DSR_d : A proposal for a low-latency, distributed working memory for CORTEX

Pablo Bustos[1], J.C. García,[1], R. Cintas[1],
E. Martirena[1], P. Bachiller[1], P. Núñez, and A. Bandera[2]

[1] Robotics and Artificial Vision Laboratory, University of Extremadura, Spain
[2] ISIS, School of Telecommunication Engineer, University of Málaga

Abstract. Robotics Cognitive Architectures (RCA) are becoming a key element in the design of robots that need to be aware of its surrounding space and of their role in it. This is especially important for robots that interact with people in household, eldercare or industrial collaborative scenarios. We have proposed in earlier works an RCA called CORTEX designed for social robots operating in HRI environments. One of CORTEX's main elements is a working memory designed as a graph-like data structure that is accessed by all the computational modules in charge of some relevant function in the system. Our current implementation is based on the concept of a real-time database, where one of the modules stores, receives and publishes changes to all modules. In this paper, we propose a new design of this element based on the Conflict-free Distributed Replicated Data Types (CRDT) theory of distributed data types. The new working memory presents important advantages over existing designs that are demonstrated with several experiments.

Keywords: social robotics, cognitive architectures, CRDT, working memory

1 Introduction

Cognitive robotics is concerned with endowing robots with the capacity to plan solutions for complex goals and to enact those plans while being reactive to unexpected changes in their environment. To pursue this goal, cognitive architectures for robotics attempt to provide a reasonable structure where all the functionalities of a working cognitive robot can be fit. If the final goal is to endow these architectures within future service robots, the new design should provide an adequate response to the demanding requirements imposed by the human-robot interaction scenario.

Despite the large efforts for making cognitive and social robots a potential counterpart of human users, it is currently not easy to find successful stories, neither in the industrial segment, nor in the academic world. In the exhaustive review by Kotseruba et al. [1], they assert that only some architectures implement multiple skills for complex scenarios. One of the cited architectures is CORTEX [2] [3], a proposal emerged from previous National and European

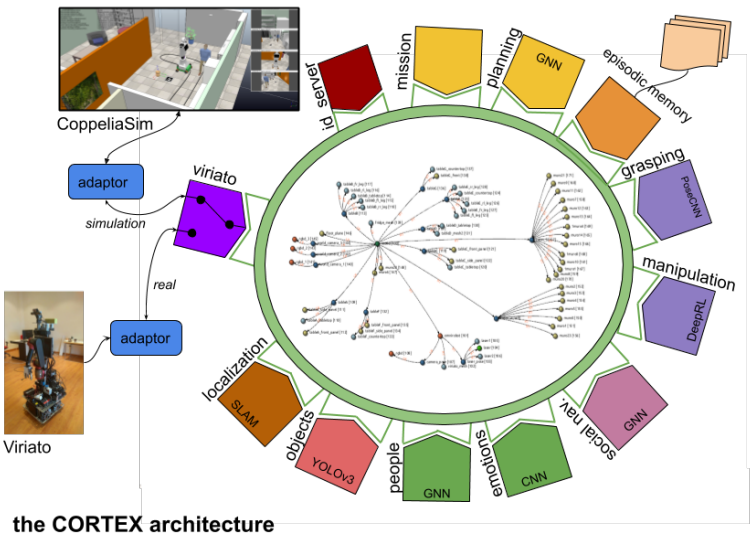

Fig. 1. The illustration shows a possible instance of the CORTEX architecture. The central part of the ring contains the DSR graph that is shared by all agents, from whom a reference implementation is presented here. Coloured boxes represent agents providing different functionalities to the whole. The purple box is an agent that can connect to the real robot or to a realistic simulation of it, providing the basic infrastructure to explore prediction and anticipation capabilities.

projects granted to our consortium. CORTEX is a long term effort to build a series of architectural designs around the simple idea of a group of agents that share a distributed, dynamic representation acting as a working memory. This data structure is called Deep State Representation (DSR) due to the hybrid nature of the managed elements, geometric and symbolic, and concrete (laser data) and abstract (logical predicates) [4]). Figure 1 shows the main elements of CORTEX in its current state.

The CORTEX architecture exposes four important dimensions of the RCA design space: the trade-off between decoupling and sharing; the trade-off between top-down/bottom-down control; the functional content of the agents; and the granularity of the functional decomposition. Decoupling is the main engineering asset to tackle complexity. Accordingly, agents are defined as software encapsulated groups of components that provide some specific, limited functionality[3]. In this sense, encapsulation provides decoupling but isolates each agent

[3] The term *agent* is used here as a synonym of module and, as such, no specific features such as goal-seeking or autonomy are imposed.

from the valuable contextual information created by the others. To combine their functionalities and show some degree of intelligence, agents must be able to share information[4]. Each of them must know something about the others, otherwise their goals will remain local, and complex sequential tasks will be outside the reach of their individual abilities. The second dimension mentioned above is the top-down/bottom-up control trade off, by which the system has to allow for the existence of simultaneous and opposite streams of control. Top-down control in CORTEX is generated by, at least, one deliberative agent with the capacity to reason about high level tasks and compute efficient plans to drive the behavior of the robot. Bottom-up reactions to unforeseen situations are locally handled by agents while trying to fulfil plan steps. It should be noted that the planning agent must react itself if the execution of the plan fails. Finally, these reactions might trigger a cascade of changes in other agents through the shared memory, giving raise to large behaviour modifications. The third design axis deals with how the functional space, that is to be managed by the RCA, is partitioned in local domains. In CORTEX, this is the problem of defining the role of each agent in the overall problem space. Finally, the fourth dimension raises the issue of how large the functional domain of these agents can be in order to fulfill two requirements: being simple enough to facilitate the design of complex architectures; and being computationally realizable in terms of CPU usage, communication delays and software maintainability.

Since the conception of CORTEX, different design choices have been studied to integrate these features. As briefly noted below and further described in the references, several implementations of the CORTEX architecture have been deployed in different types of robots working in real world scenarios during these last years. All these use cases have a deep human-robot interaction component and include situations such as attracting potential consumers to a stand in public spaces [2], conducting geriatric tests to elderly people [6], conducting physical therapies with children [7], searching and bringing objects to humans or as a companion to the householder of an adapted apartment [8], or human-aware navigation in different types of environments with people and objects [9]. The implementations of these experimental setups have grown to a considerable software complexity, reaching up to 45 interconnected components.

On such large configurations, where all agents contribute from their functional domains to a common working memory, the access to data can emerge as a relevant bottleneck. The solution presented in this work is based on two key technologies that have the potential to reduce bandwidth consumption by the set of agents and maintain local immediate responsiveness to operation on the graph. The first one is a high-performance pub/sub middleware[5] implementing RTPS (Real-time Publish-Subscribe Protocol) and configured to use UDP reliable multicast. The data sent by an agent is broadcasted to all the others at

[4] We do not claim here that a shared representation is the only way to share information among agents. The *dynamicist* approach is a well-known alternative [5].

[5] https://www.eprosima.com/index.php/resources-all/performance/40-eprosima-fast-rtps-performance

once, minimizing the required bandwidth compared to point-to-point connections. The second one is a theoretical development named Conflict-free Replicated Data Types (CRDT) that provides data structures that can be safely and asynchronously edited by a distributed set of processes without a central server. By combining both technologies we have been able to create a new, highly efficient, distributed graph (the DSR_d). Contrary to our previous design, each local DSR_d copy can be now asynchronously edited by all the agents. Their changes are sent as incremental state modifications through the network using minimal resources and, when these updates are received, the local graph in each agent is updated concurrently with the user's operation, using a thread-safe interface to the graph. This new server-less version will only exist as the set copies maintained by the agents, that will eventually converge to the same final state after all editing is stopped. To our knowledge there are no other published CRDT graphs with the functionality required here.

The rest of the paper is organised as follows: Sections 2 and 3 describe the internal structure of the graphical representation of the DSR and the relevance of the eventual consistency in our context. The implementation of the CRDT graph is presented in Section 4, where we also provide preliminary experimental validation. Finally, conclusions and future work are drawn in Section 7.

2 The Deep State Representation

Conceptually, the DSR represents a network of entities and relations among them. Relations can be unary or binary predicates, while the entities may have complex numeric properties such as pose transformation matrices that represent the kinematic relations of objects in the world and the robot's parts. Mathematically, the DSR is internalized as a directed graph with attributed edges. As a hybrid representation that stores information at both geometric and symbolic levels, the nodes of the DSR store concepts that can be symbolic, geometric or a combination of both. Metric concepts describe numeric quantities of objects in the world, which can be structures such as a three-dimensional mesh, scalars such as the mass of a link, or lists such as revision dates. Edges represent relationships between nodes. Two nodes may have several kinds of relationships but only one of them can be geometric. The geometric relationship is expressed with a fixed label called *RT*. This label stores the transformation matrix (expressed as a rotation-translation) between them.

The DSR can be described as the union of two *quivers*: one associated with the symbolic part of the representation, $\mathit{\Gamma}_s = (V_s, E_s, s_s, r_s)$, and the other related to the geometric part, $\mathit{\Gamma}_g = (V_g, E_g, s_g, r_g)$. A quiver is a quadruple, consisting of a set V of nodes, a set E of edges, and two maps $s, r : E \to V$. These maps associate each edge $e \in E$ with its starting node $\mathbf{u} = s(e)$ and ending node $\mathbf{v} = r(e)$. Sometimes we denote an edge by $e = \mathbf{uv} : \mathbf{u} \to \mathbf{v}$ 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 $\{e_1, ...e_m\}$

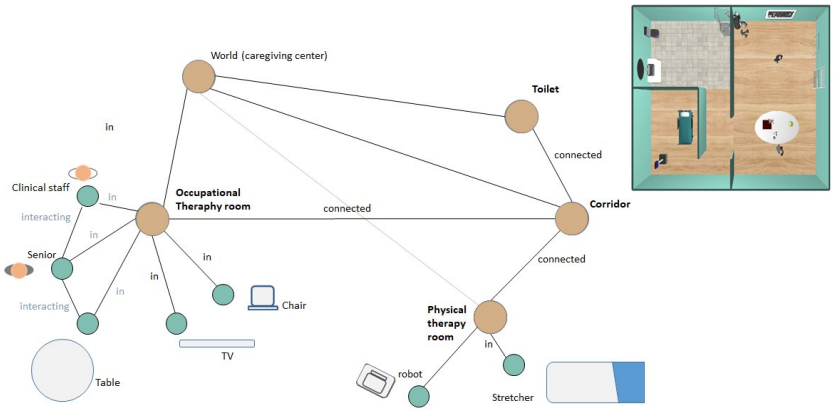

Fig. 2. Unified representation as a multi-labelled directed graph. For instance, the edge labelled as connected denotes a logic predicate between nodes and it belongs to Γ_s. On the other hand, the edge starting at the **Occupational therapy room** and ending at **the chair** is geometric and encodes a rigid transformation (RT' and RT respectively) between them. Geometric transformations can be chained or inverted to compute changes in coordinate systems.

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.

This embedding of geometric and symbolic structures within the graph is exploited in the implementation described below through specific access APIs, that provide specialized methods to compute long-range kinematic transformation among distant nodes, in the first case, and PDDL conversion or some forms of reasoning, in the second.

3 Eventual consistency and CRDTs

One problem with the standard server-oriented design that is currently being used in CORTEX is that an increase in the graph's size or in the data density will quickly increase latency and reduce throughput across the communication network. Once latency is too high, the context provided by the DSR becomes very difficult to use, since events in the world will happen faster than the reaction time required to compute a response. In the worst case, an agent could be always working on its local copy but being unable to publish its result. To address this problem we need to find a solution that provides both a minimum band-width use and a minimum amount of data exchanges among agents. Both requirements must hold while a certain level of coherence among the local copies is guaranteed.

This work proposes a new implementation of the DSR that is fully distributed, provides very low-latency and a high and sustainable throughput. Additionally, this new design scales up nicely in the number of agents, in the size of data objects (*i.e.*, raw sensor data from a camera or a LIDAR) injected at high

rates and in the number of nodes (objects) and edges (relationships or predicates). The advantages of this high-performance working memory in CORTEX are plenty: i) agents could be designed with a finer grain while maintaining full access to graph; ii) the designer could exploit this feature to think of smaller functional elements, which implies simpler software components that still can communicate through this shared medium; iii) the graph could contain a wider range of abstract nodes, going from raw laser data to complex predicates that represent human intentional states; and iv) the participating agents will be able to inject, delete or modify nodes and edges as the result of local decisions extended with the global graph data. We aim at latencies below 50 ms for a wide range of configurations and payloads. With this value and assuming that most real-world interaction loops, at a mid-abstraction level, run at 10 Hz, the agents will view the internal representation is always updated.

It is well-known that low-latency or high-availability in a distributed network of partitioned nodes is subject to the CAP theorem [10]. One cannot have both low-latency and consistency (*i.e.*, having an up-to-date copy of the data) at the same time if the data is stored in several nodes that can loose or degrade their connection at any time. Consistency requires synchronous updating of the state. If one wants consistency, it may happen that after receiving a write operation two nodes cannot communicate, their updates cannot be synchronously transmitted to the rest and a blockage will occur, thereby forfeiting low-latency. If one chooses low-latency, you need to allow that at least one node updates its state to provide a quick response, making the other nodes inconsistent and thus forfeiting consistency. It should be noted that even if all agents run on the same board or in boards connected by a network switch, they are partitioned nodes that can enter or leave the group at any time, and each one could have their own group of clients -e.g local threads- requesting information.

In this proposal we choose low-latency and give up strong consistency in favour of a form of weak consistency named *eventual consistency* [11] [12]. Under this form of consistency, changes made to one copy of the shared object are asynchronously propagated to the rest, but if all update activity stops, after a period of time all replicas of the data object will converge to the same state. This means that at some point in the future, all data in the nodes become indistinguishable from one another. In general, to ensure convergence, nodes must exchange information with another about what writes they have seen. While strong consistency requires synchronous updates to keep the semantics of a single-system image (SSI), eventual consistency uses asynchronous updates. When one node receives an update, it is executed locally, without synchronization. Then, it is sent to other nodes. All updates eventually reach all nodes, asynchronously and possibly in different orders.

CRDTs are distributed data types that are always available and eventually converge when all operations are processed at all nodes [13]. CRDTs come in two flavors: *operation-based* and *state-based*. In operation-based designs a representation of the local operation is created in the local node and shipped then to all other nodes. Once received, the representation of the operation is applied to

the local copies of the receiving nodes. Alternatively, in a state-based design an operation is applied only on the local node's copy. Periodically, the node propagates its local changes to other nodes through shipping its entire copy. The other nodes incorporate the received copy with their local one via a *merge* function that deterministically reconciles both copies[6]. Although operation-based CRDTs allow for simple implementations and require smaller band-width, they require exactly-once causal broadcast and a new operation has to be defined for each kind of modification to the graph. State-based systems do not have this limitation but they may incur in high communication overheads when shipping large size local copies. Recently, a new improvement on state-based systems has been proposed that mitigates this problem. Delta-state RDT are based on propagating only a *representation of the effect* of the update operation on the copy [14] [15]. Instead of shipping the complete copy, this new design allows the nodes to send only δ-states (i.e incremental changes) of their copies.

4 Design and Implementation

The design of the distributed graph (DSR_d) is organized in three loosely coupled layers, as shown in Figure 3. The deepest one contains the RTPS logic, as provided by eProsima in its FastDDS open source library. The next layer contains the CRDT logic, based on the C++ code kindly provided by C. Baquero [7]. We have used a CRDT implementation of a map data type as the core structure to wrap inside a graph with the characteristics described in Section 2. The third layer is the one exposed to the users of the tool and holds a view of the graph accessible through a thread-safe API and specialized sub-APIs. This design allows the replacement of each layer in case a better technology becomes available.

One of the main goals in the creation of this tool has been is to simplify robotics code development when the system is composed of many distributed components. There are several features that we believe are mandatory to achieve this goal:

1. *A very simple deployment procedure with a minimum set of configuration parameters.* To start an instance of CORTEX, the user only needs to get the robot or the simulator ready and run first a special agent named *idserver*. This agent provides a unique identifier creation service required by all agents to add new nodes to the graph, being this one of the requirements of the underlying theory. When *idserver* starts, it can read a previously stored graph on disk, as a JSON file. At this point, any other entering agent can ask for an updated copy of the graph, using a topic reserved for this purpose. Once it gets the copy it can safely start its operation.
2. *Automatic code generation of all parts non dependant on the user code.* All these new agents are still software components created within the RoboComp development framework [16]. Thus, all generic code for these agents

[6] Convergence is guaranteed by defining merge as a least-upper bound over a join-semilattice.

[7] https://github.com/CBaquero/delta-enabled-crdts

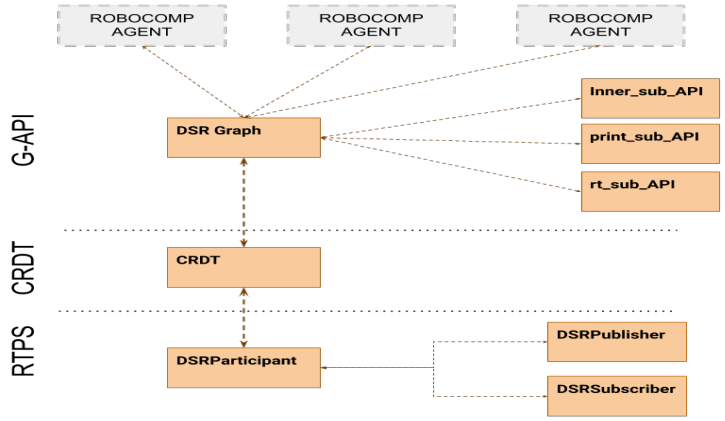

Fig. 3. Layered design of the working memory. From bottom-up, each layer is conceived to be replaceable and provides additional functionality to the system.

is automatically generated from a very simple DSL specification where only the name is mandatory. Since no topics, ports, interfaces or IPs are needed any more, a coder knowing the framework can create and start a new agent connected to an existing CORTEX instance in less than thirty seconds.

3. *A simple, intuitive and thread-safe user-level API to access the graph.* Concurrency originated between the user access to the graph and the merge operations triggered by incoming data from other agents, or by the user accessing the graph from several threads, is handled by the core methods using the new `std::shared_mutex` primitive and the associated `std::shared_mutex` operation provided in C++17. To access a node, the user requests a copy of it and the operation is protected with a read only `"shared_lock"` operation, that allows read access to several concurrent threads. When the user finishes editing the node, it is copied back to the graph using, this time, a `"unique_lock"`, that waits until all reading threads finish. Inserting new nodes or edges is not problematic and deleting nodes removes also all incoming and outgoing edges using the `"unique_lock"` mechanism.
4. *A dictionary of node and edge types, and of all possible names and types of both, node and edge attributes.* One common source of errors when building sets of components that share a structured representation like a graph is the misnaming of nodes, edges and attributes among coders. This situations usually take a long time to debug. The goal with DSR_d is to completely eliminate the source of these errors. The current solution uses a common repository of types and names that all the team must use. However, we are working in a more radical solution using the new *constexpr* functionality provided in C++17 to keep in an include file all names and types of nodes, edges and attributes. This will result in compile-time errors when the user attempts to create or edit an element using the wrong name or data type.

5. *A graphical and textual display of the current state of the graph integrated in each agent.* Integrated in the generated code, each agent can optionally include a graphical interface, implemented with Qt5, that provides several, selectable, simultaneous, real-time views of the graph. These representations are interactive allowing the inspection of graph elements, and are intended as a powerful high-level debugging tool. Four views are provided: a standard planar drawing of the graph; a 2D representation of the geometric nodes embedded in the graph using Qt 2D drawing framework; a 3D view of the same content using OpenSceneGraph[8]; and a textual tree view which can be used to hide or show groups of nodes and edges in large graphs. Additional views can be easily added since the core methods of G-API emit Qt signals whenever a change is made. Another important feature that facilitates the debugging of complex CORTEX configurations, is the possibility to draw inside this graphical representations without perturbing the graph. An example could be the drawing of landmarks, targets, bounding boxes or robot paths used or produced by an agent.

All the code that managed the graph is compiled as a dynamic library and linked to each agent. The user *sees* the graph G as a local data structure that is accessed through its G-API.

5 Examples and experiments

Since DSR_d is still a prototype under heavy testing, optimization and debugging, we will show here a series of examples of increasing complexity, each one with a corresponding video. We still don't have metrics that could give an idea of the final performance of the tool, except for the throughput and latency figures provided by eProsima on FastDDS, which are among the best of their class. The synchronization layer using CRDTs adds some overload that will depend on the impact of the *merge* operation on each node update. Also, it is difficult to provide quantitative comparisons here since we don't know of similar tools designed with the same purpose.

The first example [9] shows the initiation procedure of a CORTEX instance running against the CoppeliaSim robotics simulator. The example shows a two-step process. First, CoppeliaSim is started using a non-agent RoboComp component named *ViriatoPyRep*. This is Python program that imports PyRep [17] to run CoppeliaSim in the same loop and loads a model of RoboLab's adapted apartment, ALab. As a component, it provides RoboComp interfaces for all elements of the robot and for all external virtual devices installed in the apartment. In the second step, the agent *idserver* is started and it reads an initial graph from a JSON file. This stored graph contains the same scene that was loaded in CoppeliaSim but without the person. Once the agent has started, it opens a

[8] http://www.openscenegraph.org/

[9] Video: *first-example* available in the list https://www.youtube.com/playlist?list=PLDkfV8Ufc2i1p-viGTV3QKupFD2ute535

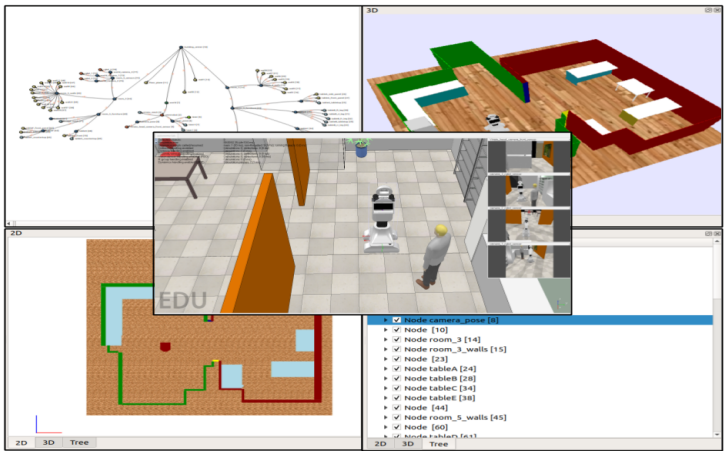

Fig. 4. Composition for Example 1 showing in the first plane the simulated scene of the ALab with the robot, a person and three additional cameras placed on the walls. Each of the four background captures shows a different representation of the graph made available to the user as part of the generated code. Top-left, is the standard graph representation; top-right is the OSG 3D view automatically computed from the graph; bottom-left is Qt 2D view also computed from the graph; and bottom-right is a textual tree view.

window with the views described before. Figure 4 shows a composition of the simulated scene and the views provided by the agent.

The second example [10] shows the deployment of a second agent, *ViriatoDSR*, that acts as a bridge between the simulated robot and the graph. On startup, this agent broadcasts, on a special topic, a request for a fresh copy of the graph. The request is served by the agent *idserver*, already running. *ViriatoDSR* is connected to *ViriatoPyrep* using RoboComp's interfaces and the Ice middleware, and to the graph using FastDDS. All low-level sensor data from the robot is injected into the graph, including the robot's RGBD camera streams. The use of reliable multicast reduces the bandwidth to the net load of one transmission plus some control information. The first running agent, *idserver*, receives the updates on the attributes of the nodes corresponding to the camera, the LIDAR and the position of the robot in the scene. The video shows how this data can be accessed from the UI in real-time. It also shows nodes and edges flashing when they are being updated by incoming data. See Figure 5

In the third example [11], a third agent is introduced that access the number and pose of the avatars included in the scene. The agent used here is a simplified

[10] Video: *second-example* available in the list https://www.youtube.com/playlist?list=PLDkfV8Ufc2i1p-viGTV3QKupFD2ute535

[11] Video: *third-example* available in the list https://www.youtube.com/playlist?list=PLDkfV8Ufc2i1p-viGTV3QKupFD2ute535

Fig. 5. Composition for Example 2 in text. A second agent has been started that acts as a bridge between the Python adapter, ViriatoPyrep and the graph. All received data is injected as it arrives and propagated to the other agent. The left part of the figure shows the first agent's UI, with the views for the LIDAR (an attribute of node *laser* that hangs from node *omnirobot*) and the camera. The right part shows the same data for the second agent. The central image corresponds to the scene being simulated. The capture in the lower side of the image corresponds to an instantaneous measure of the band-width measured with the Linux utility *iftop*. The 35Mb/s figure corresponds to one RGB stream and shows that bandwidth remains constant with the number of agents.

version of the real one that uses OpenPifPaf[12] and other DNNs to detect complete bodies, track them and estimate their orientation [18]. When this agent detects a person, it inserts it in the graph creating a new node and connects it to the world coordinate system using an RT edge. The avatar is shown in the 2D and 3D views and the new node is propagated to the other agent, where it is also depicted. The edge's rotation and translation attributes are hereinafter updated whenever a change is read from the simulator. See Figure 6.

The last example [13] shows how a fourth agent is started. This new participant takes the RGB image from the node camera's in its updated copy of the graph and processes it with the DNN YoloV3 [14]. The algorithm detects three objects in the virtual scene that are created as new nodes in the graph and made available to the other agents. See Figure 7.

[12] https://github.com/vita-epfl/openpifpaf
[13] Video: *fourth-example* available in the list https://www.youtube.com/playlist?list=PLDkfV8Ufc2i1p-viGTV3QKupFD2ute535
[14] https://pjreddie.com/darknet/yolo/

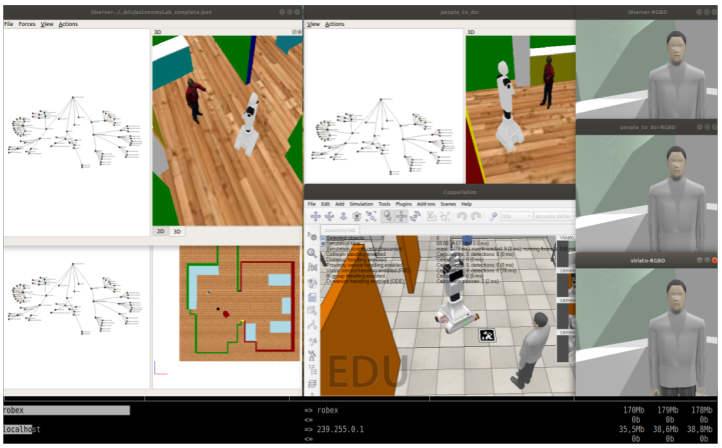

Fig. 6. Composition for Example 3 in text. The figure shows three agents running and sharing the graph. When the third agent initiates, it injects the detected person into the graph and the new nodes are propagated to the other agents. The three similar UIs show the reconstructed scene from different points of view. On the right side, the flow of images arriving to each agent is shown. Again, the total bandwidth in the multicast address is kept constant.

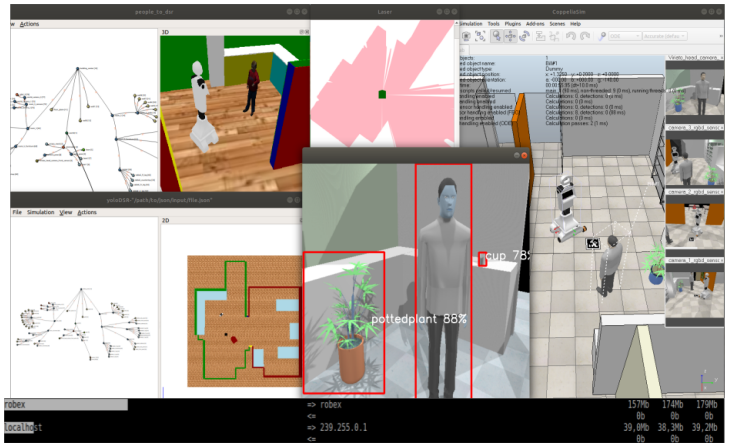

Fig. 7. Composition for Example 4 in text. A fourth agent is added that integrates the DNN YOLOv3 to detect object in the virtual scene, The left part of the composition shows the UIs of two agents, one of them displaying the LIDAR data in its graph. The right part shows the current scene in CoppeliaSim and the window in the center shows the detected objects over the processed image.

6 Conclusions and future work

The results presented in this work are a first step towards a fully operational distributed DSR_d. We have proposed a design based on a new combination of δ-CRDTs and an high-performance pub/sub middleware. The initial experiments have shown that several agents can share a non-trivial graph and edit its nodes for long periods of time without compromising its integrity. At the end of the trials, all copies have been checked to have the same contents. Also, the inspection of the traffic published by the middleware shows that only data belonging to the modified nodes or edges are published and, still, eventual consistency is achieved every time. Additionally, the window of consistency in these tests using only one host is very short as expected. All these results are good indicators of the potential that a distributed, low-latency, working-memory can have in future RCAs, where software complexity will be a major problem for developers and roboticists. As aforementioned, dealing with the software complexity that comes with an increasing number of agents, is one of our more urgent priorities and more tools will be needed to assist in debugging these large ensembles of processes.

7 Acknowledgments

This work has been partially funded by the EU RobMoSys project (H20202-732410), the project RTI2018-099522-B-C4X, funded by the Spanish Ministerio de Ciencia, Innovación y Universidades and FEDER funds, the EU INTERREG-POCTEC project 0043-EURAGE-4-E, and the Extremaduran Goverment projects GR15120 and IP IB16090

References

1. Kotseruba, I., Gonzalez, O., Tsotsos, J.: A Review of 40 Years of Cognitive Architecture Research: Focus on Perception, Attention, Learning and Applications. Tech. rep. (2016). URL http://arxiv.org/abs/1610.08602
2. Romero-Garcés, 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: ICARSC, pp. 1–7 (2015)
3. Bustos, P., Manso, L., Bandera, J., Romero-Garcés, A., Calderita, L., Marfil, R., Bandera, A.: A unified internal representation of the outer world for social robotics. In: ROBOT, vol. 2, pp. 733–744 (2015)
4. Calderita, L., Bustos, P., Mejías, C.S., Fernández, F., Viciana, R., Bandera, A.: Asistente Robótico Socialmente Interactivo para Terapias de Rehabilitación Motriz con Pacientes de Pediatría. Revista Iberoamericana de Automática e Informática Industrial RIAI **12**(1), 99–110 (2015). DOI 10.1016/j.riai.2014.09.007
5. Beer, R.: Dynamical approaches to cognitive science. Trends in cognitive sciences **4**(3), 91–99 (2000)

6. Voilmy, D., Suarez, C., Romero-Garcés, A., Reuther, C., Pulido, J., Marfil, R., Manso, L., Lan, K., Iglesias, A., González, J., Garcia, J., Garcia-Olaya, A., Fuentetaja, R., Fernández, F., Duenas, A., Calderita, L., Bustos, P., Barile, T., Bandera, J., Bandera, A.: Clarc: A cognitive robot for helping geriatric doctors in real scenarios. In: ROBOT, vol. 1, pp. 403–414 (2017)
7. Pulido, J., González, J., Suarez-Mejias, C., Bandera, A., Bustos, P., Fernández, F.: Evaluating the child-robot interaction of the naotherapist platform in pediatric rehabilitation. International Journal of Social Robotics pp. 16– (2017)
8. Bandera, A., Bandera, J.P., Bustos, P., Férnandez, F., García-Olaya, A., García-Polo, J., García-Varea, I., Manso, L.J., Marfil, R., Martínez-Gómez, J., Núñez, P., Perez-Lorenzo, J.M., Reche-Lopez, P., Romero-González, C., Viciana-Abad, R.: LifeBots I: Building the software infrastructure for supporting lifelong technologies. In: Advances in Intelligent Systems and Computing, vol. 693, pp. 391–402 (2018). DOI 10.1007/978-3-319-70833-1_32
9. Vega-Magro, A., Manso, L., Bustos, P., Núñez, P., Macharet, D.: Socially acceptable robot navigation over groups of people. In: RO-MAN 2017 - 26th IEEE International Symposium on Robot and Human Interactive Communication, vol. 2017-Janua (2017). DOI 10.1109/ROMAN.2017.8172454
10. Brewer, E.: CAP twelve years later: How the "rules" have changed. Computer **45**(2), 23–29 (2012). DOI 10.1109/mc.2012.37
11. Bailis, P., Ghodsi, A., Hellerstein, J.M., Stoica, I.: Bolt-on causal consistency. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 761–772 (2013). DOI 10.1145/2463676.2465279
12. Terry, D.: Replicated data consistency explained through baseball. In: Communications of the ACM, vol. 56, pp. 82–89 (2013). DOI 10.1145/2500500
13. Baquero, C., Almeida, P.S., Shoker, A.: Pure Operation-Based Replicated Data Types (609551), 1–30 (2017). URL http://arxiv.org/abs/1710.04469
14. Almeida, P.S., Shoker, A., Baquero, C.: Delta state replicated data types. Journal of Parallel and Distributed Computing **111**, 162–173 (2018). DOI 10.1016/j.jpdc.2017.08.003
15. Enes, V., Almeida, P.S., Baquero, C., Leitao, J.: Efficient synchronization of state-based CRDTs. Proceedings - International Conference on Data Engineering **2019-April**, 148–159 (2019). DOI 10.1109/ICDE.2019.00022
16. Manso, L., Bachiller, P., Bustos, P., Calderita, L.: RoboComp: a Tool-based Robotics Framework. In: N. Ando, S. Balakirsky, T. Hemker, M. Reggiani, O. von Stryk (eds.) Simulation, Modeling and Programming in Autonomous RobotsRobots, vol. 6472, chap. LNCS 6472, pp. 251–262. Springer, Darmstadt, Germany (2010)
17. James, S., Freese, M., Davison, A.J.: Pyrep: Bringing v-rep to deep robot learning. arXiv preprint arXiv:1906.11176 (2019)
18.