Towards the design of efficient and versatile cognitive robotic architecture based on distributed, low-latency working memory

Juan Carlos García
RoboLab research group
Universidad de Extremadura
Cáceres, Spain

Pilar Bachiller
RoboLab research group
Universidad de Extremadura
Cáceres, Spain

Pablo Bustos
RoboLab research group
Universidad de Extremadura
Cáceres, Spain

Pedro Núñez
RoboLab research group
Universidad de Extremadura
Cáceres, Spain
pnuntru@unex.es

***Abstract*—Autonomous robots will be present in our daily lives in the coming years. One of the most critical elements facilitating this expansion of robots is the concept of Cognitive Robotic Architectures (CRAs). Thanks to these CRAs, robots are aware of their state and surroundings and then build all the behaviors best suited to the scenario on this information. In recent years our team has proposed an CRA called CORTEX designed for use with autonomous robots working in human environments. CORTEX is based on a distributed graph-like working memory where software agents can read or update information. In this paper, we describe the design process of the new CORTEX architecture up to its current implementation. Among the most salient design requirements is data synchronization between the different agents in the architecture, low latency, and performance of the new architecture. To validate the effectiveness of the architecture and its versatility, we have used our CRA in different use cases, including social robot navigation and autonomous driving of connected vehicles.**

***Index Terms*—cognitive architecture, robotics architecture**

I. Introduction

The next generation of autonomous robots will coexist with humans in uncontrolled environments and multiple purposes. The main goal of cognitive robotics is to provide autonomous robots with the ability to plan complex actions and execute those plans while adapting to unexpected changes. Until now, most architectures have been problem-specific, providing the robot with the ability to perceive and act in the environment [1]. However, robotic applications are multiplying, including interactions with people. For this reason, it is necessary to design versatile and efficient cognitive architectures capable of adapting to different autonomous platforms and responding adequately to real requirements.

For decades, considerable effort has been made to endow robots with cognitive architectures. In the comprehensive review by Kotseruba et al., [1], the authors state that only very few versatile architectures implement multiple skills for complex scenarios. One of the architectures cited is CORTEX [2], [3], our Cognitive Robotics Architecture (CRA) proposal as an evolution of the RoboCog architecture [4]. In CORTEX, software agents share a distributed and dynamic working memory that acts as common knowledge. This working memory is called Deep State Representation (DSR), and its structure is a graph composed of nodes and arcs. The term Deep refers to the hybrid nature of the elements of this graph, geometric and symbolic, concrete or abstract.

Any efficient cognitive architecture must address four key dimensions of the design space: i) the trade-off between decoupling and sharing; ii) the trade-off between top-down/downward control; iii) the functional content of software agents; and iv) the granularity of functional decomposition. Since the conception of CORTEX, we studied different design choices to include these critical features. First, CORTEX defines software agents as components that provide specific and limited functionality. For knowledge sharing, while there are alternatives such as the dynamic approach [7], in our proposal, these agents have access to a distributed shared memory. The second dimension is implemented in CORTEX through deliberative agents with high-level reasoning capabilities that generate efficient plans in execution times (top-down) and through the agents themselves locally that communicate with the rest of the agents (down-top). As for the third item, CORTEX defines the role of each agent in the global problem space. Finally, the fourth dimension involves meeting a series of requirements to have complex architectures but, at the same time, be computationally efficient and operate in real-time.

CORTEX is currently used in different types of robots working in real-world scenarios. Some of these use cases implement complex scenarios involving human-robot interactions. For example, the robot working in public space, such as an airport, attracts potential consumers to a commercial stand [2]. In works presented in [8], [9], robots perform geriatric tests on older adults or physical therapies with children, respectively. Human-aware navigation in different environments with people has also been addressed in various use cases [10], [11]. In all these situations, the complexity of the software is palpable, with more than forty components interconnected through the CORTEX architecture. After almost five years of experience in these highly complex use cases, we can assure that one of the main problems is the bottleneck when accessing

978-1-6654-8217-2/22/$31.00 ©2022 IEEE

shared memory. This last causes desynchronization or delays that, in many situations, harm the efficiency of the architecture.

In [12], the authors outline the first redesign of CORTEX based on two key technologies: a high-performance pub/sub middleware that implements reliable UDP multicast and the use of Conflict-Replicated Data Types (CRDT). The **main contribution** of this paper is to extend this initial proposal, delving into the design process and evaluating CRA performance in real use cases. We formulate the problem from the beginning, providing the design requirements and the solutions proposed in our final architecture. Furthermore, we validate the technology in terms of efficiency and robustness in different complex use cases.

This paper is organized as follows: In Section II we presents a brief description of CORTEX architecture and the Deep State Representation. The design requirements and the solutions achieved are detailed in Section IV III. Section IV details the architectural design and outlines the implementation of our RCA. Experimental results are addressed in Section V, and finally, Section VI describes the main conclusions of this works.

II. CORTEX ARCHITECTURE AND DEEP STATE REPRESENTATION

The CORTEX cognitive architecture defines how to design, modularize and represent the robot's activities and information. In this architecture, activities can be seen as the cooperation and coordination between agents performing specific tasks by writing to a shared data structure. We define this distributed working memory as Deep State Representation (DSR). We find that the information stored in DSR has the own robot's knowledge regarding modularization and activity representation. This information may be known beforehand, it may have been obtained from sensor data, or it may be the result of the execution of an agent [6].

Formally, our DSR Γ_{dsr} is described as the union of two *quivers*: one associated with the symbolic part of the representation, $\Gamma_s = (V_s, E_s, s_s, r_s)$, and the other related to the geometric part, $\Gamma_g = (V_g, E_g, s_g, r_g)$. A quiver is a quadruple, consisting of a set V of nodes, a set E of edges, and two maps $s, r : E \rightarrow V$. These maps associate each edge $e \in E$ with its starting node $\mathbf{u} = s(e)$ and ending node $\mathbf{v} = r(e)$. In some situations, we denote an edge by $e = \mathbf{uv} : \mathbf{u} \rightarrow \mathbf{v}$ with $\mathbf{u} = s(e)$ and $\mathbf{v} = r(e)$. Within the DSR, both quivers are finite, as both sets of nodes and edges are finite sets. A *path* of length m is a finite sequence $\{e_1, ...e_m\}$ of edges such that $r(e_k) = s(e_{k+1})$ for $k = 1...m-1$. A path of length $m \geq 1$ is called a *cycle* if $s(e_1)$ and $r(e_m)$ are identical.

The representation of the data stored in DSR is independent of their nature, although the way they are handled can be specific. The clearest example is the representation of physical objects. Physical objects are represented in the graph as nodes and related to other elements by means of an arc. The edge includes the geometric information in a homogeneous $4x4$ matrix. Non-geometric information is subject to groups of agents creating their own abstractions of the internal representation

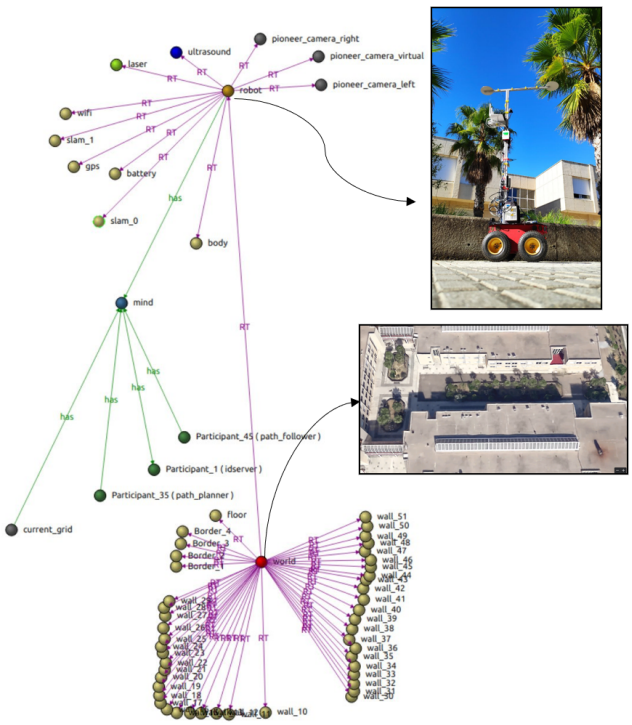

Fig. 1: A simple example of the DSR state in an instant of time. Edges labeled as has denote logic predicates between nodes. Edges starting at robot and end at specific sensors (camera, gps...) are geometric relations and encode a rigid transformation RT between them.

of Γ_{dsr}. A node can belong to more than one abstraction. Figure 1 shows a simple example of the DSR of the Pioneer P2AT robot endowed with different sensors and working in an outdoor area. DSR can be seen as a multigraph consisting of the different abstractions defined by the agents. High-level tasks are performed by decomposing them into specific, smaller tasks, which are implemented in the form of an agent. Agents can add, modify or delete DSR information. These changes are available to all other agents without the need for direct communication between them. By adding to the data structure the ability to notify agents of external changes, even reactive agents that only act on demand can be achieved. The ability to execute multiple missions may require a mission planning process that ensures the compatibility of the activities performed in the missions. This paper deals with the design and implementation of this structure, Γ_{dsr}, and the agents that execute it.

III. DESIGN REQUIREMENT

At the design level, we identify a set of critical requirements and capabilities that a cognitive architecture should support. These requirements are described next.

A. Synchronization

In a distributed system, **synchronization** is the process by which multiple system participants perform asynchronous

local operations to reach a common state through the messages sent over the network and the operations performed to integrate these messages. The synchronization process is dependent on the consistency and representation model of the information. If the information is stored in a centralized manner, the synchronization process is irrelevant, while in an environment with local replicas, it is a fundamental part [6]. Since CORTEX activities are divided into agents that work in parallel and independently of each other, maintaining local replicas seems to be more appropriate. In this section, it is necessary to discuss how synchronization will be achieved, what messages will be sent, what content the messages will have, how messages are integrated, how conflicts are resolved, the frequency of message sending, and the characteristics of the communication.

The tool used in CORTEX to perform the synchronization operations is called **CRDT** (Conflict-Free Replicated Data Types) [13]. CRDTs are data types that have the property of providing eventual consistency of distributed objects. Specifically, the CRDT type used is the **Multi-Value Register** (MVreg) [14]. This type keeps only the last local write that has been performed on the object. To identify which is the last written data, each local replica includes a monotonic counter that is updated on writes. When a synchronization message is received from another remote object, this counter is used to find if it is newer than other replicas and replace the current one. If concurrent writes have occurred that result in the same counter both objects are kept in the register, it may be necessary to establish how to choose which write is to be kept if a single value is desired.

B. Performance

When designing the architecture of Γ_{dsr}, the performance requirements of the environments in which it will be executed have been taken into account. In an execution of CORTEX we find multiple agents that obtain information, either from a sensor or from Γ_{dsr}, transform it and insert it back into Γ_{dsr}. These agents are in charge of tasks such as the navigation of a robot, the detection of objects by cameras, etc. These types of tasks require that the information accessed be as recent as possible. Keeping the time elapsed between data retrieval by one agent and its subsequent reading by a different agent within a few tens of milliseconds is a requirement for many of the tasks performed to be feasible. In a proper implementation of the data structure the biggest source of latency in the information flow of the system would be the communication network.

C. Memory

The decisions about information representation, synchronization, and the agent-based architecture of CORTEX have implications for memory usage. Each agent holding a replica of the complete state in memory has a considerable total memory cost ($M \cdot N$, where M is the total memory of the representation of Γ and N is the number of agents). This last is justifiable considering the amount of RAM that most computers have today and that the graph representation usually takes at most a few MB. However, much more information could be stored if necessary. Moreover, it seems reasonable to accept higher memory usage to simplify synchronization information access and increase availability and performance.

To mitigate the increased memory usage, the implementation of the data representation must make efficient use of memory. In addition, mechanisms have been included to allow agents to discard information they do not intend to use.

D. Network

The network specifications are directly related to the performance requirements and the synchronization model used. On the one hand, to satisfy the performance requirement, we must use technologies that allow fast information delivery to all agents, regardless of the number of agents in the system. It is also necessary that the size of the messages sent is optimal. This last is achieved in the design stage, with the use of Delta-CRDTs [14], and a suitable communication protocol, and in the implementation, with the information representation and communication libraries. On the other hand, for the synchronization model, it is necessary to ensure that all agents receive the messages sent.

We use the **RTPS** (Real-Time Publish-Subscribe) communication protocol over a reliable multicast communication using UDP [15]. RTPS is a protocol that is part of the **DDS** (Data Distribution Service) standard, published by the OMG (Object Management Group). It is designed to be real-time, error-tolerant, scalable, configurable (network trust, memory usage, performance, among others). The protocol follows a publish-subscribe pattern, allowing communication to be divided by message types and offering a typing system in the topics to eliminate possible errors.

E. Scalability

Different factors must be taken into account for the system to be scalable. Specifically, at the network level, we achieve scalability by using multicast for transport. This last allows us to significantly reduce the load on the network by reducing the number of messages per update from $M - 1$ in point-to-point communication with M participants to only a single message using multicast. The use of Delta-CRDT for synchronization results in smaller messages, resulting in less network usage and fewer synchronization operations. At the system design level, the distributed agent-based architecture allows easy addition and removal of system participants. The replication of information per agent solves a potential problem of high information access times caused by synchronization operations if the information were centralized in a single location. Moreover, using an agent or component-based architecture allows the system to be much more distributable than in a monolithic or less adaptable architecture.

F. Versatility

Several factors determine the versatility of cognitive architecture. Among them is the difficulty encountered in migrating

the architecture to other autonomous systems or the possibility of including new independent agents with little effort from users. The less effort an architecture requires to produce intelligent behavior in those environments, the greater its versatility. Using an agent-based architecture facilitates this requirement, this modularity being indispensable to increase versatility.

IV. Architectural design and implementation process

The design and implementation have been done through an iterative and incremental process based on a cycle of the definition of requirements and functionalities, implementation, search for errors and new requirements through testing and development of agents, correction of errors, optimization, and repetition of the process with the new needs found. The Test-Driven Development methodology inspires part of the definition of requirements, design, and tests. As the DSR core and the agents that use it are developed in parallel, validating the implementations and decisions made during development is necessary. The way to do this has been the definition of tests and benchmarks in the requirements identification and design stages. In addition to the unit tests commonly used in this methodology, the agents are themselves used as tests. Here, the requirements often appear as a response to the task to be accomplished by an agent. By using agents as tests, new requirements are created that can be implemented and validated almost directly in DSR. As the iterative process moves forward, the requirements are higher. In the initial iterations, the aim was to be reasonably stable and for communications to work correctly. In contrast, in an intermediate iteration, the system must be robust. In the last iterations, the aim is for the APIs to be simple for users, providing a more intelligent code generation for the agents.

The DSR architecture implements a distributed data structure based on local copies that synchronize local states with remote states. The two main elements in each local instance are the graph and the communication framework for synchronizing information between local copies. The graph elements are stored on MVReg containers, with three CRDT levels, one at the node level, one at edge level, and one at the attribute level. This granularity is intended to reduce the network load and the computational cost of consistency. The agents access the information and make local modifications through an API that encapsulates both elements. At the same time, the synchronization process is performed asynchronously and hidden from the user, giving the programmer the feeling of using a local data structure. To achieve this, the API must be thread-safe. The programming language chosen for the implementation is **C++**. The use of a typed language offers guarantees during development, compilation, and execution.

The system architecture is divided into three libraries. One for the communication elements, the types represented by the graph, the CRDT types, etc. A second one with the APIs accessed by the programmer and a third one for the user interface. To simplify development by maintaining APIs with a reasonable number of functions, the extension of DSR functions is done by implementing specific APIs for specific tasks using the available DSR API. An agent manages a DSR instance, accessed through the DSR API block with one of the public or extension APIS. Optionally, it can also access the user interface. Internally, these APIS access the graph, which is kept up to date with synchronization operations that fetch data through the Fast-DDS instance. Local changes are also published to DDS so that the same thing happens in other running agents.

V. Experimental results and discussion

To analyze the performance of Γ_{dsr}, a series of benchmarks have been performed to test the behavior of the different technologies used and the implementation decisions that have been made. These tests measure network latency in different operations, execution times and network usage. The tests were performed on a computer equipped with an Intel i9-10900K processor, 64GB of DDR4 memory on an Ubuntu 20.04 system. Our proposal is integrated into the framework roboComp[1].

A. performance analysis

The performance analysis evaluates the cost of local attribute updates. The test is focused on this operation since it is the most common in DSR. Attribute updates of edges and nodes are the same, and their cost is almost identical (in the edges update, there is extra access to a map). In figure 2a you can see a comparison between the cost of a local update for different message sizes and the cost of the same update on a remote agent. Local updates can be performed in three different ways. The first one consists of modifying the attribute by copying it and updating the node, making a copy. In the second option, the attribute is moved, and the node is copied. In the third, the attribute and the node are moved. There is no significant impact on performance for small message sizes, around 60KB, when using the different alternatives. For attributes larger than 5MB the cost of copies increases a lot. Currently, the limitation in the performance of local updates is in the implementation of MVReg, which performs a copy of the values when generating a delta. It would be appropriate to determine whether it is necessary to perform the copy or is some kind of optimization possible that avoids it. It should be noted that most attributes used in DSR are minor attributes of a few bytes, such as numbers, text strings, arrays of a few elements. Nevertheless, large attributes usually correspond to messages that are sent in short periods, such as images, so it is essential to process them efficiently.

The other operations performed on Γ_{dsr} are insertion and deletion of nodes and insertion and deletion of edges. Figure 2b shows the cost in microseconds of the operations in an environment locally and remotely on the same host. The nodes and edges inserted in the tests did not include any attributes.

Figure 2c compares the network usage for a different number of connected agents in which a single agent publishes and

[1]https://github.com/robocomp

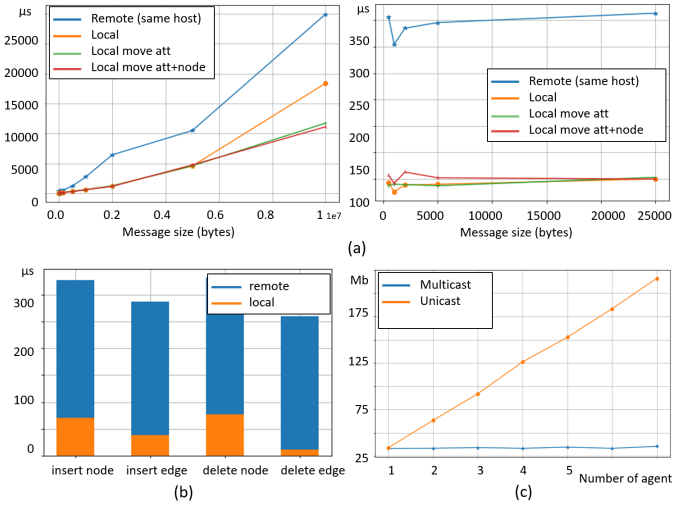

Fig. 2: a) Comparison of update latency for local and remote messages on the same host using different local update methods; b) Comparison of local and remote latency for different operations; and c) Networkg usage with one agent publishing and multiple agents receiving using multicast and unicast.

the rest receive. The publishing frequency and published data are the same throughout the test. The result on network usage is as expected. Unicast has a linear growth while Multicast keeps the network usage constant.

B. Use cases

The implementation of simulated and real use cases during the development of the libraries makes it possible to validate the decisions taken in the development stage and to check that the requirements of scalability, versatility, performance, latency and consistency are achieved. We used similar agents in all the use cases, sending information of different nature and sizes. The experiments have not been limited to a single test; on the contrary, we have repeated them with similar results[2].

1) human-awareness navigation in simulated scenario: The first use case implemented is human-awareness robot navigation in an indoor, controlled space. Social robot navigation (*i.e.*, the robot navigates in a similar way that people do) is a complex process that requires multiple tasks running in parallel. Some of these tasks are social path planning, people and obstacle detection, positioning the robot itself, and real-time path correction. Depending on the complexity of the environment, these number of tasks can be increased or reduced. The agent-based architecture allows the change in complexity to be reduced to adding agents that work on the navigation-related data. In our implementation, six agents work in parallel. We add the agent that connects to the simulator and other agents related to the interaction between the robot and the people, adding tasks such as people detection, control over navigation routes, etc.

[2]The reader can view the videos of these use cases at https://youtu.be/yzEDNB4mxj4

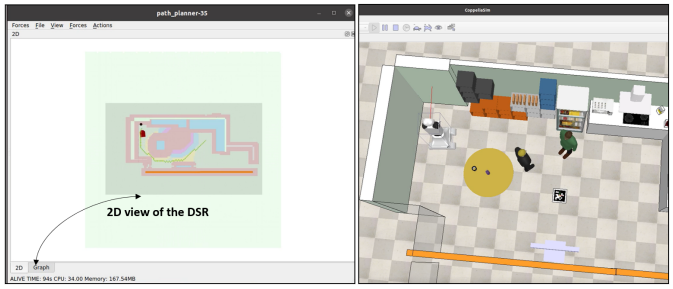

Fig. 3: First use case: human-awareness robot navigation using the Coppelia (Vrep) simulator

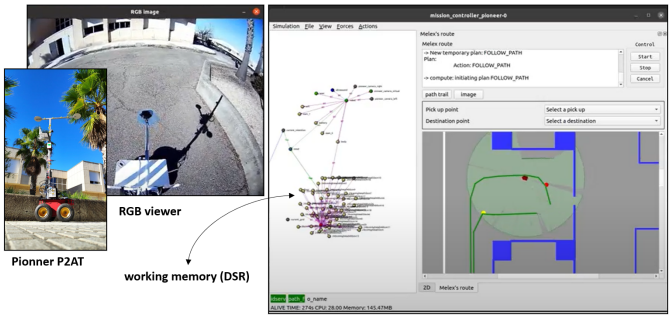

Fig. 4: Second use case: outdoor robot navigation in real scenario using the Pioneer P2AT robot. The video shows the most relevant agents of the experiment, including the local navigation agent.

The six main agents implemented are: i) path_follower, in charge of moving the robot using the route calculated by other agents and validating the movements with the information from the omnidirectional laser; ii) social_path_planner_astar, in charge of planning the route and updating it iteratively; iii) social_elastic_band, which smoothes the path designed by the previous agent using the robot's sensors; iv) mission_controller_simulated, which allows navigation by creating plans that subsequently generate the routes; v) human_social_spaces agent, in charge of the generation of personal and interaction spaces for humans; and v) bumper, which creates a virtual bumper that restricts movements very close to physical elements, correcting the robot's trajectory. Fig. 3 shows a snapshot during the navigation of the robot, where the 2D view of the DSR can be seen in the user interface of the social agent.

2) Outdoor navigation in real scenarios: The second use case is the navigation of a Pioneer P2AT robot in an outdoor environment. In addition to working in a larger environment, when leaving the simulator and using real robots, we may encounter much less predictable operations and situations than those that appear in the simulator. These differences can appear in the information from the sensors (inaccuracy in cameras, lasers, etc.), in the robot components (wheels, servomotors, etc.) and even in the environment (changes in furniture, groups of people, etc.).

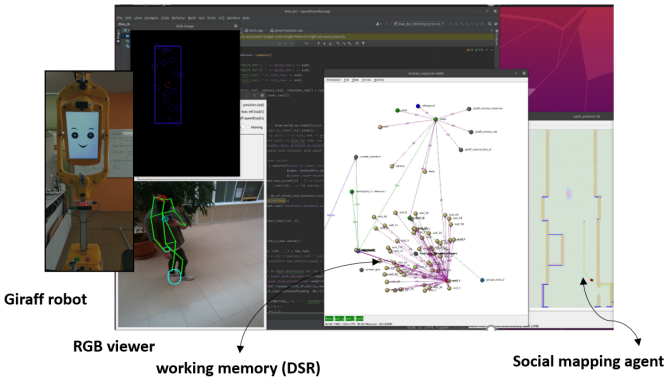

Fig. 5: Third use case: indoor robot navigation in real scenario with people. The video depicts the output of the most relevant agents of the architecture: among others, the social mapping agent, the human capturer agent, RGBViewer, and the evolution of the DSR.

Thanks to the component-based architecture, it is possible to reuse most of the navigation agents (path_planner_astar, path_follower, elastic_band, bumper, etc.) in robots with different sensors and features by changing only the configuration files. This is also because the agents do not try to operate the robot directly, but write information to the shared memory and then a specific agent for the interaction with the robot is in charge of setting the robot in motion. The result of this design and programming model is agents that are much more versatile and less dependent on the specific technologies of each use case. For each type of robot there are specific agents for the missions they perform. In the case of the Pioneer robot, it is mission_controller_pioneer, which replaces the mission_controller_simulated agent. For security and testing reasons, robocomp agents or components are also added for manual control of the robots.

3) Indoor robot navigation in real scenarios with people: In the third use case, the robot used is a differential base endowed with different sensors for its navigation in an indoor environment. The agents used are similar to those presented in the first use case but, this time, facing real situations. In addition, in these spaces, different sensors are installed that add information to DSR and allow the execution of more complex tasks or with greater precision. The representation of all the information acquired by the sensors in the physical space, the robot's sensors, the pre-known information and the logical information of the agents generates a cyber-physical space. This space is stored in DSR. Some of the sensors that are contemplated to be included are cameras, used for the detection of people, temperature, CO2 or humidity. Fig. 5 illustrates the Giraff robot during the tests. The DSR is also shown in the figure.

VI. Conclusions

In this paper, we examine the design process of a new cognitive architecture for robots that extends the CORTEX architecture to a low-latency distributed working memory Γ_{dsr}. The proposal combines δ-CRDTs and a high-performance pub/sub middleware. We have validated the architecture in terms of throughput, consistency and latency, prerequisites for multiple agents of different complexity to work together to pursue a goal. We have evaluated the architecture in three use cases of different complexity, both in simulated and real environments. Our results are good indicators of the potential that a distributed, low-latency working memory may have in future RCAs, where software complexity will be a significant issue for developers and roboticists.

Acknowledgment

This work has been partially supported by the Feder funds and by the Extremaduran Goverment projects GR21018, IB18056, and by the MICINN RTI2018-099522-B-C42.

References

[1] Kotseruba, I., Gonzalez, O., Tsotsos, J. "A Review of 40 Years of Cognitive Architecture Research: Focus on Perception, Attention, Learning and Applications", in Tech. rep, 2016.

[2] Romero-Garces, A., Calderita, L.V., Martínez, J., Bandera, J.P., Marfil, R., Manso, L.J., Bandera, A., Bustos, P. "Testing a fully autonomous robotic salesman in real scenarios", in IEEE International Conference on Autonomous Robot Systems and Competitions, pp. 1–7, 2015.

[3] P. Bustos García, L. J. Manso Argüelles, A. Bandera, J. P. Bandera, I. García-Varea, and J. Martínez-Gómez. "CORTEX: a new Cognitive Architecture for Social Robots" in Eucognition meeting – cognitive robot architectures, Viena, 2016.

[4] L. J. Manso, L. V. Calderita, P. Bustos, J. Garcia, M. Martinez, F. Fernandez, A. Romero-Garces, and A. Bandera. "A General-Purpose Architecture to Control Mobile Robots" in XV Workshop of physical agents: Book of proceedings, 2014.

[5] Bustos, P., Manso, L., Bandera, J., Romero-Garces, A., Calderita, L., Marfil, R., Bandera, A. "A unified internal representation of the outer world for social robotics", in ROBOT conference, vol. 2, pp. 733–744, 2015.

[6] L.V. Calderita. Deep State Representation: an unified internal representation for the robotics cognitive architecture CORTEX. PhD thesis, Universidad de Extremadura, 2016.

[7] RD Beer. Dynamical approaches to cognitive science. Trends in cognitive sciences, 4(3):91–99, mar 2000.

[8] D. Voilmy, C. Suarez, A. Romero-Garcés, C. Reuther, J.C. Pulido, R. Marfil, L. Manso, K. Lan, A. Iglesias, J.C. González, J. Garcia, A. Garcia-Olaya, R. Fuentetaja, F. Fernández, A. Duenas, L. Calderita, P. Bustos, T. Barile, J.P. Bandera, and A. Bandera. "Clarc: A cognitive robot for helping geriatric doctors in real scenarios", in ROBOT, volume 1, pages 403–414, 2017.

[9] J.C. Pulido, J.C. González, C. Suarez-Mejias, A. Bandera, P. Bustos, and F. Fernández. "Evaluating the child-robot interaction of the naotherapist platform in pediatric rehabilitation", in International Journal of Social Robotics, pages 16–26, 2017.

[10] Vega, A., Manso. , Luis J.. Macharetc, D., Bustos, P., and Núñez P. "Socially aware robot navigation system in human-populated and interactive environments based on an adaptive spatial density function and space affordances", in Pattern Recognition Letters, vol 118, pp. 72–84, Elsevier, 2019.

[11] A. Vega, L. V. Calderita Estévez, P. Bustos García, and P. Núñez Trujillo. "Human-aware robot navigation based on time-dependent social interaction spaces: a use case for assistive robotics", in 2020 IEEE International Conference on Autonomous Robot Systems and Competitions, 2020

[12] J.C. García García. "G: a low-latency, shared-graph for robotics cognitive architectures", Master Thesis, Escuela Politécnica, 2021.

[13] M. Shapiro, N. Pregui, C. Baquero, and M. Zawirski. "Conflict-free Replicated Data Types". Research Report RR-7687, July 2011.

[14] P. S. Almeida, A. Shoker, and C. Baquero. "Delta state replicated data types". CoRR, abs/1603.01529, 2016.

[15] https://www.eprosima.com/index.php/resources-all/whitepapers/rtps, availabe february, 2022.