Advancing Humanoid Robotics with Rust: An Open Framework for Runtime Efficiency

Team HULKs, Maximilian Schmidt[1][0009−0005−4532−7669], Hendrik Sieck[1], Ole Felber[1], Konrad Valentin Nölle[1], Luis Scheuch[1], and Patrick Göttsch[1][0000−0002−6642−8195]

Team HULKs, Hamburg University of Technology, Hamburg, Germany
hulks@tuhh.de
https://hulks.de

Abstract. The development of software frameworks plays a pivotal role in the research on humanoid robotics. This paper introduces a novel robotics framework designed to address challenges in the RoboCup and robotics in general. We present an overview of existing frameworks within the RoboCup Standard Platform League (SPL), highlighting their benefits and limitations. We define a set of requirements that serve as a guiding principle for the development of robotic frameworks. Our novel framework emphasizes modularity and parallelization while optimizing for minimal runtime overhead and complexity. Implemented in Rust, we combine computational efficiency with safety, positioning it as a robust solution for robotic applications. We show that the framework's runtime overhead on a NAO robot has little impact. We conclude with case studies of its application at Team HULKs and other teams, showing its real world use case and impact in the RoboCup SPL.

Keywords: Framework · Middleware · Rust · RoboCup.

1 Introduction

Efficient and adaptable software frameworks are crucial for advancing the capabilities of teams participating in RoboCup and robotics in general. In the context of RoboCup Standard Platform League (SPL), the challenge of humanoid robotics extends beyond creating intelligent algorithms for perception, decision-making, or motion planning; it encompasses the efficient implementation and coordination of these algorithms with heavily constrained computational resources.

This paper presents a novel robotics framework designed to overcome challenges and shortcomings of existing solutions. Our framework is engineered to optimize performance, enhance modularity, and establish a seamless interface between robotic hardware and detection as well as control algorithms.

1.1 Robotic Frameworks

Abstraction layers separate concerns into smaller parts, to cope with complexity in the software domain. A common abstraction is a software framework which separates generic code from application-specific code.

In the context of robotics, a multitude of frameworks have been introduced to facilitate the development and execution of robot control software [22]. Notable examples include ROS 1 [18], ROS 2 [15], MIRA [7], YARP [17], LCM [11], Player [8], and Urbi [2], each contributing distinct perspectives and methodologies to the field. The following sections provide an overview of existing frameworks within the RoboCup SPL, highlighting their unique characteristics and design philosophies.

Robot Operating System (ROS) Version 2 Building upon the foundation of Robot Operating System (ROS) 1, ROS 2 [15] addresses several limitations of its predecessor. ROS 2 uses the Data Distribution Service, a publish-subscribe and request-response system. This enhances the robustness and fault tolerance of the middleware compared to ROS 1. ROS 2 maintains an event-driven communication architecture via serialized messages, resulting in similar overheads and complexity as ROS 1. The ROS messaging pattern, while flexible, loses synchronicity, leading to increased complexity in processing nodes that rely on temporally coupled messages.

SPL Frameworks In the RoboCup SPL league, all teams compete using a standardized robotics hardware platform: the NAO robot by Softbank Robotics [20]. Teams prefer developing custom robot control systems instead of using established ones like ROS to enable real-time robotic algorithms on the NAO while keeping overheads in the framework as low as possible. This section will take a closer look at the different systems teams have developed for the SPL.

Bembelbots The team Bembelbots have developed a framework centered around a message passing architecture [6]. Within this framework, each module declares inputs and outputs, which are used to automatically generate a dependency graph. This graph serves as a blueprint for execution, enabling modules to run concurrently while following the specified order of dependencies. Similar to ROS, the message passing actors have the drawback of asynchronicity.

Berlin United Berlin United uses a sequential blackboard design [16]. Sequential execution per thread ensures that at any given time, at most one module reads from or writes to the thread's blackboard, eliminating the need for explicit synchronization. Each module specifies data it requires and provides, and the framework determines the execution order through topological sorting. Additionally, the framework supports serialization through the use of Protocol Buffers [9].

B-Human The B-Human framework is based on a 2007 framework of the GermanTeam, written in C++. Since then, the framework has been continuously improved and adapted to the changing requirements of the SPL. The robotics code adopts a modular structure where each module, can both provide and require artifacts from other modules. Communication between modules is handled via memory-sharing, with data being serialized and written in triple buffered

channels [19]. While serialization for message passing standardizes the interface between threads, it also leads to additional overheads. Furthermore, a processing task, responsible for merging sensor data from various threads, is activated with each new input, which requires the reprocessing of unchanged data.

NaoDevils The NaoDevils team employs a framework that originates from the 2009 B-Human framework. The framework defines three distinct domains: cognition, motion, and debug. Within each domain, modules operate in parallel, facilitated by a thread pool managed by Taskflow [12]. A blackboard data structure [10] accompanies each domain. A task graph manages the execution order of modules, ensuring that data is already provided if a module requests to read it. Despite that, the task graph handles thread synchronization, ensuring correct memory access. Between domains, a message queue handles information exchange, copying data to prevent race conditions [14].

rUNSWift Team rUNSWift uses a framework written in C++, with the robot behavior integrated using Python [1]. Communication among different modules is managed by a blackboard data structure. To prevent conflicting writes to the same blackboard address, only the top-level module of a thread is granted permission to write to a thread-specific sub-blackboard. Furthermore, team rUNSWift will consider the adoption of ROS 2 after RoboCup 2024.

Summary Despite the popularity of frameworks like ROS in the robotics community, the RoboCup SPL presents unique challenges due to the limited computational resources of NAO robots. As a result, several teams have developed custom frameworks tailored specifically to the demands of the SPL. While each framework has its own design philosophy and implementation details, they all share the common goal of optimizing performance and resource utilization for the NAO platform. These frameworks offer insights into different approaches for addressing the computational constraints of the SPL.

1.2 Requirements

Drawing inspiration from existing frameworks and the design decisions of the NaoDevils team's work [14], we establish a set of requirements that serve as a guiding principle for our framework.

Parallelization: Efficient usage of all available CPU cores is critical to the framework's performance. Performance is directly impacted by the number of cores utilized.

Motion Cycle Priority: The motion cycle has to be prioritized over any other perception tasks. Responsiveness is crucial in dynamic environments where real-time motor control is essential for successful robotic operation.

Low Runtime Overhead: On robotic systems with limited resources, minimizing execution time spent in the framework enables executing more advanced robotic algorithms. Balancing information exchange with computational efficiency is necessary to streamline context switching in a manner that avoids

unnecessary computational overheads that may impede overall system performance.

Separation between Robotics Domain and Framework: Separation of concerns in software enhances its flexibility. Establishing a clear boundary between robotic functionalities and the underlying framework decouples the application from its infrastructure. This separation not only ensures adaptability to various robotic platforms and new and evolving functionalities but also guards against overfitting, creating a more versatile and widely applicable framework.

Adaptive Computation: Adopting a computation model based on external triggers ensures that computational resources are allocated in response to environmental changes or incoming data, preventing unnecessary calculations in the absence of new information.

Temporally-Ordered Data: Mechanisms for preserving temporal ordering of incoming data enable robotic algorithms that require temporal order, like filters.

Abstraction over Hardware: Providing abstraction layers over the underlying hardware enables the framework to generalize over the actual hardware it is deployed on. This abstraction enables interoperability and allows using the framework with diverse hardware configurations.

Data Structure Serialization and Reflection: Support for serialization and reflection mechanisms enables efficient data exchange and introspection of the robotics domain. These features allow developers to gain insights into the framework's data structures of robotic algorithms.

Tooling Support: Tools for debugging, profiling, and monitoring enhance the development and maintenance process. Robust tooling support is essential for understanding runtime behavior, diagnosing issues or optimizing performance.

Programming Language: The choice of a programming language is pivotal. The language must be capable of handling the computational demands of robotic applications while providing a high degree of safety and security and supporting modern programming paradigms.

2 Framework

With the foundational requirements established to guide the development of our novel framework, we now introduce the framework itself. Our framework is designed to hide implementation detail from the user, providing a narrow interface for the application of robotic algorithms. To keep the implementation flexible, the architecture strictly separates framework and robotics code. This allows the framework to be used on different robotic platforms and to be adapted to various robotics domains beyond the RoboCup SPL.

This section provides an overview of the architecture, defining core components and concepts. We introduce the modularization of cyclers and nodes, as well as their interaction through hardware interfaces. We conclude with an overview of communication channels, code generation, and our selection of programming language.

2.1 Cyclers

Robots interact in a dynamic environment. To fulfill their tasks, they perceive their environment using sensors and act via actuators. We model this as an event-based system in which sensor measurements trigger computation to produce actuator outputs. Depending on the sensor, the rate of incoming measurement events on robotic platforms varies. For example, the NAO cameras deliver 30 images per second, while the joint position encoders deliver 83 measurements per second. We define a cycler as a unit of computation that encapsulates the data flow from sensor measurements to actuator commands. Cyclers wait for environmental events, process the events, and produce actuator commands. After producing the actuator commands, the cycler waits for the next environmental event. We call this loop a cycle.

The framework allows cyclers to be instantiated multiple times, e.g., for deploying the same detection pipeline to multiple cameras. These instances differ only in their set of configuration parameters. Different cycler instances are executed in individual operating system threads to efficiently use parallel processing resources.

2.2 Nodes

Cyclers modularize their execution by dividing the pipeline into several nodes. Nodes are small units of computation that abstract tasks, e.g., reading an image, detecting robots in the image, or executing a motion. Each node is defined by its inputs, outputs, self-contained state, and its implementation. Figure 1 gives a structural overview of a cycler and its nodes.

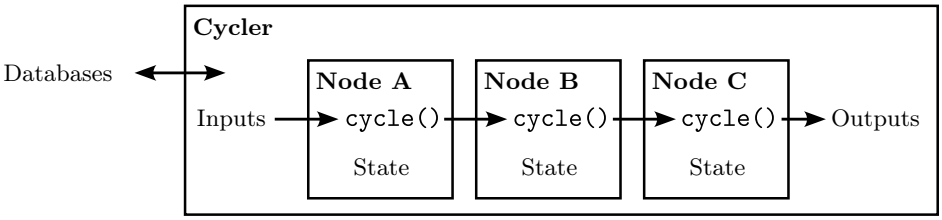

Fig. 1. Overview of a cycler and its execution pipeline of nodes.

Nodes produce strongly typed outputs, each uniquely identified by a distinct name. Inputs to nodes encompass outputs from other nodes, as well as configuration parameters. This allows nodes to form a comprehensive dependency graph. Based on this graph, the framework sorts the execution of nodes topologically, ensuring that each node is executed only after its dependencies have been produced. A cyclic dependency between nodes results in a compile-time error. During execution, each cycler traverses its nodes sequentially, ensuring systematic processing of data throughout the framework.

2.3 Starting Cycles

The start of each cycle is triggered by external events, such as new images from the camera. Special nodes, called setup nodes, interact with the environment through a hardware interface and wait for the next external event. After setup nodes are executed, the cycler starts its cycle by executing the remaining nodes.

2.4 Data Exchange within Cyclers

The framework ensures efficient data exchange within cyclers by storing node outputs in databases local to each cycler. This local storage enables access to other node's outputs through direct memory access. By executing each cycler instance in its own operating system thread, the framework guarantees contention-free access without the need for explicit synchronization. The conclusion of each cycle signifies the readiness of the database for consumption by other entities.

2.5 Data Exchange across Cyclers

The framework allows nodes to access outputs of other cycler instances. However, these accesses traverse thread boundaries, and therefore demand careful synchronization. The framework provides support for two distinct access semantics: latest non-blocking access using multiple buffering [13] and interleaved temporally-ordered access. The former allows nodes to retrieve the most recent data. The latter ensures a temporal ordering, providing an interleaved sequence based on the timestamps associated with each data entry.

Consider a localization pipeline where the robot's position must synchronize with motion decision-making, while updates need to be processed in order. In this scenario, the localization may depend on outputs of both a long-running cycle, executed every 2 s, and another, every 10 ms. Figure 2 illustrates the timing of four cyclers with interleaved execution.

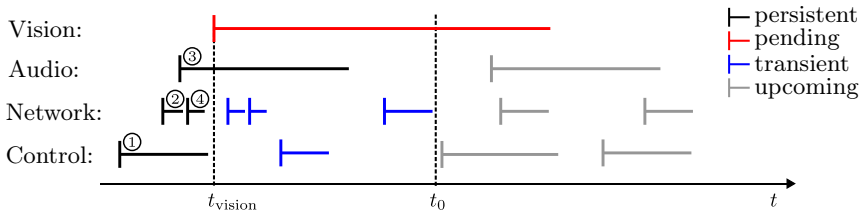

Fig. 2. Timing of four cyclers with interleaved execution.

A straightforward yet inefficient method involves delaying the processing of faster cycle data until the long-running cycle completes its computation, leading to increased latency. Instead, we solve this asynchronous characteristic by queuing futures of pending cycles. A cycler announces the start of a cycle by adding

a future to the queue. This future promises the completion of a pending cycle and its associated database. Each entry is associated with a timestamp. Cyclers consuming from the queue are thus able to respect temporal ordering.

Figure 2 illustrates the evaluation of the queue at timestamp t_0. At this point in time, the long-running vision cycle has not yet finished its computation and its results can thus not yet be consumed. Therefore, the queue is split into persistent and transient data. The persistent is all data that is timestamped before t_{vision} and already finished execution until timestamp t_0. All data that is timestamped after t_{vision}, is transient data. Despite their transient nature, nodes may consume this data to produce an estimate on top of the persistent state.

2.6 Configuration Parameters

Execution may not only depend on incoming data, but also on configuration parameters. This allows configuration of algorithm at runtime, e.g., changing the coefficients of a filter. On startup, the initial parameters are read from disk and made available to nodes by the framework. Parameters are organized and stored on disk in a priority-based directory structure. This allows quickly swapping the set of parameters based on the location, e.g., for individual soccer fields. Additionally, parameters may adopt different values for each individual robot.

2.7 Communication

User interaction for debugging and monitoring is tightly integrated into the framework's architecture. A dedicated communication channel allows observing outputs of robotics nodes, altering parameters at runtime, and receiving information about the framework's internal state. Debug client applications can connect via a JSON-oriented WebSocket protocol, which supports many client implementations. The design of communication consists of several asynchronous tasks that manage resources and communicate via message passing.

2.8 Recording and Replay

Real-world interactions with robots may sporadically uncover rare bugs within the robotics domain that are hard to reproduce. To analyze, reproduce, and fix such bugs, the framework incorporates recording and replay functionality. These capabilities are integrated through an adjustment of database storage as the framework manages the execution of nodes and cycler database operations. However, storing all computed outputs from every node across all cyclers proves impractical. Instead, the framework records inputs of each cycle and captures the state of all nodes. Paired with deterministic node implementations, this data is sufficient to replay each recorded iteration without additional dependencies.

2.9 Code Generation and Domain Specific Language (DSL)

The implementation is split into two parts: the framework itself and the robotics code. By analyzing robotics code, the framework automatically generates the infrastructure to start cyclers, spawn operating system threads, create nodes, and share data. The overall structure is specified in a DSL, the framework manifest.

2.10 Programming Language Selection

As highlighted in section 1.2, the choice of programming language is a pivotal aspect of the framework's design. Our selection, Rust, stems from its various advantages. Rust exhibits performance comparable to other leading compiled languages and is specifically crafted for concurrent and secure application development [5]. Notably, a Google survey praises Rust's compiler error messages as "amazing", emphasizing its facilitation of writing correct and review-friendly code [3]. Unique features like ownership and borrowing contribute to more concise and comprehensible code compared to languages like C. Additionally, Rust has an extensive package ecosystem which simplifies the incorporation of external libraries, enhancing flexibility and extensibility.

3 Evaluation

In this section, we present the application of the framework at Team HULKs, quantify its execution time overhead and assess its efficiency in real-world robotic applications. Additionally, we systematically examine the framework against the predefined requirements, offering an insightful analysis of how the framework aligns with each criterion.

3.1 Case Study: HULKs Robotics Domain

Team HULKs employs the presented framework to execute their robotics domain on the NAO robot. Environmental triggers originate from various sources, including new camera images, incoming network messages, microphone samples, and sensor measurements. Each trigger source is processed in dedicated cycler instances: Vision Top, Vision Bottom, Network, Audio, and Control.

The Control cycler executes the motion control algorithm, reading robot sensor measurements and sending actuator commands. The outputs from all other cyclers are treated as perceptions and consumed by the Control cycler in a temporally-ordered fashion. This decoupling allows asynchronous processing without affecting robot motion control. Most other teams run their behavior for every new sensor or camera event. At Team HULKs, we execute the behavior in the real-time cycler based on filtered perception inputs, which allows running behavior on predicted models instead of outdated data.

Team HULKs also develops various debug applications that connect to the communication interface. These applications provide real-time insights into the

state of the robot, enabling developers to monitor the execution of individual cyclers and diagnose any potential issues. Over the past two years, the framework has been used in the RoboCup competition, demonstrating its robustness and reliability.

3.2 Quantitative Evaluation of Execution Time Overheads

A critical performance measure is the execution time overhead of the framework, quantifying the duration spent by each cycler executing framework code in contrast to robotics algorithms. While an ideal scenario entails minimal overhead, several features such as database sharing between cyclers necessitate a non-zero amount of additional copying and synchronization.

Experimental Configuration To conduct our assessment, we evaluate the runtime overhead of the HULKs robotics codebase. Measuring via deterministic profiling accumulates large and difficult to quantify overheads due to node execution times within the microsecond range. Instead, we use a sampling profiler, specifically Intel VTune, to measure the CPU time expended in framework and robotics code. Data collection occurs over a 5-minute duration on a NAO robot walking across the SPL field, representing a typical usage scenario for Team HULKs.

Results The evaluation, summarized in fig. 3, presents the overhead of our framework in comparison to the time spent in robotics code for each cycler.

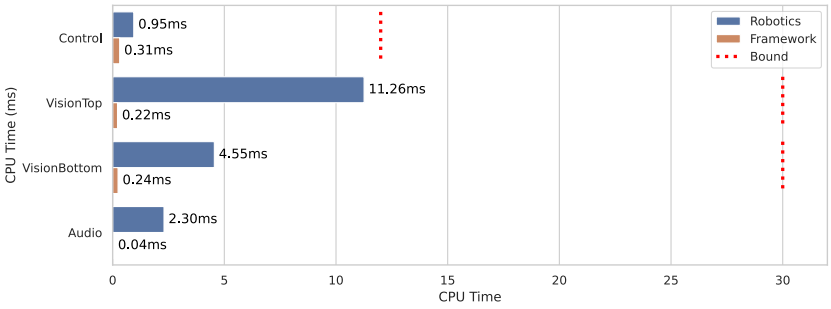

Fig. 3. Profile of the CPU time in the HULKs code base across different cyclers. For each cycler, the runtime is split into the framework overhead and the robotics code execution time, with a red upper bound, if in viewport.

The upper bound for the Control cycler is 12 ms, Vision's is 33 ms, while the Audio cycler's is 46 ms. These bounds should not be violated to prevent skipping of input data. As shown in fig. 3, the robotics code for each cycler operates well

below its upper bound. The framework overhead for the Vision cycler is measured at 0.24 ms, while the robotics code averages at 11.26 ms. The framework execution time is approximately 50 times shorter compared to the robotics code. The upper Vision cycler takes approximately 2.5 times more execution time than the lower Vision cycler, attributable to more points of interest in the larger area covered by the upper camera.

Notably, the overhead is relatively larger for the Control cycler compared to the Vision cyclers, primarily due to its consumption of futures from all other cyclers. The audio cycler, comprising only two nodes, demonstrates a relatively smaller overhead, measured at blazingly fast 0.04 ms.

Overall, the framework's overhead is significantly lower than both the execution time of robotics code and the bound of the respective cycler, emphasizing its efficiency in real-world robotic applications.

3.3 Qualitative Evaluation of Requirements

This section reviews the predefined requirements (c.f. section 1.2) that serve as benchmarks for our robotics framework.

Each cycler instance operates autonomously via dedicated operating system threads to leverage the *parallelization* from multiple CPU cores. A fundamental aspect is the decoupling of cyclers, ensuring independence in processing. This design enables long-running cyclers to asynchronously conclude their tasks, while giving *precedence to motion cycles*. A substantial amount of resource-intensive computations, including source analysis, node ordering, and execution code generation, is shifted to compile time. This paradigm ensures a *low runtime overhead*. The framework excels in providing a clear *separation* between the robotics code development and the details of node execution. Users can build on enhanced modularity and extensibility as they can extend the robotics domain by incorporating new nodes and cyclers in the framework manifest, dynamically generating fitting execution code. Cycles are initiated by setup nodes, *adapting computation* dynamically to environmental triggers. *Temporally-ordered data* access allows for coherent interaction with filters in real-time cyclers. The framework's *abstraction over hardware* permits seamless execution on diverse hardware configurations without demanding modifications. Intra-process communication between nodes is achieved via direct memory access, minimizing the need for *serialization and reflection*. Only the communication subcomponent that allows external *tooling support* uses *serialization* to encode values for network transfer. The framework and robotics domain are implemented in the Rust *programming language* which provides high computational efficiency. The Rust language's focus on memory safety and concurrency aligns well with the demands of robotic applications, reducing the likelihood of bugs and enhancing overall system reliability. In parallel, ROS, known for its robust middleware in the robotics domain, operates primarily in C++. While ROS offers a wealth of pre-existing modules and community support, the presented Rust-based framework provides a more streamlined and efficient development experience.

3.4 Adoption in RoboCup SPL

Since its open-source release, our framework has been rapidly gaining popularity among teams in the RoboCup SPL. For the 2023 RoboCup competition, the Dutch Nao Team adopted the HULKs framework [4]. Team HULKs helped the Dutch Nao Team getting started both with the framework and robotics code. Using the HULKs framework, they achieved the 4th place in the Challenger Cup at RoboCup 2023 [21]. The Rhinobots team from Brazil considers using the HULKs framework at RoboCup 2024.

4 Conclusion and Outlook

In this paper, Team HULKs present a novel robotics framework designed to overcome challenges and shortcomings of existing solutions. Our framework is engineered to optimize performance, enhance modularity, and establish a seamless interface between robotic hardware and detection as well as control algorithms. The framework has low overhead and fulfills all requirements, e.g., extensibility, temporal ordering, and adaptive computation. The code is developed as open source software at https://github.com/HULKs/hulk. Other teams are already using the framework at RoboCup, and are contributing successfully.

In the future, more features are planned, e.g., profiling, remote procedure calls, and logging. In addition, Team HULKs plans to extract the framework code to increase reuse beside the open development. Due to its separation between robotics and framework domain, this general framework may also be used in many other applications beyond the RoboCup scope.

References

1. Asavkin, M., Gopikrishnan, N., Lizura, M., Sammut, C., Schmidt, P., Vijayan, A.: rUNSWift Team Report 2022. Tech. rep., UNSW School of Computer Science and Engineering (2022)
2. Baillie, J.C., Demaille, A., Hocquet, Q., Nottale, M., Tardieu, S.: The Urbi universal platform for robotics. In: First International Workshop on Standards and Common Platform for Robotics. Citeseer (2008)
3. Bergstrom, L., Brennan, K.: Rust fact vs. fiction: 5 Insights from Google's Rust journey in 2022
4. Bolt, L., Gunnewiek, F.K., gezegd Deprez, H.L., van Iterson, L., Prinzhorn, D.: Dutch Nao Team - Technical Report. Tech. rep., University of Amsterdam (Dec 2022)
5. Costanzo, M., Rucci, E., Naiouf, M., Giusti, A.D.: Performance vs Programming Effort between Rust and C on Multicore Architectures: Case Study in N-Body. In: 2021 XLVII Latin American Computing Conference (CLEI). pp. 1–10. IEEE, Cartago, Costa Rica (Oct 2021). https://doi.org/10.1109/CLEI53233.2021.9640225
6. Ditzel, S., Hess, T., Rinfreschi, K., Siegl, J.M., Weiglhofer, F., Nonnengießer, F., Hahner, B., Schon, T., Dehen, J., Fritz, R.: Bembelbots Team Research Report for RoboCup 2019. Tech. rep., Goethe University Frankfurt (Jan 2020)

7. Einhorn, E., Langner, T., Stricker, R., Martin, C., Gross, H.M.: MIRA - middleware for robotic applications. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 2591–2598. IEEE, Vilamoura-Algarve, Portugal (Oct 2012). https://doi.org/10.1109/IROS.2012.6385959
8. Gerkey, B., Vaughan, R., Stoy, K., Howard, A., Sukhatme, G., Mataric, M.: Most valuable player: A robot device server for distributed control. In: Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180). vol. 3, pp. 1226–1231. IEEE, Maui, HI, USA (2001). https://doi.org/10.1109/IROS.2001.977150
9. Google: Protocol Buffers. https://protobuf.dev/
10. Hayes-Roth, B.: A blackboard architecture for control. Artificial Intelligence **26**(3), 251–321 (Jul 1985). https://doi.org/10.1016/0004-3702(85)90063-3
11. Huang, A.S., Olson, E., Moore, D.C.: LCM: Lightweight Communications and Marshalling. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. pp. 4057–4062. IEEE, Taipei (Oct 2010). https://doi.org/10.1109/IROS.2010.5649358
12. Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. IEEE Transactions on Parallel and Distributed Systems **33**(6), 1303–1320 (Jun 2022). https://doi.org/10.1109/TPDS.2021.3104255
13. Khan, S., Bailey, D., Gupta, G.S.: Simulation of Triple Buffer Scheme (Comparison with Double Buffering Scheme). In: 2009 Second International Conference on Computer and Electrical Engineering. vol. 2, pp. 403–407 (Dec 2009). https://doi.org/10.1109/ICCEE.2009.226
14. Larisch, A.: An Efficient Real-Time Capable Multi-Core Module Framework for the Humanoid Robot NAO. Master's thesis, TU Dortmund University (2020)
15. Macenski, S., Foote, T., Gerkey, B., Lalancette, C., Woodall, W.: Robot Operating System 2: Design, Architecture, and Uses In The Wild. Science Robotics **7**(66), eabm6074 (May 2022). https://doi.org/10.1126/scirobotics.abm6074
16. Mellmann, H., Schlotter, B., Kaden, S., Strobel, P., Krause, T., Couque-Castelnovo, E., Ritter, C.N., Martin, R.: Berlin United - Nao Team Humboldt Team Report 2019. Tech. rep., Humboldt-Universität zu Berlin (2019)
17. Metta, G., Fitzpatrick, P., Natale, L.: YARP: Yet Another Robot Platform. International Journal of Advanced Robotic Systems **3**(1), 8 (Mar 2006). https://doi.org/10.5772/5761
18. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: An open-source robot operating system. In: ICRA Workshop on Open Source Software. vol. 3, p. 5. Kobe, Japan (2009)
19. Röfer, T., Laue, T.: On B-Human's Code Releases in the Standard Platform League – Software Architecture and Impact. In: Behnke, S., Veloso, M., Visser, A., Xiong, R. (eds.) RoboCup 2013: Robot World Cup XVII. pp. 648–655. Springer, Berlin, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44468-9_61
20. SoftBank Robotics: NAO - Developer Guide — Aldebaran 2.8.7.4 documentation. http://doc.aldebaran.com/2-8/family/nao_technical/index_naov6.html
21. SPL Technical Committee: Standard Platform League Results 2023 – RoboCup Standard Platform League. https://spl.robocup.org/results-2023/
22. Tsardoulias, E., Mitkas, P.: Robotic frameworks, architectures and middleware comparison (2017). https://doi.org/10.48550/ARXIV.1711.06842