MBDA
MISSILE SYSTEMS

Technische Hochschule Ingolstadt

Distributable Software Architecture for Synthetic Environment Generation in real-time Missile Simulations and Validations

Bachelor Thesis

University	Technische Hochschule Ingolstadt
Department	Computer Science WINF-B
Name	Daniel Mehlber
Student Number	00117554
E-Mail	dam7969@thi.de
1st Examiner	Prof. Dr. Robert Gold
2nd Examiner	Prof. Dr. Hans-Michael Windisch
Supervisor	Emanuel Derbsch
Issued on	01.12.2023
Submitted on	29.01.2024

Contents

List of Abbreviations **I**

List of Figures **II**

List of Listings **IV**

List of Tables **V**

1 Introduction **1**

1.1 Motivation . . . 1
1.2 Research Questions . . . 1

2 Technical Background **2**

2.1 Introduction to System Validation in Missile Engineering . . . 2
2.1.1 Missiles as Unmanned Autonomous Systems . . . 2
2.1.2 Validation Difficulties of Missile Systems in the Real World . . . 2
2.1.3 Modelling and Simulation in the Autonomous Systems Domain . . . 3
2.2 Synthetic Environments for Modeling and Simulation . . . 4
2.2.1 Missile Simulation using Synthetic Environment Generation . . . 4
2.2.2 Distributing Synthetic Environments . . . 5

3 Fundamentals and Theory **7**

3.1 Employed Software Architecture Styles . . . 7
3.1.1 Micro-Kernel Architecture . . . 7
3.1.2 Service-Oriented Architecture . . . 8
3.1.3 Event-Driven Architecture . . . 9
3.1.4 Architecture by Example: Robotic Operating System (ROS) . . . 11
3.2 Required understanding of Computer Graphics . . . 12
3.2.1 Rendering Taxonomy . . . 12
3.2.2 Comparison of Graphics APIs . . . 13
3.2.3 Scene Graphs . . . 14
3.2.4 Distributed Rendering . . . 14

4 Requirements **18**

4.1 Mission/Business Requirements . . . 18
4.2 Functional System Requirements . . . 18
4.3 Non-Functional System Requirements . . . 19
4.3.1 Quality Requirements . . . 19
4.3.2 Performance Requirements . . . 20
4.3.3 Constraints and Assumptions . . . 21

5 High-level Design **22**

5.1 Hybrid Architecture . . . 22
5.1.1 Plugin-centered Design . . . 22
5.1.2 Service-driven Distribution . . . 23
5.1.3 Event-Driven Synchronization and Communication . . . 25
5.2 Distribution Topologies . . . 25
5.2.1 Coordinated Workers . . . 25
5.2.2 Grid Computing . . . 26

6 Core Design and Implementation 29
6.1 Subsystem Overview 29
6.2 Job System 30
6.3 Event Subsystem 33
6.4 Networking 34
6.5 Property Management 37
6.6 Service Infrastructure 37
6.7 Scene Graph and Management 40
6.8 Graphics and Rendering 41
6.9 Plugin Management 43

7 Evaluation by Example: Tile-based Rendering Plugin 45
7.1 Plugin Design and Implementation 45
7.2 Operation 47
7.3 Performance Evaluation 49

8 Conclusion 51
8.1 Summary 51
8.2 Future Work 51

References 53

List of Abbreviations

ABI Application Binary Interface. 44
API Application Programming Interface. 13, 23, 29, 33, 40, 41, 43, 45

CDB Common Data-Base. 27, 52

DDS Data Distribution Service. 11, 35, 51

ESB Enterprise Service Bus. 8

GPU Graphics Processing Unit. 13, 42, 46

HIL Hardware-In-The-Loop. 4, 6, 12, 20

IR infra-red. 3–6, 41

M&S modelling and simulation. 2–4, 18, 28
MDS Mathematically Digital Simulation. 4, 12
MOM Message-Oriented Middleware. 8

OS operating system. 11–13, 20, 21, 30

P2P Peer-to-Peer. 25–27, 39, 51

ROS Robotic Operating System. 11, 12, 22, 24, 35

SOA Service-Oriented Architecture. 8, 9, 11, 17, 24, 25

UAS Unmanned Autonomous System. 1–5, 12, 17, 26

VSG Vulkan Scene Graph. 40, 41

List of Figures

2.1 Components of Closed-loop simulations, as shown in [14, p. 11] 3
2.2 Hardware-In-The-Loop (HIL) simulations for the Taurus KEPD cruise missile. Generated infra-red images are fed into the seeker hardware using a heat projector. 4
2.3 Synthetically generated infra-red images for validating missiles equipped with infra-red seekers using closed-loop simulations and synthetic environments. 5
2.4 Dome projector of the WTD-81 (German Defense Technology Center for Information Technology and Electronics) located in Greding (Germany) from the outside (left) and the inside (right). 6
3.1 Components and structure of a micro-kernel architecture [23, p. 150] . . . 7
3.2 Broker topology of the event-driven architecture [16, p. 44] 9
3.3 Mediator topology of the event-driven architecture [16, p. 43] 10
3.4 Object-based distributed rendering: Agents manage and render different objects of the same synthetic environment. 16
3.5 Tile-based distributed rendering: Tiles are supplied by agents and assembled to the final image. 16
5.1 A single process running the system core and its loaded plugins. 22
5.2 Service Communication Interface supports a range of communication protocols and hides their implementations from both service-caller and provider. 23
5.3 Newly loaded plugin registers a service on its node. The core informs another one about its availability and calls the service. 24
5.4 A central *coordinator* node controls its *workers* to distribute workload. . . 26
5.5 Nodes collaborating in a grid computing scenario. Every node acts both as a client and a server. 27
6.1 Modules of the core system 29
6.2 Plugins and subsystems can schedule jobs for concurrent execution. 30
6.3 Software component design of the job system module. 31
6.4 Software component design of the event system. 33
6.5 Comparison of transfer times of generated frames in different sizes based on their encoding. 35
6.6 Software components of the networking subsystem (blue) and their websocket implementation (orange) using the Boost library. 36
6.7 Software Component Design of the property management subsystem. . . . 38
6.8 Software component design of the service infrastructure. This implementation provides both a service registry capable of remote service calls and an offline alternative (orange). Components needed for other subsystems, like implementations of message consumers for the networking subsystem, were omitted in this diagram. 39
6.9 Theoretical software component design of the scene subsystem. 40
6.10 Practical software component design of the scene subsystem omitting the interface. 41
6.11 Software component design of the graphics subsystem. 42
6.12 Buffers used for capturing rendered frames for offscreen rendering. 43
6.13 Software component design of the plugin management subsystem. 44

7.1 Tiled rendering plugin acts as coordinator and central composition agent, requesting strips of the same frame from different worker nodes. 45
7.2 Subdivision of the current view's frustum for three tiles. 46
7.3 Software component design of the tiled rendering plugin. 47
7.4 Tile-based rendered frame displayed on start-up and when connecting further nodes. 48
7.5 Tile-based rendered frame displayed by the coordinator employing varying counts of workers and tiles. 49
7.6 Frame-rates of different counts of worker nodes for increasing frame sizes using the tile-based rendering plugin. 49

List of Listings

1 Usage of the Job System and its API 32
2 Service request using callers 40
3 Plugin implementation that can be exported to a dynamic library and loaded by the core at runtime. 44
4 Bash command for starting the Coordinator node using the core's CLI. . . 47
5 Bash command for starting a Worker node using the core's CLI. 48

List of Tables

2 Mission/business requirements 18
3 Functional system requirements 19
4 Subset or prioritized quality requirements 20
5 Performance requirements 20
6 System Assumptions 21
7 System constraints 21

1 Introduction

1.1 Motivation

Hardware and software components of Unmanned Autonomous System (UAS), like self-driving cars or robots, are continuously gaining complexity due to increasing demands in their operational fields. With more elaborate perception algorithms, behavioral patterns, and increasing autonomy, they need to be carefully tested and evaluated. In the early development stages of an UAS, doing so in real-world scenarios can be risky and inefficient. For obvious reasons, self-driving cars, which are still unfinished or unsafe, can't be tested in real-life traffic. Simulations and virtual 3d environments pose a safer, cheaper, and more reproducible method for analyzing or prototyping new algorithms or behaviors for such systems. Because the reliability of such simulations is linked to the realism of its employed virtual worlds, their requirements also increase. Domain-specific open-source software solutions exist for this purpose: CARLA and LGSVL employ traffic simulations and are popular tools for evaluating and training autonomously driving vehicles. Another tool called Gazebo is heavily used in robotics for the same purpose [11].

Due to its technical background, missile engineering faces similar challenges. Defense systems, such as self-navigating cruise missiles or drones equipped with swarm intelligence, are constantly increasing the need for validation and evaluation. Since real-world missile tests are very costly, simulations in virtual worlds are indispensable. However, there is little to no publically accessible market for simulation tools for such a niche as guided missiles yet. Therefore, MBDA started developing its own tools for validating the perception of developed seekers, algorithms, and the behavior of systems in various mission scenarios. Their software architecture is usually monolithic and runs only on a single machine. Distributed computing could be helpful in enabling virtual environments, also known in this field as synthetic environments, with higher fidelity, accuracy, and realism. However, this approach requires a new, non-monolithic software architecture that is more scalable, extendable, and adaptable to future use cases. This work aims to offer such an architecture for distributable synthetic environment generation.

1.2 Research Questions

The aim of this thesis is to evaluate ways of improving future missile simulations at MBDA by optimizing the process of generating synthetic sensory data. It is largely based on the hypothesis that distributed computing might achieve this goal. Conditions and constraints must be determined and taken into account to apply this concept successfully. This step is essential to narrow down meaningful use cases for this approach. Furthermore, a software architecture that enables such distributed simulations has to be proposed and implemented. Current tools at MBDA focus on real-time image generation. Thus, distributed real-time rendering for generating sensor images must be clarified and evaluated. Therefore, this work's research questions can be summarized as follows:

- How is an extensible software architecture enabling distributed generation of synthetic environments structured, and how can it be applied?
- What are constructive use cases for this distributed approach, and what criteria are essential? Can every simulation use case be improved by distributing its workload?
- Is distributed real-time image generation such a use case?

2 Technical Background

Before diving into the theoretical foundation of this work, a technical background and understanding of synthetic environments is required. This chapter will outline their use cases and applications in missile engineering and provide background information on the purpose of this work.

2.1 Introduction to System Validation in Missile Engineering

2.1.1 Missiles as Unmanned Autonomous Systems

Although human personnel can be part of the overall system, UAS accomplish their mission goals with different levels of autonomy, ranging from remote control to fully automated decision-making. They can adapt to changing operational fields and must be able to operate under uncertain or extreme environmental conditions [17].

Due to their fire-and-forget mechanic, missiles for military and defense applications are such UASs. After its operator locks onto a target and fires the missile system, it has to function independently. During its mission, the missile must navigate its airborne environment, make decisions, and sometimes adapt to changing tactical conditions. It must maintain these capabilities under challenging environmental conditions, such as rough weather, and its behavior must stay stable even in extended or unexpected scenarios. This requirement poses a challenge to engineers during missile development.

2.1.2 Validation Difficulties of Missile Systems in the Real World

UASs must be capable of operating under uncertain environmental conditions without failing their mission or task. This requirement demands extensive testing and validation of the system's behavior in different extended scenarios to establish operational trust. In the missile domain, these scenarios range from challenging weather conditions that reduce sensor quality to particular combat scenarios in which the system must still be able to operate successfully. Engineers must validate, analyze, and challenge the developed missile's perception, decision-making, and guiding algorithms in various testing scenarios to guarantee a stable product and rule out possible side effects [9, p. 18-22].

Real-world testing of missiles requires extensive resources:

- Vast testing ranges of different biomes and vegetation, like deserts or forests.
- Employment of personnel for operating missile platforms, like aircraft or control centers.
- Realistic models for ground, naval, or airborne targets.
- Setup of complex combat scenarios in diverse field conditions for various testing cases.

Therefore, real-world testing is very costly in both money and time, increasing the missile's price and its time-to-market. Furthermore, it can only be applied at the end of the UAS's development life-cycle when a physically constructed prototype exists. Both reasons render frequent real-world testing during the missile's development life-cycle unpractical. Therefore, modelling and simulation (M&S) techniques enable testing of UASs even during early phases in the development life-cycle [9, p. 18-22].

2.1.3 Modelling and Simulation in the Autonomous Systems Domain

M&S techniques do not always require physical prototypes or hardware for testing UASs. Engineers can study and optimize the behavior of their algorithms and sub-systems or evaluate different design choices on a simulated model in the pre-deployment phase using M&S [6, p. 617]. It renders validating and testing missile systems in diverse operational field conditions more affordable and frequently repeatable.

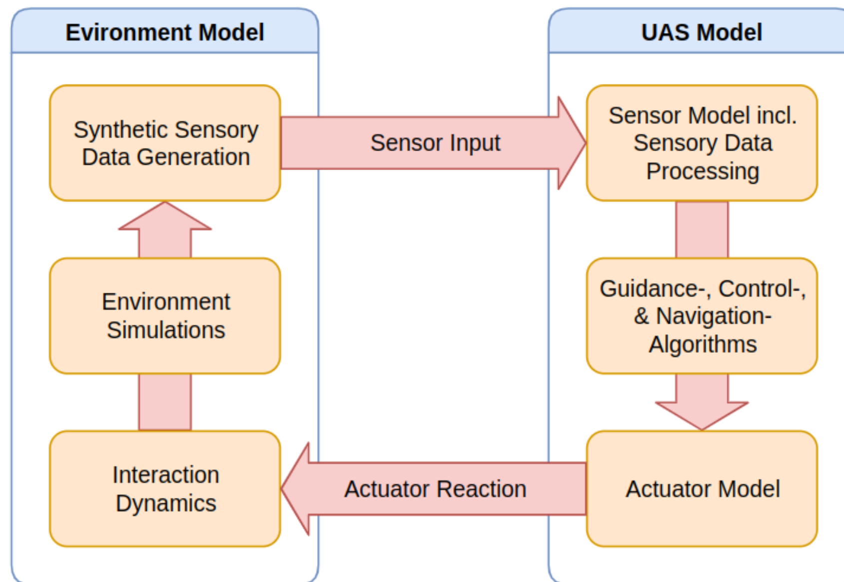


Figure 2.1: Components of Closed-loop simulations, as shown in [14, p. 11]

Closed-loop simulations enable testing, validating, and analyzing missile systems, robots, or other autonomous systems. As depicted in figure 2.1, the closed loop consist of two major components:

- The UAS model: It contains sensors, actuators, and algorithms for input processing, guidance, decision-making, and various other functions based on sensory data from its environment.
- The environment model in which the UAS operates generates sensory stimuli for the UAS model and simulates interactions caused by actuators.

A closed-loop simulation models the interaction between the UAS and its environment, like a self-driving car participating in road traffic or a cruise missile navigating toward its mission target. The environment model generates synthetic sensory data for the UAS model, which computes actions and decisions based on it. The UAS model executes these actions using its actuators, updating the environment model. This cycle of generating and processing stimuli repeats, creating a closed loop. When simulating a missile equipped with an infra-red (IR) sensor, the environment model will generate synthetic IR images from the view of the missile's camera. Based on its image processing and guidance algorithms, the missile's model adjusts its current trajectory using its fins [14, p. 11].

A closed-loop simulation in which the environment and the UAS models are purely virtual, using M&S, is called a Mathematically Digital Simulation (MDS). They enable testing without a physical prototype or its hardware. Engineers can use MDS to validate and analyze their algorithms and sub-systems, like a cruise missile's IR image recognition or navigation, early in development [14, p. 11].

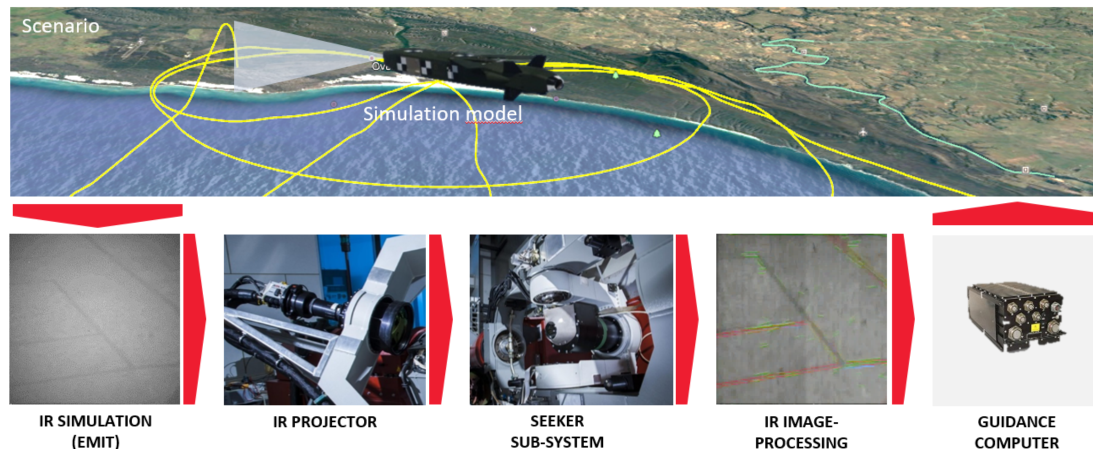


Figure 2.2: Hardware-In-The-Loop (HIL) simulations for the Taurus KEPD cruise missile. Generated infra-red images are fed into the seeker hardware using a heat projector.

When hardware components, such as sensors, seekers, actuators, or onboard computers, are available, a Hardware-In-The-Loop (HIL) simulation can replace the purely digital MDS. The UAS's hardware replaces parts of its digital model in the simulation loop, allowing for more accurate validations and stress tests [14, p. 12]. The closed loop of HIL simulations of the Taurus KEPD Cruise missile is depicted in figure 2.2. Synthetic IR images generated by the environment model are fed into the seeker hardware using a heat-projector. The seeker's image-processing and the missile's guidance computer react to their sensory inputs using its virtual fins, acting as feedback to the simulated environment model.

While an MDS is not required to run in real-time because the simulation loop controls the UAS's update rate, a HIL simulation is. The environment model must adapt to the UAS's hardware, which operates at its own processing rate. This real-time constraint in HIL simulations challenges complex environment models with high fidelity [14, p. 14]. Producing synthetic sensory data for IR or radar seekers, primarily used in missile systems, requires computationally expensive calculations that are challenging to run synchronously to the hardware's relatively high processing rate.

2.2 Synthetic Environments for Modeling and Simulation

2.2.1 Missile Simulation using Synthetic Environment Generation

Synthetic environments are digital environment models used in MDS and HIL simulations. They represent the natural environment with high realism at a given geographical location. As shown in figure 2.3, they can contain:

- Both man-made and natural structures
- Terrains
- People and animals
- Dynamic models, such as weather models or others for simulating various physical processes

Due to their versatile capability of modeling, simulating, and visualizing environmental processes and elements, synthetic environments are not only used for validating the behavior of UASs. Applications range from traffic simulations, video games, and manufacturing to flight simulations [24, p. 2].

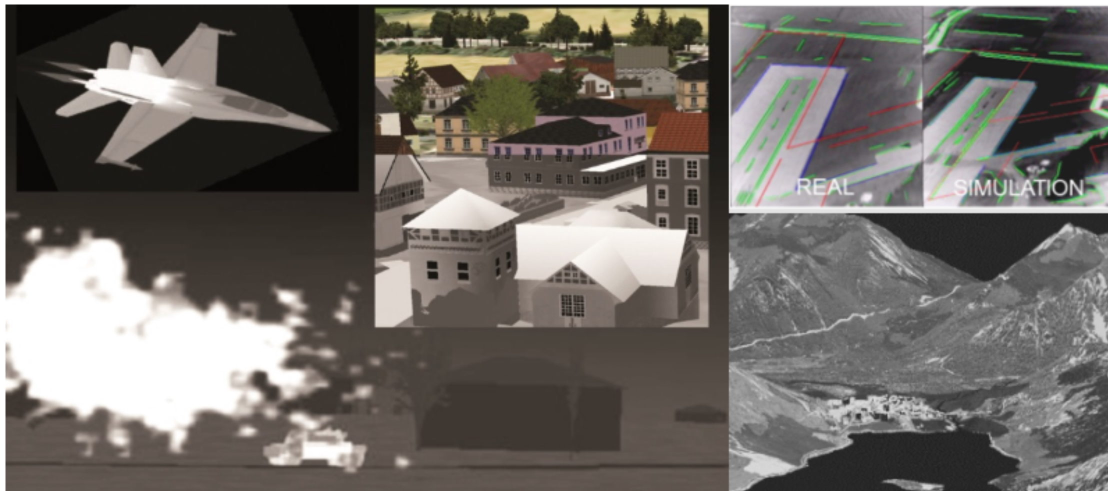


Figure 2.3: Synthetically generated infra-red images for validating missiles equipped with infra-red seekers using closed-loop simulations and synthetic environments.

MBDA, for instance, uses synthetic environments to generate IR images, as shown in figure 2.3, for validating missiles equipped with IR seekers. It allows observation of the missile's behavior in various synthetic environments, i.e., representing different tactical scenarios.

To ensure reliable validation results, synthetic environments must replicate reality as closely as possible to minimize the difference between simulations and actual field tests in the real world. Therefore, their development must aim at the highest possible *fidelity*, being the degree of accuracy of a model compared to its real-world counterpart [32, p. 9].

2.2.2 Distributing Synthetic Environments

Often, single computers cannot provide the fidelity, scale, and realism required for complex synthetic environments [29, p. 19]. For instance, generating synthetic IR sensory data requires costly thermal calculations to determine the emitted and reflected heat radiation of every surface in the scene. For flights of short duration, these calculations can be pre-computed without reducing the synthetic environment's fidelity considerably. On the other hand, simulations of longer duration can require thermal re-calculations at runtime. Carrying out all computations on a single machine will become inefficient in

such a scenario.

Distributing the synthetic environment to multiple machines can be a practical approach to achieve larger scale and higher complexity in simulations through workload-sharing [6, p. 618]. However, distribution is only a valid solution for some performance problems. It shifts performance bottlenecks from the processing units to their communication network. No advantage is gained if the created communication bottlenecks are worse than the old ones. This correlation implies that synthetic environment simulations that are perfectly running on a single machine do not benefit from distribution but rather suffer from it [32, p. 12]. Nevertheless, this approach can still benefit some use cases and applications.

Figure 2.4: Dome projector of the WTD-81 (German Defense Technology Center for Information Technology and Electronics) located in Greding (Germany) from the outside (left) and the inside (right).

Another possible application of distributable synthetic environments is the projector dome in Greding shown in figure 2.4. The *German Defense Technology Center for Information Technology and Electronics* (WTD-81) conducts HIL target simulations in the optical and IR spectral range for various systems. The dome is equipped with several projectors that illuminate the inside with synthetic images. It serves as an environment for various HIL simulations of optical and optronic components and systems [34]. Instead of using a single computer to supply all projectors with synthetic images, the image generation process can be distributed. Each projector's image can be generated on a dedicated computer for more complex synthetic environments. To implement this architecture, the synthetic environment itself must be distributable.

3 Fundamentals and Theory

3.1 Employed Software Architecture Styles

The following section will introduce software architecture styles that greatly influenced the final hybrid architecture of this work.

3.1.1 Micro-Kernel Architecture

Micro-kernel architectures, or plugin architectures, are used by many commercial and open-source software products, like web browsers or software development tools [23, p. 158]. They are also widely applied in the domain of 3D software: Game Engines like *Unity* [31] or 3D modeling suites such as *Blender* [30] employ this architecture to ensure extensibility and maintainability of their product.

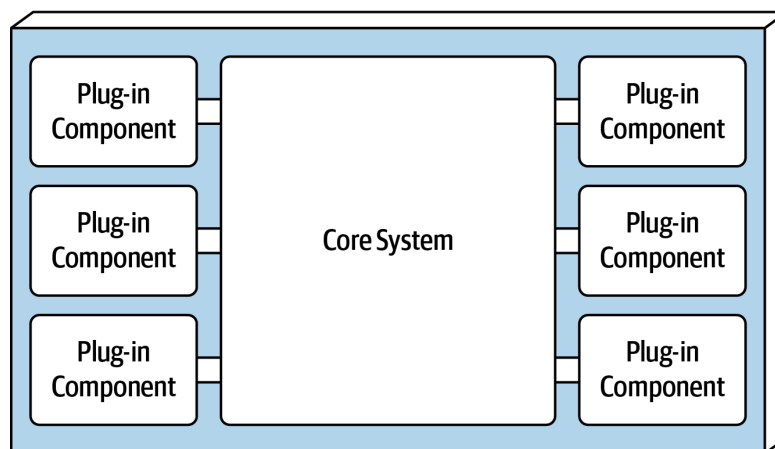


Figure 3.1: Components and structure of a micro-kernel architecture [23, p. 150]

As depicted in figure 3.1, this architecture consists of two basic component types:

- A single **core system** containing the minimum functionality required for execution. It only provides essential services and the fundamental structure of the overall application, like a framework.
- Multiple **plugin components** providing additional functionality and services to the core. They are built upon its foundation and contribute specific services and features to the end-user's application.

  Depending on their implementation, plugins can be created locally or remotely. Remote plugin components can be distributed to other machines and accessed using network protocols. Local plugins, on the other hand, are usually loaded into the core as shared libraries or scripts.

The main thought behind such an architecture is to move the system's complexity from the core to its plugins. Therefore, it is split into smaller, more manageable chunks that can be maintained, tested, and deployed independently. This division inherently isolates application features into loosely coupled components, increasing the system's maintainability and possibly reducing development times. End-users can leverage this architecture by loading plugins provided by the core-developer team, independent third parties, or

themselves and tailoring the application to their individual needs. Third-party contributions can ease the burden on small core-developer teams. Users can develop their own plugins for specific use cases and do not depend on the core-developers' support. Additionally, unloading unneeded features and adapting to specific use cases can improve the system's overall performance [23, p. 149-161].

In conclusion, this architecture is suited for software that must remain adaptable and extensible by various parties during its lifetime due to uncertain or rapidly changing use cases.

3.1.2 Service-Oriented Architecture

As its name implies, the Service-Oriented Architecture (SOA) is centered on loosely coupled components providing services to each other. These services can be requested using a shared communication protocol and contracts. Distributed systems commonly use this architecture to distribute their workload onto several machines acting as service providers [16, p. 351].

A service in the SOA has four properties:

- It represents a business activity with a defined outcome,
- is self-contained,
- is opaque to its users,
- and may be composed of other services.

Similar to micro-kernel plugins (⇒ Section 3.1.1), Services can be maintained, developed, and tested separately through their self-contained, opaque, and reusable nature. This approach splits the overall complexity of the system into loosely coupled components. Engineers can focus on single business activities by working on isolated services rather than considering all possible tightly coupled interactions within a monolith. This can reduce development times and help to tame the overall system's complexity [16, p. 358f].

Services can be located and called both locally and remotely, rendering this architecture suitable for distributed software projects. The SOA does not specify the service's implementation, but some layer providing communication between parties for calling and providing services is required. Furthermore, routing, translating, or validating service requests and responses is often necessary. This can be implemented using the following technologies:

- An Enterprise Service Bus (ESB) is one of the oldest approaches for implementing SOA. As a separate, event-driven hardware or software component, it allows heterogeneous services to collaborate via its bus infrastructure [16, p. 351-353].
- Web-services provide communication using World-Wide-Web protocols, such as HTTP. Although open standards are preferred, implementations can also use proprietary solutions [16, p. 353f].
- Using a Message-Oriented Middleware (MOM) that is usually a part of an ESB's implementation directly. It utilizes message queues, streaming protocols, and brokers to communicate messages between recipients [16, p. 354-356].

This communication layer introduces costs regarding resource consumption and the time required to establish it. In practice, SOA also lacks uniform testing frameworks and homogenous tooling. In some cases, updating the services' platform can lead to high maintenance costs because developer teams have to adapt their services to changes [16, p. 360].

3.1.3 Event-Driven Architecture

Another architecture achieving extensibility and loose coupling is the event-driven architecture. It revolves around decoupled components firing and processing events propagated through so-called *channels*. The event processors are responsible for reacting to them and triggering appropriate actions or consecutive events asynchronously. A chain of events is called a *workflow*. Although this architecture shares many similarities with the SOA (⇒ Section 3.1.2), it differs by only reacting to events instead of responding to them. In comparison to service requests, events are *fire-and-forget* in their nature [23, p. 179, 204f], [16, p. 42-46].

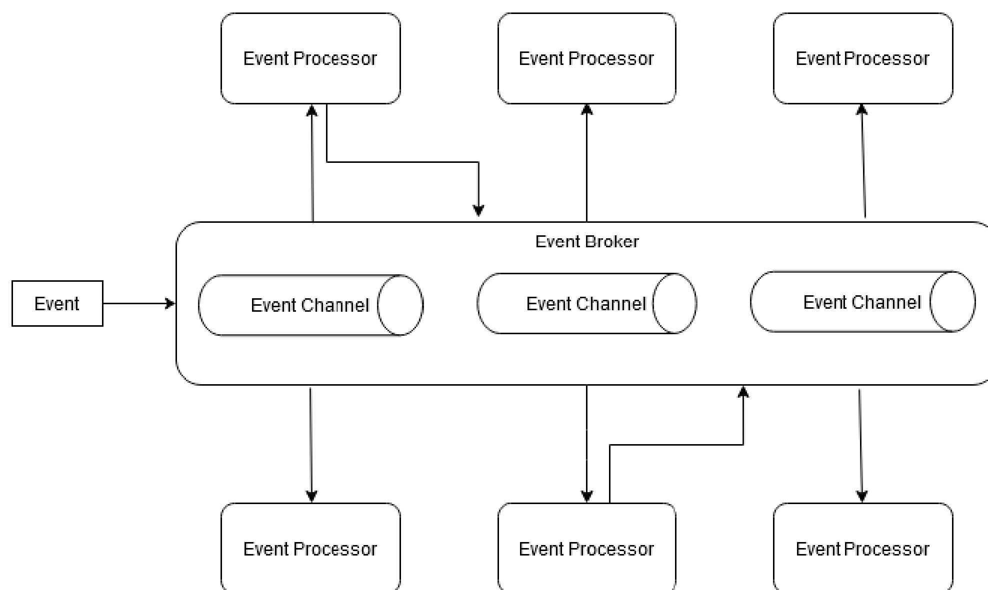


Figure 3.2: Broker topology of the event-driven architecture [16, p. 44]

This architecture differentiates two topologies. The broker topology, depicted in figure 3.2, employs the name-giving central component responsible for passing events to event processors. The broker maintains channels containing events of a certain type, also called a *topic*. Processors register their interest in a specific topic by subscribing to its designated channel. The broker broadcasts fired events through their topic's channel directly to event processors. They are responsible for handling their occurrence by triggering appropriate actions or subsequent events. The advantages of this topology are highly decoupled components because event producers and processors do not need to know about each other. It also provides scalability and high extensibility. New components can seamlessly integrate and contribute to the system by processing and triggering events. On the other hand, error handling and workflow control are difficult using the broker topology.

Event chains can be very dynamic and difficult to predict, test, and debug. For instance, in case of an event processor's crash, the system is fault-tolerant, but workflows can get stuck due to incomplete event chains. The broker cannot resolve such incidents or coordinate the workflow in any way [23, p. 180f]. For the same reason, it cannot detect cyclic event chains and prevent them from invoking themselves indefinitely. Event processors that cause chains of events that inevitably lead to their own invocation can cause faulty behavior or even crash the system.

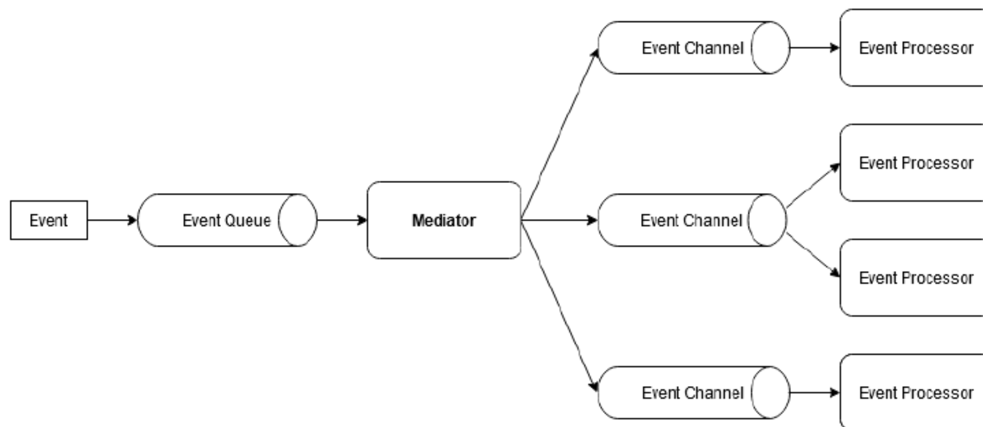


Figure 3.3: Mediator topology of the event-driven architecture [16, p. 43]

The mediator topology, as shown in figure 3.3, addresses some weaknesses of its broker counterpart. In contrast to a broker, the mediator has knowledge of the system's workflows, is responsible for coordinating them, and keeps their overall state. Events do not uncontrollably spread through the system but are processed in a controlled and orderly manner. Event processors respond to the mediator after completion and report if the event has been handled successfully to avoid incomplete workflows and allow error handling. They cannot broadcast events to other processors but only to the mediator for coordinated execution. The main intent behind a mediator is to render event processing more deterministic and enable better error handling, recovery, and control capabilities than a broker can offer. However, it comes at the cost of higher coupling and coordination overhead. Additionally, the mediator can only work with well-defined and modeled workflows [23, p. 185-195].

Although both are event-driven approaches, their conceptual understanding of events differs. The broker topology thinks of an event as some occurrence or happening in the system that needs to be handled. Its mediator counterpart treats them like commands or actions that an event processor must execute. The decision on which topology should be used depends on what the event should represent in the underlying use case. Furthermore, the mediator's strengths revolve around well-defined workflows, like business transactions that require execution in a more orderly step-by-step manner. The choice between both variants is basically a trade-off between the mediator's control and error-handling capabilities and the broker's performance and scalability [23, p. 195].

3.1.4 Architecture by Example: Robotic Operating System (ROS)

All previously discussed software architecture styles can be combined seamlessly to form practical synergies in complex software projects. An example is the Robotic Operating System (ROS) project. It employs a hybrid architecture, providing a framework for distributed robotic applications. Despite its name, it is not an operating system (OS) in the traditional sense but a communication layer for heterogeneous computing clusters. High complexity, daunting amounts of code, and extensive integration efforts are impeding challenges in robotic software engineering. ROS aims at solving these by decomposing complex software into smaller, more manageable, and reusable components that can be distributed across different computing devices [12].

ROS software architecture introduces the following concepts and terms [20], [12]:

- **Nodes** are processes performing computations. They are connected in a peer-to-peer manner using a communication protocol. ROS 1 and ROS 2 utilize TCP/IP and the Data Distribution Service (DDS) protocol respectively.
- **Messages** pass strictly typed data structures between nodes. The interface definition language (IDL) can be used to define message types and their contents.
- **Topics** implement a publish-subscribe method for passing messages asynchronously between nodes.
- **Services** are analog to web services and provide synchronous request-response message passing. Their name references them, comparable to a URL.

The presence and influence of previously discussed architectural styles are undoubtedly noticeable. ROS utilizes a micro-kernel architecture (⇒ Section 3.1.1). The core provides various generic services like its communication layer and domain-specific functionality for its distributed tools, which act as plugins in the reference architecture [20]. The SOA (⇒ Section 3.1.2) enables nodes to offer and consume distributed services inside a ROS cluster. Its publish-subscribe mechanism resembles the broker topology of the event-driven architecture (⇒ Section 3.1.3). Therefore, ROS employs a hybrid architecture based on the previously discussed styles to enforce its design principles: Scalability, modularity, distributivity, and asynchrony in complex robotic projects [12].

The combination of these three styles creates a symbiosis of their strengths that the framework profits from:

- The cooperation of the micro-kernel and SOA facilitates the overall development complexity of robotic software by splitting it into smaller chunks that can be dealt with separately. It enforces isolation and loose coupling, increasing maintainability and possibly reducing development times. Above all, they provide extensibility and versatility for various robots and their operational fields.
- The SOA encourages distribution of isolated components. Some robots utilize off-board computation for heavy calculations that the on-board processor struggles with [20].

- The interplay of the micro-kernel and event-driven architectures enables tools to integrate into the system seamlessly by contributing and processing events. Furthermore, it allows for global state updates across distributed software components.

These architectural strengths of ROS do not only apply to robotic use cases. As will be demonstrated in the following sections, such a hybrid architecture is beneficial for the purpose of this work as well.

3.2 Required understanding of Computer Graphics

Synthetic environments often utilize image-generation techniques to synthesize realistic sensory data. Therefore, some basic knowledge in the field of computer graphics is required. This section will briefly introduce relevant topics for generating images of Synthetic Environments.

3.2.1 Rendering Taxonomy

Although it is out of this work's scope to present a deep dive into computer graphics and rendering techniques, the primary taxonomy and basic concepts must be introduced to understand the image generation part of this work.

Rendering processes come in many different shapes and use cases. At first, it is essential to understand the fundamental trade-off between the generated frames' quality and their throughput. **Realtime rendering** focuses on the latter and is suited for interactive applications like games or training simulations. A HIL Simulation also requires synthetic image generation in real-time adapted to the hardware's update rate. Its counterpart is **non-realtime rendering**, which prioritizes quality above throughput. It allows for more costly and time-consuming computations and rendering techniques, like raytracing or radiosity calculations, but does not guarantee stable or intractable frame rates. An MDS does not require real-time image generation and can benefit from higher-quality rendering results [15, p. 25f].

Rendering processes also differ in their so-called render targets and how their image results are used. **Onscreen rendering** usually stores its outcome in a frame buffer accessed by a window of the host OS to display it on screen. However, not every use case of computer graphics aims to show rendered outcomes on a screen. Some of them include:

- Image generation for non-human consumers which are not supplied via screens, like digital UAS models in MDS or sensor hardware in HIL simulations employing special heat-projectors.
- Remote image generation that must transfer its results via some network or bus to its target.
- Rendering on embedded systems or OSs that do not support graphical user interfaces or screens.

These use cases utilize **offscreen rendering** that targets buffers instead. Their results can be stored on disk, transmitted over a network, or shared across processes. However, this approach often introduces additional overhead compared to its onscreen

counterpart. It involves performance-critical operations, such as passing buffers between Graphics Processing Unit (GPU) and host memory, applying compression, or performing transportation over networks and busses [5].

3.2.2 Comparison of Graphics APIs

As already shown, this work requires 3D computer graphics for synthetic sensory image generation. Graphics Application Programming Interface (API)s allow the integration of GPU hardware and provide access to its functionality. OpenGL and Vulkan are the two major graphics APIs that qualify for this work's purpose. DirectX is not considered because it is only supported by the Windows OS and therefore conflicts with cross-platform requirements (⇒ Section 4.3.1) [15, p. 42].

OpenGL 1.0 was released in 1992 and has since become the most widely used graphics API of 2019. Purely specialized in computer graphics applications and rendering, it is actively maintained and governed by the Khronos Group. Known for its high compatibility, OpenGL operates independently from its underlying graphics hardware and platform. It supports most major OS, like Windows and many UNIX derivatives. Further implementations for web browsers and embedded systems, called WebGL and OpenGL ES, respectively, are also available. OpenGL delivers a powerful programmable 2D and 3D graphics pipeline using so-called shaders, programs executed on the GPU itself. The OpenGL API was designed as a central state machine. Buffers containing geometry data, textures, or other stateful attributes are bound to the current OpenGL context, embodied by the state machine, and thereupon rendered. However, this design does not leverage multi-core processing and introduces some bottlenecks, leading to Vulkan's new API design [15, p. 39-46].

In 2016, the Khronos Group released Vulkan, a relatively new graphics API compared to OpenGL. It is not solely specialized in computer graphics but can be employed seamlessly in various other parallel computing applications involving the GPU. Vulkan's distinct design aims for a more adaptable and multi-threading-friendly API. It strips away much of the graphics driver's overhead and gives that control directly to the developer. Although its minimalism renders Vulkan more flexible and performant in specific use cases, additional implementation efforts are required to substitute missing procedures. It also avoids a central state machine and employs more parallel data structures. Its computational model revolves around individual GPU commands that can be recorded in command buffers and pushed into a collection of queues for processing on the GPU device. This design aims to avoid driver bottlenecks on the host side and to leverage its multi-core processors [1, p. 40f], [27, p. 2f], [15, p. 75-80].

Although Vulkan avoids the bottleneck and overhead of the OpenGL driver, it does not automatically guarantee better or faster rendering results. In certain scenarios, Vulkan's flexibility can force some extra grains of performance out of the GPU, but that's not always the case. Furthermore, programmers must implement procedures that other APIs, like OpenGL, take care of automatically, like memory management or command synchronization. Also, considering that OpenGL is still actively maintained by the Khronos Group, these APIs are not replacing each other; they co-exist.

3.2.3 Scene Graphs

The following section is based on [8, p. 693-697] and [15, p. 574f].

To leverage the power of graphics APIs in 3D environments, they must be represented by a data structure that allows the application of various algorithms and rendering operations. A scene graph is such a structure. It hierarchically manages geometry, attributes, and other scenery information for efficient rendering. Usually, it is implemented using a tree structure. It employs various types of nodes. Leaf nodes usually contain geometry, while internal nodes allow for more elaborate organization and behavior, like Level-Of-Detail (LOD) switching or transformation. Algorithms are applied by traversing the scene graph from its root node down to its leaves and performing operations on each passed node. One example is the culling process, in which the three-dimensional space is partitioned to discard regions invisible to the camera's frustum. Using a tree-like scene graph, this algorithm can be optimized from $O(n)$ to $O(\log n)$, saving rendering time. Besides altering its structure, traversals can also query a scene graph. Collision detection and raytracing, for instance, use intersection queries to find possible collision points between geometry nodes.

A scene graph's structure can be manually created or generated procedurally using some subdivision algorithm. An octree, for instance, subdivides the scene's three-dimensional space recursively into quadrants. Each recursive subdivision is added as a node to the graph containing further subdivisions as its children. This procedure aims to distribute the amount of renderable primitives among leaf nodes equally. It results in more subdivisions in geometry-dense quadrants. The frustum culling algorithm can be further optimized using this scene graph type.

Other types of procedurally built Scene Graphs include:

- Bounding-Sphere Trees (BST): Same concept as applied in octrees, but using spherical quadrants and subdivisions.
- Binary Space Partitioning (BSP): Recursive subdivision using a plane results in a binary search tree. This is heavily used in collision detection and constructive solid geometry.

3.2.4 Distributed Rendering

The main objective of distributed rendering is the visualization of synthetic environments of higher complexity and fidelity than a centralized rendering process on a single computer can deliver [29, p. 19], [21], [7]. A selected distribution strategy splits the rendering process into multiple tasks and assigns them to different machines responsible for execution. These so-called agents are running on different but interconnected host machines communicating through a network [6, p. 618]. Therefore, multiple computing resources split and carry out the overall rendering workload instead of a single powerful one. This can reduce execution times and hardware costs [21].

However, the load-sharing argument only wins in certain scenarios. Physically distributing computational tasks means moving the bottleneck from the processors to their communication network. It is essential to apply loose coupling to limit the message traffic

between agents. If the relocated communication bottleneck is worse than the old one, no advantage is gained [32, p. 12]. This relation limits the set of use cases in which distribution yields benefits and fulfills its purpose. If a single machine effortlessly carries out a computational task, i.e., the rendering process, there is no advantage gained by distributing it onto multiple hosts. Therefore, distributed rendering targets synthetic environments of the highest fidelity or very costly rendering techniques a single machine struggles with. A good indicator for expedient distribution is when the task's nature is easily decomposable [32, p. 12f].

Although many strategies can be applied to distributed rendering, three abstract steps can be identified. They always involve a central agent splitting the workload, delegating it to other agents, and collecting the partial outcomes to assemble the final result [7]:

1. **Task Distribution**: A central agent splits the overall rendering process into smaller tasks using a strategy selected for the underlying use case. These tasks are then assigned to agents. Popular strategies and distribution approaches are discussed further in this section.

2. **Distributed Rendering**: Agents execute their rendering tasks. This can either be initiated by a request of the central agent or automatically based on a fixed time interval, depending on the chosen approach.

3. **Result Composition**: The produced fragments are then returned to the central agent responsible for assembling the final rendering result.

As already mentioned, there are multiple strategies for distributing the rendering workload. The two main approaches applicable in real-time rendering are the distribution of the synthetic environment at an object level of the render surface, also known as tiling.

When distributing the synthetic environment, the central agent assigns others a distinct subset of its objects. Each agent is responsible for managing their state and rendering them. As depicted in figure 3.4, one agent might render armed forces, like aircraft, while another generates realistic clouds. Each image and its z-buffer, which holds depth information of the rendered outcome, is then passed to the central agent. It assembles the final result containing all objects by overlaying them in the correct order, determined by their depth information. Using this approach, only the environment's global time must be synchronized between agents, not each object's states. Furthermore, this approach integrates well with inherently distributed synthetic environments where, for instance, the behavior and state of armed forces are already simulated on dedicated computers [21].

When tiling a target image, the central agent subdivides its surface area into smaller tile fragments and distributes them among agents. As depicted in figure 3.5, each agent renders the identical synthetic environment but from a slightly different view angle. The camera's frustum, defining its perspective, must be calculated to arrange the resulting image fragments back together like puzzle pieces. All states of dynamic objects must be synchronized, and all agents need to know the entire synthetic environment because it must not vary across tiles. Otherwise, each tile would display a different state, and they would not fit together [28]. Although this approach may sound more costly than its object-based counterpart, it can increase performance by utilizing its agents more effectively. Some tiles may contain more detailed objects and require more expensive

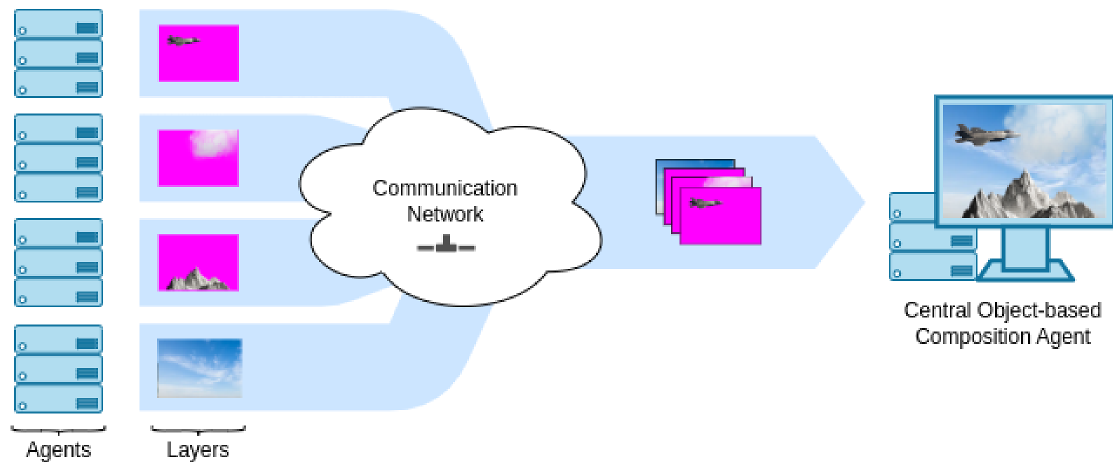


Figure 3.4: Object-based distributed rendering: Agents manage and render different objects of the same synthetic environment.

computation. By dynamically sizing tiles based on performance statistics from previous frames, the overall workload can be distributed more equally [35, p. 2].

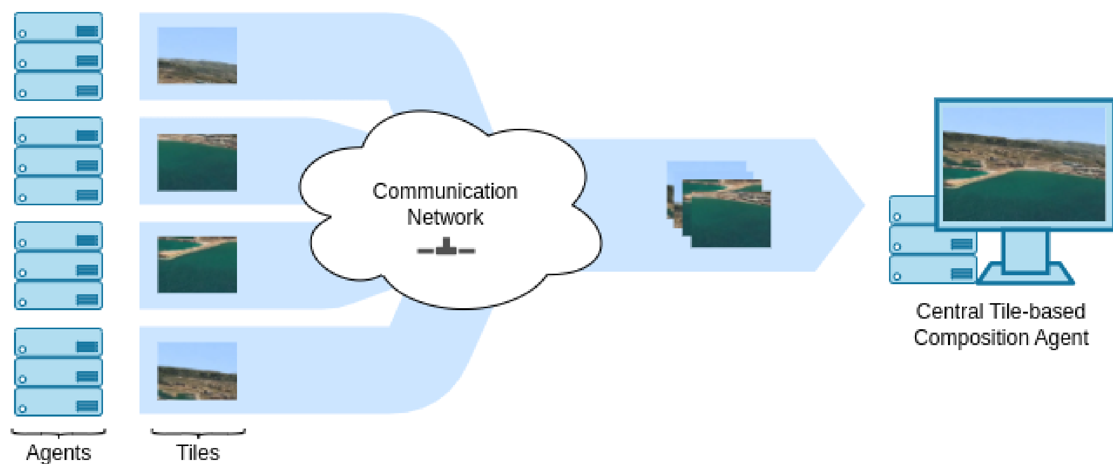


Figure 3.5: Tile-based distributed rendering: Tiles are supplied by agents and assembled to the final image.

Further approaches, which are worth mentioning but do not apply to this work, include:

- **Distributing samples of the same frame**: Some rendering techniques generate multiple samples for a single frame. A sample is the resulting image of a rendering iteration. In raytracing, for instance, light is simulated by rays bouncing non-deterministically through the scene. In a single sample, this leads to randomized noise scattered across the image. Combining multiple samples reduces this effect. Because an individual sample can be independently rendered from all others, they can be distributed among multiple agents. This approach is not inspected further in this work because it is more suited for non-realtime techniques [28].
- **Distributing different frames of an image sequence**: Looking at real-time

rendering and the closed simulation loop (⇒ Section 2.1.3), it becomes apparent that its frames can only be rendered sequentially. They depend on the UAS model's feedback and, thereby, each other. However, in non-realtime rendering, it is possible to know the contents of future frames beforehand, like in 3D animations. All frames can be theoretically rendered in parallel because they do not depend on each other and can, therefore, be distributed [7].

- **Distributing images of different perspectives**: As mentioned in the projector-dome example (⇒ Section 2.2.2), rendering from different perspectives is a practical application of distributed rendering. Each perspective can be generated independently from the others and qualifies for distribution [10, p. 247].

After rendering tasks have been assigned to agents, two methods, as already mentioned, exist for invoking their execution for each frame. The request-based approach requires a central process to order its agent to render each frame. This approach is comparable to a service request in the SOA. Additional information, like the scene's current state or rendering parameters, can be passed along with this request. In terms of performance, this approach causes further delay due to request transmissions before every frame. On the other hand, including the scene's overall state in the rendering requests reduces synchronization efforts beforehand. Furthermore, consistency between rendered fragments is guaranteed if the same request parameters are sent to all agents. Its counterpart, the polling-based approach, does not require a request. The agents invoke themselves at fixed time intervals and push their rendered frames automatically to buffers of the central composition process. While this approach avoids repeated request transmissions to achieve better performance, it can cause inconsistent results. If the agents' states or rendering intervals are even slightly unsynchronized, they capture the same scene at different points in time or at incompatible states. In this case, the generated image fragments do not fit together. Additionally, when an agent does not deliver its frame on time, the final image may be incomplete or inconsistent after composition. More effort and demanding procedures must be spent on synchronization to avoid these scenarios [21]. Therefore, the choice between the request-based and polling-based approach is a trade-off between consistency and throughput.

4 Requirements

This work considers requirements of different categories [4, p. 139-149]:

- **Mission/business requirements** are high-level goals or objectives that the system must achieve. They hint at the underlying business needs that the result must meet.
- **Functional system requirements** describe the behavior of the system and *what* results it should attain or exhibit.
- **Non-functional system requirements** expose constraints, conditions, and quality measures. It describes *how* the system achieves its functional system requirements and greatly influences the software architecture design.

This work does not consider the so-called **user & stakeholder requirements**. They describe the users' point of view and their inputs/outputs in concrete use cases. These are future requirements bound to a specific application that will emerge after the completion of this work.

4.1 Mission/Business Requirements

The target system's underlying business needs and high-level objectives are formulated in table 2. Although they can't be clearly evaluated or tested due to their high-level nature, they are listed to reveal expectations and the underlying purpose of this work.

ID	High-level Objective
MB1	Shall provide a framework for developing further synthetic image generation tools for M&S in the future.
MB2	Shall reduce maintenance costs and enable faster development cycles of M&S tools.
MB3	Shall adapt to future technologies and use cases that are yet unconsidered or unknown.
MB4	Shall be scalable enough to support future synthetic environments of higher fidelity and complexity.

Table 2: Mission/business requirements

4.2 Functional System Requirements

Functional system requirements are listed in table 3. This work evaluates them by employing automated unit tests in its implementation. Compared to other projects, not many functional requirements are provided for this work because it is merely a framework for satisfying future ones.

ID	Description	Priority
F1	Shall provide an on-screen image generation pipeline that renders to a native window.	High
F2	Shall provide an off-screen image generation pipeline that renders to a buffer.	Medium High
F3	Shall load scenes from the following file formats: osgt, osgb, and obj.	Medium High
F4	Shall provide a scene graph to apply operations and queries at loaded scenes.	High
F5	Shall be accessible through REST requests for remote operation.	Medium

Table 3: Functional system requirements

4.3 Non-Functional System Requirements

Non-functional system requirements can be subdivided into further categories [4, p. 143f].

- **Quality requirements** prioritize a specific subset of the entire spectrum of quality measures. Because they can be mapped to the strengths and weaknesses of architectural styles, their prioritization greatly influences the final architectural design.

- **Constraints and assumptions** define the project's legal, physical, and environmental boundaries to which the system must adhere.

- **Performance requirements** define timing boundaries and deadlines.

4.3.1 Quality Requirements

The sole purpose of quality requirements in this work lies in supplying preferences and priorities for designing the software architecture. Since their evaluation proves difficult because their achievement and impact are difficult to measure, their effectiveness is only demonstrated by example (⇒ Section 7). The full list of quality measures that were considered for prioritization extends to *maintainability*, *availability*, *reliability*, *accessibility*, *visibility*, *testability*, *complexity*, *interchangeability*, *sustainability*, *flexibility*, *portability*, *manufacturability*, *usability*, *learnability*, *error tolerance*, *efficiecy*, *accuracy*, *capacity*, *adaptability*, *survivability*, *fault isolation*, and *manageability* [4, p. 145-146].

A single system can't possibly achieve every quality measurement in this list. Trade-offs are inevitable, especially due to conflicts between measures, like *maintainability* and *complexity*. Different operational fields and use cases cause the priorities of these requirements to shift. While there are many more to consider, this work focuses on the ones comprised in table 4.

Name	Description	Reason for Prioritizationn
Maintainability	Formally, a measure for how easily the system can be returned to operational status once a maintenance action is required.	Frequently changing engineers and students contribute work to this project. They want to be able to quickly familiarize themselves with the system in order to make productive changes in short development cycles. Components, especially delivered by students and graduates, must have good testability.
Accessibility	Determines how easily components or subsystems can be accessed for maintenance or changes.	
Testability	Describes how easily faults can be detected and isolated in the system.	
Interchangeability	Determines how easily components and subsystems can be replaced without extensive calibration.	Various areas of application and future use cases can cause components to become obsolete. Engineers must be able to replace or remove them effortlessly.
Adaptability	The ability to adapt to a range of capabilities without major changes to the design or implementation.	
Portability	The system's ability to be transferred between systems and platforms.	Supported platforms range from popular OS, like Windows and Linux, to possibly embedded systems.
Scalability	The system's ability to grow in its capacity to meet increasing demands.	Future simulations require higher fidelity. When this system can scale, developing a new one after some years is unnecessary.
Accuracy	Required precision of the result that the system must meet.	Accuracy of generated sensory data is more important than its throughput, i.e., determined by the system's efficiency.

Table 4: Subset or prioritized quality requirements

4.3.2 Performance Requirements

Obviously, a software architecture for real-time synthetic environment generation is bound to performance constraints. They are compiled in table 5.

ID	Description	Reason
PR1	Synthetic sensory data and images shall be generated with at least 21 frames/samples per second.	HIL Simulations require steady frame rates at this rate.

Table 5: Performance requirements

4.3.3 Constraints and Assumptions

This work's purpose is based on a few assumptions that might improve how simulations are carried out at MBDA Germany. Table 6 lists them. On the other hand, the constraints of table 7 limit and influence design decisions.

ID	Assumption
A1	Distribution of synthetic environments could enable higher fidelity and more accurate results.
A2	Distribution of rendering tasks could decrease the maximum frame rate, but still to a usable degree.

Table 6: System Assumptions

ID	Constraint
C1	The system must run on Windows 8, 10, and 11.
C2	The system must run on Linux-based OS.
C3	The rendering pipeline must run on platforms without a window manager or screen.
C4	The Vulkan Scene Graph (VSG) library should be used for the scene graph and graphics implementation for evaluation purposes.

Table 7: System constraints

5 High-level Design

Using the requirements of section 4, the high-level architecture takes shape. Considering that it is optionally distributable, two aspects emerge:

- The hybrid architecture of an individual node's core enables them to communicate and collaborate in distributed simulations.
- Distribution topologies describing a cluster's structure, organizational hierarchy, and how its nodes collaborate to share their work.

5.1 Hybrid Architecture

As mentioned in section 4.3, quality requirements greatly influence the software design by matching the strengths and weaknesses of various architecture styles. Since a single architecture style does not satisfy all quality requirements (⇒ Section 4.3.1), a hybrid one similar to the ROS project (⇒ Section 3.1.4) represents the most practical solution.

The following section mostly focuses on the architectural structure employed inside individual nodes and how it grants them distributive capabilities.

5.1.1 Plugin-centered Design

The micro-kernel architecture (⇒ Section 3.1.1) yields technical and organizational advantages for this work. As shown in future 5.1, a running instance of the system can be subdivided into two types of components: The core embodies the underlying infrastructure required for running and integrating plugins. In return, they equip the system with capabilities and features. A standalone core without any plugins can barely exhibit useful functionality.

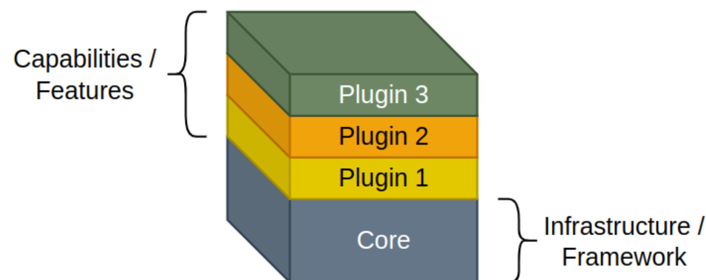


Figure 5.1: A single process running the system core and its loaded plugins.

First of all, utilizing a micro-kernel in the final architecture brings organizational advantages during development. Engineers can integrate features more easily and remove or replace them more effortlessly. In a monolithic architecture, this could require various changes across components. Due to their isolated nature, they rate high in this work's quality requirements *maintainability*, *accessibility*, *testability*, and *interchangeability*. Furthermore, other departments or users are not bound to the relatively small

core-developer team's support at MBDA Germany. They can create their own plugins and customize the system to fit their individual needs. Multiple tools can emerge from the same core system using a different set of plugins, reducing development times and avoiding a total rewrite each time. The main technical advantage is the ability to increase the system's performance by disabling unnecessary features. It can specialize in a specific use case and discard superfluous routines and calculations.

However, since abstraction almost always comes with costs, there are disadvantages to consider. If plugins are loaded at runtime, the compiler must employ runtime constructs, like so-called v-tables in C++ for virtual function calls of code unknown and unresolvable at compile-time. They introduce overhead and fixed performance costs when calling a plugin function because its location has to be determined at runtime before the actual call [16, p. 17]. Furthermore, the *fragile base problem* can arise when the core's API is altered in isolation, not considering how plugins use it. They may break as a result [22, p. 36f].

5.1.2 Service-driven Distribution

Services are a common method to distribute the overall workload in a network of service providers (⇒ Section 3.1.2). A single running instance of the core system with networking capabilities and its loaded plugins is called a *node*. A set of interconnected nodes forms a *cluster*.

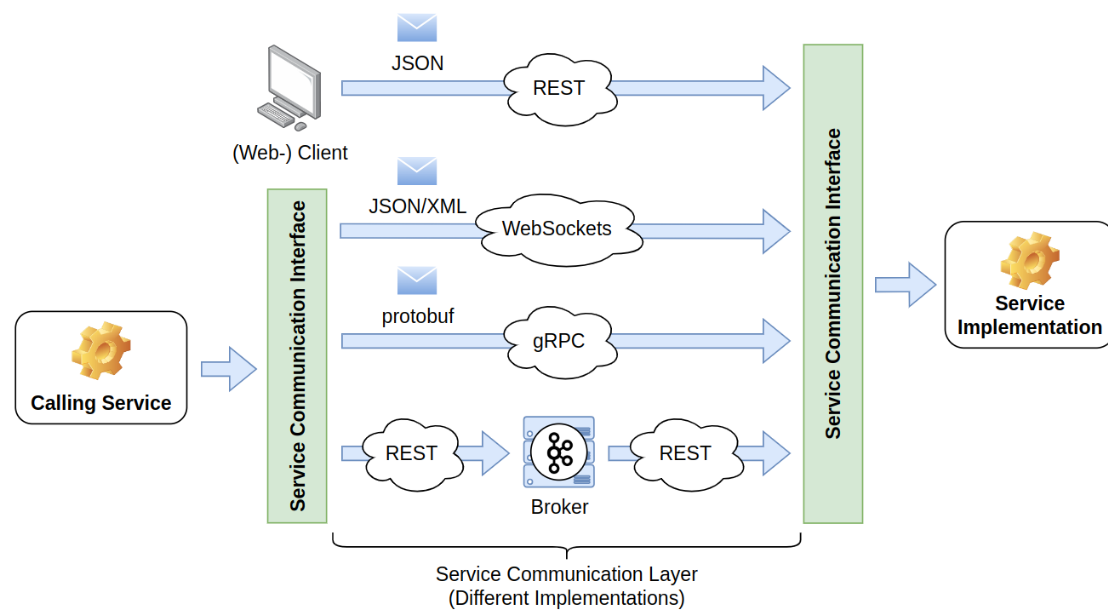


Figure 5.2: Service Communication Interface supports a range of communication protocols and hides their implementations from both service-caller and provider.

Now that the basic terminology is set, methods and approaches to providing and accessing services can be discussed. They can be called locally if provided by the same node or remotely. This calls for a communication layer between nodes used for request and response propagation. Due to many possible implementations (⇒ Section 3.1.2), services

should not depend on underlying network protocols, infrastructures, and transmission formats. Thus, it is important to hide these details from service implementations. Otherwise, when the underlying protocol is changed from REST to gRPC, for instance, all service implementations would be broken and need an update. This would result in bad maintainability and must be avoided due to this work's quality requirements (⇒ Section 4.3.1). The solution is an interface wrapping supported protocols, infrastructures, and transmission formats. The *service communication interface*, depicted in figure 5.2, renders these implementation details, which the core system must provide, exchangeable. It can also provide a multitude of protocols for different callers and, therefore, even integrate external clients or foreign infrastructures, like ROS or message brokers.

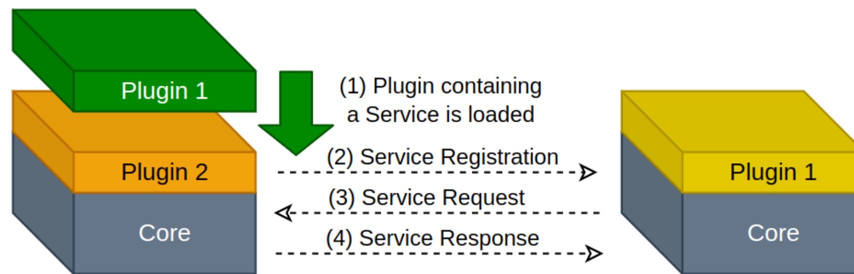


Figure 5.3: Newly loaded plugin registers a service on its node. The core informs another one about its availability and calls the service.

While the core provides the infrastructure and communication network, plugins supply services that members in the cluster can call. It requires a discovery mechanism to locate services in the network. There are multiple possible methods for nodes to know which services others provide. In figure 5.3, for instance, the node sends a *service registration* message to other connected ones. After they have been notified about the newly available service, they can call it using a *service request*, containing parameters. After the service has finished, the called node returns a *service response* containing the result, if any. This example further demonstrates how well the SOA and plugin architecture integrate for this use case.

This part of the overall architecture creates useful synergies with distributed rendering (⇒ Section 3.2.4). Image generation can be provided as such a service. The central composition node can request this rendering service for different perspectives, scenes, image resolutions, or other parameters. By establishing some load-balancing mechanism in the service infrastructure, rendering requests can be equally distributed among nodes. Nonetheless, it generates some overhead like the plugin-centered design (⇒ Section 5.1.1). Although its complex infrastructure is wrapped by the *service communication interface*, it still requires implementation efforts and maintenance. Furthermore, loosely coupled service requests introduce performance costs compared to direct function calls. Thus, not every routine should be implemented as a service, but only those possibly called by remote nodes. The paradigm *Everything-as-a-Service* is ineffective and should be avoided.

5.1.3 Event-Driven Synchronization and Communication

When multiple interconnected nodes share parts or replicas of the same synthetic environment, a synchronization problem arises. When a change to the environment or another crucial event occurs inside a node, the others must be notified to keep its parts or replicas consistent. Therefore, they must interchange event messages over their communication network. For this purpose, the event-driven architecture (⇒ Section 3.1.3) can employ event brokers passing these events to their designated subscribers. If the subscriber is located at a connected node, it can use the same messaging infrastructure used by the SOA for passing service requests and responses.

The broker topology is more applicable to this use case than its mediator counterpart. Events represent happenings and occurrences in the system that must be handled instead of commands that require processing. Furthermore, a simulation does not provide well-defined workflows essential for the mediator topology. Instead, events can occur in various simulation components running on different nodes. A dynamic set of subscribers, extendable through plugins, must be able to process these events. This constraint renders deterministic and fixed workflows impossible.

Considering a single node, the event-driven architecture is beneficial for some of this work's quality requirements (⇒ Section 4.3.1). Its software components can exchange information by contributing and processing events, creating a communication bus inside a node. This increases the overall architecture's maintainability, accessibility, and interchangeability. Plugins, for instance, can integrate more effortlessly with others and the core system by accessing this communication bus. On the other hand, this approach inherits all weaknesses of the broker topology. Its highly dynamic event flows are difficult to predict, debug and test. Additionally, the plugin developer is responsible for avoiding cycles in event chains if their plugins contribute to them.

5.2 Distribution Topologies

Nodes can organize themselves in a cluster for collaboration in different topologies. This section discusses possible structures and hierarchies of a cluster. Because the core does not enforce concrete topologies and only provides a basic messaging infrastructure, they can be implemented using plugins. For instance, for giving a node the ability to contribute to a Peer-to-Peer (P2P) network. This decision allows the architecture to adapt various topologies for different use cases and distribution strategies. This section will cover two possible topologies that could prove useful for distributed simulation and give an example of the usage of hybrid architecture.

5.2.1 Coordinated Workers

This topology follows the Client-Server-Server (C^+SS^+) paradigm and was formerly known as *master-slave*. Just like in the Client-Server (C^+S) paradigm, clients want to access services provided by servers. However, they do not access these servers directly. Instead, an intermediate server, the coordinator, is installed and takes organizational roles. As depicted in figure 5.4, it manages a set of workers, providing fine-grained services and processing assigned workloads. They are usually unaware of each other's existence and only serve the coordinator [2, p. 61f].

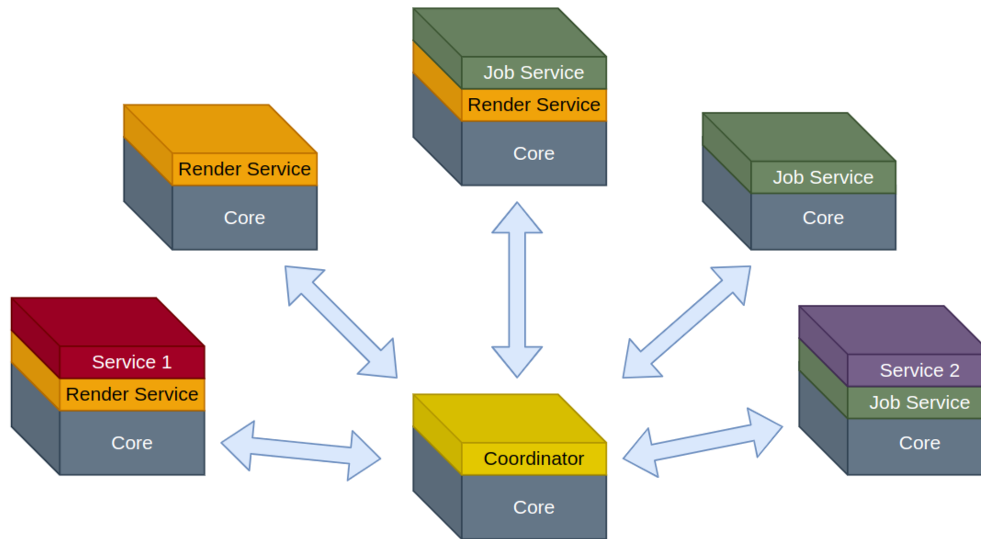


Figure 5.4: A central *coordinator* node controls its *workers* to distribute workload.

Other simulation participants, like UAS models or human operators, act as clients and interface with the coordinator node for interacting with the synthetic environment. Based on their plugins, workers can provide various services to the coordinator or process generic units of work, called jobs, distributed between them. They can join the cluster by registering themselves at the coordinator. It could also be useful to give it the ability to spawn new workers dynamically with certain capabilities as required.

As for most centralized organizational structures in distributed computing, the central coordinator can become a bottleneck or a single point of failure [33, p. 13]. Although this might be a relatively simple method for splitting the overall workload, compared to P2P related approaches, its scalability is limited and always bound to the coordinator's capacity. On the other hand, it allows a central state of the synthetic environment, possibly reducing synchronization costs between nodes. Distributed rendering (⇒ Section 3.2.4) can utilize this topology. The coordinator can represent the central rendering node and assign rendering tasks to its workers.

5.2.2 Grid Computing

Grid computing is derived from the P2P approach. In contrast to the C^+SS^+ hierarchy, they differentiate themselves through their symmetry. Every peer in these networks acts as both client and server, consuming and providing services, computational resources, or memory storage. There is no need for a central server in P2P and grid computing networks. Due to the absence of this central bottleneck and the network's self-organizing nature, they allow for higher scalability than their client-server counterpart. Their applications range from *digital content sharing* to scientific computations, where expensive calculations run on a set of computing resources instead of a single supercomputer. The difference between the P2P and grid computing approach lies in their contributors and their trust. While the former is highly dynamic and volatile because peers can leave and join anytime, grid computing is more stable and well-structured in this aspect. It focuses on pricier high-end computing resources and is more exclusive by only allowing

a certain set of dedicated peers to contribute to the network. On the other hand, P2P barely restricts member access and is more open to cheaper but also possibly malicious edge devices. Therefore, security and trust measures have higher significance and require more effort in P2P approaches. Grid computing peers can collaborate in *good faith* due to their mutual trust. Challenges in both approaches are data consistency and integrity across peers, routing and resource discovery, and security [33, p. 2-9].

Employing grid computing instead of P2P is more practical for this work's purpose. Missile simulations require dedicated computers and servers equipped with pricy graphics hardware instead of weaker edge devices. Given the assumption that simulation computers are running in an isolated environment, this choice also allows the neglect of costly security measures due to their shared trust. For instance, encryption and decryption of every passed message between peers can be omitted to save computational costs. Obviously, this decision is based on assumptions and increases the security risk. However, this tradeoff can improve the cluster's performance and efficiency in real-time simulations.

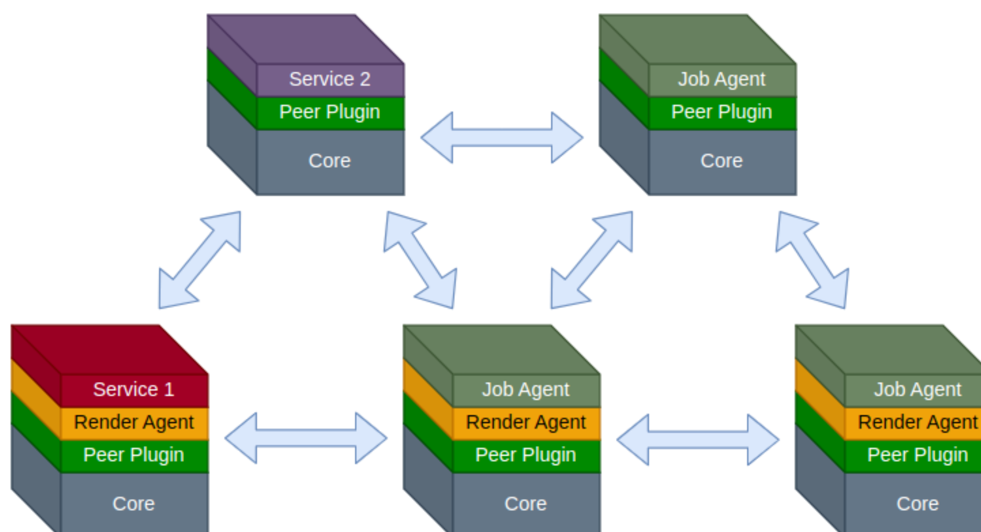


Figure 5.5: Nodes collaborating in a grid computing scenario. Every node acts both as a client and a server.

As depicted in figure 5.5, there is no central node in a grid computing cluster. In this scenario, nodes only run on dedicated servers equipped with costly graphics hardware instead of weaker edge devices, like workstations or mobile devices. This could increase the performance of the overall distributed simulation because services can be requested directly without a central instance acting as a relay. On the other hand, it inherits weaknesses of the P2P approach and grid computing. This approach requires more complex service discovery and routing algorithms than its client-server alternative [33, p. 39ff]. Additionally, the absence of a central instance increases the synchronization efforts of the synthetic environment's state.

A central database is an alternative for managing the global state of the environment. The Common Data-Base (CDB) is an open database specification of the Open Geospatial Consortium for synthetic environments. It is specialized in storing geospatial information

for high-performance M&S applications. Supporting a wide variety of NATO standards and protocols, tiling, and level-of-detail interpolation, it can stream the current environment's state to cluster members [24]. Obviously, a new bottleneck and single point of failure can emerge by installing such a central component.

6 Core Design and Implementation

6.1 Subsystem Overview

As depicted in figure 6.1, the core system consists of multiple modules, also called subsystems, subdivided by their purpose and technical domain. The implementation of a subsystem is hidden behind its API and can only be accessed through it. It enforces loose coupling and allows exchanging implementations without affecting other modules [22, p. 1-7]. For instance, the same *job system* API, providing functionality for scheduling routines to other modules or plugins, has both single-threaded and multi-threaded implementations that can be interchanged or extended by new ones in the future.

Before this section discusses each subsystem's implementation and technical background, their coupling and collaboration must be clarified. Often, they depend on others further down the stack, but there is no guarantee that an implementation is always provided. For instance, the networking module only *exists* if the core was configured for operating in a cluster. Otherwise, the compiler will skip this module, or the configuration will deactivate it at runtime. Therefore, the existence of each subsystem is optional and must be queried before use. The subsystem management layer governs implementations for subsystem APIs and couples them loosely. Using this component, other modules can query, load, or replace them if necessary.

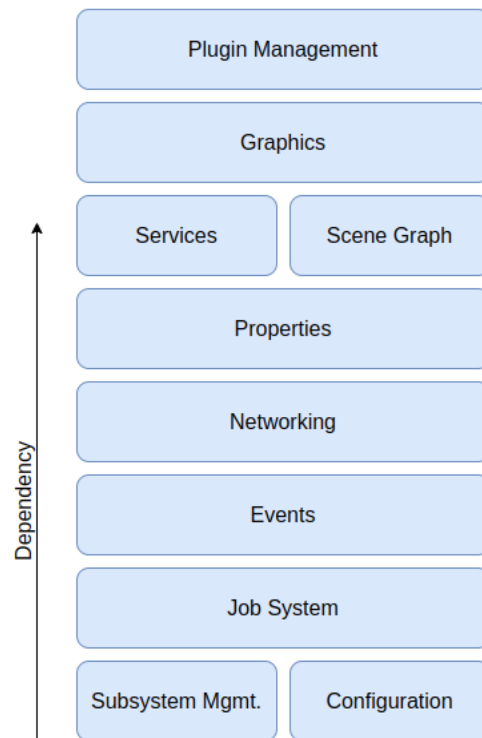


Figure 6.1: Modules of the core system

The programming language of choice for the core implementation is C++ because the central library used for 3D Vulkan graphics only provides a C++ API (⇒ Section 4.3.3). Establishing library compatibility for other languages by wrapping them or generating interfaces would require a significant development overhead in this bachelor thesis and was therefore omitted. More high-level languages, like Rust or C#, are worth considering in the future when necessary bindings or library alternatives are available. Script bindings are often used in game engines for convenient access to features and faster development cycles for rapid prototyping due to a simplified language design [8, p. 1135f], [22, p. 329-359]. The core API can provide such a scripting API in the future for the same purpose.

The software architecture of game engines inspires some subsystems discussed in this section. In computer science, games are soft, real-time, agent-based interactive computer simulations [8, p. 9]. From this perspective, the generation of synthetic environments for real-time simulations is highly related to interactive video games. For this reason,

the software architecture of game engines, acting as frameworks for game development, inspired this work.

6.2 Job System

Plugins contribute their own functionalities, services, and routines to the system (⇒ Section 5.1.1). Therefore, the core must provide an extensible way for scheduling and processing arbitrary units of work accessible by plugins and subsystems. For instance, the *Naughty Dog* game engine employs a so-called job system for this purpose. It allows game developers to divide game logic, rendering tasks, and input-output operations into independent units of work called jobs. Figure 6.3 shows that pending jobs are maintained in a queue and scheduled for concurrent execution in a thread pool. Operations on native threads provided by the OS's kernel require the running process to switch from user space to kernel space. This switch consumes many CPU cycles and introduces performance costs when done for every job. For this reason, a set of threads is kept and re-used for processing instead of costly spawning a new one for each job. Otherwise, the job system would be more busy creating and cleaning up threads than following its main purpose of executing jobs. This way of processing work in the engine enabled significant speed-ups in the transition to newer-generation gaming consoles [8, p. 253f, 549–558]. A job system also proves very useful for this work's purpose. Like video game routines, developers can decompose the entire simulation logic into finely granular jobs and hand them to the job system for execution, following a *Everything-as-a-Job* paradigm.

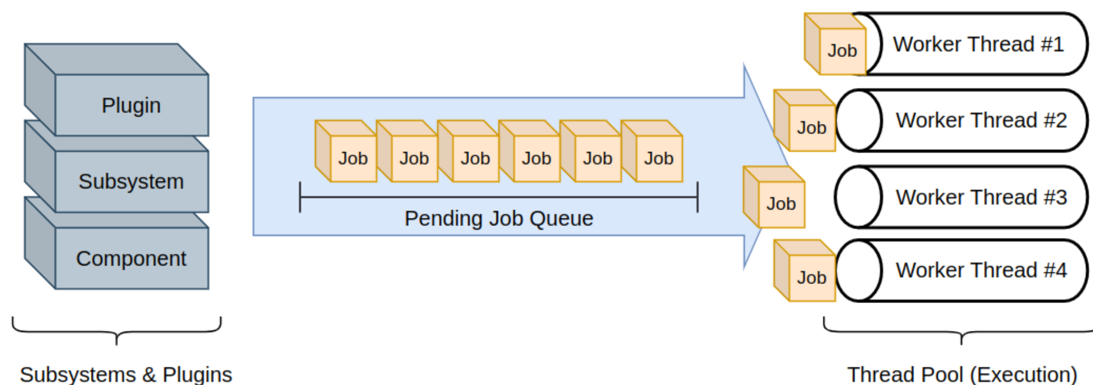


Figure 6.2: Plugins and subsystems can schedule jobs for concurrent execution.

There are multiple approaches for implementing the job system's execution component. Besides simple worker threads, so-called *fibers* are applicable as job executors and can bring some performance benefits. The main goal of the job system revolves around keeping its worker threads busy and their utilization as high as possible. Thus, idle or blocking operations reduce efficiency by occupying processing resources, like threads, without using them to capacity. There are multiple occasions in which jobs could block a thread, such as waiting on the completion of others or pending input-output operations. To ensure the highest processor utilization, it is essential that the job execution can put waiting or blocking jobs aside and resume when they are ready. In the meantime, it can start or continue the execution of others. Ideally, switching jobs avoids blocking a thread or keeping it busy waiting, essentially increasing the utilization of processing resources.

However, it requires exchanging the current call stack and register values, known as a thread's context, at runtime. Ordinary threads are not capable of doing so. Fibers, on the other hand, are specialized threads that can schedule their contexts cooperatively and switch between them as needed. Therefore, the job system of this work employs fibers instead of ordinary worker threads. It provides a performance benefit due to the parallelized processing of jobs and allows plugins to schedule their own routines, increasing the system's overall extensibility. However, its concurrent processing requires locks for synchronization in all higher-level components. Otherwise, jobs executed on different fibers could cause data races and non-deterministic behavior [8, p. 553ff].

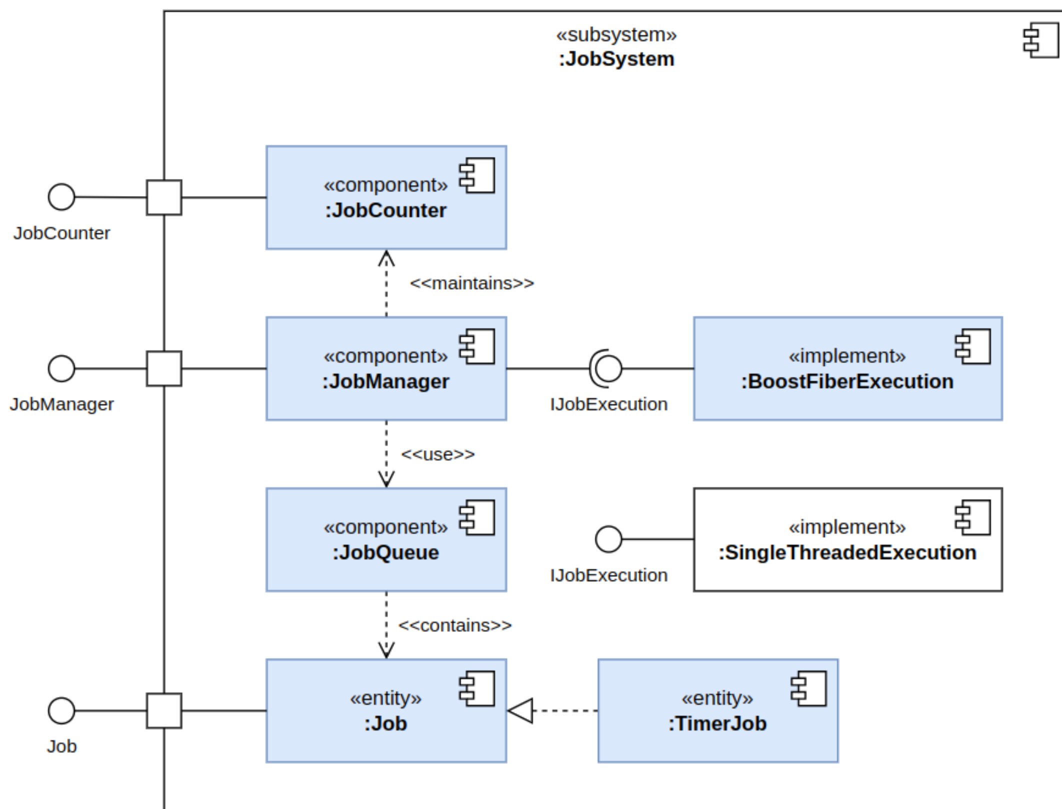


Figure 6.3: Software component design of the job system module.

As shown in the component diagram in figure 6.3, the job system essentially consists of the following components:

- Implementations of the *IJobExecution* consumes jobs and processes them. It receives them by the *JobManager* and acts as a central processor for the system. The main implementation uses fibers and context switching provided by the Boost library, but there is also a single-threaded alternative.

- The *JobManager* is responsible for maintaining job queues, passing jobs to the job execution in an orderly manner, and coordinating their execution. In the implementation of this work, the manager keeps three queues for different phases of the execution cycle: Initialization, processing, and cleanup.

- A *Job* contains workload, usually as lambda functor, and can be specialized to serve specific needs. For instance, a *TimerJob* executes not every cycle but only in timed intervals.

- A *JobCounter* tracks the completion of jobs and can be used for their synchronization. Jobs can be assigned to counters so that others can wait until they have been completed.

```
// get current job manager subsystem
auto job_manager = subsystems->Require<JobManager>();

// create a counter for synchronization
auto cntr = std::make_shared<JobCounter>();

// create a job loading something (blocking) with counter
auto loading_job = std::make_shared<Job>([&](JobContext *ctx) {
    // wait for future inside job
    std::future fut = load_something();
    ctx->GetJobManager()->WaitForCompletion(fut);
    LOG_INFO("loading job completed");
    return JobContinuation::DISPOSE;
}, "loading-job");
loading_job->AddCounter(cntr);

// create a job waiting on the other one's completion
auto waiting_job = std::make_shared<Job>([cntr](JobContext *ctx) {
    ctx->GetJobManager()->WaitForCompletion(cntr);
    LOG_INFO("waiting job completed");
    return JobContinuation::DISPOSE;
}, "waiting-job");

// job executing every second for 5 times
std::atomic<int> repetition = 0;
auto interval_job = std::make_shared<TimerJob>(
    [&repetition](JobContext *) {
        if(repetition < 5) {
            repetition++;
            return JobContinuation::REQUEUE; // re-queue for next cycle
        }
        return JobContinuation::DISPOSE; // don't re-queue job again
      },
      "timed-job", 1s); // set time interval to 1 second

// kick jobs and start cycle
job_manager->KickJob(waiting_job);
job_manager->KickJob(loading_job);
job_manager->KickJob(interval_job);
job_manager->InvokeCycleAndWait(); // Starts execution of kicked jobs
```

Listing 1: Usage of the Job System and its API

Jobs are accumulated in the job manager's queues until a new execution cycle is invoked. A cycle is divided into three phases: Initialization, main processing, and cleanup. Many resources, like plugins or services, need to be initialized before their usage and are unusable after their clean-up. Otherwise, unwanted behavior or crashes can occur in the same cycle. To avoid such conflicts and overly complicated synchronization efforts, the manager maintains three separate queues for each cycle phase, which are executed ex-

clusively in order. Code snippet 1 demonstrates the job systems's API usage.

After implementing all modules in a way that they are based on the job system, further benefits and obstacles become apparent. On the one hand, the ability to execute jobs in timed intervals reduces complexity and implementation efforts in higher-level modules. For instance, the graphics subsystem (⇒ Section 6.8) can implement its rendering loop by simply scheduling a timed interval job executing 60 times per second for this purpose. On the other hand, the inherently concurrent fiber implementation requires the wise installation of locks in all higher-level modules. This can become a real challenge, leading to grueling, non-deterministic bugs during development due to racing conditions.

Although the coordinated-worker topology (⇒ Section 5.2.1) suggests the distribution of jobs to workers, this is not a capability implemented in this job system. It would require methods for encapsulating the workload of a job in a transferrable or serializable format. Otherwise, participants of the cluster cannot share their jobs with others. Lambda functors, which are essentially pointers to local functions, do not qualify for this use case. In contrast, small routines written in scripting languages do because they can be transmitted in their text format. However, the core does not yet employ a scripting API and cannot share jobs with other nodes for this reason.

6.3 Event Subsystem

Game engines use event systems, sometimes also called messaging systems, to transmit interesting changes in the environment or the game's state [8, p. 531]. The same principle can also be useful for distributed missile simulations. Components, subsystems, plugins, and other nodes can listen to events of certain types and get notified about changes in the synthetic environment. As mentioned in section 5.1.3, events can be used to exchange information between components of the same node and inside a cluster. Furthermore, they carry global state changes to interested parties for synchronization.

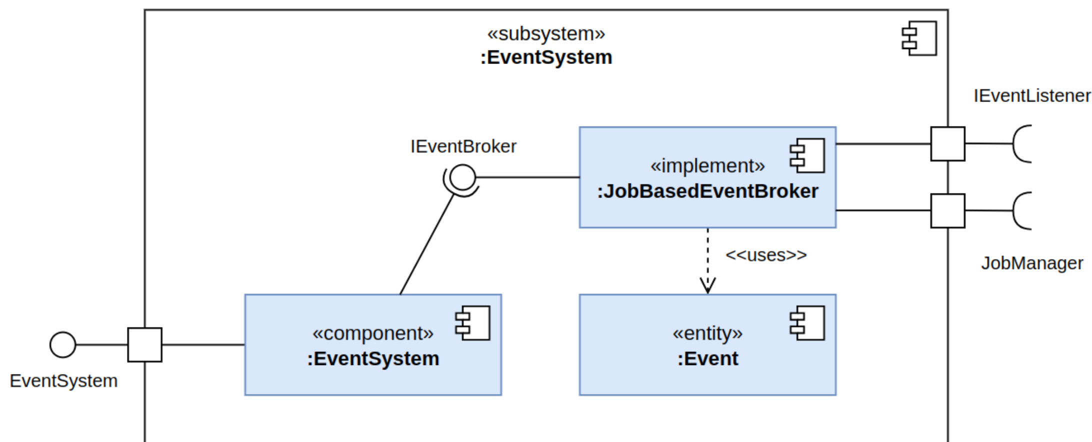


Figure 6.4: Software component design of the event system.

The component diagram shown in figure 6.4 is kept as extensible as possible. The current implementation of the event broker is very minimalistic and utilizes the job system to

propagate events to their listeners. Other modules supply these listeners acting as callback routines in case of an occurring event of a certain type. Event types are represented by a string value for loose coupling between the triggering party and their listeners. An event's arguments contain context information and additional parameters that listeners could require for processing. A use case for this subsystem is an event that triggers when another node has established a connection. Its hostname, IP address, port, and other information can be attached to the fired event. The service module is interested in new connections to consider them for future service requests. For this purpose, it implements and registers a listener for this event type to process new connections to other nodes in the cluster.

As mentioned in the previous section, event systems often maintain queues, temporarily storing events before processing. They allow deferred event handling in future cycles or phases. Like timer jobs, timed events can be realized this way [8, p. 1124-1129]. The job system renders the broker's implementation very lightweight. It simply has to schedule a job that notifies the listener and executes its callback routine. By doing so, it uses the job system's queue instead of providing its own one.

However, the event subsystem itself does not provide the networking functionality required for transmitting events to other nodes in a cluster. It only supports instance internal event propagation. A dedicated subsystem adding networking capabilities to the event flow is discussed in the next section (⇒ Section 6.4).

6.4 Networking

The networking subsystem is responsible for establishing and managing connections to other nodes. It supplies them to higher-level subsystems or plugins, creating a unified communication layer for sending and receiving data in a cluster. Other facilities use this layer to satisfy their external communication demands, like the *Service Communication Interface* (⇒ Section 5.1.2) sending service requests and responses. Hiding implementation details, like the underlying protocol or message format, is essential. This communication layer must not bind to specific protocols or other implementation details because there might be alternative protocols to evaluate and consider in the future. Furthermore, other components are not concerned with them but only interested in passing messages to other nodes. Furthermore, the implementation of concrete topologies is avoided in this subsystem. It focuses on point-to-point connections, which can be established and used from either side. Every node must be able to initiate connections and listen for incoming ones. Therefore, concepts like clients or servers are avoided at this level. They are implementable in higher-level components using point-to-point communication as their basis.

The first implementation of the communication layer utilizes so-called web-sockets. They are suited for real-time data exchange and specialize in low latency by reducing communication overhead, like repeated handshakes before every message. Based on HTTP, Web-Sockets provide full-duplex bidirectional communication and outperform traditional approaches using polling [18]. This means both endpoints can send and receive messages through a single persistent connection with low latency. This protocol was chosen since generating synthetic environments in real-time requires quick transmission of state changes or binary data, like images in distributed rendering (⇒ Section 3.2.4). Obvi-

ously, there are alternative protocols to evaluate in the future. The ROS project (⇒ Section 3.1.4), for example, employs the DDS protocol [25] for this purpose. Another interesting approach, especially for real-time image streaming in distributed rendering, is WebRTC: Google's former cloud platform *Google Stadia* offered users to play demanding video game titles over the internet at a fixed monthly rate instead of investing in an expensive gaming console. User inputs, sounds, and rendered frames were streamed in real-time between the user's end device and the remotely running game using WebRTC. Unfortunately, this protocol is fairly complex and requires cumbersome facilities provided by default in web browsers, but it would have to be manually implemented at great expense for this project [3]. Therefore, web-sockets were chosen due to time constraints. As already mentioned, the networking subsystem is designed so that engineers can add support for such protocols in the future.

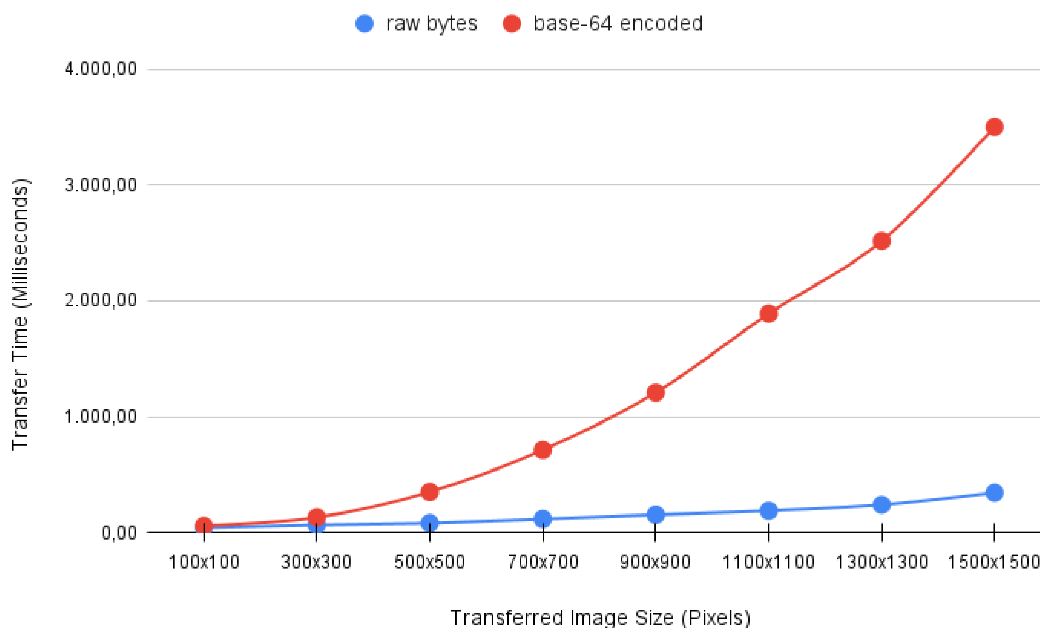


Figure 6.5: Comparison of transfer times of generated frames in different sizes based on their encoding.

To transmit information to other nodes, messages act as envelopes and central entities of the networking subsystem. Each message contains a payload organized in key-value pairs carrying the information to its recipients. Serialization is necessary to transform message objects into a format transportable by the underlying protocols. At first, JSON was considered as the transmission format. It worked fine for numeric or string payloads but not for binary data, like images or files. Serializing raw bytes can result in special characters or bit sequences corrupting the JSON string and its UTF-8 encoding. To avoid this corruption, binary data must be encoded using an algorithm like Base64. The result is a string containing a file's or image's bytes that must be decoded at the receiving end. However, as the benchmark shown in figure 6.5 reveals, this approach does not scale well. Generated frames were transferred using web-socket messages using Base64 encoding to another node running on the same host. The recipient then decoded the received frames

and copied them into a buffer. The elapsed time of this entire process was measured for different frame sizes to evaluate the performance footprint of the Base64 algorithm (red). For reference, the transfer time of raw bytes without any encoding was captured as a baseline (blue). Introducing the Base64 algorithm to message serialization causes significant overhead and performance losses in use cases like distributed rendering. Therefore, JSON does not represent a viable solution for messages containing binary data because it requires the costly encoding of raw bytes. Another approach is to send the message in multiple parts, like attachments, which can contain raw binary data and omit additional encoding. Multipart Form-Data is such a format. As its name implies, it allows a single message to carry multiple contents or files by sectioning it using boundaries. Every part is assigned a name and content type in the new implementation [13]. Every message's first part carries meta information in JSON, like its unique identifier or topic. Similar to attachments of an e-mail, the succeeding parts contain the message's payload and binary data.

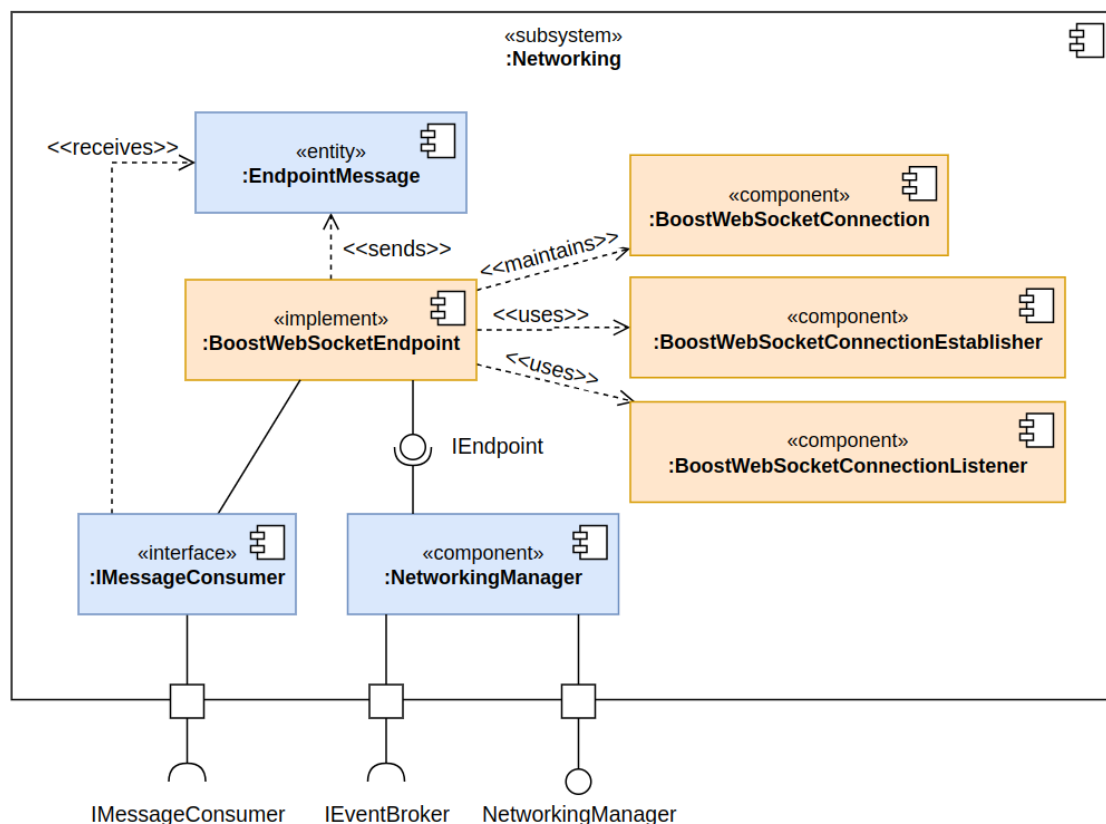


Figure 6.6: Software components of the networking subsystem (blue) and their web-socket implementation (orange) using the Boost library.

Figure 6.6 shows the main components of the networking subsystem.

- An *EndpointMessage* encapsulates sent or received information. This subsystem revolves around transmitting and handling instances of this component. Every message belongs to a type that describes its purpose and contents.
- The *IEndpoint* interface defines functionality for sending and receiving messages.

As discussed earlier, there are many possible implementations. The *BoostWebSocketEndpoint* uses the Boost library to provide web-socket connections transmitting *EndpointMessage*s using Multipart Formdata.

- The *NetworkingManager* starts implementations of the *IEndpoint* interface and configures them.
- Higher-level components want to receive *EndpointMessage*s of a certain topic. They can implement *IMessageConsumer*s and register them for incoming messages. For instance, the service subsystem (⇒ Section 6.6) registers the *ServiceRequestConsumer* to process incoming messages of this topic.

This module integrates with the event subsystem to inform interested components about established or closed connections, like the service infrastructure. Therefore, it requires an event broker. Although this module integrates well with the event architecture, this implementation is incompatible with the job system. Theoretically, jobs can execute sending and receiving actions of web-sockets. In practice, however, Boost's web-socket implementation spawns and maintains its own threads under the hood, breaking the *Everything-as-a-Job* paradigm.

6.5 Property Management

Components, subsystems, or plugins all rely on and operate on data. It can act as their configuration or state and is usually maintained by the component itself. This decentralized approach works fine until these components need to access or modify each other's data. Due to their loose coupling, information is not always available. An example should clarify this constraint: A plugin that renders water surfaces in the scene, like an ocean, depends on the lighting information of another plugin managing the sky and atmosphere. Calculating the ocean's shading might require the sun's position and the atmosphere's ambient color. Both plugins are loosely coupled and are not guaranteed to be loaded simultaneously. Furthermore, dynamic data like the sun's position can change over time. In this case, the ocean plugin must also be notified to adjust its shading. This example shows that data management can get messy and difficult to maintain if components implement it on their own in an isolated manner.

Properties are information that can be accessed and modified by multiple parties. This subsystem manages them centrally and notifies dependent components when their value changes. This notification is an event thrown by the property provider and handled by listeners. It forms a data layer between different subsystems and plugins, increasing the system's overall accessibility and maintainability. Using the networking subsystem, it can also synchronize properties between nodes in the cluster. The component design is shown in figure 6.7.

6.6 Service Infrastructure

As covered in section 5.1.2, considering the following constraints is essential in the service infrastructure implementation.

- Multiple nodes in the cluster can provide the same service. A caller must alternate between providers to balance the workload.

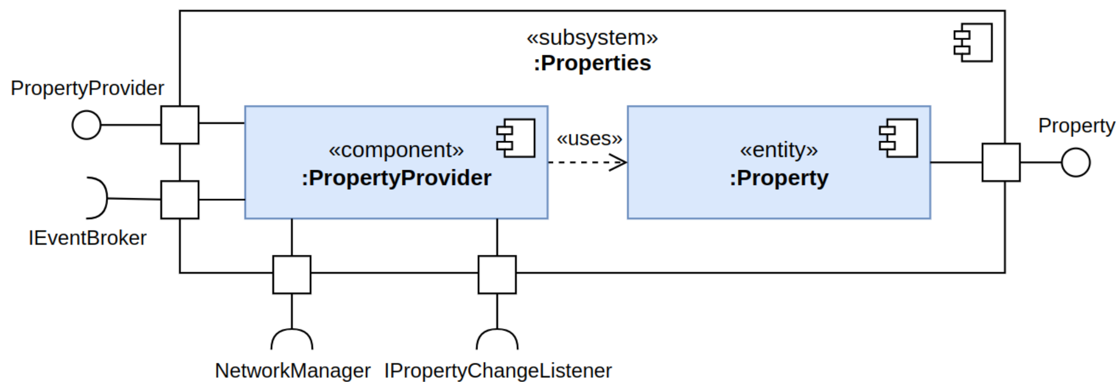


Figure 6.7: Software Component Design of the property management subsystem.

- A requested service can be located on the same node or on another one in the cluster. Therefore, a method for informing a cluster's members about currently available services and their providers must be established.
- A node should be able to function offline, outside of a cluster, only calling locally available services.
- Service implementations must not depend on the underlying details of the communication infrastructure.

As shown in figure 6.8, the main component provided by this subsystem is the service registry. This module uses the implementation capable of remote service calls by default, but a simpler local-only alternative is also available. To understand the purpose of caller components, like the *RoundRobinServiceCaller*, the concept of executors must first be discussed. They represent a service's implementation, which can either be located locally on the same node or a remote endpoint. A *LocalServiceExecutor* directly calls the service implementation, which is essentially an ordinary function call wrapped by a job. *RemoteServiceExecutors* use the networking subsystem's communication capabilities to perform remote service calls. A *ServiceRequest* wrapped in a message is sent to another endpoint. Its registry receives the request and performs the service execution. Since services are request-response oriented, in contrast to events, the requesting endpoint awaits a *ServiceResponse* message containing the ordered results. Due to the job system, this can be done without blocking a worker thread, allowing an arbitrarily high amount of pending services.

As mentioned, multiple executors in the cluster can provide the same service. A caller, like the *RoundRobinServiceCaller*, is an intermediate handler containing all known executors of a distinct service in the cluster. The requesting party is only interested in having its service request processed. It does not care which specific executor takes on this responsibility. The task of selecting an executor using some strategy and criteria is delegated to the *IServiceCaller*. The registry collects all known executors associated with the requested service's name in a caller instance and returns it. Listing 2 shows this in line 6. When the service is called, the used *IServiceCaller* selects a *IServiceExecutor* from its list. This selection can be made using various strategies. For instance, the *RoundRobinServiceCaller* tries to alternate between executors every call to balance the workload among them. More complex strategies considering runtime statistics and more

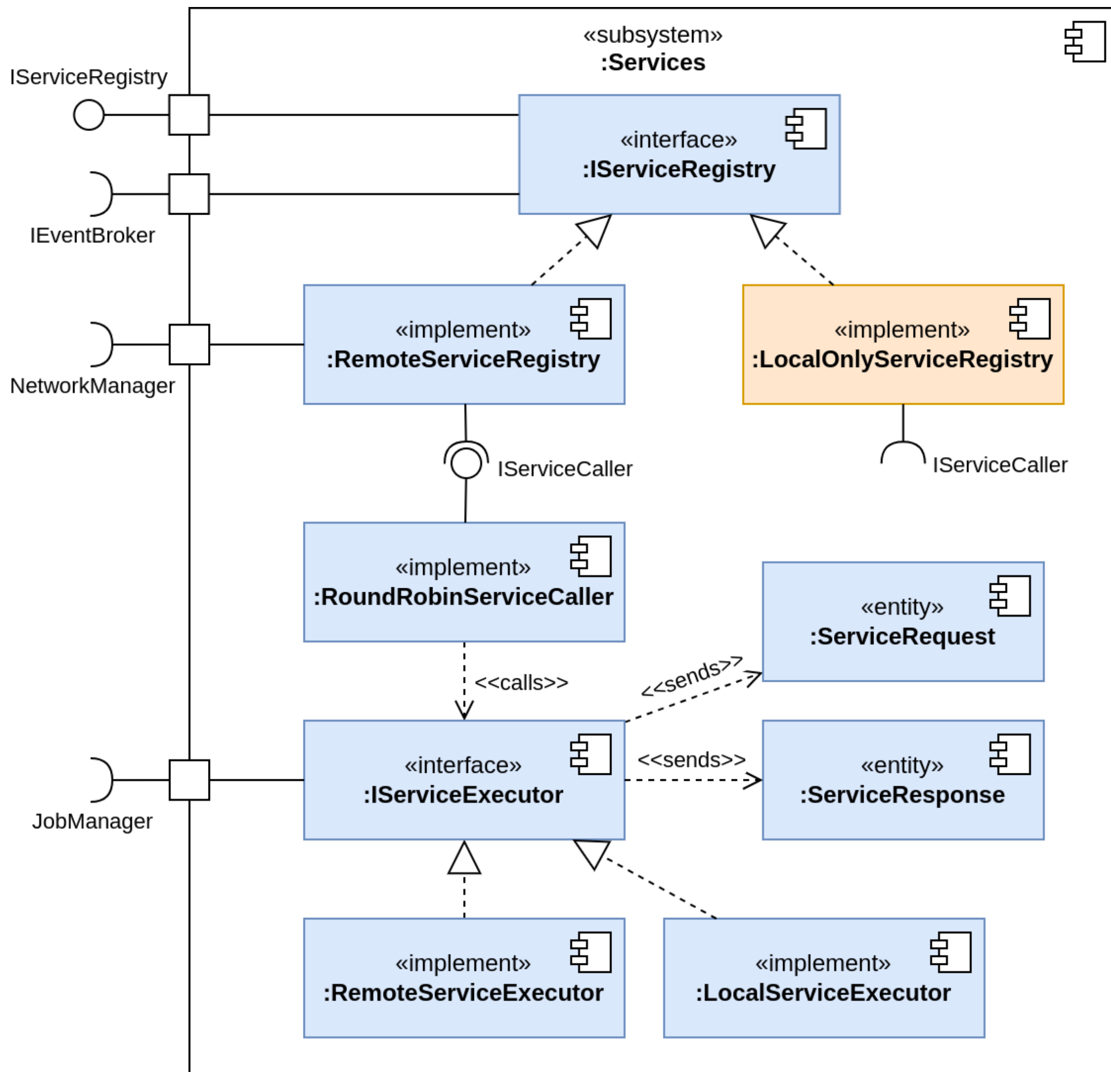


Figure 6.8: Software component design of the service infrastructure. This implementation provides both a service registry capable of remote service calls and an offline alternative (orange). Components needed for other subsystems, like implementations of message consumers for the networking subsystem, were omitted in this diagram.

effective load-balancing are implementable in the future.

Yet, an endpoint can't call a remote service if it does not know exactly which node provides it. It is mandatory that nodes can offer their services to others or discover them in the cluster. Different cluster topologies require different approaches to service discovery. A client-server environment, for instance, can provide a central service registry that other nodes can query, while a pure P2P topology requires more advanced and decentralized algorithms [26, p. 213f]. In this implementation, the foundation for many approaches is the *service registration* message that offers a service to another node. With this basic building block, other discovery methods can be realized. A central node acting as the server could offer a cluster-wide registry service to recently joined nodes. They

```
// get registry implementation as subsystem
auto registry = subsystems->Require<IServiceRegistry>();
auto job_manager = subsystems->Require<JobManager>();

// represents all executors known to the registry
auto caller = registry->Find("my-service-name");

// perform service request inside a job
ServiceRequest req; // contains parameters
std::future<ServiceResponse> res = caller->Call(request, job_manager);

job_manager->WaitForCompletion(res);
```

Listing 2: Service request using callers

could use this service to locate other services, using it as a centralized registry.

6.7 Scene Graph and Management

A scene graph (⇒ Section 3.2.3) can hierarchically organize objects of a synthetic environment. Special types of scene graphs can optimize use-case-dependent rendering techniques, like raytracing. In theory, it would be beneficial to make their implementations interchangeable for different types of scene graphs. The open-source *OGRE 3D* rendering engine demonstrates this interface abstraction [8, p. 38f] by providing an interchangeable scene graph and graphics API.

Looking at this work's requirements, the implementation's library is already determined (⇒ Section 4.3.3). Vulkan Scene Graph (VSG) is the successor of the Open Scene Graph (OSG) library, which is used in other simulation tools at MBDA. As its name implies, it comes with a Vulkan (⇒ Section 3.2.2) rendering component.

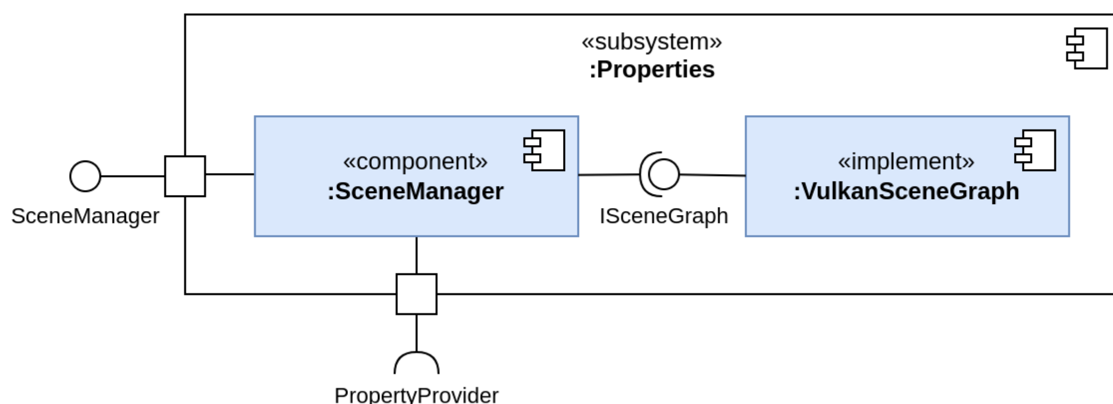


Figure 6.9: Theoretical software component design of the scene subsystem.

However, the interchangeable design shown in figure 6.9 is difficult to apply in practice. The *ISceneGraph* interface for such a complex library is difficult to construct. It has to abstract the functionality of various scene graph types. Additionally, its implementation would have to wrap VSG's API and all its objects. This is not feasible to accomplish in the time frame of this bachelor's thesis. Furthermore, such a big and complex interface is

difficult to maintain. Although this decision reduces the scene graph's interchangeability, this implementation omits the interface. The result is a *more practical* component design tightly coupled to VSG, shown in figure 6.10.

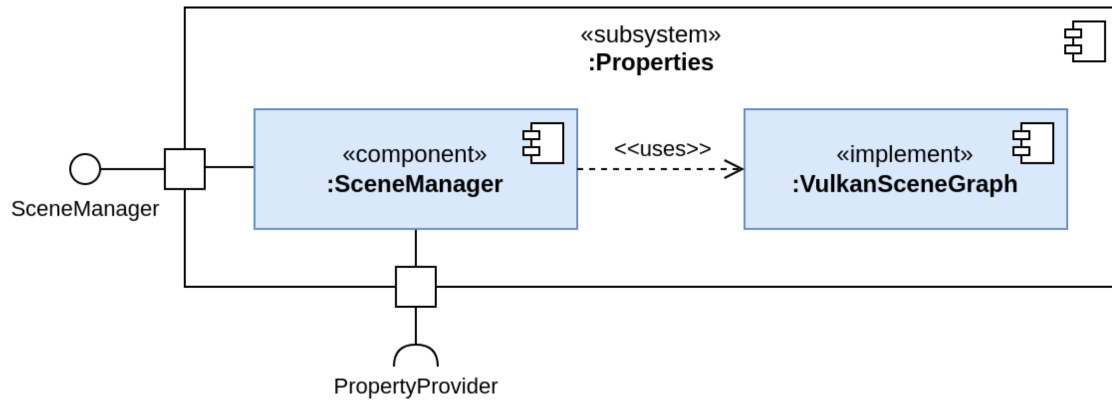


Figure 6.10: Practical software component design of the scene subsystem omitting the interface.

The *SceneManager* can also integrate with the property system (⇒ Section 6.5). Properties can store states of the scene's objects, like positions or orientations. Thus, it can synchronize these states across nodes in a cluster or make them accessible to other components. This feature has not yet been implemented but can be added in the future when required.

6.8 Graphics and Rendering

The graphics subsystem aims to generate sensory data for image-based sensors like IR cameras or radar seekers. Both an onscreen and offscreen renderer must be functional to supply images to human and non-human operators alike. A service for this purpose must be established to enable remote image generation and distributed rendering.

Two graphics APIs, Vulkan and OpenGL, have been discussed before (⇒ Section 3.2.2). Looking at this work's constraints (⇒ Section 4.3.3), VSG was already selected as the library for the scene graph implementation. Therefore, this subsystem employs Vulkan for image generation instead of OpenGL. The basis for this decision consists of multiple advantages for this use case. The same Vulkan API runs on all supported platforms, like embedded systems. There is no need for portations to more lightweight implementations, like OpenGL ES, for this purpose. Additionally, Vulkan's computational model is more open to non-graphical use cases, like thermal calculations, making their implementation easier than OpenGL. Another important reason is its multi-threading capability, which could improve the performance in future use cases.

As the component diagram in figure 6.11 shows, this subsystem provides the *IRenderer* interface implemented for onscreen and offscreen applications. Another implementation is contributed by the distributed rendering plugin (⇒ Section 7.1). All implementations use VSG as an abstraction layer for Vulkan. As already mentioned, its API is very complex, low-level, and verbose. VSG wraps commonly used procedures and API calls in

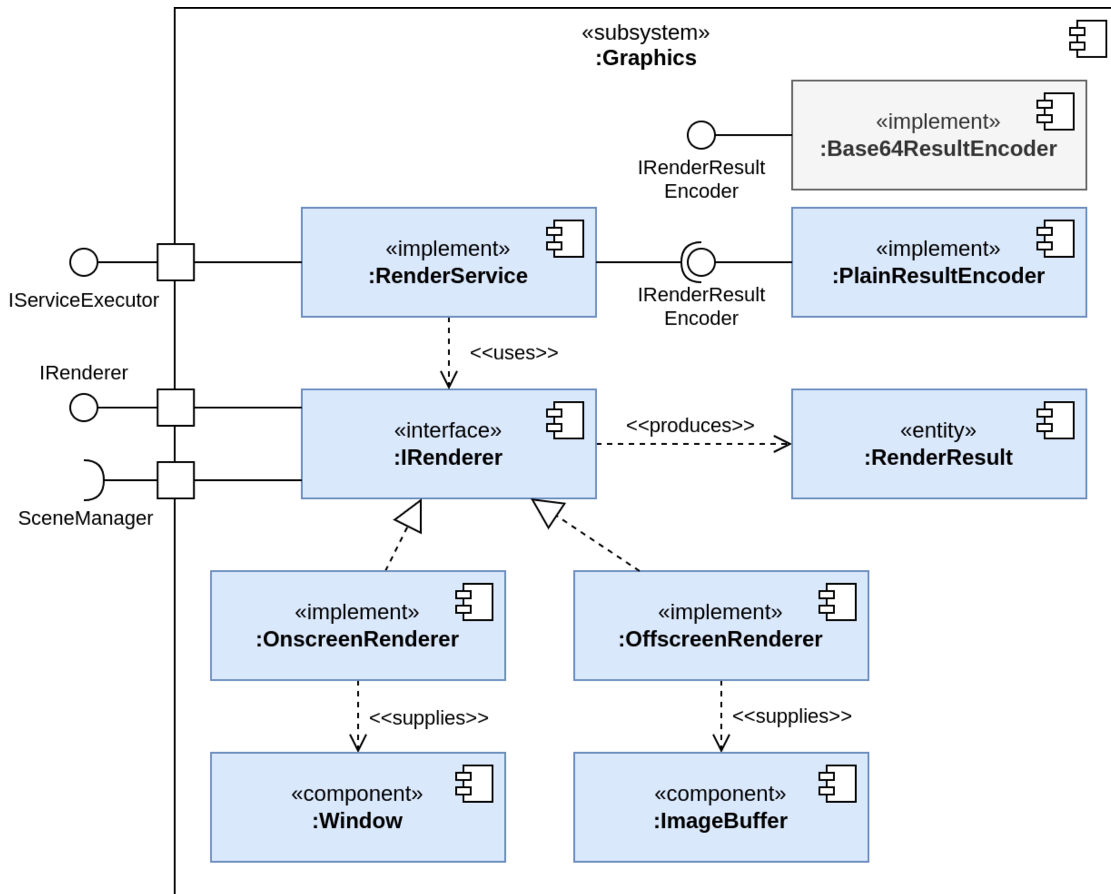


Figure 6.11: Software component design of the graphics subsystem.

objects, simplifying their employment and reducing development times. The onscreen renderer was not difficult to implement. Creating windows for various platforms and displaying rendered frames on them is the library's go-to behavior. The *OncreenRenderer* implementation contains only about 120 lines of code. On the other hand, the *Offscreen-Renderer* is more cumbersome and comprises about 800 lines. It requires the setup of image buffers, custom commands for copying the frame buffer's contents to them, and a custom render pass for Vulkan's rendering pipeline.

Figure 6.12 shows the extraction process of rendered images. The *OffscreenRenderer* creates a new render pass, including a custom frame buffer for Vulkan's rendering pipeline during its setup. After a new frame is rendered to the frame buffer, its contents are copied to a separate image buffer in step 1 using custom commands. Before this can happen, Vulkan requires transitioning the layout of the source and destination image buffers to one optimized for efficiently performing copy operations. The first copy step is embedded into the command buffer of the render pass and executed for every frame. Therefore, the image buffer always holds the last generated image of the *OffscreenRenderer*. A similar process is performed for the depth buffer's extraction. Yet, the copied image remains on the GPU, called *device memory* in Vulkan terms. In step 2, the renderer transfers its contents to *host memory*, which is a byte array on the node's heap. This is essential for accessing the image's bytes and using them outside the rendering pipeline, like sending

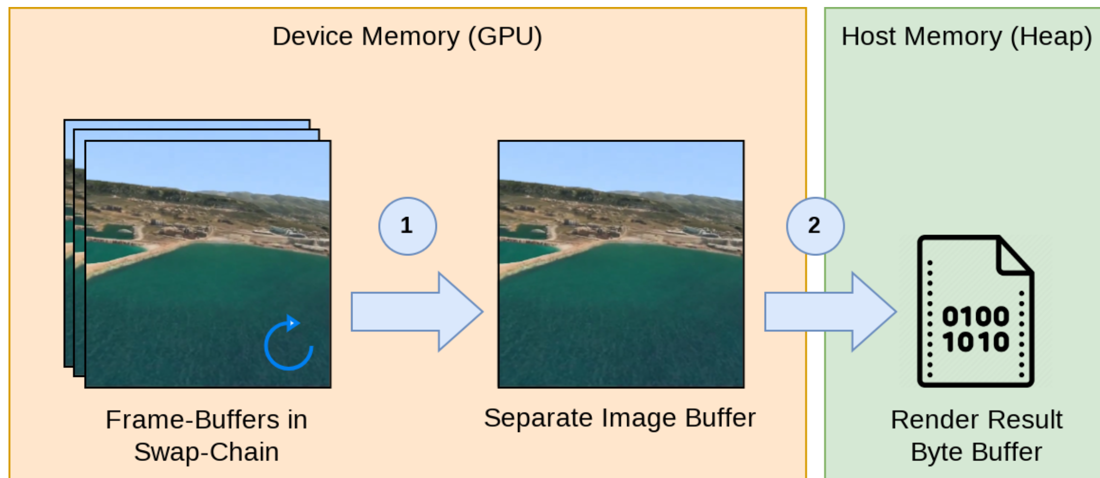


Figure 6.12: Buffers used for capturing rendered frames for offscreen rendering.

them to other nodes in the cluster. The second step is only carried out when a component requests the *RenderResult* to save significant delays caused by transferring the image from device memory to host memory.

When a node needs to generate images continuously without a preceding request to the rendering service, it can schedule a job timed at fixed intervals for this purpose. For instance, a job executed every 16 milliseconds limits the rate to 60 frames per second. The other alternative, as already mentioned, is requesting the rendering service. Due to the networking subsystem's capability to transmit raw bytes, no additional encoding, like Base64, must be applied to the image buffer. However, the *IRenderResultEncoder* interface can support compression algorithms for faster network transfer of large images, such as gzip or other post-processing methods in the future.

6.9 Plugin Management

The plugin-centered design (⇒ Section 5.1.1) requires a separate subsystem for loading, unloading, and managing plugins. In contrast to subsystems or modules, which are linked at compile-time, plugins are loaded at runtime. It must be the highest-level module on top of the subsystem stack to provide access to all others. There are multiple ways of implementing plugins. Scripting languages, like Python or Lua, are commonly used to extend a core's functionality. Scripted plugins are cross-platform and written in higher-level languages, reducing their development complexity and rendering the API's usage more convenient for developers unfamiliar with the core [22, p. 329f]. However, it requires a binding wrapping of the C++ core to establish compatibility in such a language. Due to the time constraints of this thesis, this relatively complex method was avoided. Instead, as shown in figure 6.13, the *PluginManager* loads compiled C++ code as dynamic libraries at runtime. They contain implementations of the abstract base class *IPlugin* that the manager instantiates and operates. As demonstrated in code snippet 3, it must override an initialization and shut-down routine that the manager calls when loading and unloading the plugin, respectively.

However, the chosen approach also has some downsides. First of all, C++ does not pro-

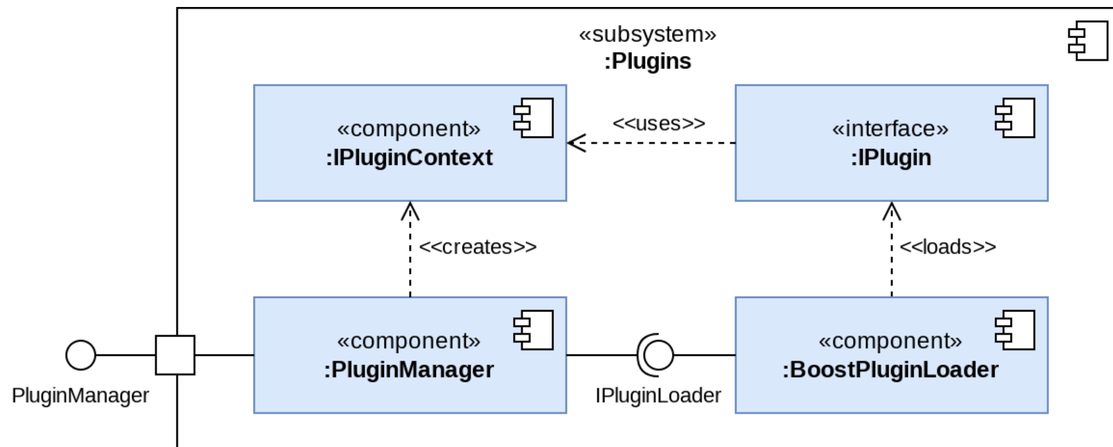


Figure 6.13: Software component design of the plugin management subsystem.

vide a unified Application Binary Interface (ABI). Artifacts, such as dynamic libraries of plugins, created using different compilers or other versions than the core system may be incompatible. In this case, the core fails to load them. Therefore, the core and its plugins should share the same compiler and version. A script binding would circumvent this issue and simplify the plugin distribution process. For the same reason, throwing exceptions or using the standard library across artifacts generated by different compilers is not guaranteed to work. Furthermore, memory management should not be mixed between the core and a loaded plugin. A plugin allocating a block of memory must also be the one to free it [22, p. 366f].

```
1  class SomePlugin : public plugins::IPlugin {
2      // used as unique identifier of the plugin
3      std::string GetName() override { return "some-plugin"; }
4
5      // called once the plugin has been loaded from a library
6      void Init(plugins::SharedPluginContext context) override {
7          auto subsystems = context->GetCoreSubsystems();
8          /* Schedule jobs, register event listeners, provide services,
9          add properties, register network message consumers, ... */
10         LOG_INFO("SomePlugin has been initialized");
11     }
12
13     // called before the plugin will be unloaded
14     void ShutDown(plugins::SharedPluginContext context) override {
15         LOG_INFO("SomePlugin has been shut down");
16     }
17 }
```

Listing 3: Plugin implementation that can be exported to a dynamic library and loaded by the core at runtime.

7 Evaluation by Example: Tile-based Rendering Plugin

To evaluate the core system and the framework it's providing, it must be put to use. A plugin using nearly all APIs provided by the infrastructure has been developed and allows for stress tests. Therefore, the target plugin should incorporate rendering tasks, the service infrastructure, scheduling jobs, connecting to other nodes, and contributing to the event flow. Distributed rendering (⇒ Section 3.2.4) is an ideal use case for this hybrid architecture. This plugin's main purpose is to demonstrate the core frameworks's flexibility and extensibility, not necessarily to be used in production.

The following section first discusses how this plugin works and puts distributed rendering into practice. Afterwards, the performance of the core infrastructure is evaluated using the plugin.

7.1 Plugin Design and Implementation

As already covered, there are multiple specializations of distributed rendering. This plugin subdivides the render area into a set of tiles and distributes the task of rendering them to connected nodes. The count of tiles is bound to the count of rendering services available at any moment. Therefore, the tiling setup is dynamic and must adapt when a render service joins or exits the cluster. For instance, three connected nodes providing a render service each will result in three tiles. When two new services join, the tiling algorithm of the task distribution step (⇒ Section 3.2.4) is repeated. For simplicity, the employed tiling algorithm simply generates equally sized stripes.

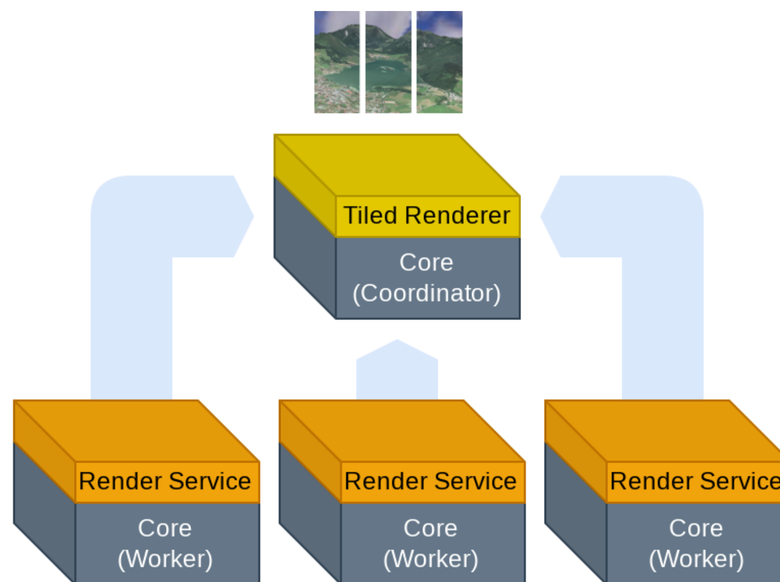
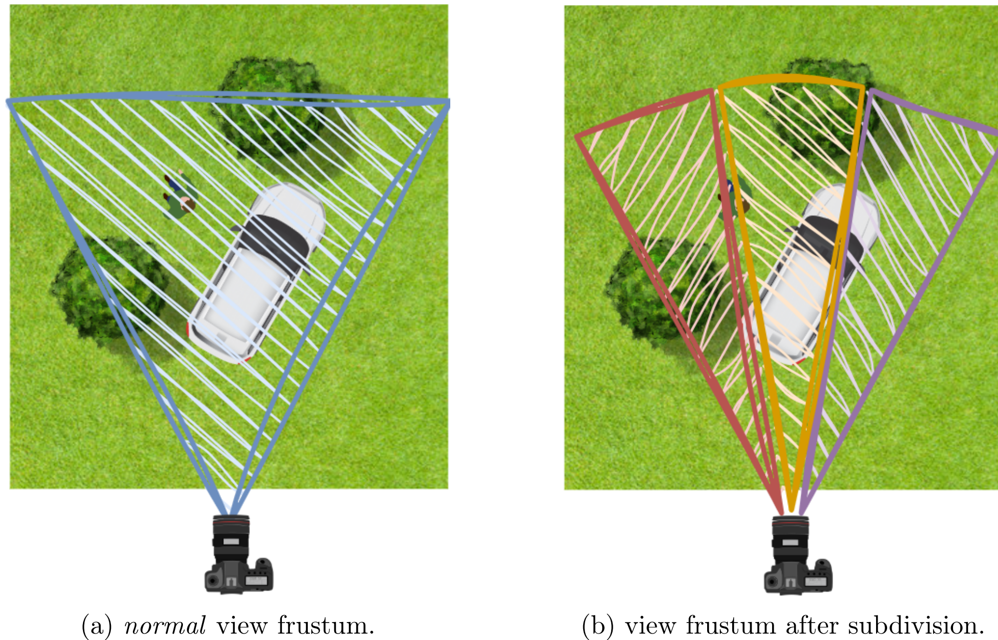


Figure 7.1: Tiled rendering plugin acts as coordinator and central composition agent, requesting strips of the same frame from different worker nodes.

As shown in figure 7.1, this plugin applies the coordinator-worker topology (⇒ Section 5.2.1). A single node runs the plugin, acting as a central composition agent, and awaits connections to worker nodes. They don't have to load the plugin themselves but simply provide a rendering service for the coordinator. Nodes must render these tiles from

slightly different view angles to generate images the coordinator can piece back together into a single frame. Therefore, the tiling algorithm must also re-calculate the perspective for each tile. Suppose the view of a rendered frame is displayed through a virtual camera. Its visible area seen in figure 7.2a is called its *frustum*. It defines the views' transformation and perspective attributes required for setting up the virtual camera before rendering [15, p. 227f].



(a) *normal* view frustum. (b) view frustum after subdivision.

Figure 7.2: Subdivision of the current view's frustum for three tiles.

In reality, the view is calculated by matrix multiplications on the GPU every frame. Two matrices are involved in this process and must be manipulated to achieve the tiling effect shown in figure 7.2b [15, p. 205]:

- The *view matrix* defines the virtual camera's position and orientation in the scene. It must be rotated to capture a different angle of the image.
- The *projection matrix* represents perspective attributes, like the view's field-of-view. It must be recalculated to scale down the frustum.

When rendering a tiled frame, the coordinator node re-calculates both the projection and view matrix for each tile. It passes them as request parameters to the designated worker's rendering service. Based on these parameters, each worker adjusts its camera, renders the tile, and sends it back to the coordinator. After it receives all tile images, it puts them back together by painting them on rectangular surfaces and arranging them like puzzle pieces. For this purpose, these tile images must be decoded and loaded into GPU memory as textures. An additional camera captures the rearranged frame and displays it on screen.

As the component diagram in figure 7.3 shows, his process requires extensive use of the core's subsystems:

- The plugin subsystem loads the plugin and installs it.

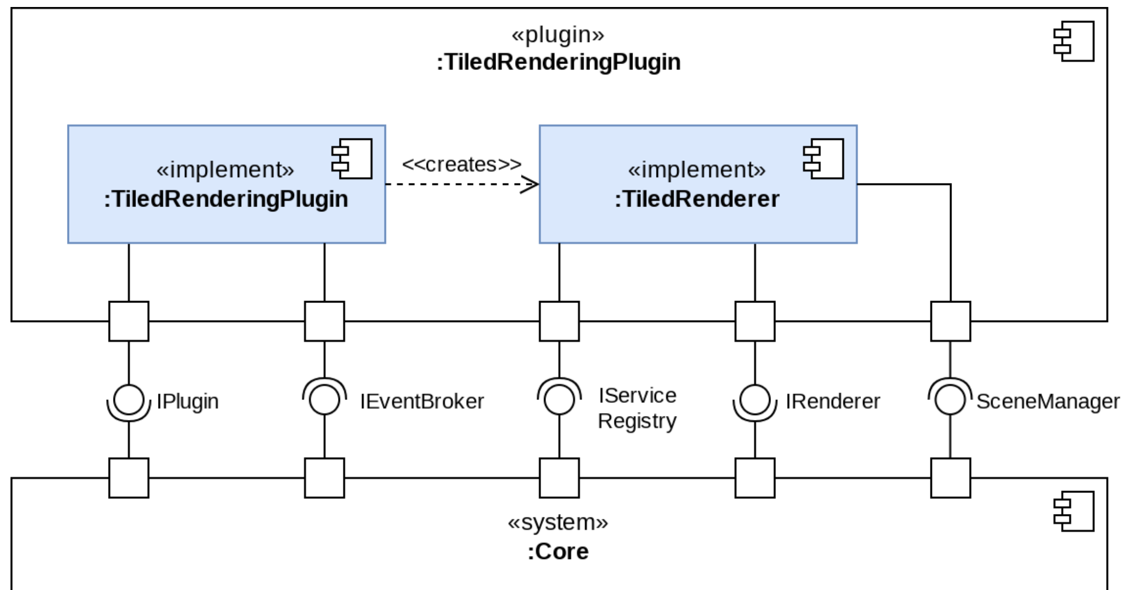


Figure 7.3: Software component design of the tiled rendering plugin.

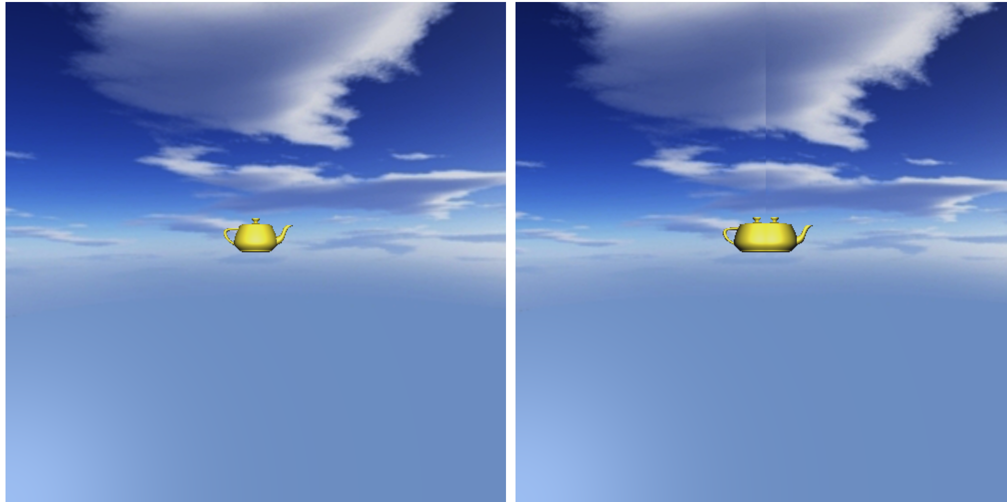
- A new *IRenderer* implementation, the *TiledRenderer*, is created and passed to the graphics subsystem. It queries the count of currently available render services to tile the render area accordingly.
- The event subsystem notifies the *TiledRenderingPlugin* about new render services in the cluster to re-configure the renderer's tiling layout.
- The service infrastructure provides render services to the *TiledRenderer* implementation.
- The scene manager supplies the current scene graph that should be rendered.

7.2 Operation

To create a cluster running and performing tiled-based rendering, the user must first start the coordinator node using the core's CLI (Command Line Interface) shown in listing 4. In this example, it listens for incoming connections on port 9000 and automatically spawns a window because on-screen rendering was requested. This node also provides a rendering service and should start rendering the scene immediately, as shown in figure 7.4a.

```
./standalone # standalone executable of the core
    -s ./data/models/scene.vsgt    # scene file to load (sample scene)
    -r onscreen                    # use on-screen renderer; spawns window
    --enable-render-job            # render automatically
    --enable-render-service        # setup render service
    -p <plugin-dir>                # loads plugins in this directory
    --port 9000                    # listen on port 9000
```

Listing 4: Bash command for starting the Coordinator node using the core's CLI.



(a) A single node and tile. (b) Using two nodes and tiles.

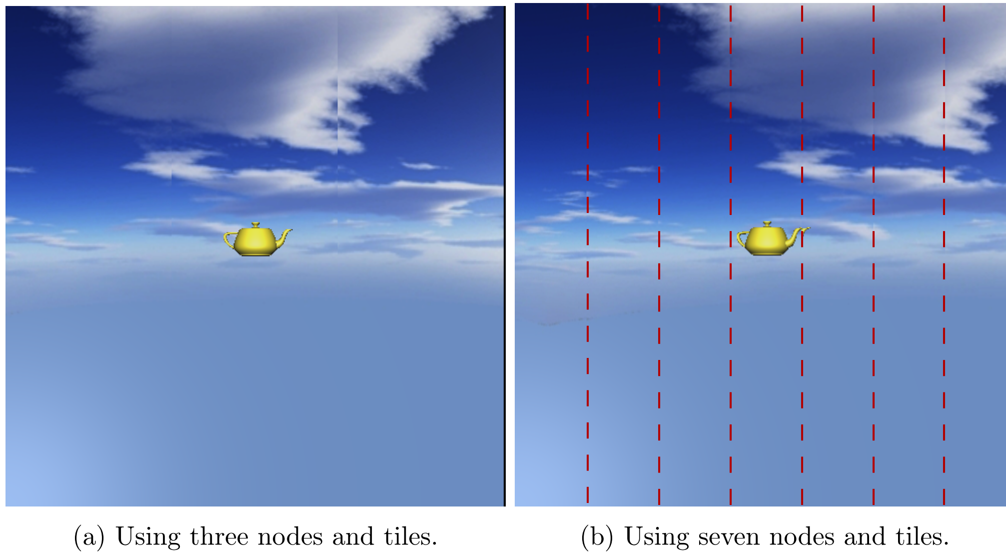
Figure 7.4: Tile-based rendered frame displayed on start-up and when connecting further nodes.

Once the central coordinator is running, worker nodes can join the cluster by connecting to it. It is important for a worker to load the same scene file as the central node. Otherwise, they would render different environments. An offscreen renderer is sufficient for these nodes because their image tiles are not displayed on screen. The user must also ensure that every worker node listens on another port to avoid crashes. Finally, the node tries to connect to the central coordinator and register its rendering service. When this operation is completed successfully, the central node will re-configure its tiling setup and include its new contributor. The cluster produces two tiles like in figure 7.4b. Their seam is visible sometimes because their frustum might overlap and not fit perfectly together. This is due to the improvised frustum subdivision algorithm, which is worthy of improvement in the future. However, it suffices to demonstrate the architecture's flexibility.

```
./standalone
    -s ./data/models/scene.vsgt    # loading the same scene
    -r offscreen                   # use off-screen renderer; no window
    --enable-render-service        # setup render service
    --port 9001                    # listen on port 9001
    --connect 127.0.0.1:9000       # connect to coordinator node
```

Listing 5: Bash command for starting a Worker node using the core's CLI.

An arbitrary amount of worker nodes can be spawned, as figures 7.5a and 7.5b show. Therefore, this plugin successfully demonstrates the flexibility and extensibility of the core framework for distributed computer graphics applications. The final implementation of this plugin comprises about 320 lines of code. Compared to core components, like the offscreen renderer, it is relatively small, simple in its structure, and lightweight by cleverly using underlying core facilities.



(a) Using three nodes and tiles. (b) Using seven nodes and tiles.

Figure 7.5: Tile-based rendered frame displayed by the coordinator employing varying counts of workers and tiles.

7.3 Performance Evaluation

To evaluate this plugin's implementation, purpose, and applicability from a performance-oriented point of view, various counts of worker nodes were advised to produce frames of increasing sizes. Their mean frame rate was measured and depicted in figure 7.6 in comparison to other counts of worker nodes.

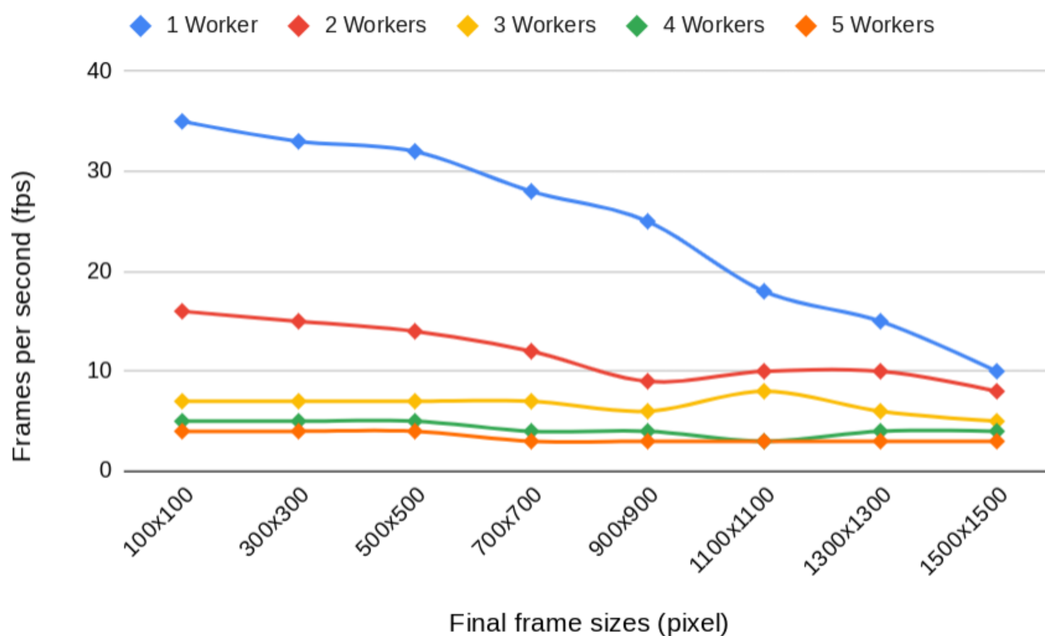


Figure 7.6: Frame-rates of different counts of worker nodes for increasing frame sizes using the tile-based rendering plugin.

Since there is no reason to embellish the results of this benchmark, it must be clearly stated that they are very insufficient for this work's performance requirements (⇒ Section 4.3.2). Additional nodes increase the communication overhead and accumulate networking costs, which is essentially the latency of streaming images to the coordinator. But considering one of the fundamental laws in distributed computing, which has been briefly discussed in section 2.2.2, these results are as expected. A bottleneck relocated from the processors to their communication network can worsen the overall efficiency in some scenarios. In this one, there was initially no bottleneck because a single processor can easily handle the rendering workload of this demonstration scene and produce at least 60 frames per second. Therefore, the act of applying distribution anyway created unnecessary overhead. This plugin emphasizes the importance of carefully assessing the underlying use case before blindly applying distribution. On the other hand, when employing more costly rendering techniques, like raytracing, this plugin can be more effective.

As already mentioned, the main purpose of this plugin is to demonstrate the flexibility and versatility of the core infrastructure and its hybrid architecture, not the efficiency of tiled-based rendering.

8 Conclusion

8.1 Summary

This work introduced and implemented a hybrid software architecture capable of connecting to other running instances. It can form clusters of nodes sharing their overall workload and distributing the synthetic environment generation process onto multiple hosts. The service infrastructure allows nodes to outsource their tasks to others and share their workload. They synchronize shared data and global state by publishing events to fellow nodes. Clusters are not limited to a single topology but can implement both P2P and centralized architectures, like the coordinator-worker pattern. This implementation's major strength is its ability to adapt to a wide range of use cases or future technologies by loading specialized capabilities in the form of plugins as needed. This also allows third-party engineering departments or other organizations to tailor the system to their individual needs by deploying custom plugins. Although this architecture allows the distribution of workloads, its target use cases must be assessed and its application carefully considered. It is not an *everything solver*, but only brings advantages to scenarios where the relocated communication bottleneck is a lesser evil and effectively relieves processors. As the example plugin demonstrates, this architecture fails in all other scenarios, not meeting this criteria. Due to communication bottlenecks and network latencies, real-time rendering processes are difficult to distribute onto multiple nodes without violating their deadlines and frame-rate expectations. However, more costly rendering techniques, like raytracing, could be promising candidates for this architecture.

8.2 Future Work

The question arises as to which scenarios and use cases this architecture is applicable and for what it will be used at MBDA. A few example use cases qualify for distribution and this architecture. The first involves distributed raytracing for generating large synthetic radar images. The currently used system runs on a single machine and needs almost half a minute to generate a single frame due to costly rendering techniques, causing a processor bottleneck. Using the tile-based rendering plugin, nodes can share their ray tracing workload and generate big images more efficiently. Another application that can be optimized using this architecture utilizes so-called Monte-Carlo simulations. Simulation parameters, like weather conditions, target positions, or the missile's starting point, are randomly altered and mutated to generate an arbitrary amount of new test scenarios from an original one. Currently, these outcoming simulation scenarios execute sequentially on a single machine. Depending on their amount, this process may take many hours and run overnight. By distributing these simulations onto multiple nodes for parallel execution, engineers can perform Monte-Carlo simulations in smaller timeframes and leverage their results sooner. Another use case is provisioning a simulation cluster in a scalable cloud environment called *simulation-as-a-service*. Using network protocols, engineers can request validations of their prototypes, algorithms, and behaviors from a remote cluster that is able to spawn new nodes and scale on demand.

Besides applying this architecture to specific use cases, its implementation can also be improved in the future. Further protocols and message formats, besides web sockets and multipart form data, need evaluation and can potentially contribute to the core infrastructure. As already mentioned, DDS and WebRTC are promising candidates for

such an evaluation and may substitute the custom web-socket implementation for inter-node communication. On the other hand, communication from or to external endpoints, like web clients, is not properly supported yet. The REST protocol can be included in the networking subsystem and service infrastructure for this purpose. This is useful for *simulation-as-a-service* applications leveraging this architecture. Furthermore, fault-tolerance measures are required for production-ready solutions but have not yet been implemented. Handling crashed nodes that were processing pending service requests or interrupted connections is essential. Such failures could gradually and recursively freeze the entire cluster when nodes wait indefinitely for unresolvable service responses.

Integration and synergies with other technologies or systems are other means for allowing this architecture to mature. Containerization of nodes can enable the use of orchestrators like Kubernetes and leverage cloud-native features: An automatically scaling, self-healing, and zero-downtime simulation cluster. Of course, further evaluation efforts are required to integrate such a technology successfully. Especially since cloud-native containers must not maintain an internal state in order to leverage the above-mentioned capabilities. Instead of repairing, re-using, or moving them, Kubernetes straightforwardly destroys and re-spawns containers elsewhere, rendering them ephemeral. On everyday operations, like transferring running simulation nodes to other machines for load-balancing purposes, Kubernetes would also destroy their synthetic environment's state [19, p. 3f, 27f]. Adding support for some persistent database, like the CDB (⇒ Section 5.2.2), would transfer state management to a central facility and allow nodes to leverage cloud-native capabilities. As already mentioned, this assumption requires further evaluation in future works.

References

[1] Tomas Akenine-Mller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Fourth Edition.* 4th. USA: A. K. Peters, Ltd., 2018. ISBN: 0134997832.

[2] G. Bengel. *Grundkurs Verteilte Systeme: Grundlagen und Praxis des Client-Server und Distributed Computing.* Springer Fachmedien Wiesbaden, 2015. ISBN: 9783834821508. URL: https://books.google.de/books?id=GG_dBgAAQBAJ.

[3] Marc Carrascosa and Boris Bellalta. "Cloud-gaming: Analysis of Google Stadia traffic". In: *Computer Communications* 188 (2022), pp. 99–116. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2022.03.006. URL: https://www.sciencedirect.com/science/article/pii/S0140366422000810.

[4] J.A. Crowder and C.W. Hoff. *Requirements Engineering: Laying a Firm Foundation.* Springer International Publishing, 2022. ISBN: 9783030910778. URL: https://books.google.de/books?id=ZJpXEAAAQBAJ.

[5] Leah Emerson and Thomas Marrinan. "Real-Time Compression of Dynamically Generated Images for Offscreen Rendering". In: *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV).* 2019, pp. 91–92. DOI: 10.1109/LDAV48142.2019.8944381.

[6] Alberto Falcone, Alfredo Garro, Anastasia Anagnostou, and Simon J.E. Taylor. "An introduction to developing federations with the High Level Architecture (HLA)". In: *2017 Winter Simulation Conference (WSC).* 2017, pp. 617–631. DOI: 10.1109/WSC.2017.8247820.

[7] C. Glez-Morcillo, D. Vallejo, J. Albusac, L. Jimenez, and J.J. Castro-Schez. "A New Approach to Grid Computing for Distributed Rendering". In: *2011 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing.* 2011, pp. 9–16. DOI: 10.1109/3PGCIC.2011.12.

[8] J. Gregory. *Game Engine Architecture, Third Edition.* CRC Press, 2018. ISBN: 9781351974288. URL: https://books.google.de/books?id=1g1mDwAAQBAJ.

[9] Jan Hodicky. "Modelling and Simulation in the Autonomous Systems' Domain – Current Status and Way Ahead". In: *Modelling and Simulation for Autonomous Systems.* Ed. by Jan Hodicky. Cham: Springer International Publishing, 2015, pp. 17–23. ISBN: 978-3-319-22383-4.

[10] Martin Hoppen, Ralf Waspe, Malte Rast, and J. Rossmann. "Distributed Information Processing and Rendering for 3D Simulation Applications". In: *International Journal of Computer Theory and Engineering* 6 (Feb. 2014), pp. 247–253. DOI: 10.7763/IJCTE.2014.V6.870.

[11] Prabhjot Kaur, Samira Taghavi, Zhaofeng Tian, and Weisong Shi. "A Survey on Simulators for Testing Self-Driving Cars". In: *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD).* 2021, pp. 62–70. DOI: 10.1109/MetroCAD51599.2021.00018.

[12] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. "Robot Operating System 2: Design, architecture, and uses in the wild". In: *Science Robotics* 7.66 (2022), eabm6074.

[13] L Masinter. *RFC2388: Returning Values from Forms: multipart/form-data*. 1998.

[14] A. Nischwitz, M. Fischer, P. Haberäcker, and G. Socher. *Bildverarbeitung: Band II des Standardwerks Computergrafik und Bildverarbeitung*. Springer Fachmedien Wiesbaden, 2020. ISBN: 9783658287047. URL: `https://books.google.de/books?id=3tiCywEACAAJ`.

[15] A. Nischwitz, M. Fischer, P. Haberäcker, and G. Socher. *Computergrafik: Band I des Standardwerks Computergrafik und Bildverarbeitung*. Springer Fachmedien Wiesbaden, 2019. ISBN: 9783658253844. URL: `https://books.google.de/books?id=8J2SDwAAQBAJ`.

[16] A. Ostrowski and P. Gaczkowski. *Software Architecture with C++: Design modern systems using effective architecture concepts, design patterns, and techniques with C++20*. Packt Publishing, 2021. ISBN: 9781789612462. URL: `https://books.google.de/books?id=GrAmEAAAQBAJ`.

[17] M. G. Perhinschi, M. R. Napolitano, and S. Tamayo. "Integrated Simulation Environment for Unmanned Autonomous Systems: Towards a Conceptual Framework". In: *Model. Simul. Eng.* 2010 (Jan. 2010). ISSN: 1687-5591. DOI: `10.1155/2010/736201`. URL: `https://doi.org/10.1155/2010/736201`.

[18] Victoria Pimentel and Bradford G. Nickerson. "Communicating and Displaying Real-Time Data with WebSocket". In: *IEEE Internet Computing* 16.4 (2012), pp. 45–53. DOI: `10.1109/MIC.2012.64`.

[19] N. Poulton. *The Kubernetes Book*. NIGEL POULTON LTD, 2023. ISBN: 9781916585195. URL: `https://books.google.de/books?id=enTJEAAAQBAJ`.

[20] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009.

[21] R. Rangel-Kuoppa, C. Aviles-Cruz, and D. Mould. "Distributed 3D rendering system in a multi-agent platform". In: *Proceedings of the Fourth Mexican International Conference on Computer Science, 2003. ENC 2003.* 2003, pp. 168–175. DOI: `10.1109/ENC.2003.1232891`.

[22] M. Reddy. *API Design for C++*. Elsevier Science, 2011. ISBN: 9780123850041. URL: `https://books.google.de/books?id=IY29LylT85wC`.

[23] M. Richards and N. Ford. *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, 2020. ISBN: 9781492043423. URL: `https://books.google.de/books?id=xa7MDwAAQBAJ`.

[24] Sara Saeedi, Steve Liang, David Graham, Michael F. Lokuta, and Mir Abolfazl Mostafavi. "Overview of the OGC CDB Standard for 3D Synthetic Environment Modeling and Simulation". In: *ISPRS International Journal of Geo-Information* 6.10 (2017). ISSN: 2220-9964. DOI: 10.3390/ijgi6100306. URL: https://www.mdpi.com/2220-9964/6/10/306.

[25] J.M. Schlesselman, G. Pardo-Castellote, and B. Farabaugh. "OMG data-distribution service (DDS): architectural update". In: *IEEE MILCOM 2004. Military Communications Conference, 2004.* Vol. 2. 2004, 961–967 Vol. 2. DOI: 10.1109/MILCOM.2004.1494965.

[26] Cristina Schmidt and Manish Parashar. "A peer-to-peer approach to web service discovery". In: *World Wide Web* 7 (2004), pp. 211–229.

[27] G. Sellers and J.M. Kessenich. *Vulkan Programming Guide: The Official Guide to Learning Vulkan.* Always learning. Addison-Wesley, 2017. ISBN: 9780134464541. URL: https://books.google.de/books?id=kUJujwEACAAJ.

[28] Anton Sigitov, Thorsten Roth, Florian Mannuss, and Andre Hinkenjann. "DRiVE: An example of distributed rendering in virtual environments". In: *2013 6th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (SEARIS).* 2013, pp. 33–40. DOI: 10.1109/SEARIS.2013.6798106.

[29] M.R. Stytz. "Distributed virtual environments". In: *IEEE Computer Graphics and Applications* 16.3 (1996), pp. 19–31. DOI: 10.1109/38.491182.

[30] Blender Documentation Team. *Scripting and Extending Blender: Introduction.* Jan. 2024. URL: https://docs.blender.org/manual/en/latest/advanced/scripting/introduction.html (visited on 01/14/2024).

[31] Unity Technologies. *Unity Plug-ins.* Jan. 2024. URL: https://docs.unity3d.com/Manual/Plugins.html (visited on 01/14/2024).

[32] Okan Topçu, Umut Durak, Halit Oğuztüzün, and Levent Yilmaz. *Distributed Simulation - A Model Driven Engineering Approach.* Feb. 2016. ISBN: 978-3-319-03050-0. DOI: 10.1007/978-3-319-03050-0.

[33] Q.H. Vu, M. Lupu, and B.C. Ooi. *Peer-to-Peer Computing: Principles and Applications.* Springer Berlin Heidelberg, 2009. ISBN: 9783642035142. URL: https://books.google.de/books?id=kd8_AAAAQBAJ.

[34] *Wehrtechnische Dienstelle 81.* URL: https://www.bundeswehr.de/de/organisation/ausruestung-baainbw/organisation/wtd-81 (visited on 11/05/2023).

[35] Huabing Zhu, Kai Yun Chan, Lizhe Wang, and Wentong Cai. "DPBP: a sort-first parallel rendering algorithm for distributed rendering environments". In: *Proceedings. 2003 International Conference on Cyberworlds.* 2003, pp. 214–220. DOI: 10.1109/CYBER.2003.1253457.