UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Nombre de los Estudios

Trabajo Fin de Máster

G: a low-latency, shared-graph for robotics cognitive architectures

UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica

Nombre de los Estudios

Trabajo Fin de Máster

Título trabajo fin de máster

Autor: Juan Carlos García García

Tutor: Pablo Bustos García de Castro

Resumen

La arquitectura para robótica cognitiva CORTEX propone un modelo de ejecución distribuido basado en componentes, llamados agentes en la arquitectura, con una memoria compartida que representa la información conocida por el robot del mundo y de sí mismo. Esta memoria de trabajo recibe el nombre de Deep State Representation (DSR). En este proyecto se realiza una implementación de esta memoria distribuida con el objetivo de maximizar el rendimiento utilizando un modelo de representación basado en replicación mediante copias locales, con capacidad para almacenar información independientemente de su naturaleza o su representación y con un sistema de comunicación fiable y de baja latencia para mantener los estados de las réplicas sincronizados.

Abstract

The CORTEX cognitive robotics architecture proposes a distributed execution model based on components, called agents in the architecture, with a shared memory that represents the information known of the world and of the robot itself. This working memory is called Deep State Representation (DSR). In this project, an implementation of this distributed memory is presented, aiming to maximise performance using a representation model based on replication through local copies, with the capacity to store information regardless of its nature or representation and with a reliable, low-latency communication system to keep the states of the replicas synchronised.

Índice general

Índice de tablas

Índice de figuras

Capítulo 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 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

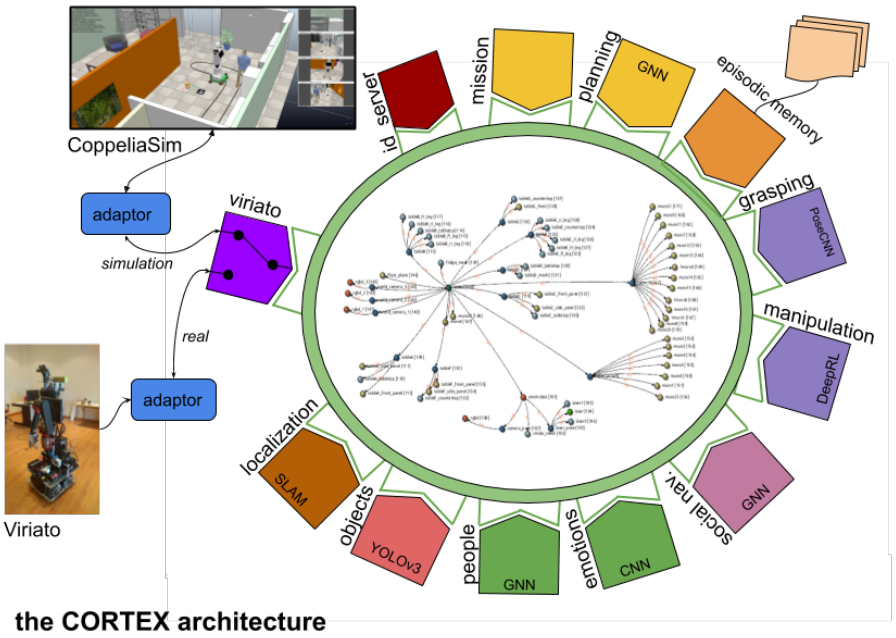


Figura 1.1: The illustration shows a possible instance of the CORTEX architecture. The central part of the ring contains the DSR graph that is shared by all agents, from whom a reference implementation is presented here. Coloured boxes represent agents providing different functionalities to the whole. 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.

predicates) [4]). Figure 1.1 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[1]. In this sense, encapsulation provides decoupling but isolates each

[1]The term *agent* is used here as a synonym of module and, as such, no specific features such

agent 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[2]. 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

as goal-seeking or autonomy are imposed.

[2]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].

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[3] 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 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.

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

Capítulo 2

Goals and achievements

Goals

The main goal of the development is the implementation of the distributed data structure G, called DSR in the CORTEX architecture, for use in the Robocomp framework. In addition to being distributed, the data structure must have the characteristics of high availability, be as fault tolerant as possible and ensure as much consistency as possible. During development, different possibilities for each of these aspects are explored and decisions are made. For example, it is possible to offer a centralized distributed access structure or a replicated structure, the consistency model can vary from not guaranteeing any consistency at all to ensuring strong consistency. The same is true for fault tolerance. Each of these decisions has implications on the ultimate performance and reliability of the system. Once these decisions have been made, it is necessary to do the same with the dependencies used; libraries, communication protocols, development technologies, etc. To make these decisions, it is important to clearly identify and specify what the system needs are.

In the implementation process, it is necessary to define how the data structure is going to be used, how to work with the information, what is the most appropriate programming model for the situations in which it is going to be used, etc.

The development is done through an iterative and incremental process, guided by the needs of the programmers and systems using the implementation. The ultimate goal is to evaluate whether the proposed model and its implementation are good enough to be used in applications as constrained as robotics, identifying weaknesses and possible improvements of the design and implementation.

Achievements

Among all the content in the following chapters of this document, the contributions of this TFM are:

- The design of the G architecture, selection of the development methodology and choice of the programming practices and techniques used for the implementation.
- The selection, configuration and integration of the technologies used for the implementation of G.
- The complete implementation of the G data structure, G types, support classes (Type checking, ThreadPool, Network classes, crdt, ...) and its access API.
- Testing.
- Performance analysis.
- DoubleBuffer class used in agents.

The final result has been integrated with pre-existing and third-party work made by other Robocomp developers. Specifically, G has been successfully integrated into the Robocomp components, resulting in the agents described in this document. It has also been integrated with the DSL for code generation and with different Information Viewers that are used by agents and programmers. Details on the integration and use of G can also be found in some of the *System*

Design sections. In addition, the *Use Cases* chapter shows how these tools have been successfully used by RoboComp developers to implement real use cases.

Capítulo 3

The CORTEX cognitive architecture

The CORTEX cognitive architecture defines how to design, modularize and represent the activities and information of a robot. In CORTEX, activities can be seen as the cooperation and coordination between agents performing specific tasks by writing to a shared data structure in the form of a graph called Deep State Representation (DSR). Regarding modularization and activity representation we find that the information stored in the graph is the representation of all the knowledge that the robot has about itself and the environment it is in at the current moment. 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. In the first type of information we find the specifications of the robot and the characteristics of the environment in which it is working. In the second we have the information that is obtained in real time from the sensors, such as images, lasers, the position of the robot, etc. The third type is the result of the execution of tasks, such as the identification of objects, people, non-physical concepts such as interaction, etc.

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 of G. A node can belong to more than one abstraction. 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 implementation of this structure, G, and the agents that execute it.

Capítulo 4

Design requirements

At the design level, according to the necessities defined in the CORTEX architecture and leaving the implementation details for its specific section, we can identify the following points as elements on which to develop the requirements of the developed system:

- Synchronization.
- Perfomance.
- Memory.
- Network.
- Scalability.
- Robocomp integration.

4.1. Synchronization

In a distributed system, **synchronization** is the process by which multiple system participants, performing local operations asynchronously, 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. Since in CORTEX activities are divided into agents that work in parallel and independently of each other, the option of maintaining local replicas seems to be more appropriate. It is necessary to discuss in this section 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, the characteristics of the communication, etc.

The tool used in DSR_d to perform the synchronization operations is called **CRDT** (Conflict-Free Replicated Data Types)[10]. 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). This type keeps only the last local write that has been performed on the object. To identify which is the last write, 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 out if it is newer and replace the current one. If concurrent writes have occurred that result in the same counter both objects are kept in the register, so it may be necessary to establish how to choose which write is to be kept if a single value is desired. More detail on how these types work is given in the specific section on this topic and in the implementation section.

4.1.1. Eventual consistency and other consistency models

Eventual consistency is a consistency model commonly used in highly distributed systems where data sharding is not possible and information is replicated across all system participants. In this type of consistency model, the state of the agents may differ at one point in the execution but tends to converge to a common state within a reasonable period of time. The need to use eventual

consistency with respect to strong consistency is given by the limitations proposed by the **CAP theorem**, which states that in a distributed information system no more than two of the three guarantees mentioned (Consistency, Availability and Partition Tolerance) can be offered. The consistency guarantee refers to the system's ability to maintain a synchronized state among the system's participants, the availability guarantee allows the information to be accessible at all times, even if it is not in its most current state, and the partitioning tolerance is the system's ability to continue running when any number of participants disappear from the system.

Seeking a balance between the CAP theorem constraints and CORTEX features, we have choosen a system with high availability, partitioning tolerance and eventual system consistency. The reason for preferring availability and partitioning tolerance is that, in the proposed model, each agent shares the complete state of the world but performs specific tasks on only a small part of the data, which means that there is no dependency between the operations of most of the agents working in parallel. Since there is no dependency between their operations, it is preferable to allow agents to perform their operations asynchronously, and can keep them working even if some agents stop or disconnect from the network. This independence of operation makes the possibility of encountering outdated or conflicting information less important than the restrictions that the system would have if a strong consistency model were used. In that case, it would be necessary to perform operations synchronously using, for example, a distributed lock manager (DLM). This solution would reduce performance to some extent (increased latency in read and write operations) and would loose tolerance to partitioning (a minimum number of active agents in the system is necessary) or availability (to ensure that the data is up to date it is necessary to do everything synchronously).

The use of Eventual Consistency adds some complexity to the system, as a strategy is needed to decide how the synchronization is going to be performed,

what information is sent, etc. Synchronization messages could include the operation performed to be replicated to the other agents, the complete status or the result of the operation. A strategy is also needed to resolve conflicts and agree on what is the latest update on the data. Another problem is that the solution is specific to the developed application, so its maintenance can be difficult if there are changes in the implementation or communication. This complexity can be reduced by using an existing tool such as CRDTs, which ensure convergence between agent states.

4.1.2. CRDT

Distributed systems experience different problems related to synchronization. This is because in most distributed systems there are resources that require some kind of control over their access or use. The most prominent cases are single access to non-concurrent resources (requiring mutual exclusion) and the ordering of events in non-sychronised environments. Depending on the characteristics of the problem, solutions may include the use of message passing systems to synchronise algorithms, the use of centralised systems for synchronisation, or in cases where full synchronisation is not necessary, the use of logical clocks, consensus algorithms or other techniques that provide consistency of operations.

Lamport logical clocks [11] are one of the most famous solutions for determining the order of events in a distributed system. Lamport defines the ”happened beforerelationship to define a partial order between events. An event A can happen before B, happen after B or happen neither before nor after, meaning they are concurrent. To determine this relationship, it is necessary to know at what point in time the event occurred. In the proposed solution, each process maintains a clock that assigns a number to each event in the system. This is usually implemented with counters, as system clock timestamps are not consistent between machines. Any timestamp can be used that meets the properties of: - For an event A occurring before a process B, both coming from process P, $C\langle A\rangle$ is

less than $C\langle B\rangle$. - For an event A sent from process P_i and an event B that is the receiving of the event A in P_j, $C_i\langle A\rangle$ is less than $C_j\langle B\rangle$. - A process P increments its clock between two successful events. - $C_i\langle A\rangle$ must be the timestamp value of process P_i at the time of the event. In addition, upon receipt of the event by P_j, it updates the value of C_j to a value greater than or equal to its current value and greater than the value of C_i.

To achieve a total order, only the partial order is necessary and to define an arbitrary ordering condition between the processes, so that the total order is determined when possible by the partial order (Counters) and when it is not possible (concurrent events), by the order of processes.

A state-of-the-art solution that ensures consistency between in distributed enviroments are CRDTs. CRDTs (Conflict-Free Replicated Data Types) are formally defined data types that ensure eventual strong consistency of information. These data types have the following characteristics:

1. Allow updates without synchronization.

2. Updates converge to a common state.

3. They do not depend on network availability, replication or system fault tolerance.

4. High availability and scalability.

5. They are formally defined.

There are three main ways to define a CRDT object. The first two are **state-based convergent CRDT objects** (state-based convergent CRDT) and operation-based commutative CRDT objects (operation-based commutative CRDT)[10]. The third type, which is the one used in DSR_d, are **objects based on state deltas** (delta-state CRDT)[12].

State-Based CRDT

To introduce state-based CRDTs it is first necessary to define state-based replicated objects. A state-based object is defined as a tuple (S, s^0, q, u, m) where S is the state of the object, s^0 is the initial state, q is a method to obtain the state content, u is a method to update the state content and m is an operation to integrate the state of another replica. In these objects an update is performed on a single replica and integrated by operation m on the others.

A state-based CRDT is formed by a triplet (S, M, Q) in which S is a **semilattice** with a **Least Upper Bound** (LUB) that has the properties of being commutative, idempotent and associative, M is a set of mutators that perform updates that create new states and Q is a set of functions that do not modify the content of the state and therefore do not generate new states.

Under the guarantee that there are no infinite loops in any of the replicas and that the information at some point is going to be received by the rest of the distributed objects and they are going to perform the integration operations, any state-based object in which the partial order property $(O, \leq)$ if the set of states S of the object forms a semilattice ordered by $\leq$, the state is monotonically non-decreasing (an update generates a state equal to or greater than the current state) and the merging of two states s and s' results in the least upper bound (LUB) of the two states.

For practical purposes this property can be obtained in a general way by adding to the state-based replicated object a monotonic counter that stores the current and previous values and is incremented on update operations. The partial order property $(\leq)$ is satisfied over the counter. It is also required that the merge operation on two states computes the least upper bound between the counter values of the two states.

Operation-Based CRDT

On the other hand, operation-based CRDTs are constructed with replicated operation-based objects. These objects are defined with the tuple (S, s^0, q, t, u, P). S, s^0 and q have the same meaning as in state-based objects. In this type of object the update is divided into two operations, a method t called prepare-update is introduced which has no effect on the state and a method u, called effect-update, which performs the update and is executed on each of the replicas. The last element is P, which represents the delivery precondition, which is a necessary condition for a replica to execute the u update. As long as the condition is not true, the execution of this will be delayed. As a result, it is necessary to ensure that the operations are sent in the order in which they were performed and that concurrent operations are commutative, since not in all communication protocols their order of arrival can be ensured. This precondition is automatically met if the operation has the property of being commutative.

4.1.3. Delta-State CRDT

A third type of CRDT, used in DSR_d, is the Delta-State CRDT. The differential feature between this CRDT and state-based CRDTs is the introduction of delta-mutators (m^δ), a function associated with an update operation that returns a delta-state. A δ-CRDT is defined as a triplet (S, M^δ, Q) in which S is again a semilattice with a least upper bound, M^δ is a set of delta-mutators and Q is a set of functions with no effect on the state. In this type of CRDT the transition between states is produced by performing the merge (join, LUB) of the current state s with the delta-state obtained by applying the delta-mutator on the current state $m^\delta(s)$. That the transition between states is performed by joining a delta-mutator with the current state implies that to distribute among the rest of replicas an update only the delta-state is necessary, while in a state-based CRDT the complete state is necessary to perform the operation; therefore, communications and operations are much more efficient.

4.1.4. MVReg

The selection of the CRDT in DSR_d has been done by looking for the most optimal object for the data structure representing the graph. Types that work as counters (GCounter, PNCounter, ...) or flags (EWFlag, DWFlag) are not compatible with what we need, types based on sets (AWORSet, RWORSet, ...) are varied but do not fit well with the representation used in the graph, unless you want to add a temporal dimension (even then it is not sure that it is the best solution). The most suitable types are those based on registers. Within this type, the one that fits the best is the Multi-Value Register (MVRegister, MVReg). The MVReg[13] represents a container for an object of the type to be stored. The access returns the most updated value in the causal history of the CRDT. In case of concurrent updates more than one object could be returned, although in the implementation used, a mechanism has been developed to avoid the user having to deal with object selection. Other register-based CRDTs are the RWLWWSet (Last-writer-wins set with remove wins bias) and the LWWReg (Last-writer-wins register). Both rely on a monotonic sequence generator to specify the order of the states, usually using a timestamp although in most systems timestamps do not satisfy this property. For this reason they have been discarded.

4.2. Performance

When designing the architecture of G, the performance requirements of the environments in which it will be executed have been taken into account. In an execution of DSR_d we find multiple agents that obtain information, either from a sensor or from G, transform it and insert it back into G. 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.

These requirements have conditioned the selection of some of the technologies used in DSR_d. This topic is discussed in depth in the Architectural Design and Implementation section.

4.3. Memory

The decisions about information representation, synchronization, and the agent-based architecture of CORTEX have implications on memory usage. Having each agent holding a replica of the complete state in memory has a large total memory cost ($M \cdot N$, where M is the total memory of the representation of G and N is the number of agents). This is justifiable considering the amount of RAM that most computers have today and the fact that the graph representation usually takes at most a few MB, although much more information could be stored if necessary. Moreover, it seems reasonable to accept higher memory usage in exchange for simplifying synchronization, information access and increasing 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.

4.4. Network

The network requirements that exist are directly related to the performance requirements and the synchronization model that is used. To satisfy the performance requirement we must use technologies that allow fast delivery of information 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 is achieved

both in the design, with the use of Delta-CRDTs and a suitable communication protocol, and in the implementation, with the information representation and communication libraries. For the second requirement, for the synchronization model, it is necessary to ensure that the messages sent are received by the agents. More details on the used protocol and its configuration can be found in the implementation section.

The communication protocol selected for development is **RTPS** (Real-Time Publish-Subscribe) over a reliable multicast communication using UDP. 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, etc.). 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.

4.5. Scalability

There are different factors that must be taken into account for the system to be scalable. The decisions taken and described in the previous sections are partly motivated by the need for system scalability.

Specifically, at the network level, we achieve scalability by using multicast for transport. This allows us to significantly reduce the load on the network by reducing the number of messages per update from $M-1$ in a point-to-point communication with M participants to only a single message with the use of 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, in scalability seen from the perspective of the tasks performed rather than the network, the use of an agent or component-based architecture allows the system to be much more distributable than in a monolithic or less adaptable architecture.

4.6. Robocomp integration

The last requirement is the integration with Robocomp tools, libraries and dependencies. **Robocomp** has a DSL-based code generation tool called RobocompDSL. The agents generated by this tool, in addition to being able to connect using DSR, must also be able to take advantage of the already existing communication architecture that uses ZeroC-Ice middleware. By fulfilling this requirement, the new agents will have two ”sides”: a pub/sub communication capability to connect with the DSR using reliable multicast, and a more general RPC and pub/sub capability, currently based on Ice but that can be extended to include other middlewares and protocolos such as MQTT, HTTP, WebSockets, Apache’s Thrift or Google’s Protocolo Buffers, to name just a few. Agents defined this way are very powerful processing units that share a highly organized data-structure while, at the same time, are able to communicate with external resources.

Capítulo 5

Architectural design

The DSR architecture implements a distributed data structure based on local copies that communicate to 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 node level, one at edge level and one at 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, while the synchronization process is performed asynchronously and hidden from the user, both at the time of obtaining the state of the connection and during the execution of the agent, 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 that other dynamic languages such as Python or JavaScript with NodeJS cannot achieve. Being also a compiled language and without garbage collection, you tend to get better performance than in interpreted languages such as Java. It also tends to have lower memory usage in object representation. Within

the category of typed, compiled languages with manual memory management, C++ is the one that offers the best library environment for development, although it offers fewer development guarantees than other languages such as Rust.

DSR is designed to be used in an agent-based architecture in which each agent executes a specific task or part of it. For an agent, the existence of other connected agents in the system is something invisible that they do not need to worry about. The reason behind this decision is to simplify program logic and to parallelize task execution as much as possible by taking advantage of the fact that each agent maintains a local copy of the graph. For this the implemented libraries must hide all possible details of synchronization, communication and concurrency. To simplify the compilation and development of agents, the libraries are compiled as shared libraries against which the agents are linked. The advantage of using a shared library instead of a static library is that the size of the binaries is reduced since the library is not included in the file.

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 general DSR API.

Figure 5.1 shows a schematic representation of the DSR architecture design. An agent manages a DSR instance, which is 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.

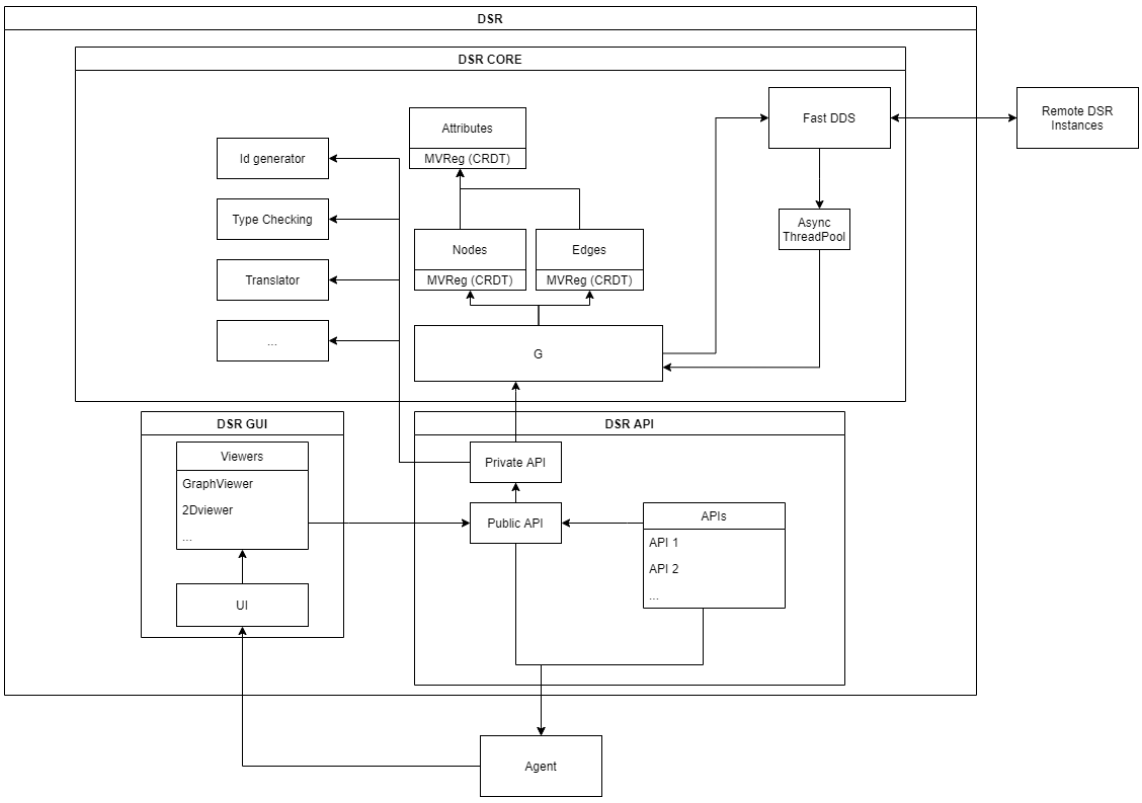


Figura 5.1: Simplified diagram of DSR architecture.

Capítulo 6

System design

In the following sections, different aspects of the system design and its implementation are presented. First, the development methodology followed and the different tests developed are discussed. Then, the C++ features used to provide a system less prone to programming errors and the dependencies selected in the development are shown. Finally, dividing the work into three sections, G, Agents and G core, the main aspects of DSR in each of the three groups are discussed. In the first one, G is shown at a high level. In the second one, the same is done with the agents. The third one shows at low level the most important implementation and design details of G.

6.1. Development methodology

The design and implementation has been done through an iterative and incremental process based on a cycle of definition of requirements and functionalities, implementation, search for errors and new requirements through testing and development of agents, correction of errors, optimisation and repetition of the process with the new needs found. Part of the definition of requirements, design and tests are inspired by the Test-Driven Development methodology. As the DSR core and the agents that use it are developed in parallel,

a way to validate 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 that are commonly used in this methodology, the agents being implemented are themselves are used as tests. In this particular case, 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 at a higher level. In the initial iterations, the aim was for the system to be reasonably stable and for communications to work correctly, while in an intermediate iteration the system is required to be robust, and in the last iterations the aim is for the APIs to be simple for users, providing a more intelligent code generation for the agents, etc.

The development cycle can be divided into three stages. In the first stage of development, which corresponds to the initial iterations, the basic functionalities that the system needs to work were defined. In this stage, work was performed on the validation and integration of the different technologies that had been selected in the design and on the implementation of the fundamental parts of the architecture such as the representation of the graph, the CRDTs, communications and a simple user interface. In parallel to this development, the tests in the design stages are used to validate the implementation and to detect errors. The use of agents that make use of the libraries for the testing allow both the performance and suitability of the API to be evaluated, as well as detecting new requirements and other errors in the implementation. The second stage of development corresponds to the intermediate iterations. The tasks in this stage focus mainly on error detection and correction, focusing on a robust core implementation, the most important task at this stage for the agents is the development of a basic API that limits the errors that users can make. The third stage, which we are currently in, focuses on the development of specific APIs that

implement actions that are commonly performed by agents.

The tools used for debugging errors throughout the project have been the debugger gdb [14] and the memory analyser valgrind[15]. For the evaluation of performance problems, the perf[16] tool was used.

6.2. Tests

Los tests desarrollados sobre DSR son de tres tipos. Los dos primeros tipos aparecen con el uso de una metodología que se inspira en la metodología Test-Driven Development. La motivación detrás de estos tests y su utilidad se comenta en la sección anterior. Los tests unitarios se implementan para comprobar el correcto funcionamiento de los métodos de las APIs. Están implementados con la librería catch2 y cubren principalmente la API base pública. Los segundos test son los propios agentes implementados durante en desarrollo de DSR. El tercer tipo de test, utilizados durante todo el proceso, son los tests de integración, en los que se comprueban las características de consistencia del sistema, la cual no es posible probar en test unitarios al ejecutarse en entornos distribuidos. Para estos tests definen agentes específicos para cada tipo de operación. Inserción y Borrado de nodos, Inserción y borrado de arcos. Inserción, modificación y borrado de atributos y una combinación de todas las operaciones. Estos tests se pueden ejecutar con diferentes frecuencias de publicación, diferentes niveles de concurrencia y con un número arbitrario de agentes. Al tratarse de tests distribuidos, la comprobación de resultados no es tan trivial como en los tests unitarios. En estos tests se utiliza la API Utils para guardar el estado de los ficheros de cada agente al terminar la ejecución, y se utilizan herramientas externas como jq para la comparación de los ficheros.

Otro tipo de tests, aún no implementados por completo, son tests de rendimiento que pueden ser utilizados para comprobar que los cambios en las clases internas de DSR o en la API no tengan un impacto muy negativo en el rendimiento de los agentes. Para estos tests se utiliza la librería benchmark de

google.

6.3. Detección de errores durante la compilación. Templates y C++20

Una decisión importante durante el desarrollo de cualquier librería es la elección de las técnicas de detección de errores que se van a utilizar. En este proyecto se ha han seguido dos premisas para tomar decisiones en este aspecto. La primera es que cuando antes se detecte un error, más sencillo será para el usuario depurar sus aplicaciones y detectar bugs. Si la detección se realiza durante la compilación, el programador ni siquiera necesitará hacer debug. La segunda es que las excepciones representan situaciones excepcionales en la lógica de la aplicación, y por lo tanto no deben utilizarse en situaciones que no se corresponden con esta situación. Por ejemplo, en vez de utilizar excepciones para el control de la posible inexistencia en los datos a los que se accede, se utiliza la clase **optional**, que representa precisamente ese comportamiento. Esto simplifica la depuración, al no haber una propagación del error como existe en las excepciones, ofrece más información al programador, al saber que una función puede devolver o no un valor y evita la mala práctica de capturar cualquier excepción sin importar si realmente es recuperable o no.

Para la detección de errores durante la compilación se utilizan las características añadidas al estándar de C++ desde su versión **C++11** hasta **C++20**, la más actual. Las principales herramientas utilizadas para estos controles son el uso de **templates** junto a **concepts**, expresiones **if-constexpr** y **traits**. Los templates permiten la definición de funciones y clases genéricas sobre los tipos o valores constantes que reciben como parámetro. La **metaprogramación** con templates es el punto de partida para la realización de estas comprobaciones gracias a que la sustitución de los tipos utilizados en los templates se realiza durante la compilación y esta sustitución debe ser

consistente. En C++ los traits son structs y clases genéricas con valores constantes y estáticos asociados a los tipos con los que son especializados. Los traits permiten obtener información sobre los tipos con los que se especializa un template. La información que puede obtenerse es muy variada y va desde los constructores que implementa a los calificadores del tipo (es un puntero, una referencia, qué tipo de referencia, etc.). Esta información puede utilizarse, por ejemplo, para modificar el comportamiento de la función genérica, habilitar el uso del template o denegarlo. Los **concepts**, añadidos en el estándar C++20, definen propiedades semánticas sobre los tipos genéricos que permiten validar los argumentos de los templates durante la compilación. Permiten obtener los mismos resultados que los traits mencionados anteriormente, pero con una sintaxis más clara y mejores mensajes de error. Un uso habitual es la definición los métodos que debe implementar un tipo para ser utilizado en una función genérica, de manera similar al uso de herencia para implementar una interfaz y la definición de una función que toma un objeto de la clase base, pero realizando la sustitución durante la compilación y sin el coste del acceso a una tabla virtual (vtable). Por último, las expresiones if-constexpr, permiten definir dentro de funciones genéricas diferentes comportamientos dependiendo del resultado de la evaluación de la expresión. Estas expresiones deben estar disponibles durante la compilación, por lo que deben ser un concept, un atributo marcado como constexpr o como constinit, o la invocación de un método marcado como **consteval** o constexpr. El uso de if-constexpr permite reducir el número de especializaciones que se realiza sobre una función genérica, simplificando el código. Otros elementos utilizados para la verificación durante la compilación es el uso de **static_assert**, que obliga a que una expresión booleana verificable durante la compilación deba evaluar a verdadero para que la sustitución del template sea correcta.

Un ejemplo práctico del uso de estas características es la definición y validación de los atributos en los nodos y arcos. En el fragmento de código 6.1 aparece la definición de los atributos. Los parámetros del template (#1)

son una variable de tipo string_view y una clase o struct de cualquier tipo. Cada atributo está compuesto por una variable booleana estática, definida como constexpr y cuyo valor es el resultado de la evaluación de la expresión #2. Esta expresión hace uso de los concepts y traits bool_constant de la librería estándar, allowed_types y unwrap_reference_wrapper_t, definidos en 6.2. El concept unwrap_reference_wrapper_t (#5, #6) se define utilizando un trait con una implementación genérica y una especialización para tipos que van encapsulados en un reference_wrapper. El concept allowed_types (#8) se define como el resultado de la evaluación del trait one_of (#7), que toma un número variable de parámetros, siendo el primero el tipo que se quiere evaluar y los n siguientes los tipos válidos. Utilizando una expresión de reducción, se comprueba si el trait is_same evalua a verdadero en alguno de los tipos. El segundo elemento del atributo es una variable constexpr estática de tipo string_view, que contiene la representación como cadena de texto del atributo (#3). El último elemento de un atributo es una variable estática con el tipo nativo que representa el atributo (#4), se utiliza para comprobar la validez de un tipo cuando se intenta acceder o modificar un atributo.

```
template<const std::string_view& n, typename Tn> // #1
struct Attr {
    static constexpr bool attr_type =
    ↪  std::bool_constant<allowed_types<unwrap_reference_wrapper_t<Tn>>>();
    ↪  // #2
    static constexpr std::string_view attr_name = std::string_view(n); // #3
    static Tn type; // #4
};
```

Source Code 6.1: Attribute definition.

```
template <typename T> struct unwrap_reference_wrapper { typedef T type;  };
```

```
template <typename T> struct
↪  unwrap_reference_wrapper<std::reference_wrapper<T>> { typedef T type; };
↪  //#5

template<typename T> using unwrap_reference_wrapper_t =
↪  unwrap_reference_wrapper<T>::type; //#6

template <typename T, typename ... Ts> struct one_of {
    static constexpr bool value = (std::is_same<std::remove_cvref_t<T>,
    ↪  Ts>::value || ...); //#7
};

template<typename T>
concept allowed_types = one_of<T , std::int32_t, std::uint32_t,
↪  std::uint64_t,
           std::string, std::float_t, std::double_t,
           std::vector<float_t>, std::vector<uint8_t>,
           bool, std::vector<uint64_t>, std::array<float, 2>,
           std::array<float, 3>, std::array<float, 4>,
           std::array<float, 6>>::value; //#8


template<typename Va>
concept any_node_or_edge = one_of<Va, DSR::CRDTNode, DSR::CRDTEdge,
↪  DSR::Node, DSR::Edge>::value; //#9

template<typename T>
concept is_attr_name = requires(T a) { //#10
  { std::is_same_v<decltype(T::attr_type), bool> };
  { std::is_same_v<decltype(T::attr_name), std::string_view> };
  { allowed_types<unwrap_reference_wrapper_t<decltype(T::attr_type)>> };
};
```

Source Code 6.2: Attribute definition.

El uso de estos concepts puede verse en la función get_attrib_by_name 6.3.

En el fragmento de código aparece la implementación incompleta del método utilizado para obtener la representación nativa del contenido de un atributo de G. La función se define como una función genérica que toma dos parámetros y tiene como restricción los concepts any_node_or_edge sobre el parámetros Type y el concept is_attr_name sobre el parámetro name (#11). Las restricciones aplicadas a las funciones mediante concepts se indican con la expresión requires. La función devuelve un objeto de tipo optional del tipo definido por el atributo type del parámetro name. Para obtener la definición del tipo se utiliza la palabra reservada decltype. En el concept is_attr_name (#10) se muestra otra sintaxis para definir un concept. Cada una de las expresiones en el requires debe estar definida o evaluada a verdadero si el resultado de la expresión es booleana. Las restricciones en este caso son que los atributos attr_type y attr_name tengan los tipos correctos y el tipo del atributto sea compatible con allowed_types. En la función también se utilizan expresiones if-constexpr (#12) para obtener el tipo nativo del variant que se utiliza en los atributos. Tras comprobar todas las alternativas con los constexpr se utiliza la alternativa final para impedir la compilación con un static_assert que evalúa a falso. La llegada a ese punto supone la existencia de un error de programación en alguno de los concepts, y puede darse cuando se añaden nuevos tipos a la representación nativa pero no se actualizan todas las funciones.

```
template <typename name, typename Type>
inline std::optional<decltype(name::type)> get_attrib_by_name(const Type &n)
    requires(any_node_or_edge<Type> and is_attr_name<name>) //#11
{
    using name_type =
    ↪  std::remove_reference_t<std::remove_cv_t<decltype(name::type)>>;

    auto &attrs = n.attrs();
    auto value = attrs.find(name::attr_name.data());
    if (value == attrs.end()) return {};

    const auto &av = value->second;
```

```
    if constexpr (std::is_same_v< name_type, float>) //#12
        return av.fl();
    else if constexpr (std::is_same_v< name_type, double>)
        return av.dob();
    else if constexpr (std::is_same_v< name_type,
    ↪  std::reference_wrapper<const std::string>>)
        return av.str();
    else if constexpr (std::is_same_v< name_type, std::int32_t>)
        return av.dec();
    ...
    else {
        []<bool flag = false>() { static_assert(flag, "Unreachable"); }();
        ↪  //#13
    }
}
```

Source Code 6.3: Putting it all together.

Por comodidad, la definición de los tipos se puede realizar con un macro 6.4 que define el string_view con el contenido del atributo, un alias del tipo attr con sus parámetros y registra la función en un mapa que se utiliza para las comprobación de tipos que no se pueden realizar durante la compilación. Esta situación se da por ejemplo cuando se itera sobre todos los atributos de un nodo o un arco.

```
template<typename tp>
static constexpr auto reg_fn = []() -> auto {
    if constexpr (is_reference_wrapper<tp>::value) {
        using tp_c = std::remove_const_t<typename tp::type>;
        return tp_c();
    } else {
        static_assert(std::is_constructible_v<tp>, "tp is not constructible
        ↪  without arguments, register your type manually");
        return tp();
    }
};
```

```
#define REGISTER_FN(x, it, stream)  \
    [[maybe_unused]] inline bool x ##_b =  attribute_types::register_type(
↪  x##_str, reg_fn<it>(), stream);     \

#define REGISTER_TYPE(x, ot, stream) \
    static constexpr auto    x ##_str = std::string_view(#x ); \
    using x##_att = Attr< x##_str, ot>;                         \
    REGISTER_FN(x, ot, stream) \

REGISTER_TYPE(level, int, false)
REGISTER_TYPE(pos_x, float, false)
```

Source Code 6.4: Definition of an Attribute.

6.4. Dependencies

The dependencies selected for the development have been chosen with the idea of maintaining compatibility with the previously existing Robocomp tools, which has led to the use of Qt. The new libraries have been selected by comparing among the different alternatives available and in general, only using external libraries where it is really necessary.

6.4.1. Qt5

The **Qt framework**[17] offers a large number of libraries for the development of C++ applications. For the development we only make use of the libraries for the creation of user interfaces for the agents, the libraries for the management of JSON files and the Qt language extension (MOC, Meta-Object Compiler), which enables the definition of signals that can be used to pass messages between objects.

In the graphical interfaces developed with Qt the communication between the user's actions and the different graphical elements is done by sending events.

These events are managed in an event loop that is implemented by default when using the QApplication class. The architecture of the libraries allows the development of applications with multiple programming models, being the preferred one the Model/View one, making the communication between both parts using signals. The Qt libraries include a large number of ready-to-use graphical elements and interface design tools.

The use of signals allows communication between instances of objects in which an instance can connect to signals emitted by an instance of another object. This message passing model has the following characteristics:

1. It is asynchronous. The object does not need to check whether there are signals to be attended or not.

2. It works as message passing, allowing it to operate in parallel architectures without the need to control access to data.

3. It allows different communication models. It is possible to define signals with a single receiver, with multiple receivers and set the execution at the time of signal emission or the event loop to handle it.

The code snippet 6.5 shows how signals are defined in a class. In order to define signals it is necessary that the object inherits from the QObject class and uses the QOBJECT macro. It is not necessary to implement the functions, since the Qt compiler extension is in charge of generating the necessary files for the connection. The signal emission is performed as indicated in 6.6. It can be seen that the syntax is not part of the C++ standard. This is because it is also converted by the Qt pre-compiler into a call to functions that it implements automatically. Finally, the connection to signals from an object can be seen in 6.7. The syntax is as follows, first the sender of the signal to be received, then the particular type of signal, third the receiver and finally the function to execute when the signal is received.

```
class DSRGraph : public QObject
{
Q_OBJECT
...
signals:
    void update_edge_signal(uint64_t from, uint64_t to,  const std::string
    ↪  &type);
    void update_edge_attr_signal(uint64_t from, uint64_t to, const
    ↪  std::vector<std::string>& att_name);
...
};
```

Source Code 6.5: Definition of signals in a c++ class.

```
emit update_edge_signal(node.id(), k.first, k.second);
```

Source Code 6.6: Sending a signal to connected receivers.

```
connect(G.get(), &DSR::DSRGraph::update_node_signal, this,
↪  &SpecificWorker::add_or_assign_node_slot);
connect(G.get(), &DSR::DSRGraph::update_edge_signal, this,
↪  &SpecificWorker::add_or_assign_edge_slot);
connect(G.get(), &DSR::DSRGraph::update_attrs_signal, this,
↪  &SpecificWorker::add_or_assign_attrs_slot);
```

Source Code 6.7: Connecting and object to a signal.

6.4.2. FastDDS

FastDDS, formerly known as FastRTPS, is a open implementation of the DDS (Data Distribution Service) standard. FastDDS[18] implements a Publish-Subscribe communication protocol oriented to reliable and scalable

real-time communications. An API is implemented over this protocol to make it highly configurable and flexible to different distributed system architectures.

Information is exchanged using a many-to-many unidirectional communication model (many publishers can write to many subscribers). The characteristics of this communication are established with QoS (Quality of Services) policies, which serve as a contract for the entities that manage publishers and subscribers to fulfill the communication requirements. DDS conceptually separates publishers from subscribers and groups and isolates all participants of a communication through an abstraction called Domain. Within the Domain, publishers and subscribers write and read respectively from specific topics, which are associated with a data type and have a specific QoS configuration.

At the communication level, some of the features that can be configured include:

- Real-Time. It is possible to specify time restrictions for message reception, validity time, time to wait for confirmation or even if confirmation is required.
- Publication mode. It is possible to configure a synchronous or asynchronous communication model.
- Communication reliability. Best Effort and Reliable communication modes are offered even in communications over UDP. These parameters change the behavior of both publishers (they have to save messages to resend them in case of loss) and subscribers (they have to send confirmation messages when they receive a message).
- Transport. It supports UDP and TCP in both IPv4 and IPv6 and a shared memory protocol (SHM).
- Flow control. Allows to limit the frequency of sending information.

The serialization and deserialization of information in FastDDS is configurable and only requires the implementation of a series of methods for the types to be shared in the topics. By default, the FastCDR library, developed by the same company, is used. FastCDR offers the serialization mechanism defined in the RTPS protocol specification. This library offers superior performance to widely used libraries such as Apache Thrift and Google Protocol Buffers[1].

6.4.3. Others

In addition to those already mentioned, some libraries are used for specific tasks. The library **OpenSceneGraph**[19] is used for the 3D representation of the representable elements of the graph. The **cppitertools**[20] library is used to simplify some operations on collections. This library provides most of the operations that can be performed by the Python library **itertools**. Finally, the **Eigen**[21] library is used to implement kinematic transformations and other operations on arrays.

6.5. G

En esta sección se introducen a alto nivel los elementos principales de G sin entrar en detalle en las tecnologías concretas utilizadas para la implementación de todas las funcionalidades que G realiza internamente. Se tratan principalmente las características del grafo (cómo se almacena la información), el acceso a la estructura de datos y las interfaces de usuario disponibles,

6.5.1. Grafo

La implementación del grafo puede reducirse a un mapa de nodos o vértices, en el que la clave es un identificador único. Cada nodo guarda un identificador, un nombre único, un tipo, un mapa de atributos y un mapa de aristas o arcos.

[1]https://www.eprosima.com/index.php/resources-all/performance/apache-thrift-vs-protocol-buffers-vs-fast-buffers

Se trata de un grafo dirigido, por lo que las aristas tienen dirección, siendo el origen el que almacena la información de la arista. Para conseguir conexiones bidireccionales entre nodos basta con crear las aristas en ambos sentidos. Un arco está formado por dos identificadores para el origen y el destino, un tipo y un mapa de atributos. También se almacenan mapas de índices sobre nodos y arcos 6.8 que permiten la optimización de algunas de las operaciones de G como el borrado de nodos, la obtención de todos los nodos de un tipo o la obtención de los arcos que apuntan a un nodo. Estos índices se actualizan en cada operación que se realiza sobre G. Actualmente los índices disponibles son un conjunto de identificadores de nodos borrados, dos mapas para obtener tanto el identificador de un nodo dado su nombre como el nombre dado su identificador, un mapa que guarda todos los tipos de arco que hay entre dos nodos, un mapa que guarda todos los arcos que apuntan a un nodo y dos mapas para guardar todos los nodos y arcos de un tipo. Para simplificar el mantenimiento, los índices guardan únicamente identificadores y tipos en lugar de punteros a los elementos del mapa. El coste de utilizar índices en la estructura en lugar de punteros es casi nulo (el uso de punteros requiere de una única desreferenciación, mientras que el índice en una estructura con coste de acceso mediante índice $O(1)$ es de un acceso y una desreferenciación).

Por el momento la implementación de las clases unordered_map y unordered_set utilizadas son las de la librería estándar al ser su rendimiento ha sido suficiente en los casos de uso que se han realizado, aunque se plantea la posibilidad de utilizar otro tipo de implementación que permita un acceso más eficiente de manera concurrente.

```
class DSRGraph
{
...
private:
std::unordered_map<uint64_t , mvreg<CRDTNode>> nodes;
...
////////////////////////////////////////////////////
```

```
// Cache maps
////////////////////////////////////////////////////
std::unordered_set<uint64_t> deleted;     // deleted nodes, used to avoid
↪  insertion after remove.
std::unordered_map<std::string, uint64_t> name_map;     // mapping between
↪  name and id of nodes.
std::unordered_map<uint64_t, std::string> id_map;       // mapping between
↪  id and name of nodes.
std::unordered_map<std::pair<uint64_t, uint64_t>,
↪  std::unordered_set<std::string>, hash_tuple> edges;      // collection
↪  with all graph edges. ((from, to), key)
std::unordered_map<uint64_t , std::unordered_set<std::pair<uint64_t,
↪  std::string>,hash_tuple>> to_edges;      // collection with all graph
↪  edges. (to, (from, key))
std::unordered_map<std::string, std::unordered_set<std::pair<uint64_t,
↪  uint64_t>, hash_tuple>> edgeType;  // collection with all edge types.
std::unordered_map<std::string, std::unordered_set<uint64_t>> nodeType;  //
↪  collection with all node types.
...
}
```

Source Code 6.8: Implementation of G.

Durante la primera iteración del desarrollo los atributos de los nodos y arcos se almacenaban y transmitían entre agentes en forma de texto y se transformaban al tipo que representaban cuando el usuario quería acceder a la información. Esto suponía un incremento en el uso de memoria y en el tiempo de acceso a los atributos, que debían ser transformados de un tipo string al tipo destino cuando el usuario accedía a ellos, y posteriormente debían ser convertidos a string cuando fuesen modificados de nuevo. Esto se apreciaba principalmente en los atributos que más memoria ocupan, como la imagen rgb de una cámara. Aunque este acercamiento permite la representación de cualquier tipo de información y puede verse en uso en algunas bases de dato NOSQL, es cierto que no hace un uso óptimo de los recursos. Posteriormente se reimplementó utilizando tipos nativos

que evitan las transformaciones entre strings y el tipo nativo y reducen el uso de memoria de manera notable. Los tipos soportados actualmente pueden verse en la tabla 6.1. Otra de las ventajas del uso de tipos nativos es la posibilidad de asignar de manera explícita el tipo que debe tener cada atributo. Esto permite comprobar los tipos en tiempo de compilación cuando sea posible y durante la ejecución cuando no lo sea, detectando errores que pudiesen darse en la transmisión o en la manipulación de la información.

Tipos en C++	Descripción
int32_t	Números enteros de 32 bits.
uint32_t	Números sin signo de 32 bits.
uint64_t	Números sin signo de 64 bits.
float	Números en coma flotante de 32 bits.
double	Números en coma flotante de 64 bits.
string	Cadenas de caracteres.
bool	Tipo de dato lógico.
vector<float>	Vector de tamaño variable de floats.
vector<uint8_t>	Vector de tamaño variable de números sin signo de 8 bits.
vector<uint64_t>	Vector de tamaño variable de números sin signo de 64 bits.
array<float, 2>	Array de floats de 2 elementos.
array<float, 3>	Array de floats de 3 elementos.
array<float, 4>	Array de floats de 4 elementos.
array<float, 6>	Array de floats de 6 elementos.

Tabla 6.1: Native types suported in attributes

6.5.2. APIs

Como se introduce en el apartado de Diseño de la arquitectura de DSR el acceso a la estructura de datos se realiza mediante una API thread-safe diseñada para ofrecer todas las funcionalidades mínimas para manipular toda la información almacenada. Las operaciones de ésta API son operaciones CRUD sobre el nodos, arcos y atributos. Ésta API básica se incluye en el objeto de G y realiza una gestión de todas las estructuras de datos y objetos internos (grafo, índices e instancia del framework de red).

A partir de la implementación de ésta se ofrecen otras APIs que permiten

la realización de tareas específicas. Las razones principales para no incluir todas las funcionalidades de DSR en una única API son; primero simplificar el mantenimiento y desarrollo al mantener conjuntos de operaciones reducidos y fáciles de manejar, sin necesidad de realizar un control de la concurrencia o de la red y segundo, simplificar el uso de DSR al ofrecer una división de las operaciones basadas en conceptos de más alto nivel, de modo que sea sencillo buscar qué operación utilizar para cada tarea. Si un usuario quiere hacer uso de un API específica únicamente debe instanciar un objeto y empezar a utilizarla.

Este modelo permite que la extensión de las funcionalidades de DSR sea simple y se integre de manera sencilla en la base de código (Solo es necesario añadir el fichero con la nueva API). Hasta el momento las APIs desarrolladas hasta el momento son dsr_agent_info, dsr_camera, dsr_inner_eigen, dsr_rt y dsr_utils. La información sobre los métodos ofrecidos en cada una de las APIs se encuentra en el Anexo B.

El acceso a la información de G se realiza mediante la obtención de copias locales que son modificadas y posteriormente resinsertadas en el grafo. El uso de copias locales permite que no sea necesario mantener bloqueos sobre elementos del mapa mientras se realizan operaciones, por lo que el sistema es más concurrente y el control de ésta es más sencillo. Además, para la API hace uso de las técnicas de detección de errores comentadas en uno de los apartados anteriores para detectar la mayor cantidad de errores posibles durante la compilación.

6.5.3. Python

Uno de los objetivos del desarrollo es la integración con Robocomp. El framework Robocomp permite el desarrollo de componentes interoperables en C++ y Python. Dado que el trabajo de implementación ya se ha llevado a cabo en la versión de C++, lo más conveniente es utilizar las propias librerías de DSR como módulos de extensión para Python. Por suerte, existen librerías que simplifican este trabajo como pybind11[22]. Con pybind11 puede definirse un

modulo de python con acceso a los tipos, miembros, métodos y funciones de objetos C++. En este caso se da acceso tanto a APIs similares a las de DSR y a los tipos de la representación de usuario de tal forma que se utilizan objetos C++ como si se tratase de tipos nativos de Python. El uso de módulos de extensión en Python tiene el beneficio de ofrecer más rendimiento que en una posible reimplementación en python, al poder saltarse el GIL (Global interpreter lock) y ejecutar código compilado, y además evitar tener que mantener varias versiones de la misma librería en diferentes lenguajes.

API Base. La API Base ofrece los siguientes métodos, todos ellos con el mismo comportamiento que su equivalente en C++.

Operaciones sobre Nodos.

- get_.
- delete_node.
- insert_node.
- update_node.

Operaciones sobre edges.

- get_edge.
- delete_edge.
- insert_or_assign_edge.

Métodos de conveniencia.

- get_nodes_by_type.
- get_edges_by_type.
- get_edges_to_id.

Los tipos disponibles en la API de Python son DSRGraph, Node, Edge y Attribute. El acceso a los métodos de DSRGraph para la modificación no son necesarios, ya que la naturaleza dinámica de Python permite el acceso al variant de los atributos de manera simple manteniendo las restricciones de tipado en los atributos. Pese a que el tipado dinámico de Python sea útil en la situación anterior, también introduce algunos problemas en la conversión de los tipos entre C++ y Python. El primer problema se encuentra en que en Python únicamente utiliza un tipo para cualquier tipo de enteros, sea cual sea su tamaño. Lo mismo sucede con los números en coma flotante. Esto hace que la transformación entre los diferentes tipos de C++ utilizados por el usuario (uint32_t, int32_t, uint64_t float, double e incluso bool) requiera de lógica adicional. El segundo problema encontrado es en la conversión de listas de Python a C++. Las listas de Python pueden almacenar objetos de cualquier tipo, por lo que para realizar una conversión segura es necesario comprobar el tipo de todos los elementos. Esto puede solucionarse al utilizar arrays de numpy, que tienen un tipo asociado y además utilizan la representación en memoria de objetos de C. El último problema es la conexión de funciones Python en las señales de Qt. Para solucionar este problema es necesario utilizar el módulo typing de Python y marcar con tipos la declaración de las funciones que van a utilizarse en las señales.

API RT. La API RT en el módulo de Python ofrece los mismos métodos que la API RT de C++.

InnerEigen API. La API InnerEigen en el módulo de Python ofrece los mismos métodos que la API InnerEigen de C++. Pybind11 soporta de manera nativa la librería Eigen convirtiendo de manera directa sus tipos a objetos ndarray de numpy.

Otros componentes de DSR como las interfaces de usuario no pueden ser expuestos de manera sencilla a Python ya que existen conflictos entre las librerías compartidas utilizadas por PySide2 (utilizada en Python) y las librerías compartidas de Qt en C++.

6.5.4. GUI

Las interfaces de usuario implementadas ofrecen diferentes representaciones para la visualización de la información de G. Se han implementado 4 representaciones que pueden ser utilizadas de manera opcional por los agentes. Además, los agentes pueden implementar sus propios widgets e integrarlos con los visores existentes. Las cuatro vistas implementadas son GraphViewer, 2D viewer, 3D osg viewer y tree viewer. Las interfaces de usuario hacen acceso a G mediante sus APIS públicas, permitiendo que no intervengan en el control de la concurrencia del grafo. Para no desaprovechar recursos, la interacción de los visores con G se produce mediante la recepción de las señales que envían los métodos de G por parte de los visores.

Graph Viewer. El visor Graph Viewer representa el contenido de G en forma de grafo. Cada nodo se representa como una esfera y cada arco con una línea que une dos nodos. La información de los nodos y arcos es accesible con vistas en forma de tabla, en la que se representan todos los atributos y visualizaciones específicas para arcos RT, láseres y nodos rgdb. Es la vista por defecto de G si se decide utilizar interfaz de usuario. En las figuras 6.1, 6.2 y 6.3 aparecen todas las representaciones de esta vista.

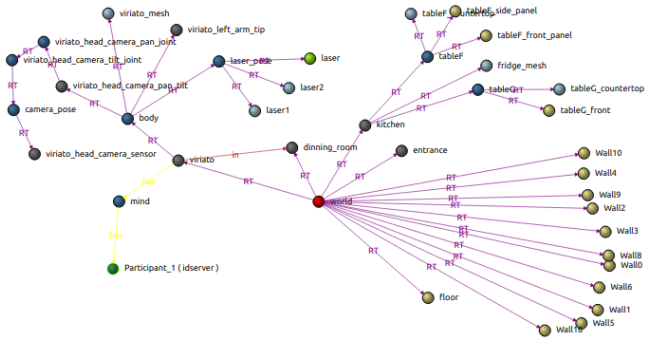
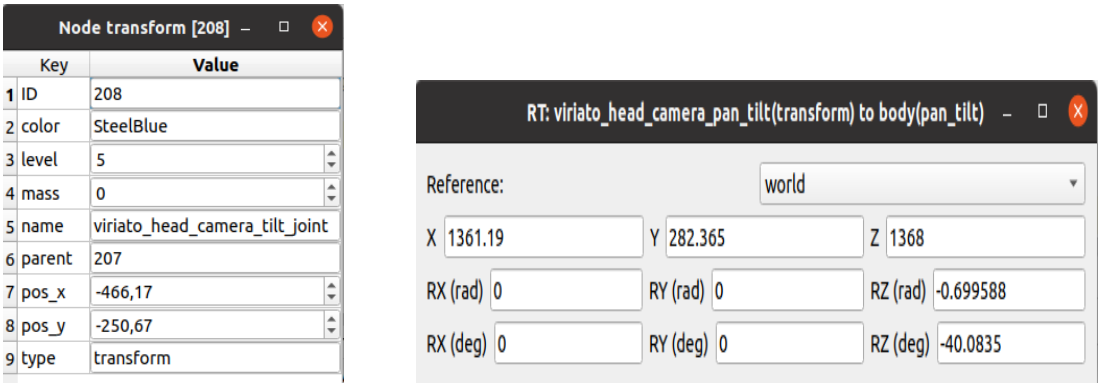


Figura 6.1: Graph View Representation



(a) Table View (b) RT edge View

Figura 6.2: On the left is the representation of the attributes of a node or an edge. On the right is the specific representation of an RT edge, where the values can be represented with respect to any other node in the RT tree.

(a) RGBD View (b) Laser View

Figura 6.3: On the left is the representation of rgb image and a depth image. On the right is the representation of a omnidirectional laser in the head of the robot

2D Viewer. Visor que representa la visualización cenital en 2 dimensiones de los objetos con representación física del mundo representado por el grafo. Este widget es utilizado por algunos agentes para representar la actividad que realizan, dibujando los datos del laser, del camino calculado del robot, etc. En la figura **??** aparece la representación 2D, a la que se le ha añadido un grid y un camino para el robot.

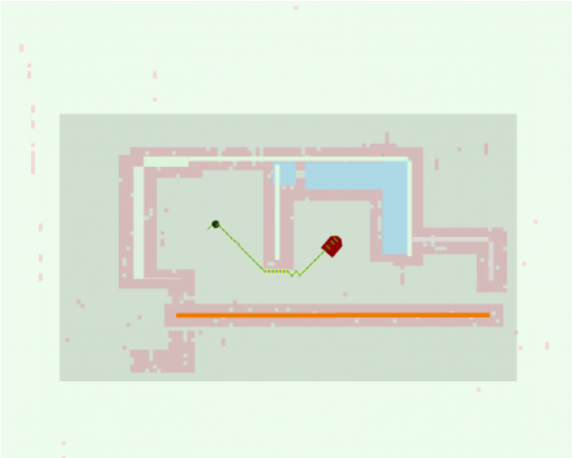


Figura 6.4: 2D View Representation with a grid representing the occupiable spaces in the área and a path to a point.

3D OSG Viewer. Visor que representa la visualización en 3 dimensiones de los objetos con representación física del mundo representado por el grafo, utiliza la librería OpenSceneGraph.

Tree Viewer. Widget que ofrece una representación de G en forma de árbol, utilizada para la consulta rápida de los atributos.

6.6. Agentes

Al estar basados en los componentes de Robocomp y ser generados por una herramienta de generación de código automática, los agentes que utilizan DSR siguen, en general, la misma estructura que los componentes de Robocomp. Actualmente se trabaja en hacer más modulares los agentes, haciendo que algunas de las dependencias de los agentes sean opcionales.

6.6.1. Arquitectura de los agentes

Los agentes, además de las librerías de DSR, requieren de las librerías de Qt5, el framework de comunicación ZeroC Ice (RPC) y opcionalmente IceStorm (Publish-Subscribe), también de los mismo desarrolladores que Ice. Hasta la llegada de DSR, los componentes de robocomp se comunicaban utilizando uno de

estos dos frameworks utilizando interfaces definidas y generadas por robocompdsl. Un agente DSR normalmente sigue una arquitectura como la planteada en la imagen 6.5. Los elementos principales de un agente son el bucle de eventos de Qt, que controla la ejecución de la interfaz de usuario, el procesamiento de eventos y el envío de señales, el SpecificWorker y si es necesario para el agente, la instancia de Ice. El SpecificWorker crea las instancias de DSR y DSRViewer y normalmente ejecuta de manera periódica una función compute, que está conectada mediante un timer al bucle de eventos de Qt. En esta señal se aprovecha para obtener datos de las interfaces de Ice, para comunicarse con componentes que no forman parte de DSR, hacer lecturas y modificaciones en G, etc. Como complemento al compute, los agentes se conectan a algunas de las señales de DSR, permitiendo reducir el consumo de CPU al trabajar únicamente cuando se producen cambios en G y simplificar la lógica al no ser necesario ir a buscar la información. Con el contenido de las señales un agente sabe si debe ejecutar alguna tarea o no.

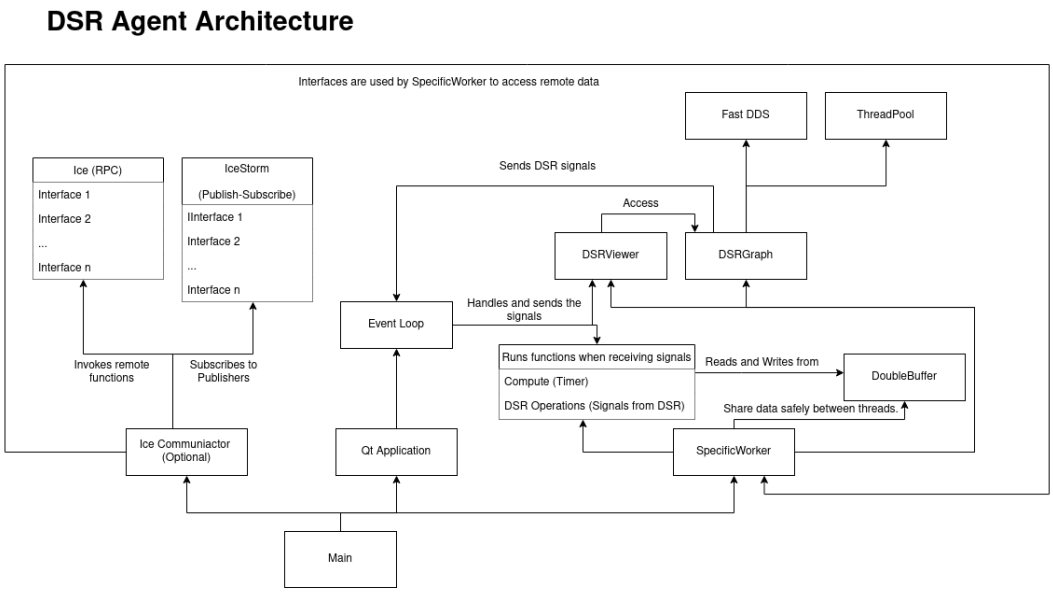


Figura 6.5: Typical agent architecture.

6.6.2. DoubleBuffer

La clase DoubleBuffer se utiliza para compartir de manera asíncrona datos entre threads. Ofrece un canal para múltiples productores y un único consumidor con capacidad para un único elemento en el que los productores pueden escribir sin esperar a que los datos sean leídos por el consumidor, a menos que se haya indicado una limitación en la frecuencia de publicación. El DoubleBuffer permite definir tipos diferentes para la escritura y la lectura con la condición de que o bien el tipo de lectura sea implícitamente convertible desde el tipo de escritura, o que los tipos de origen y destino sean iterables y el tipo del iterador sea implícitamente convertible, o que se proporcione una función de conversión entre ambos tipos. Los métodos princiaples de acceso a DoubleBuffer aparecen en el fragmento de código 6.9. La lectura de del DoubleBuffer se puede realizar mediante dos métodos (#1 y #3). El primero es una versión bloqueante en la que si no hay datos disponibles se espera un tiempo máximo de milisegundos indicado en el parámetro de la invocación de la función, por defecto 200ms. En caso de superar el tiempo máximo se lanza una excepción (#2). La función devuelve el contenido del buffer e indica que ha sido consumido. La segunda versión devuelve un optional con el cotenido del buffer, sin realizar ningún tipo de espera (#4). Ambas funciones toman un lock único sobre el buffer. La función de escritura (#5), toma un rvalue como entrada y opcionalmente una función si se quiere realizar una conversión entre tipos. Si no hay un limitador en la frecuencia de publicación o se está publicando con la frecuencia adecuada (#6), se lanza de manera asíncrona (#7) la ejecución del intercambio de valores en el buffer de lectura por el de escritura, ejecutando si es necesario la función de conversión (#8). Una vez realiza la posible conversión, se intercambian los buffers de lectura y escritura (#9) y se notifica a los lectores que pudiesen estar esperando para leer (#10). Actualmente la ejecución de la función se ejecuta independientemente de si posteriormente se consume o no. Esta solución permite que tanto escritor como lector no necesiten bloquearse ejecutando la conversión, pero podría no ser la más adecuada en todas las situaciones. Se plantea

para un desarrollo futuro la posibilidad de ofrecer una versión lazy en la que la conversión se realice en el momento de la demanda. Esta versión tendría un mayor coste computacional en la lectura, pero puede ser conveniente en situaciones en las que no está asegurada la lectura de buffer y la conversión es costosa.

```
template <class I, class O>
class DoubleBuffer
{
...
public:

    O get(std::chrono::milliseconds t = 200ms ) {//#1
        std::shared_lock lock(bufferMutex);
        if (!cv.wait_until(bufferMutex, td::chrono::steady_clock::now() + t
        ↪  ,
           [this]() { return !empty.load();})){  throw
           ↪  std::runtime_error("Timeout");} //#2
        empty.store(true);
        return readBuffer;
    }

   std::optional<O> try_get(){ //#3
        if (empty.load()) return {}; //#4
        std::shared_lock lock(bufferMutex);
        empty.store(true);
        return readBuffer;
    }

    bool put(I &&d, std::function<void(I &&, O &)> t = empty_fn){ //#5
        auto now = std::chrono::steady_clock::now();
        if (std::chrono::duration_cast<std::chrono::microseconds>(now -
        ↪  last_write)  > write_freq) { //#6
            last_write = now;
            worker.spawn_task([&, this, d = std::move(d), t =
            ↪  std::move(t)]() mutable { //#7
```

```
                O temp;
                if (this->ItoO(std::move(d), temp, t)) //#8
                {
                    std::unique_lock lock(this->bufferMutex);
                    this->writeBuffer = std::move(temp);
                    std::swap(writeBuffer, readBuffer); //#9
                    empty.store(false);
                    cv.notify_all(); //#10
                }
            });
            return true;
        } else  { return false; };
    }
    ...
}
```

Source Code 6.9: DoubleBuffer Code.

6.6.3. Señales

Las señales de Qt emitidas por G son la herramienta más habitual en los agentes de DSR para implementar el comportamiento ante cambios en la información del grafo. Las señales de G se emiten en las siguiente situaciones:

- **update_node_signal:** Se envía cuando se crea un nodo en el grafo. Incluye la información del identificador del nodo y su tipo.
- **update_node_attr_signal:** Se envía cuando se modifican los atributos de nodo. Incluye la información del identificador del nodo y una lista con los nombres de todos los atributos modificados.
- **update_edge_signal:** Se envía cuando se crea un nuevo arco entre dos nodos. Incluye la información del identificador del nodo origen, el identificador del nodo destino y el tipo del arco.

- **update_edge_attr_signal:** Se envía cuando se modifican los atributos de un arco. Incluye la información del identificador del nodo origen, el identificador del nodo destino, el tipo del arco y una lista con los nombres de todos los nodos modificados.
- **del_edge_signal:** Se envía cuando se elimina un arco. Incluye la información del identificador del nodo origen, el identificador del nodo destino y el tipo del arco.
- **del_node_signal:** Se envía cuando se elimina un nodo. Incluye la información del identificador del nodo.

Un ejemplo del uso de las señales en un agente real puede verse en los bloques de código 6.10 y 6.11. Los agentes se conectan a las señales que van a utilizar en la función initialize, que se ejecuta tras la creación del objeto SpecificWorker del agente. La conexión se realiza con la invocación de la función connect (#1) y enlaza la recepción de una señal de tipo update_node_attr_signal con la ejecución de la función change_attrs_slot. Al ser una conexión encolada (QueuedConnection), el bucle de eventos de Qt se encarga de la ejecución. En muchas ocasiones se utiliza la clase DoubleBuffer junto a este patrón para el intercambio de información entre hilos de ejecución.

```
void SpecificWorker::initialize(int period) {
    G = std::make_shared<DSR::DSRGraph>(0, agent_name, agent_id);
    ...
    connect(G.get(), &DSR::DSRGraph::update_node_attr_signal, this,
    ↪  &SpecificWorker::change_attrs_slot, Qt::QueuedConnection); //#1
    ...
}
```

Source Code 6.10: Connection to Signal in an agent.

Las funciones conectadas a las señales suelen tener la estructura de la función change_attrs_slot. En primer lugar se comprueba si entre los elementos enviados

en la señal en el vector de atributos modificados se encuentra el atributo con el que se quiere trabajar (#2). Esta comprobación puede tener otras restricciones, como por ejemplo comprobar el identificador del nodo para el que se envía la señal sea uno en concreto. Si la condición se cumple normalmente se obtiene el atributo modificado (#3) y se realiza algún tipo de operación con él.

```
void SpecificWorker::change_attrs_slot(std::uint64_t id, const
↪  std::vector<std::string>& att_names) {
    if(id == 206 && const auto node = G->get_node(206); node.has_value())
        if (auto att_name = std::ranges::find(att_names,
        ↪  "viriato_head_pan_tilt_nose_pos_ref"); att_name !=
        ↪  std::end(att_names)) // #2
        {
            std::vector<float> target =
            ↪  G->get_attrib_by_name<viriato_head_pan_tilt_nose_pos_ref_att>
            (node.value()).value().get(); // #3
            ...
        }
}
```

Source Code 6.11: Code executed when a signal is invoked.

6.6.4. Generación de Código

El generador Robocompdsl permite generar código para agentes en C++ y Python capaces de conectarse entre ellos y hacer uso de las interfaces de los componentes previamente existentes de Robocomp. El objetivo a corto plazo es hacer la generación de código más inteligente, pudiendo definir y generar la lógica del agente en el compute y las señales, indicando sobre qué tipos de nodo y arcos quiere escuchar señales, qué atributos son interesantes para él, etc. Esto reduce el tiempo necesario para empezar a desarrollar agentes y reduce los posibles errores de programación al tratarse de código que está probado con anterioridad.

6.7. G Core

En esta sección se expone las tecnologías e implementaciones de los elementos sobre los que está construido G. En los diferentes apartados se tratan tecnologías de más bajo nivel que los mostrados en las secciones de G y Agentes. Los temas tratados son la implementación del CRDT, el flujo de información de los datos y sus diferentes representaciones, la configuración e integración de FastDDS, el uso de estructuras auxiliares para la mejora del rendimiento y algunas optimizaciones implementadas.

6.7.1. Representación de la información

En lo referente a la representación de la información se manejan tres representaciones diferentes. Una para la comunicación, parcialmente generada por un generador de código y que está definida en un fichero **IDL** (interface definition language), una segunda que es la utilizada por G para el almacenamiento y que hace uso de los tipos CRDT y una tercera en la que se ocultan estos tipos al usuario y que es la utilizada por la API de acceso a G. Un diagrama esquemático de las tres representaciones puede verse en la figura 6.6. Tanto las clases de usuario como las clases CRDT comparten la misma representación de los atributos y son bastante similares en el resto de atributos, siendo la única diferencia el uso de la clase MVReg en las clases CRDTNode y CRDTEdge y sobre la clase Atributo. Las clases generadas por el IDL pretenden replicar la representación CRDT manteniendo la compatibilidad con el framework de transporte. Puede verse como hay cierta redundancia en clases como MvregNode, MvregNodeAttr, MvregEdge y MvregEdgeAttr. Esto se debe a que el generador de código no soporta la creación de clases con tipos genéricos. El generador tampoco soporta algunas clases de conveniencia que se utilizan en las otras representaciones como la clase pair. Se plantea en el futuro la eliminación de esta representación, ya que es posible implementar de manera manual los métodos necesarios para el uso de RTPS con la representación interna de G.

Las operaciones de transformación entre los diferentes tipos se han diseñado para que sean lo más eficiente posible. Al ser los objetos IDL objetos temporales que únicamente se utilizan en el momento de su recepción por la red, la conversión entre estos y objetos CRDT se realiza siempre mediante la operación move, permitiendo que no sea necesario realizar copias costosas. La operación inversa se realiza mediante una copia casi exacta de la información, añadiendo algunos campos donde se necesita más información del contexto. La transformación entre CRDT y tipos de usuario se realiza mediante una copia siempre. Como se verá más adelante el programador trabaja con copias locales desconectadas del grafo que debe reinsertar cuando termine de realizar modificaciones, permitiendo que el grafo pueda mantenerse siempre actualizado y accesible. Por otra parte, los objetos de la representación Usuario pueden ser movidos o copiados, según interese al programador.

As stated in the architecture design, the development is divided into three shared libraries, **dsr_core**, **dsr_api** and **dsr_gui**. dsr_core includes the functions, classes, structures, traits and concepts needed to implement DSR. dsr_api includes all the DSR user APIs. dsr_gui includes the viewers and classes to extend the DSR user interface.

6.7.2. Networking

La comunicación entre agentes está dividida en tres elementos, participantes, publicadores y suscriptores. Los participantes (Participant) definen un dominio en el que existen publicadores y suscriptores que publican en Topics concretos. Los publicadores (Publishers) escriben información en un Topic, que tiene asociado un tipo de datos. Los suscriptores (Subscribers) reciben los mensajes de un Topic. Para los tres elementos se ha implementado una clase que gestiona la configuración y tiempo de vida.

La clase Participant encapsula la clase participant de FastDDS y se encarga de la creación de los topics, la selección del transporte utilizado en la comunicación,

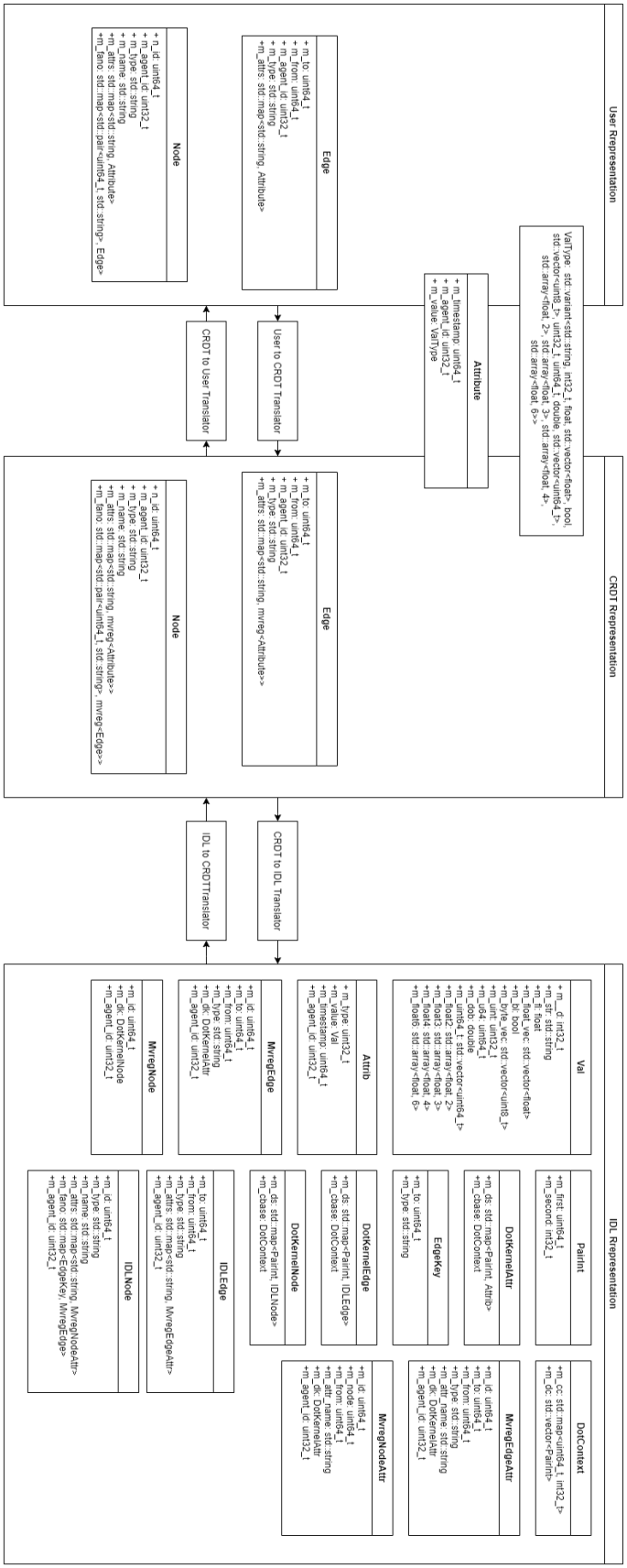


Figura 6.6: Diagram of type representations

la configuración de los tamaños de los buffers del transporte, el uso de interfaces de red, la selección del protocolo de descubrimiento de otros participantes, la gestión de los objetos **subscriber**, **reader**, **publisher** y **writer** que utilizan las clases Publihser y Subscriber y la detección de la conexión y desconexión de participantes, que se utiliza para controlar que el identificador de los agentes es único. Los parámetros de configuración que no están en sus valores por defecto pueden verse en la tabla 6.2.

Parámetro de confguración	Valor	Notas
Transporte	UDPv4.	
Tamaño del Buffer de envío	33554432 Bytes (32MB)	
Tamaño del Buffer de recepción	33554432 Bytes (32MB)	
Tamaño máximo del mensaje	65000 Bytes	Limitado por el propio estándar, mensajes más grandes son fragmentados.
Interfaces de red permitidas	127.0.0.1	
Filtros	FILTER_SAME_PROCESS	Ignora mensajes que vengan del mismo proceso.
Tiempo de anuncio	3s	Tiempo que el participante se anuncia en la red para establecer la conexión con otros participantes.
Tiempo de validez	6s	Tiempo que se considera a otros participantes como válidos sin dar señales de viveza.

Tabla 6.2: DDS Participant Configuration.

La clase **Subscriber** encapsula la clase subscriber y la clase reader de FastDDS y tiene como tarea ejecutar un callback cada vez que se recibe un mensaje. Para esto la clase subscriber de FastDDS permite la creación de un objeto listener que puede implementar el comportamiento deseado ante difrentes eventos. En este caso el único que es interesante para nosotros es el método **on_data_available**. La clase reader de FastDDS se encarga a bajo nivel de la comunicación, la confirmación de los mensajes, la lectura de los buffers, el control del historial, etc. Para obtener un comportamiento en el que no se pierdan mensajes en la comunicación es necesario configurar algunas de las políticas de calidad (**Qos**) de esta clase reader. Las propiedades a configurar son la fiabilidad de la comunicación, el historial de recepción, el límite de recursos del historial, el modelo de duración de los mensajes, el modelo de memoria utilizado para la recepción y almacenamiento y la latencia máxima para aceptar un mensaje. En la tabla 6.3 puede verse la configuración habitual. Para mejorar el rendimiento en

situaciones en la que se envían mensajes de gran tamaño con mucha frecuencia también está disponible una configuración alternativa que ofrece menos fiabilidad 6.4.

Parámetro de confguración	Valor	Notas
Fiabilidad	RELIABLE_RELIABILITY_QOS	
Historial	KEEP_ALL_HISTORY_QOS	Se guardan en el historial todos los mensajes hasta que son confirmados por todos los agentes u otro parámetro de configuración lo descarta.
Número de elementos	400	Número máximo de elementos en el historial.
Número de elementos en memoria	400	Número máximo de elementos en memoria.
Durabilidad	VOLATILE_DURABILITY_QOS	Mensajes anteriores a la conexión del agente son ignorados.
Política de memoria del historial	DYNAMIC_REUSABLE_MEMORY_MODE	Se reserva memoria a necesidad, puede tener un impacto en el rendimiento en los primeros momentos de la comunicación.
Latencia máxima válida	50ms	
Endpoint locator	Multicast UDPv4	

Tabla 6.3: DDS Reader Configuration.

Parámetro de confguración	Valor	Notas
Fiabilidad	BEST_EFFORT_RELIABILITY_QOS	
Historial	KEEP_LAST_HISTORY_QOS	Se mantienen en el historial los últimos N mensajes.
Número de elementos	50	Número a mantener en el historial.
Confirmación de recepción	No	No se envían Acks cuando se recibe un mensaje

Tabla 6.4: DDS Reader Configuration for streaming messages.

La clase **Publisher** encapsula las clases publisher y writer de FastDDS. Como en el caso anterior, la clase publisher ofrece una interfaz de alto nivel, mientras que la clase writer se encarga del envío, la fragmentación, el reenvío, etc. De la clase publisher no se ha implementado ninguno de los callbacks disponibles. Las propiedades configurables del publicador son la fiabilidad de la comunicación, el historial de recepción, el límite de recursos del historial, el modelo de duración de los mensajes, el modelo de memoria utilizado para la recepción y almacenamiento, el modelo de publicación de mensajes, y otros parámetros de control. En la tabla 6.5 puede verse la configuración habitual para un writer. La configuración compatible con suscriptores que envían mensajes en streaming es la de la tabla 6.6.

Cada uno de los Topics definidos tiene asociado un único tipo. Para poder crear un Topic, debe implementarse una clase que herede de la clase **TopicDataType** e implemente los métodos serialize, deserialize, getSerializedSizeProvider, getKey,

Parámetro de confguración	Valor	Notas
Fiabilidad	RELIABLE_RELIABILITY_QOS	
Historial	KEEP_ALL_HISTORY_QOS	Se guardan en el historial todos los mensajes hasta que son confirmados por todos los agentes u otro parámetro de configuración lo descarta.
Número de elementos	300	Número máximo de elementos en el historial.
Número de elementos en memoria	300	Número máximo de elementos en memoria.
Durabilidad	VOLATILE_DURABILITY_QOS	No es necesario enviar los mensajes del historial a agentes que no estaban conectados en el momento del envío.
Política de memoria del historial	DYNAMIC_REUSABLE_MEMORY_MODE	Se reserva memoria a necesidad, puede tener un impacto en el rendimiento en los primeros momentos de la comunicación.
Modo de publicación	ASYNCHRONOUS_PUBLISH_MODE	Los mensajes pueden enviarse y confirmarse de manera asíncrona.
Periodo de comprobación de Acks	20ms	
Valided del mensaje	1s	Tiempo máximo que un mensaje puede mantenerse en el historial del writer aunque no haya sido confirmado.
Endpoint locator	Multicast UDPv4	

Tabla 6.5: DDS Writer Configuration.

Parámetro de confguración	Valor	Notas
Fiabilidad	BEST_EFFORT_RELIABILITY_QOS	
Historial	KEEP_LAST_HISTORY_QOS	Se mantienen en el historial los últimos N mensajes.
Número de elementos	50	Número a mantener en el historial.
Confirmación de recepción	No	No se espera la recepción de Acks.

Tabla 6.6: DDS Writer Configuration for streaming messages.

createData y deleteData. La implementación de estos métodos es automática al utilizar el generador de códifo de FastDDS. En la tabla 6.7 se indican los topics existentes y los tipos de datos que se envían por ellos. Los topics DSR_NODE y DSR_EDGE envían mensajes en los momentos de creación y borrado de nodos y arcos. Como se comentó en el apartado sobre la representación, se utiliza la representación IDL para el envío de mensajes. Los Topics DSR_NODE_ATTS y DSR_EDGE_ATTS se utilizan para el envío de modificaciones en los atributos de los nodos y los arcos. El tipo IDL con el que están asociados en MvregNodeAttr y MvregEdgeAttr, aunque se envían en vectores de estos tipos, al haber comprobado durante el desarrollo que las actualizaciones habitualmente implican la modificación de múltiples atributos a la vez y la reducción de la frecuencia de mensajes favorece la fiabilidad de la comunicación. El Topic GRAPH_REQUEST únicamente se utiliza en el momento de conexión a la red para solicitar el estado actual del grafo. Una vez recibido el escritor se desconecta de este topic. La información transmitida es el identificador único del agente que se conecta. El último topic, GRAPH_ANSWER, se utiliza para el envío del grafo completo.

Una vez recibido el suscriptor se desconecta de este topic.

Topic	Tipo
DSR_NODE	IDL::MvregNode
DSR_EDGE	IDL::MvregEdge
DSR_NODE_ATTS	IDL::MvregNodeAttrVec
DSR_EDGE_ATTS	IDL::MvregEdgeAttrVec
GRAPH_REQUEST	IDL::GraphRequest
GRAPH_ANSWER	IDL::OrMap

Tabla 6.7: DDS Topics ans Types.

La limitación principal en la disponibilidad se debe a que es necesario que un único agente se encargue de la sincronización del envío del estado actual del grafo cuando se conecta un nuevo agente. Anteriormente también nos encontrábamos con el problema de generar identificadores únicos para los nodos del grafo, que requería también de un generador de identificadores centralizado. Para cumplir con estas dos labores se crea un agente llamado idserver, que debe estar en ejecución para añadir agentes al sistema. Actualmente este agente únicamente se encarga de la sincronización de nuevos agentes, ya que la generación de identificadores se ha descentralizado gracias a una implementación basada en los generadores de identificadores **sonyflake**[23] de sony y snowflake de twitter. El funcionamiento del generador de ids únicamente requiere de que el identificador de los agentes sea único, lo cual es comprobado en el momento de la conexión al sistema.

Cada identificador se representa con un número sin signo de 64 bits (uint64_t en C++). De los 64 bits se reservan 40 para una unidad de tiempo que representa un periodo de 10 milisegundos desde la fecha 2021-01-01 00:00:00 GTM, 12 bits para un contador que permite diferenciar entre fechas generadas en el mismo periodo y 12 bits para el identificador único del agente. Este generador permite que existan 4096 agentes, que generen en cada periodo de 10 milisegundos 4096 identificadores cada uno sin posibilidad de que existan colisiones. El uso de 40 bits para los timestamp permite generar identificadores durante más de 374 años

desde la fecha de inicio. El generador de identificadores ofrece una API thread safe para hacer posible su uso desde varios hilos. En el fragmento de código 6.12 se puede observar el algoritmo. En primer lugar (#1) se obtiene un instante de tiempo en nanosegundos, se convierte a un periodo de 10 milisegundos y se ajusta al timestamp de inicio. Si estamos en un periodo de tiempo más nuevo que el anterior se genera un id nuevo con contador a 0 (#2). Si el tiempo actual es igual, se incrementa el contador y se le aplica la máscara de 12 bits para comprobar overflows (#3). Después de incrementar el contador se comprueba si se ha llegado al límite del contador. En caso de que así sea (#4) se incrementa el tiempo transcurrido, se calcula el tiempo restante hasta el siguiente periodo y se espera hasta ese momento. Por último se construye el id aplicando las máscaras de bits a cada elemento (#5).

```
uint64_t id_generator::generate() {
    constexpr uint16_t mask_counter = (1 << counter_size) - 1;
    std::unique_lock<id_generator::mtx_type> lock(mtx);
    uint64_t current = get_time_point().time_since_epoch().count() /
    ↪  time_unit - start_time / time_unit; //#1
    if (elapsed_time < current) { // #2
        elapsed_time = current;
        counter = 0;
    } else {
        counter = (counter + 1) & mask_counter; // #3
        if (counter == 0) { // #4
            elapsed_time += 1;
            auto overtime = elapsed_time - current;
            auto tp =
                std::chrono::system_clock::time_point{
                    std::chrono::milliseconds{overtime * 10}} -
                    std::chrono::system_clock::time_point {
                        std::chrono::nanoseconds{
                         std::chrono::time_point_cast
                         <std::chrono::nanoseconds>
```

```
                        (std::chrono::system_clock::now())
                        .time_since_epoch().count()
                        % (int64_t)time_unit}};
            std::this_thread::sleep_for(tp);
        }
    }
    if (elapsed_time >= (static_cast<uint64_t>(1) << time_size)){
        throw std::logic_error("time is over the limit, change start
        ↪  timestamp");
    }
    return (static_cast<uint64_t>(elapsed_time) << (agent_id_size +
    ↪  counter_size)) |
           (static_cast<uint64_t>(counter) << agent_id_size) |
           static_cast<uint64_t>(agent_id); //#5
}
```

Source Code 6.12: Id generation.

Volviendo al problema de añadir agentes al sistema. Actualmente el rol de idserver puede ser reemplazado de manera sencilla en ejecuciones realizadas en entornos virtuales y físicos en los que sobre el mundo interactúa un único agente de manera directa. Esta es la arquitectura más común para la ejecución de DSR en un robot. En estas situaciones el rol de idserver puede ser tomado por el agente específico que se encarga de la tarea de comunicarse con el simulador o el robot. En este tipo de escenario no tiene mucho sentido mantener la disponibilidad en el acceso al sistema si el agente encargado de la interacción con el robot se desconecta. Para situaciones diferentes existen varias opciones. El requisito para poder servir el grafo es que no se estén ignorando atributos del grafo con los filtros disponibles para los agentes.

1. La primera opción consiste en que todos los agentes capaces de servir peticiones de sincronización inicial atiendan las peticiones. El problema de esta solución es que enviar el grafo completo es costoso tanto para el agente que lo envía como para la red. En el agente que se conecta, sería necesario

aceptar únicamente el primer mensaje recibido.

2. La segunda opción consiste en identificar a todos los agentes que pueden servir el estado actual a los nuevos agentes y realizar un consenso entre ellos para decidir quién debe encargarse de hacerlo. Puede utilizarse el identificador único de los agentes y la información ofrecida por DDS sobre participantes conectados para esta tarea. Esta solución tiene menos impacto por en la red que la anterior.

Para ambas soluciones sería conveniente que los agentes pudiesen identificarse como no capaces de realizar llevar a cabo la tarea de sincronización si existen restricciones de rendimiento con las que deben cumplir.

6.7.3. MvReg

The implementation of the CRDT MVReg has been done using as a basis the reference implementation of its original author[2]. The original version, although correct, is not optimal since it is a very generic implementation that does not take advantage of the knowledge about the domain in which it is used. The main problems of the original implementation are the increase in memory usage for each operation performed in the MVreg, the unnecessary copying of different operations and the impossibility of using the move operation in write and join operations, which involves copying objects that may be large (nodes, edges). The increase in memory usage is because in the original implementation all the historical elements of the causal context and the dot cloud of the dotcontext class and the dot map of the dotkernel class are maintained. These structures store the information of previous states (increasing monotonic counters) and previous values stored in the CRDT. This makes sense in set-based CRDTs, where receiving and merging a state may involve adding new information to the object even if the state is not the most updated one, but this is not the case in the MVReg, which is register-based

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

and only keeps a useful value, so it should be possible to keep only the most updated value for the state and the counter (as the last update prevails, older counter states can be considered as already included in the set by the property <of the current value of the counter, required by the CRDT object definition). This is possible because in our use case there is no requirement for causal consistency as neither the order of the messages nor the content of the states is relevant for the MVreg current state. As part of this work, changes have been made to reduce the memory usage and the computational cost of data access and join operation of the CRDT. This improvement has been achieved by making both write and join operations keep at most one element in the dot kernel and a maximum of a single element in both the causal context and the dot cloud. As explained above, this is possible given the characteristics of the MVReg, in which newer writes prevail over older writes, even when an older write arrives later. The same applies for the causal context. If there exists in the causal context an element with a higher counter than the received one, we consider that this element is already part of the causal context and the join does not modify the current state. It has also been necessary to establish a method for automatic conflict resolution. A technique similar to obtaining total order in Lamport's logic clocks by arbitrarily defining an order between processes is used to perform the conflict resolution.

All these changes allow the memory usage not to grow during execution (In the original version the memory usage grows unreasonably in attributes that are frequently updated) and the access to the MVReg content to be optimal (For the previous reason, copying the content of an MVReg with a very large history was expensive).

The 6.7, 6.8 and 6.9 diagrams allow visualizing the behavior of the MVReg under different scenarios. The proposed scenario is a system with two independent copies of the MVReg in which the propagation of changes is not immediate (operations can be executed in the other agent before sending). In the first case, the merging of a delta with a counter higher than the local one is considered.

In this case, the result of the join operation is the state of the newest delta. In the second case, the delta being merged has in its state an older counter than the local one, so the execution of the merge does not change the local state. In the third case, it is the case that the delta state and the local state have the same counter. The solution used in the standard implementation (a) keeps both values in the register, leaving it up to the user to use the most convenient. In the modified version we use (b), an external variable known to both copies is used to select a single value.

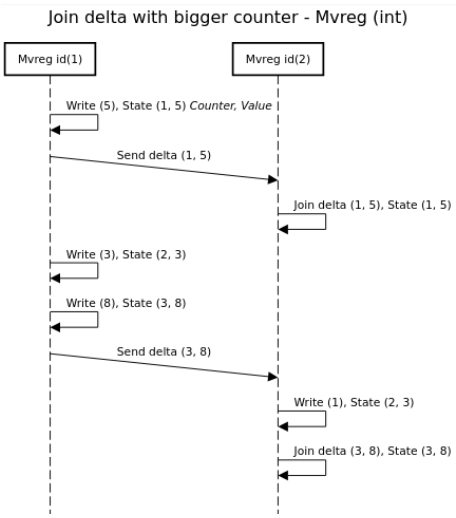


Figura 6.7: Join delta with newer counter.

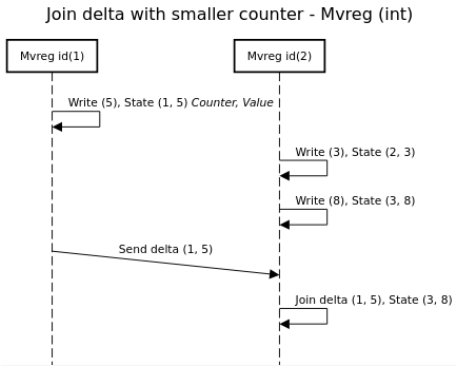
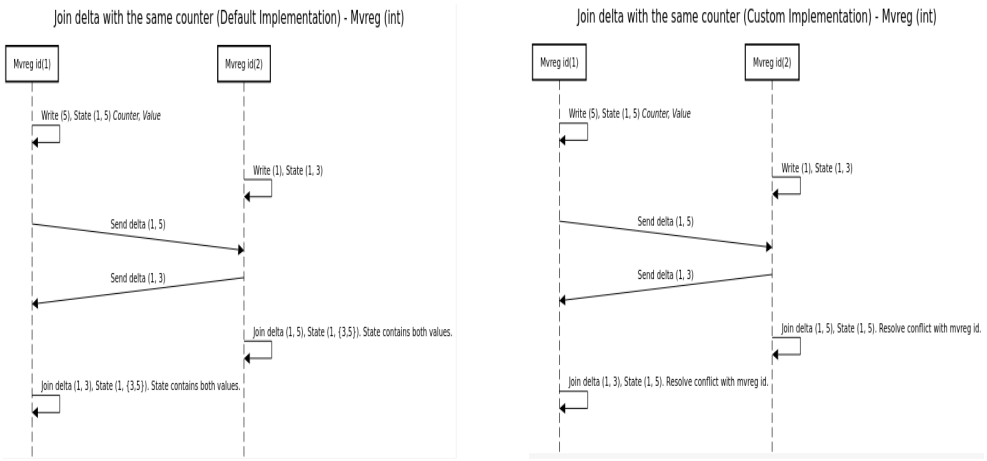


Figura 6.8: Join delta with older counter.



(a) Join delta conflict with the default implementation

(b) Join delta conflict with the custom implementation

Figura 6.9: Mvreg. Different approaches to conflicts.

The code blocks in the Appendix A show in a simplified way the implementation of this type. As can be seen in A.3, it is a generic class on the type stored. The attributes that form the type are an identifier, in this case a 64-bit unsigned integer and a dot_kernel A.2, which has the function of storing the map of dots with their respective values and maintaining the context of the MVReg (dot_context A.1). The main methods provided by the MVReg are the write operation and the join operation. The write operation corresponds to the creation of a local modification in the CRDT that generates a delta that can be propagated to the other replicas and integrated with the join operation.

A write implies the modification of the dotkernel, in this case by deleting all the elements of the map and updating the context with the rmv (remove) operation. The new element is then inserted into the dotkernel map. Both operations generate a delta. By performing the join operation between the two generated deltas, the delta to be used by the rest of the replicas to integrate the local changes is obtained.

The join operation calls the join_replace_conflict method of the dotkernel, which performs a traversal over the point maps of two mvregs. For each element

of each of the maps (this and o):

1. If the current element of the this map has a smaller counter than the current element of the or map, the current entry of this is removed.

2. If the current element of the map o is smaller than the current element of this, it is incremented to the next element of o.

3. If both elements have the same id and the content is different, it is chosen using an attribute that will be present in all the types on which the mvreg is used.

For practical purposes, as the mvreg will have at most one element, only one iteration will be performed, but the dotkernel and dotcontext operations are generic for any type of crdt and are maintained in case another crdt is incorporated in the future.

The MVReg API is not thread-safe and access must be controlled from different threads.

6.7.4. ThreadPool

The **ThreadPool** class was created to provide a mechanism for the execution of asynchronous tasks, whether or not a result of the execution of the tasks is required. Initially it was thought to reduce the latency between the reception and processing of a message by the network framework (being the reception the moment when the message is offered to the user and not the moment when it is actually received) by delegating the execution of the operations to be done with those messages to other threads and thus not keeping the network framework busy executing DSR code. The reason is that while the sending and receiving of messages is asynchronous, the execution of message reception callbacks is not. Delegating the execution of this callback to a separate thread prevents messages from accumulating in the RTPS structures. As will be seen later in the DDS

configuration this is important. On the one hand, we have a generic container to store any object that can be invoked as a function (either a function, a lambda or a functor) and its parameters A.4. In order to store it, a base class is used in which all its methods are virtual. The specialization inherits from this base class and is generic over the function and any number of parameters that are stored in a tuple without making any copies of the data. On the other hand we have the implementation of the ThreadPool A.6. The API offers only two methods, one to invoke functions without waiting for a result (spawn_task) and another to invoke functions for which a result may be desired after its invocation or some kind of synchronization is desired and returns a future object on which the wait operation can be performed (spawn_task_waitable). This future is generated by using the class packaged_task. To avoid errors related to the lifetimes of the parameters used in the invocation of the functions, a concept is declared that only allows the use of rvalues in the invocation of both functions A.5. This forces all parameters to be moved (using the move semantics introduced in C++11) or a temporary object created in-place.

Internally, the ThreadPool starts the number of threads specified by the user. These threads execute the thread_loop function, in which they iterate infinitely and wait in a non-active manner to be available for execution. The synchronization process between all the threads is performed with a mutex for the access to the task queue and a conditional_variable for waiting for the arrival of tasks.

6.7.5. Micro-Optimizations

In addition to the rigorous work done on the core of the system, mainly avoiding copies and using the move operation on the most costly operations, to improve performance, small optimisations have been implemented that serve as a proof of concept for future functionalities and can help agents reduce their workload. Some of these operations are:

- **Filter attributes**. The ability to ignore messages associated with particular attributes is provided. This allows agents who are only interested in some very specific attributes to reduce the number of join operations performed when processing external modifications. In the future, it is possible that there may be agents that only publish information, without requiring the internal representation of the network.

- **Filter own message**. Messages sent from the agent are not processed as external changes by the agent itself, although the merge result would not alter its state.

- **Filter false modifications**. When attributes of a node or an edge are modified, it is checked if the content is the same as in the previous version of the attribute. If it is, no modification delta is generated.

Capítulo 7

Performance analysis

To analyze the performance of G, 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.

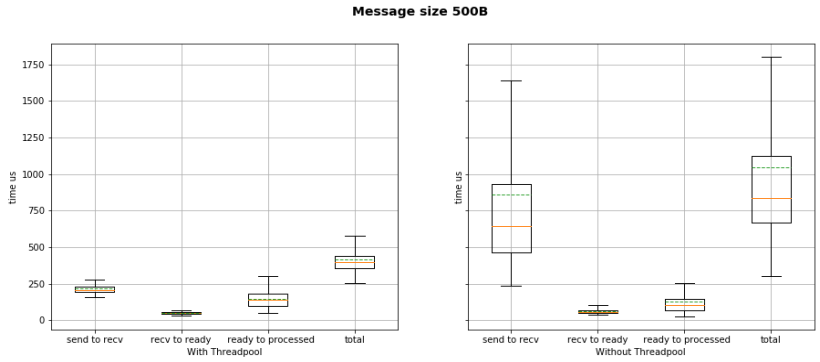
The first test compares the time elapsed between the sending of an attribute update message in FastDDS to the time when is processed 7.1. This time interval has been divided into three stages. The first is the receive time, which is the time it takes for the message to be sent over the network and read by DDS. The second is the time elapsed between the reception of the message and the invocation of the callback. The third and last is the time consumed in processing the message. These three stages together with the total time can be seen in the box plots. In the box plots the median is represented as an orange line and the mean as a green dashed line. In each of the images we find on the left the times using the ThreadPool and on the right without using it. As already explained in the implementation section, the motivation behind asynchronously processing messages is to avoid processing messages by blocking the message receiving and delivery thread. This is possible among other things because the CRDT used has

the property of being commutative. This test has been performed for messages of different sizes with a publication frequency of 15 milliseconds and with three agents performing the same type of operations in parallel.

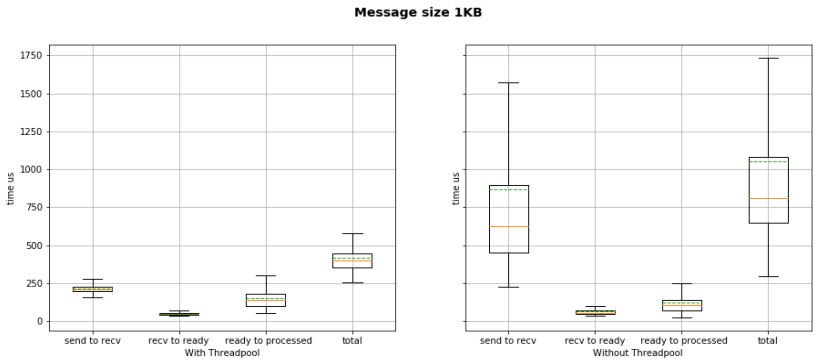
Analyzing the results we can see that as a general rule the time between sending and receiving is slower when the ThreadPool is not used even if it does not intervene at this stage. This could be due to the fact that the execution of the second and third FastDDS threads perform some kind of synchronization that may delay the reading of new messages. This time increase is not noticeable since we find sub-millisecond times when the communication is between agents on the same machine. With sizes of 150KB (e) and above that size, the reception time is no longer the most time-consuming step.

One of the main improvements obtained by using the ThreadPool, in addition to lower total times, is that the times in all stages follow much more uniform distributions and with much smaller intervals of values. As can be seen in the graphs on the left, the mean at all stages is quite close to the median, with the exception of the second stage for 500KB messages (f). In addition, the difference between the first and third quartile value is small, while in the first two stages of the plots in the diagrams on the right, the range of values between first and third quartile is quite wider and for message sizes of 150KB and 500KB, the mean is above the typical values of the distribution. For message sizes of 1MB (g) and above, in both, there is a larger variation in performance. The majority of the cost occurs between message reception and callback invocation, so it is probably related to synchronization issues and the cost of making copies of the messages in FastDDS. In the ThreadPool version the range of values between the second and third quartiles is much wider than between the first and the median. Being the median values around 1 millisecond and the mean and third quartile values over 8 milliseconds and 13 milliseconds respectively. In the version without ThreadPool the median and median values are closer to each other but higher than in the first version. It would be convenient to analyze more in depth the inner workings

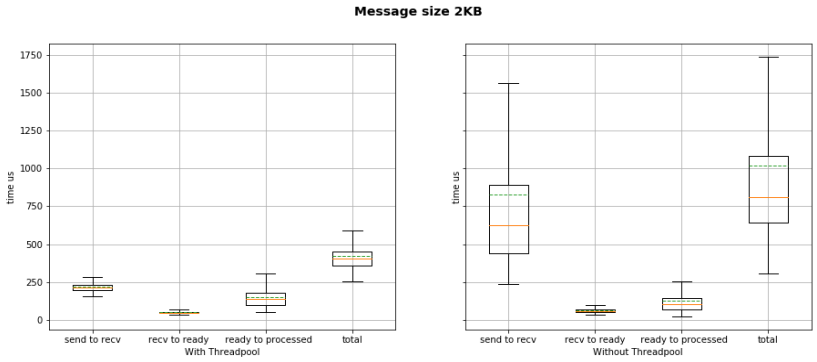
of this FastDDS stage to try to reduce the times in this type of messages.



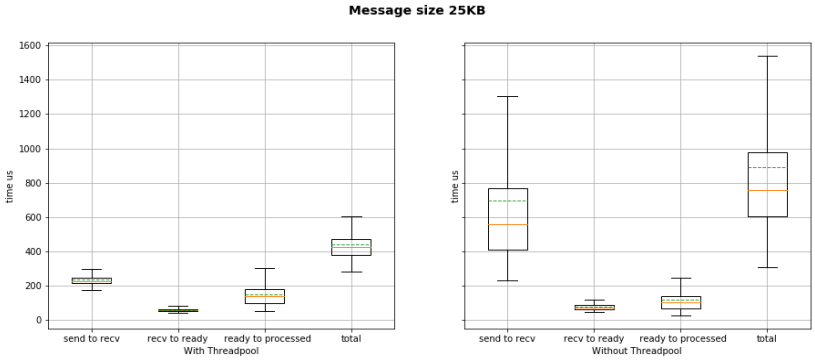
(a) Time spent in the different stages of message processing with message size of 500 Bytes using and not using the ThreadPool.



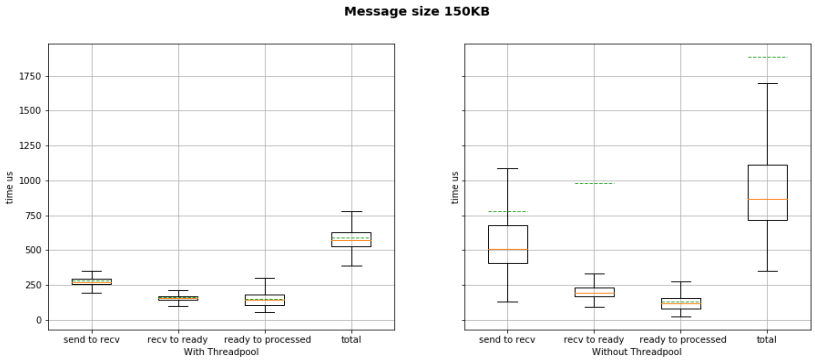
(b) Time spent in the different stages of message processing with message size of 1 Kilobyte using and not using the ThreadPool.



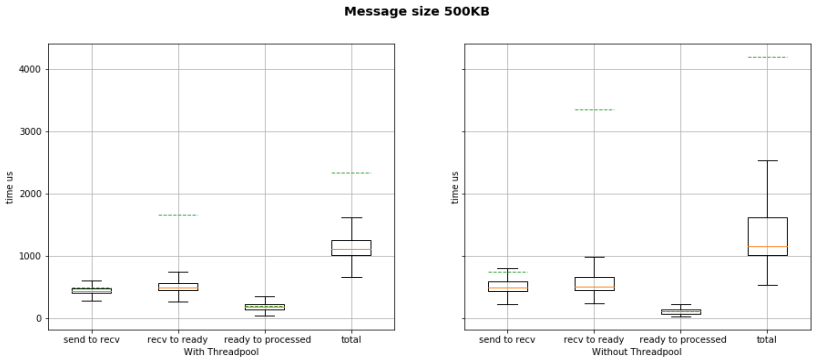
(c) Time spent in the different stages of message processing with message size of 2 Kilobytes using and not using the ThreadPool.



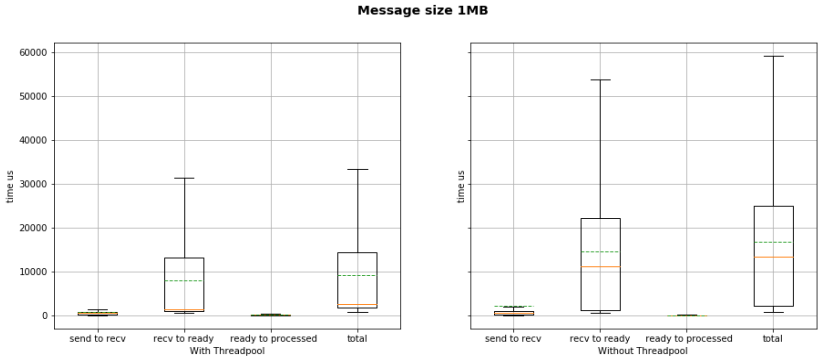
(d) Time spent in the different stages of message processing with message size of 25 kilobytes using and not using the ThreadPool.



(e) Time spent in the different stages of message processing with message size of 150 Kilobytes using and not using the ThreadPool.



(f) Time spent in the different stages of message processing with message size of 500 kilobytes using and not using the ThreadPool.



(g) Time spent in the different stages of message processing with message size of 1 Megabyte Bytes using and not using the ThreadPool.

Figura 7.1: Time spent in the different stages of message processing with different message sizes using and not using the ThreadPool.

The average values of the total time for each of the two versions are shown in figure 7.2. Performance is between 50 % and 100 % when using the ThreadPool under the release conditions represented in the test.

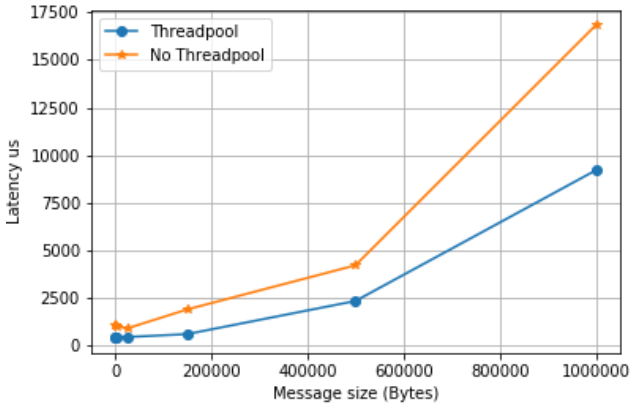


Figura 7.2: Comparison of the total time using and not using the ThreadPool.

The second group of tests performed is that of 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 an extra access to a map). In figure 7.3 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. In contrast to the previous graphs, the remote version also includes the local time of the agent performing the update. 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 as well. In the second option, the attribute is moved and the node is copied. In the third, the attribute and the node are moved. For small message sizes, around 60KB, there is not a big impact on performance 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 small attributes of a few bytes, such as numbers, text strings, arrays of a few elements, etc. But large attributes usually correspond to messages that are sent in fast periods, such as images, so it is important to process them efficiently.

Local updates can also be decomposed into three stages. The first is from the start of the operation to writing to the graph, the second is writing to FastDDS, and the third is sending Qt signals. Figures 7.4 show the box plots for the update using the move operation (right) and without using it (left). The values for the different attribute sizes show that the main cost is due to the attribute modification stage, node update. The cost of sending Qt signals is close to that of writing for some message sizes even when they do not send information. The cost of this step can increase in an unlimited way if the connection to the signals is done by a direct connection (Qt::DirectConnection, used by default in most situations) instead of a queued connection (Qt::QueuedConnection), when the execution is done at invocation time. During the test there were no directly connected signals, so the reason for the cost is unclear.

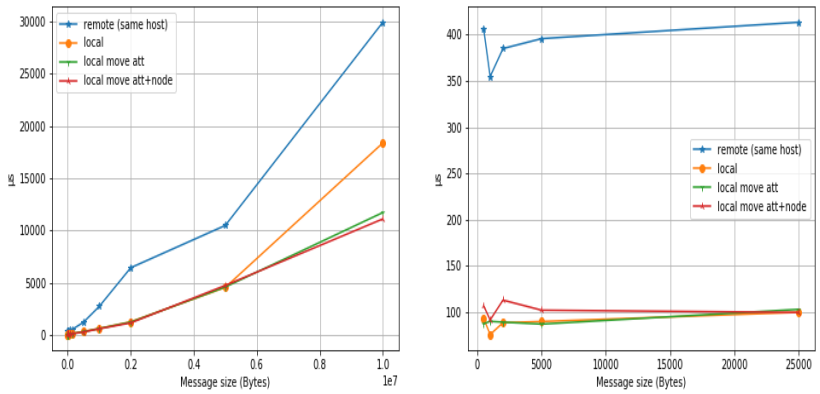
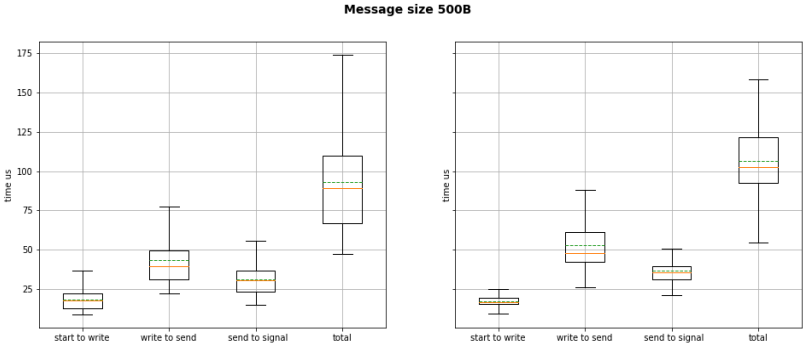
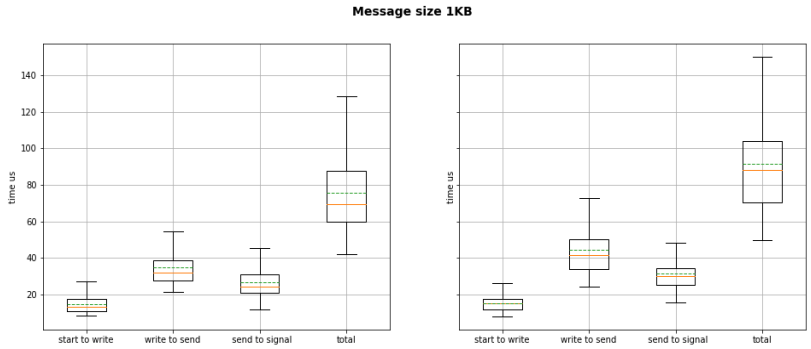


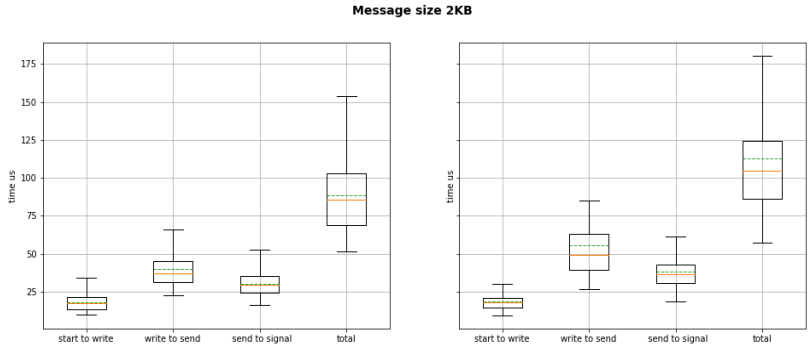
Figura 7.3: Comparison of update latency for local and remote messages on the same host using different local update methods.



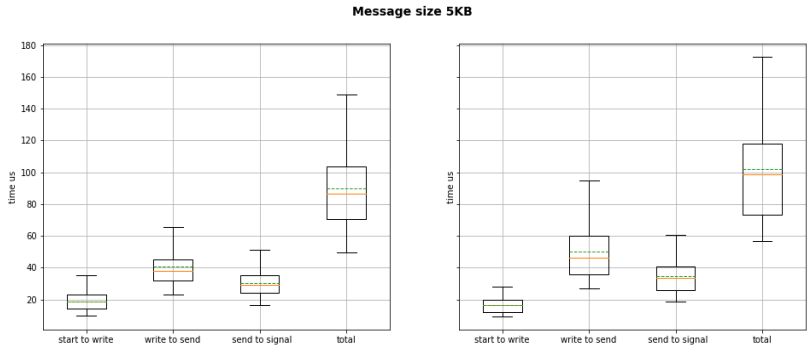
(a) Time spent in the different stages of local attribute update with message size of 500 Bytes moving and not movinf the node.



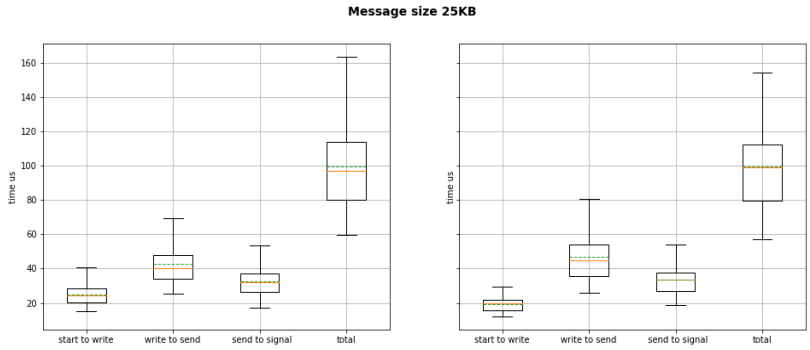
(b) Time spent in the different stages of local attribute update with message size of 1 Kilobyte moving and not movinf the node.



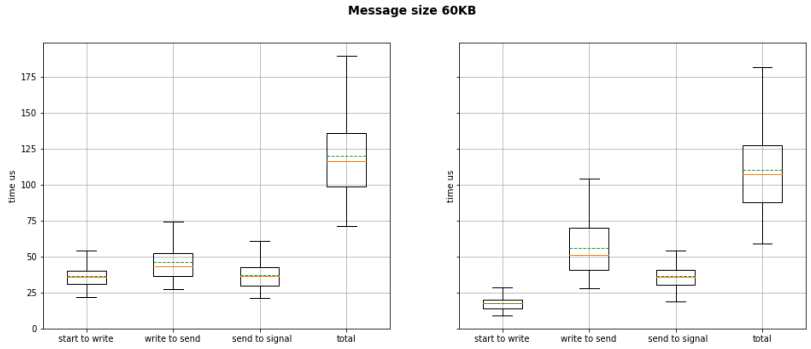
(c) Time spent in the different stages of local attribute update with message size of 2 kilobytes moving and not movinf the node.



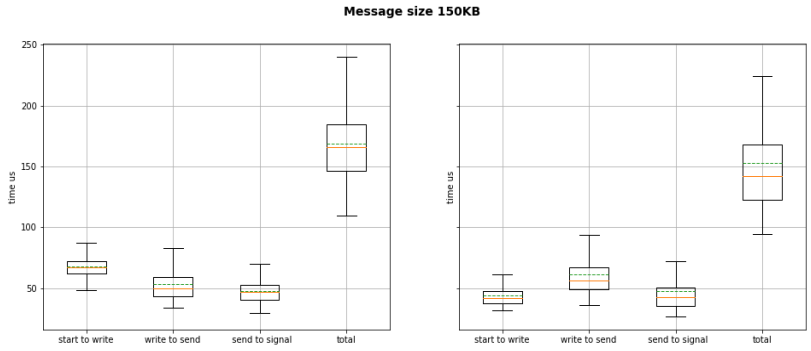
(d) Time spent in the different stages of local attribute update with message size of 5 kilobytes moving and not movinf the node.



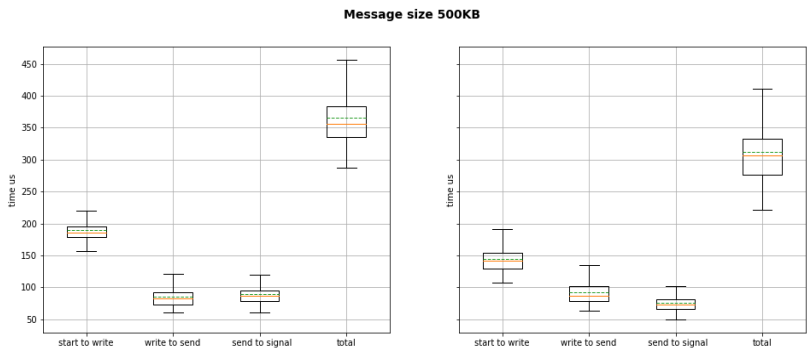
(e) Time spent in the different stages of local attribute update with message size of 25 kilobytes moving and not movinf the node.



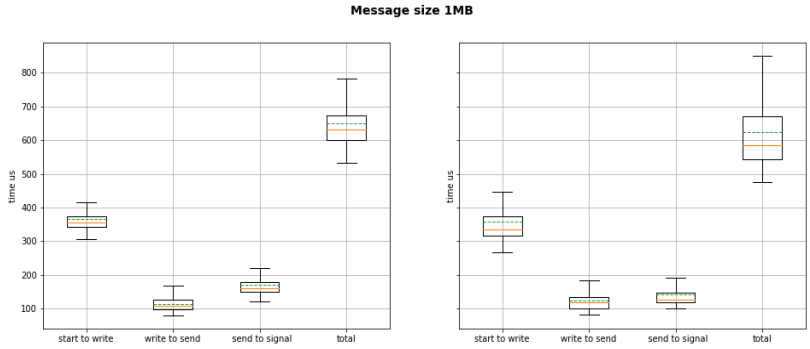
(f) Time spent in the different stages of local attribute update with message size of 60 kilobytes moving and not movinf the node.



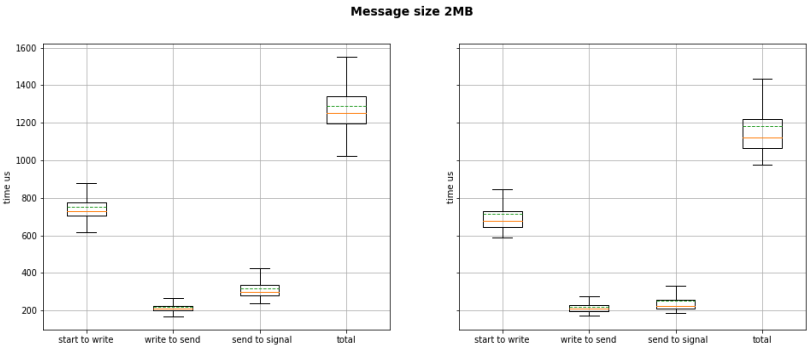
(g) Time spent in the different stages of local attribute update with message size of 150 kilobytes moving and not movinf the node.



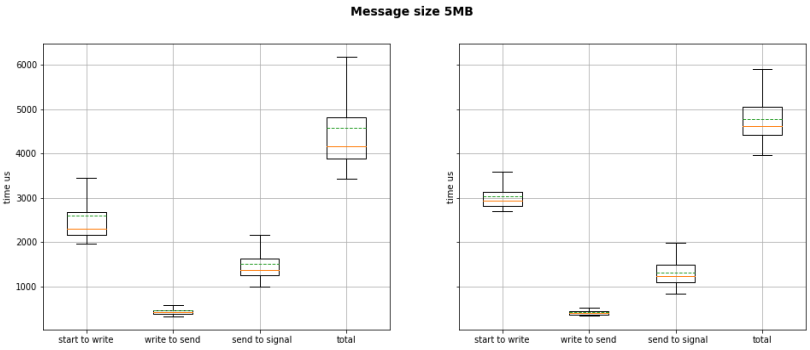
(h) Time spent in the different stages of local attribute update with message size of 500 kilobytes moving and not movinf the node.



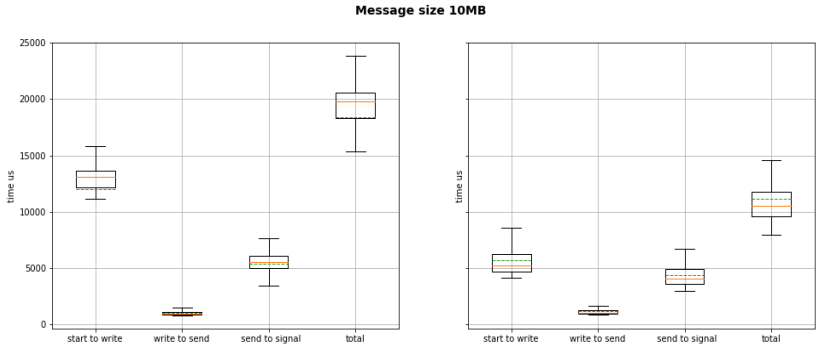
(i) Time spent in the different stages of local attribute update with message size of 1 Megabyte moving and not movinf the node.



(j) Time spent in the different stages of local attribute update with message size of 2 Megabytes moving and not movinf the node.



(k) Time spent in the different stages of local attribute update with message size of 5 Megabytes moving and not movinf the node.



(l) Time spent in the different stages of local attribute update with message size of 10 Megabytes moving and not movinf the node.

Figura 7.4: Comparison of local update latency moving and not moving the nodes with different attribute sizes.

The last test related to the performance of the attributes update and FastDDS is the update with an agent connected on a different computer. The communication is performed with the configuration indicated in the FastDDS section (Reliable Multicast over UDP). The results of the tests can be seen in figure 7.7. The update time as expected is higher for agents on different hosts. Starting with 500KB attributes the send time follows a linear growth where each megabyte has a send time of about 10 milliseconds. The times could be reduced by using a specialized switch instead of a generic router.

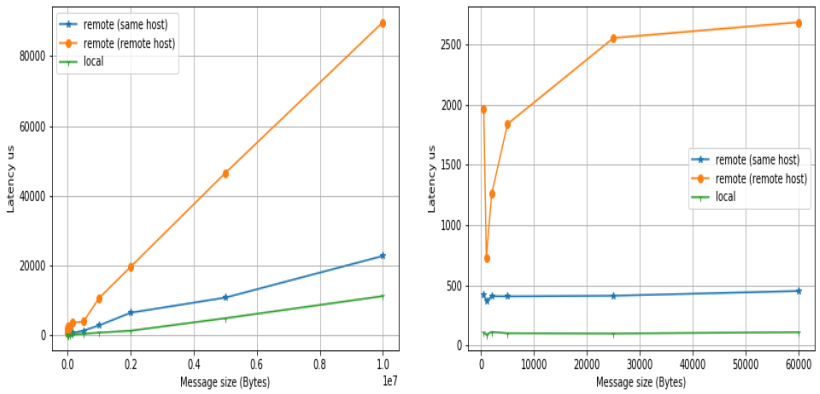


Figura 7.5: Latency for different message sizes in local updates, remote updates in the same host and remote updates in a different host.

The other operations performed on G are insertion and deletion of nodes and insertion and deletion of edges. Figure 7.6 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.

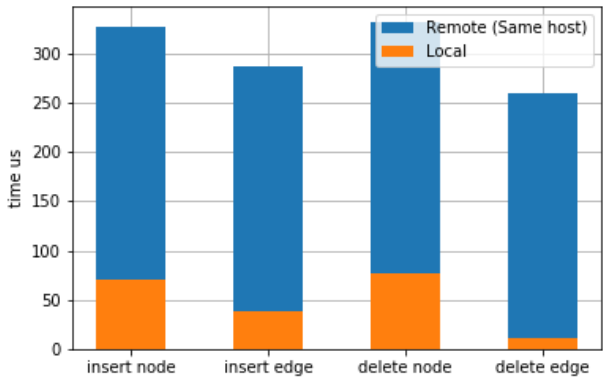


Figura 7.6: Comparison of local and remote latency for different operations.

Figure 7.7 compares the network usage for a different number of connected agents in which a single agent publishes and 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.

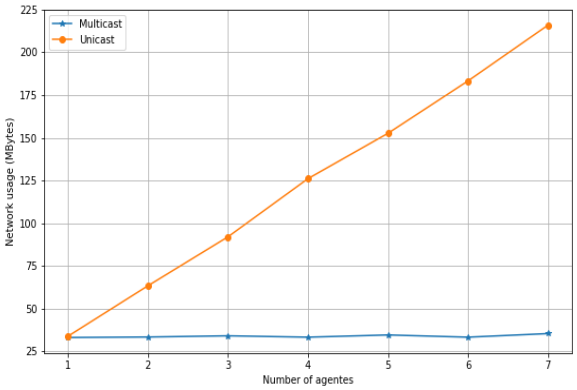


Figura 7.7: Networkg usage with one agent publishing and multiple agents receiving using multicast and unicast.

The last test performed is the comparison of the performance in the modification of attributes between Python and C++. In figure 7.8 can be seen how the performance is quite similar for attributes up to 5MB, where there is a performance loss of between 0% and 10%. The Python wrapper cannot take advantage of the move operation, so it is not straightforward to avoid having to copy the content. Yet, the cost of the update in Python is only about 25 milliseconds.

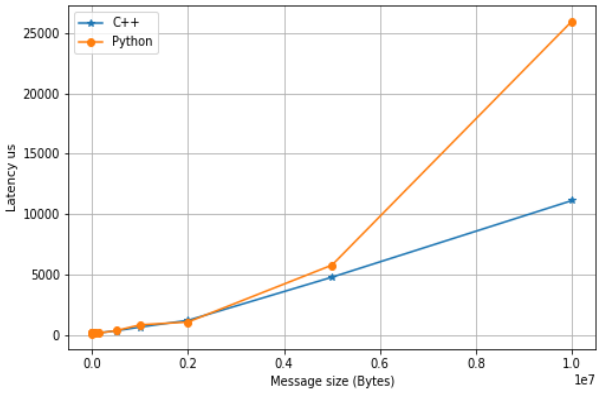


Figura 7.8: Latency in attribute updates in C++ and Python.

Capítulo 8

Use cases

The implementation of real use cases in DSR during the development of the libraries makes it possible to validate the decisions taken in the development and to check that the requirements of scalability, performance, latency and consistency are met. To speed up the development process, instead of using a real robot, a simulator is used to represent specific scenarios more quickly. The simulator used for the development is CoppeliaSim. The interaction between the robot simulator and the agents is performed by a specific agent for this task. This communication is bidirectional and serves as a bridge between the agents dedicated to the use case and the robot. Each of the robots represented in CoppeliaSim represents a real robot, with specific devices. The access to information in the bridge agent replicates as closely as possible the characteristics of the real robots.

8.1. Viriato's navigation in ALab

The first use case implemented is robot navigation in an indoor, controlled space. Robot navigation is a complex process that requires multiple tasks running in parallel. Some of these tasks are path planning, obstacle detection, positioning of the robot itself and real-time path correction. Depending on the complexity of the environment in which the robot is working or the constraints that exist,

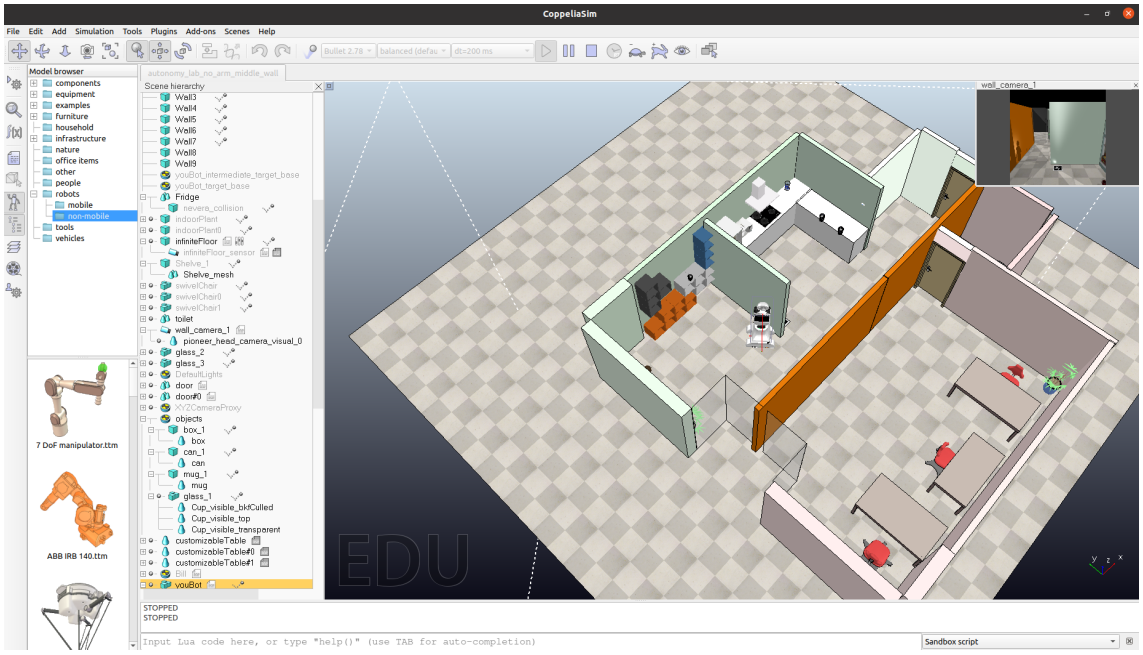


Figura 8.1: CoppeliaSim Simulator.

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, five agents work in parallel, in addition to the agent that connects to the simulator and can add other agents related to the interaction between the robot and the people, which add tasks such as people detection, control over navigation routes, etc.

The five main agents implemented are 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, path_planner_astar, in charge of planning the route and updating it iteratively, elastic_band, which smoothes the path designed by the previous agent using the robot's sensors, mission_controller_viriato, which allows navigation to be controlled by creating plans that subsequently generate the routes, and bumper, which creates a virtual bumper that restricts movements very close to physical elements, correcting the robot's trajectory.

8.2. Giraff and Pioneer Navigation in the IT hall

The second use case is the navigation in the computer hall of two real robots in a much larger environment than ALab. 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.).

Figura 8.2: Pionner Robot.

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 flexible 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, it is mission_controller_pionrer, which replaces the mission_controller_viriato agent. For security and testing reasons, robocomp agents or components are also added for manual control of the robots.

8.3. Social navigation: integration of human interaction rules

In the third use case, agents are added that allow the robot to act in environments where humans are present. 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.

Examples of the agents implemented in this field include the generation of personal and interaction spaces for humans (human_social_spaces agent), which is then used in the navigation of robots (social_navigation agent). Social navigation adds new restrictions to the navigation with which we were working to ensure that the coexistence between a robot and a group of people is optimal. When generating a route, it is necessary to determine the position of the people in order to avoid passing through their personal spaces, to avoid passing through the interaction spaces of several groups of people, etc. This requires precise detection

of the position and interaction of humans.

8.4. Detection, prediction and learning of objects in the environment

The fourth use case is the detection of household objects and their positioning in the world. Object detection is divided into two main tasks, the first is the identification of objects, usually done by cameras. The second is the positioning of the objects in the world, as identification alone does not give the robot much information. Positioning the objects is necessary to allow the robot to interact with the world. Positioning can be done using devices that provide 3D information such as stereoscopic cameras or LIDAR, or using 2D images together with algorithms that position the object in 3D. Once the position of the object with respect to the sensors is obtained, the position of the objects can be calculated using as a reference any other element whose position is known (for example the robot arm) using the RT tree of G. In this use case, two agents are implemented in addition to the one that connects to the simulator. Yolov4_tracker is in charge of the identification of objects in the robot's camera and positioning in the world using the position relative to the camera using the G RT_API and Inner_Eigen_APIs and performing an iterative process in which the position of the detected objects is specified. This process allows the information of the positioned objects to last until they are detected by the camera. The detection is performed using the Yolov4 library. The second agent, pan_tilt_attention_control, is in charge of orienting the robot's camera and tracking the object that is being monitored at any given moment.

Capítulo 9

Conclusion and future work

After the process of design, implementation and operation, the following conclusions about the system have been reached.

- The new DSR agents make the component-based development on which robocomp is based easier than with classic components. The main reasons are that in the new agents the information is centralised in G, whereas previously access was via the Ice or IceStorm interfaces. Instead of on-demand access, a shared-memory is kept up to date and consistent in a way that is invisible to the user, which would require a manual and component-specific implementation. This makes the development process much easier for the programmer, especially for users without a lot of experience in developing distributed, component-based applications. Another advantage is, for example, that making changes to an Ice interface would require regenerating all the components that use it, while with DSR the agents that consume the information do not need to make changes.

- The DDS standard provides an abstraction of the communication between agents that integrates well with the system design, but in specific situations it can limit the communication performance. The main case is the publication of attributes with a large size. As seen in the perfomance test, these perfomance limitations can be observed from around 3MB

upwards, regardless of the publication frequency. This can be solved in several ways. The first is to have relaxed reliability requirements for certain types of attributes. The second is to design and implement logic around specific attributes. An example of an attribute could be the publication of uncompressed images in the form of video. For this information stream, a compression of the image stream could be applied with an existing algorithm such as h264 or h265. The disadvantage for this case is that a specific implementation is required for each attribute, although a generic implementation could be offered to compress any attribute optionally. Another possible solution is to split attributes into fragments so that the image could be split into its colour components, sending 3 messages with 1/3 of the size.

- Features in the latest versions of C++ make it possible to provide more guarantees during compilation and reduce debugging time during development. This type of checking did not exist in the original agents.

- Scalability can be achieved in DSR vertically, by using machines with more resources, or horizontally, by distributing the execution of the agents among different machines. Each option has its advantages and disadvantages. For development and communication efficiency, vertical scaling is more interesting as it simplifies deployment and reduces latency to a minimum. The disadvantage of vertical scaling is that vertical scaling is limited by the technology available at the time. Horizontal scaling, on the other hand, is not so limited by the available technology, although it has other disadvantages such as local network configuration, agent deployment, etc.

Although the developed libraries are fully functional, it is still possible to further develop certain areas to improve performance, adjust development to the specific needs of users or simplify development. Some of the possible changes are discussed below.

- **Modularity**. Regarding the modularity of DSR, it would be of interest to modularize some of the main components of the libraries. The communication framework could be selected depending on the execution environment and its requirements. In a complex environment where reliability is needed and agents run in a distributed manner, Fast-DDS is a good choice. In an environment where agents run on a single machine, transport could be replaced by a shared memory or other protocols optimised for local information sharing. In environments where computers have reduced computational capabilities, a protocol such as MQTT could be used. The user interface framework could allow the creation of agents that instead of using Qt use web technologies or other libraries. The same would be true for G signals. These changes require a redesign of the API interfaces to allow G to be independent of the technologies used internally.

- **Optimizations**. There is room for improvement in the performance of accesses to the map used for storing the graph. Currently locks are performed on objects in the code block in the lock scope. Ideally, access to different map elements should be able to be performed concurrently, which would allow faster integration of remote changes and local modifications. This improvement would require the implementation of a data structure with a different concurrent access management or a lock-free one.

  As mentioned in the modularity improvements, one of the possibilities is the use of a shared memory for all agents running on a local machine. The impact on memory usage, memory access and write times would need to be analysed before this idea is implemented.

- **Extension**. In the design of DSR agents, one of the objectives is to simplify the development of new agents by generating code or extending them through APIs that include functionalities used by most agents. There are two possibilities for this. The first is the implementation of a query API that

allows access to G information following the most common access patterns. The most usual pattern is the execution of operations when changes in attributes and types of edges and specific nodes are detected.

Another extension of the usability of attributes and node and edge types is to add additional semantics to the most interesting ones. This can be done by providing a specific API for them, as in the case of the camera and RT edges. Another, more complex option is to use a knowledge base about the concepts represented in G and a logical reasoner to execute complex tasks. This can be useful for some complex tasks such as robot activity planning, navigation, human interaction, world understanding and interaction with objects in the world.

- **Security**. Although agents typically run in local or wired environments, it would be desirable to add a layer of authentication and the possibility to encrypt messages in DSR.

Anexos

Apéndice A

Code

A.1. MVReg

```
class dot_context {
public:

std::map<key_type, int> cc; // Compact causal context
std::set<std::pair<key_type, int> > dc; // Dot cloud

...

bool dotin(const std::pair<key_type, int> &d) const {
    const auto itm = cc.find(d.first);
    if (itm != cc.end() && d.second <= itm->second) return true;
    if (not dc.empty() and d.second < dc.rbegin()->second) return true;
    if (dc.count(d) != 0) return true;
    return false;
}

void compact() { // Compact DC to CC if possible
    bool flag; // may need to compact several times if ordering not best
    do {
        flag = false;
```

```
        for (auto sit = dc.begin(); sit != dc.end();) {
            auto mit = cc.find(sit->first);
            if (mit == cc.end()) // No CC entry
                if (sit->second == 1) // Can compact
                {
                    cc.insert(*sit);
                    dc.erase(sit++);
                    flag = true;
                } else ++sit;
            else // there is a CC entry already
            if (sit->second == cc.at(sit->first) + 1) // Contiguous, can
            ↪  compact
            {
                cc.at(sit->first)++;
                dc.erase(sit++);
                flag = true;
            } else if (sit->second <= cc.at(sit->first)) // dominated, so
            ↪  prune
            {
                dc.erase(sit++);  // no extra compaction oportunities so
                ↪  flag untouched
            } else ++sit;
        }
    } while (flag == true);
}

std::pair<key_type, int> makedot(const key_type &id) {
    // On a valid dot generator, all dots should be compact on the used id
    // Making the new dot, updates the dot generator and returns the dot
    if (auto [it, res] = cc.insert(std::make_pair(id, 1)) ; !res) {
        it->second+=1;
        return *it;
    } else {
        return *it;
    }
```

```
}

void join(const dot_context &o) {
    if (this == &o) return; // Join is idempotent, but just dont do it.
    auto mit = cc.begin();
    auto mito = o.cc.begin();
    do {
        if (mit != cc.end() && (mito == o.cc.end() || mit->first <
        ↪  mito->first)) {
            ++mit; // entry only at here
        } else if (mito != o.cc.end() && (mit == cc.end() || mito->first <
        ↪  mit->first)) {
            cc.insert(*mito);
            ++mito; // entry only at other
        } else if (mit != cc.end() && mito != o.cc.end()) {
            cc.at(mit->first) = std::max(mit->second, mito->second); // in
            ↪  both
            ++mit;
            ++mito;
        }
    } while (mit != cc.end() || mito != o.cc.end());
    // DC Set
    for (const auto &e : o.dc)
        insertdot(e, false);

    compact();
}
```

Source Code A.1: Dot Context Code.

```
template<typename T>
class dot_kernel {
public:
```

```
std::map<std::pair<key_type, int>, T> ds;  // Map of dots to vals
dot_context c;

...

void join_replace_conflict(dot_kernel<T> &&o) {

    if (this == &o) return; // Join is idempotent, but just dont do it.
    // DS. will iterate over the two sorted sets to compute join
    auto it = ds.begin();
    auto ito = o.ds.begin();
    do {
        if (it != ds.end() && (ito == o.ds.end() || it->first < ito->first))
        ↪ {
            if (o.c.dotin(it->first)) { // other knows dot, must delete here
                ds.erase(it++);
            } else { // keep it
                ++it;
            }
        } else if (ito != o.ds.end() && (it == ds.end() || ito->first <
        ↪ it->first)) {
            if (!c.dotin(ito->first) || ds.empty()) { // If I dont know,
            ↪ import
                ds.insert(std::move(*ito));
            }
            ++ito;
        } else if (it != ds.end() && ito != o.ds.end()) {
            if (it->second.agent_id() > ito->second.agent_id() && *it !=
            ↪ *ito) {
                it = ds.erase(it);
                ds.insert(std::move(*ito));
            } else {
                ++it;
            }
            ++ito;
```

```
            }
        } while (it != ds.end() || ito != o.ds.end());
        // CC
        c.join(std::move(o.c));
    }

    dot_kernel<T> add(key_type &id, T &&val) {

        dot_kernel<T> res;
        std::pair<key_type , int> dot = c.makedot(id);
        ds.insert(std::pair<std::pair<key_type , int>, T>(dot, val));
        res.ds.insert(std::pair<std::pair<key_type , int>, T>(dot, val));
        res.c.insertdot(dot);
        return res;
    }

    dot_kernel<T> rmv()  // remove all dots
    {
        dot_kernel<T> res;
        for (const auto &dv : ds)
            res.c.insertdot(dv.first, false);
        res.c.compact();
        ds.clear(); // Clear the payload, but remember context
        return res;
    }
```

Source Code A.2: Dot Kernel Code.

```
template<typename V>
class mvreg
{
public:
key_type id;
dot_kernel<V> dk; // Dot kernel
```

```
...

mvreg<V> write(const V &val) {
    mvreg<V> r, a;
    r.dk = dk.rmv();
    a.dk = dk.add(id, val);
    r.join(std::move(a));
    assert(r.dk.ds.size() <= 1);
    return r;
}

void join(mvreg<V> &&o) {
    dk.join_replace_conflict(std::move(o.dk));
    assert(dk.ds.size() <= 1);
}
```

Source Code A.3: MvReg Code.

A.2. ThreadPool

```
class function_wrapper_base
{
public:
    virtual ~function_wrapper_base(){};
    virtual void operator()() {};
};

template <typename Function, typename... Arguments>
class function_wrapper : public function_wrapper_base
{
public:
    function_wrapper(Function &&fn, std::tuple<Arguments...> args)
```

```
        : f(std::forward<Function>(fn)), args(std::move(args)){};
    void operator()() override
    {
        std::apply(f, std::move(args));
    }
private:
    Function f;
    std::tuple<Arguments...> args;
};
```

Source Code A.4: Container for callable objects.

```
template<typename ... T>
concept only_rvalues  =
    (std::negation<
        std::bool_constant<std::is_lvalue_reference<T&&>::value>
        >::value  && ...);
```

Source Code A.5: Constrain parameters to rvalues.

```
class ThreadPool
{
public:
    ...

    template <typename Function, typename... Arguments>
    void spawn_task(Function &&fn, Arguments &&... args)
        requires (only_rvalues<Arguments&& ...> and
        ↪  std::is_invocable<Function &&, Arguments &&...>::value)
    {
        std::unique_lock<std::mutex> task_queue_lock(tp_mutex,
        ↪  std::defer_lock);
        task_queue_lock.lock();
```

```
    auto tmp_ptr = std::unique_ptr<function_wrapper_base>(new
    ↪  function_wrapper<Function,
    ↪  Arguments...>(std::forward<Function>(fn),
    ↪  std::forward_as_tuple(std::move(args)...)));
    tasks.emplace(std::move(tmp_ptr));
    task_queue_lock.unlock();
    cv.notify_one();
}

template <typename Function, typename... Arguments>
auto spawn_task_waitable(Function &&fn, Arguments &&... args)
    requires (only_rvalues<Arguments&& ...> and
    ↪  std::is_invocable<Function &&, Arguments &&...>::value)
{
    std::unique_lock<std::mutex> task_queue_lock(tp_mutex,
    ↪  std::defer_lock);

    auto task = std::packaged_task<std::invoke_result_t<Function,
    ↪  Arguments...>()>(
            [fn_ = std::forward<Function>(fn), args_ =
            ↪  std::forward_as_tuple(args...)]() mutable -> auto {
                return std::apply(std::move(fn_), std::move(args_));
            }
    );

    auto future = task.get_future();

    task_queue_lock.lock();
    auto tmp_ptr = std::unique_ptr<function_wrapper_base>(
            new
            ↪  function_wrapper<std::packaged_task<std::invoke_result_t<Function,
            ↪  Arguments...>()>>(
                    std::move(task), std::tuple<>{}));
    tasks.emplace(std::move(tmp_ptr));
    task_queue_lock.unlock();
```

```
        cv.notify_one();

        return future;
    }

private:

    void thread_loop(int i)
    {
        [[maybe_unused]] static thread_local uint32_t thread_index = i;
        std::unique_lock<std::mutex> task_queue_lock(tp_mutex,
        ↪  std::defer_lock);
        while (!done)
        {
            task_queue_lock.lock();

            cv.wait(task_queue_lock,
                    [&]() -> bool { return !tasks.empty() || done; });

            if (done) {
                task_queue_lock.unlock();
                cv.notify_all();
                break;
            }

            std::unique_ptr<function_wrapper_base> t =
            ↪  std::move(tasks.front());
            tasks.pop();
            task_queue_lock.unlock();

            if (t != nullptr)
            {
                (*t)();
            }
```

```
            }
        }

        std::vector<std::thread> threads;
        std::queue<std::unique_ptr<function_wrapper_base>> tasks;
        static thread_local uint32_t thread_index;
        std::condition_variable cv;
        std::atomic_bool done = false;
        mutable std::mutex tp_mutex;
    };
```

Source Code A.6: ThreadPool Code.

Apéndice B

APIs

EL acceso a G se realiza mediante una **API thread safe** que incluye métodos para la lectura, inserción, modificación y borrado de nodos y arcos. Esta API permite acceder al grafo de tal manera que el programador no necesita tener en cuenta aspectos como la sincronización de los hilos y la propagación de los cambios por la red. La extensión de las operaciones sobre G se realiza mediante el desarrollo de APIs de más alto nivel que hacen uso de esta y ofrecen abstracciones sobre el grafo u operaciones que se realizan de manera habitual. Las APIs desarrolladas hasta el momento son dsr_agent_info, dsr_camera, dsr_inner_eigen, dsr_rt y dsr_utils.

B.1. API Base

La API base está compuesta por **API privada**, que realiza las operaciones CRUD sobre el grafo sin realizar control sobre la concurrencia, los índices y la integración de cambios de la red. Sobre estos métodos existe una API pública con control sobre la concurrencia utilizando exclusión mutua. La API pública también incluye métodos para el acceso a los tipos nativos del **variant** utilizado en los atributos, además de su modificación. El modelo de acceso a la información de G se basa en la obtención de una copia, la lectura o modificación de los atributos y

posteriormente su reinserción. El acceso mediante copias permite mayor control sobre el estado del grafo, mayor eficiencia en la propagación de los cambios y mayor disponibilidad en la concurrencia, al reducir los bloqueos a los momentos de obtención e inserción de la información.

La API pública está dividida en un API pública y un API privada. A continuación se va a describir el funcionamiento de las funciones más importantes.

Nodos. Operaciones CRUD sobre nodos.

- **get_node**. Recibe como parámetro de entrada el identificador único del nodo o una referencia constante al string del nombre, que también es único. Devuelve un elemento de tipo optional que contiene el Nodo utilizando la representación de usuario. Si el nodo no existe devuelve nullopt_t. Esta operación implica un lock compartido (permite lecturas concurrentes pero no escrituras), la búsqueda en el mapa de nodos (unordered_map, acceso mediante hash) y la copia del contenido utilizando. Esta función invoca la función de la api privada get_.

- **delete_node**. Recibe como parámetro de entrada el identificador único del nodo o una referencia constante al string del nombre, que también es único. Devuelve un booleano indicando si se ha borrado o no el nodo. La operación toma un lock único e invoca a la función delete_node_ de la API privada, en la que si el nodo existe en G, se elimina del mapa generando un delta, después se utiliza el índice de arcos que apuntan a un nodo para borrar todos los arcos que apuntan al nodo borrado y se generan sus deltas. Posteriormente se actualizan los mapas de índices . Envía la señal del_node_signal con el id del nodo borrado y una señal del_edge_signal para cada arco que sale y apunta a dicho nodo. Se envían deltas de del nodo borrado y todos los arcos que apuntan al nodo.

- **insert_node** Recibe como parámetro una referencia mutable nodo utilizando la representación de usuario. Devuelve un optional con el

identificador único generado para el nodo si la operación se realiza con éxito. Rquiere de un lock único. La operación genera un identificador único para el nodo utilizando el generador de ids. Si al nodo se le ha asignado un nombre y este no existe ya en el grafo, se utiliza ese nombre, en caso contrario, se generado un nombre con el tipo del nodo y la representación hexadecimal del identificador. La función invoca al método insert_node_ de la API privada, en la que se añade el nodo al mapa una copia del nodo utilizando su representación interna (CRDT), se genera un delta de la operación y se actualizan los mapas. Envía una señal update_node_signal con el identificador del nodo y una señal update_edge_signal con cada arco incluido en el nodo. Se publica el delta del nodo.

- **update_node** Recibe como parámetro una referencia mutable nodo utilizando la representación de usuario. Devuelve un booleano indicando si la operación se realiza con éxito. Requiere de un lock único, de que el nodo haya sido previamente borrado y de que exista en el grafo. Invoca el método update_node_ de la API privada, en el que se modifican los atributos existentes, se añaden los nuevos y se borran los que ya no existen. Se genera un vector de deltas para todos los elementos modificados. Envía una señal update_node_signal con el identificador del nodo y una señal update_node_attr_signal con los atributos que han sido añadidos, modificados o borrados. Envía un vector de deltas de atributos modificados.

Edge. Operaciones CRUD sobre arcos.

- **get_edge**. Recibe como parámetros de entrada los identificadores de origen y destino del arco y una referencia constante al tipo del arco que se quiere obtener o referencias constantes a los nombres de los nodos de origen y destino junto al tipo del arco. Devuelve un elemento de tipo optional que contiene el Edge utilizando la representación de usuario. Si el arco no existe devuelve nullopt_t. Esta operación implica un lock compartido e invoca el método get_edge_, en que se devuelve una copia del arco.

- **delete_edge** Recibe como parámetros de entrada los identificadores de origen y destino del arco y una referencia constante al tipo del arco que se quiere eliminar o referencias constantes a los nombres de los nodos de origen y destino junto al tipo del arco. Devuelve un booleano con el resultado de la operación. Requiere de un lock único e invoca al método delete_edge_ de la API privada, en el que se borra el arco del nodo origen, se genera un delta y se actualizan los índices. Envía una señal del_edge_signal con la información del arco y el delta del arco borrado.

- **insert_or_assign_edge** Recibe como parámetro de entrada una referencia constante a un Edge y devuelve un booleano con el resultado de la operación. Requiere de un lock único y e invoca al método de la API privada insert_or_assign_edge_, en el que si el arco ya existe, se actualizan los atributos de la misma forma que en la operación update_node, generando deltas de las modificaciones en los atributos. En caso de que el arco sea nuevo, se inserta directamente en el nodo y se genera un delta del arco. Envía una señal update_edge_signal se envían los deltas. En caso de actualización, también se envía una señal update_edge_attr_signal con todos los atributos modificados.

Métodos sobre atributos. Métodos utilizados para la modificación de atributos de manera simple sin necesidad de acceder de manera directa a los atributos de un nodo o un arco. Por cada método de moficación de atributos, existe otro equivalente con el prefijo runtime_checked_, en el que la comprobación sobre los tipos de los atributos se realiza durante la ejecución en vez de durante la compilación. Esto permite que puedan acceder a estos métodos agentes u otros elementos del sistema que requieran de acceder a atributos de manera arbitraria, sin conocer su nombre de antemano.

- **get_attrib_by_name**. Método genérico que recibe como parámetros un elemento de tipo Node, CRDTNode, Edge o CRDTEdge y un atributo y

deuelve un optional con el tipo contenido el atributo en vez del variant, evitando tener que utilizar un switch o un visitador en todos los lugares en los que se utiliza el atributo. Los parámetros deben cumplir con los concepts any_node_or_edge para limitar la función a nodos o arcos y is_attr_name para el atributo. Dado que los optional no pueden almacenar referencias, los atributos que tienden a tener copias costosas como vectores se obtienen como std::optional<std::reference_wrapper<const T> >. En este caso es necesario que el programador controle la validez de las referencias.

- **add_or_modify_attrib_local**. Método genérico que recibe como parámetros un elemento de tipo Node, CRDTNode, Edge o CRDTEdge, un atributo y el valor del atributo a insertar. Debe cumplir los concepts any_node_or_edge para limitar la función a nodos o arcos, is_attr_name para el atributo, allowed_types para validar el tipo del atributo que se quiere insertar y la función constexpr valid_type, que valida el tipo del atributo con el atributo seleccionado. El método añade o modifica el atributo en el nodo o arco de manera local.

- **add_attrib_local**. Método genérico con los mismos parámetros y restricciones que add_or_modify_attrib_local, con la salvedad de que solo sirve para añadir un valor, no para modificarlo. El método añade el atributo en el nodo o arco de manera local.

- **modify_attrib_local** Método genérico con los mismos parámetros y restricciones que add_or_modify_attrib_local, con la salvedad de que solo sirve para modificar un valor, no para añadirlo. El método modifica el atributo en el nodo o arco de manera local.

- **remove_attrib_local** Método genérico que recibe como parámetros un elemento de tipo Node, CRDTNode, Edge o CRDTEdge y un atributo y deuelve un bool con el resultado de la operación. Los parámetros deben cumplir con los concepts any_node_or_edge para limitar la función a nodos

o arcos y is_attr_name para el atributo. El método elimina el atributo si existe en el nodo o arco de manera local.

Métodos de conveniencia. Métodos para el acceso a múltiples nodos o edges a la vez. Tienen la ventaja de aprovechar los índices que permiten no realizar búsquedas en el grafo.

- **get_nodes_by_type**. Receibe como parámetro el nombre de un tipo y devuelve un vector con copias de todos los nodos encontrados para ese tipo. Requiere un un lock compartido. Utiliza el índice de node_types.
- **get_edges_by_type**. Receibe como parámetro el nombre de un tipo y devuelve un vector con copias de todos los arcos encontrados para ese tipo. Requiere un un lock compartido. Utiliza el índice de edge_types.
- **get_edges_to_id**. Receibe como parámetro el identificador de un nodo y devuelve un vector con copias de todos los arcos encontrados que apuntan a ese nodo. Se utiliza el índice to_edges para acceder directamente sin recorrer el mapa del grafo. Requiere un un lock compartido.

Otros métodos de G.

- **set_ignored_attributes**. Método genérico que permite un número de argumentos variable que satisfagan el concept is_attr_name. todos los elementos son añadidos al set de atributos ignorados, que se utiliza para descartar cambios en esos atributos.
- **G_copy**. Devuelve una copia de G desconectada de la red. Esta copia tampoco envía señales. Es útil para realizar pruebas en múltiples nodos antes de modificar el grafo compartido.

Operaciones de fusión. Consumen deltas de agentes remotos y las integran en el grafo. Estas funciones se invocan como callbacks desde los readers de FastDDS.

- **join_delta_node**. Recibe como parámetro un delta de un Nodo. Estos deltas se generan únicamente en la creación o borrado de un nodo, ya que en la actualización únicamente se envían deltas de atributos, que se procesan en el método join_delta_node_attr. El comportamiento es el mismo que la invocación de la operación insert_node o delete_node, dependiendo del contenido del delta, aunque al realizarse mediante la operación join del crdt, el dotkernel es el del delta. Requiere un lock exclusivo. Envía las mismas señales que se hubiesen enviado en la operación local.

- **join_delta_edge**. Recibe como parámetro un delta de un Edge. Estos deltas se generan únicamente en la creación o borrado de un arco, ya que en la actualización únicamente se envían deltas de atributos, que se procesan en el método join_delta_edge_attr. El comportamiento es el mismo que la invocación de la operación insert_node o delete_node, dependiendo del contenido del delta, aunque al realizarse mediante la operación join del crdt, el dotkernel es el del delta. Requiere un lock exclusivo.Envía las mismas señales que se hubiesen enviado en la operación local.

- **join_delta_node_attr**. Recibe como parámetro el delta de modificación de un atributo de un nodo. Requiere de un lock único y el resultado de la operación es la modificación, inserción o borrado del atributo. La función no envía señales, sino que la función devuelve información suficiente para que el métodos desde el que es invocado envía las señales de la misma forma que lo haría en una actualización local.

- **join_delta_edge_attr**. Recibe como parámetro el delta de modificación de un atributo de un arco. Requiere de un lock único y el resultado de la operación es la modificación, inserción o borrado del atributo. La función no envía señales, sino que la función devuelve información suficiente para que el métodos desde el que es invocado envía las señales de la misma forma que lo haría en una actualización local.

- **join_full_graph**. Recibe como parámetro la representación completa de G y la integra como si realiza una inserción de cada nodo utilizando la operación insert_node, aunque al realizarse mediante la operación join del crdt, el dotkernel es el del delta. Envía las mismas señales que en la inserción de todos los nodos. Requiere un lock exclusivo.

B.2. Agent Info API

La API Agent Info publica información sobre el agente que se ejecuta. La publicación se realiza de forma periódica e incluye datos como el uso de CPU del agente, la memoria utiliza, tiempo de ejecución, timestamp de lanzamiento, etc. Por el momento la información publicada no es extensible. Permite que los agentes identifiquen qué agentes están conectados.

Esta API únicamente tiene métodos para parar el timer y comprobar si se está ejecutando. Al inicializarla, se crea un hilo con un bucle que ejecuta que la función stop_timer es incovada o se elimina el objeto. En el bucle se ejecuta una función que obtiene información del proceso mediante llamadas a la línea de comandos.

B.3. Camera API

. Camera API ofrece métodos para realizar oeperaciones sobre los datos de nodos rgb. Las operaciones de esta API están asociadas a un nodo rgb, del que se obtiene información como la resolución de la imagen, la focal, etc. Estos atributos se usan para obtener la imagen a partir del vector de bytes, obtener la imagen de profundidad, obtener la nube de puntos de la imagen, la proyección de puntos 3d en la imagen, etc.

B.4. RT API

La API RT (Rotation-Transform) ofrece funciones para la creación y el acceso a una abstracción lógica sobre el grafo G. Los nodos que representan objetos o estructuras físicas de las que se necesita conocer su posición están conectados mediante enlaces que incluyen los atributos de rotación y traslación respecto a otro nodo. Además de la rotación y la translación actual, también es posible almacenar un histórico de estas, lo que permite realizar correcciones y predicciones en los cálculos. La abstracción creada por los arcos RT crea un árbol al que están conectados todos estos nodos y que puede ser recorrido para calcular movimientos, posiciones, etc. Esta abstracción es una de las bases de la navegación del robot y la detección de la posición de los objetos.

Por razones de rendimiento, los métodos de la API RT tienen acceso a la API privada de G y realizan su propia gestión de la concurrencia. Los métodos disponibles en esta API son:

- **insert_or_assign_edge_RT**. Recibe como parámetros una referencia a un nodo, un identificador de destino, un vector de traslación y otro de rotación. El método toma un lock único sobre G e invoca el método de la api privada insert_or_assign_edge_ para insertar el arco con los nuevos atributos y update_node_ para modificar, si es necesario, los atributos level y parent del nodo destino. Esta función propaga deltas de creación de un arco (en caso de que el arco sea nuevo), de actualización de atributos para el arco y de modificación de atributos para el nodo si ha sido necesario modificar algo. Envía también las señales update_edge_attr_signal, update_edge_signal para el arco RT y si es necesario update_node_signal y update_node_attr_signal para el nodo destino

- **get_edge_RT**. Recibe como parámetros una referencia constante a un nodo y el identificador del destino. Devuelve el arco con tipo RT entre ellos utilozando un optional.

- **get_edge_RT_as_rtmat**. Recibe como parámetro una referencia constante a un nodo y un timestamp representado como un entero sin signo de 64 bits y devuelve un objeto RTMat, que es un alias para el tipo Eigen::Transform<double, 3, Eigen::Affine> . Este objeto representa una matriz con los componentes de rotación en cada uno de los ángulos y una traslación. En caso de ser 0 el valor del timestamp, el método utiliza los últimos valores conocidos de rotación (atributo rt_rotation_euler_xyz) y traslación (atributo rt_translation) para calcular la matriz. En caso contrario, se utiliza el timestamp más cercano al parámetro de entrada y se calcula la matriz con los valores de los atributos asociados a él.

- **get_translation**. Recibe como parámetros una referencia constante a un nodo o un identificador de origen, un identificador de destino y un timestamp. y devuelve un array de 3 elementos de Eigen. La función accede a los valores del atributo rt_translation utilizando el timestamp de la misma manera que el método get_edge_RT_as_rtmat.

B.5. Inner Eigen API

Inner Eigen API ofrece métodos para calcular transformaciones cinemáticas entre nodos y operaciones para convertir la matriz de transformación entre diferentes representaciones. Esta API hace uso de la librería Eigen y la API RT. Las operaciones de esta API se utilizan para recorrer el árbol RT. Al tratarse de cálculos que pueden ser costosos computacionalmente, cada vez que se calcula una transformación, se guarda su resultado en una caché que invalida los cálculos al recibir señales que involucren cambios en los arcos.

Las operaciones disponibles en la API son:

- **get_transformation_matrix**. Recibe como parámetros los nombres de un nodo de origen y de destino y un timestamp. Los deben formar parte del árbol RT, pero no necesitas estar conectados de manera directa. La

operación devuelve la transformación del nodo destino respecto al nodo origen representada por un objeto RTMat. Para realizar este cálculo, se recorren los arcos RT de manera ascendente, calculando la transformación de cada uno de los nodos respecto a cada nodo de nivel superior hasta llegar al primer antecesor común. El cálculo es costoso en primera instancia, pero gracias al uso de la caché de transformaciones que se actualiza durante todo el proceso cada vez que se calcula una transformación entre nodos. El timestamp se utiliza para el acceso histórico a los valores de los arcos RT.

- **transform**. Recibe como parámetros los nombres de un nodo de origen y de destino, un vector de Eigen de 3 elementos (representando una traslación) y un timestamp. Calcula la transformación entre los nodos y devuelve la posición del vector3 respecto a la transformación.

- **transform_axis**. Recibe como parámetros los nombres de un nodo de origen y de destino, un vector de Eigen de 6 elementos (representando los 3 primeros una traslación y los 3 últimos los ángulos de Euler de la rotación) y un timestamp. Calcula la transformación entre los nodos, calcula la traslación del vector6 respecto a la transformación y la rotación respecto a la transformación. Devuelve un vector de 6 elementos con la rotación y la rotación.

- **get_rotation_matrix**. Recibe como parámetros los nombres de un nodo de origen y de destino, que no necesitan estar conectados de manera directa mendiante un arco RT y un timestamp. Calcula la matriz transformación y devuelve su rotación como una matriz 3x3 en un optional.

- **get_translation_vector**. Recibe como parámetros los nombres de un nodo de origen y de destino, que no necesitan estar conectados de manera directa mendiante un arco RT y un timestamp. Calcula la matriz transformación y devuelve su traslación como un vector de 3 elementos en un optional.

- **get_euler_xyz_angles**. Recibe como parámetros los nombres de un nodo de origen y de destino, que no necesitan estar conectados de manera directa mendiante un arco RT y un timestamp. Calcula la matriz transformación y devuelve su rotación representada por sus ángulos de euler en un vector de 3 elementos en un optional.

B.6. Utils API

La API Utils incluye métodos de lectura y escritura en ficheros JSON de la representación de G. Lo que permite guardar el estado del grafo para reutilizarlo más tarde.

Bibliografía

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

[2] Adrián Romero-Garcés, Luis Vicente Calderita, Jesús Martínez, Juan Pedro Bandera, Rebeca Marfil, Luis J Manso, Antonio Bandera, and Pablo Bustos. Testing a fully autonomous robotic salesman in real scenarios. In *ICARSC*, pages 1–7, 2015.

[3] P. Bustos, L. Manso, J.P. Bandera, A. Romero-Garcés, L. Calderita, R. Marfil, and A. Bandera. A unified internal representation of the outer world for social robotics. In *ROBOT*, volume 2, pages 733–744, 2015.

[4] L.V. Calderita, P. Bustos, C. Suárez Mejías, F. Fernández, R. Viciana, and A. Bandera. 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.

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

[6] 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.

[7] 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. *International Journal of Social Robotics*, pages 16–, 2017.

[8] Antonio Bandera, Juan P. Bandera, Pablo Bustos, Fernando Férnandez, Angel García-Olaya, Javier García-Polo, Ismael García-Varea, Luis J. Manso, Rebeca Marfil, Jesús Martínez-Gómez, Pedro Núñez, Jose M. Perez-Lorenzo, Pedro Reche-Lopez, Cristina Romero-González, and Raquel Viciana-Abad. LifeBots I: Building the software infrastructure for supporting lifelong technologies. In *Advances in Intelligent Systems and Computing*, volume 693, pages 391–402. 2018.

[9] A. Vega-Magro, L. Manso, P. Bustos, P. Núñez, and D.G. Macharet. Socially acceptable robot navigation over groups of people. In *RO-MAN 2017 - 26th IEEE International Symposium on Robot and Human Interactive Communication*, volume 2017-Janua, 2017.

[10] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. Research Report RR-7687, July 2011.

[11] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978.

[12] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. *CoRR*, abs/1410.2803, 2014.

[13] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. *CoRR*, abs/1603.01529, 2016.

[14] GDB contributors. Gdb – the gnu project debugger, 1986-2021. https://www.gnu.org/software/gdb/.

[15] Valgrind Developers. Valgrind, 2000-2021. https://www.valgrind.org/.

[16] Perf contributors. perf: Linux profiling with performance counters, 2009-2021. https://perf.wiki.kernel.org/.

[17] The Qt Company. Qt : cross-platform application and ui framework, 1992-2021. https://doc.qt.io/.

[18] eProsima. eprosima fast dds, 2014-2021. https://github.com/eProsima/Fast-DDS.

[19] Don Burns Robert Osfield and OSG Contributors. Openscenegraph, 1998-2021. http://www.openscenegraph.org.

[20] Ryan Haining. Cppitertools, 2015. https://github.com/ryanhaining/cppitertools.

[21] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[22] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. pybind11 – seamless operability between c++11 and python, 2017. https://github.com/pybind/pybind11.

[23] Sony. Sonyflake - a distributed unique id generator inspired by twitter's snowflake., 2015. https://github.com/sony/sonyflake.