Results in Engineering 32 (2026) 112048

Contents lists available at ScienceDirect

Results in Engineering

journal homepage: www.sciencedirect.com/journal/results-in-engineering

Research paper

DLS2: A distributed microcomponent-based robotic software architecture

Douglas Wildgrube Bertol [a,b,*], Geoff Fink [c,b], Marco Marchitto [b], Ylenia Nisticò [b], Michele Pestarino [b], Claudio Semini [b]

[a] Universidade do Estado de Santa Catarina (UDESC), Joinville, 89219-710, SC, Brazil
[b] Dynamic Legged Systems (DLS) lab, Istituto Italiano di Tecnologia (IIT), Genova, 16163, GE, Italy
[c] Thompson Rivers University (TRU), Kamloops, V2C 0C8, BC, Canada

ARTICLE INFO

Keywords:
Robotics
Software architecture
Distributed systems
Real-time
Scalability

ABSTRACT

In modern robotic systems, scalable, flexible, and robust software architectures are critical for enabling efficient collaboration and computation across distributed hardware. This paper introduces DLS2, a truly distributed robot-centric software architecture designed for complex, highly dynamic robots. Unlike conventional middleware, the DLS2 architecture natively integrates dynamic distribution of processing, containerized microservices, seamless application migration, and real-time performance constraints without relying on centralized orchestration. DLS2 manages different abstractions of the robotic stack by defining a structured, omnipresent layer-based organization. Through a formal architectural case study and a qualitative comparison with state-of-the-art frameworks, the structural advantages of this design paradigm are highlighted, particularly in terms of fault tolerance, modularity, and resource isolation. This research lays the foundational framework for future advancements in distributed robotic systems, offering an alternative design reference for resilient multi-robot environments. The DLS2 code can be found at https://github.com/iit-DLSLab/dls2-barebone

1. Introduction

The software development for highly dynamic robots, such as quadrupeds and humanoids, is becoming increasingly complex. For example, it is generally infeasible to deploy all software components on a single computational platform and expect it to simultaneously support high-frequency control loops and computationally intensive state-estimation algorithms. The field is therefore increasingly forced to adopt more sophisticated solutions that incorporate soft and hard guarantees, distributed systems, modular development practices, and related design principles. However, distributing complexity across the system imposes stringent requirements on both the software architecture and the development process. Unless these requirements are integrated within a robust and coherent foundation, the reliability of the entire system is compromised. Ultimately, how these requirements are specified and enforced fundamentally determines whether the robotic system operates correctly or fails.

Nowadays, the vast majority of the robotics community naturally adopts ROS 2 as the foundational framework for software development. It has become the de facto standard due to several factors, most notably its extensive ecosystem of ready-to-use tools and packages. To illustrate this paper's foundational motivation, users who have attempted to guarantee strict real-time performance across a distributed ROS 2 network know that achieving such behavior is rarely a plug-and-play experience, as reported in [1–3]. Getting that deterministic behavior usually means diving deep into DDS (Data Distribution Service) configurations and requiring extensive system-level tuning. In particular, the official ROS 2 documentation notes that appropriate QoS (Quality of Service) profiles are required to meet deadlines in real-time computing systems [4]. Furthermore, relying too much on node-specific or partly centralized architectures can introduce data bottlenecks. In dynamic environments, fault recovery under such constraints can become a major engineering challenge.

That is exactly why DLS2 was proposed. It is a way to address the requirements of software architecture and development for handling a complex system in highly dynamic robots. At its core, DLS2 is a microcomponent-based architecture composed of containerized microservices, combined with "omnipresent functional layers", to dynamically meet computational requirements such as software robustness, real-time constraints, load balancing, etc. The DLS2 proposal provides a way to maintain high-level architectural flexibility without sacrificing the low-level, real-time performance that legged robots absolutely demand. An illustration of these core concepts, and how they interact to provide true distributed system capabilities, is shown in Fig. 1.

This manuscript does not present a large-scale empirical benchmarking study; rather, it serves as a foundational case study of the

* Corresponding author.
E-mail address: douglas.bertol@udesc.br (D. Wildgrube Bertol).

https://doi.org/10.1016/j.rineng.2026.112048
Received 25 March 2026; Received in revised form 24 June 2026; Accepted 17 July 2026
Available online 18 July 2026

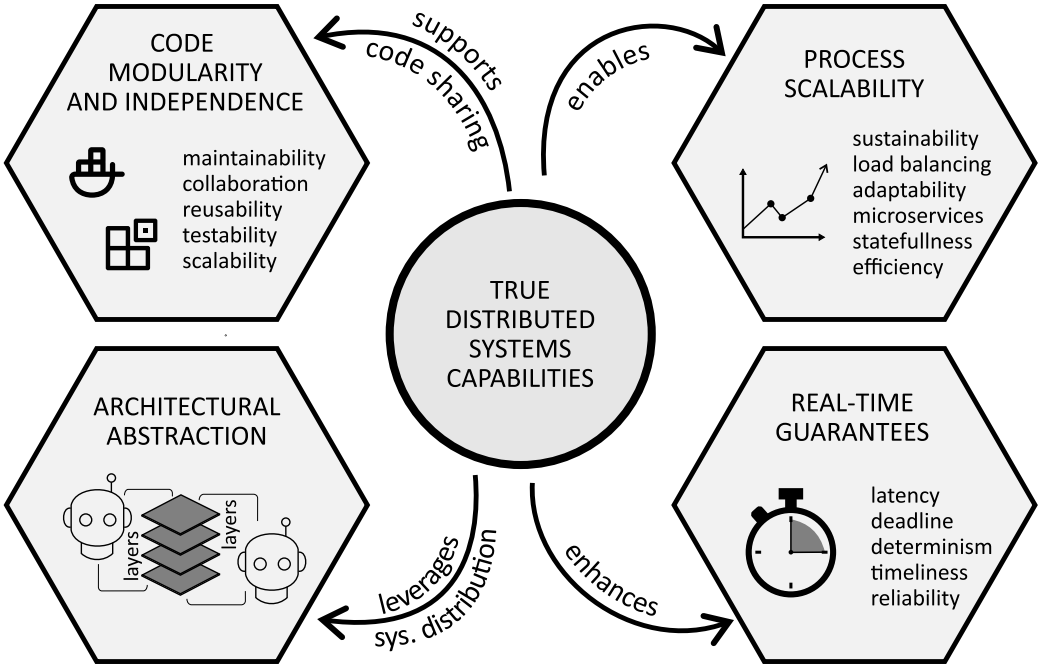

Fig. 1. Concepts in the software architecture.

architectural choices behind DLS2. Shifting to a new system solution with a fundamental rethink of software design requires, first, a formal documentation of these concepts. The goal with this paper is to establish DLS2 as a design reference, offering the community an alternative paradigm for robot software engineering. Comprehensive empirical results, such as extensive stress tests, real-time guarantees, congestion handling, and long-term hardware deployments, are currently the subject of ongoing research.

Given this context, the core contributions of this work are:

- Conceptualizing "omnipresent layers" that organize the software based on how the code behaves at runtime, rather than where it physically sits on the robot.
- Building a native split between real-time and best-effort tasks so that critical control loops never get starved for resources.
- Designing an architecture that completely drops centralized coordination, favoring horizontal scalability and much better fault tolerance.

The remainder of this paper is organized as follows: Section 2 provides a review of related work, framing DLS2 relative to other robotic framework solutions and situating it within the broader literature. Section 3 breaks down the actual architecture, its core concepts, and maps it to a real application. Section 4 shows the current development stage of the platform. Section 5 grounds everything with a practical example: running the MUSE state estimator [5]. Finally, Section 6 summarizes the proposal's conclusions and outlines the next steps for the architecture.

2. Related work

Contextualizing the architectural decisions behind DLS2 requires examining the historical evolution of robotic software and the persistent bottlenecks that continue to challenge the field. Rigid, hardware-bound implementations characterized early robotic control systems. This monolithic approach severely limited code reusability and scalability, ultimately forcing a necessary paradigm shift [6,7]. Consequently, the engineering community gravitated toward component-based designs and the adoption of standardized middleware [6,8–13].

Foundational frameworks such as Player/Stage [14], CARMEN [15], and YARP [16,17], along with contemporaneous platforms such as Orca [18] and Urbi [19], played instrumental roles in this transition. Furthermore, proprietary and hardware-centric environments, including Microsoft RDS [20], ARIA [21], and NaoQi [22], contributed significantly to standardizing hardware abstraction and human-robot interaction interfaces, albeit within closed ecosystems. Collectively, these systems introduced critical modularity and abstracted low-level device communication, effectively establishing the baseline for modern framework development.

To provide a structured taxonomic overview of this technological evolution, Table 1 categorizes these foundational and contemporary software platforms by their primary programming languages, underlying middleware protocols, specific research focuses, and current developmental statuses. This historical progression highlights a clear trend away from rigid, single-robot control interfaces and toward decentralized middleware networks capable of handling heterogeneous hardware topologies.

2.1. The ROS 2 ecosystem

Presently, the Robot Operating System (ROS) operates as the de facto standard for robotic software engineering [24]. Its widespread adoption is largely due to its resolution of legacy architectural bottlenecks, notably by eliminating the centralized master node in ROS 1 [23] in favor of a decentralized, per-node parameter model. The possibility of choosing different types of communication middleware, initially through integration with the Data Distribution Service (DDS) standard, was a particularly critical advancement, providing the underlying infrastructure for real-time (RT) communication, a domain requirement that the ROS community has actively prioritized since 2016.

Despite these capabilities, engineering a highly deterministic, multi-node robotic system exclusively within ROS 2 frequently demands substantial manual intervention and deep system-level tuning [1,3,25]. Although expert engineering teams can successfully configure DDS Quality of Service (QoS) profiles and kernel parameters to satisfy strict real-time constraints, this process introduces significant boilerplate effort. DLS2 does not seek to replace ROS 2. Rather, it is introduced as an alternative solution that can operate alongside ROS 2 while preserving interoperability through bridges, thereby enabling coexistence. This architectural divergence is rooted in a distinct design philosophy: DLS2 is

Table 1
Comparison of robotic software platforms.

Platform	Language	Middleware	Focus area	Status
Player/Stage [14]	C, C++, Python, Java	Custom (TCP/IP)	Multi-robot systems, 2D sims	D
MS RDS [20]	C#, Visual Basic	.NET	Education, simulation	D
Orca [18]	C++, Python, Java	Ice	Distributed systems	D
CARMEN [15]	C	Custom	SLAM, navigation	D
Urbi [19]	URBI, C++, Python	YARP	Event-driven programming	D
YARP [16]	C++, Python, Java	YARP	Communication, modularity	A
ARIA [21]	C++	Custom	Adept robot control	D
NaoQi [22]	Python, C++, JS	Custom	HRI, interaction	A*
ROS 1 [23]	C++, Python, Lisp	XML-RPC/ ROSComm	Modularity, distributed systems	A**
ROS 2 [24]	C++, Python, Java	DDS	Real-time, scalability, cross-platform	A

Legend: A - Active; D - Discontinued; *proprietary; **deprecated.

built to offer real-time determinism and strict computational resource isolation as native, omnipresent structural guarantees, avoiding the need for extensive user-driven configuration.

2.2. Model-driven engineering and predictability

As robotic platforms transition from controlled laboratory environments to highly dynamic, unpredictable real-world settings, purely code-centric development paradigms are proving increasingly inadequate. Achieving reliability requires a methodological shift towards Model-Driven Engineering (MDE) [26–30]. MDE methodologies emphasize the creation of abstract, structured models before deployment, ensuring that software components exhibit predictable behavior before execution. A central tenet of this approach involves the rigorous modeling of non-functional properties.

Research by Vicente-Chicote et al. [31], for example, has highlighted the efficacy of leveraging QoS metrics to estimate and constrain system behavior during the design phase. The DLS2 architecture aligns closely with these MDE principles by focusing not only on QoS configurations. By implementing abstract “omnipresent layers” and explicitly segregating real-time and best-effort execution pathways, the system inherently enforces predictable execution bounds. This structural approach ensures that timing constraints are met by design, rather than relying on empirical, post-compilation fixes.

2.3. Distributed architectures and the supervisor alternative

The realization of truly distributed architectures, in which the system operates without any central orchestrator, remains a very active research frontier, especially as computational demands extend to the edge and the cloud [32–36]. Contemporary literature features several promising efforts in this direction. The CORAL architecture [37], for instance, presents a robust structural framework for managing distributed robotic deployments. While such solutions offer sophisticated distributed management capabilities, DLS2 introduces a distinct methodology for governing global system behavior.

Rather than centralizing coordination logic or designating a singular supervisory node, DLS2 decentralizes behavioral orchestration entirely. The framework implements a distributed, automata-based supervisor [38], where complex state transitions, such as initiating a safe emergency shutdown or transferring control authority between modules, are governed by formal rules implemented as local automata embedded in individual processes, such as hardware drivers and estimators. This localized decision-making paradigm eliminates centralized points of failure. Consequently, the architecture possesses the innate capacity to dynamically migrate containerized microcomponents across the network of hardware in response to node congestion or localized hardware degradation, maintaining operational continuity. This distributed supervision is one of the main and most complex architectural paradigms proposed in this work, due to the interdisciplinarity required to make it a reality; that is, it requires knowledge of distributed systems, control of discrete-event systems, software and hardware engineering, etc.

2.4. Monitoring

Intelligent high-level decision-making relies on continuous monitoring of representative indicators that capture the health status of the robotic system. Within a coordinated orchestration framework, runtime monitoring is a key enabler of resilience, providing the information needed to detect anomalies and respond to unexpected events. Existing approaches have primarily focused on two complementary aspects: process monitoring, including node lifecycle management, real-time compliance, and hardware resource utilization [24]; and communication monitoring, addressing data synchronization, latency, and consistency of exchanged information [3,39]. Over the last decade, considerable effort has been devoted to improving the robustness of software execution on real robotic platforms. Proposed solutions range from standalone monitoring modules that can be integrated into ROS-based systems [40] to complete frameworks that incorporate supervision capabilities as a core architectural feature [41].

Within the ROS ecosystem, ROSMon [42] extends the standard infrastructure with process-level observability and resource monitoring capabilities. While effective in supervising node execution and resource usage, it lacks a system-wide health model and does not address data integrity or communication consistency. Conversely, the ROS Diagnostics Stack [43] provides a distributed health-reporting and aggregation infrastructure that enables subsystem-level monitoring. However, it does not include integrated lifecycle management, process supervision, or advanced timing verification mechanisms.

When considering frameworks that natively incorporate monitoring capabilities, Orocos RTT [44] is among the strongest foundations for execution supervision. It provides a rigorous real-time execution model with explicit component lifecycles and extensive runtime introspection, though it lacks a unified system health-management layer.

A more comprehensive approach is offered by the NASA Core Flight System (cFS) [45], which provides dedicated application-level monitoring and platform health management mechanisms. However, its focus on data quality and synchronization is limited, and its complexity makes integration into lightweight robotic architectures challenging.

Similarly, the automotive-oriented AUTOSAR Adaptive platform [46] provides explicit process-state supervision, along with mechanisms for deadline, alive, and logical monitoring, making it particularly suitable for safety-critical distributed systems. Nevertheless, like cFS, it is highly domain-specific and introduces significant integration complexity.

Overall, ROS-based solutions are lightweight and easy to integrate. Still, they typically address only specific aspects of system supervision and do not provide a unified framework capable of jointly managing lifecycle coordination, process and hardware health, resource utilization, and communication quality. In contrast, frameworks that natively

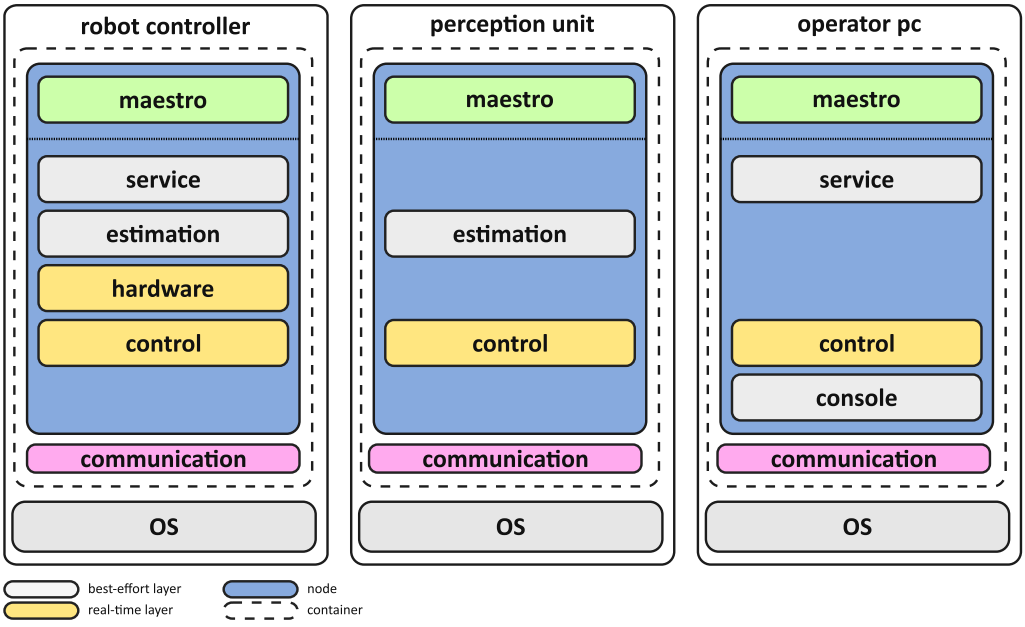

Fig. 2. System overview diagram.

incorporate monitoring capabilities offer a more comprehensive foundation for system supervision, but are often complex, highly domain-specific, and difficult to integrate into existing robotic architectures. As a result, no existing solution simultaneously combines ease of deployment, process-level monitoring, data-quality supervision, and scalable health management.

In this context, DLS2 is proposed as a general-purpose software framework that provides decentralized process-level health monitoring, covering resource utilization, data quality, and data synchronization, as well as machine-level monitoring of overall computational load.

3. The architecture

This section presents DLS2, a truly distributed, robot-centric software architecture that organizes computation across multiple hardware nodes without a central coordinator. The design goals are modularity, scalability, predictable real-time behavior, and operational resilience in dynamic environments.

3.1. Definitions and design premises

In DLS2, a "node" is the virtual abstraction of a physical processing unit (e.g., an embedded controller, an onboard computer, or an operator workstation). Nodes are bound to their respective hardware and do not migrate across devices. An "application" is a deployable software unit packaged for execution (e.g., a controller, estimator, or service). Applications may move between nodes at runtime, subject to resource and policy constraints. A "layer" is a system-wide functional stratum (e.g., control, estimation) that groups applications with similar roles and runtime properties (real-time or best-effort). Layers are omnipresent; they exist conceptually across the entire system and become active on any node that has the required capabilities to host them.

3.2. Distributed node model

At the highest level, DLS2 is composed of a set of heterogeneous nodes (blue box in Fig. 2) interconnected by a real-time-capable communication substrate. Each node is equipped with a set of tools to manage both local configurations and behaviors, as well as to interface with the framework. This set, represented by the maestro in Fig. 2, provides core functionalities for achieving distributed behavior, including discovery, information sharing, local monitoring, and recovery from anomalies.

DLS2 differs from decentralized arrangements by distributing both decision-making and computation across all nodes, without a privileged coordinator. This distributed nature enables horizontal scalability (by attaching additional hardware to form new nodes) and improves fault tolerance (by replicating critical applications and state). Task placement is dynamic: a node may offload or accept applications to balance load, reduce latency to sensors/actuators, or recover from failures. While applications can be moved, nodes themselves remain attached to their hardware, preserving an explicit mapping between physical resources and their virtual representations.

This distributed node architecture offers several key benefits. It enhances scalability by allowing the system to handle increased workloads through horizontal scaling. However, unlike other systems that may add virtual nodes or scale in software, DLS2 requires adding new hardware to increase the number of nodes. Fault tolerance is improved because replicating data and processes across multiple nodes ensures that the failure of a single node does not disrupt the entire system, maintaining continuous operation. Resource sharing is optimized, enabling efficient use of hardware, software, and data across the network, resulting in greater cost-effectiveness and improved performance.

The native distributed concept of the system is a key contribution of this architecture. Compared to other solutions already developed, DLS2 effectively implements this distributed node approach, ensuring that all nodes collaborate and perform tasks in a fully distributed manner, thereby enhancing scalability, fault tolerance, and efficiency.

3.3. Layered approach

The layered approach in DLS2 defines a logical, system-wide abstraction for organizing functionality across a distributed robotic architecture. Unlike conventional frameworks, where components are often tied to specific nodes or centralized coordinators, DLS2 layers are conceptual domains that span the entire system. They are not physical partitions but functional groupings that can be instantiated on any node with sufficient resources. In Fig. 3, the layers as perceived by the virtual system are illustrated as a single entity, distinct from the reality in which they are composed of an undefined number of processes distributed across different nodes.

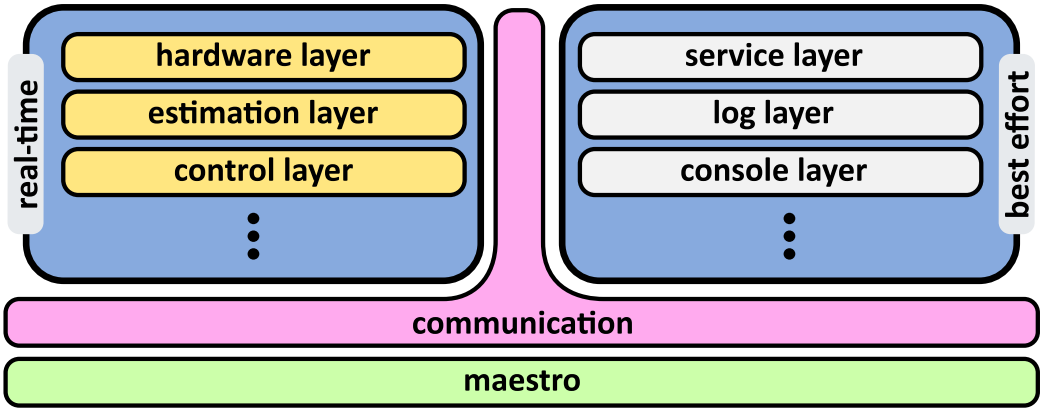

Fig. 3. Layers in the architecture.

This design introduces several advantages:

- Separation of Concerns: Each layer encapsulates a distinct functional domain-such as control, estimation, or logging-simplifying complexity and improving maintainability.
- Omnipresence and Uniformity: Layers exist globally. Any node can host applications from any layer, ensuring consistent behavior and shared knowledge across the system.
- Scalability and Adaptability: Because layers are abstract, new layers can be introduced or existing ones merged without disrupting the architecture, supporting evolution as requirements change.
- Real-Time Awareness: Layers are classified by execution requirements (real-time or best-effort), enabling deterministic scheduling and resource isolation for critical tasks.

This approach differs fundamentally from traditional robotic frameworks such as ROS or YARP, where functionality is often distributed but not omnipresent. In those systems, nodes typically host predefined roles, and adding new capabilities usually requires explicit configuration or centralized coordination. In contrast, DLS2 layers are system-wide and dynamic, allowing any capable node to assume responsibilities without manual reassignment or dependency on a master node.

For example, consider the control layer, which manages all controllers across the system. Instead of being confined to a single robot or computer, this layer is available on every node with adequate computational resources. If a controller fails on one node, another node can seamlessly instantiate a redundant controller, maintaining operational stability without human intervention. This redundancy and flexibility are critical for fault tolerance in multi-robot environments. This idea of omnipresence could be seen in Fig. 4, where the control layer is composed of 3 parts (inside all 3 nodes). Additionally, in Fig. 4, redundancy can be observed in the controllers `ctrl1` and `ctrl2`, which are present in more than one node, ensuring fault robustness.

By adopting this layered abstraction, DLS2 achieves a uniform, distributed architecture that supports real-time responsiveness, modularity, and resilience, essential characteristics for next-generation robotic systems operating in dynamic and unpredictable conditions.

3.4. Supervisor: behavior and coordination

The Supervisor in DLS2 is a distributed mechanism for managing high-level system behaviors through automata-based modeling. Unlike traditional frameworks that rely on centralized or single supervisory processes, the Supervisor in DLS2 is neither a layer nor a unique process. Instead, it is implemented as a distributed automaton, composed of local behavioral definitions embedded within each process (e.g., hardware drivers, controllers, estimation modules). These local definitions collectively form the global system behavior through coordination, following principles of distributed automata and formal language theory.

This design enables the Supervisor to enforce structured execution across the architecture without introducing a single point of failure. Each node contributes its own state-transition rules, which are dynamically combined to govern system-wide operations, such as initialization, safe controller switching, and emergency handling. By leveraging distributed automata concepts, the Supervisor ensures predictable transitions between operational states while maintaining scalability and fault tolerance.

The innovation lies in its ability to compose global behaviors from local specifications, in contrast to conventional robotic frameworks (e.g., ROS-based systems) that typically implement supervisory logic as centralized or node-specific scripts. This approach integrates tacit knowledge, allowing operational sequences-such as when to switch controllers or halt operations-to be encoded as formal rules distributed throughout the system.

From a theoretical perspective, this mechanism draws on the foundations of distributed automata and grammar systems, which model computation as a set of cooperating components that generate or accept a language under defined protocols (e.g., t-mode, *-mode, k-step modes). These models provide the formal basis for expressing distributed behaviors and coordination in complex systems [38].

By adopting this paradigm, DLS2 achieves a robust supervisory layer without centralization, ensuring that the system remains reliable, responsive, and adaptable under dynamic conditions. This capability is critical for distributed robotic architectures operating in real-time, safety-critical environments.

3.5. Microservices and containerization

DLS2 uses a microservice-based architecture, combined with containerization, to standardize application deployment, isolation, and versioning across heterogeneous nodes. Each application is encapsulated in a container, ensuring reproducibility and control for supervisors while eliminating inconsistencies caused by platform variations. This approach introduces several critical capabilities:

- It guarantees reproducibility and maintainability by encapsulating dependencies and runtime environments, simplifying version control, and ensuring deterministic behavior during deployment.
- It enables dynamic application migration between nodes to optimize resource utilization, reduce latency by moving closer to sensors or actuators, and recover from failures without manual intervention. Migration policies are coordinated by the Supervisor, allowing adaptive system behavior.
- It supports heterogeneity by isolating dependencies within containers, facilitating interoperability across diverse hardware and operating systems.

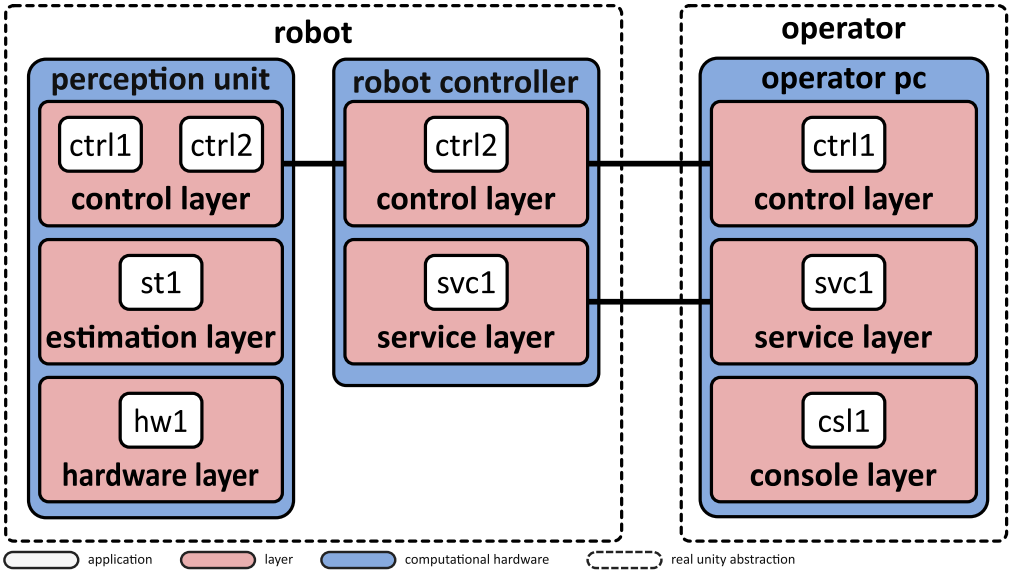

Fig. 4. Omnipresence of layers.

A lightweight runtime on each node manages the lifecycle of containerized applications, including start, stop, update, health monitoring, and placement decisions based on system policies. Unlike traditional monolithic deployments, which bind applications to specific hardware and require extensive manual configuration for updates or scaling, containerization provides portability, fault isolation, and rapid redeployment. These capabilities are essential for distributed robotic systems operating under real-time constraints and dynamic workloads, ensuring modularity, scalability, and resilience in control for supervisors.

3.6. Communication model

DLS2 implements a communication layer designed for scalability, determinism, and interoperability in distributed robotic systems. Inter- and intra-node communication combines publish-subscribe for time-series data streams and request-response for transactional interactions. A DDS-class messaging substrate underpins this layer, providing automatic discovery, low-latency delivery, and configurable Quality of Service (QoS) parameters, including reliability, history depth, and resource limits. Asynchronous messaging decouples producers and consumers, reducing blocking and improving responsiveness, while fault-tolerant data synchronization ensures consistency of replicated state across nodes.

Unlike traditional frameworks, the communication layer in DLS2 is agnostic to hosted applications and exposes uniform interfaces to all architectural layers. A distinctive feature is the use of multiple DDS domains based on the message's nature rather than the robot's identity. For example, control signals, sensor data, and reference trajectories are grouped in one domain, while service requests and logging messages reside in separate domains. This segmentation prevents congestion in a single domain and improves determinism under high-load conditions. In contrast to ROS2, where domains typically define a robot or deployment setup, DLS2 domains define message semantics and can be shared across multiple robots within the same infrastructure, enabling collaborative, resource-efficient communication.

This architecture supports dynamic node discovery, allowing new nodes to join or leave without major reconfiguration, and maintains real-time responsiveness even under variable network conditions. By combining QoS-driven DDS communication with domain-based segregation, DLS2 achieves predictable performance, scalability, and robustness-capabilities essential for distributed robotic systems operating in dynamic environments.

3.7. Real-time behavior

Real-time characteristics in DLS2 are treated as first-class design concerns and explicitly integrated into the architecture. Layers are categorized according to execution requirements: real-time (RT) for time-critical tasks and best-effort (BE) for non-critical operations. Control and portions of the Hardware and Estimation layers operate under RT constraints, while the Service, Console, and Log layers typically follow BE semantics. This separation of concerns ensures predictable scheduling and resource isolation, enabling efficient resource management.

RT applications declare priorities, CPU affinity, and timing parameters based on node capabilities, leveraging RTOS kernels or, where available, Linux with PREEMPT_RT extensions. The runtime enforces these policies by reserving cores and time budgets for RT workloads, preventing interference from BE tasks. Deterministic I/O paths are maintained by minimizing data copies and context switches, and by preallocating memory for control loops to eliminate allocation jitter. End-to-end deadlines are explicitly modeled: applications specify periods and deadlines, and the runtime validates feasibility before placement while continuously monitoring latency budgets during execution.

Communication QoS profiles are tuned for RT requirements. DDS topics carrying control signals or sensor streams utilize configurations optimized for bounded latency, such as best-effort delivery for freshest-only data or reliable transmission with limited history, as necessary. BE channels prioritize throughput over determinism. Fault handling under RT constraints is coordinated by the Supervisor, which enforces time-bounded failover policies. In the event of controller failure or missed deadlines, predefined transitions activate redundant controllers or initiate safe-stop behaviors within guaranteed reaction times.

To complement these mechanisms, a real-time evaluation tool is being developed. This tool validates configuration conformance against RT policies and reports measurable metrics-including jitter, latency, and deadline miss ratios-without requiring the user to have deep real-time expertise. Together, these features provide a robust foundation for predictable, resilient operation in distributed robotic systems.

3.8. Scalability and modularity

Scalability and modularity are foundational principles of DLS2, enabling the architecture to grow and adapt without compromising predictability or resilience. The system scales horizontally by attaching new hardware units, each of which forms a node that joins the network

through dynamic discovery and advertises its capabilities. This mechanism supports seamless integration and reintegration of nodes, ensuring that the system can expand or recover gracefully under changing operational conditions.

To maintain robustness, DLS2 employs replication of critical applications and state, eliminating single points of failure and enhancing fault tolerance. A distributed task allocation strategy optimizes application placement based on compute capacity, memory availability, network proximity, and timing constraints, while preserving real-time isolation for critical paths. Resource sharing across nodes further enhances efficiency, enabling dynamic workload balancing without compromising determinism.

Modularity complements scalability by structuring the architecture into loosely coupled components that can be reused, replaced, or extended independently. This design minimizes integration overhead and accelerates adaptation to new hardware platforms, algorithms, or application domains. In contrast to monolithic or centralized frameworks, where scaling often requires extensive reconfiguration and introduces bottlenecks, DLS2's modular and distributed approach ensures that complexity does not grow disproportionately with system size. However, achieving these benefits depends on disciplined engineering practices, including configuration management, monitoring, and well-defined operational policies; without these, distribution can increase complexity rather than robustness.

By combining horizontal scalability with modularity, DLS2 establishes a flexible and fault-tolerant foundation for next-generation robotic systems, capable of supporting heterogeneous platforms and dynamic workloads in real-world environments.

3.9. Interoperability and integration

To facilitate adoption and coexistence with established ecosystems, DLS2 incorporates interface bridges that expose standardized communication protocols and mediate interactions with external frameworks and robotic platforms. These bridges enable gradual integration and support mixed deployments, allowing legacy components or third-party software to operate alongside DLS2 without compromising its architectural principles of layering, isolation, and QoS enforcement. By abstracting protocol differences and encapsulating translation logic, the bridges preserve modularity and maintain deterministic behavior for real-time paths while accommodating heterogeneous environments.

This approach addresses a critical challenge in distributed robotics: leveraging existing investments in software and hardware while transitioning to more scalable and resilient architectures. Unlike monolithic systems, which often require complete rewrites or rigid coupling, DLS2's interoperability layer promotes incremental migration, reuse of proven components, and cross-framework collaboration. These capabilities are essential for research and industrial contexts where diverse platforms coexist, and operational continuity is paramount.

3.10. Summary of the architecture proposal

The architectural proposal detailed throughout Section 3 introduces a truly decentralized, robot-centric environment optimized for highly dynamic platforms. By dropping centralized coordination in favor of a true distributed model, the framework natively achieves horizontal scalability, application migration, and strong fault tolerance. These capabilities are reinforced by organizing runtime tasks into system-wide omnipresent functional layers that strictly isolate hard real-time control loops from best-effort workflows.

The design paradigms that distinguish this framework from state-of-the-art alternatives are systematically compiled in Table 2. As outlined in the referenced table, features such as automata-based distributed supervision, containerized microservice isolation, and semantic QoS messaging abstractions represent native structural guarantees within the system. Together, these unified mechanisms provide a robust, modular blueprint that ensures predictable execution bounds across heterogeneous hardware topologies.

4. Current stage of development

The implementation of DLS2 is progressing incrementally, translating its architectural principles as distribution, real-time awareness, modularity, and platform independence, into operational software components. Development milestones are organized per architectural layer and subsystem, enabling iterative validation and controlled evolution. Each architectural element has been addressed through targeted development efforts, resulting in a system that is both operational and extensible. This modular approach ensures flexibility, scalability, and maintainability.

The current implementation includes the following capabilities:

- **Distributed Node Architecture:** A node management subsystem abstracts each hardware unit as a node and supports dynamic application placement, coordinated lifecycle management, and comprehensive health monitoring at both the device and process levels, including execution status, real-time compliance, resource utilization, and data integrity and synchronization checks.
- **Omnipresent Layers:** Functional domains (Hardware, Control, Estimation, Service, Console, Log) are instantiated on any capable node, with explicit classification into real-time (RT) or best-effort (BE) execution contexts.
- **Controller Layer:** Orchestrates multiple controllers, and aggregates control signals under deterministic scheduling constraints.
- **Distributed Supervisor:** Local automata embedded within processes (e.g., drivers, estimators, controllers) compose global behavior, enabling safe initialization and controlled switching.
- **Containerization Runtime:** Packages applications with dependencies to ensure reproducible deployment across heterogeneous nodes and enable migration under policy constraints.
- **QoS-Aware Communication:** A DDS-class messaging substrate provides publish–subscribe and request–response semantics with configurable QoS, semantic domain segregation, and automatic discovery.
- **Real-Time Conformance Tooling:** Preliminary tooling validates configuration against RT policies (e.g., latency budgets, jitter bounds, deadline miss ratios) and reports metrics to guide placement and scheduling.
- **Console and Logging:** Operator-facing tools provide diagnostics, structured logs, and runtime introspection of processes and topics.
- **Dynamic Discovery and Interoperability:** Nodes can join or leave with minimal reconfiguration; interface bridges expose standardized protocols to integrate selected third-party components while preserving isolation and QoS guarantees.
- **Plugin Architecture:** External libraries can be attached as plugins, promoting reuse and framework independence.

Table 3 summarizes how architectural concepts have been instantiated in practice, and Table 4 provides descriptions of each implemented component.

These components form a functional baseline of the architecture and have been validated in internal testbeds for discovery, orchestration, and RT/BE isolation under representative loads.

The current implementation of the DDS standard is based on [47]. From an implementation standpoint, the vast majority of the framework's features and tools are developed in C++. However, Python-based processes are also integrated through a dedicated binding that exploits the proposed plugin architecture. This solution preserves real-time guarantees and ensures consistent node lifecycle management across programming languages, while enabling rapid prototyping and development by leveraging the extensive Python library ecosystem.

Ongoing work focuses on: (i) refining placement policies with end-to-end deadline modeling; (ii) expanding Supervisor rule sets for broader operational scenarios; (iii) extending interoperability bridges; and (iv)

Table 2
Comparison of architectural capabilities across robotic software frameworks.

Architectural Capability	Legacy (e.g., YARP)	ROS 1	ROS 2	CORAL	DLS2 (Proposed)
Fully Distributed & Coordinated Topology	×	×	✓	×	✓
Omnipresent Functional Layering	×	×	×	×	✓
Automata-Based Distributed Supervision	×	×	×	×	✓
Containerized Microservice Isolation	×	×	×	✓	✓
Semantic QoS Messaging Abstraction	×	×	×	×	✓
Native Real-Time & Best-Effort Segregation	×	×	×	×	✓
Dynamic Placement & Resource Scaling	×	×	×	~	✓
Multi-Framework Coexistence & Interoperability	✓	×	✓	~	✓

Key: ✓ = Natively Supported; × = Not Supported / Requires Custom Implementation; ~ = Partially Supported

Table 3
Mapping of DLS2 architectural concepts to implemented components.

Architectural Concept	Implemented Component
Distributed Node Architecture	Node Management System
Layered Architecture	Hardware, Control, Estimation, Service, Console, Log
Control Layer	Controller Manager
Supervisor	Automata-based Supervisor
Microservice Architecture	Containerization System
Communication Framework	DDS Messaging System
Real-Time System	Real-Time Evaluation Tool
System Monitoring	Console and Logging Tools
Scalability	Dynamic Node Discovery
Interoperability	Interface Bridges
Code Modularity & Independence	Plugin Architecture

hardening monitoring and recovery paths. As DLS2 evolves, components will continue to be validated against increasingly complex distributed configurations while preserving architectural invariants of omnipresent layers, fault isolation, and real-time determinism.

The DLS2 code can be found at https://github.com/iit-DLSLab/dls2-barebone.

5. Use case

The DLS2 framework has been employed to manage real-time communication between the robot's sensors, the MUSE state estimator described in [5], and the MPC controller of [48]. In this section, MUSE is used as a representative use case to demonstrate the effectiveness of the DLS2 framework in supporting complex, real-time robotic software systems. MUSE is a state estimator for quadruped robots that fuses multiple sensor modalities to achieve robust and accurate state estimation. The layered architecture of DLS2 ensures that components with distinct functional and timing requirements, such as control modules, estimation processes, and user interfaces, are logically grouped and independently managed, allowing flexible deployment across computational nodes.

The modular, layered design of DLS2 supports distributed deployment, enabling components to run across multiple nodes with minimal configuration overhead. This architecture facilitates resource optimization and system robustness, as computational processes can be dynamically relocated to balance system load or provide redundancy in the event of node failure. The schematic of the DLS2 software architecture shown in Fig. 3 highlights the framework's principal layers. The **Control Layer** handles modules requiring real-time scheduling and low-level hardware access. It governs resource-intensive processes such as robot controllers, ensuring minimal latency and fail-safe execution. The **Estimation Layer** manages signal acquisition, state estimation, and sensor data processing. It provides robust and accurate data integration from multiple sources, which is essential for making informed and reliable decisions. The MUSE state estimator is integrated within this layer. The **Service Layer** provides middleware services, including communication management, resource allocation, and distributed coordination among components. The **Logging and Console Layers** manage non-critical operations such as diagnostics, user interfaces, and data logging, ensuring usability without affecting performance-critical processes.

5.1. Integration of MUSE into DLS2

MUSE [5] has been seamlessly integrated into the DLS2 framework as a collection of modular plugins. Each module of MUSE follows DLS2's standardized plugin structure, ensuring consistent interfaces and predictable real-time behavior. This integration leverages DLS2's containerized and distributed runtime to enable efficient, low-latency communication between MUSE modules and other components in the control and estimation pipelines. It is important to clarify the scope of this use case. MUSE integration does not aim to provide a complete system-level benchmark of the architecture, but rather to validate the functional integration of a complex state-estimation pipeline within the DLS2 programming model.

Each module in MUSE operates as an independent plugin that retrieves input data, executes its core computational logic, and publishes outputs to the rest of the system. Specifically, each plugin contains three fundamental functions:

1. `read_inputs()` which gathers required data such as sensor readings or intermediate data from other modules.
2. `run()` which executes the module's core estimation algorithm.
3. `publish_outputs()`: which makes the computed results available to other modules or layers.

A complete evaluation of DLS2's distributed capabilities requires additional system-level stress tests. In particular, future experiments will consider node failures, artificial congestion of selected compute nodes, runtime migration of estimation and control components, activation of redundant modules, and quantitative measurements of latency, jitter, deadline-miss ratio, and failover time. These experiments will enable assessment of whether DLS2 can preserve the execution bounds of critical routines while redistributing software resources across heterogeneous compute nodes. Such evaluations are part of the ongoing development roadmap and will be reported in future work.

5.2. Modules in MUSE

The MUSE estimator is organized into two main groups of estimation modules. The **low-level estimation** modules process raw sensor information and perform preliminary computations that provide the fundamental physical quantities required by higher-level estimators. In contrast, **the high-level estimation** modules integrate and refine these outputs, combining multiple sensing modalities to generate consistent and drift-corrected state estimates. Together, these two groups establish a hierarchical estimation pipeline that balances real-time responsiveness with global accuracy.

The low-level modules include:

- **Contact Detection** which uses joint-state readings from encoders to determine which legs are in contact with the ground, essential for stability and locomotion control.

Table 4
Descriptions of implemented components.

Component	Description
Node Management System	Treats each hardware unit as a node; supports application migration and load balancing.
Omnipresent Layers	Functional domains instantiated on any capable node; classified as RT or BE.
Controller Manager	Orchestrates multiple controllers; enables redundancy and control-signal gathering.
Automata-based Supervisor	Composes global behavior from local state machines; ensures safe initialization and failover.
Containerization System	Packages apps with dependencies; ensures reproducible deployment and migration.
DDS Messaging System	Provides pub/sub and request-response with QoS; semantic domain segregation.
Real-Time Evaluation Tool	Verifies RT compliance (latency, jitter, deadlines).
Console and Logging Tools	Operator interface, and runtime diagnostics.
Dynamic Node Discovery	Nodes join/leave with minimal reconfiguration.
Interface Bridges	Protocol adapters for external frameworks; preserves isolation and QoS.
Plugin Architecture	Attaches external libraries as plugins; promotes reuse and independence.

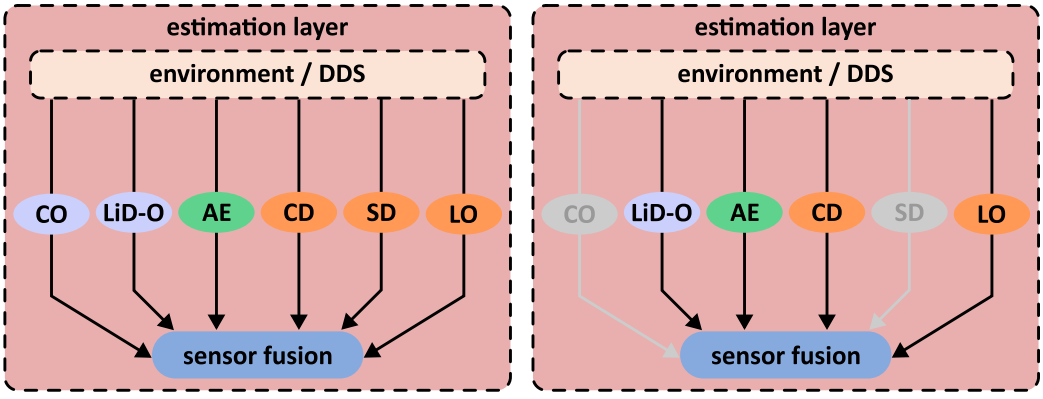

Fig. 5. Dynamic management of DLS2 modules. CO stands for Camera Odometry, LiD-O is LiDAR Odometry, AE for Attitude Estimation, CD Contact Detection, SD is Slip Detection, and finally, LO is Leg Odometry. The left panel shows a configuration with all modules active. In the right panel, the camera-odometry and slip-detection modules are deactivated. This capability enables the online reconfiguration of the system to accommodate changes in sensor availability or environmental conditions. Figure adapted from [49].

- **Attitude Estimation** which processes Inertial Measurements Unit (IMU) accelerometer and gyroscope data to compute robot orientation and angular velocity, providing fundamental attitude information for navigation.
- **Leg Odometry** which estimates robot motion from leg-joint kinematics, using inputs from the contact-detection and attitude-estimation modules, encoders, and IMU.
- **Slip Detection** which identifies slip events by comparing expected and measured leg motions, using data from the contact-detection and joint-state modules.

The high-level modules comprise:

- **Camera Odometry** which employs visual data to estimate the robot's position and orientation in space.
- **LiDAR Odometry**: which uses 3D spatial data from LiDAR sensors to refine positional accuracy.
- **Sensor Fusion**: This is the main module. It combines data from multiple sources, contact detection, attitude estimation, leg odometry, and optionally slip detection, camera, and LiDAR odometry, to produce a robust and accurate estimate of the robot's full state (pose and velocity).

Each plugin can be dynamically activated or deactivated during operation, enabling on-the-fly configuration based on the task or available sensors. For instance, visual odometry can be turned off in low-light environments and replaced with LiDAR-based estimation. An example of this is shown in Fig. 5. The modular structure simplifies experimentation with new algorithms or sensors without affecting the rest of the system.

Operating within DLS2's distributed, real-time scheduling environment, MUSE achieves an average execution time of approximately 0.05 ms per plugin `run()`. This frequency ensures real-time feedback to the robot's locomotion controller while maintaining strict timing guarantees. By isolating computational tasks within independent plugins and integrating them into the distributed DLS2 execution model, the combined DLS2-MUSE system provides a functional validation of the proposed modular and layered organization. While this integration supports future scalability and fault tolerance, a comprehensive quantitative assessment of these properties requires dedicated stress tests that simulate node congestion, component migration, and failure-recovery scenarios.

The reported execution time of approximately 0.05 ms per plugin `run()` characterizes the computational cost of the individual MUSE modules under the current implementation. It should not be interpreted as direct evidence that DLS2, by itself, improves the estimator's execution time. The contribution of DLS2 in this use case is instead related to the architectural organization of MUSE into independent plugins, their integration within the Estimation Layer, their compatibility with real-time scheduling policies, and their ability to be activated or deactivated according to sensing conditions and task requirements.

5.3. *Ongoing deployment and development status across robotic platforms*

While Section 5 focuses on a use case of the DLS2 framework, it is important to clarify the current development status of the overall architecture and its ongoing validation on real robotic platforms. DLS2 is still under active development, but several components are functional and already being utilized in real systems. At this stage, deployments serve as iterative validation steps that guide the architecture's evolution rather than as finalized demonstrations of performance.

DLS2 is currently being tested on a range of legged robotic platforms, including the hydraulic quadrupeds HyQ and HyQ-Real, and on the commercial quadrupeds Unitree Go1, Go2, and B2, as well as the G1 humanoid, and on HyQ-Real2 in preliminary experiments. These platforms are used to evaluate different aspects of the architecture-such as distributed real-time control, multi-node estimation, communication se-

mantics, and robustness of layer abstractions-under diverse hardware constraints and operational conditions. These tests provide essential feedback for refining the distributed node model, tuning QoS profiles, validating the Supervisor's behavior, and maturing the microcomponent deployment workflow.

The DLS2 code can be found at https://github.com/iit-DLSLab/dls2-barebone.

6. Conclusion and future works

The development of DLS2 represents a significant step toward true distribution in robotic software architectures. By eliminating central points of failure and enabling applications to migrate dynamically across nodes, DLS2 enhances system flexibility, scalability, and fault tolerance. The architecture effectively manages complex robotic operations while ensuring real-time responsiveness through a structured layering approach and microservice-based design. Our findings highlight the feasibility of implementing true distributed robotic systems, addressing key challenges in computational load balancing, fault recovery, and interoperability. Future work will optimize system performance, integrate advanced AI-driven decision-making, and further validate DLS2 in real-world robotic applications. The insights gained from this research contribute to the ongoing evolution of autonomous robotic frameworks, offering a robust foundation for next-generation distributed robotics.

Future work will focus on a complete system-level experimental evaluation of DLS2 under representative distributed operating conditions. Dedicated stress tests will be designed to quantify robustness, scalability, flexibility, and real-time responsiveness beyond the partial validations presented in this work. These tests will include artificial congestion of selected compute nodes, node failure and recovery scenarios, activation of redundant components, and runtime migration of software resources across heterogeneous hardware nodes. The evaluation will measure latency, jitter, deadline-miss ratio, failover time, and the architecture's ability to preserve the execution bounds of critical routines under degraded or dynamically changing computational conditions.

CRediT authorship contribution statement

Douglas Wildgrube Bertol: Writing – review & editing, Writing – original draft, Software, Project administration, Methodology, Investigation, Funding acquisition, Data curation, Conceptualization; **Geoff Fink:** Writing – original draft, Supervision, Software, Investigation, Conceptualization; **Marco Marchitto:** Writing – review & editing, Software, Project administration; **Ylenia Nisticò:** Writing – original draft, Software, Investigation; **Michele Pestarino:** Writing – review & editing, Software; **Claudio Semini:** Writing – review & editing, Supervision, Funding acquisition.

Disclosure statement

Generative AI tools assisted in portions of the writing, editing, or restructuring of this manuscript. These tools were used solely to support text refinement, organization, and phrasing, and all technical content, scientific claims, architectural descriptions, and experimental information were authored, verified, and validated by the human researchers. No AI system contributed to conceptual development, analysis, experimental design, results, or scientific conclusions.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This study was supported by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Brazil (PROAP/AUXPE DS) under Grant 88881.181788/225-01 and partially financed by the ESA funded project ANT (ESA AO/1-10289/20/NL/RA).

Data availability

No data was used for the research described in the article.

References

[1] Y. Ye, Z. Nie, X. Liu, F. Xie, Z. Li, P. Li, ROS2 real-time performance optimization and evaluation, Chin. J. Mech. Eng. 36 (1) (2023). https://doi.org/10.1186/s10033-023-00976-5
[2] M.L. Gambo, A. Danasabe, B. Almadani, F. Aliyu, A. Aliyu, E. Al-Nahari, A systematic literature review of DDS middleware in robotic systems, Robotics 14 (5) (2025). https://doi.org/10.3390/robotics14050063
[3] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, G. Fettweis, Latency analysis of ros2 multi-node systems, (2021). arXiv preprint arXiv:2101.02074
[4] ROS2, ROS2 documentation: Quality of Service settings, 2026, (https://docs.ros.org/en/rolling/Concepts/Intermediate/About-Quality-of-Service-Settings.html). Accessed: 2026-06-12.
[5] Y. Nisticò, J.C.V. Soares, L. Amatucci, G. Fink, C. Semini, MUSE: A real-time multi-Sensor state estimator for quadruped robots, IEEE Robot. Autom. Lett. 10 (5) (2025) 4620–4627. https://doi.org/10.1109/LRA.2025.3553047
[6] D. Brugali, A. Shakhimardanov, Component-based robotic engineering (Part II), IEEE Robot. Autom. Mag. 17 (1) (2010) 100–112. https://doi.org/10.1109/mra.2010.935798
[7] D. Brugali (Ed.), Software engineering for experimental robotics, Springer Berlin Heidelberg, 2007. https://doi.org/10.1007/978-3-540-68951-5
[8] A. Ahmad, M.A. Babar, Software architectures for robotic systems: a systematic mapping study, J. Syst. Softw. 122 (2016) 16–39. https://doi.org/10.1016/j.jss.2016.08.039
[9] D.E. Perry, A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Softw. Eng. Notes 17 (4) (1992) 40–52. https://doi.org/10.1145/141874.141884
[10] D.J. Miller, R.C. Lennox, An object-oriented environment for robot system architectures, in: IEEE International Conference on Robotics and Automation, IEEE Comput. Soc. Press, 1990, pp. 352–361. https://doi.org/10.1109/robot.1990.126001
[11] A. Elkady, T. Sobh, Robotics middleware: a comprehensive literature survey and attribute-Based bibliography, J. Robot. 2012 (2012) 1–15. https://doi.org/10.1155/2012/959013
[12] M.V.D. Veloso, J.T.C. Filho, G.A. Barreto, SOM4R: a middleware for robotic applications based on the resource-oriented architecture, J. Intell. Robot. Syst. 87 (3-4) (2017) 487–506. https://doi.org/10.1007/s10846-017-0504-y
[13] S. García, D. Strüber, D. Brugali, A. Di Fava, P. Pelliccione, T. Berger, Software variability in service robotics, Empir. Softw. Eng. 28 (2) (2022). https://doi.org/10.1007/s10664-022-10231-5
[14] R.T. Vaughan, B.P. Gerkey, Reusable robot software and the Player/Stage project, volume 30 of [7], 2007, pp. 267–289. https://doi.org/10.1007/978-3-540-68951-5_16
[15] M. Montemerlo, N. Roy, S. Thrun, Perspectives on standardization in mobile robot programming: the Carnegie Mellon navigation (CARMEN) toolkit, in: International Conference on Intelligent Robots and Systems (IROS), 3 of *IROS-03*, IEEE, 2003, pp. 2436–2441. https://doi.org/10.1109/iros.2003.1249235
[16] G. Metta, P. Fitzpatrick, L. Natale, YARP: Yet another robot platform, Int. J. Adv. Robot. Syst. 3 (1) (2006). https://doi.org/10.5772/5761
[17] G. Metta, L. Natale, F. Nori, G. Sandini, The icub project: an open source platform for research in embodied cognition, in: Advanced Robotics and Its Social Impacts, 2011, pp. 24–26. https://doi.org/10.1109/ARSO.2011.6301975
[18] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, A. Orebäck, Orca: a component model and repository, volume 30 of [7], 2007, pp. 231–251. https://doi.org/10.1007/978-3-540-68951-5_13
[19] J.C. Baillie, URBI: Towards a universal robotic low-level programming language, in: International Conference on Intelligent Robots and Systems (IROS), IEEE, 2005, pp. 820–825. https://doi.org/10.1109/iros.2005.1545467
[20] J. Jackson, Microsoft robotics studio: a technical introduction, IEEE Robot. Autom. Mag. 14 (4) (2007) 82–87. https://doi.org/10.1109/m-ra.2007.905745
[21] A. Whitbrook, Programming mobile robots with Aria and Player, Springer London, 2010. https://doi.org/10.1007/978-1-84882-864-3
[22] A. Bolotnikova, P. Gergondet, A. Tanguy, S. Courtois, A. Kheddar, Task-space control interface for softbank humanoid robots and its human-robot interaction applications, in: 2021 IEEE/SICE International Symposium on System Integration (SII), IEEE, 2021, pp. 560–565. https://doi.org/10.1109/ieeeconf49454.2021.9382685
[23] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A.Y. Ng, et al., ROS: An open-source robot operating system, in: ICRA Workshop on Open Source Software in Robotics, Kobe, Japan, 2009, p. 5.
[24] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, W. Woodall, Robot operating system 2: design, architecture, and uses in the wild, Sci. Robot. 7 (66) (2022) eabm6074. https://doi.org/10.1126/scirobotics.abm6074

[25] S. Lee, J. Kang, K.-J. Park, Dependency chain analysis of ROS 2 DDS QoS policies: from lifecycle tutorial to static verification, (2025). arXiv:2509.03381
[26] F. Ciccozzi, D. Di Ruscio, I. Malavolta, P. Pelliccione, J. Tumova, Engineering the software of robotic systems, in: IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, 2017, pp. 507–508. https://doi.org/10.1109/icse-c.2017.167
[27] I. Malavolta, G.A. Lewis, B. Schmerl, P. Lago, D. Garlan, Mining guidelines for architecting robotics software, J. Syst. Softw. 178 (2021) 110969. https://doi.org/10.1016/j.jss.2021.110969
[28] C. Nam, S. Lee, J. Lee, S.H. Cheong, D.H. Kim, C. Kim, I. Kim, S.-K. Park, A software architecture for service robots manipulating objects in human environments, IEEE Access 8 (2020) 117900–117920. https://doi.org/10.1109/access.2020.3003991
[29] L. Asprino, P. Ciancarini, A.G. Nuzzolese, V. Presutti, A. Russo, A reference architecture for social robots, J. Web Semant. 72 (2022) 100683. https://doi.org/10.1016/j.websem.2021.100683
[30] P. Daubaris, J. Helovuo, N. Mäkitalo, Augmenting robot software development process with flexbot, in: IEEE/ACM International Workshop on Robotics Software Engineering (RoSE), IEEE, 2023, pp. 69–72. https://doi.org/10.1109/rose59155.2023.00015
[31] C. Vicente-Chicote, D. García-Pérez, P. García-Ojeda, J.F. Inglés-Romero, A. Romero-Garcés, J. Martínez, Modeling and estimation of non-functional properties: leveraging the power of QoS metrics, in: J.M. Ferrández Vicente, J.R. Álvarez-Sánchez, F. de la Paz López, J. Toledo Moreo, H. Adeli (Eds.), From Bioinspired Systems and Biomedical Applications to Machine Learning, Springer International Publishing, Cham, 2019, pp. 380–388. https://doi.org/10.1007/978-3-030-19651-6_37
[32] M. Amoretti, M. Reggiani, Architectural paradigms for robotics applications, Adv. Eng. Inform. 24 (1) (2010) 4–13. https://doi.org/10.1016/j.aei.2009.08.004
[33] J. Zhang, F. Keramat, X. Yu, D.M. Hernández, J.P. Queralta, T. Westerlund, Distributed robotic systems in the edge-cloud continuum with ROS 2: a review on novel architectures and technology readiness, in: International Conference on Fog and Mobile Edge Computing (FMEC), IEEE, 2022, pp. 1–8. https://doi.org/10.1109/fmec57183.2022.10062523
[34] A.D. Hristozov, E.T. Matson, J.C. Gallagher, M. Rogers, E. Dietz, Resilient architecture framework for robotic systems, in: International Conference Automatics and Informatics (ICAI), IEEE, 2022, pp. 18–23. https://doi.org/10.1109/icai55857.2022.9960094
[35] T. Taira, N. Yamasaki, Functionally distributed control architecture for autonomous mobile robots, J. Robot. Mechatron. 16 (2) (2004) 217–224. https://doi.org/10.20965/jrm.2004.p0217
[36] A. Orebäck, H.I. Christensen, Evaluation of architectures for mobile robotics, Auton. Robots 14 (1) (2003) 33–49. https://doi.org/10.1023/a:1020975419546
[37] S. Swanbeck, M. Pryor, CORAL: A unifying abstraction layer for compositional robotics software, in: 2026 IEEE/SICE International Symposium on System Integration (SII), IEEE, 2026, pp. 956–963. https://doi.org/10.1109/SII64115.2026.11404692
[38] K. Krithivasan, R.R. Balan, Distributed processing in automata, Int. J. Found. Comput. Sci. 20 (4) (2009) 543–556. https://doi.org/10.1142/S012905410900678X
[39] Y. Maruyama, S. Kato, T. Azumi, Exploring the performance of ROS2, in: Proceedings of the 13th International Conference on Embedded Software, EMSOFT '16, Association for Computing Machinery, New York, NY, USA, 2016, pp. 1–10. https://doi.org/10.1145/2968478.2968502
[40] A. Santos, A. Cunha, N. Macedo, The high-assurance ROS framework, in: 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE), IEEE Press, 2021, p. 37-40. https://doi.org/10.1109/RoSE52553.2021.00013
[41] D. Bozhinoski, M. Garzon Oviedo, N. Hammoudeh Garcia, H. Deshpande, G. van der Hoorn, J. Tjerngren, A. Wasowski, C. Hernandez Corbato, MROS: Runtime adaptation for robot control architectures, Adv. Robot. 36 (11) (2022) 502–518. https://doi.org/10.1080/01691864.2022.2039761
[42] M. Schwarz, rosmon: A ROS Process Monitor, 2018, (ROSCon). https://github.com/xqms/rosmon.
[43] ROS Contributors, diagnostics: Tools for analyzing and diagnosing the state of robots running ROS, 2026, (https://github.com/ros/diagnostics). Accessed: 2026-06-17.
[44] H. Bruyninckx, Open robot control software: the OROCOS project, Proc. IEEE Int. Conf. Robot. Autom. (2001). https://www.orocos.org/rtt.
[45] D. McComas, J. Wilmot, A. Cudmore, The core flight system (cFS) community: providing low cost solutions for small spacecraft, in: Proceedings of the 30th Annual AIAA/USU Conference on Small Satellites, Technical Session IV: Advanced Technologies I, Utah State University, Logan, UT, 2016, pp. 1–8. https://digitalcommons.usu.edu/smallsat/2016/TS4AdvTech1/1/.
[46] S. Fürst, M. Bechter, AUTOSAR For connected and autonomous vehicles: the AUTOSAR adaptive platform, in: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W), 2016, pp. 215–217. https://doi.org/10.1109/DSN-W.2016.24
[47] eProsima, Fast-DDS, 2016, (https://github.com/eProsima/Fast-DDS). Accessed: 2026-06-12.
[48] L. Amatucci, G. Turrisi, A. Bratta, V. Barasuol, C. Semini, Accelerating model predictive control for legged robots through distributed optimization, in: 2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2024, pp. 12734–12741. https://doi.org/10.1109/IROS58592.2024.10801676
[49] Y. Nistico, Enhancing State Estimation in Quadruped Robots: Multi-Sensor Fusion for Challenging Terrains, Phd thesis, Università degli studi di Genova, Genova, IT, 2025.