2*, and *M3* in the example). A module is a piece of executable software (in the form of a container image), combined with its metadata. Included in the metadata are *input* and *output* definitions (illustrated by small squares) with type information regarding messages being sent or received. *Links* connect inputs and outputs of compatible types. Multiple links can be connected to a single input or output, to support messaging patterns described in Section 4.3.4. A link also specifies which *transport* (e.g. UDP or IPC) is to be used. This information is not required from a purely functional point of view, but necessary to fully specify an application for performance tests.

The *common hardware interface* acts as a meta-module that provides generic inputs and outputs for various types of hardware. These hardware types are not specific products but abstractions with a generic interface. Modules can be written against these interfaces to improve reusability.

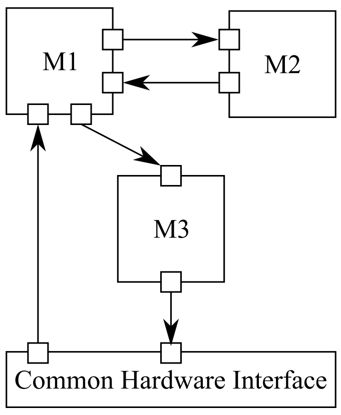


Figure 4.1: Functional view of the architecture

Applications following this architecture can be deployed on *hosts*. Such hosts are Linux machines with some additional preparations (see Figure 4.2). In full configuration, the kernel has both the *PREEMPT_RT* patch and the Cobalt kernel patch applied. This allows to run different modules in both kernels at the same time, depending on their criticalness. The Cobalt kernel needs to be patched in order to support PID namespaces. It is also possible to only apply one of these two patches.

Drivers can be installed both in the Linux kernel and in Cobalt, to be used by modules running on the corresponding kernel. All modules run in Docker containers, managed by the *Docker daemon*. They interact with drivers via device files mapped within the container.

Communication between modules takes place according to the links and configured transport methods. The *corteX runtime* (*cortex* standing for *container orchestration for real-time environment X*) is responsible for creating and updating Docker containers and supporting infrastructure such as volumes and networks. It informs the containers about connected links and makes sure they can connect using the specified transport methods, e.g. by assigning related modules to a shared network or IPC namespace.

The common hardware interface is translated to a set of modules that implement the required device drivers. Upon deployment, the user can choose the specific drivers according to the used hardware.

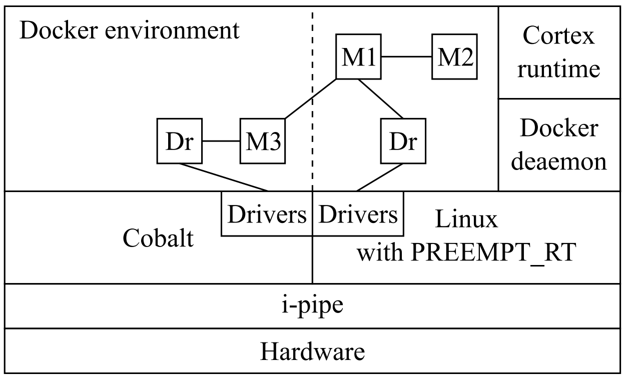


Figure 4.2: System view of the architecture

5 Evaluation

5.1 Prototypical Implementation

To evaluate the proposed architecture, we developed a prototypical implementation of the core aspects. These include running multiple real-time Docker containers on one host, using messages to communicate, and accessing native drivers through containers. The prototype consists of several modules, a common framework for these models (*libcortex*), and a command-line interface tool to manage orchestrations.

For consistency in performance and to simplify interacting with the kernel, modules are written in C. All modules need to implement messaging functionality with various transport methods. Therefore, this has been extracted into the library *libcortex*. It offers a very simple messaging API: *cortex_input* and *cortex_output* to open channels, and *cortex_send* and *cortex_receive* to send and receive byte sequences. The library parses environment variables to determine which transport to use for each input and output. Most transports (*tcp*, *udp*, *ipc* and *inproc*) are handled via the library ZeroMQ. In addition, the *shm* transport implements the Ring Buffer Transport in shared memory as described in Section 4.3.5. There is also a variant *shm+sem* that uses semaphores to communicate the availability of new messages.

A command-line interface written in Python servers as a simplified *cortex runtime*. It reads an application description from a YAML file and creates and configures the necessary Docker containers, networks and volumes. The application description includes modules and their configuration (environment variables, capabilities, and device files) as well as links. For each link, the source module and output, the target module and input, and the transport protocol can be specified. All modules share an IPC namespace, a network, and a volume to share IPC files. This way, multiple applications can be run simultaneously without effecting each other, and still allow free communication between all the modules.

5.2 Benchmarks

In order to evaluate the real-time behavior of a containerized system communicating via messages, we set up a benchmark. This benchmark is based on the prototypical implementation introduced in the last section. It consists of two modules: The benchmark module and a relay module. The benchmark module runs in a loop and executes multiple *runs* that each generate one set of aggregated statistics. A run is executed within a thread with real-time priority. It executes a configured number of cycles with a configured interval time. The sequence of one run is illustrated in Figure 5.1. Each cycle sends a message through its output channel and then waits for a reply on the input channel. The time it spends sending and receiving (including the waiting time for the reply) is recorded as *runtime*. *Latency* on the other hand is the time difference between the scheduled wake-up time and the actual wake-up time between two cycles.

Due to the short period lengths, a benchmark usually executes a large number of cycles (more than a million in ten minutes at a period of 500 μs). To simplify working with the results, the tool aggregates the metrics *runtime* and *latency* on the benchmark level. Via the number of benchmarks and the number of cycles per benchmark, the resolution of the output data can be configured. The aggregations include minimum, maximum, mean, median, percentiles (99% to 99.999%), standard deviation and the mean absolute percentage error.

5.3 Results

We ran the benchmark tool described above on a machine with Linux 4.9.53, the *PREEMPT_RT* patch and the cobalt kernel patch. We used three different transport methods (UDP, IPC and the ring buffer as described in Section 4.3.5. Each benchmark consists of 500 runs of 121000 cycles with a period of 500 μs each. During the test, the stress generator *stress-ng*[1] was running with the options *–cpu 4 –io 4* to use the CPU and keep the kernel busy with *sync* calls.

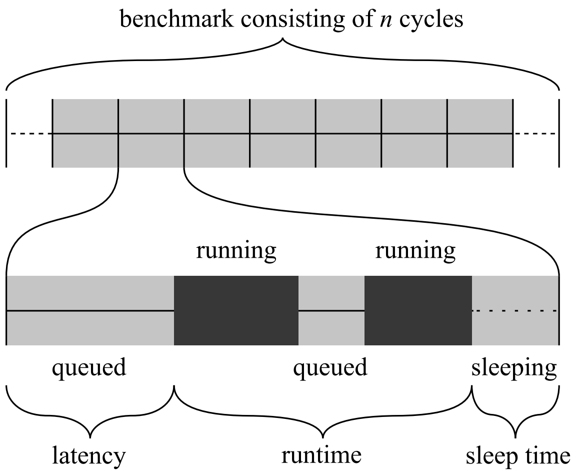
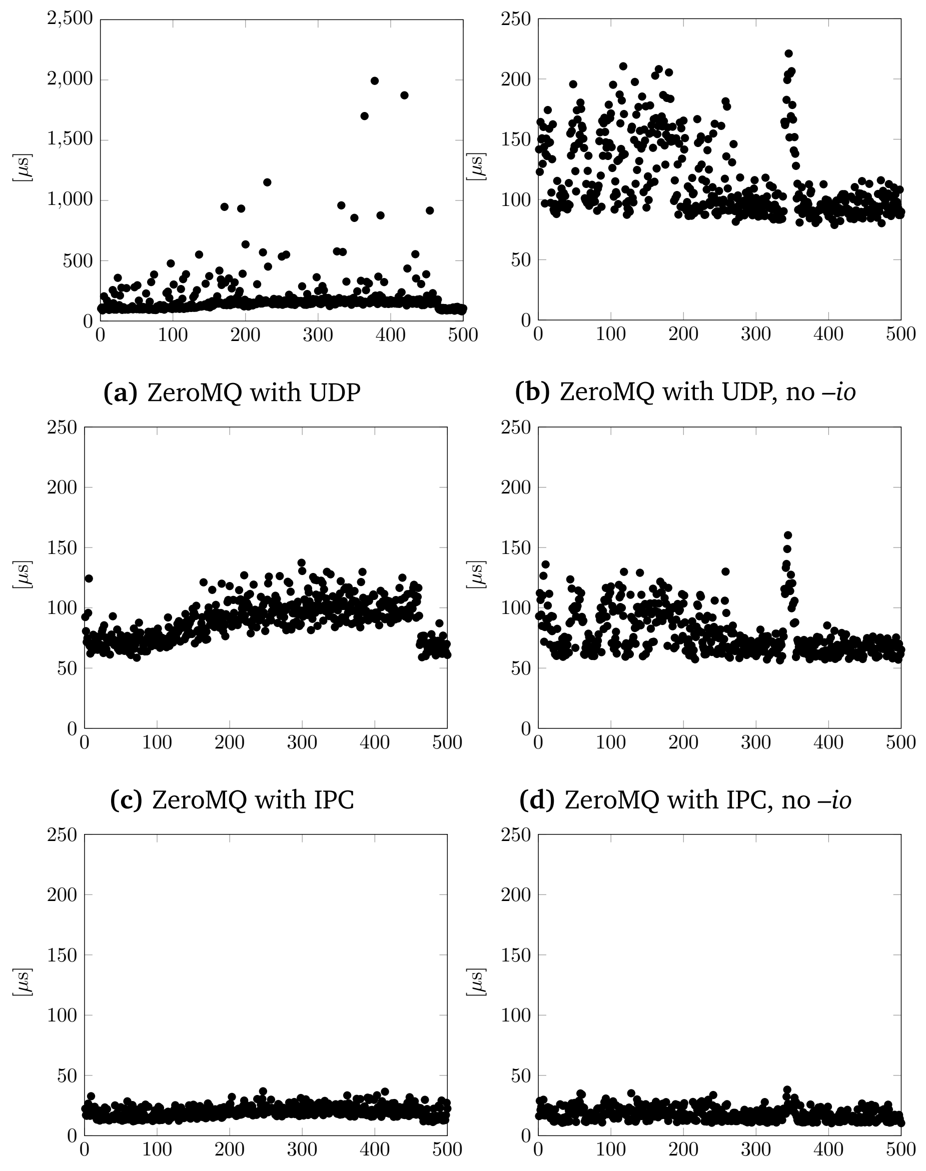


Figure 5.1: Diagram showing a benchmark that is split evenly into cycles, and a possible cycle execution

[1]http://kernel.ubuntu.com/~cking/stress-ng/

Figure 5.2 plots the maximum round-trip time within each run. We are only interested in the maximum times because a real-time system must meet its time constraints also in the worst case. Note that the UDP diagram uses a different time scale because its worst-case performance is worse by a factor of ten. This behavior only occurs if the *–io* option is specified on the stress test – otherwise, the round-trip times of UDP are only slightly higher than those of IPC (while the IPC times are not affected by the *–io* option), as seen in charts (b) and (d).



(a) ZeroMQ with UDP **(b)** ZeroMQ with UDP, no *–io*

(c) ZeroMQ with IPC **(d)** ZeroMQ with IPC, no *–io*

(e) Shared memory **(f)** Shared memory, no *–io*

Figure 5.2: Round-trip time from benchmark module to relay module and back, with different transports (a,c,e: with *–io* option in stress-ng; b,d,f: without *–io* option)

Plot (e) and (f) show the results of the custom ring buffer transport using shared memory and semaphores, as described in Section 4.3.5. In comparison to ZeroMQ's IPC, its worst-case times are lower by a factor of three. The difference is to be attributed to the difference in complexity – the shared-memory approach does not even use separate threads for communication.

6 Conclusion

We analyzed the opportunities and challenges that come with a real-time control application based on containers and proposed a reference architecture that enables reusability, portability, and flexibility. The architecture incorporates solutions for communication between containers and between containers and hardware. We showed that the *PREEMPT_RT* patch can be combined with the Cobalt kernel and that Cobalt-based applications can be run within containers.

With a prototypical implementation of the container orchestration runtime, we executed benchmarks that test round-trip time of messages with different transport methods. The results suggest that round-trip times between 50 and 150 μs in the worst case are feasible and thus it is possible to implement an application with a periodic interval of 500 μs can be split into several dependent modules.

As part of future work, we consider developing standards for message types to increase the reusability of modules within different applications. From a technical viewpoint, the performance differences with different transport protocols need to be further analyzed, i.e., it should be determined if the IPC functionality of ZeroMQ can be lifted to comparable round-trip times as the ring buffer transport using shared memory.

Bibliography

[BK13] K. D. Bettenhausen, S. Kowalewski. "Cyber-physical systems: Chancen und Nutzen aus Sicht der Automation." In: *VDI/VDE-Gesellschaft Mess-und Automatisierungstechnik* (2013) (cit. on pp. 9, 10).

[DIN14] E. DIN. *62264 DIN EN 62264: Integration von Unternehmensführungs-und Leitsystemen*. 2014 (cit. on p. 9).

[GHS16] T. Goldschmidt, S. Hauck-Stattelmann. "Software Containers for Industrial Control." In: *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2016, pp. 258–265. DOI: 10.1109/SEAA.2016.23 (cit. on p. 11).

[GTJ13] O. Givehchi, H. Trsek, J. Jasperneite. "Cloud computing for industrial automation systems—A comprehensive overview." In: *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE. 2013, pp. 1–4 (cit. on p. 11).

[Gil+97] J. H. Gilmore et al. "The four faces of mass customization." In: *Harvard business review* 75.1 (1997), pp. 91–101 (cit. on p. 9).

[Gol+15] T. Goldschmidt, M. K. Murugaiah, C. Sonntag, B. Schlich, S. Biallas, P. Weber. "Cloud-based control: A multi-tenant, horizontally scalable soft-plc." In: *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*. IEEE. 2015, pp. 909–916 (cit. on p. 11).

[Hoh03] G. Hohpe. *Hub and Spoke [or] Zen and the Art of Message Broker Maintenance*. 2003. URL: http://www.enterpriseintegrationpatterns.com/ramblings/03_hubandspoke.html (cit. on p. 14).

[LM13] R. Langmann, L. Meyer. "Architecture of a web-oriented automation system." In: *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*. IEEE. 2013, pp. 1–8 (cit. on p. 10).

[McK05] P. McKenney. *A realtime preemption overview*. 2005. URL: https://lwn.net/Articles/146861/ (cit. on p. 14).

[Nak02] B. Nakatani. *User Mode Drivers*. 2002. URL: http://www.linuxjournal.com/article/5442 (cit. on p. 31).

[Pro99] F. M. Proctor. *Linux, Real-Time Linux, & IPC*. 1999. URL: http://www.drdobbs.com/open-source/linux-real-time-linux-ipc/184411098 (cit. on p. 30).

[Rab10] RabbitMQ. *Broker vs Brokerless*. 2010. URL: https://www.rabbitmq.com/blog/2010/09/22/broker-vs-brokerless/ (cit. on p. 15).

[Raj+10] R. R. Rajkumar, I. Lee, L. Sha, J. Stankovic. “Cyber-physical systems: the next computing revolution.” In: *Proceedings of the 47th design automation conference*. ACM. 2010, pp. 731–736 (cit. on p. 9).

[Ric53] H. G. Rice. “Classes of recursively enumerable sets and their decision problems.” In: *Transactions of the American Mathematical Society* 74.2 (1953), pp. 358–366 (cit. on p. 32).

[Rot16] A. Roth. *Einführung und Umsetzung von Industrie 4.0: Grundlagen, Vorgehensmodell und Use Cases aus der Praxis*. Springer-Verlag, 2016, pp. 47–72 (cit. on p. 9).

[Scha] *Deadline Task Scheduling. Linux Kernel Documentation*. URL: https://www.kernel.org/doc/Documentation/scheduler/sched-deadline.txt (cit. on p. 17).

[Schb] *sched(7). The Linux Man Pages*. URL: http://man7.org/linux/man-pages/man7/sched.7.html (cit. on p. 16).

[Sus08] M. Sustrik. *Tests on Linux Real-Time Kernel*. 2008. URL: http://zeromq.org/results:rt-tests-v031 (cit. on p. 27).

[Zer17] ZeroMQ. *Broker vs. Brokerless*. 2017. URL: http://zeromq.org/whitepapers:brokerless (cit. on p. 15).

All links were last followed on March 17, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature