Matheus Vinícius Teotonio do Nascimento Andrade

Architecture, Implementation and Performance Analysis of a Microservices-Based System for the RoboCup Small Size League

Federal University of Pernambuco
secgrad@cin.ufpe.br
www.cin.ufpe.br/~graduacao

Recife
2024

Matheus Vinícius Teotonio do Nascimento Andrade

Architecture, Implementation and Performance Analysis of a Microservices-Based System for the RoboCup Small Size League

A B.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering.

Concentration Area: *Software Engineering*
Advisor: *Augusto Cezar Alves Sampaio*

Recife
2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Andrade, Matheus Vinícius Teotonio do Nascimento.
Architecture, Implementation and Performance Analysis of a Microservices-Based System for the RoboCup Small Size League / Matheus Vinícius Teotonio do Nascimento Andrade. - Recife, 2024.
77 : il., tab.

Orientador(a): Augusto Cezar Alves Sampaio
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2024.
Inclui referências, apêndices.

1. Microservices architecture. 2. RoboCup Small Size League. 3. Robotic systems. 4. Distributed systems. 5. Event-Driven Architecture. I. Sampaio, Augusto Cezar Alves. (Orientação). II. Título.

000 CDD (22.ed.)

Matheus Vinícius Teotonio do Nascimento Andrade

Architecture, Implementation and Performance Analysis of a Microservices-Based System for the RoboCup Small Size League

A B.Sc. Dissertation presented to the Center of Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Bachelor in Computer Engineering. Defended and approved on October 22, 2024 by the following board of examiners:

Augusto Cezar Alves Sampaio
Advisor/CIn-UFPE

Vinicius Cardoso Garcia
Examiner/CIn-UFPE

ACKNOWLEDGEMENTS

Attending university is far from simple. The road to learning is arduous, whether starting school, passing the entrance exam, graduating from university, or any other learning process. However, the steps become lighter and more secure when we have people who love and support us. I was lucky and blessed to have great people by my side during my journey. I want to thank these people for all their support. Thank God for allowing me to live what I dream of and have the essential people by my side.

The main thanks go to my family, my beloved parents, Elda and Marcelo, my references and inspirations for everything I do. I am incredibly grateful to my brother, Marcelo Júnior (Juninho), for all the moments we shared while we were able to live together and enjoy life. I know you're watching over us and shining down from heaven, and I'm sure you're happy and proud of my achievements and everything we've experienced since you left. We miss you and will never forget you. Your name and your life are in everything we do. Thank you to my beloved little dog, Luly, for all her love and affection, especially for our mother. To you, my lovelies, thank you so much for all the teachings, for all the hugs, for all the affection, for all the support. Thank you so much for all the love we've experienced. I love you more than anything, and I will be eternally grateful to God for the family I have been given.

I'm incredibly grateful to the family I've gained over the years, especially my great love, Maria Eduarda. Thank you so much for all your support and love during my journey from school to graduation, for calming me down and making me see that I can achieve everything we dream of. Thank you for being my best friend and the best companion I could ever have; I love you so much. Thank you to the family I gained with you, my parents-in-law, Gildeci and Hortênsia, and my sister-in-law, Maria Alyne. You were fundamental to my career choice and development, including during my studies at the Escola Santa Maria and the Curso Solução, which were essential to my growth as a student and gave me the resources to enter university.

I would also like to thank my aunt and uncle, Naide and Misso, who became parents during my degree, giving me a home to live in during university and love and support throughout the process. Also, I would like to thank my cousins, Vitinho and Peu, for all the moments we shared during the course and for all the laughs and jokes that, directly or indirectly, made graduation lighter.

I want to thank Centro de Informática, my home for almost seven years, for providing the resources, programs, and people essential for my personal and professional development while studying Computer Engineering. In this context, I would like to thank you for the different projects I participated in, which made me see the world differently with each learning experience. First of all, many thanks to RobôCIn, especially Lucas Cavalcanti, Roberto Fernandes, and Riei Joaquim, for being by my side in research, development, competitions, and achievements with SSL, building friendships beyond RobôCIn. Thanks also to the Apple De-

veloper Academy, all the mentors, students, and staff who are part of this project, which has become a turning point in my academic life and has made me grow immensely, making me see my ability and passion for developing projects with purpose. Many thanks also to INES, especially Professor Augusto Sampaio, Madiel Conserva, and Filipe Arruda, as well as to the team made up of Andresa Almeida, Alexandre Burle, José Victor (Zé), Karine Gomes, and Uanderson Ricardo (Sonson), for the support, mentoring, and discussions that made this work possible.

Finally, I couldn't fail to thank the people who helped me in my day-to-day life at university, my friends and classmates, who always made classes, studies, exams, and assignments lighter with their jokes and support during the university time. A special thanks to my friends who were there for me from the beginning to the end of my degree and outside the university. Thank you so much for everything, Burle, Humberto (Biro), Rodrigo (Rod), Uanderson (Sonson), and Vaz. It's always great to be with you, inside or outside the CIn.

Ouvi palavras que me deram mais poder
O seu espírito está entre nós
Não vamos te esquecer, não vamos nos perder
Nossa família agora é o meu bem maior
— Pra Sempre – CPM22

ABSTRACT

The RoboCup Small Size Soccer League (SSL) is a robot soccer competition that demands extremely low latency software systems due to the fast-paced nature of the matches. Traditionally, teams in this category have relied on monolithic software architectures. However, these architectures often struggle with scalability, maintenance, and flexibility issues, making evolution extremely hard. A viable alternative is to adopt a microservices approach, which enables the system to be more modular and adaptable.

This project proposes to design a microservices-based architecture tailored for SSL, with the goal of achieving modularity, efficiency and low latency. Additionally, the project implements and analyzes the communication overhead introduced by the distributed nature of the proposed architecture.

Keywords: Microservices architecture. RoboCup Small Size League. Robotic systems. Distributed systems. Event-Driven Architecture

RESUMO

A RoboCup Small Size Soccer League (SSL) é uma competição de futebol de robôs que exige sistemas de software de latência extremamente baixa devido à dinâmica acelerada das partidas. Tradicionalmente, as equipes dessa categoria têm utilizado arquiteturas de software monolíticas. No entanto, essas arquiteturas frequentemente enfrentam problemas de escalabilidade, manutenção e flexibilidade, o que torna a evolução extremamente difícil. Uma alternativa viável é a adoção de uma abordagem baseada em microsserviços, que permite que o sistema seja mais modular e adaptável.

Este projeto propõe modelar uma arquitetura baseada em microsserviços aplicada ao SSL, com o objetivo de alcançar modularidade, eficiência e baixa latência. Além disso, o projeto implementa e analisa a sobrecarga de comunicação introduzida pela natureza distribuída da arquitetura proposta.

Palavras-chave: Arquitetura de microsserviços. RoboCup Small Size League. Sistemas robóticos. Sistemas Distribuídos. Arquitetura Dirigida a Eventos

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

BPMN	Business Process Model and Notation
DDD	Domain-Driven Design
EDA	Event-Driven Architecture
ESB	Enterprise Service Bus
gRPC	Google Remote Procedure Call
MOM	Message-Oriented Middleware
MRS	Multi-Robot Systems
ROS	Robot Operating System
RPC	Remote Procedure Calls
SCA	Service Component Architecture
SOA	Service-Oriented Architecture
SOC	Service-Oriented Computing
SRP	Single Responsibility Principle
SSL	Small Size League
SysML	Systems Modeling Language
UML	Unified Modeling Language
ZMQ	ZeroMQ

CONTENTS

1

INTRODUCTION

A monolithic architecture is a software design approach where all components and functionalities of an application are integrated into a single and cohesive unit. This design paradigm facilitates straightforward development and deployment processes, making it an attractive option for many early-stage projects and small-scale applications. In a monolithic architecture, all application's components, such as the user interface, business logic, and data access layer, are tightly coupled and run as a single process. While this approach simplifies initial development and offers high performance, it also has significant drawbacks.

The tightly coupled approach of monolithic systems compromises the independence of the components and, consequently, flexibility, modularity, and maintainability. Therefore, any change in a single module necessitates a full-scale redeployment of the entire application. This tends to slow down the development cycle and to increase the risk of introducing bugs and system failures. Additionally, scaling specific parts of the application independently is complex, often resulting in higher operational costs.

Many organizations are turning to microservices-based architecture to address these issues as a more scalable and flexible solution [54]. Unlike monolithic systems, a microservices architecture breaks down the application into smaller, loosely coupled services, each responsible for a specific functionality [6]. This modular approach allows for independent development, deployment, and scaling of services, enhancing agility and reducing the risk of system-wide failures. Additionally, microservices can leverage different technologies and frameworks best suited for their specific tasks, promoting innovation and technological diversity.

In the field of robotics competition, RoboCup [1] [38] stands out as a renowned international robotics competition that serves as a platform for advancing autonomous robotics and artificial intelligence. Within the RoboCup, the Small Size League (SSL) (Figure 1) is one of the oldest categories, utilizing intelligent, collaborative omnidirectional robots controlled in a highly dynamic environment. Compared to other RoboCup categories, the SSL imposes strict speed requirements due to the game's fast-paced nature and challenges in agility and decision-

[1] RoboCup is a world competition of several modalities of autonomous robots that have existed since 1997, which has the ambitious intention of developing a team of humanoid robots capable of defeating the most recent FIFA World Cup champion in the middle of 21st century [28].

making, compelling teams to design robots capable of navigating dynamic environments while executing complex strategies.

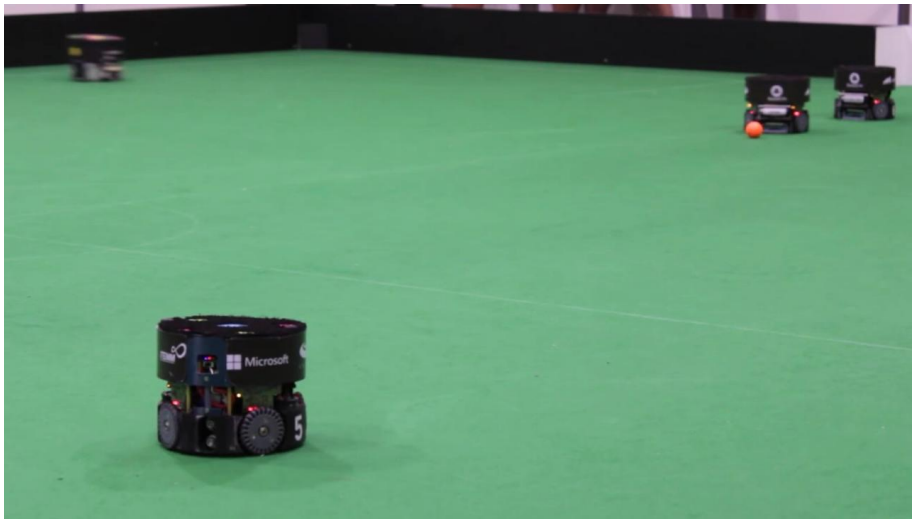

Figure 1: SSL robots in RoboCup field during a real game. Source: The author.

Traditionally, SSL teams have relied on monolithic software architectures to control their robots [59, 82, 3]. In these systems, parallel execution is achieved through state sharing and multi-threading, enabling fast response times and precise control over robot behavior. As the complexity of robotic systems increases and the demand for more sophisticated functionalities grows, monolithic designs can become unwieldy and difficult to manage. Additionally, the tightly coupled nature of these systems can hinder the adoption of emerging technologies and inhibit experimentation with alternative approaches, such as existing ones for rendering/visualization and user interaction, in which performance requirements are less restrictive. However, since they are coupled in the same application, they use frameworks in the same programming languages and technologies already used to develop the main requirements.

To overcome these challenges, microservices architectures, as previously mentioned, are a modern and promising solution for creating modular, flexible, and scalable software systems. Adopting this approach, however, requires careful design to handle component communication, concurrency, and synchronization, especially in systems with high parallelism.

This work proposes an architecture, implementation, and performance analysis of a microservices-based system applied to the RoboCup Small Size League soccer category. Despite the benefits of microservices, their implementation in real-time robotic applications, particularly those with strict latency requirements like SSL, has presented challenges. However, through careful design and optimization, the advantages of microservices can be leveraged without compromising performance. The viability and benefits of this alternative paradigm are demonstrated through experimentation and analysis, paving the way for future advancements in autonomous robotic systems within the RoboCup community.

1.1 OBJECTIVES

The main objective of this project is to study and propose an efficient, low-latency microservices-based architecture for a multi-robot system software applied in the RoboCup Small Size League. The specific objectives of the project are:

- Modeling the architecture components, defining the microservices, and the interactions between the modules that compose the system following Domain-Driven Design principles with informal diagrams and the Unified Modeling Language.
- Implementing a subset of the final control system and the base structure of the architecture components to ensure scalable integration of the microservices.
- Evaluating, via experimentation, the performance impact due to components communication of the microservices-based architecture.

This work is divided into 6 chapters. Chapter 2 explores the necessary background for understanding the development of software architectures, covering various architectural paradigms and software modeling techniques. Chapter 3 delves into the work proposal, exploring the RoboCup SSL and robotics systems focused on the development of the SSL VAR system. Chapter 4 analyzes the performance impacts of communication and processing within the microservice architecture pipeline, with a particular focus on latency in real-time scenarios relevant to SSL. Chapter 5 provides a review of related works, exploring the context of existing research and developments in the robotics field. Chapter 6 concludes the study, summarizing the findings and implications for future work in advancing robotic systems through microservices architectures.

2

BACKGROUND

This chapter provides the necessary background for understanding the development and modeling software architectures. The chapter introduces various software architecture paradigms, including monolithic, distributed, service-oriented, microservices, and event-driven architectures, exploring each approach's benefits and downsides. Additionally, it delves into software modeling, focusing on microservices design and the use of the Unified Modeling Language to represent complex systems. This background lays the groundwork for the practical applications and case studies discussed in the following chapters.

2.1 SOFTWARE ARCHITECTURE

This section explores monolithic and distributed architectures, with a focus on Service-Oriented Architecture, microservices-based architecture, and Event-Driven Architecture. Additionally, the chapter explores strategies for each architectural style, emphasizing their respective benefits and drawbacks. It also discusses the role of message-oriented middleware in facilitating communication within microservices-based architectures, particularly those employing event-driven designs.

2.1.1 Overview

The software architecture of a system covers its structure, the architectural features it must support, the architectural decisions made during design, and the design principles that guide the overall development [63]. It acts as a high-level blueprint that defines a system's structure, behavior, and interactions between its components.

Software architecture deeply aligns with the organizational dynamics of development teams. Melvin Conway, a prominent figure in computer science, articulated an observation now known as Conway's Law [17], which posits that the design of systems by organizations inherently mirrors their communication structures. A development team's organizational layout and communication channels provide substantial influence over software systems' architectural decisions and outcomes. Understanding and effectively managing these dynamics is pivotal for

crafting cohesive and efficient software architectures, aligning with technical imperatives and organizational objectives.

Furthermore, the chosen architectural style significantly influences the communication patterns among system components. Architecture styles can be broadly classified into two main categories: monolithic and distributed [63]. Monolithic architectures consolidate all code into a single deployment unit, whereas distributed architectures consist of multiple deployment units connected via remote access protocols. Architectural paradigms prescribe how components discover one another, exchange data, and handle potential failures. Each style deeply impacts scalability, maintainability, and performance, necessitating careful consideration during architectural design.

A team structured into isolated development units tends to favor a monolithic architecture, where all functionalities are tightly integrated [25]. Even when the separation of concerns is taken into account (for instance, in a layered architecture), the organizational structure of a monolith naturally leads to the development of tightly coupled systems. In such environments, each development unit operates independently with minimal interaction with other teams, leading to a lack of collaboration and limited communication. This can hinder scalability and flexibility, as making changes or additions to the system requires understanding and modifying a large, interconnected codebase.

In 2003, Eric Evans introduced Domain-Driven Design (DDD) [23] as a response to the growing challenges posed by monolithic systems, by promoting better organization, maintainability, and scalability. DDD provides a base to implement a Service-Oriented Architecture (SOA), since both methodologies prioritize clear boundaries and separation of concerns. Companies that embrace distributed architectures often structure their teams around service boundaries rather than traditional technical partitions. This strategy, known as the Inverse Conway Maneuver [25], is beneficial because the team's structure significantly influences various aspects of software development and should align with the problem's size and scope. Figure 2 shows two types of high-level partitioning according to technical and domain aspects.

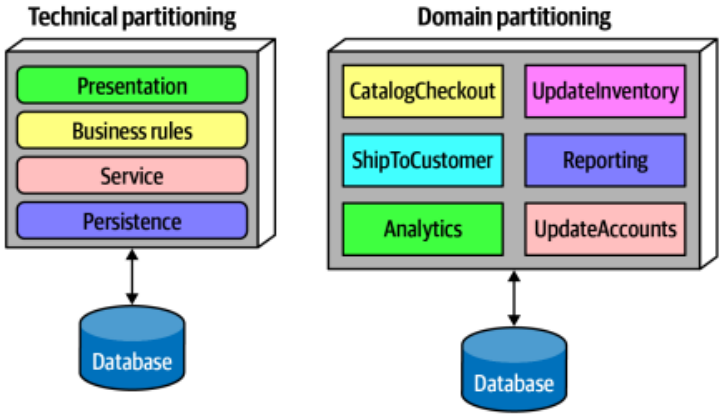

Figure 2: Two types of top-level partitioning in architecture. Source: Mark Richards and Neal Ford [63].

While Conway's Law serves as a guiding principle rather than a strict rule, it underscores the critical importance of considering the development team's structure throughout the architecture design process. A well-aligned architecture facilitates streamlined communication and collaboration within the team, contributing to the long-term maintainability and sustainability of the system.

Moreover, at the core of software architecture lies the system's structural organization, which dictates how the system is decomposed into manageable components. These components encapsulate specific functionalities and interact to deliver the overall system behavior. Decisions regarding these components' identification, size, and complexity are pivotal in shaping the architecture's effectiveness.

2.1.2 Monolith Architecture

Monolithic architecture traditionally refers to a software design where all components of an application -— such as the user interface, business logic, and data access layers -— are tightly integrated [54, 42]. Even though a monolith architecture can be modularized within a layered architecture, the organizational structure of a monolith naturally leads to the development of tightly coupled systems.

The monolithic approach has historically been favored for its simplicity in development and deployment processes. Besides, in terms of performance, monolithic applications benefit from all components sharing the same memory space, resulting in lower latency and higher efficiency. This unified structure allows for performance optimization. However, monolith architecture has significant downsides. It inherently lacks modularity, which restricts scalability and results in collective resource sharing within a single system [46]. Therefore, scaling the entire application rather than individual components can lead to inefficient resource utilization and increased operational costs [41]. As the application grows, resource bottlenecks may arise, degrading overall system performance. Additionally, the development, maintenance, and modification of monolithic applications become progressively more challenging and time-consuming due to their expanding size and complexity [76]. As the codebase grows, it becomes increasingly difficult to understand, maintain, and modify, leading to slower development processes and a higher risk of defects. Deployment risks also increase, as changes to a monolithic application can affect the entire system, requiring longer release cycles and more extensive testing.

Furthermore, monolithic architectures are inflexible, making it difficult to adapt to technological advancements and changing business requirements. Implementing changes often requires extensive modifications to the entire project, impeding agility. Large development teams working on the same code may also face coordination challenges, leading to potential conflicts and reduced productivity.

2.1.3 Distributed Architecture

A distributed architecture contrasts a monolithic design, comprising multiple services operating within their own ecosystems, communicating via networking protocols [63]. Distributed architectures offer significant advantages over monolithic and layered-based architectures, including better scalability, decoupling, and control over development, testing, and deployment. Components within a distributed architecture tend to be more self-contained, allowing for better change control and easier maintenance, which leads to more robust and responsive applications. Distributed architectures also lend themselves to more loosely coupled and modular applications.

2.1.3.1 Service-Oriented Architecture

A SOA can effectively address issues inherent in a monolithic architecture. SOA is a methodology within Service-Oriented Computing (SOC) [83], applying distinct software components to construct business applications. Each service represents a specific business capability and can communicate with other services across various platforms and programming languages. This approach allows developers to reuse services across different systems and combine multiple independent services to perform complex operations.

The core components of SOA include three key elements: the service implementation (the code performing specific functions like authentication or invoice calculation), the service contract (detailing the service's terms, conditions, prerequisites, and costs), and the service interface (facilitating communication between services and systems by specifying how to invoke the service). This design minimizes dependencies and allows users with a limited understanding of the underlying logic to utilize the service. In SOA, service providers create, maintain, and offer services, while service consumers request and utilize these services, which are governed by a service contract ensuring clear rules and expectations.

Services are accessed remotely from the user interface using protocols like REST [45], messaging [48], RPC [60], or SOAP [75]. An API layer consisting of a reverse proxy or gateway between the user interface and services can be incorporated (Figure 3). This approach brings together common shared concerns like metrics, security, and service discovery, and moves them away from the user interface.

At the beginning of SOA in the 1990s, the core was centered on enterprise-level reuse, aiming to reduce software rewriting through strategic architectural structuring [63]. This approach organized various layers to achieve this goal. Business services, which define domain behaviors without containing code, are typically defined by business users. Enterprise services are fine-grained, shared implementations designed to build atomic behaviors around specific business domains. These enterprise services aim to prevent rewriting parts of the business workflow and accumulating reusable assets over time. However, the dynamic nature of business requirements often complicates this goal. Application services address specific needs without

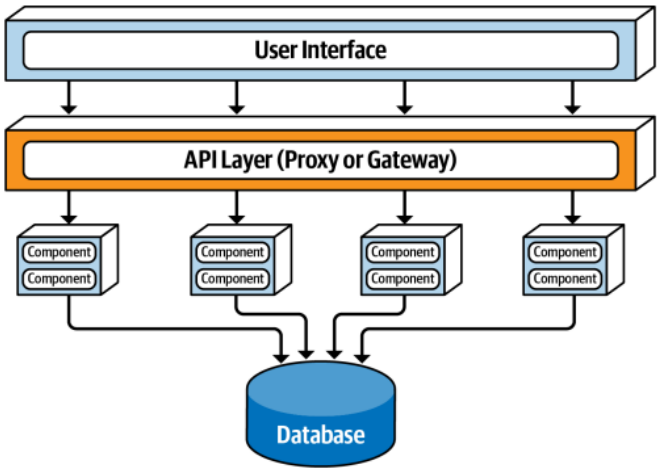

Figure 3: Basic topology of the service-based architecture style with an API Layer. Source: Mark Richards and Neal Ford [63].

the same level of granularity or reuse. In contrast, infrastructure services manage operational concerns such as monitoring and security, typically handled by a shared infrastructure team.

SOA typically utilizes an Enterprise Service Bus (ESB) to facilitate communication and coordination among services [8]. The ESB is a central hub that integrates various services, ensuring seamless interoperability across different platforms and languages, as illustrated in Figure 4 which shows the topology of orchestration-driven service-oriented architecture. The ESB provides essential functions such as message routing, transformation, and protocol mediation, which streamline the interaction between services and enhance the system's overall efficiency [5]. However, the ESB also present several limitations,

- **Increasing Inter-dependencies**: Implementing an ESB can introduce significant complexity. The centralized nature of an ESB requires meticulous configuration and management to ensure seamless communication among services, especially when they share many resources and need to coordinate to perform their functionality;
- **Single Point of Failure**: As a central hub, the ESB represents a single point of failure. Any disruption or failure in the ESB can impact the entire service network, leading to potential system outages and reduced reliability. Clients and services cannot communicate with each other at all if the ESB goes down;
- **Performance Bottlenecks**: As the ESB serves as a central point for message routing and processing, it can become a performance bottleneck. High volumes of service interactions can lead to increased latency and reduced throughput. Also, scaling an ESB to handle growing service demands can be difficult and resource-intensive. The need for powerful infrastructure to support an ESB adds to the overall system complexity.

SOA is a natural fit when doing DDD [23]. SOA's approach of using coarse-grained ser-

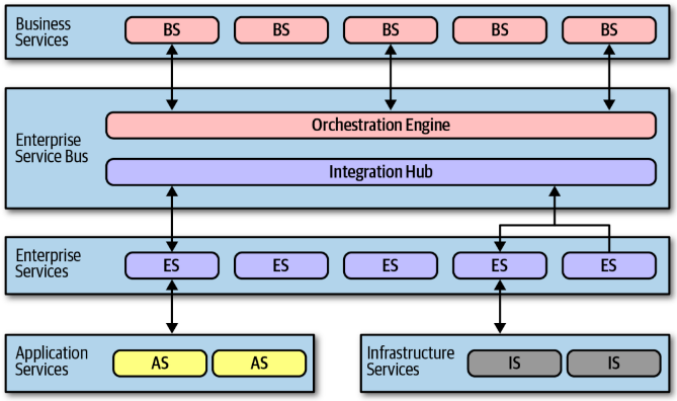

Figure 4: Topology of orchestration-driven service-oriented architecture. Source: Mark Richards and Neal Ford [63].

vices to encapsulate specific domain functionalities matches the idea of bounded contexts from DDD [37]. This mutual focus facilitates loose coupling and supports reusability, emphasizing isolated contexts. Integrating SOA and DDD principles contributes to creating functionally robust software systems that are easier to maintain, scale, and structure.

2.1.3.2 Microservices Architecture

Microservices architecture is a SOA rooted in the concept of small components that focus on performing one single task well [53]. First mentioned in 2006 by Werner Vogels [71], microservices emerged as a modern and promising approach to creating modular, portable, and scalable software. A microservice-based architecture methodology closely aligns with the Single Responsibility Principle (SRP), a foundational concept in clean architecture introduced by Robert C. Martin (Uncle Bob) [43], which posits that a component should have only one reason to change.

Microservices architecture represents a significant evolution of the SOA architectural style, addressing its limitations to better align with modern cloud-based enterprise environments. Unlike SOA, which relies on a centralized ESB, microservices architecture consists of highly granular and autonomous software components known as microservices. These microservices are designed to be fine-grained, ensuring each microservice operates independently. In a microservices-based architecture, each component has its own communication protocols, which are exposed through lightweight APIs. This decentralization eliminates the need for a centralized ESB, as consumers interact directly with the microservices through their respective APIs (Figure 5). Additionally, in SOA, databases are typically centralized, whereas microservices architectures promote decentralized data management, with each microservice having its own local database. This independence allows each microservice to manage its data autonomously, reducing inter-service dependencies and enhancing scalability, flexibility, and fault isolation.

Microservices offer a compelling solution for these challenges by promoting a high

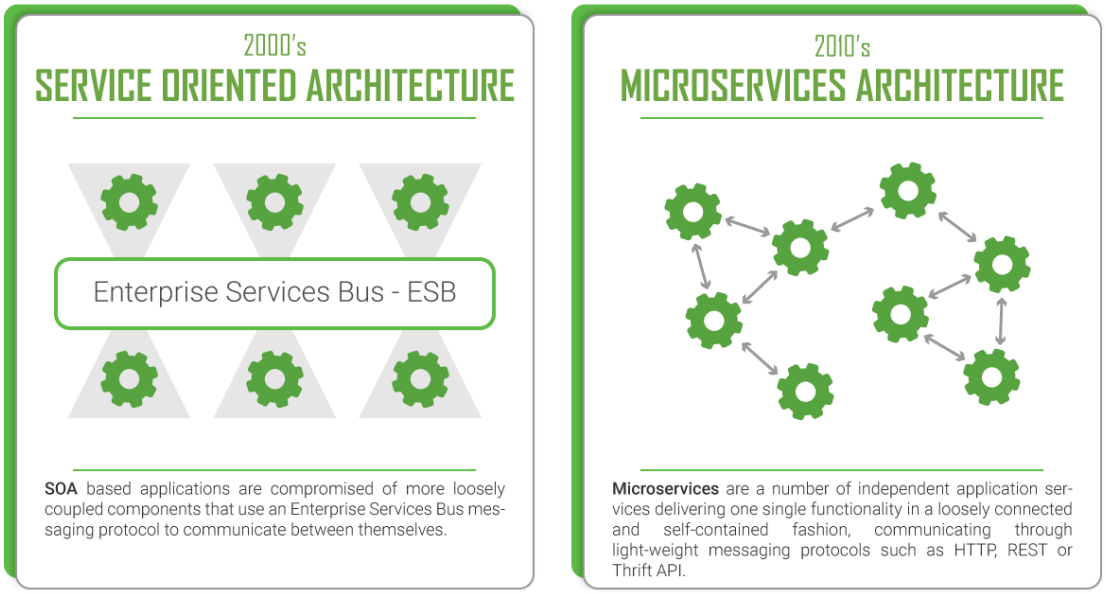

Figure 5: Difference between SOA and microservices. Source: DZone [22].

degree of modularization, directly aligning with the SRP, where each microservice operates independently, leading to minimal coupling and enabling scalability, agile management, and technological versatility. Besides, the independence of microservices allows them to be written in different programming languages and use different databases, facilitating greater flexibility and innovation in software development compared to monolithic approaches.

As microservices inherently derive from a SOA approach, a microservices-based architecture is naturally well-suited to Domain-Driven Design. This alignment arises from the fundamental principles shared by both paradigms, emphasizing modularity, scalability, and the clear demarcation of service boundaries. Historically, the microservice-based architectural style has been deeply interconnected with domain-driven design, to the extent that the terms microservice and bounded context are often used interchangeably [37]. This close relationship underscores the efficacy of DDD in structuring and managing microservices, enabling the development of robust, scalable, and maintainable systems that align closely with business domains and processes. Figure 6 illustrates the topology of the microservices architecture style.

The benefits of microservices extend beyond traditional software development, offering significant advantages to several application domains, including robotics. The increasing complexity of robotic systems, especially those with modular architectures, makes microservices a compelling solution. This high degree of modularization allows each microservice to independently handle a specific robotic function such as environment perception, behavioral decisions, navigation, and object manipulation.

Nevertheless, a microservice architecture presents challenges in real-time robotics environments compared to monolithic architecture. Ensuring low latency and high throughput, crucial for real-time applications – such as those found in SSL environments – are major con-

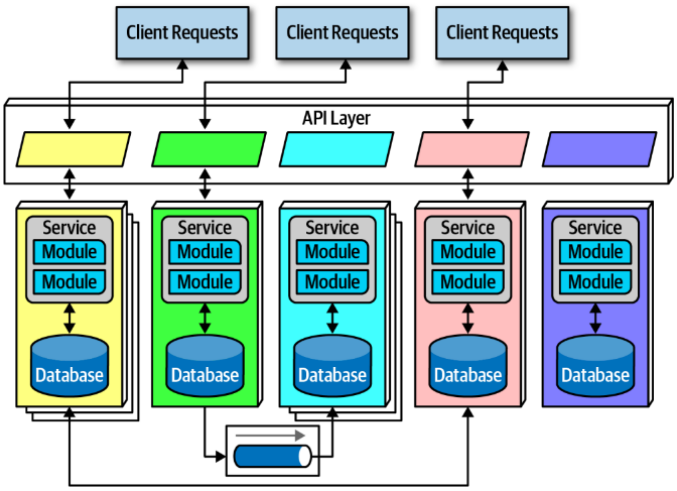

Figure 6: The topology of the microservices architecture style. Source: Mark Richards and Neal Ford [63].

cerns related to microservices approaches. Communication overhead between components can also introduce delays, impacting system responsiveness. Moreover, managing distributed transactions and ensuring consistency across components becomes complex, particularly when dealing with real-time data. Scalability is another issue, requiring careful orchestration to scale microservices while maintaining performance dynamically. Additionally, monitoring and debugging in real-time environments pose challenges, as traditional methods may not suffice.

By addressing issues related to real-time applications through meticulous design and optimization, the benefits of microservices can be harnessed without compromising performance. Furthermore, leveraging appropriate technologies can facilitate the successful implementation of microservices in real-time scenarios.

2.1.3.3 Event-Driven Architecture

The Event-Driven Architecture (EDA) style is widely adopted for its ability to create scalable, high-performance applications. It operates asynchronously with decoupled event processing components that receive and handle events [63]. This architecture is versatile and suitable for both small applications and complex systems. Moreover, EDA integrates seamlessly with other architectural styles, offering a versatile approach to system design, such as event-driven microservices [53].

Event-based communications are not a replacement for request-response communications but a completely different way of communicating between services, emphasizing decoupling and asynchronous interaction between services [10]. A distinct advantage of EDA lies in its foundational decoupling of event producers from consumers. This architecture allows producers to broadcast events without precise knowledge of the consumers. In parallel, consumers can asynchronously process events, conferring a notable degree of system design flexibility and resilience. Such decoupling also permits individual system components' independent evolution,

deployment, and scalability, enhancing overall system robustness.

In an EDA, two primary topologies are utilized, broker and mediator, each suitable for distinct requirements [63]. The mediator topology is chosen when precise control over event workflow is necessary, while the broker topology stands out in scenarios demanding high responsiveness and dynamic event processing. Understanding these topologies' characteristics and implementation strategies is crucial for selecting the most appropriate one for a given situation.

The broker topology works without a central event mediator. Instead, message flow is decentralized across event processor components, facilitated by a lightweight message broker. Key elements within the broker topology include the initiating event, which triggers the event flow; the event broker, which manages event channels and facilitates event processing; event processors, which handle specific tasks associated with events; and processing events, which communicate task completion asynchronously within the system. Event processors initiate tasks in response to events from the event broker and propagate processing outcomes through new events. Figure 7 illustrates the event processing flow within the broker topology, highlighting its event broker setup across clustered instances.

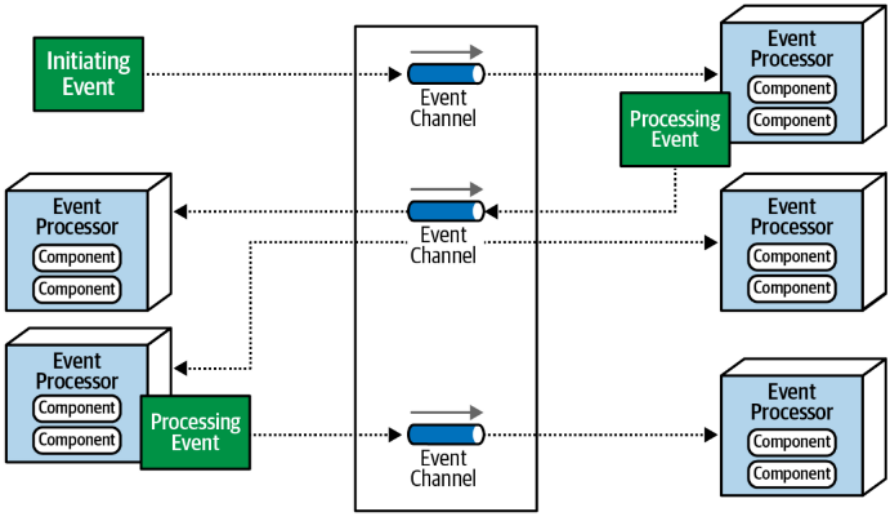

Figure 7: Broker topology. Source: Mark Richards and Neal Ford [63].

The mediator topology in EDAs addresses the limitations of the broker topology by incorporating an event mediator to manage and control workflows requiring coordination among multiple event processors. The mediator's centralized control allows it to maintain event state, manage error handling, and handle recoverability, contrasting with the decentralized approach of the broker topology. However, this results in a more coupled architecture than the broker topology [25]. Key components include the initiating event, event queue, event mediator, event channels, and event processors. Unlike the broker topology, where events are broadcast, the mediator topology routes the initiating event to an event queue, which the event mediator then processes. The mediator generates processing events and sends them to dedicated event chan-

nels point-to-point. Event processors listen to these channels, process the events, and respond to the mediator without broadcasting their actions. Figure 8 illustrates the event processing flow within the mediator topology.

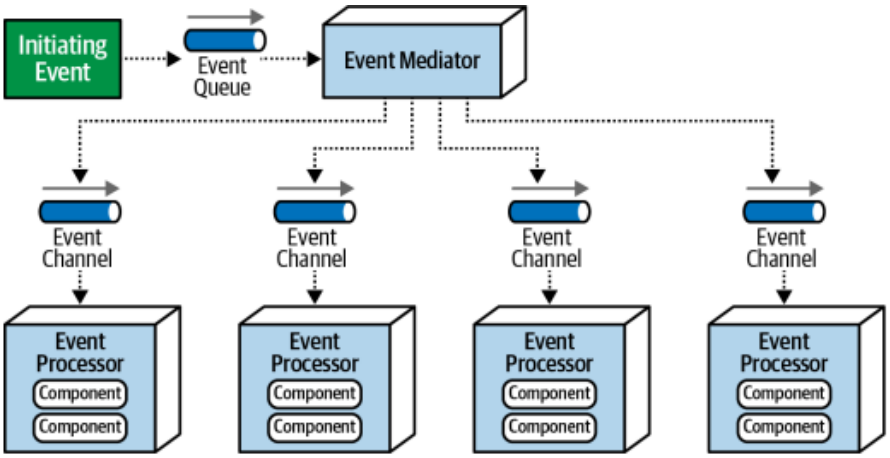

Figure 8: Mediator topology. Source: Mark Richards and Neal Ford [63].

In the mediator topology, processing occurrences are conceptualized as commands, representing actions that need to occur rather than events that indicate actions that have already occurred [63]. Commands in the mediator topology necessitate execution and cannot be disregarded, ensuring that specific tasks are carried out in a predetermined sequence. In contrast, events in the broker topology serve as notifications of completed actions, and subsequent processors may choose to ignore them based on their relevance. This difference underscores the mediator topology's emphasis on coordinated, mandatory workflows as opposed to the broker topology's more flexible, event-driven approach.

Additionally, while the mediator topology resolves some issues of the broker topology, it introduces its own challenges. Modeling dynamic processes in complex event flows can be difficult, often necessitating a hybrid model. The mediator must scale to avoid bottlenecks, and performance may be lower due to centralized control over event processing. Additionally, event processors are less decoupled, impacting overall system performance.

2.1.3.4 Middleware

Traditionally, communication between distributed software components has relied on sockets, utilizing protocols such as Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). As distributed systems have become more prevalent, middleware has emerged as a higher-level, more resilient, and simpler way of connecting components across different processes, machines, and networks. Middleware is an intermediary layer between applications and the operating system, bridging the gap between application functionalities and the underlying software/hardware infrastructure [12] (Figure 9).

Typically, middleware facilitates interaction and communication between diverse ap-

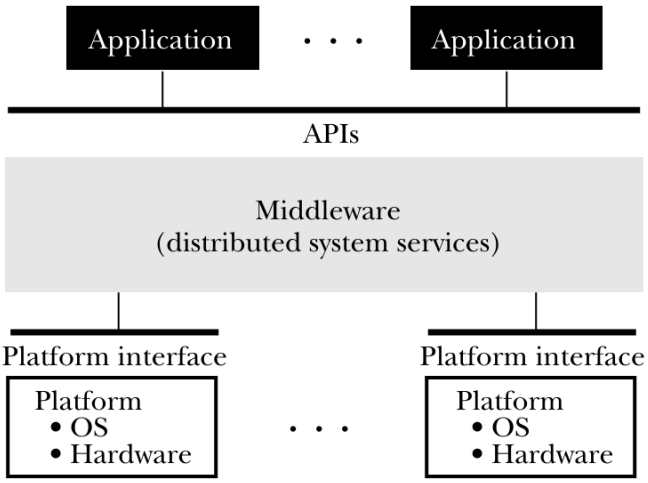

Figure 9: Traditional middleware architecture. Source: Bernstein [12].

plications across distributed components. Its primary goal is simplifying distributed systems by enabling application developers to abstract away lower-layer implementations. Banavar *et al.* [7] have underscored its significance in enabling interoperability and communication across disparate systems and platforms. Additionally, middleware effectively masks the heterogeneity of various architectures, operating systems, programming languages, and networking technologies, thereby streamlining application development and management [61].

Mascolo *et al.* [44] categorize middleware solutions based on their communication primitives: procedural, object/component, transactional, and message-oriented.

- **Procedural Middlewares (RPCs)**: Use Remote Procedure Calls (RPC) for function calls across systems, behaving as local calls. RPCs lack group and asynchronous communication support, limiting scalability and being unsuitable for dynamic environments.

- **Object/Component Middlewares**: Enable communication between distributed objects where a client requests operations from a server object on a different node. Suitable for fixed systems but not for dynamic environments due to high operational costs and reliance on synchronous communication.

- **Transactional Middlewares**: Support systems with components spanning multiple nodes, ensuring operations occur on all nodes via the two-phase commit (2PC) protocol. Despite supporting synchronous and asynchronous communication, transactional middlewares incur high overhead, which may be impractical in dynamic scenarios.

- **Message-Oriented Middleware (MOM)**: Facilitate communication between distributed components using message exchanges. Clients send requests to servers,

which respond with execution results via messages. MOM supports asynchronous communication, which is essential for decoupling clients and servers.

These foundational concepts are enhanced by actual implementations that add another abstraction layer with customized resources and functionalities. The choice of middleware largely depends on the communication pattern—such as client-server, publish-subscribe, or producer-consumer—and synchronization requirements.

Despite the advantages, using middleware in distributed systems introduces certain costs. The abstractions and transparencies it provides, such as hiding details like location, technology, and fault tolerance, can lead to increased latency and resource usage, affecting computational efficiency. This is particularly critical in real-time systems, such as those used in robotics, where stringent requirements for computational efficiency, availability, reliability, robustness, and responsiveness must be met. The communication primitives selected during software development, including middleware or socket choice, communication patterns, and package definitions, are crucial for developing systems that meet these demands.

MOM is particularly well-suited for microservices-based architectures that employ event-driven designs. It facilitates asynchronous communication, which decouples services and allows them to interact without direct dependencies, enhancing system resilience and scalability. This decoupling ensures that a failure or slowdown in one service does not directly impact others.

MOM architectures are particularly notable for their ability to facilitate communication across heterogeneous systems. They typically consist of three key components:

- **Messages**: Data packets exchanged between middleware nodes, including notifications, events, and data requests. Message size is typically flexible and managed by the middleware for network transport.
- **Message Queues**: Intermediate queues that decouple data and execution flow between nodes, supporting asynchronous communication.
- **Message Broker**: A server in MOM architectures coordinating message exchange, handling message queues, validation, translation, and routing.

Asynchronous communication is essential for maintaining loose coupling between services within microservices architectures. MOM enables services to publish events to other services independently, eliminating the need to wait for remote code. It also facilitates time and logic decoupling, promoting scalability and flexibility. This approach promotes scalability and flexibility, ensuring that services operate independently of each other's existence [53].

In EDAs, MOM enables event sourcing by capturing, storing, and making all events available for consumption by relevant services. This empowers services to react quickly to changes, promoting responsive systems, which is especially beneficial in dynamic environments like robotics.

MOM offers two primary approaches, broker-based and brokerless, distinguished primarily by the presence of a centralized broker. Broker-based architectures (Figure 10) centralize message routing and queuing through a dedicated broker, offering benefits like space and time decoupling. However, they may introduce significant latency due to message transit through the central broker, which can be detrimental in applications requiring real-time responsiveness [40].

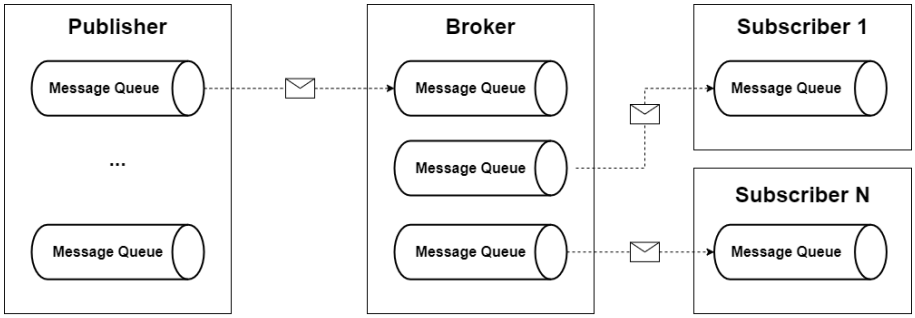

Figure 10: Broker-based MOM approach. Source: Adapted from Lilis *et al.* [40].

Brokerless architectures (Figure 11) emerge as an alternative to mitigate latency issues inherent in broker-based systems [40]. By eliminating the central broker, brokerless architectures enable direct communication between nodes, significantly reducing message transit times and enhancing overall system responsiveness. This feature is particularly advantageous in scenarios where minimal latency is critical for timely decision-making and operational efficiency in robotics and automation. A directory service [25] can be implemented to address the challenge of service coupling in brokerless architectures [40], as illustrated in Figure 12. It replaces centralized broker functionality by facilitating efficient node discovery and management, improving service discovery in dynamic and distributed environments.

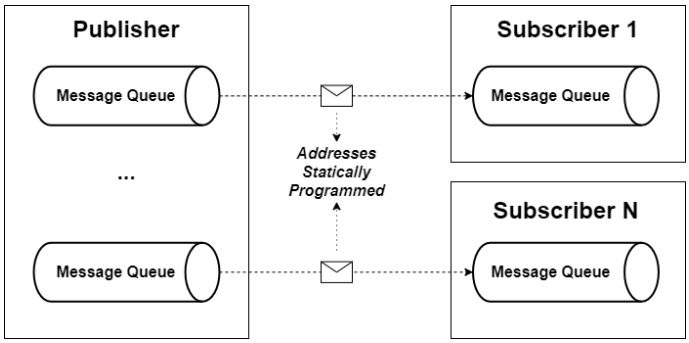

Figure 11: Brokerless MOM approach. Source: Adapted from Lilis *et al.* [40].

MOM is essential for implementing robust, scalable, and maintainable microservices-based architectures with event-driven designs. Balancing the benefits of abstraction and transparency against potential impacts on latency and resource usage is key to leveraging the full potential of MOM in microservices architectures.

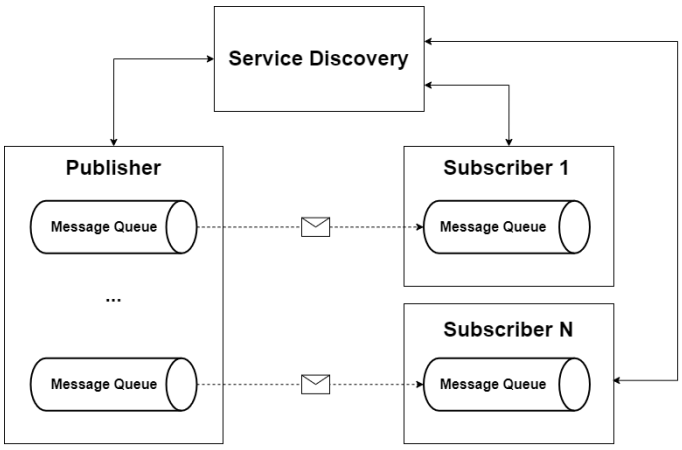

Figure 12: Brokerless MOM approach with Service Discovery. Source: Adapted from Lilis *et al.* [40].

2.1.4 The Architectural Approach Adopted in This Work

The architecture proposed in this work is based on a distributed architecture using an Event-Driven microservices approach. It applies a MOM to enable asynchronous communication in a dynamic, low-latency environment, which is necessary for the RoboCup SSL scenario. The proposed approach is based on discussions with RobôCIn's [68] SSL team, who acted as domain experts following DDD principles.

2.2 SOFTWARE MODELING

This section explores software modeling, highlighting the crucial role of modeling languages in designing and developing complex systems. It begins with an introduction to modeling languages, mainly about their significance in effectively capturing, documenting, and communicating system designs. The primary focus is on the Unified Modeling Language (UML), a widely adopted standard in software engineering. The section examines the importance of UML, detailing its components and their applications across different stages of software development, such as visualizing, specifying, constructing, and documenting.

2.2.1 Overview

Modeling languages are critical tools in designing and developing complex systems. They provide structured methods for representing, analyzing, and communicating various aspects of a system, which is essential for managing complexity and ensuring effective collaboration among stakeholders and developers. Over the past few decades, the development and adoption of these languages have been significantly advanced by the Model-Driven Engineering and Model-Driven Development approaches [14].

A modeling language describes and designs systems through a set of notations and

constructs [51]. These notations may include various forms such as diagrams, which visually represent system structure and behavior; pseudo-code, which outlines algorithms and processes in a simplified text format; executable code, which demonstrates system functionality through actual programming code; and textual descriptions, which provides detailed narratives of system components and interactions.

The significance of modeling languages lies in their ability to simplify complexity, enhance communication, and improve design and analysis processes. By allowing designers to abstract and focus on key aspects of a system, modeling languages help reduce cognitive load and facilitate a clearer understanding of intricate details. Visual notations, such as diagrams, offer a straightforward view of system components and their interactions, aiding in comprehending system architecture and behavior.

Modeling languages also play a crucial role in effective communication among stakeholders. A well-defined modeling language ensures that all parties, including developers, designers, and clients, have a shared understanding of the system. This common language helps minimize misunderstandings and promotes clearer communication. Also, models serve as valuable documentation that captures design decisions, system architecture, and functional requirements, essential for future maintenance, development, and knowledge transfer.

Regarding design and analysis, modeling languages facilitate validation and verification by enabling designers to simulate different scenarios and interactions before implementation. This allows potential issues to be identified early in the development process. Furthermore, modeling supports iterative improvement, allowing designers to refine and enhance the system based on feedback and analysis. This iterative approach helps adapt the design to meet evolving requirements and constraints.

Modeling languages are widely used across various domains, including software engineering, systems engineering, and business process modeling. An architectural language refers to any form of expression used to describe software architecture, ranging from informal notations like box-and-line diagrams to formal Architecture Description Languages, such as Systems Modeling Language (SysML) [18] for systems engineering, Business Process Model and Notation (BPMN) [31] for business processes are tailored to specific needs, and UML [11], one of the most recognized modeling languages in software engineering, which provides a standardized set of notations and diagrams to represent different aspects of software systems, making it a versatile tool for designing, visualizing, and documenting software architecture.

2.2.2 Microservices Modeling

Microservice architectures offer substantial benefits, including the ability to design, develop, test, and release services with increased agility. However, architecting microservices is challenging due to the complexity of managing a distributed system. These challenges require careful design and robust strategies to maintain system reliability and performance in a

decentralized environment.

Research interest in microservices architecture has increased since 2015, reflecting the growing adoption and exploration of microservices in various industries [19]. This trend was followed by a marked increase in publications on architectural languages for microservices beginning in 2018, where researchers frequently extended existing modeling languages, such as UML, BPMN, and graph-based frameworks, with profiles to develop more effective tools for describing and managing these architectures [32].

Di Francesco *et al.* [19] observed that most proposed microservice architectures were described using informal languages, with UML being used only occasionally. They identified nine distinct languages that had either been utilized or proposed for modeling specific aspects of microservice architectures, highlighting the absence of a predominant architectural language. As noted in their study, this diversity may hinder effective communication and modeling, indicating a need for a standardized language to improve consistency, facilitate reasoning about system qualities, and enhance stakeholder communication.

Zaafouri *et al.* [85] conducted a systematic review highlighting the importance of modeling languages in SOA to manage complexity and scalability. The study identifies the UML, BPMN, and Service Component Architecture (SCA) [56] as the most widely used languages in large-scale SOA implementations. UML, in particular, is noted for its versatility in modeling complex systems' structural and behavioral aspects.

The analysis of current microservices modeling practices reveals the critical role that UML plays in effectively managing the complexities of microservices architectures. UML provides a wide set of diagrams that can accurately represent the relationships and components within a microservices system. Additionally, the Object Management Group (OMG) extended UML with the SoaML [55] standard to cater specifically to SOAs, incorporating service concepts such as consumers, providers, contracts, and choreography. Despite the existence of other standards, UML continues to be the preferred language among practitioners for modeling SOA and microservice architectures, as noted by Zaafouri *et al.* [85].

The challenges of microservices architecture faced by architectural languages include microservice identification and decomposition [32]. To address these issues, many experts have identified DDD as an effective solution to define the microservices scope through bounded contexts [53, 64]. Unlike Object-Oriented Modeling [13], which centers on structuring software around objects, DDD emphasizes understanding the business domain and its complexities [81]. DDD relies on informal diagrams and a ubiquitous language to define bounded contexts, fostering communication between technical and non-technical stakeholders and allowing flexibility in expressing domain concepts [23].

The proposed framework defines a microservices-based architecture for SSL using principles from DDD – to define services based on key concepts and relationships within the context –, and applies different UML diagrams to increment the system design, where each diagram represents a subset of the system. Table 1 briefly describes the diagrams used in this work. In

Section 3.2.1 we further detail the relevance of each of these diagrams in the integrated model we propose.

Table 1: Applied UML Diagrams and Their Modeling Capabilities. Adapted from Miles *et al.* [51].

Diagram Category	**Diagram Type**	**Purpose and Usage**
Structural	**Class**	Defines the system's static structure by modeling classes, types, interfaces, and the relationships among them, such as inheritance and associations.
	Component	Depicts the high-level components of the system and their interfaces, focusing on how these components interact and are organized within the architecture.
Interaction	**Sequence**	Describes the sequence of interactions between objects in the system, emphasizing the order and timing of messages passed between them.

3

PROPOSED APPROACH

This chapter discusses the work proposal, exploring the SSL environment, emphasizing the challenges of real-time decision-making, coordination, and low-latency communication and the software's critical role in meeting these demands. Finally, the chapter outlines the final objectives and the specific approach: the SSL VAR (Small Size League Video Assistant Referee).

3.1 OVERVIEW

RoboCup is a prestigious international robotics competition designed to advance the fields of autonomous robotics and artificial intelligence. Within the competition, the SSL is one of the oldest and most challenging categories. The league's emphasis on speed, agility, and decision-making presents unique challenges, requiring teams to design robots capable of navigating dynamically changing environments while executing complex strategies. In the SSL robot soccer competition, matches are played between two teams of mobile robots. The competition relies heavily on SSL Vision [88], a standardized vision system which tracks all the objects on the field and works by processing data from one or more cameras that are mounted above the playing surface. This system tracks robots identified by color patterns on their tops (Figure 13), the orange golf ball used in play, and the field lines.

The game is controlled by the community-maintained ssl-game-controller [66], operated by a human referee. This software translates the decisions of the referee into Ethernet communication signals broadcast to the network. This centralized control system ensures that all teams operate under the same conditions and helps maintain fairness and consistency throughout the competition.

One or more automatic referee applications can supervise a game and report events to the game controller. The competition defines an additional vision tracker protocol that contains filtered and enriched tracking data. These messages are not published by SSL Vision but are provided by systems such as TIGERs AutoReferee [77]. This protocol is intended for use by the game controller and teams that do not yet have their sophisticated filters.

The SSL Vision software operates on a dedicated computer, processing field data and

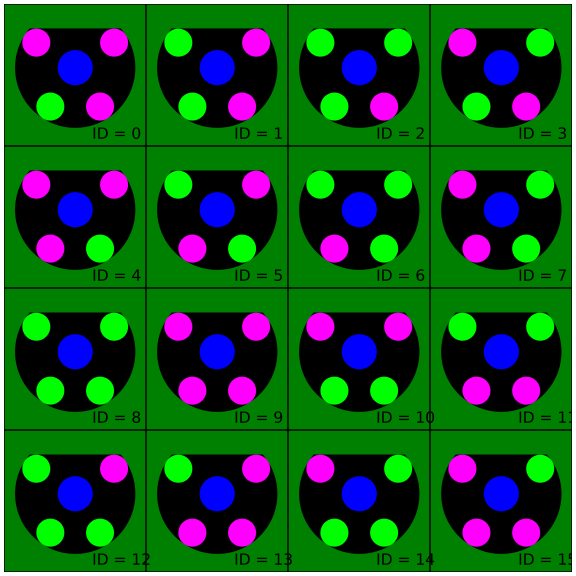

Figure 13: Standard Vision Pattern Colors. Source: RoboCup Federation [24].

broadcasting the global positions of the robots and ball(s) across the network. The ssl-game-controller operates on another dedicated computer, processing referee information and broadcasting it to teams via Ethernet. Each team uses an off-field computer to receive positional information and referee commands, enabling them to define team strategies, which dictate the actions and behaviors of the robots in the real environment. Each team's computer performs most of the computation and exchanges information with the robots using wireless communication. This setup allows for real-time adjustments and decision-making, which are crucial for the fast-paced nature of SSL matches. Figure 14 shows the general dataflow for the SSL environment.

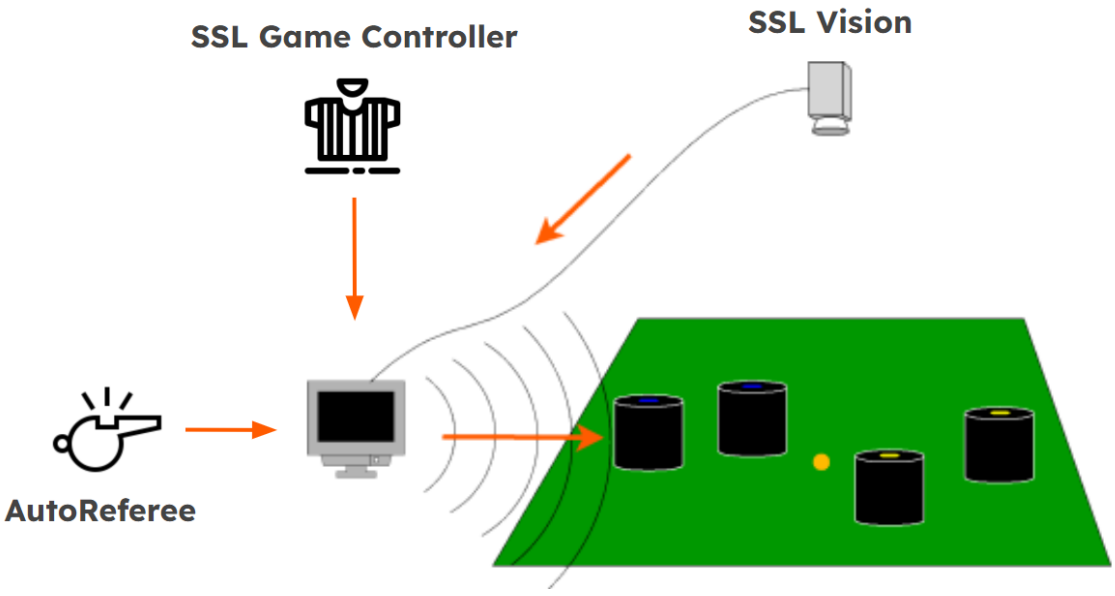

Figure 14: General Dataflow for Small Size League Environment. Adapted from RoboCup Federation [24].

3.2 PROPOSAL

The central objective of this work is to develop a microservice-based architecture for a strategy pipeline to Multi-Robot Systems (MRS), explicitly focusing on the RoboCup Small Size League category. In conjunction with the strategy pipeline, the architecture aims to include a playback component that provides live data for game streaming and facilitates data segmentation for external applications requiring time-based queries, such as replay systems and time series modeling. The proposed architecture draws inspiration from the essential components a robotic agent needs to navigate a physical environment, emphasizing the integration of perception, decision-making, behavior, and control [58]. The main components of robot navigation, as illustrated in Figure 15, include:

- *Sensors*: The physical components that provide the necessary input for the agent's perception, such as cameras, ultrasound or infrared sensors.
- *Perception*: Processes sensor data to extract key information, such as obstacle locations or the robot's position, applying algorithms for object detection.
- *Localization*: Determines the robot's location using techniques like odometry, GPS, or Simultaneous Localization and Mapping (SLAM).
- *Mapping*: Creates a representation of the environment using methods such as occupancy grids or geometric mapping.
- *Path Planning*: Finds an optimal path from the robot's current location to its destination, while avoiding obstacles in the environment.
- *Control*: Executes the planned path by sending commands to the robot's actuators, which controls the robot's actions.

However, navigating in the dynamic environment of SSL poses significant challenges to understanding and debugging plays. Due to the speed and complexity of the game, determining the precise sequence of events and identifying the cause of unexpected behaviors can be difficult. Human-referee decisions become particularly challenging when discrepancies arise between the automatic referee's commands and the actual events on the field. Currently, the common system for reviewing game moments is ssl-logtools [65]. However, ssl-logtools does not support an authentic replay of a team's past decisions. It only re-transmits the vision and referee packets through the network, simulating real packets and requiring the strategy software to run and process the incoming packets again, thus re-executing the strategy. Any changes in the software, as well as the inherent non-determinism of dynamic decisions within the software, can result in slightly different behaviors.

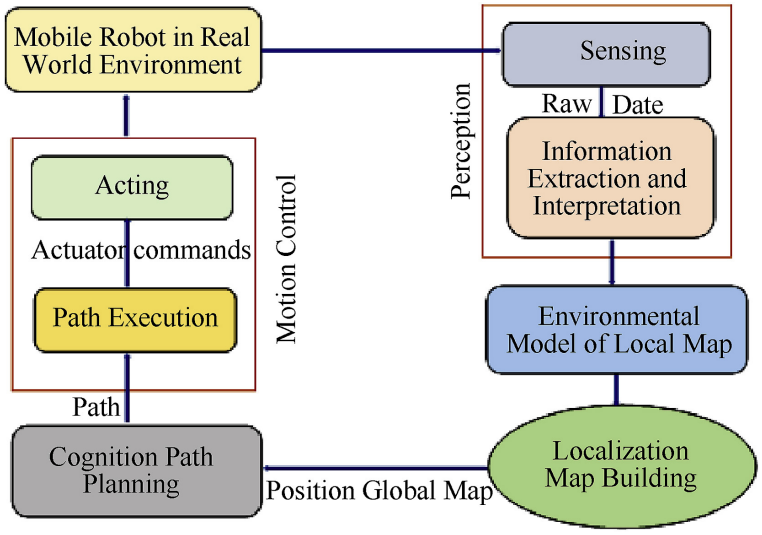

Figure 15: Flow diagram for mobile robot navigation that inspired the strategy microservices-based architecture. Source: Patle *et al.* [58].

Furthermore, building a complete microservice-based architecture from scratch presents significant challenges, especially for a complex domain like a robotics competition. Developing a subset of the target architecture as a proof of concept in software engineering is a well-established practice. It enables developers to identify potential issues early, validate assumptions, and gather empirical data on the system's performance and scalability. According to Martin Fowler, building a proof of concept helps in understanding the feasibility of the architecture and mitigating risks associated with unproven technologies or methodologies [26]. Similarly, iterative development, as illustrated by the Agile methodology, emphasizes the value of incremental delivery and continuous feedback to enhance the overall design and implementation process [9].

Additionally, microservices architectures are known for potential latency challenges due to inter-service communication. Acknowledging this, the proposed work is designed as a case study to evaluate whether microservices can meet the strict latency requirements of the SSL environment. The objective is to explore whether this architectural approach, despite its inherent communication overhead, can be effectively applied to the RoboCup SSL system.

Therefore, given the inherent challenges in real-time decision-making and system debugging within SSL matches, this work focuses on developing a subset of the entire architecture, the SSL VAR system, an advanced microservice-based architecture designed to process both external vision and referee data to create an understanding of the environment, providing an API for live data and time-based queries for the raw and processed information.

3.2.1 Software Modeling

Effective software modeling ensures the design and implementation phases align with the system's intended functional requirements. Defining bounded contexts is fundamental in

microservices architecture, as it delineates clear boundaries for each service, ensuring they are responsible for distinct parts of the system. Dynamic interaction diagrams play a significant role in depicting the flow of information and control among various system components. These diagrams help visualize how components interact over time, highlighting potential issues and optimizing system performance. Component architecture diagrams provide a high-level overview of the system's structure, illustrating how different components interact and depend on each other. This macro perspective is essential for ensuring that all system parts are well-integrated and that the overall architecture supports the system's goals.

3.2.1.1 Service Definition

Traditionally, use cases outline scenarios where a system is utilized to fulfill user requirements, focusing on specific functionalities the system must deliver. These use cases are instrumental in guiding traditional object-oriented development's design, testing, and documentation processes. However, fine granularity is not necessarily suitable to capture microservices-based functions (services).

In a microservices architecture, the emphasis shifts from defining individual use cases to establishing bounded contexts, as DDD illustrates. Rather than broad scenarios, microservices development begins with understanding the problem domain and defining clear boundaries within which specific services operate. While use cases may assist in identifying functionalities, they are less effective in microservices environments, where the primary focus is defining domain boundaries and service responsibilities from the beginning. This ensures that each microservice is aligned with a specific business function, contributing to a more resilient, scalable, and maintainable system.

As outlined previously, a key objective for future work on this project is to develop a strategy pipeline for SSL that draws inspiration from the fundamental components a robotic agent requires to navigate a physical environment effectively. However, unlike traditional robot navigation architectures, this design targets external control software for a multi-robot system, incorporating intermediate and boundary modules to ensure cohesive communication and effective orchestration of the system's agents on the physical environment.

Additionally, the proposed architecture closely aligns with the efforts of RobôCIn's [68] SSL team, where the solution design emerged through detailed discussions with the team members. These members acted as domain experts in line with DDD principles, and their insights played a critical role in shaping the architectural choice. The RobôCIn's SSL team is informally divided into areas such as vision, strategy, navigation, and embedded systems. These divisions align well with DDD's concept of bounded contexts, where each area operates relatively independently while maintaining cohesion within its specific domain.

The design of this architecture is driven by the specific responsibilities assigned to each microservice, as detailed in Table 2. Additionally, Figure 16 illustrates the designed bounded

contexts inspired by the main components of robot navigation (Figure 15).

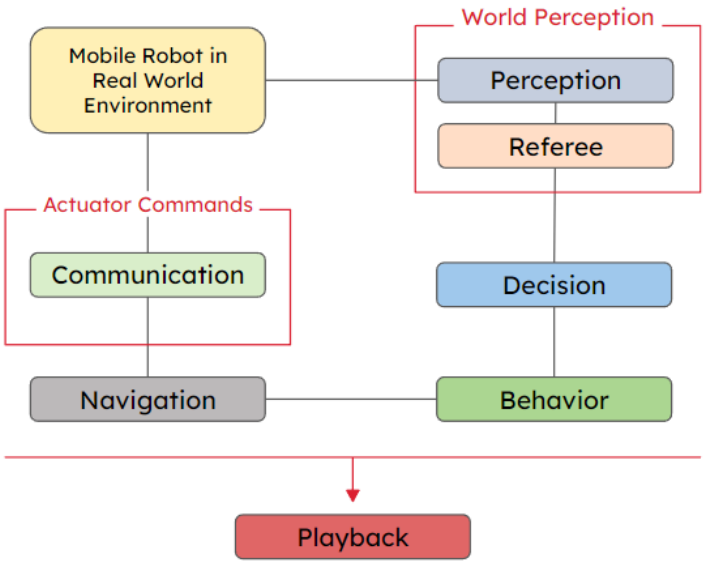

Figure 16: Proposed bounded contexts inspired by Patle *et al.* [58]. Source: The author.

Table 2: Responsibility of each modeled microservice.

Microservice	**Responsibility**
Perception	Receives visual information from the external world, sent by the software provided by the SSL competition, which contains data on the location of entities and the field.
Referee	Receives information about the game stage, sent by the competition's arbitration software.
Decision	Processes and extracts relevant information about the robot's environment, indicating the behavior the agents should take individually to guarantee coordinated behavior between the agents.
Behavior	Coordinates the sequence of actions that each robot must execute to complete a specific behavior, which involves controlling local movement and action decisions.
Navigation	Manages and moves the agent intending to find an ideal path from the robot's current location to its destination and avoid obstacles and other environmental hazards.
Communication	Represents the link between the external control software and the physical agents, being responsible for sending the necessary information for the robot to perform the desired behavior in the real environment.
Playback	Aggregates data from all strategy microservices to provide comprehensive match information. It supports real-time game streaming and enables time-based queries for the UI component or external applications, such as time series analysis.

3.2.1.2 Interaction Diagrams

Interaction diagrams play an essential role in modeling the dynamic interactions between the components of a system, serving as an integral part of the logical view of the design [51]. These diagrams depict how various parts of the system communicate to execute specific responsibilities, effectively bridging the gap between the system's structural design and operational behavior.

In the context of SSL VAR, the system is organized into three core services: Perception, Referee, and Playback. Each service has its interaction diagram, which is fundamental for understanding the flow of data and communication between system components, known as participants. Each participant is represented by a lifeline, a vertical line indicating an object's presence at a particular moment in the sequence. To offer an integrated view of the system and illustrate the structure of an interaction diagram, Figure 17 presents the interaction diagram for the PlaybackService. The interaction diagrams for the Perception and Referee services are detailed in Appendix A.1.

The Playback interaction diagram outlines the sequence of interactions between system participants across three main flows:

1. **onReceiveProcessingData**: The PlaybackService begins receiving and processing data from the PerceptionService and RefereeService to gather detection and game status data, respectively, saving both to the appropriate collections and aggregating them to create a Sample, which is provided on a live match.

2. **onReceiveLiveMatch**: The user initiates the process by selecting to watch a live match, and then the live match is streamed continuously using a loop while the match is running, with the PlaybackService retrieving live samples in real-time.

3. **onReceiveChunkRequest**: The user initiates the process by selecting to watch a replay from a given start timestamp, and then PlaybackService retrieves the corresponding data chunk. The system retrieves both detection and game status data for the specified time range from the collections, providing the replayed information back to the user.

The key strength of sequence diagrams is their ability to depict the sequential flow of messages between participants. These messages can be either synchronous, where the sender pauses and waits for the receiver to process the message before proceeding, or asynchronous, where the sender continues executing without waiting for a response, allowing the system to manage multiple processes concurrently. Asynchronous programming messages often translate into threads or tasks that operate independently of the main program flow, enabling parallel execution and enhancing system performance. They are also fundamental to event-driven programming paradigms, where the sender dispatches a message and continues without delay, leaving the receiver to handle the message as it arrives. In Figure 17, *startLiveStream*

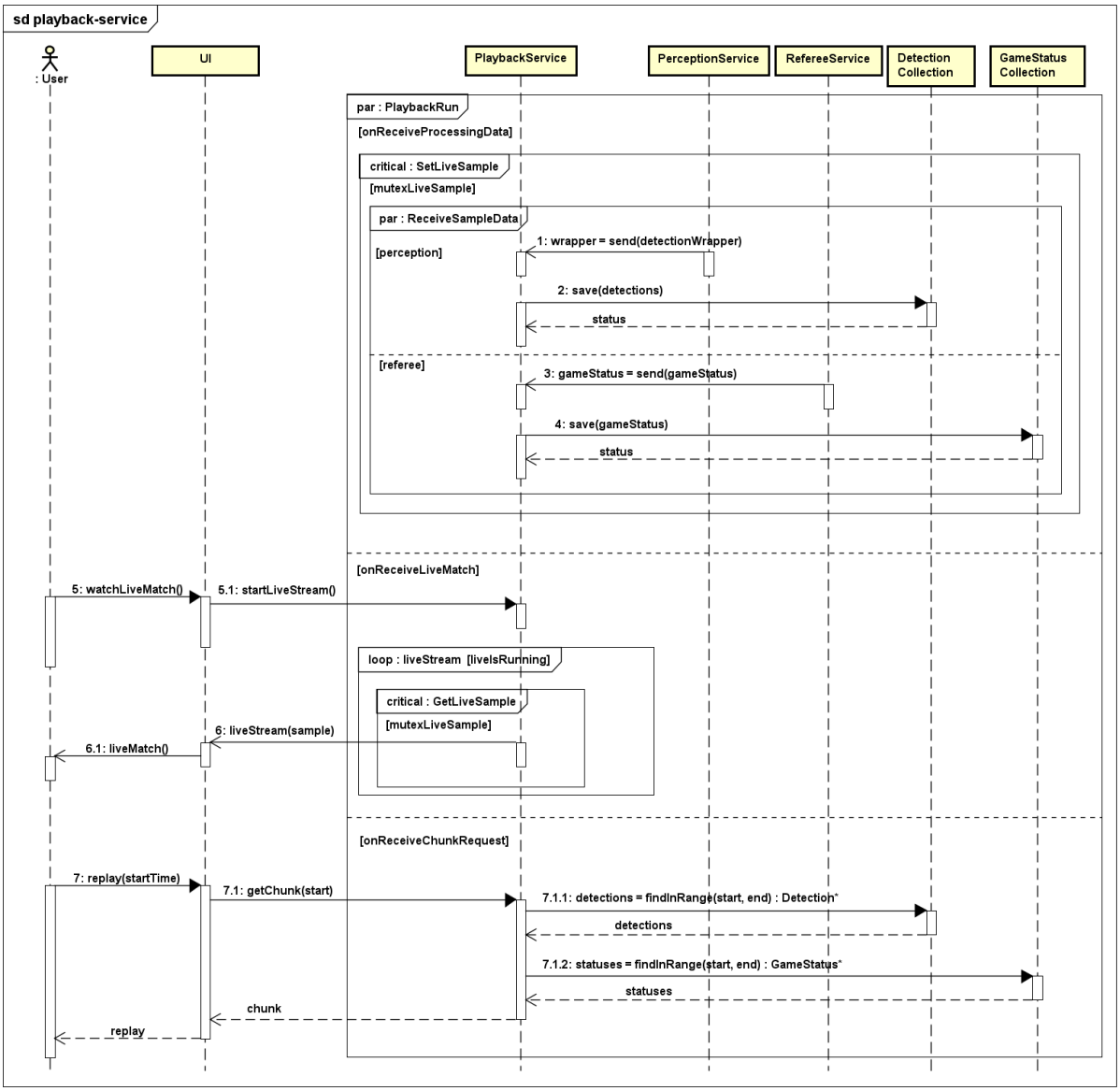

Figure 17: The interaction diagram for Playback service. Source: The author.

and *replay(startTime)* represent synchronous messages, while the message receiving through *send(detectionWrapper)* and *send(gameStatus)* represent asynchronous communication.

However, while sequence diagrams are practical for modeling straightforward interactions, they can become cumbersome and difficult to manage when dealing with more complex scenarios like loops, conditionals, or alternative flows. To address this, sequence fragments were introduced [51]. A sequence fragment is a structural element in UML sequence diagrams that encapsulates a specific portion of the diagram within a bounded box. This box not only groups related interactions but also provides a means to represent control flow constructs in a clear and structured way. Each sequence fragment is labeled with an operator in its top-left corner, indicating the type of interaction it governs. These operators greatly enhance the expressiveness of sequence diagrams, enabling a more organized and modular approach to modeling complex interactions. The Astah UML software offers a variety of operators [4], such as *par*

for parallel processes, allowing interactions within the fragment to coincide without any thread synchronization; *loop* for repetitive actions, which iterates through interactions until a specified condition evaluates to false; *critical* for critical regions where interactions within the fragment involve shared resources that must be accessed in a thread-safe manner; and *opt* for optional sequences, where interactions are executed only if a guard condition evaluates to true.

3.2.1.3 Component Architecture

The component architecture diagram in UML is a crucial tool for visualizing the high-level structure of a system, focusing on its components and their interactions, which provides a static view of the system's architecture. This diagram highlights the dependencies between components, their interfaces, and the overall configuration of the system, offering insights into how different parts of the system collaborate to achieve the desired functionality.

Understanding the component architecture diagram is essential for grasping the overall system design and its integration points. The component architecture diagram provides a foundational understanding of how these functionalities are structurally organized and interconnected within the system. The proposed diagram represents a microservice-based architecture designed for the RoboCup SSL competition that allows for flexible handling of live and historical match data, ensuring smooth integration between all microservices and external systems. Figure 18 illustrates the SSL VAR architecture diagram.

This architecture integrates various services to handle the flow of information, both from the external environment (vision system and referee) and internal team decision-making processes. Below is an explanation of the workflow based on the illustrated services:

- **WebUI**: The front-end component of the system interacts with the Gateway, where the user can view live data streams or a replay from a given timestamp.
- **Gateway**: Serves as an intermediary between the UI and internal services. It manages requests for updating parameters, retrieving chunks, or streaming live data.
- **Perception Service**: Interacts with external vision systems, such as SSL Vision [88], and simulators like grSim [57] and simulator-cli [67], to receive raw vision data. It also integrates tracked vision sources, such as TIGERs AutoReferee [77]. It processes both raw and tracked vision data to deliver relevant information to the RefereeService and PlaybackService.
- **Referee Service**: Processes game-state information from the referee system, managed by the ssl-game-controller [66], along with processed vision data from the PerceptionService. It provides live game status updates, which are then published to the PlaybackService.

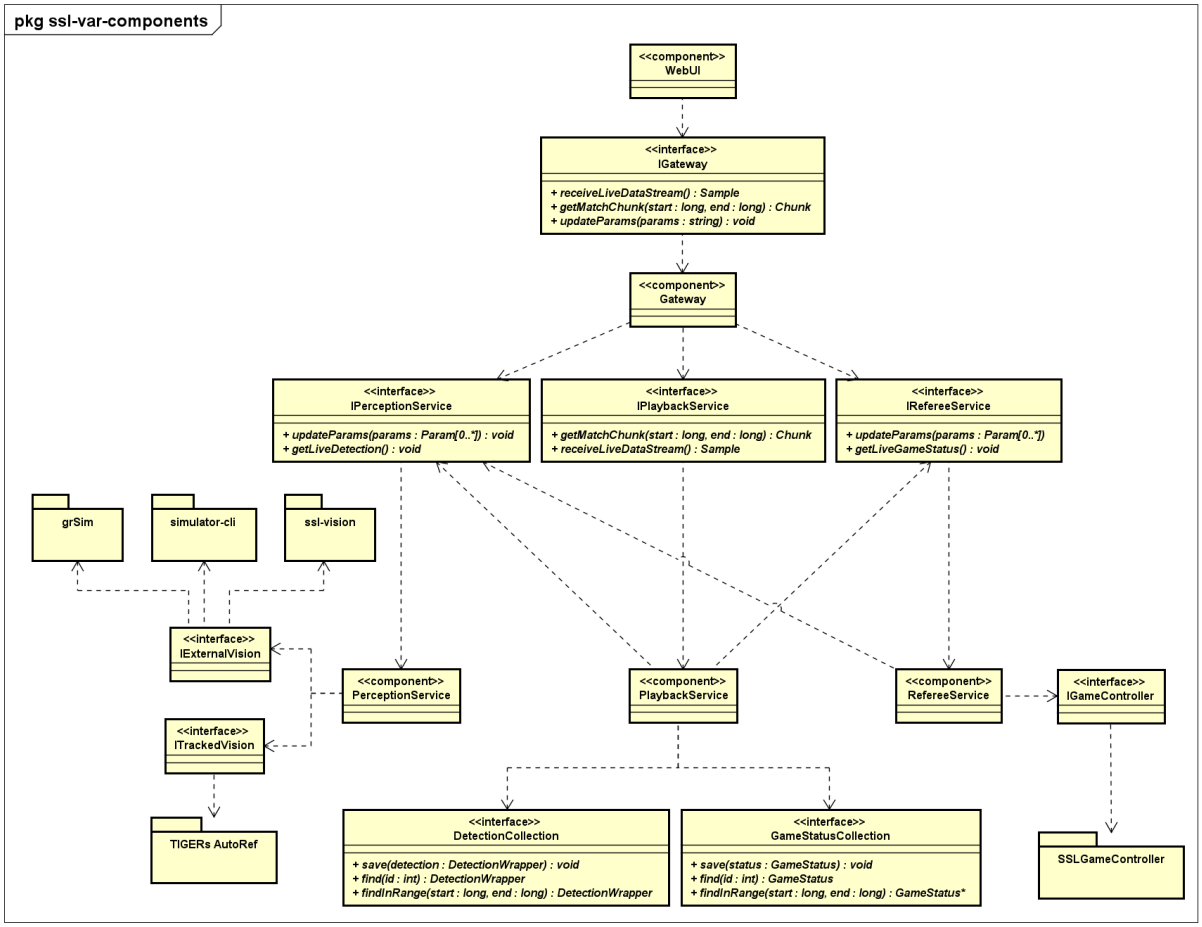

Figure 18: The component diagram for SSL VAR architecture. Source: The author.

- **Playback Service**: Represents the central service of the architecture to integrate users with strategy services, being responsible for real-time data streaming and replay functionality. It retrieves match chunks from the Detection Collection and GameStatus Collection, allowing users to replay matches or analyze specific timestamps. It ensures the synchronization of visual and game-state information from the Perception and Referee services during live streams and replays.

- **Detection and GameStatus Collections**: Represents the collections to store processed data. The Detection Collection holds positional and detection-related information, while the GameStatus Collection stores game events such as goals, fouls, and game states. Both collections are essential for accurate replays and detailed analysis of past matches.

- **External Vision Systems**: Integrates with additional vision systems that follow a defined protocol, which provides raw vision data from a physical camera above the field or from simulators.

- **Tracked Vision Systems**: Works alongside external vision systems providing enriched tracking data (e.g., the TIGERs AutoReferee [77]), mainly for teams that do

not yet have their sophisticated filters.

During a match, the PerceptionService receives real-time raw positional data from cameras (or running simulators), processes it, and publishes it to the RefereeService and PlaybackService. The RefereeService collects game status information (e.g., game stages, fouls, goals) from the ssl-game-controller [66] and detection-related information from PerceptionService, processes the data to determine the game status and publishes the processed information to the PlaybackService. The PlaybackService aggregates all published data and stores it in the Detection and GameStatus collections. The PlaybackService allows users, via the WebUI, to stream live matches with synchronized data from Perception and Referee services, and request specific match timestamps for replay. The communication between the frontend and the PlaybackService is intermediate by the Gateway component.

3.2.1.4 Detailed Services

Class diagrams are foundational tools in modeling the architecture of object-oriented systems, serving as a critical means for visualizing the structure and relationships among various system components. These diagrams meticulously depict the system's classes, highlighting their attributes, operations, and the intricate interconnections that define the system's architecture. One of the most significant aspects of class diagrams is the use of interfaces, which specify a set of operations that any implementing class must adhere to. By abstracting the behavior from its implementation, interfaces foster a modular design approach, enabling different system parts to interact seamlessly without becoming tightly coupled to specific class implementations.

The strategic use of interfaces in UML enhances the flexibility and maintainability of the system's design. By relying on interfaces instead of concrete classes, developers can craft software that is easier to extend or modify and more resilient to change. This design philosophy supports creating modular and loosely coupled systems, ensuring that changes in one part of the system do not adversely affect others. This approach to design is crucial in developing systems that can evolve and adapt to new requirements with minimal disruption to the existing architecture [51].

These design principles were rigorously applied in the model and development of the SSL VAR system, where interfaces played a pivotal role in creating a robust and flexible architecture. Additionally, the SSL VAR system's architecture employs several key design patterns [27] to enhance flexibility, scalability, and maintainability.

Figure 19 presents the class diagram for the PlaybackService to offer an integrated view of the central component of the architecture and illustrate the service structure focusing on the main workflow without external types. The complete detailed service diagrams for the SSL VAR system and the third-party data needed to understand the SSL context are illustrated in Appendix A.2.

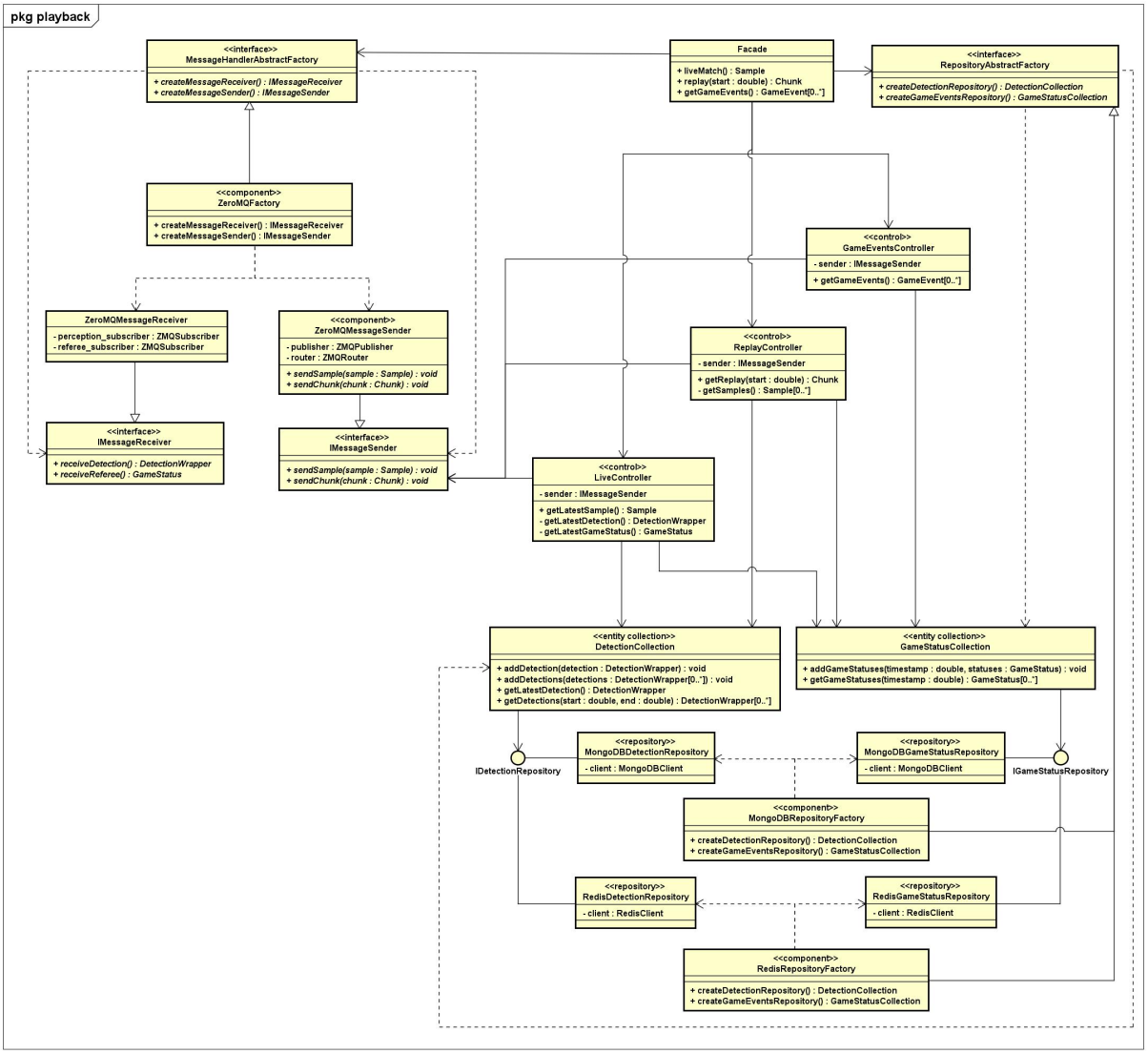

Figure 19: The class diagram for Playback service. Source: The author.

3.2.2 Implementation

Implementing the SSL VAR system involves a meticulously designed architecture that ensures performance and flexibility. This section delves into the strategies and technologies to realize the system's core components. It outlines the deployment of strategy microservices and the configuration of the API Gateway, which together facilitate the efficient handling and communication of data within the system. The choice of technologies and design patterns reflects a commitment to achieving low-latency operation while accommodating the complex demands of real-time robotic systems.

A microservices-based architecture is designed to be language-agnostic, allowing for the implementation of components in various programming languages. C++ and Golang were utilized in the current implementation, capitalizing on their performance and concurrency features. However, the architecture is flexible enough to accommodate other programming languages and

data persistence technology, as further details are provided in the sequel.

3.2.2.1 Microservices

The backend microservices are integral to the SSL VAR system, each serving a specific yet interconnected role essential for the effective operation of the multi-robot system. This subsection delves into the functionalities of the Perception, Referee, and Playback microservices, which collectively manage critical data processing and system coordination tasks. The Perception microservice is responsible for capturing and analyzing visual data from the external environment, the Referee microservice handles game-related information and enforces game rules, and the Playback microservice oversees the management and retrieval of match data, supporting functionalities such as live game streaming and replay capabilities. This modular design facilitates specialized management of each microservice's functions, enhancing the system's efficiency and adaptability.

The system is designed around an Event-Driven Architecture, utilizing asynchronous communication through an inter-process communication (IPC) protocol with a publisher-subscriber pattern enabled by ZeroMQ (ZMQ) [33]. ZMQ is selected for its exceptional performance, lightweight footprint, and fine-grained control, making it particularly well-suited for low-latency applications. This event-driven architecture allows each microservice to function independently while ensuring seamless event propagation and distribution throughout the system. For scenarios requiring synchronous communication, such as the interaction between the Playback microservice and the API Gateway for time-based match data chunks, ZMQ also facilitates communication using the dealer-router pattern. Unlike other middleware systems commonly employed in robotics, ZMQ offers performance optimization and control ideal for real-time systems' stringent demands. Additionally, the system employs Protocol Buffers [36] for serializing all inputs and outputs, ensuring efficient and consistent data processing across the architecture.

To minimize communication overhead and enhance performance within the system, several strategies were implemented, particularly in the Perception and Referee services. A crucial element of this optimization was the adoption of a producer-consumer design pattern complemented by multi-threading. In this architecture, the producer controller efficiently manages input data by utilizing a poller [86] to aggregate messages from multiple sources at specified intervals. Instead of sending individual messages as they arrive, the producer consolidates these inputs into a single payload and forwards it to the consumer controller for processing. This approach allows the consumer controller to focus solely on processing functionality, eliminating the need to handle incoming message traffic directly.

This design helped reduce the communication overhead by preventing the consumer controller from being interrupted by each new message, thus enabling more efficient processing. Additionally, it prevented the message queue between the producer and consumer controllers from becoming overcrowded, which could otherwise lead to delays and negatively impact real-

time performance. By efficiently managing the flow of data between components, this approach helped ensure the system maintained low latency while processing incoming data in a timely manner.

Additionally, the microservices architecture inherently supports polyglot persistence, enabling each service to select a database tailored to its specific requirements, optimizing performance and data management. The choice of an appropriate Database Management System (DBMS) is crucial. It must be informed by the unique needs of each microservice, with a particular emphasis on achieving optimal read and write performance.

NoSQL databases offer distinct advantages in handling unstructured data efficiently, allowing for rapid read/write operations in various formats, such as JSON documents or key-value pairs. Among the prominent NoSQL databases -— Redis, MongoDB, and Cassandra -— each is suited to different use cases. While graph-based NoSQL databases excel in applications centered on complex data relationships, they are less suited to scenarios like the one addressed in this work, where rapid processing of unstructured data is paramount [70]. MongoDB, as a document-oriented database, typically provides faster read/write operations than Cassandra, a column-based database, in most scenarios [2]. However, Redis, a key-value store, surpasses MongoDB and Cassandra in speed, leveraging in-memory data storage to deliver exceptionally high-speed operations [74]. Redis was selected as a proof of concept for this architecture due to its outstanding performance in low-latency, high-throughput environments.

The final architecture, illustrated in Figure 20, encompasses critical components such as Perception, Referee, Decision, Behavior, Navigation, and Communication. The SSL VAR is a subset of this architecture; the components needed for building the SSL VAR system are detailed in Figure 21.

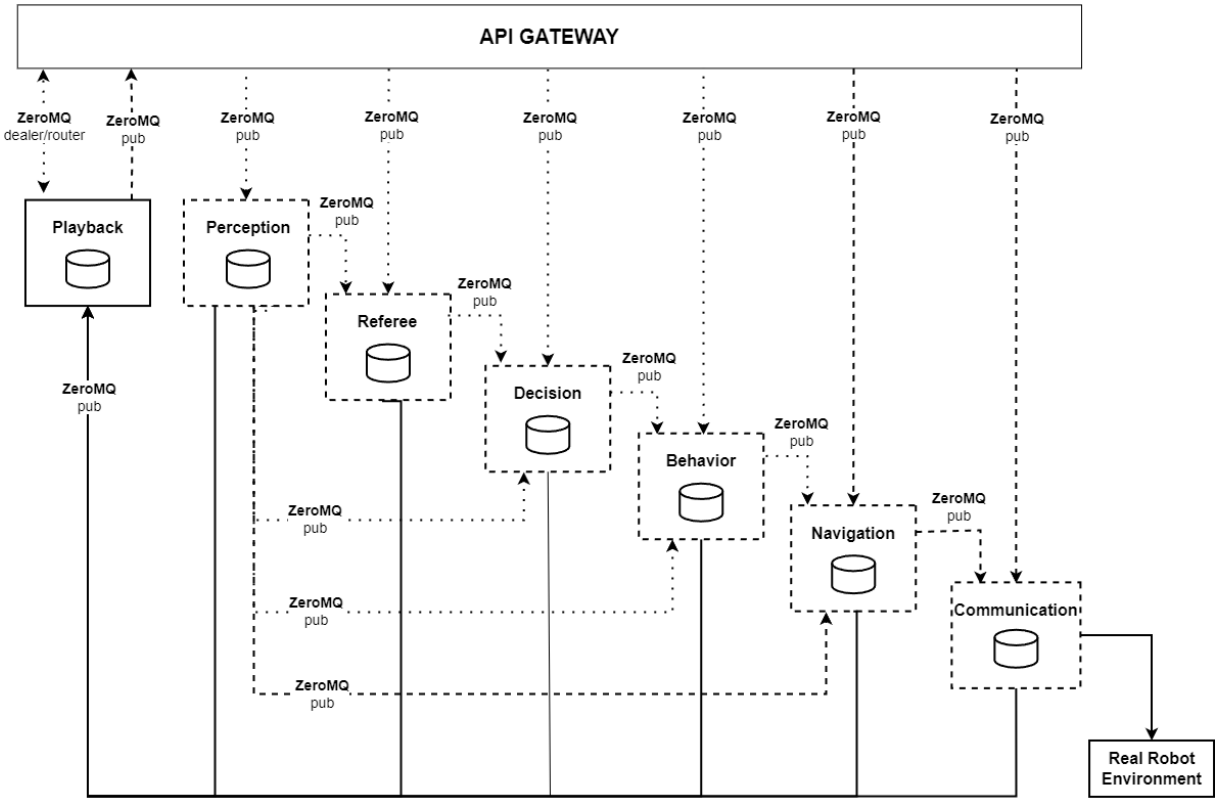

Figure 20: The complete microservices-based architecture for SSL control software. Source: The author.

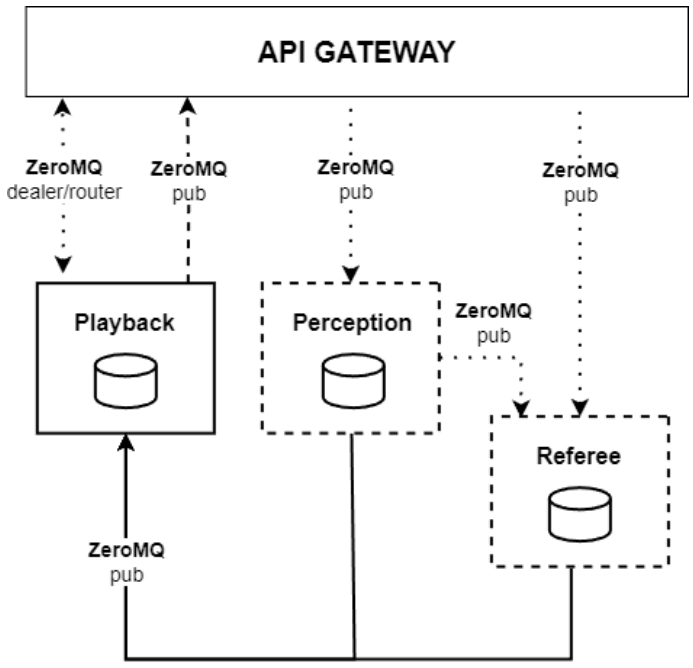

Figure 21: The microservices-based architecture for SSL VAR system. Source: The author.

3.2.2.2 API Gateway

The API Gateway is a critical interface within the SSL VAR system, managing the interactions between internal microservices and external components. Its role is to abstract the complexities of service communication and provide a cohesive entry point for various requests. The API Gateway primarily utilizes the facade design pattern [27] to enable communication between systems and components operating with different technologies. Internally, the gateway uses a modular approach to handle communication channels independently, ensuring efficient and scalable performance.

The gateway architecture is designed to handle multiple communication protocols concurrently, ensuring seamless integration and efficient data flow between different system components. It features UDP multicast workers that listen for messages from various external sources, including External Vision [88, 57, 67], TIGERs AutoReferee [77], and the ssl-game-controller [66]. Each worker is assigned a specific multicast address and processes incoming data, forwarding it to a central proxy channel for further handling.

Beyond UDP multicast, the gateway integrates a ZeroMQ server that serves as a broker, facilitating the conversion and routing of messages from UDP to ZeroMQ. This ensures efficient message delivery across the system while maintaining the integrity of the communication protocol and package structure. The gateway effectively manages the serialization, deserialization, and routing of messages to the appropriate microservices through its multi-protocol framework.

Additionally, the gateway includes a Google Remote Procedure Call (gRPC) [34] server to manage synchronous communication with external clients, enabling real-time interactions. gRPC is a middleware based on remote procedure calls commonly used for client-server communication, ensuring a common interface between components through the language of Protocol Buffers. gRPC provides a standardized interface and efficiently supports various communication needs, including unary and streaming requests/responses. Therefore, given the need for specific functionalities such as timestamp-based and game live requests, it is an ideal fit for the particular domain of interest.

The comprehensive microservices-based architecture, which integrates frontend, gateway, backend, and external components, is illustrated in Figure 22. This figure represents the system's design and the communication protocols that enable seamless interaction between the elements. The architecture's scalability is evident in its fully decoupled structure, where each component communicates through well-defined messaging protocols, ensuring flexibility and ease of expansion. This architecture is particularly suited for real-time applications like robot soccer, where the ability to handle multiple communication streams with low latency is crucial.

3.2.2.3 Infrastructure

The development and deployment of the SSL VAR system are built on a robust, platform-independent infrastructure through Docker [47]. Docker allows developers to package applica-

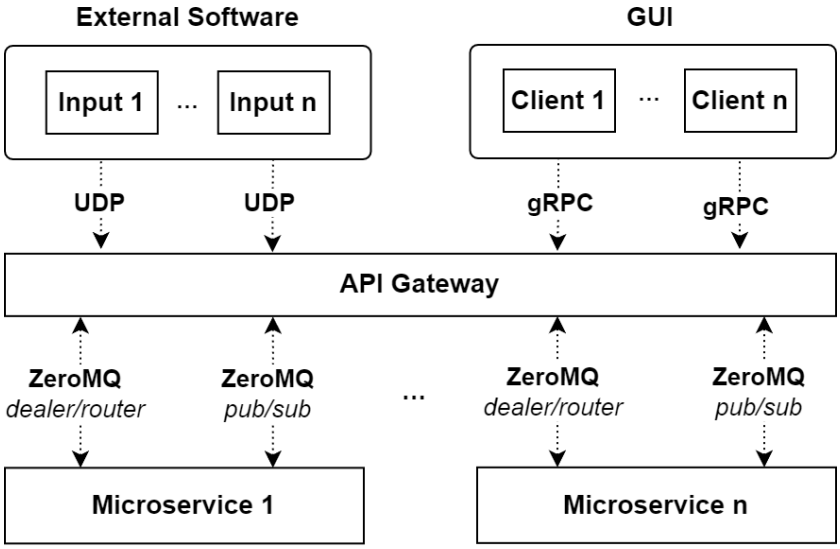

Figure 22: The general microservices-based architecture with generic components. Source: The author.

tions and their dependencies into lightweight, portable containers, ensuring that the software behaves consistently across different environments. This is particularly beneficial in dynamic environments like RoboCup, where various system components must interact seamlessly.

The development work was performed using Visual Studio Code [49], integrated with a Visual Studio Code Dev Containers [50] setup. Devc Containers provide a standardized development environment that can be easily shared among team members. By defining a development container, developers can ensure that everyone works with the same dependencies and tools, regardless of their local machine setup. This not only simplifies the onboarding process for new developers but also accelerates the development cycle, as the environment setup time is minimized.

To optimize the build process, Dockerfiles for the SSL VAR system were designed to support Docker multi-stage builds [20]. Multi-stage builds enable developers to create separate build stages within a single Dockerfile, allowing them to efficiently manage the creation of both development and production images. The choice of lightweight base images, such as Distroless [30] and Scratch [21], significantly minimizes resource consumption, improving both the performance and scalability of the system. Table 3 summarizes the resource usage of the implemented services and third-party containers.

To optimize the build process for the SSL VAR system, Dockerfiles were designed to leverage Docker multi-stage builds [20]. Multi-stage builds enable developers to create distinct build stages within a single Dockerfile, which facilitates the efficient management of both development and production images. This approach allows for a clear separation of the build environment from the runtime environment, ensuring that only the necessary components are included in the final production image. By minimizing the size of the image and eliminating unnecessary files, developers can significantly decrease build times and simplify deployment processes. Fur-

thermore, this streamlined workflow promotes better organization within the codebase, making it easier to manage dependencies and updates over time. Selecting lightweight base images, such as Distroless [30] and Scratch [21], is crucial as it significantly reduces resource consumption, enhancing both the performance and scalability of the system.

Table 3 summarizes the resource usage of the implemented service container images compared to commonly used third-party containers in the RoboCup SSL community, including TIGERs AutoReferee [77] and the simulator-cli [67]. Notably, the proposed service images demonstrate substantially better resource usage than the currently employed third-party containers.

Table 3: Resource Usage of SSL VAR and Third-Party Containers

Source	**Component**	**Image Size (MB)**	**Base Image**
SSL VAR	Gateway	28.5	Distroless
	Perception	7.79	Scratch
	Referee	9.2	Scratch
	Playback	34.1	Distroless
Third-Party	TIGERS AutoReferee	610	N/A
	simulator-cli	471	N/A

4

EVALUATION

This chapter analyzes the performance impacts associated with communication and processing within the microservice architecture pipeline, as outlined in earlier chapters. It examines how interactions among various components within a microservices-based architecture influence the system's latency.

To ensure the scalability and efficiency of the SSL VAR system's architecture, it was crucial to conduct preliminary experiments focusing on the communication overhead within the microservices architecture. This experiment assessed the system's ability to handle communication loads, simulating conditions encountered in a fully operational strategy pipeline for a future and complete strategy software for robot control in RoboCup SSL games. By evaluating the communication efficiency and latency under simulated conditions, the experiment aimed to establish whether the architecture could support the stringent real-time requirements of the category games. Understanding the communication overhead is crucial, as it directly impacts the overall system's ability to process and relay information promptly, ensuring that the architecture can handle the complexities of a fully integrated strategy pipeline in future implementations.

A performance analysis [35] was conducted to evaluate a microservice architecture (Figure 23) within the context of RoboCup SSL competitions. Although not all components were fully implemented, this does not impact the analysis, as the primary focus was to confirm that the architecture could meet the demanding performance criteria necessary for timely and efficient command generation despite the communication overhead due to the distributed architecture. The evaluation was performed on a system configured with Ubuntu 20.04, featuring an Intel® Core™ i7-8565U CPU @ 1.80GHz × 8, 16 GB of RAM, and a 256 GB SSD. Docker containers were employed to orchestrate the environment, providing a consistent and isolated setup for each test. The experiments involved simulating a runtime workload with 3000 messages. To ensure the precision of the results, the initial and final 1000 messages were excluded, thereby concentrating the analysis on the middle 1000 messages that offer a clear view of the system's core performance metrics.

The primary focus of this experiment was to validate the essential architectural flow of the proposed architecture, with particular attention to robot strategy processing. These components were selected due to their critical role in real-time operations during robot soccer matches.

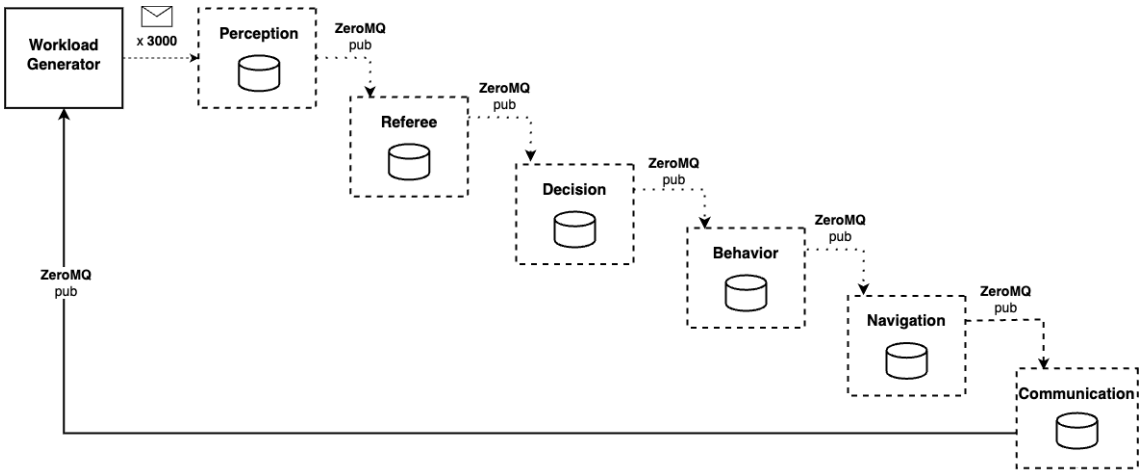

Figure 23: The proposed architecture's pipeline used for evaluating communication overhead. Source: The author.

The primary performance metric was the total duration required for the pipeline to receive, process, and transmit external input as robot commands. This metric provides insight into the efficiency and responsiveness of the microservice architecture, highlighting its ability to meet the stringent latency requirements of real-time scenarios.

A baseline latency of 16 milliseconds (ms) was established to assess performance, corresponding to the 60 Hz frame rate at which SSL Vision [88] transmits images in real-time SSL games. This benchmark was crucial in determining whether the proposed architecture could handle third-party data processing and generate robot commands within the required time frame, considering the communication overhead. This metric provides insight into the efficiency and responsiveness of the microservice architecture, highlighting its ability to meet the stringent latency requirements of real-time scenarios.

As illustrated in Figure 24, the proposed architecture achieved a maximum communication overhead of 0.43 ± 0.03 ms in the most critical scenario, which represents 0.03% of the given processing budget (16 ms). This level of overhead falls well within acceptable limits, and shows the efficiency of the microservice architecture in managing communication between its various components. This minimal overhead reflects the system's capability to handle the intricate interactions among microservices without introducing significant delays.

This finding shows that a microservices approach can be used effectively without compromising the RoboCup SSL latency, keeping the robustness and reliability of the microservice architecture. The ability of the system to adhere to stringent performance requirements confirms its suitability for compelling gameplay. By consistently generating and transmitting commands within the precise time constraints imposed by the game's dynamic nature, the architecture demonstrates its readiness for real-world application and its capability to support high-performance requirements in competitive scenarios. This ensures that the system can deliver timely and accurate responses crucial for successful gameplay, validating the efficacy of the architectural design and its implementation.

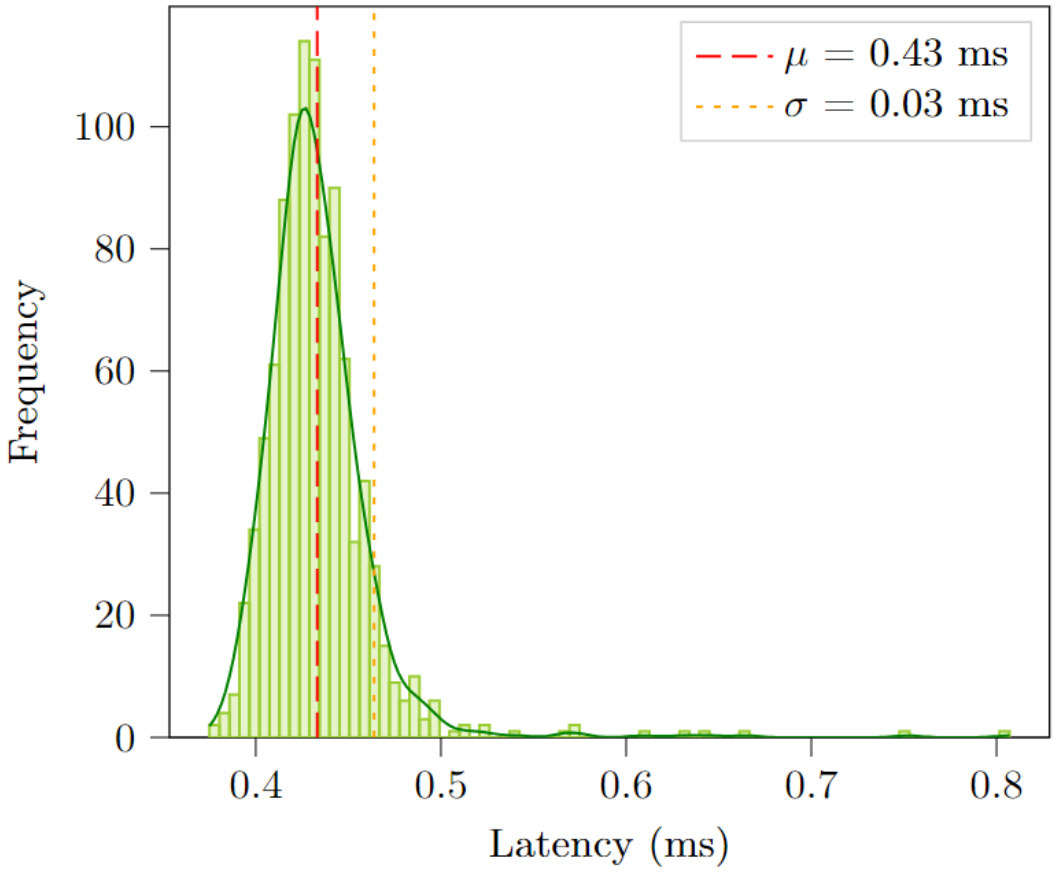

Figure 24: Distribution of pipeline latency for complete robot strategy architecture based only on communication overhead. Source: The author.

5

RELATED WORK

The evolution of software architectures in MRS has seen a significant shift from traditional monolithic designs to more modular and scalable approaches, such as microservices. Historically, monolithic architectures have been favored for their simplicity and performance advantages, particularly in applications where low latency and high efficiency are paramount. This preference has been evident in the SSL of robot soccer, where teams have traditionally employed monolithic systems to control their robots [3, 59, 82].

The Robot Operating System (ROS) has emerged as a pivotal framework in the development of distributed architectures for robotics systems, providing a versatile platform that supports complex and modular designs [79]. Expanding on ROS's capabilities, Rahimi *et al.* [62] introduces a microservices architecture specifically crafted for the robot soccer domain. However, despite its popularity, ROS presents several significant drawbacks. First, ROS can be overly complex and cumbersome, requiring substantial effort for codification and maintenance. This complexity can lead to increased development time and difficulty in debugging. Second, ROS introduces considerable overhead, which can degrade performance, particularly in real-time applications with critical low latency requirements. Third, the tightly integrated nature of ROS components can make it challenging to adopt new technologies or experiment with alternative approaches without significant rework. Due to these limitations, previous studies in the SSL domain have opted to discontinue using ROS [1, 73].

However, as robotic systems' complexity has increased, more flexible and scalable solutions are required. The field of MRS has seen significant exploration and innovation in software architectures. The service-oriented approach has become a foundational element in robotics, offering key modularity, scalability, and interoperability advantages. Trifa *et al.* [78] demonstrates how service technologies can streamline communication and control, mainly through standardized interfaces in a swarm of robots, facilitating seamless tele-control. Similarly, Chen *et al.* [16] proposes the design of robots or devices as SOA units capable of executing a broad range of robotic functions, thereby enhancing the versatility and integration potential of these systems. Narita *et al.* [52] highlights the RoboLink Protocol, which employs Web Services communication standards to ensure reliable inter-robot communication.

Several studies have explored the integration of microservices within robotics, high-

lighting its potential to enhance system efficiency and scalability. For instance, Georgiades *et al.* [29] propose a microservice-based software framework for multi-drone autonomous systems that is fault-tolerant, expandable, and easily monitored. Additionally, Schäffer *et al.* [72] illustrate a microservice architecture for a web-based configurator for robot-based automation solutions, demonstrating how this approach enables functionalities such as collaborative multi-user configuration and role-based configuration sessions. Furthermore, Xia *et al.* [84] propose a microservice-based architecture for cloud robotics in intelligent spaces, utilizing container technology to provide a highly efficient and flexible development/deployment mechanism and cost-effectiveness compared to traditional monolithic architectures. Zhou *et al.* [87] introduces a virtual microservices-based model for robotic swarm systems, demonstrating how the approach simplifies task composition and swarm control by focusing on high-level virtual services rather than managing individual robot actions, as illustrated through a detailed rescue mission case study. Kudriashov [39] proposes a distributed architecture to robotics systems named ALKETON. It designs two significant conceptual components to reduce the coupling of microservices and improve scalability. Similarly, Schäffer *et al.* [71] propose a concept of microservices architecture with standardized robot configurators, including 2D and 3D visualizations, among other functionalities. However, these works do not specify latency and performance analysis based on overall communication overhead, primordial to low-latency MRS.

The transition to microservices architectures in robotics represents a significant advancement in addressing the growing need for modularity, scalability, and system flexibility. Microservices enable the independent development, deployment, and scaling of components, making integrating new technologies and adapting to evolving requirements easier. This modular approach enhances robotic systems' overall resilience and maintainability, offering a more robust foundation for complex and dynamic environments. While the benefits of microservices are clear, it remains essential to ensure that these advantages are not undermined by potential increases in communication overhead. Careful design and optimization are key to maintaining the low-latency performance required for critical robotics applications, ensuring that the full potential of microservices can be realized in advancing the capabilities of modern robotic systems.

6

CONCLUSION

This work introduces SSL VAR, a cutting-edge microservices architecture explicitly designed for low-latency MRS, catering to the intricate requirements of the RoboCup SSL environment. The SSL VAR framework features a Video Assistant Referee (VAR) system, which employs a microservices-based approach to deliver essential functionalities. Key features include dedicated microservices for perception and referee processing, enabling precise and real-time analysis of the robot's environment and game conditions. This modular design supports the independent development and scaling of these specialized components, allowing for the integration of various technologies and ensuring a highly flexible system.

Furthermore, SSL VAR provides robust support for live match data streaming and replay functionalities. The architecture includes capabilities for real-time game streaming and time-based query support, which is crucial for implementing replay systems and other external components. This provides integration with external applications, providing detailed match insights and historical data retrieval. An API Gateway effectively manages communication between the microservices and external components, facilitating seamless data exchange and system integration. The architecture's layered design ensures adaptability and scalability while adhering to strict low-latency performance requirements, thus enhancing the overall efficiency and responsiveness of the system.

The results of this study demonstrate that the microservices approach can be effectively utilized without compromising performance and that the careful selection and implementation of suitable technologies can enable successful real-time applications. The proposed architecture represents a significant advancement in robot competitions, showcasing the potential of microservices and innovative software patterns in a challenging, performance-critical environment. The successful implementation of SSL VAR in real-time scenarios underscores its potential for broader application and highlights the advantages of a microservices architecture in complex, high-stakes settings.

Future work will focus on transitioning RobôCIn's [68] SSL software from its current monolithic architecture [3] to the microservices-based architecture proposed in this document. This transition will involve a more detailed performance analysis to optimize latency and ensure the system's long-term scalability and efficiency. A comparative analysis of the microservices

architecture against the existing monolithic structure is also a critical next step. This will involve conducting quantitative performance evaluations to assess key metrics such as latency, throughput, and system responsiveness, aiming to validate the improvements brought by microservices over the monolithic approach.

Furthermore, the work will explore the system's scalability under more demanding competition scenarios. The transition will consider how the architecture can handle increased complexity and workload while maintaining real-time performance. By analyzing scalability in greater detail, future enhancements can be better guided to ensure that the system is robust and efficient in the face of higher demands. Another critical area for future work is the evaluation of system robustness. The ability of the system to handle failures or errors in a highly dynamic, real-time environment will be rigorously tested, focusing on fault tolerance and error recovery strategies. This will ensure the architecture maintains stability and functionality in high-pressure competition scenarios.

Regarding software modeling, one potential area for improvement in the architectural modeling of this system is the transition from the UML to the C4 Model [15] for representing software architecture. While UML is a traditional and widely adopted tool for software design, it can sometimes introduce unnecessary complexity, particularly in systems involving microservices. Given the need for both high-level abstraction and clear communication in a fast-paced, real-time system, adopting the C4 Model could provide several key advantages. The C4 Model offers a clear, structured approach that emphasizes the relationships and interactions between components, making it easier for teams to understand the architecture without getting lost in the details. It promotes clarity and focus by providing different levels of abstraction – from context to component diagrams – allowing for high-level and detailed views as needed. This could enhance communication among stakeholders and streamline the development process, ultimately supporting better collaboration in a rapidly evolving environment.

Lastly, the integration of the microservices architecture with both frontend and backend components will progress as an open-source project under the RobôCIn initiative [1]. The design of a complementary microservices architecture for the frontend is currently under development [80], and the combination of these efforts aims to significantly enhance the system's overall functionality and performance, making it better suited for the stringent demands of RoboCup SSL competition. Figure 25 presents the target architecture, providing a comprehensive overview of the interactions among the various services within the system.

[1] Available at `https://github.com/robocin/ssl-core`

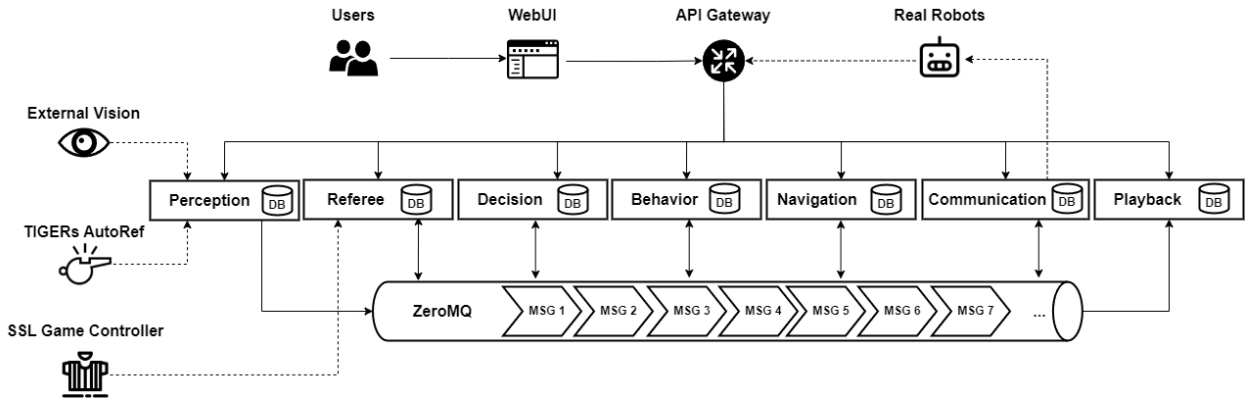

Figure 25: The final microservices-based architecture aimed for a future work. Source: The author.

REFERENCES

[1] Almoallima, A., Sousa, C., Sturn, D., Antoniuk, D., Cremad, F., Bryant, H., Lew, J., Liu, J., Wakaba, K., Bontkes, L., & Zhou, Y. (2022). 2022 team description paper: Ubc thunderbots.

[2] Andor, C.-F. (2021). Runtime metric analysis in nosql database performance benchmarking. In *2021 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 1–6.

[3] Araújo, V., Rodrigues, R., Cruz, J., Cavalcanti, L., Andrade, M., Santos, M., Melo, J., Oliveira, P., Morais, R., & Barros, E. (2024). Robôcin ssl-unification: A modular software architecture for dynamic multi-robot systems. In Buche, C., Rossi, A., Simões, M., & Visser, U., editors, *RoboCup 2023: Robot World Cup XXVI*, 313–324.

[4] Astah (2022). Service oriented architecture modeling language. `https://astah.net/support/astah-pro/user-guide/sequence-diagram/#combined`. Accessed: September 10, 2024.

[5] Aziz, O., Farooq, M. S., Abid, A., Saher, R., & Aslam, N. (2020). Research trends in enterprise service bus (esb) applications: A systematic mapping study. *IEEE Access*, 8:31180 – 31197. Cited by: 26; All Open Access, Gold Open Access.

[6] Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52.

[7] Banavar, G., Chandra, T., Strom, R., & Sturman, D. (1999). A case for message oriented middleware. In Jayanti, P., editor, *Distributed Computing*, 1–17.

[8] Bean, J. (2010). *SOA and Web Services Interface Design: Principles, Techniques, and Standards*. Morgan Kaufmann OMG Press. Morgan Kaufmann.

[9] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., & Thomas, D. (2001). Manifesto for agile software development.

[10] Bellemare, A. (2020). *Building Event-driven Microservices: Leveraging Organizational Data at Scale*. O'Reilly Media.

[11] Bennett, S., McRobb, S., & Farmer, R. (2005). *Object-oriented systems analysis and design using UML*. McGraw Hill Higher Education.

[12] Bernstein, P. A. (1996). Middleware: a model for distributed system services. *Communications of the ACM*, 39(2):86–98.

[13] Blaha, M. & Rumbaugh, J. (2005). *Object-oriented Modeling and Design with UML*. Pearson Education.

[14] Brambilla, M., Cabot, J., & Wimmer, M. (2017). *Model-driven software engineering in practice*. Morgan & Claypool Publishers.

[15] Brown, S. (2018). Software architecture for developers-volume 2: Visualise, document and explore your software architecture. *Victoria, British Columbia, Canada: Leanpub*.

[16] Chen, Y., Du, Z., & Garcia-Acosta, M. (2010). Robot as a service in cloud computing. In *2010 Fifth IEEE International Symposium on Service Oriented System Engineering*, 151–158.

[17] Conway, M. E. (1968). How do committees invent. *Datamation*, 14(4):28–31.

[18] Delligatti, L. (2013). *SysML Distilled: A Brief Guide to the Systems Modeling Language*. Addison-Wesley Professional, 1st edition.

[19] Di Francesco, P., Lago, P., & Malavolta, I. (2019). Architecting with microservices: A systematic mapping study. *Journal of Systems and Software*, 150:77–97.

[20] Docker, Inc. (2013). Multi-stage builds. `https://docs.docker.com/build/building/multi-stage`. Accessed: October 22, 2024.

[21] Docker, Inc. (2024). Scratch. `https://hub.docker.com/_/scratch`. Accessed: October 22, 2024.

[22] DZone (2017). Microservices vs. soa – is there any difference at all? `https://dzone.com/articles/microservices-vs-soa-is-there-any-difference-at-al`. Accessed: August 23, 2024.

[23] Evans, E. (2004). *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

[24] Federation, R. (2015). About robocup small size league (ssl). `https://ssl.robocup.org/about/`. Accessed: January 21, 2024.

[25] Ford, N., Parsons, R., Kua, P., & Sadalage, P. (2022). *Building Evolutionary Architectures*. O'Reilly Media.

[26] Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA.

[27] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns*. Addison Wesley, Boston, MA.

[28] Geiling, N. (2014). Robocup: Building a team of robots that will beat the world cup champions. `https://www.smithsonianmag.com/innovation/robocup-building-team-robots-will-beat-world-cup-champions-180951713`. Accessed: June 11, 2024.

[29] Georgiades, C., Souli, N., Kolios, P., & Ellinas, G. (2024). Creating a robust and expandable framework for cooperative aerial robots. In *2024 International Conference on Unmanned Aircraft Systems (ICUAS)*, 1192–1199.

[30] Google (2024). Distroless. `https://github.com/GoogleContainerTools/distroless`. Accessed: October 22, 2024.

[31] Group, O. *et al.* (2011). Business process model and notation (bpmn) version 2.0. *OMG Standard*.

[32] Hernandez-Aparicio, C. C., Ocharan-Hernandez, J. O., Cortes-Verdin, K., & Arenas-Valdes, M. A. (2022). Architectural languages for the microservices architecture: A systematic mapping study. In *2022 10th International Conference in Software Engineering Research and Innovation (CONISOFT)*, 192–201.

[33] Hintjens, P. (2013). *ZeroMQ*. O'Reilly Media, Sebastopol, CA.

[34] Indrasiri, K. & Kuruppu, D. (2020). *gRPC: Up and Running: Building Cloud Native Applications with Go and Java for Docker and Kubernetes*. O'Reilly Media.

[35] Jain, R. (1991). *The art of computer systems performance analysis*. John Wiley & Sons, Nashville, TN.

[36] Jean, C. (2024). *Protocol Buffers Handbook: Getting Deeper Into Protobuf Internals and Its Usage*. Packt Publishing.

[37] Khononov, V. (2021). *Learning Domain-driven Design: Aligning Software Architecture and Business Strategy*. O'Reilly Media, Incorporated.

[38] Kitano, H., Asada, M., Kuniyoshi, Y., Noda, I., & Osawa, E. (1997). Robocup: The robot world cup initiative. In *International Conference on Autonomous Agents*.

[39] Kudriashov, A. V. (2022). Two-tier architecture of the distributed robotic system «alketon». *Procedia Computer Science*, 213:816–823. 2022 Annual International Conference on Brain-Inspired Cognitive Architectures for Artificial Intelligence: The 13th Annual Meeting of the BICA Society.

[40] Lilis, G. & Kayal, M. (2018). A secure and distributed message oriented middleware for smart building applications. *Automation in Construction*, 86:163–175.

[41] Lloyd, W., Ramesh, S., Chinthalapati, S., Ly, L., & Pallickara, S. (2018). Serverless computing: An investigation of factors influencing microservice performance. In *Serverless Computing: An Investigation of Factors Influencing Microservice Performance*, 159–169.

[42] Maisto, S. A., Di Martino, B., & Nacchia, S. (2020). *From Monolith to Cloud Architecture Using Semi-automated Microservices Modernization*, 638–647. Springer International Publishing, Cham.

[43] Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series. Prentice Hall, Boston, MA.

[44] Mascolo, C., Capra, L., & Emmerich, W. (2002). Mobile computing middleware. *Advanced Lectures on Networking: NETWORKING 2002 Tutorials 2*, 20–58.

[45] Masse, M. (2011). *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. O'Reilly Media.

[46] Megargel, A., Shankararaman, V., & Walker, D. K. (2020). *Migrating from Monoliths to Cloud-Based Microservices: A Banking Industry Example*, 85–108. Springer International Publishing, Cham.

[47] Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239).

[48] Mesnil, J. (2014). *Mobile and Web Messaging: Messaging Protocols for Web and Mobile Devices*. O'Reilly Media.

[49] Microsoft. Visual studio code. `https://code.visualstudio.com/`. Accessed: October 22, 2024.

[50] Microsoft. Visual studio code dev containers. `https://code.visualstudio.com/docs/devcontainers/containers`. Accessed: October 22, 2024.

[51] Miles, R. & Hamilton, K. (2006). *Learning UML 2.0*. Learning Series. O'Reilly Media.

[52] Narita, M., Shimamura, M., & Oya, M. (2005). Reliable protocol for robot communication on web services. In *2005 International Conference on Cyberworlds (CW'05)*, 8–pp.

[53] Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 1st edition.

[54] Newman, S. (2019). *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Incorporated.

[55] Object Management Group (OMG) (2012). Service oriented architecture modeling language. `https://www.omg.org/spec/SoaML/About-SoaML`. Accessed: September 10, 2024.

[56] Paik, H.-y., Lemos, A. L., Barukh, M. C., Benatallah, B., & Natarajan, A. (2017). *Service Component Architecture (SCA)*, 203–250. Springer International Publishing, Cham.

[57] Parsian (2011). grsim. `https://github.com/RoboCup-SSL/grSim`. Accessed: October 15, 2024.

[58] Patle, B., Babu L, G., Pandey, A., Parhi, D., & Jagadeesh, A. (2019). A review: On path planning strategies for navigation of mobile robot. *Defence Technology*, 15(4):582–606.

[59] Perun, B., Ryll, A., Leinemann, G., Birkenkampf, P., König, C., Berthold, G., & Scheidel, S. (2013). Tigers mannheim team description for robocup 2011.

[60] Peterson, L. & Davie, B. (2003). *Computer Networks: A Systems Approach*. The Morgan Kaufmann series in networking. Morgan Kaufmann Publishers.

[61] Puder, A., Römer, K., & Pilhofer, F. (2011). *Distributed Systems Architecture: A Middleware Approach*. The MK/OMG Press. Elsevier Science.

[62] Rahimi, M., Shirazi, M. M., Arfaee, M., Gholian, M. A. N., Zamani, A. H., Hosseini, H., Chaleshtori, F. H., Moradi, N., Ahsani, A., Jafari, M., Zahedi, A., Abdollahi, P., Zolanvari, A., Azam, M., & Khosravi (2018). Parsian 2018 extended team description paper.

[63] Richards, M. & Ford, N. (2020). *Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Incorporated.

[64] Richardson, C. (2018). *Microservices Patterns: With examples in Java*. Manning.

[65] RoboCup SSL (2016). Robocup ssl ssl-log-tools. https://github.com/RoboCup-SSL/ssl-logtools. Accessed: September 10, 2024.

[66] RoboCup SSL (2019). Robocup ssl ssl-game-controller. https://github.com/RoboCup-SSL/ssl-game-controller. Accessed: September 10, 2024.

[67] Robotics Erlangen e.V. (2011). simulator-cli. https://github.com/robotics-erlangen/framework. Accessed: October 15, 2024.

[68] RobôCIn (2015). Robôcin. https://www.robocin.com.br. Accessed: October 22, 2024.

[69] RobôCIn (2024). Robocup's small size league messages. https://github.com/robocin/ssl-core/tree/f9a23b35f03ae5279c6541384d89970d5d1449a1/protocols/protocols/third_party. Accessed: September 10, 2024.

[70] Sadalage, P. J. & Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 1st edition.

[71] Schäffer, E., Pownuk, T., Walberer, J., Fischer, A., Schulz, J.-P., Kleinschnitz, M., Bartelt, M., Kuhlenkötter, B., & Franke, J. (2018). System architecture and conception of a standardized robot configurator based on microservices. In Schüppstuhl, T., Tracht, K., & Franke, J., editors, *Tagungsband des 3. Kongresses Montage Handhabung Industrieroboter*, 159–166.

[72] Schäffer, E., Mayr, A., Fuchs, J., Sjarov, M., Vorndran, J., & Franke, J. (2019). Microservice-based architecture for engineering tools enabling a collaborative multi-user configuration of robot-based automation solutions. *Procedia CIRP*, 86:86–91. 7th CIRP Global Web Conference – Towards shifted production value stream patterns through inference of data, models, and technology (CIRPe 2019).

[73] Seegemann, L., Zeug, F., Ebbighausen, P., Waldhoff, L., Westermann, M., Knackstedt, S., K¨anner, M., F¨uchsel, T., Hart, R., & Pahl, T. (2023). luhbots soccerteam description for robocup 2023. Institute of Automatic Control, Gottfried Wilhelm Leibniz University Hannover, Appelstr. 11, 30167 Hanover, Germany, soccer@luhbots.de.

[74] Seghier, N. B. & Kazar, O. (2021). Performance benchmarking and comparison of nosql databases: Redis vs mongodb vs cassandra using ycsb tool. In *2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI)*, 1–6.

[75] Snell, J., Tidwell, D., & Kulchenko, P. (2007). *Programming Web Services with SOAP*. O'Reilly Media.

[76] Thones, J. (2015). Microservices. *IEEE Softw.*, 32(1):116.

[77] TIGERs Mannheim (2016). Tigers autoreferee. https://github.com/TIGERs-Mannheim/AutoReferee. Accessed: September 10, 2024.

[78] Trifa, V. M., Cianci, C. M., & Guinard, D. (2008). Dynamic control of a robotic swarm using a service-oriented architecture. In *13th International Symposium on Artificial Life and Robotics (AROB 2008)*.

[79] Tsardoulias, E. & Mitkas, P. (2017). Robotic frameworks, architectures and middleware comparison. *arXiv preprint arXiv:1711.06842*.

[80] Uanderson Ricardo Ferreira da Silva (2024). Micro-frontend architecture for multi-robot user interfaces: A systematic approach with design rationale. .

[81] Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional, 1st edition.

[82] Wu, Y., Yin, P., Zhao, Y., Shen, Y., Tong, H., & Xiong, R. (2013). Zjunlict team description paper for robocup 2013.

[83] Wu, Z. (2014). *Service Computing: Concept, Method and Technology*. Academic Press, Inc., USA.

[84] Xia, C., Zhang, Y., Wang, L., Coleman, S., & Liu, Y. (2018). Microservice-based cloud robotics system for intelligent space. *Robotics and Autonomous Systems*, 110:139–150.

[85] Zaafouri, K., Chaabane, M., & Rodriguez, I. B. (2021). Systematic literature review on service oriented architecture modeling. In *Computational Science and Its Applications–ICCSA 2021: 21st International Conference, Cagliari, Italy, September 13–16, 2021, Proceedings, Part III 21*, 201–210.

[86] ZeroMQ (2024). Zmq poller. `http://api.zeromq.org/3-0:zmq-poll`. Accessed: October 22, 2024.

[87] Zhou, G., Zhang, Y., Bastani, F., & Yen, I.-L. (2012). Service-oriented robotic swarm systems: Model and structuring algorithms. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 95–102.

[88] Zickler, S., Laue, T., Birbach, O., Wongphati, M., & Veloso, M. (2010). Ssl-vision: The shared vision system for the robocup small size league. In Baltes, J., Lagoudakis, M. G., Naruse, T., & Ghidary, S. S., editors, *RoboCup 2009: Robot Soccer World Cup XIII*, 425–436.

A

APPENDIX

A.1 INTERACTION DIAGRAMS

This appendix provides detailed interaction diagrams for the Perception and Referee services, complementing the main discussion on system architecture. These diagrams are instrumental in visualizing the flow of data and interactions between system participants, facilitating a deeper understanding of the operational structure of the SSL VAR system.

The PerceptionService interaction diagram highlights the reception and processing of visual data from external vision systems. The PerceptionService interaction diagram outlines the sequence of interactions between system participants across three main flows:

1. **onReceiveParams**: The user triggers the process by updating system parameters through the UI. The updated parameters are received by the PerceptionService, which then updates its vision processing parameters accordingly.

2. **onReceiveVision**: The PerceptionService continuously receives raw vision data from external vision systems (e.g., SSL Vision [88], grSim [57], simulator-cli [67]). In parallel, it receives tracked vision data from TIGERs AutoReferee [77] with a pre-processed detection information.

3. **onReceiveData**: The PerceptionService continuously processes the received vision data (raw and tracked) in a loop. It takes unprocessed vision data from a queue, and if the queue is not empty, it processes the vision data and publishes the detection results. These results are then sent to both the RefereeService and the PlaybackService, ensuring that detection and vision tracking data are synchronized across the system.

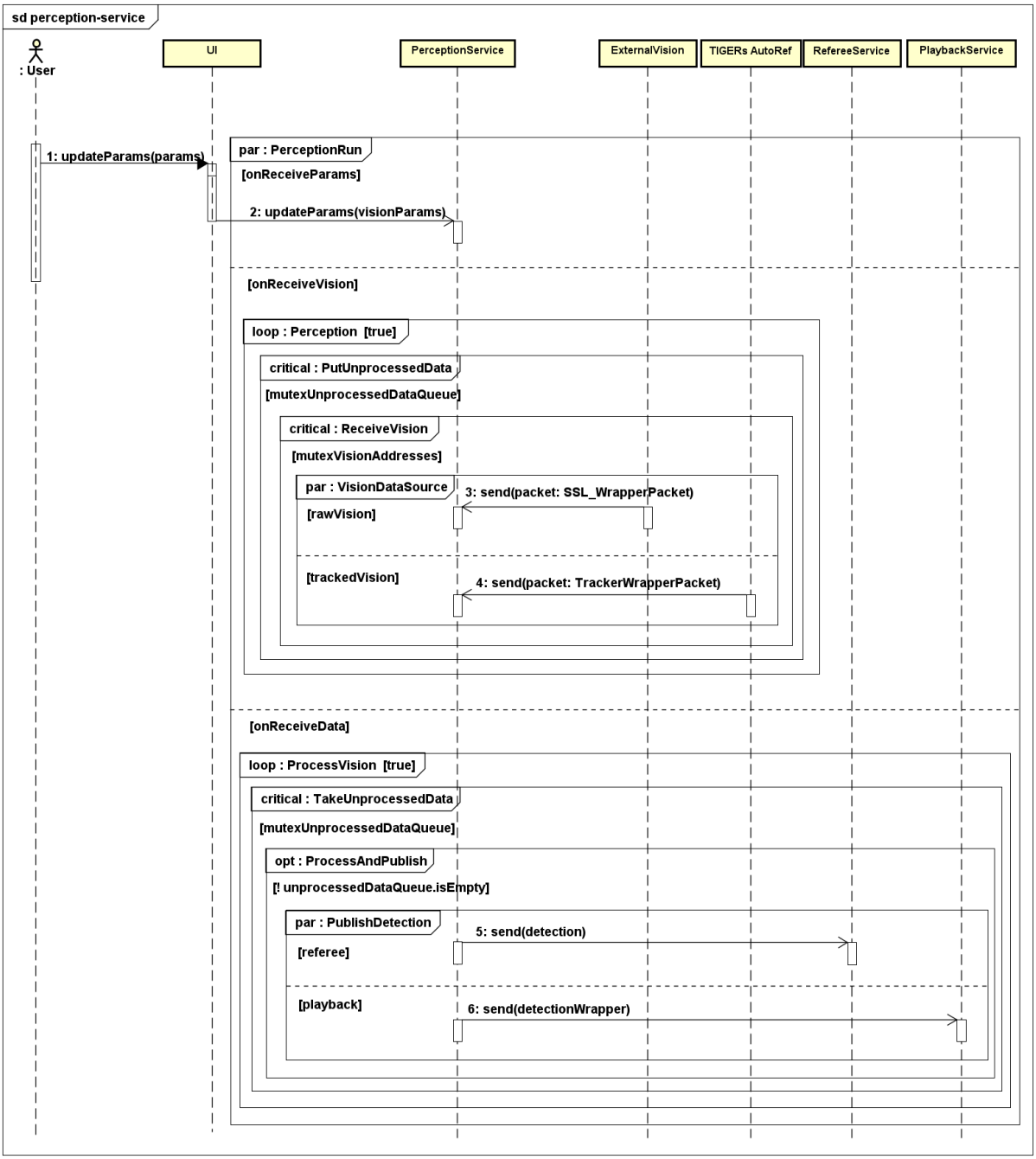

Figure 26: The interaction diagram for Perception service. Source: The author.

The RefereeService interaction diagram details the flow of game-state information processed from both referee commands and detection data from the PerceptionService. The diagram captures the sequence of interactions between system participants across three main flows:

1. **onReceiveParams**: The process is initiated by the user, who updates system parameters via the UI. These updated parameters are received by the RefereeService, which adjusts its processing behavior based on the new inputs.

2. **onReceiveMessage**: The RefereeService continuously receives referee commands from the external ssl-game-controller [66]. Simultaneously, it processes detection

data received from the PerceptionService, integrating referee commands with on-field information to maintain accurate game state awareness.

3. **onReceiveData**: The RefereeService processes the incoming data in a continuous loop. It retrieves unprocessed data from a queue, and if data is available, it processes it and publishes the resulting game status. This information is then sent to the PlaybackService, facilitating live streaming and replay functionalities.

Along with the PlaybackService interaction diagram presented in Section 3.2.1.2 (Figure 17), these interactions form the core of the system's data processing and communication steps.

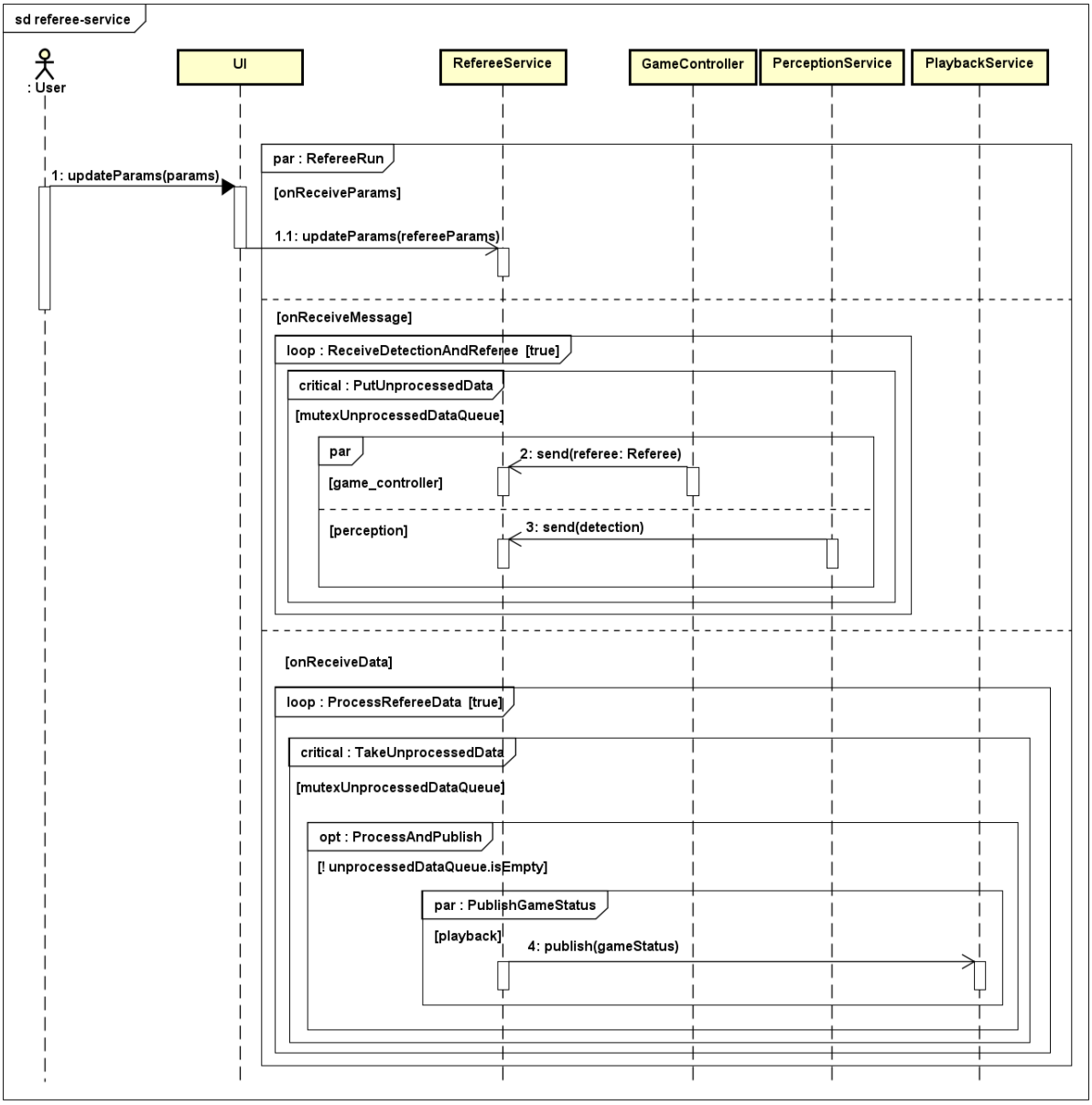

Figure 27: The interaction diagram for Referee service. Source: The author.

A.2 DETAILED SERVICES

This appendix provides detailed class diagrams for the Perception and Referee services, and the third-party values provided by the competition, complementing the main discussion on system architecture in Section 3.2.1.4. The complete detailed service diagrams for the SSL VAR system, as presented in Figures 28, 29, and 30, illustrate how interfaces and modular design principles were embedded into the system's architecture.

The third-party classes shown in the diagrams are Protocol Buffers files, standardized across the RoboCup SSL community and maintained by the competition organizers. These files have been integrated into the SSL VAR repository for version control [69]. To prevent cluttering the service diagrams with these standard classes, they have been excluded from the service diagrams and are instead presented separately in Figures 31, 32, 33 and 34.

These diagrams depict the structured interactions and relationships within the system and underscore the importance of interface-driven design in maintaining the system's adaptability and ease of integration.

Additionally, as mentioned in Section 3.2.1.4, the SSL VAR system's architecture employs several key design patterns [27] to enhance flexibility, scalability, and maintainability. The Abstract Factory pattern was implemented to create message handlers specific to middleware. This pattern enables the integration of alternative middleware solutions by defining interfaces for MessageReceiver and MessageSender and providing new factory implementations as needed. This approach ensures the system can adapt to different middleware components while preserving its architectural integrity. Moreover, the Abstract Factory pattern was applied to repository management across the service. It provides a structured approach for creating repositories with different technologies by employing a single factory interface. This allows for the consistent and efficient creation of repositories, as developers can produce families of related objects without specifying their concrete classes.

The Decorator pattern was utilized to compose various filtering algorithms for ball and robot entities on the field, including advanced techniques essential for dynamic robotics environments. These filtering algorithms enhance the system's ability to handle noisy data and distinguish between significant movements and minor disturbances. Incorporating these algorithms significantly improves tracking accuracy and state estimation while minimizing false positives in high-speed scenarios. The Decorator pattern allows for the dynamic augmentation of these filtering algorithms, enabling sophisticated filtering processes without altering the underlying structures of the filters themselves. This flexibility promotes code reusability and adaptability.

The Builder pattern was employed to construct complex data structures by applying filtering algorithms to ball and robot entities. This pattern separates the construction process of a complex object from its representation, facilitating incremental assembly through method calls. This approach is handy when dealing with multiple filtering algorithms, each focusing

on different aspects of entity behavior and applying these filters to various properties of each entity. The modular construction improves code readability and maintainability, especially in scenarios involving intricate filtering processes.

The Facade pattern was introduced to provide a simplified interface to the complex processing logic within the system. By encapsulating the details of internal operations, this pattern ensures that business logic remains decoupled from underlying complexities. For example, the ConsumerController instances interact with the system through a unified Facade interface, which handles payloads' breaking down and processing. This abstraction layer enhances modularity and ease of use, allowing developers to manage the system through a more coherent and streamlined interface.

Lastly, the Factory Method pattern was applied to create diverse game commands based on external inputs and the game environment. By defining a common interface for creating game commands and implementing concrete factories, this pattern allows for the dynamic creation of commands suited to various conditions. This flexibility is crucial for managing different game scenarios and adapting to changing environments.

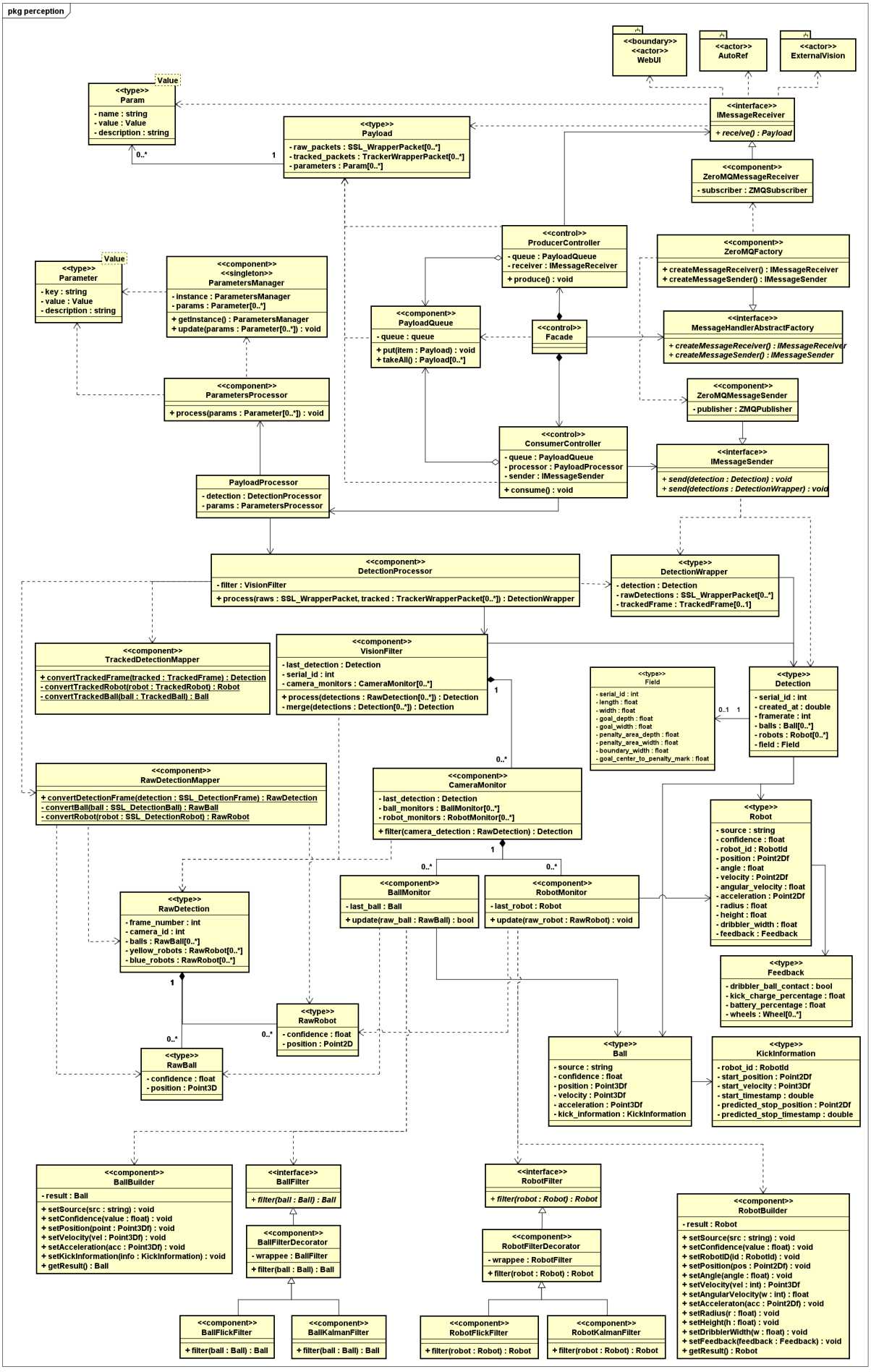

Figure 28: The class diagram for Perception service. Source: The author.

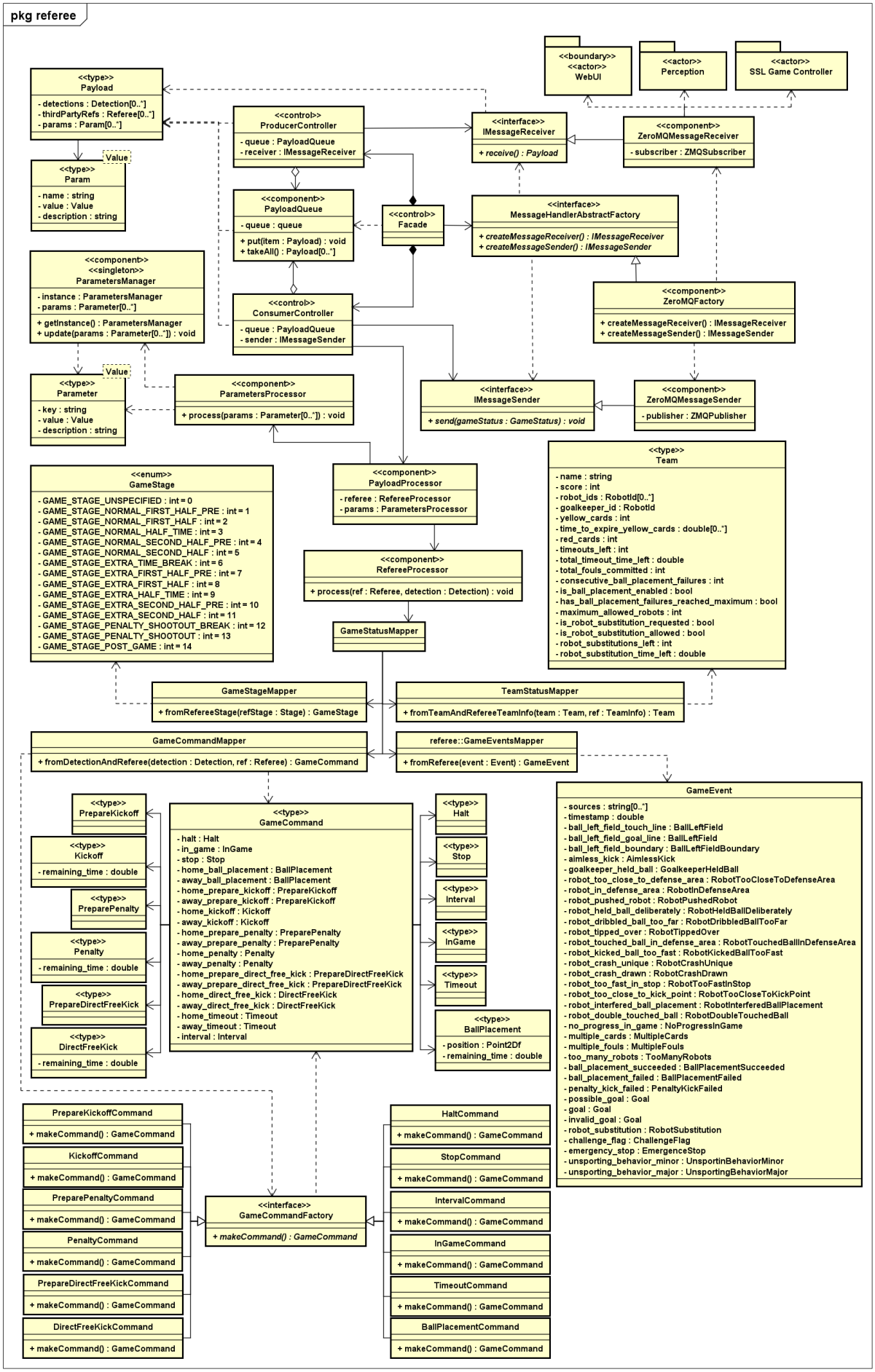

Figure 29: The class diagram for Referee service. Source: The author.

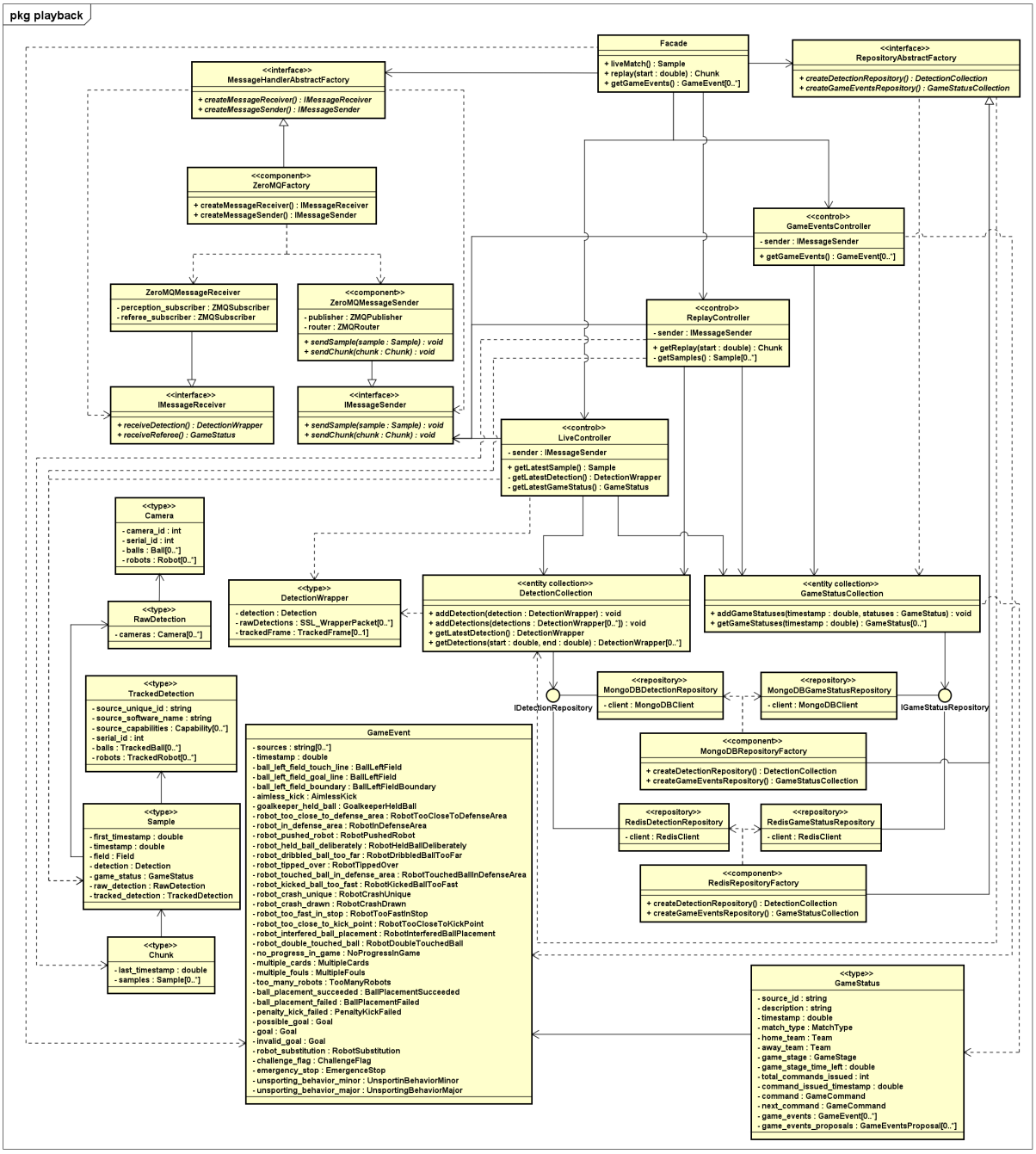

Figure 30: The class diagram for Playback service. Source: The author.

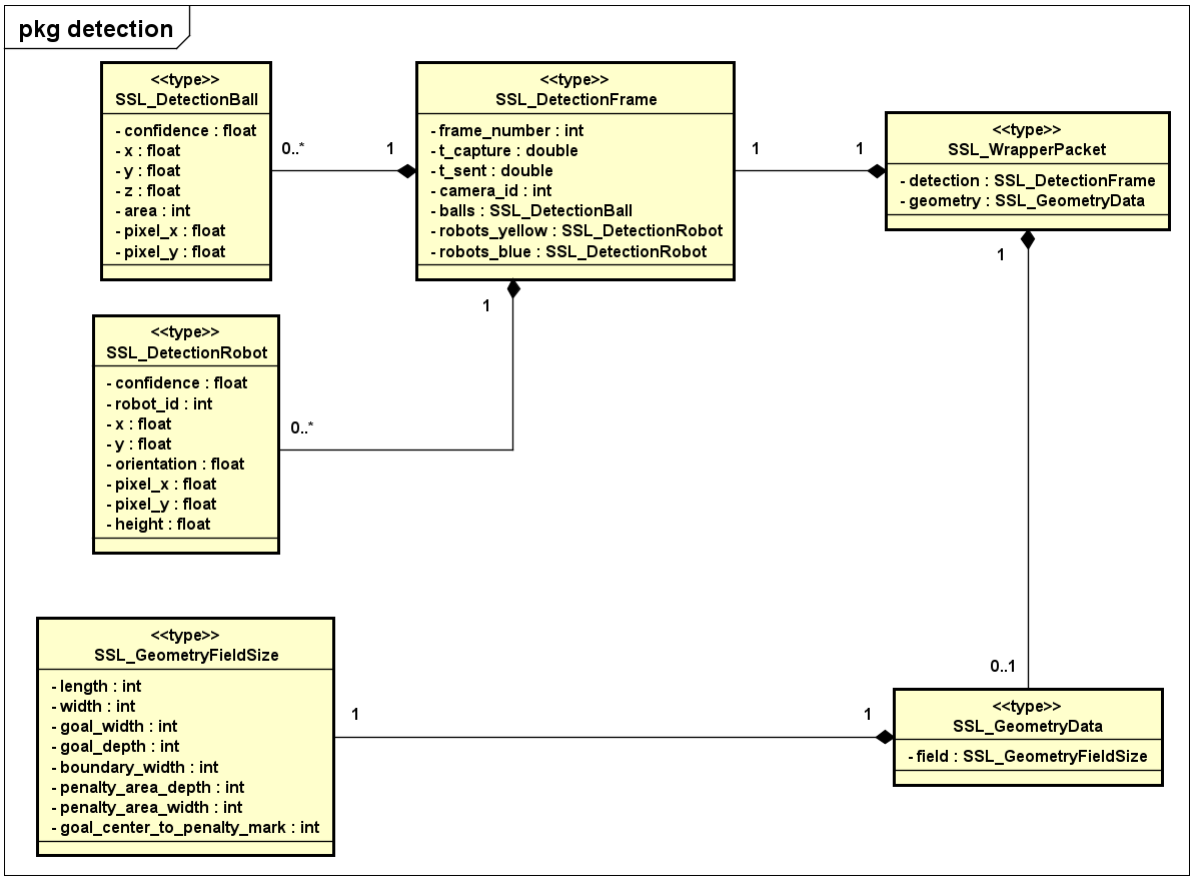

Figure 31: The class diagram for third-party detection data. Source: The author.

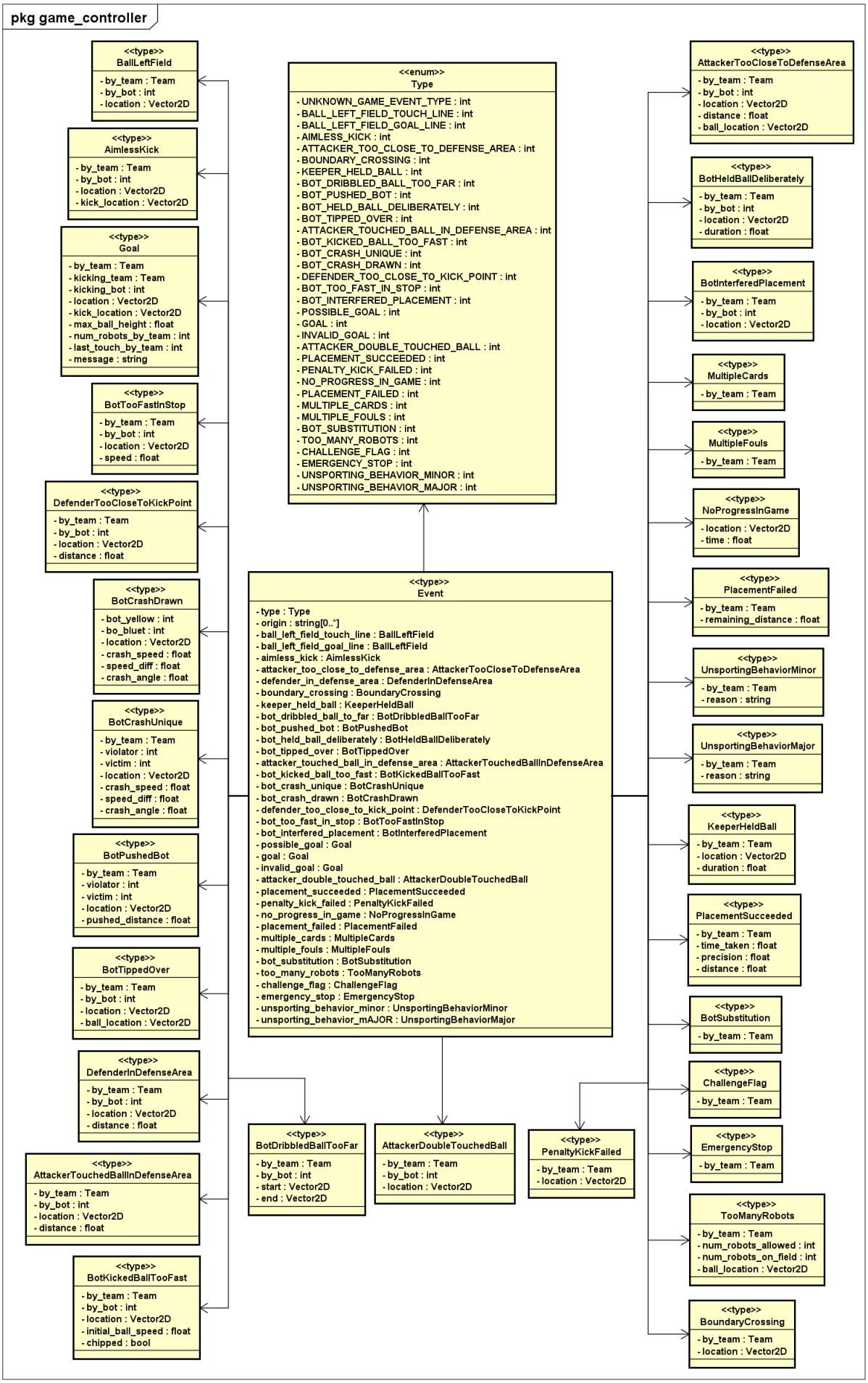

Figure 32: The class diagram for third-party game events data. Source: The author.

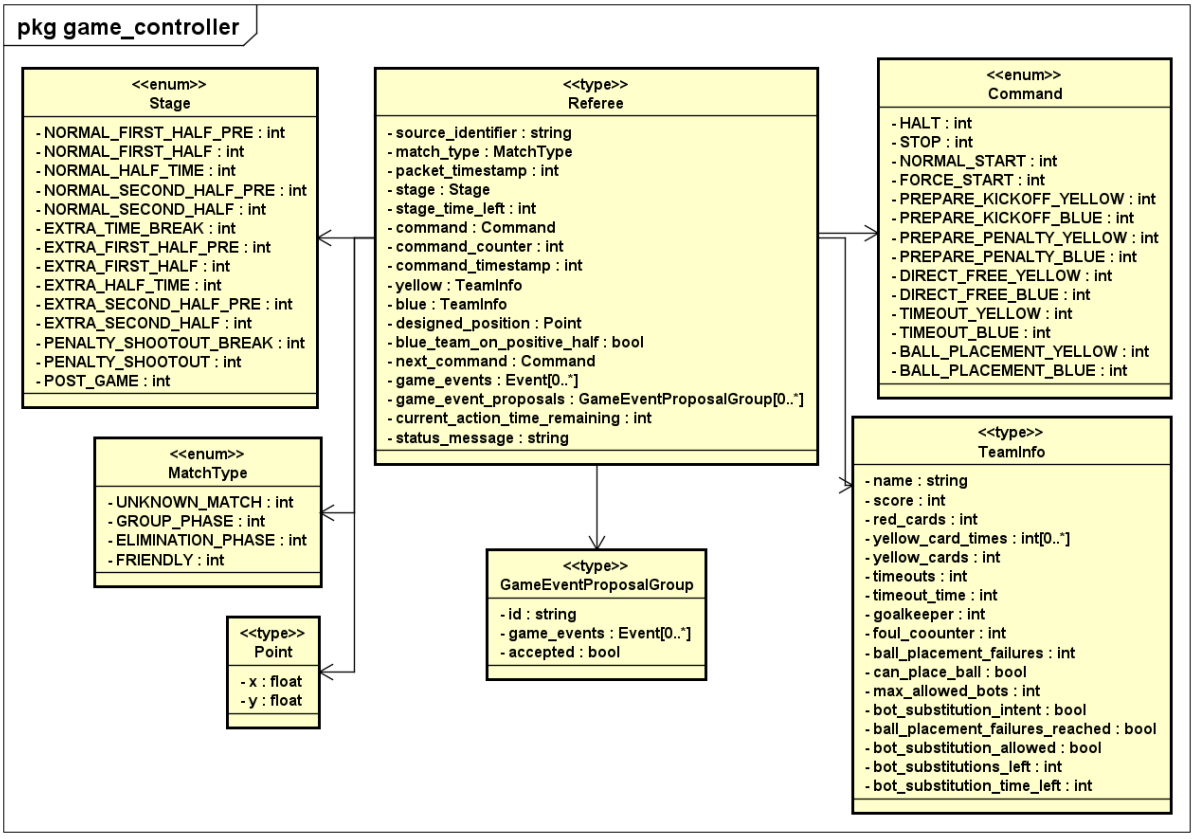

Figure 33: The class diagram for third-party referee data. Source: The author.

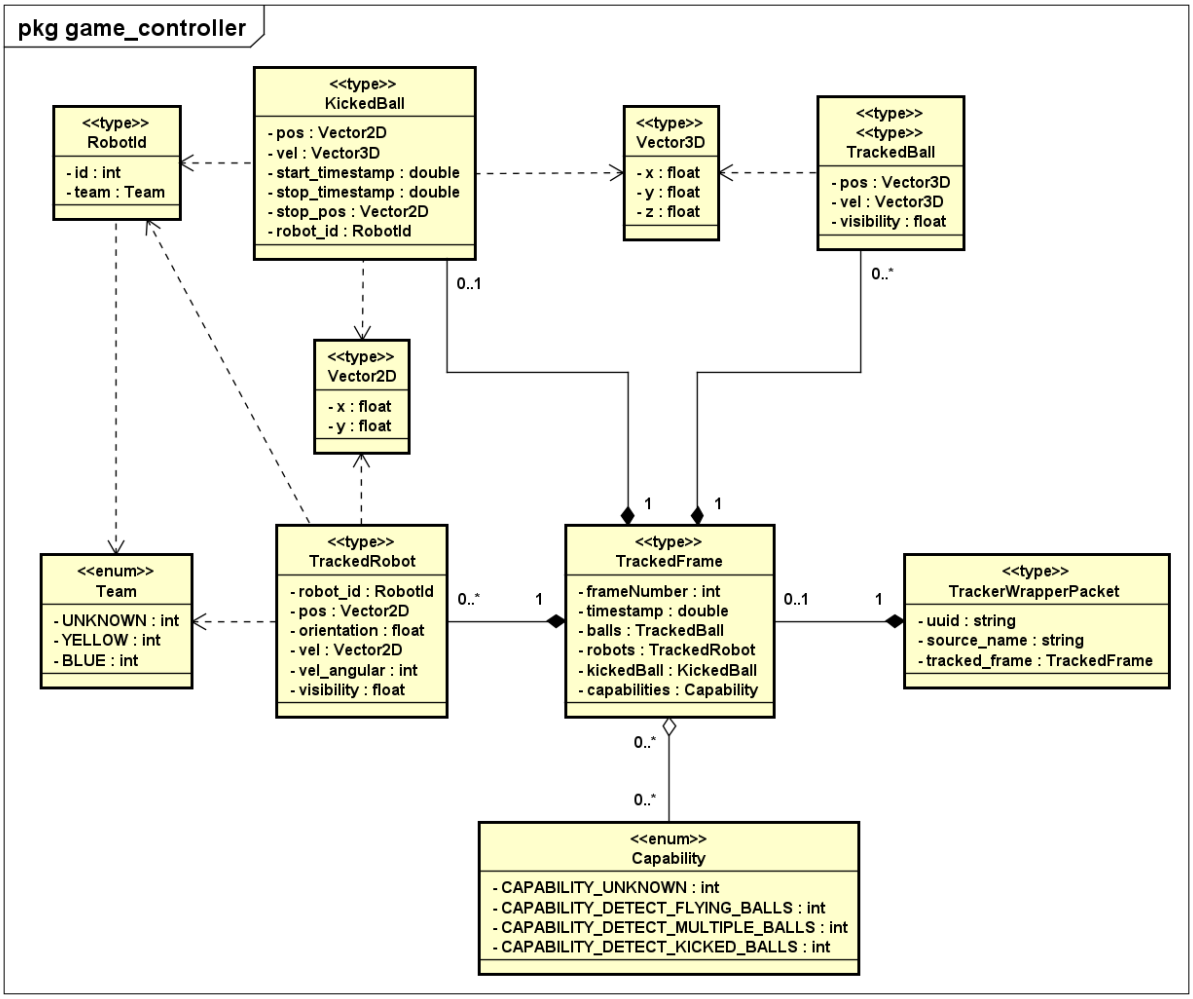

Figure 34: The class diagram for third-party tracked vision data. Source: The author.