Real-time Simulation Executive Architecture and Subsystem Containerization

Zack Kirkendoll, Matthew Lueck
CymSTAR, LLC
Tulsa, OK
zack.kirkendoll@cymstar.com, matthew.lueck@cymstar.com

ABSTRACT

The design, development, and deployment of a containerized real-time simulation executive architecture for scheduling and executing high fidelity software system models poses many challenges. Both industry and the Department of Defense (DoD) are researching technologies and software platforms to provide efficiency and effectivity benefits through DevSecOps and containerization. While DevSecOps has propagated toward the modeling and simulation industry, there are still significant obstacles to be addressed to support a systematic paradigm shift.

Real-time simulation domains are rarely plug and play environments, especially for full-flight simulators. There are external aspects to include haptic, visual, kinetic, aural, and temporal components. Real-time considerations include preemptive task switching and sequencing, prioritization, jitter, and latency. Necessary real-time services include shared memory, semaphores, inter-process communications, and peripheral interfaces to include Ethernet, MIL-STD-1553, ARINC series, serial, standard I/O, and distributed training environments.

This paper evaluates these considerations within the applied domain of a real-time full-flight simulation executive architecture running high fidelity software system models within a containerized architecture. The proposed simulation executive architecture runs on a Linux operating system with a real-time kernel patch. The real-time executive runs a collections of software modules scheduled and executed in a deterministic real-time fashion for the simulation. This executive architecture has been utilized for multiple DoD USAF full-flight simulators including FAA 14 CFR Part 60 Level D compliant systems requiring Security Content Automation Protocol (SCAP) compliance.

First, a review of industry and DoD activity as related to this concept model is decomposed. Second, the problem domain and challenges to its application are defined. Finally, a detailed open framework solution for implementing a containerized real-time simulation executive architecture and subsystems is provided. This is critical to enable system designers to achieve a successful deployment for real-time flight simulation applications.

ABOUT THE AUTHORS

Zack Kirkendoll is a Principal Software Engineer at CymSTAR, LLC. Zack began in the simulation industry in 2013 while pursuing a degree in Electrical Engineering from the University of Tulsa. Zack has attained broad experience in aircraft simulation and training to include various programs for both military and commercial flight simulators and maintenance trainers. Zack's current role is as the C-5 Aircrew Training Systems (ATS) Program Engineer.

Matthew Lueck is a Principal Software Engineer at CymSTAR, LLC. Matthew began in the simulation industry in 2000 after graduating from Rose Hulman Institute of Technology with a degree in Computer Engineering. Matthew is the lead architect that designed and developed the real-time simulation executive used as the basis for this research.

Real-time Simulation Executive Architecture and Subsystem Containerization

Zack Kirkendoll, Matthew Lueck
CymSTAR, LLC
Tulsa, OK
zack.kirkendoll@cymstar.com, matthew.lueck@cymstar.com

INTRODUCTION

Aircraft avionics providers have been experimenting with and leveraging software architectures that support continuous integration, continuous delivery, and continuous deployment (CI/CD). CI/CD requires a sufficient Development, Security, and Operations (DevSecOps) pipeline to deploy software that can be easily and rapidly updated while maintaining a continuous Authority to Operate (cATO). An important architecture to support this initiative is container-based virtualization (containerization). To achieve and maintain simulator concurrency with the aircraft, the efficiency and performance benefits of containerization will become necessary. These same benefits propagate toward improved reliability, maintainability, sustainability, and supportability of the training system simulators while providing cross-platform utilization of common software systems to minimize costs and maximize benefits for the users.

Real-time processes must provide guaranteed maximum response times for external inputs, enable high-priority processes to preempt lower priority processes for central processing unit (CPU) utilization, maintain exclusive access to the CPU, and control the precise order in which component processes are scheduled.

Container-based virtualization is a widely used lightweight alternative to hypervisor-based virtualization. Its high utilization and viability are found in large cloud-based systems and supported by orchestration and monitoring tools. However, in industrial domains requiring real-time software containerization the technology adoption and utilization rates have been low due to its immaturity (Hofer, 2021). The containerization of real-time components in operational training systems suffer from similar shortfalls experienced by containerized software in industrial domains, leaving an open field of research to be explored.

Containerization leverages self-contained individual containers, or a group of containers, to provide all necessary functionality for a specific application. For example, utilizing modular component-based and model-based software design principals, as is common in simulation. Component-based software design, development, and engineering is a common approach to address complexity in software applications providing improved modularity, reusability, and distribution. Complex software systems are decomposed into distinct modules, or subsystems, described by well-defined interfaces, with each subsystem developed and tested independently, and various subsystems integrated onto the target platform. With real-time software, those subsystems must also include non-functional attributes such as timing requirements.

If containerization is to be widely adopted for real-time modeling and simulation, it must provide clear advantages over legacy approaches to real-time simulation and virtualization.

First, a review of industry and DoD activity as related to this concept model is decomposed. Second, the problem domain and challenges to its application are defined. Finally, a detailed open framework solution for implementing a containerized real-time simulation executive architecture and subsystems is provided.

BACKGROUND AND RELATED WORK

Prior to discussing the proposed open framework, the motivation and rationale for the use of virtualization and containerization will be covered. Both options provide value to simulation systems, but to further understand containerization and its applicability, its historical context will be decomposed. The focus will then converge on real-time simulation applicability. Figure 1 provides a conceptual representation of virtual machines and containers.

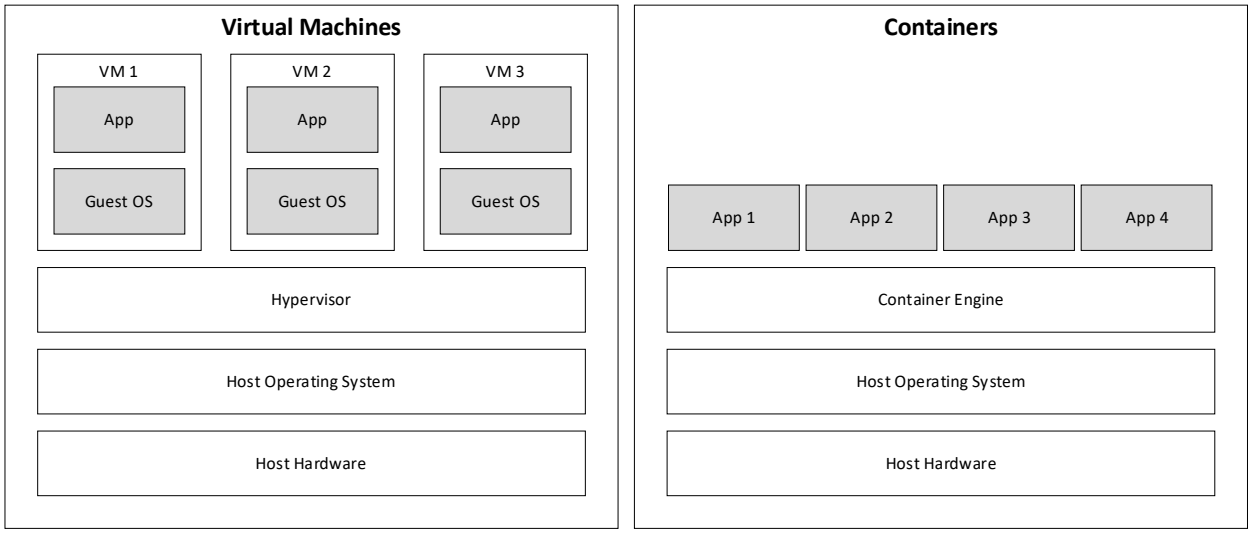

Figure 1. Conceptual Architecture for Virtual Machines and Containers

Without Virtualization and Containerization

Without the utilization of virtualization and containerization, developers need to manually prepare the environment, install libraries and modules, manage required software packages, and at best leverage installation scripts and configuration files for some efficiency gains. The result is that all software and their dependencies get installed to the machine. For new projects or systems, the developer would manually go through that again for each new system. Often, development and production environments would have differences either intentionally or unintentionally by deviating over time with different setups and configurations, where required production updates would increase that deviation. This issue is further compounded by the historical long lead times of large-scale software upgrades and deployments including operational flight program (OFP) upgrades, and impacts of cyber security compliance updates.

By the time deployment would occur, enough deviation between development and production environments with different versions would exist that may cause functionality to break or perform differently. There are often issues with conflicting dependencies of software installed on the production environment, and it becomes a configuration issue with bloated applications and systems. Managing different versions of different modules and libraries for dependencies is a significant version control issue. Development and production environments become out of sync from the configuration managed baseline and become difficult to track, manage, and reproduce as deviations are typically undocumented. This is exacerbated by different platforms and operating systems. Software and package management for starting, stopping, restarting, and managing applications and services becomes complex and tedious. Efforts to move, reproduce, and rehost to a new environment become time-consuming and costly. OS patching and security updates become tightly coupled to the software applications and their environment running on the respective systems. This is currently the standard common state for real-time modeling and simulation systems.

Virtualization

To provide encapsulation and separation for a system consisting of an operating system and one or more applications, virtual machines have become the most common approach. Virtualization requires an entire operating system or virtual machine to run on the server. Encapsulation is valuable but it comes at a performance cost and is often excessive for its intended purpose (Felter, 2015). Especially for including overlapping setups, different versions or instances of applications, and handling resource management and prioritization. To create, manage, and monitor virtual machines, a hypervisor is required. A hypervisor allows one system to support multiple guest virtual machines by allowing operating systems to run independently from the underlying hardware while virtually sharing computing resources including storage, memory, processing, and I/O. Virtualization allows for a software environment configured in a very specific way while providing the isolation, portability, and security that developers need. This solution is very flexible, allowing full configuration for the entire OS that is being virtualized to be captured and packed into an ISO image file that can be deployed on different host systems. Non-real-time systems and support software have begun leveraging virtualization within the modeling and simulation domain.

Containerization

Deployed software (one or more applications) is bundled with their entire runtime environment including supporting files, libraries, dependencies, environment variables, and configuration files in an isolated package to create a lightweight and portable application (single unit, image) that can run the same on a variety of operating systems, environments, and hardware. User space is where the applications and some drivers execute, and kernel space is strictly reserved for running a privileged operating system kernel. The host OS constrains the container's access to physical resources, such as CPU, storage, and memory, so a single container cannot consume all a host's physical resources. Containers maximize portability, since an application has everything required to run within the container engine, it becomes easy to move the contained application between environments while retaining full functionality (no degradation). Figure 2 shows a container application compared to a regular process within the hierarchy of user space, kernel space, and the hardware platform. The GNU C library (glibc) is commonly used in real-time simulation as a dependency that would be included along with the containerized application.

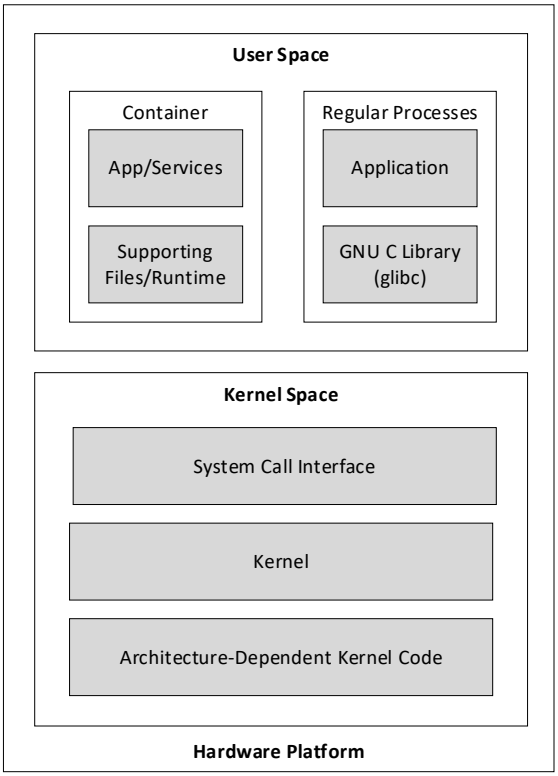

Figure 2. Containerized Application within User Space

Operating System (OS) Considerations

For real-time and simulation considerations, Linux is the preferred OS due to real-time capabilities, open-source, cyber security compliance, large ecosystem, hardware support, performance, and reliability. Similar functionality can be achieved using proprietary solutions such as VxWorks but that is not addressed in this research as the focus is on avoidance of proprietary approaches and following recent simulation trends toward Linux. Linux containerization has emerged as an instrumental open-source solution utilizing lightweight application packaging and isolation with highly flexible deployment capabilities. The Linux Container Daemon (LXD) tool is an extension to Linux Containers (LXC) as a method for accomplishing containerization within Linux in a similar method as Docker. Processes running inside the container appear to be running on a normal Linux system although they are sharing the underlying kernel with processes located in other namespaces. There are three main Linux kernel features to address (1) namespaces, (2) control groups (Cgroups), and (3) capabilities which are shown in Figure 3 (Kerrisk, 2010).

Namespaces virtualize global resources (processes, inter-process communication) providing the appearance that each container has its own OS, limiting what a process can see and restricting resources available. Systems can have multiple namespaces that are isolated from each other which aid in organization, security, control, and performance.

Cgroups are fundamental to containers as a mechanism to aggregate and partition sets of tasks (processes, groups of processes) into hierarchical groups, allocate and limit accessible resources (CPU, memory, and I/O), monitor and meter resources, and provide granular control of the containers.

Capabilities allow granular privileged access control for processes performing functions. Each privileged operation is associated with a particular capability, and processes can be assigned specific subsets of those capabilities.

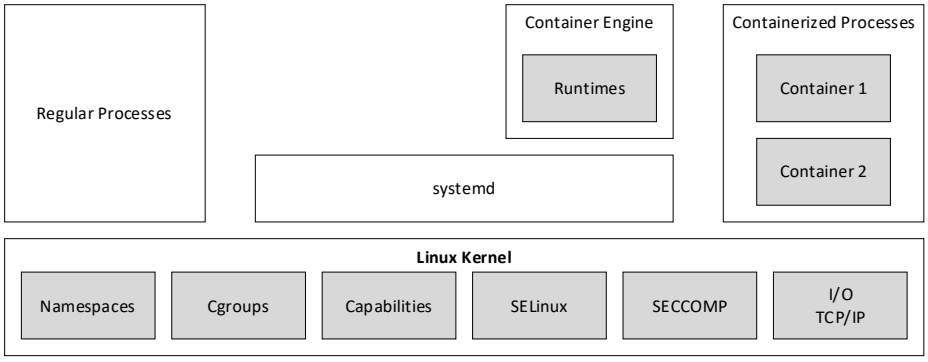

Figure 3. Containers, Processes, and Kernel Interactions

Comparison to Virtualization

For Virtual Machines, software is packaged with the entire OS, requiring more resources than we typically need for real-time functions, and it takes over all memory that is available to it even though the application requires much less memory. For example, a basic application that performs a simple computation still must be encapsulated into an entire OS. Containerization provides many of the same benefits at a much lower cost by being more lightweight and requiring fewer computing resources. Container based approaches do not require a hypervisor and therefore it provides near-native performance, rapid deployment times, and a low overhead while still retaining a sufficient level of resource isolation and resource control. Containerization outperforms virtualization for all real-time considerations (Felter, 2015). It is worth noting that containers can run in virtualization, but that has less applicability for real-time applications. Typically, Linux containers would run on Linux, and Windows containers on Windows, but there are some mechanisms to run cross-platform utilizing an additional layer of virtualization but that comes at a cost of performance and loss of real-time control.

Common Platforms: Docker and Kubernetes

An objective of containerization platforms is to remove manual development and management of containers. Docker is a platform, or container engine, for packaging and running containers on a single system. Docker can build standard containers and images that include all their necessary dependencies. Docker simplifies and streamlines the process of creating, packaging, distributing, and using software on any platform and at scale.

Kubernetes orchestrates containers across a cluster. Kubernetes is an open-source platform that provides active observations, automates container operations, eliminates manual processes for deployment and scaling, and schedules and runs containers. Kubernetes provides resiliency (self-healing, restarting), embedded security, adaptability (dynamically swap, reduce downtime), automation, and auto-scaling (computation and memory requirements). All Kubernetes deployments have at least one cluster. A cluster contains nodes. Nodes host pods. Pods include a running set of containers. Kubernetes manages all of this. Kubernetes may be an overkill on non-enterprise-scale level systems and was originally designed for Google initiatives and requirements.

One of the most critical aspects is to avoid vendor lock-in. To achieve this objective, Open Container Initiative (OCI) containers and Cloud Native Computing Foundation (CNCF) Kubernetes compliant cluster orchestration are required. This allows use of different platforms and tools for containerization control, management, and orchestration. This also means that Docker and Kubernetes are not required, they are only common tools that are compliant to the appropriate standards.

Red Hat Enterprise Linus (RHEL) Container Tools Module and OpenShift

The Docker platform is not shipped or supported directly by Red Hat starting with RHEL 8. Red Hat does not recommend using Docker, but rather to use the command-line container-tools module that can operate without a container engine. The equivalent container-tools to Docker are Podman, Skopeo, and Buildah for OCI tooling to create, run and manage Linux containers. RHEL provides a container orchestration platform called OpenShift which is an enterprise ready Kubernetes platform. RHEL Container Tools Module and OpenShift are OCI specifications and CNCF Kubernetes compliant. Docker, Kubernetes, and LXD/LXC can still be used on RHEL, in theory.

For real-time considerations, OpenShift has begun to add automatic tuning to achieve low latency performance and consistent response times using a Performance Addon Operator at the enterprise-scale level orchestration. This addon enables declarative management of machines and host operating systems to provide the necessary real-time tunings and configuration sets for the real-time kernel patch, CPU processor core allocation, and network interface controllers (NIC). This approach provides another mechanism for real-time and low latency optimization at the container orchestration level.

General Benefits

Containerization technology provides consolidation of computational resources, a flexible environment for running real-time applications, interruption-free hardware and software maintenance, dynamic system redundancy, reliability, convenience, faster deployment, elasticity, and performance. Containers result in near-zero overhead, less RAM usage, and less memory space requirements overall, especially compared to virtualization. Containers are stateless and can be restarted without impact (instant replication) allowing for faster start, stop, and restart times.

Containerization provides enterprise level benefits including integration for different code bases and architectures, sharing containers across different platforms and different runtime environments. Containerization leverages the DevSecOps culture with improved testing and CI/CD, portability, security, and version control. With hardware and software development moving more rapidly, the ability afforded by containerization to focus on the application development rather than the underlying hardware allows the reuse of existing certified code to speed development of new capabilities. This allows OS updates to be separated from application container development and allowing centralized OS patch validation and testing to occur outside of software application development.

Real-time Container Research

The currently available containerization platforms (engines and orchestrators) are not designed to easily enable integration with the standard real-time full-flight simulation architectures.

Research evaluating the performance of native bare metal systems and container engine systems have similar results, while virtualization exhibited much worse performance (Felter, 2015). There was no significant overhead on CPU, memory usage, or bandwidth, with largely the performance impacts occurring on I/O and OS interactions through kernel space and system calls. The use of network address translation (NAT) exhibited performance issues for containers, where no use of NAT has identical performance to native systems. The quantity of NICs is a likely bottleneck for real-time systems and their interfaces to external physical systems.

There are many active research topics providing differing mechanisms for providing real-time (hard, firm, and soft) capability to containerization in Linux (Fiori, 2022) (Hofer, 2021) (Struhár, 2021). There were four approaches to enabling real-time behavior for containers investigated including real-time patch based (PREEMPT_RT kernel patch), co-kernel based (Xenomai), hierarchical scheduling of containers, and custom methods. PREEMPT_RT and Xenomai are likely best performing options, with PREEMPT_RT being more commonly used for the modeling and simulation industry. Various deterministic scheduling policies (SCHED_FIFO, SCHED_RR, SCHED_DEADLINE) for scheduling containerized applications in the same manner as native applications were used across the literature.

The primary shortfalls and technological immaturities identified for real-time containerization were grouped into three categories: (1) lack of tools for real-time container management and orchestration, (2) communication between and across real-time containers and the environment, and (3) issues with memory, I/O, security, and needing more clear performance measures for task determinism and real-time (Struhár, 2020).

APPLICATION TO REAL-TIME FULL-FLIGHT SIMULATORS (RTFFS)

Domain Definition

Real-time full-flight simulators include haptic, visual, kinetic, aural, and temporal components. To achieve the training objectives, all components must occur deterministically as modeled within the real-time simulation. There are complex and distributed software and hardware systems needed to achieve the realism and training fidelity required. The real-time simulation executive is the primary scheduler and orchestrator for all real-time systems and tasks available on the simulator. Scheduling ensures low-latency preemptive task switching and sequencing. Jitter as the variance in time expected for a task to execute must be minimized within the cycle boundary. The scheduler must handle CPU processor core allocation, timing rates, prioritization (real-time, asynchronous, background), and multiple temporal execution phases (initialization, runtime, shutdown, freeze, and diagnostics).

Real-time Simulation Executive Architecture

Default Linux does not provide any time guarantees on execution of tasks and therefore determinism is low. However, the real-time preempt kernel patch PREEMPT_RT provides preemptive task switching capability for the Linux kernel.

The framework used within this body of work is an open-source real-time simulation executive framework allowing for the integration of high-fidelity aircraft systems modeling and simulation. The simulation executive software framework operates on a Linux operating system using a real-time Linux kernel patch. The real-time executive framework provides an environment in which collections of software modules are scheduled and executed in a deterministic real-time fashion for the software-in-the-loop simulation while enabling real-time hardware-in-the-loop control. The executive framework defines threads of execution based on sequencers. Each sequencer represents one thread and may optionally be assigned to a specific central processing unit (CPU) processor core. A high priority synchronizer thread (scheduler) is responsible for controlling the execution of the sequencers, based on Portable Operating System Interface (POSIX) high resolution timers provided by the operating system which allows the accuracy of the timers used to be as accurate as the underlying hardware allows. In turn, each sequencer is responsible for executing specific software modules. Each sequencer is tied to a defined clock rate that specifies the frame rate for that sequencer. Each sequencer executes its assigned modules within the context of an execution cycle consisting of a certain number of frames. Modules and subsystems will be used interchangeably despite some differences.

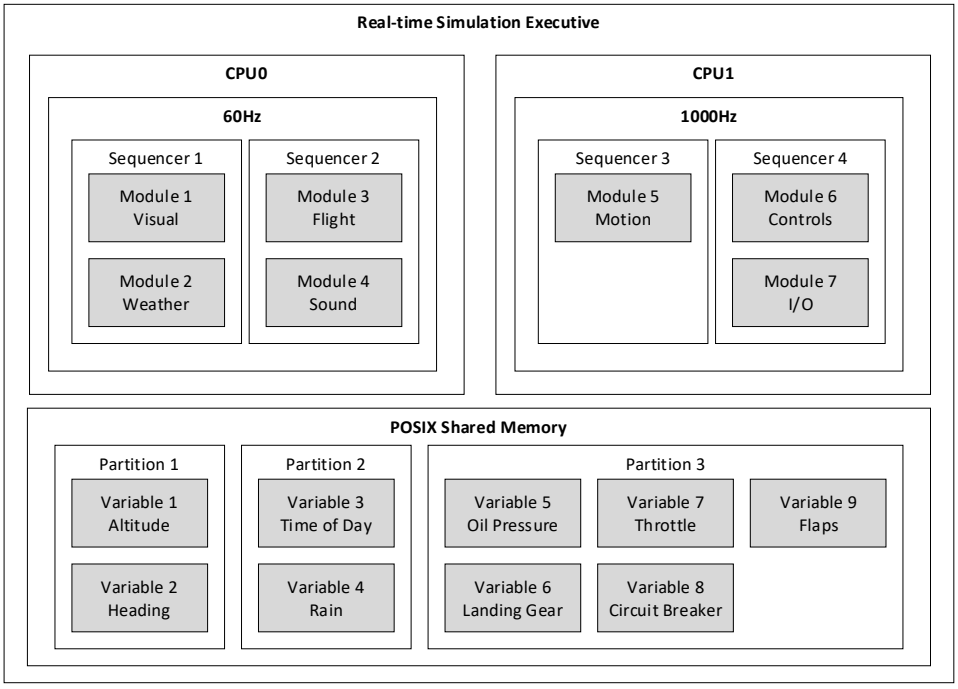

Figure 4. Real-time Simulation Executive Architecture

The framework utilizes threads of execution to allow shared (global) memory access between threads, rapid thread creation, and attribute inheritance. Threads also allow for distinct real-time scheduling policies and prioritization, error handling, processor core affinities, and distinct local stack memory. The framework utilizes semaphores as a mechanism for achieving mutual exclusion and synchronization with priority inheritance providing a viable way to access shared resources across threads. These mechanisms enable a global shared memory architecture capable of achieving real-time performance.

The real-time executive framework, as shown in Figure 4, has been deployed to multiple DoD USAF full-flight simulators including FAA 14 Code of Federal Regulations (CFR) Part 60 Level D compliant systems. Implementation has also included creation and deployment of an automated RHEL equivalent Secure Host Baseline (SHB) for real-time systems requiring Security Content Automation Protocol (SCAP) compliance to improve system cyber security posture for achieving an authority to operate (ATO) (Kirkendoll, 2020). This real-time executive framework has been utilized for research level fixed-wing aircraft automatic safety system modeling and simulation using run-time assurance architectures (Kirkendoll, 2021).

Due to multiple processes and threads depending on rapid communication and sharing the same region of memory, mechanisms are required to coordinate those activities efficiently. Inter-process communication (IPC) is the collective name for methods which processes use to manage shared data. Figure 5 shows the limited taxonomy of UNIX IPC facilities as it relates to real-time full-flight simulation. For real-time considerations, IPC can be further decomposed into communication that exchanges data between processes and threads, and synchronization for the actions of processes and threads. Communication can be decomposed into data transfer facilities that require kernel memory access, and shared memory which allows processes to exchange information by placing it in a region of memory that is shared between processes. Shared memory communication does not require system calls or data transfer between user memory and kernel memory, so shared memory can provide faster communication at the cost of more complex synchronization methods (Kerrisk, 2010). The primary synchronization facilities include semaphores, file locks, and mutexes. For real-time full-flight simulation IPC across threads, the primary mechanisms used are POSIX shared memory for communication synchronized through semaphores. This approach works exceedingly well for large scale and complex architecture designs that must maintain shared states and large data structures temporally.

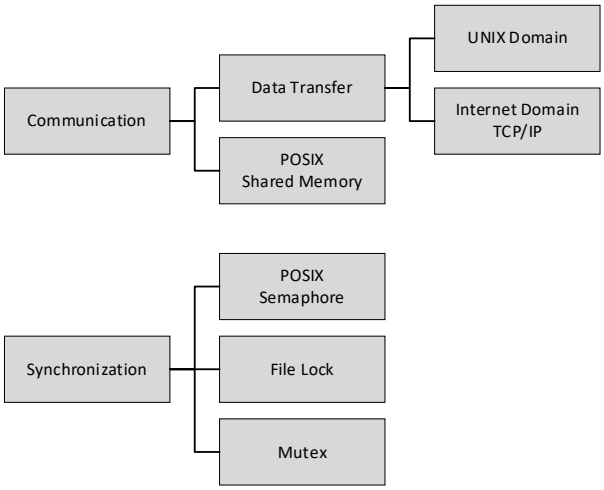

Figure 5. Limited Taxonomy of UNIX IPC Facilities

Communication using data transfer facilities requiring network communication can be accomplished using sockets. The two available domains are the UNIX domain for communication between processes on the same system, and the Internet domain for communication between processes on different hosts over TCP/IP. UNIX domain socket communication approaches are less common but still used situationally. Internet domain socket communication is very common for both real-time and non-real-time communication to external systems including I/O, control loading, motion, visual, aural, operator controls, distributed training environments, threat generation, and other local but isolated systems. Internet domain network communication and driver configuration are one of the more complex types of drivers in Linux especially for real-time performance and must be considered very carefully.

MIL-STD-1553, ARINC series, serial, standard I/O, and other avionics data bus communication protocols are very common in real-time full-flight simulation for both simulated and stimulated systems. These communication methods are accomplished by onboard PCIe and offboard Ethernet based controllers. Both systems require similar levels of complexity for driver configuration and tuning as network communication to ensure real-time performance.

A Linux signal is a notification to a process that an event has occurred, acting as a software interrupt. While signals are most likely be used within some part of the overall simulation framework (e.g., at initialization), there is less emphasis for them as part of this research as their usage is more nuanced and specific to the exact architecture and design approach used. But it is worth noting that they are not the primary mechanism for IPC synchronization.

It is important to note that there are decades of precedence for all of modeling and simulation in using these or similar architectural design decisions for real-time performance capabilities. Less complex simulation environments may only utilize a subset of these design aspects.

Challenges

While translating the real-time full flight simulation domain requirements and approaches to containerization utilization poses significant challenges itself, there are further considerations to assess. Many of these challenges are due to the immaturity level of containerization technology and its non-real-time domain utilization.

These concerns include simulation specific real-time constraints within the containers, interfacing to hardware systems (real, simulated) including aircraft hardware, dealing with complex proprietary software and systems, legacy computational systems, and non-standardization. There is a lack of tools for real-time container orchestration and management that can schedule real-time containers based on pre-configured timing parameters, middleware that is aware of communication and real-time performance isolation needs, a framework to expose runtime requirements of real-time applications running inside containers, and to enforce optimal allocation of containers to resources.

There are significant challenges to orchestrating communication and interfaces between containers and other processes. Most real-time systems only utilize threads rather than synchronizing with multiple processes, and containers act as distinct processes. This communication challenge includes real-time inter-process communication among containers, data management shared across containers, and consideration for both industry and DoD technology initiatives and utilization trends which indicate an increased reliance on and necessity for Ethernet based real-time network communication. This communication includes standard TCP/IP and Ethernet based avionics data bus communication to offboard systems.

OPEN FRAMEWORK SYSTEM COMPONENTS AND APPROACH

The technical considerations and domain challenges defined in the previous sections will be decomposed into an open framework system and approach to solving those challenges for real-time full-flight simulation containerization. The detailed open framework proposed addresses the three primary shortfalls identified in the literature for real-time container management and orchestration tools, real-time container communication, and memory, I/O, and measures of performance. This approach is applicable to all of DoD and commercial modeling and simulation but especially for real-time full-flight simulators. The framework allows for leveraging existing real-time simulation executives, migrating existing subsystems to containers, and building new systems using containerization. The proposed framework bridges the gap between standard DoD real-time simulation and industry technological trends toward software containerization. This approach and actionable guidance are critical to enable system designers to achieve a successful deployment for real-time full-flight simulation applications.

The open framework system approach is Portable Operating System Interface (POSIX) and Single UNIX Specification (SUS) compliant using the PREEMPT_RT kernel patch to ensure real-time performance is met across UNIX/Linux distributions and architectures. This compliance allows precise CPU processor core affinity allocation, real-time scheduling policies, inter-process communication and synchronization, and priority preemption. The real-time scheduler utilizes a first in first out (SCHED_FIFO) scheduler policy with POSIX high resolution timers.

Explicit processor affinity allocation between real-time and non-real-time processes is enabled. Setting explicit processor affinity for real-time processes avoids performance impacts caused by invalidation of cached data, improves

performance for multiple threads and processes accessing the same data (avoid cache misses), and allows non-real-time processes to be confined to other non-real-time processor cores. For example, on an 8-core system, processor cores 0-3 are for real-time and processor cores 4-7 are for non-real-time.

Internet domain network communication and driver configuration are one of the more complex types of drivers in Linux especially for real-time performance. The New Application Programming Interface (NAPI) functionality can facilitate tuning for real-time performance through configuration of interrupt moderation, disabling of interrupt coalescing, and adjusting receive side scaling (RSS) and receive packet steering (RPS). Explicit real-time (RT) and non-real-time NIC allocation is used. NAT utilization is avoided for real-time. The quantity of RT NICs available is a bottleneck for real-time performance and must be adequately allocated. Figure 6 shows this recommended network communication and driver configuration approach.

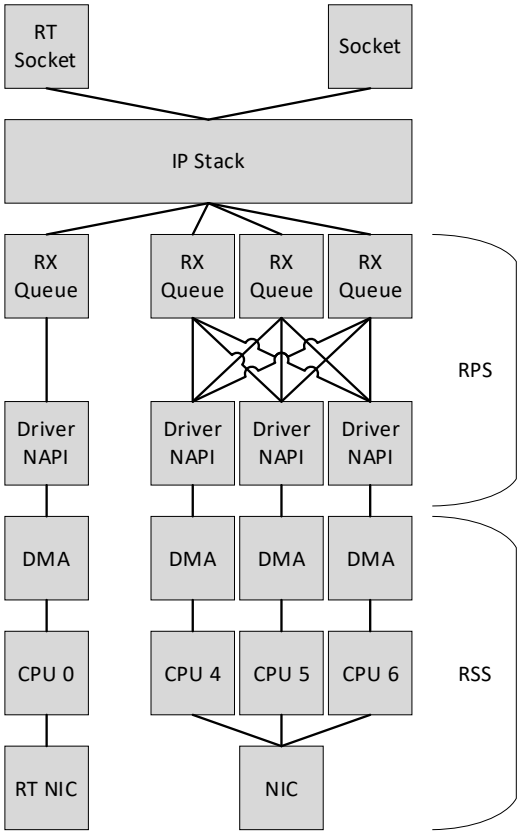

Figure 6. Network Communication and Driver Configuration

The containerization management and orchestration approach is OCI/CNCF compliant. The real-time simulation executive is extended to support a limited subset of OCI/CNCF compliant container management and orchestration operations. This extension provides a sufficient real-time container management and orchestration executive framework. It is possible for the real-time executive framework to interface directly through existing OCI/CNCF compliant container engines (Docker, Kubernetes, RHEL Container Tools and OpenShift, LXD/LXC) to leverage the existing technology available as long as the real-time scheduling and orchestration is managed by the simulation executive. Orchestration and management by the simulation executive is essential for real-time determinism, prioritization, synchronization, and control.

Specific system recommendations include utilization of Red Hat Enterprise Linux (RHEL) on an x86-64 architecture. While today RHEL is the recommended OS of choice, none of these approaches or frameworks are exclusive to RHEL but rather are focused on POSIX, SUS, the real-time Linux kernel patch, and OCI/CNCF and therefore will scale over time based on the preferred Linux distribution in the future.

There are multiple approaches and many considerations for achieving real-time containerization of applications while maintaining resource isolation, resource control, inter-process communication (IPC) and synchronization, time determinism, and pre-defined behavior for the containerized application. For all real-time simulation platforms, there likely already exists a real-time simulation executive framework to be leveraged. At this phase of containerization utilization for modeling and simulation, the primary real-time simulation executive does not need to be containerized itself. The focus is on containerization of individual (or groups of) modules, subsystems, utilities, and other applications that interact with the primary simulation executive in real-time. IPC between containers and the real-time simulation executive can be accomplished through POSIX shared memory, UNIX domain sockets, and TCP/IP solutions which each includes various tradeoffs and design considerations (Miljö, 2018).

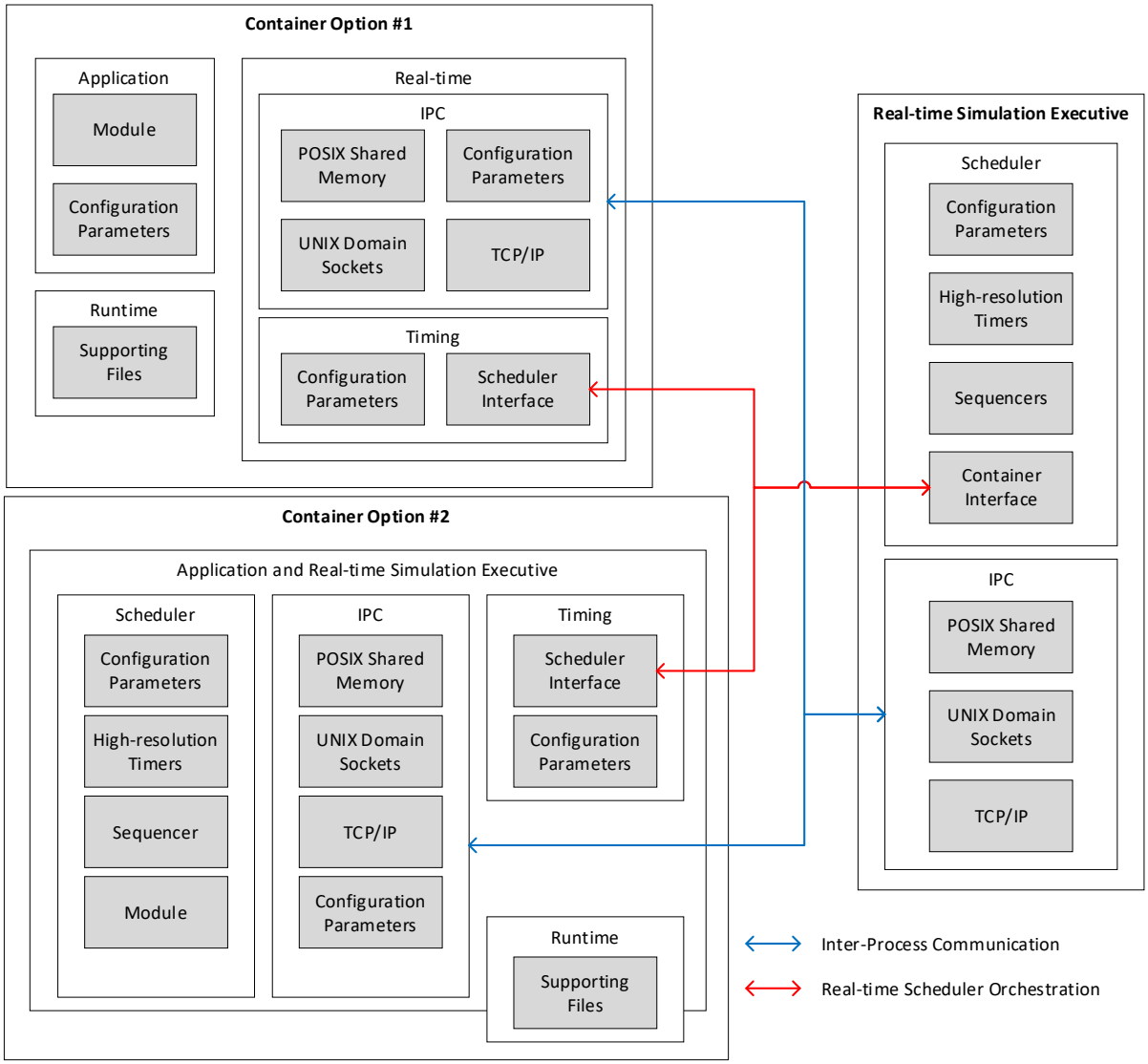

Figure 7. Real-time Simulation Executive Architecture and Subsystem Containerization Approach

There are two primary real-time containerization frameworks proposed as shown in Figure 7. These adaptable frameworks are new and unique as applicable to real-time full-flight simulation and containerization. Option #1 packages the application as a standalone process to be scheduled outside of the container by the real-time executive scheduler. Option #2 packages the application along with its own full real-time simulation executive. Option #2 is applicable to larger and more complex systems that are being containerized for specific reasons such as line replaceable units (LRUs) with many subprocesses that cannot easily be decomposed into individual containers. Both options require an interface to the real-time simulation executive but with different configuration parameters and interface design considerations. Both options require the real-time simulation executive to be the primary orchestrator

and scheduler for the overall simulation. The real-time simulation executive controls the timing, sequence, and synchronization of the containers as if they are standard modules being scheduled as part of the overall simulation.

Modules and subsystems that are containerized for real-time simulation must consider additional design mechanisms. Ideally, containerized modules and subsystems must utilize modular software design principles with well-defined interfaces. A common situation for legacy simulation subsystems that has propagated through many iterations and upgrades is to be less modular and be more distributed among various other subsystems with no clearly defined interface or separation. To containerize existing subsystems without significant architectural rework, it may be necessary to combine multiple subsystems that cannot be easily separated. While not ideal, this is likely an acceptable approach in the near-term for existing platforms while still allowing distribution to other platforms and simulators for integration if provided with sufficient design documentation. Newly developed containerized modules are recommended to follow modern modular software design principles where possible. Containerized modules are able to be inserted into any simulation executive scheduler's sequencer and within any order of those sequencers, as controlled by the scheduler. A timing interface from the container to the simulation executive scheduler is required to ensure real-time orchestration and management. Containerized modules must consider the simulation execution phases for initialization, runtime, shutdown, freeze, and diagnostics as part of the design. Initialization and shutdown phases must be handled to ensure the container is fully operational prior to runtime and that the application is cleanly shutdown prior to exiting, such as allocation and deallocation of system resources and IPC dependencies. Leveraging standard real-time simulation IPC within the containers themselves allows new and legacy subsystems to be more easily containerized without significantly altering the standard software architecture.

POSIX shared memory and semaphores allows processes, containers, and the real-time executive to utilize the same shared virtual memory by leveraging Linux namespaces, cgroups, capabilities, and other simulation specific configuration parameters. This approach provides the fastest IPC available and extends the standard full-flight real-time simulation approach to IPC across both threads and processes. Utilization of shared memory and its necessary synchronization across processes and containers requires additional upfront coordination, planning, standardization, and configuration as a part of the container design. Utilization of shared memory across containers and processes requires standardization (or interface description) for its format and structure to include types, number of elements, sizes, initialization values, and descriptors to be made available through IPC configuration parameters. Depending on the platform, there will be additional verification and validation required to ensure real-time performance is maintained due to the increase in complexity when compared to the socket interface design. Utilization of shared memory specifically allows both legacy and new subsystems to be more easily containerized without significantly altering the software architecture, minimizing the barrier to adoption. Typically, these additional complexities and mechanics are abstracted away from the developer once the baseline capability has been established.

Data transfer using both UNIX domain sockets and TCP/IP (loopback) are sufficient options for real-time communication by leveraging Linux namespaces, cgroups, capabilities, and other simulation specific configuration parameters. UNIX domain IPC will use user datagram protocol (UDP) as there are no downsides to a connectionless protocol on the same host. Throughput benchmarks show UNIX domain sockets can be significantly faster than TCP/IP loopback when both peers are on the same host. However, UNIX domain does assume that the container is on the same host system as the simulation executive, which is likely the case at this phase. Since TCP/IP functionality is not significantly more work or wholly unique from UNIX domain approaches then it is recommended to support both options for higher portability, scalability, and additional use cases.

Regardless of the IPC approach, there are multiple considerations that must be addressed as part of the design. Containers can communicate with both the simulation executive and other containers. The overall design architecture scales sufficiently well with both approaches as the containers themselves are orchestrated and managed by the simulation executive and its scheduler. Since the simulation executive creates, orchestrates, and manages the shared memory structures, containers will utilize that same shared memory space by using a common configuration descriptor to access it. There may be exceptions and situations in which the container itself needs to create and communicate over a shared memory structure, which works, but must be considered as part of the interface design and it will drive additional modifications to the simulation executive. Most real-time IPC approaches assume that the containers and simulation executive are on the same host system. If a distributed system across multiple host systems is used, then additional considerations and approaches are required such as focusing on TCP/IP network protocols for communication but it will retain the rest of the open framework with a modified scheduler interface. The selected IPC approach must be tested individually and at scale to ensure the often large and complex simulation system architecture

of flight simulators can sufficiently handle the throughput and magnitude of IPC selected, however, IPC throughput is not expected to be a primary constraint on modern systems with sufficient network and driver tuning.

To support real-time simulation executive orchestration and management of containers, additional configuration parameters and attribute values are required, especially to ensure portability for unique executive frameworks across platforms. Linux configuration includes namespaces, cgroups, and capabilities settings. Timing configuration parameters for the container includes applicable CPU processor core affinity allocation, timing rates (in Hz), priority (real-time, asynchronous, background), execution phase (initialization, runtime, shutdown, freeze, and diagnostics), and frame start. Applicable IPC configuration parameters for the container includes which IPC approach is utilized, specifics for shared memory descriptors and memory structure layout, and specifics for UNIX domain and TCP/IP network communication. These additional configuration complexities and mechanics are abstracted away from the developer once the baseline capability has been established.

Containerization does not resolve all issues for using proprietary solutions on full-flight simulators, but it does reduce the dependence on using the proprietary application with specific and often undocumented dependencies. Containerization of proprietary systems using the proposed framework allows the application to be used on a wider range of systems and for a longer period of time without incurring significant additional costs from the OEM.

Measures of Performance

The most useful real-time measurements of performance and for detecting issues are through memory allocation, long-term storage I/O, page faults, process timing overruns, and network throughput during real-time execution.

Tertiary storage I/O and memory allocation involves a process ceding execution to the kernel for a potentially lengthy, indeterminate amount of time, before resuming execution. Linux strace is a useful diagnostic and debugging tool for tracing all system calls conducted by a particular process and can identify instances where this is occurring for real-time processes. Real-time process requests for memory allocation using system calls (e.g. brk and mmap from glibc) can be identified, which can identify areas for further investigation where memory allocation is occurring after initialization and during real-time runtime phases which often times can be problematic.

Page faults occur when a process attempts to access an item in memory that is not immediately accessible, which are generally a source of latency. Real-time processes must not exhibit page faults after initialization phases and during their runtime phase. The Linux ps command can be used to identify these occurrences.

Measurements of process and thread execution duration to ensure maximum timing is less than the expected duration, and the average is statistically as expected, ensures no overruns are occurring on the system. This measurement and reporting is typically handled by the real-time simulation executive.

For network communication and driver tuning, real-time performance is having the interval between an event occurring (e.g., an interrupt) and the real-time process accessing the device or data sufficiently small so that the real-time process meets its deadline deterministically. Linux ethtool can identify dropped packets at the hardware level. Linux ftrace can identify sources of latency and measure time spent in functions in the kernel. Linux /proc/softnet_stat contains NAPI statistics for packets processed, dropped, and occurrences of a time squeeze. Linux /proc/interrupts shows all interrupts in the system and number of occurrences per core which identifies what interrupts are on the system, which interrupts are firing, how frequently, and if the distribution is as expected.

CONCLUSIONS AND RECOMMENDATIONS

In this paper, we have proposed, analyzed, and detailed an open framework solution for a real-time simulation executive architecture and subsystem containerization approach. The proposed framework addressed the three primary shortfalls identified in the literature for real-time container management and orchestration tools, real-time container communication, and memory, I/O, and measures of performance. This approach is applicable to all of DoD and commercial modeling and simulation but especially for real-time full-flight simulators. Approaches for migration and transition of legacy simulator systems has been considered along with new modern and modular software development paradigms. The framework allows for leveraging existing real-time simulation executives, migrating existing subsystems to containers, and building new systems using containerization. This approach allows multiple platforms

to benefit from the distribution of subsystem containers to achieve minimization of costs and maximization of benefits for the users. The proposed framework bridges the gap between standard DoD real-time simulation and industry technological trends toward software containerization.

Future efforts include using the proposed framework for the migration of existing modular subsystems and applications to containers, integration of containers into an FAA 14 CFR Part 60 Level D compliant RTFFS, and evaluation of measurements of performance to ensure real-time compliance. Further evaluation includes cybersecurity implications using industry standards, best practices, and risk mitigation to protect container development, container images, and their orchestration in support of RMF activities. Further research includes support for Simulator Common Architecture and Requirements Standards (SCARS) and Model-Based Systems Engineering (MBSE) initiatives.

Near-term Opportunities for Containerization in Real-time Full-flight Simulators (RTFFS)

There are many near-term common platform opportunities available to pursue including integration of enterprise repository components with RTFFS applicability such as Iron Bank, containerization of LRUs (e.g., radios), containerization of navigation databases (DAFIF) and their use, and containerization of DIS standard libraries. For weapon systems there can be containerization of off-rail weapon systems and in-weapon launch acceptability region (LAR) that can be provided by the OEM and used in the real-time simulation executive and other threat generation systems for increased accuracy and training value.

For an entirely self-contained real-time simulation executive architecture to be used within each individual container, an open source and standard solution must be designed, developed, and made available for future containerization efforts under the GNU General Public License (GPL). This approach will enable standardization, reduce development costs and timelines, reduce the barrier to entry, and increase development efficiency by keeping containerization development focused on the application being developed rather than the real-time performance mechanisms. This approach does not inhibit multiple solutions for real-time architecture standards but will provide a readily available open-source solution to facilitate containerization development at scale for technological initiatives across the DoD.

REFERENCES

Kirkendoll, Z., Lueck, M., Hutchins, N., Hook, L. (2020). Automated Linux Secure Host Baseline for Real-Time Applications Requiring SCAP Compliance. *National Cyber Summit (NCS), Advances in Intelligent Systems and Computing, vol 1271*, doi: 10.1007/978-3-030-58703-1_10.

Kirkendoll, Z., Hook, L. (2021). Automatic Ground Collision Avoidance System Trajectory Prediction and Control for General Aviation. *IEEE/AIAA 40th Digital Avionics Systems Conference (DASC), pp. 1-10*, doi: 10.1109/DASC52595.2021.9594506.

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. In 2015 IEEE international symposium on performance analysis of systems and software (ISPASS) (pp. 171-172). IEEE.

Fiori, S., Abeni, L., & Cucinotta, T. (2022). RT-Kubernetes—Containerized Real-Time Cloud Computing. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC '22). Association for Computing Machinery, New York, NY, USA, 36–39, doi:10.1145/3477314.3507216

Struhár, V., Behnam, M., Ashjaei, M., & Papadopoulos, A. V. (2020). Real-time Containers: A Survey. In 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Kerrisk, M. (2010). *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press.

Miljö, V., Johansson, F., & Lindstrom, C. (2018). Inter-Process Communication in a Virtualized Environment.

Hofer, F., Sehr, M., Sangiovanni-Vincentelli, A., & Russo, B. (2021). Industrial control via application containers: Maintaining determinism in IAAS. Systems Engineering, 24(5), 352-368.

Struhár, V., Craciunas, S. S., Ashjaei, M., Behnam, M., & Papadopoulos, A. V. (2021). REACT: Enabling Real-Time Container Orchestration. In 2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA) (pp. 1-8). IEEE.