TEC | Tecnológico de Costa Rica

Carrera de Ingeniería Mecatrónica

Diseño de un sistema de visualización para la memoria a largo plazo en la arquitectura cognitiva *Multilevel Darwinist Brain*

Informe de Proyecto de Graduación para
optar por el título de

Ingeniero en Mecatrónica con el grado académico de Licenciatura

Autor
Gabriel Enrique
Barrantes Gómez

Asesor
Dr. Juan Luis Crespo Mariño

21 de Enero, 2021

INSTITUTO TECNOLÓGICO DE COSTA RICA
CARRERA DE INGENIERÍA MECATRÓNICA
PROYECTO DE GRADUACIÓN
DECLARATORIA DE AUTENTICIDAD.

Declaro que el presente Proyecto de Graduación ha sido realizado enteramente por mi persona, utilizando y aplicando literatura referente al tema e introduciendo conocimientos propios.

En los casos en que he utilizado bibliografía, he procedido a indicar las fuentes mediante las respectivas citas bibliográficas.

En consecuencia, asumo la responsabilidad total por el trabajo de graduación realizado y por el contenido del correspondiente informe final.

Cartago, 29 de enero de 2021

Gabriel Enrique Barrantes Gómez
Céd: 305130584

INSTITUTO TECNOLÓGICO DE COSTA RICA
CARRERA DE INGENIERÍA MECATRÓNICA
PROYECTO DE GRADUACIÓN
ACTA DE APROBACIÓN DEL INFORME FINAL

El Profesor Asesor del presente trabajo final de graduación, indica que el documento presentado por el estudiante cumple con las normas establecidas por la Carrera de Ingeniería Mecatrónica, como requisito para optar por el título de Ingeniero en Mecatrónica, con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Estudiante: Gabriel Enrique Barrantes Gómez

Proyecto: Diseño de un sistema de visualización para la memoria a largo plazo en la arquitectura cognitiva Multilevel Darwinist Brain.

JUAN LUIS CRESPO MARIÑO (FIRMA)

Firmado digitalmente por JUAN LUIS CRESPO MARIÑO (FIRMA)
Fecha: 2021.01.19 17:03:39 -06'00'

Dr. Juan Luis Crespo Mariño

Asesor

Cartago, 29 de enero 2021

INSTITUTO TECNOLÓGICO DE COSTA RICA
CARRERA DE INGENIERÍA MECATRÓNICA
PROYECTO DE GRADUACIÓN
ACTA DE APROBACIÓN

Proyecto de Graduación defendido ante el presente Jurado Evaluador como requisito para optar por el título de Ingeniero en Mecatrónica con el grado académico de Licenciatura, del Instituto Tecnológico de Costa Rica.

Estudiante: Gabriel Enrique Barrantes Gómez

Proyecto: Diseño de un sistema de visualización para la memoria a largo plazo en la arquitectura cognitiva Multilevel Darwinist Brain.

Miembros del Jurado

FELIPE GERARDO MEZA OBANDO (FIRMA)
Digitally signed by FELIPE GERARDO MEZA OBANDO (FIRMA)
Date: 2021.01.29 13:42:20 -06'00'

MSc. -Ing. Felipe Meza Obando

Jurado 1

ROGER STUART MELENDEZ POLTRONIERI (FIRMA)
Firmado digitalmente por ROGER STUART MELENDEZ POLTRONIERI (FIRMA)
Fecha: 2021.01.29 16:16:16 -06'00'

Ing. Roger Meléndez Poltronieri

Jurado 2

RODOLFO JOSE PIEDRA CAMACHO (FIRMA)
Digitally signed by RODOLFO JOSE PIEDRA CAMACHO (FIRMA)
Date: 2021.01.29 12:41:33 -06'00'

Ing. Rodolfo Piedra Camacho

Jurado 3

Los miembros de este Jurado dan fe de que el presente trabajo de graduación ha sido aprobado y cumple con las normas establecidas por la Carrera de Ingeniería Mecatrónica.

Cartago, 29 de enero 2021

Resumen

En el presente informe se desarrolla un sistema de visualización para la memoria a largo plazo de la arquitectura cognitiva *Multilevel Darwinist Brain*, desarrollada en el Grupo Integrado de Ingeniería de la Universidade da Coruña. El problema que se solucionará es la falta de una herramienta de visualización para la experimentación en la *MDB*. Se busca presentar una herramienta interactiva, donde se permita la manipulación y análisis de múltiples tipos de informaciónn en la memoria a largo plazo. Para resolver el problema, se inició con un estudio de la bibliografía correspondiente, con la finalidad de comprender los mecanismos de operación de la *LTM*. A continuación, se realizó la selección de conceptos, con la que se definió que la solución se implementaría como una página web desarrollada con *Python* y *Dash*. En cuanto a la transferencia de datos entre la herramienta y la *LTM*, se definió el sistema de visualización como un nodo *ROS*. Esto derivó la definición de una serie de mensajes y canales de comunicación para cada tipo de dato necesario de la *LTM*. Finalmente, se realizó una validación para la página web, con datos de prueba dados por el cliente, y una validación de la integración con *ROS* mediante la experimentación con un robot Baxter.

Palabras clave— memoria a largo plazo, MDB, robótica cognitiva, ROS, visualización

Abstract

The present document details the development of a visualization system for the long term memory of the cognitive architecture *MDB*, developed by the Integrated Research Group of the University of A Coruña. The problem that will be solved is the lack of a visualization tool for experimentation in the *MDB*. The tool should be interactive, and allow the manipulation and analysis of multiple types of information in the long term memory. To solve this problem, a study of the relevant bibliography was conducted, in order to understand the underlying mechanisms of the *LTM*. Them, a concept selection was developed, which derived in the decision of implementing the system as a web application that would be developed in *Python* with *Dash*. For the data transference between the tool and the *LTM*, it was proposed the implementation of the system visualization as a ROS node, just like the *LTM*. This derived in the selection of messages and topics for each data type required from the *LTM*. Finally, a validation for the web application was conducted, with test data defined by the client, and a validation of the integration with *ROS* was conducted by experimenting with a Baxter robot.

Keywords— cognitive robotics, long term memory, MDB, ROS , visualización

DEDICATORIA

Dedicada a mi familia, el más grande regalo que la vida me ha dado

AGRADECIMIENTOS

En primer lugar, agradezco a mi profesor tutor, el Dr. Juan Luis Crespo Mariño, por la guía que me dió durante el desarrollo del proyecto, las enseñanzas que logró transmitirme en clases y por abrirme las puertas para trabajar como asistente del Laboratorio de Inteligencia Aritficial para las Ciencias Naturales. Agradezco la disponibilidad y acceso con el que siempre me recibió.

Al Dr. José Antonio Becerra Permuy, por permitirme desarrollar el proyecto para el Grupo Integrado de Ingeniería de la Universidade da Coruña, por su asesoría en el desarrollo del proyecto y por la atención con la que siempre me recibió cuando requería de su ayuda.

Al Ing. Ronald José Loaiza Baldares, por su acompañamiento en el desarrollo del proyecto, por su ayuda a encontrar este proyecto, por abrirme las puertas para trabajar con él en el LIANA, pero más importante por su amistad.

A todos los profesores de los que aprendí durante mi carrera, en especial a los de Ingeniería Mecatrónica, por hacer crecer la carrera día con día.

A todos mis compañeros de carrera, por sobrepasar juntos los momentos donde los cursos nos demandaban en exceso, por los momentos felices y todo lo que me enseñaron. En especial a Diego Jiménez Ulate, por ser el mejor compañero con el que pude contar, por abrirme las puertas de su casa y en especial por su amistad.

Finalmente agradezco a mi familia. Por amarme y apoyarme siempre en todo lo que hago, por motivarme a seguir adelante y por estar siempre a mi lado, sin importar la situación.

Agradezco a mi mamá Aracelly, por darme la vida, por el amor y apoyo incondicional que siempre me ha dado a mí y a mi hermana, y por el gran esfuerzo que día con día hace para que no nos falte nada.

Agradezo a mi abuela Yolanda, mi segunda madre, por cuidarme desde siempre, por

ponerme siempre en sus oraciones, por la comida que con mucho amor me prepara a diario y por quererme como un hijo.

Agradezco a mi tío Gustavo, por ser la voz de la sensatez que tanto llegué a necesitar en los momentos más difíciles de la carrera, por hacerme reír y por estar ahí siempre dispuesto a escucharme.

Finalmente, agradezco a mi hermana Rebeca, por alegrar mis días y motivarme a dar lo mejor de mí.

CONTENIDO

LISTA DE FIGURAS

LISTA DE CUADROS

ACRÓNIMOS

dcc Dash Core Components

DOE Design of Experiments

FM Forward Model

GII Grupo Integrado de Ingeniería de la Universidad de la Coruña

html Dash HTML Components

LTM Memoria a largo plazo (*Long-Term Memory*)

MDB Multilevel Darwinist Brain

ROS Robot Operating System

STM Memoria a corto plazo (*Short-Term Memory*)

VF Value Function

CAPÍTULO 2

INTRODUCCIÓN

El proyecto presentado en este documento se desarrollará para el Grupo Integrado de Ingeniería de la Universidade da Coruña, España. Creado en 1999, este es "un grupo interdisciplinar de investigación aplicada en ingeniería, orientado a la transferencia de conocimiento y a la generación de nuevos productos en el entorno industrial." [1]

Entre las líneas de investigación que se desarrollan allí se encuentran el diseño naval y oceánico, la dinámica de fluidos computacional, inteligencia computacional y robótica y cognición. Esta última línea tiene como objetivo general desarrollar sistemas reales que puedan operar con un grado de autonomía tal que puedan ser capaces de responder a las condiciones cambiantes del medio sin tener que precisar de la ayuda de un operador. El *Multilevel Darwinist Brain* es una arquitectura cognitiva desarrollada en esta línea de investigación desde el año 2000 y se ha aplicado en diversas tareas de aprendizaje con robots como el *Hermes, Pioneer 2DX, Aibo* y *Baxter*. [2]

Dentro de la línea de investigación de Robótica y Cognición del GII se encuentra la línea derivada de mecanismos cognitivos. En ella se desarrolla una arquitectura para robots autónomos llamada Multilevel Darwinist Brain (*MDB*), que utiliza algoritmos

evolutivos y redes neuronales para el aprendizaje de modelos que depende de muestras reales que el robot adquiere de la interacción con el mundo y un sistema de memorias con una a corto plazo y otra a largo plazo. La memoria a largo plazo, el objeto de estudio del presente proyecto, se compone de nodos de conocimiento que se crean durante el aprendizaje del robot. La visualización de estos nodos y la información que se genera en el proceso es de gran importancia para extraer conclusiones sobre el rendimiento de la memoria. Sin embargo, actualmente la herramienta de visualización no presenta la información relevante en su totalidad y se desarrolla desde la *LTM*.

El problema que presentan es que la herramienta de visualización se implementa desde *LTM*, cuando debería haber un desacople de esta actividad para desarrollarla en otro módulo de la *MDB*, y ejecutar en la *LTM* únicamente el desarrollo del conocimiento que le permita a un agente desarrollar una tarea. La presentación parcial de los datos es otro problema, pues no se puede ver información de la red al mismo tiempo que se ven datos propios de algunos nodos de conocimiento. La única forma de visualizar estos datos es a través de un procesamiento posterior a la ejecución de un experimento.

Por esta razón, el desarrollo de una herramienta de visualización, que funcione para monitorizar múltiples datos de la *LTM* de forma simultánea, le permitirá a los investigadores aumentar su productividad científica al permitirles plantear y comprobar hipótesis en el experimento con datos actualiazos en tiempo prácticamente real.

2.1 Objetivos

2.1.1 Objetivo General

Diseñar una herramienta para la visualización en tiempo real de la arquitectura cognitiva Multilevel Darwinist Brain implementada en un robot industrial.

2.1.2 Objetivos Específicos

- Diagnosticar el sistema de visualización actualmente en funcionamiento en la arquitectura cognitiva. mediante una búsqueda bibliográfica de las publicaciones derivadas de la MDB y sistemas de visualización.
- Diseñar un sistema de visualización que permita observar la evolución de la arquitectura, interactuar con sus nodos y pausar o reanudar el proceso.
- Implementar el sistema de visualización junto con el robot Baxter y ROS.
- Validar el sistema de visualización en la simulación con ROS y en el robot real.

Al finalizar el proyecto, se habrá creado un sistema de visualización para ser aplicada en una aplicación robótica como lo es la memoria a largo plazo de la *MDB*. Dicho sistema será construido como una página web que se comunica con la *LTM* por medio de *ROS*, que se caracteriza por tener un alto componente de interactividad. Se entregará a los investigadores del Grupo Integrado de Ingeniería, una interfaz gráfica de usuario base, con la que se podrá interaccionar con la *LTM* y a la que se le puede añadir nuevas funcionalidades en un futuro.

2.2 Estructura del documento

En el capítulo 3 se abordan los conecptos teóricos más relevantes para el desarrollo del proyecto. En el 4 se describe la metodología seguida para solucionar el problema planteado. En el capítulo 5 se explica el proceso de diseño ejecutado para desfarrollar le herramienta de visualización. En el capítulo 6 se muestran y analizan los redsultados obtenidos del proceso de diseño y en el 7 se presentan las conclusiones y recomendaciones del proyecto.

CAPÍTULO 3

MARCO TEÓRICO

En el presente capítulo se abordarán los conceptos teóricos necesarios para comprender el desarrollo del proyecto. En la sección 3.1 se detalla el tema de arquitecturas cognitivas. En la sección 3.2 se hace una revisión bibliográfica de la *MDB*, se explican algunas características de esta arquitectura y el funcionamiento de la *LTM*. En la sección 3.3 se abordan aspectos relevantes a la visualización de redes como el proceso de visualización de redes y algoritmos relevantes. En la sección 3.4 se detallan aspectos de diseño de experimentos, necesarios para la validación del proyecto. En la sección 3.5 se explican detalles del desarrollo de software, necesarios para el diseño de la solución.

3.1 Arquitecturas cognitivas

Para comprender el estudio de las arquitecturas cognitivas es necesario primeramente definir los términos de cognición y de ciencias cognitivas. Según [3] la cognición engloba las funciones mentales, como lo son la percepción, aprendizaje, memoria y pensamiento; con las cuales se obtiene, almacena y se usa el conocimiento. Otra definición propuesta

por [4] es que la cognición trata sobre procesos conductuales que permiten adquirir información sensorial, tomar decisiones y producir acciones que les permite interaccionar con el entorno dinámico en el que se encuentran. Por otro lado, las ciencias cognitivas se pueden definir como el estudio interdisciplinario de la mente y de la inteligencia. Abarca áreas como filosofía, psicología, inteligencia artificial, neurociencia, lingüística y antropología [5].

Kotseroba y Tsotsos [6] explican que que el estudio de las arquitecturas cognitivas forma parte de la investigación en inteligencia artificial, cuyo comienzo se puede definir en la década de 1950 y que tiene como objetivo crear programas que puedan razonar sobre problemas en diversos dominios, desarrollar percepciones y adaptarse a nuevas situaciones.

Para una definición de arquitectura cognitiva, se puede ver el trabajo de Sun [7], donde la describe como "un modelo cognitivo computacional genérico de dominio amplio, que captura la estructura y los procesos esenciales de la mente, para ser utilizado para un análisis de comportamiento amplio, de múltiples niveles y múltiples dominios". Por otro lado, Duro et al. [8] detallan que la arquitectura cognitiva es una estructura operacional compuesta de elementos de procesamiento que no son específicos para un dominio ni tarea en específico, sino que pueden adaptarse a cualquier tarea o dominio a través de la adquisición de conocimiento.

De este modo, se puede introducir el concepto de *robótica cognitiva* como el campo de estudio enfocado en dotar a un robot (o agente) con la capacidad de razonar, desarrollar soluciones y *ejecutarlas*, en un ambiente cambiante [9]. Duro et al. [8] expanden sobre este concepto, indicando que el propósito de una arquitectura cognitiva es dotar a un agente *motivado*, como lo puede ser un robot, con los medios para escoger acciones que le permita cumplir con sus objetivos.

Es importante destacar que debido a la falta de una definición clara y de una teoría general sobre la cognición, el diseño de arquitecturas cognitivas se puede basar en

una serie de premisas y supuestos específicos sobre la cognición [6]. Por esta razón, es difícil clasificar y comparar el progreso y el funcionamiento de las diversas arquitecturas cognitivas que existen, Para propuesta de clasificación se puede ver [6], donde se dividen las arquitecturas en tres tipos principales según la representación que se emplee para el conocimiento adquirido: emergentes, simbólicas o híbridas.

3.2 Multilevel Darwinist Brain

El Multilevel Darwinist Brain, o *MDB* por sus siglas en inglés, es una arquitectura cognitiva desarrollada en el Grupo Integrado de Ingeniería de la Universidad de la Coruña. El diseño del *MDB* se basa en la robótica desarrollativa [10], que se fundamenta en modelos de desarrollo de la cognición humana. De este modo, a un robot se le dota con una arquitectura cognitiva que, empezando con un conocimiento innato dado por el diseñador, sea capaz de generar nuevo conocimiento de manera autónoma a lo largo de toda su vida [11]. De este modo el robot aprenderá de forma abierta, es decir se enfrentará a una cantidad ilimitada de tareas desconocidas en una cantidad ilimitada de entornos desconocidos.

Según Duro et al. [12] el *MDB* se compone principalmente de las siguientes estructuras,

- **Modelos:** estructuras de predicción en forma de modelos de mundo o forward models (FM). El modelo de mundo es el FM que dado un estado y una policy, permite determinar el siguiente estado [8].

- **Policy:** es una estructura de decisión que debe ser aprendida y que define una acción que se ejecutará en el tiempo $t+1$ según las entradas sensoriales en t.

- **Episodios:** son muestras del mundo real que se adquieren de los sensores y actuadores tras ejecutar una acción, además contienen la acción aplicada en un tiempo t y los niveles de satisfacción en $t+1$ tras la ejecución de la acción.

- **Memorias:** contiene dos estructuras de memoria: una memoria a corto plazo STM y una memoria a largo plazo LTM. La primera contiene modelos útiles para la tarea actual y un buffer de episodios donde almacena los episodios recientes. La segunda almacena modelos que se han consolidado por su relevancia y el comportamiento asociado.

Dos mecanismos de decisión importantes para el MDB deben ser definidos: prospección y experiencia. Por prospección se entiende al proceso de predicción de estados futuros para que puedan ser evaluados, y así escoger la mejor acción o policy que permite el logro del objetivo de la arquitectura [8].

El funcionamiento principal de la LTM se basa en que cualquier nodo de conocimiento nuevo, ya sea que provenga de la misma memoria o de algún otro componente de la arquitectura, se puede conectar con otros nodos si co-ocurren en una situación relevante.

Ahora bien por experiencia se entiende el proceso que consiste en definir la acción o policy más adecuada basándose en una serie de asociaciones que se han hecho cuando la arquitectura ha sido exitosa en el alcance de su objetivo [8].

3.2.1 Memoria a Largo Plazo

La memoria a largo plazo, o LTM por sus siglas en inglés, debe cumplir con dos objetivos principales. El primero es el de almacenar elementos de conocimiento adquiridos por el sistema. Estos elementos se denominan nodos de conocimiento y corresponden a Forward models FM, goals (G_k) y sus respectivas Value Functions (VF), policies (π_k), clases perceptuales ($PNodes$) y clases contextuales ($CNodes$) [12].

En segundo lugar debe ser capaz de almacenar relaciones de contexto entre los nodos de conocimiento recién detallados [12]. Este último objetivo busca desarrollar un nivel de toma de decisiones basado en la experiencia.

En la figura 3.1 se representa una estructura de la LTM. En primer lugar, se tiene

que la capa más a la izquierda y que se representa en color azul, corresponde a sensorizaciones elementales, como lo pueden ser la imagen de una cámara RGB o un sensor de fuerza en una articulación de un robot. A continuación, estas sensorizaciones son redescritas. Por ejemplo, la imagen RGB se puede redescribir de modo que represente la distancia y el ángulo de un objeto con respecto a la cámara.

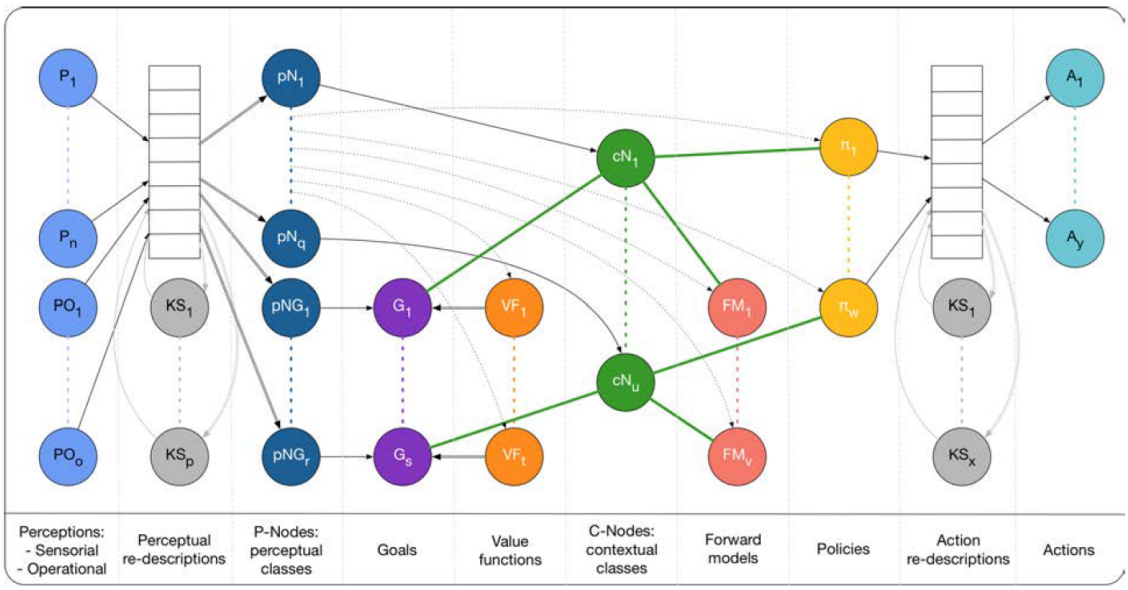

Figura 3.1: Diagrama de la estructura actual de la LTM. Recuperado de [8]

Las metas o *Goals*, son nodos que representan un objetivo que el agente donde se implementa la *LTM* debe cumplir.

Los nodos de la tercera capa son los *PNodes*, que son componentes funcionales que determinan si un conjunto de percepciones (o estado perceptual) pertenece a una clase perceptual. Ahora bien, cada *PNode* está asociado a un *CNode*. Esta estructura permite definir contextos, que son asociaciones de metas (*Goals*), modelos de mundo (*Forward Models*) y *Policies*. El propósito de estos contextos es que si el agente se encuentra en un contexto ya conocido o muy similar, las activaciones de los nodos de este contexto se propagarán, dando lugar a una *policy* más activa, que representa la acción más adecuada para la situación presente, según la experiencia previa del agente. [8]

Finalmente, la *policy* más activa sigue una redescripción que da lugar a una ac-

ción del robot. Por ejemplo, una *policy* puede ser el levantar un objeto, con lo que la redescripción consistirá en definir las acciones para mover el brazo al punto donde se encuentra el objeto, bajar el brazo, abrir el *gripper*, etc.

3.2.1.1 Puntos y antipuntos

Una percepción que pertence a un *PNode* se puede representar como un punto de activación (o simplemente punto) o un punto de inhibición (o antipunto). Los puntos de activación son puntos que el sistema experimentó en el espacio perceptual y en el que el *PNode* debe tener una activación de 1. Por el contrario, el punto de inhibición, corresponde a un punto del espacio perceptual donde el *PNode* debe tener una activación de -1.

Los puntos y antipuntos sirven como el *ground truth* para determinar la activación del *PNode* según las percepciones actuales. La activación puede calcularse calculando la distancia al punto o antipunto más cercano, o puede ser la salida de una red neuronal. Esto quiere decir que la representación de un *PNode* puede variar.

La figura 3.2 muestra los mapas de activación para un *PNode* en diferentes iteraciones del experimento, así como los mapas de activación resultantes.

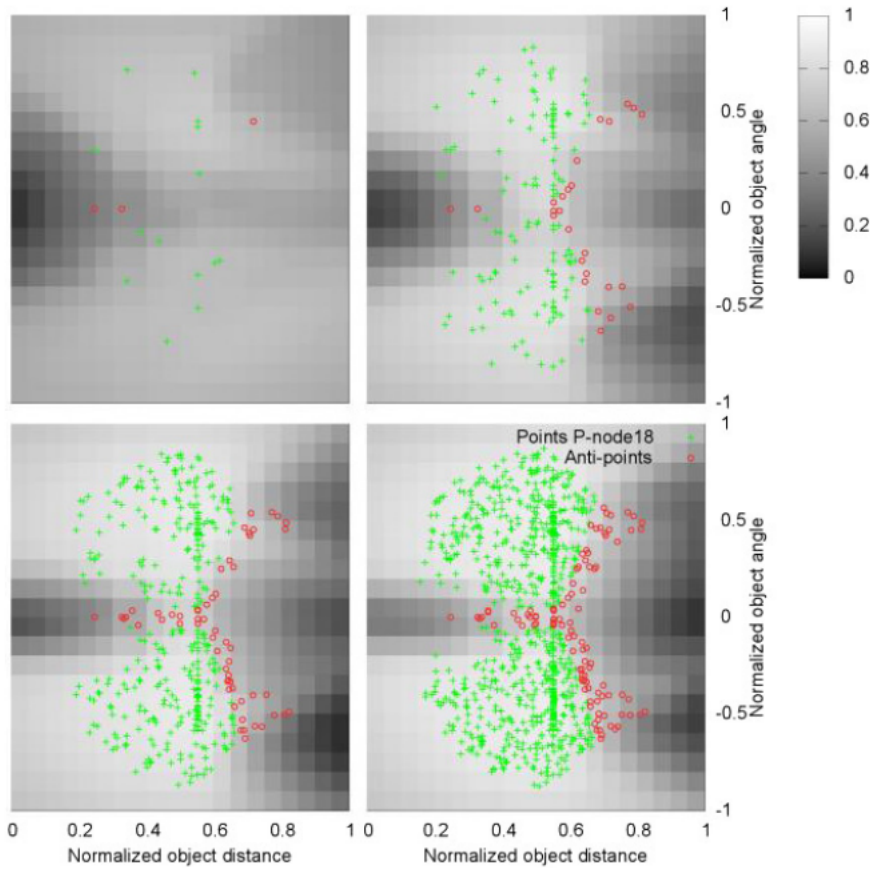

Figura 3.2: Mapas de activación y puntos y antipuntos de un *PNode* para diferentes iteraciones. Recuperado de [8]

3.2.1.2 Creación de *CNodes* y *PNodes*

Los *CNodes* y *PNodes* se crean en parejas. Como se mencionó previamente, los primeros sirven para determinar contextos que el agente aprende por la interacción con el mundo.

A la hora de ejecutar una *policy*, por cualquier mecanismo (puede ser aleatorio si el agente no tiene conocimiento previo), el agente recibe una recompensa. Esta depende de la meta y del modelo de mundo actual. Por ejemplo, si una meta es colocar una bola dentro de una caja, pero el robot la lanza lejos, la recompensa es baja. Si por el contrario, introduce la bola dentro de la caja, la recompensa es alta.

Al inicio de un experimento, la *LTM* puede no tener información de experiencias

pasadas, es decir, no tiene *CNodes* ni *PNodes* cargados previamente. Cuando sucede esto, el primer par de nodos se creará al recibir la primera recompensa. [8]

En general, el proceso de creación de *CNodes* y *PNodes*, siempre empieza tras determinar la recompensa tras ejecutar una *policy*. Si la recompensa es menor a un valor umbral, se buscan los *CNodes* conectados a la *policy* que dio lugar a la recompensa, se buscan los *PNodes* conectados a estos y se agrega la percepción previa como un antipunto a dichos *PNodes*. Si, por el contrario, la recompensa es mayor al umbral la percepción previa se añade como un punto.

Ahora bien, si no hay ningún *CNode* conectado a la *policy*, se crea uno nuevo que se conectará al modelo de mundo más activa, la meta más activa, la *policy* que dio lugara a la recompensa y a un nuevo *PNode* al que se le agregará la percepción previa como un punto o un antipunto en función de la recompensa.

3.3 Visualización de redes

Una red o grafo *G = (V, E)* es una estructura de datos que consta de un conjunto de vértices o nodos V y un conjunto de aristas $E \subseteq \{(u, v) \,|v \in V, u \neq v\}$ que permite representar las relaciones existente entre nodos [13].

La visualización de redes en dos dimensiones abarca campos como el dibujo de circuitos, animación de algoritmos, diagramas en ingeniería de software, representación de redes sociales, redes biológicas, redes de dispositivos, entre otros. En estos casos, el estándar gráfico por defecto es el de representar mediante algún símbolo (círculos, cuadrados, triángulos, etc) cada nodo y las aristas como curvas abiertas que unen los nodos [14]. A este estándar también se le conoce como el diagrama nodo-enlace, y es el estándar que se empleará en el presente proyecto. Otro tipo de representación de grafos es el basado en la matriz de adyacencia, sin embargo esta no refleja atributos propios de cada nodo [13].

Battista et al. [14] definen los siguientes criterios estéticos como medidas cuantitavias, para promover que la representación gráfica de un grafo pueda ser interpretada fácilmente.

- Minimizar el número de cruces entre aristas
- Minimizar el número de dobleces en las aristas
- Minimizar la longitud de las aristas
- Minimizar la superposición de los nodos
- Minimizar el área de dibujo
- Maximizar la simetría

El seguimiento de estos criterios, que mejoran la visualización del grafo, permite brindar un mayor soporte cognitivo a quien esté analizándolo. Por ejemplo, el hecho de minimizar cruces, número de dobleces y la longitud de las aristas disminuyen el número de operaciones perceptuales, pues organizan el grafo de una forma más eficiente. Además, según Huang et al. [15] la visualización efectiva de un grafo puede servir como una memoria externa, disminuyendo así las demandas cognitivas en la persona. También indica que la visualización permite revelar patrones que pueden permanecer escondidos en otras formas de representación más abstractas.

3.3.1 Proceso de visualización

Gómez-Romero et al. [16] definen las siguientes 5 etapas para la visualización de información en redes:

1. **Recuperación de la información:** etapa que consiste en obtener la información que pertenece a la red.

2. **Construcción de la red:** tras recuperar la información que pertenece a la red, se construye un grafo en memoria o como un archivo de texto. En esta etapa se busca definir la red en un formato con el que la máquina pueda procesarlo. Para esta etapa se puede usar librerías orientadas a la construcción y análisis de redes, como lo son *Networkx*, *igraph* o *graph-tools*.

3. **Cálculos de la red:** en esta etapa se puede realizar diversos cálculos con los datos de la red, con la finalidad de mejorar su visualización. Por ejemplo se puede determinar, tanto para los nodos como para las aristas, el color, grosor o tamaño en función de una métrica definida.

4. **Cálculo de la disposición de la red:** en esta etapa se calcula las posiciones de los nodos para que su visualización sea fácil de interpretar. En la sección 3.3.2 se especifican algunos algoritmos que realizan este cálculo.

5. **Rendering:** etapa final donde algún software dedicado a la visualización de redes (Gephi [17] , Cytoscape [18], entre otros) se encarga de traducir comandos de alto nivel en instrucciones que se envían a la tarjeta gráfica para desplegar el grafo en pantalla.

3.3.2 Algoritmos de visualización

Un algoritmo de visualización de redes se encarga de extraer información de un grafo almacenado en memoria y retornar una representación gráfica de la red según un estándar definido [14].

Los algoritmos presentados a continuación se emplean en grafos no dirigidos y siguen el estándar de visualización donde las aristas son líneas rectas.

3.3.2.1 Algoritmo de Eades

Eades [19] definió un algoritmo para calcular las posiciones de los nodos de un grafo, al considerarlos como anillos con la misma carga eléctrica y las aristas como resortes, de manera que los nodos se inicializan en cualquier posición y el sistema se libera hasta alcanzar un estado de energía mínima.

Una característica del algoritmo de Eades es que no concordaba con la ley de Hooke, sino que definió su propia fórmula para la fuerza en los resortes. Además, a diferencia de lo que se observa en la naturaleza, la atracción se da solo entre nodos que se encuentran conectados por una arista.

3.3.2.2 Algoritmo de Fruchterman-Reingold

El algoritmo de Fruchterman-Reingold se basa en dos supuestos: los vértices conectados por una arista deben atraerse entre sí y aquellos que no, no deben dibujarse muy cerca entre sí [20].

Este algoritmo se basa en el funcionamiento del algoritmo de Eades, al modelar los nodos como partículas atómicas o cuerpos celestes que ejercen fuerzas de atracción y repulsión entre sí. De este modo se cumple con los dos supuestos mencionados en el párrafo anterior [20].

La clave para diferenciar este algoritmo del de Eades es que Fruchterman y Reingold usan las fuerzas para calcular *velocidades* para cada intervalo de tiempo, contrario a la naturaleza donde las fuerzas inducen *aceleración* [20].

La velocidad calculada de la fuerza genera un desplazamiento del nodo, cuyo valor no puede exceder un valor límite que Fruchterman y Reingold denominan como *temperatura*. Este valor desciende conforme la visualización del grafo mejore [20].

El algoritmo se inicializa colocando los nodos en posiciones aleatorias. Luego por un número de iteraciones se procede a repetir las siguientes 3 etapas principales: calcular las fuerzas atractivas en cada nodo, luego calcular las fuerzas repulsivas y finalmente

limitar el desplazamiento del nodo por la temperatura.

3.3.2.3 Algoritmo de Kamada-Kawai

Similar al algoritmo de Fruchterman-Reingold, el algoritmo de Kamada-Kawai se basa en el algoritmo de Eades. En este caso, se asume que entre cada nodo se ha conectado un resorte cuya longitud se calcula con el algoritmo de Floyd.

A continuación se inicializan los nodos en los vértices de un polígono regular cuyo número de lados es igual al número de nodos en el grafo y que está circunscrito en un círculo cuyo diámetro también es calculado por el algoritmo [21].

Luego, se disminuye la energía del sistema (recordando la analogía con los resortes) paso a paso, hasta que cada nodo haya alcanzado una posición estable, es decir cuando se haya minimizado la energía.

3.4 Diseño de experimentos

El diseño de experimentos (o *DOE* por sus siglas en inglés) corresponde al proceso de planificar, diseñar y analizar un experimento que busca obtener, de manera efectiva y eficiente, conclusiones válidas y efectivas [22].

Juristo y Moreno [23] definen una serie de conceptos relvenates al *DOE*, a continuación se detallan algunos que serán importantes para el diseño de los experimentos que se emplearán para validar el presente proyecto.

- **Unidad experimental:** también conocido como objeto experimental, es el objeto sobre el cual se ejecutará el experimento. En el caso de un proyecto de software, la unidad experimental puede ser todo el proyecto en conjunto o subsistemas que se van desarrollando progresivamente en el proyecto.

- **Sujeto experimental:** la persona que aplicará los métodos y técnicas sobre la unidad experimental. En el desarrollo de software, el sujeto experimental por lo

general es el mismo desarrollador o el cliente.

- **Variables de respuesta:** conocidas también como variables dependientes, y son las salidas producidas por los cambios efectuados sobre los factores de la unidad experimental.

- **Parámetros:** son aquellas características en el proyecto de software que se mantienen invariables en la experimentación. Son características que no afectan el resultado del experimento ni las variables de respuesta.

- **Factores:** también conocidas como variaciones provocadas, son aquellas características que son modificadas con el propósito de obtener un cambio en las variables de respuesta.

- **Alternativas y niveles:** los posibles valores que pueden tomar los factores en cada experimento individual.

- **Interacciones:** se dan cuando el efecto de un factor depende del valor de otro. El estudio de estas interacciones es de suma importancia para comprender cómo los factores afectan las variables de respuesta.

En el diseño de experimentos se realizan cambios intencionales sobre variables o factores para evaluar el efecto que provocan sobre variables de respuesta [22]. La importancia del *DOE* es que permite extraer información importante sobre la unidad experimental, al brindar una guía al experimentador que le permita obtener conclusiones con la mínima cantidad de trabajo, tiempo dinero, o algún otro tipo de recurso limitado.

Jiju [24] señala que tres conceptos son claves para evitar los sesgos experimentales. El primero de ellos es la *aleatoriedad*, que se utiliza para simular los efectos que el ruido en los factores tiene sobre la unidad experimental que se está analizando.

El segundo concepto es el de *replicación*. Este consiste en ejecutar el experimento en más de una condición, esto con el propósito de analizar las variaciones que introducen factores como lo son personas, materiales o máquinas. La replicación tiene 3 propiedades importantes: permite estimar el error experimental, permite obtener con mayor precisión del efecto de las interacciones o de los factores y permite además reducir el error experimental [24].

El tecer y último concepto es el de *bloqueo*, que busca disminuir las variaciones debidas al ruido en los factores. Para ello se separan en **bloques** corridas del experimento que sean homogéneas entre sí, de modo que las observaciones recogidas de bajo las mismas condiciones experimentales pertencene al mismo bloque.

3.5 Desarrollo de software

De acuerdo con Sommerville [25], el desarrollo de software, sin importar el tipo de modelo que se utilice, consta de 4 etapas principales:

- *Especificación del software:* etapa donde se definen los requerimentos del software, así como sus limitaciones.
- *Diseño e implementación del software:* desarrollo del software para cumplir con los requerimentos definidos.
- *Validación del software:* ejecución de pruebas en el software para garantizar que se cumple con las necesidades del cliente.
- *Evolución del software:* adaptación del software según los cambios en las necesidades del cliente.

A continuación se detalla el modelo de desarrollo de software de validación y verificación o modelo V y el modelo de desarrollo incremental, utilizados en el capítulo 4 para definir la metodología seguida.

3.5.1 Modelo V

El modelo V se desarrolló para integrar la etapa de análisis y desarrollo con la etapa de pruebas y validación tal y como se observa en la figura 3.3, donde la fase de análisis se observa en la línea superior y la fase de desarrollo e integración se observa en la línea inferior.

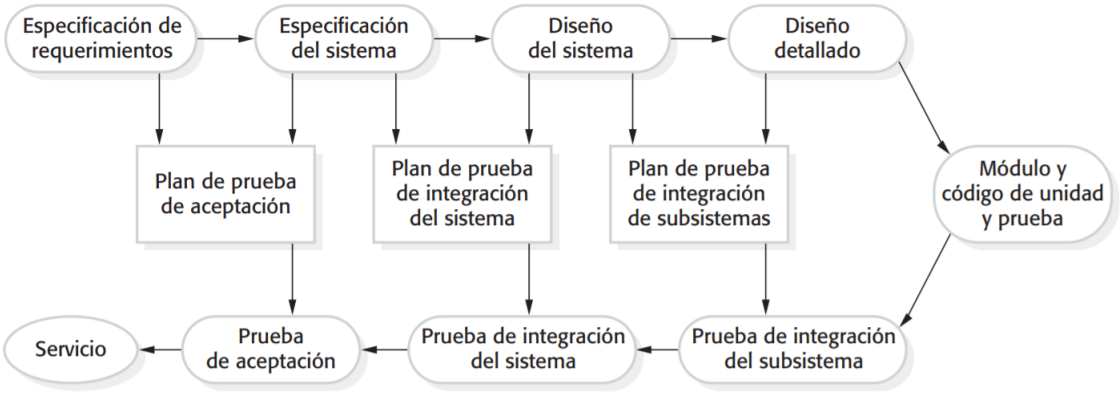

Figura 3.3: Modelo V de desarrollo de software. Recuperado de [25]

La ventaja que provee el modelo V, es que para cada etapa de desarrollo se tiene un plan de prueba que permite detectar errores en etapas tempranas del proyecto, minimiza errores futuros y garantiza que el proyecto se adhiera a las especificaciones del cliente [26].

3.5.2 Modelo de desarrollo incremental

Este modelo se basa en el desarrollo de una versión inicial del software, exponerla al cliente, recibir sus comentarios y luego desarrollarlo de forma incremental, agregando funcionalidad al software en función de los comentarios del cliente, hasta lograr un sistema adecuado [25]. En este modelo, las etapas de especificación, desarrollo y validación se encuentran entrelazadas, tal y como se muestra en la figura 3.4.

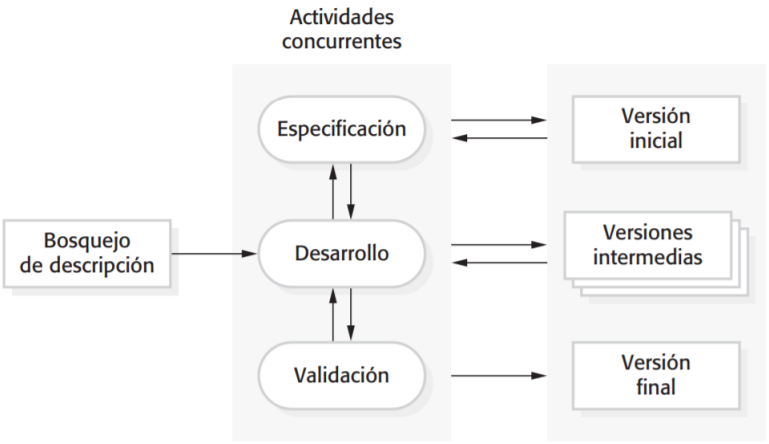

Figura 3.4: Diagrama del modelo incremental. Recuperado de [25]

3.5.3 Pruebas en el desarrollo de software

Las pruebas en el desarrollo de software son esenciales para la etapa de validación y verificación del proyecto. Estas permiten determinar si el programa desarrollado cumple con las especificaciones y expectativas del cliente. También cumplen con el objetivo de encontrar situaciones o comportamiendos específicos que no son correctos para las especificaciones del proyecto, con la finalidad de realizar las correcciones pertinentes.

Sommerville [25] señala los siguientes tipos de pruebas.

- **Pruebas de desarrollo**

  Estas pruebas son todas las actividades de prueba que el desarrollador realizó para construir el software. En estas pruebas se pueden encontrar las pruebas de unidad, pruebas del componente y pruebas del sistema [25].

  - *Pruebas de unidad:* se ponen a prueba unidades del programa o clases de objetos individuales. Se enfocan en verificar la funcionalidad de los objetos y los métodos previo a la integración con otros. Algunas tareas que forman parte de estas pruebas son verificar todas las operaciones asociadas al objeto o poner el objeto en todos los estados posibles.

- *Pruebas de componente:* se ponen a prueba componentes complejos que surgen de la integración de más de una unidad. Se enfocan principalmente en probar las interfaces ya sea de memoria compartida, de procedimiento, de parámetro o que pasan mensajes.
- *Pruebas del sistema:* son las pruebas que se desarrollan que los componentes creados son compatibles, interaccionan de forma correcta y que la transferencia de datos es correcta.

- **Pruebas de versión**

  Se emplean con la finalidad de probar versiones particulares del sistema. Por lo general la versión del sistema es para los clientes o usuarios, o para otros equipos que desarrollan sistemas con los que se relaciona. El objetivo de este tipo de pruebas es convencer al cliente que el sistema es apto para ser entregado. Dentro de estas pruebas se encuentran las pruebas basadas en requerimientos, pruebas de escenario y pruebas de rendimiento [25].

  - *Pruebas basadas en requerimientos:* son pruebas enfocadas a la validación y no a encontrar defectos. Es un enfoque en el que se comprueba que el sistema cumpla con los requerimientos.
  - *Pruebas de escenario:* en este tipo de pruebas se desarrollan casos basados en los escenarios sobre los cuales podría operar el sistema en el entorno final.
  - *Pruebas de rendimiento:* pruebas que consisten en verificar que el sistema diseñado cumpla con la carga pretendida. Por lo general, implican aumentar la carga sobre el sistema gradualmente, hasta que este no sea aceptable. De este modo se garantiza que el programa cumple con las especificaciones y se detectan posibles fuentes de error.

- **Pruebas de usuario**

  Pruebas que consisten en probar el sistema de acuerdo a la asesoría del cliente

o usuario final. Se distinguen tres tipos de pruebas de usuario, la alfa, beta y de aceptación [25].

- *Pruebas beta:* se pone a disposición del usuario final una versión del software, esto con el fin de que experimenten y encuentren problemas que los desarrolladores obviaron.
- *Pruebas alfa:* los usuarios trabajan con el equipo de diseño para probar el software desarrollado.
- *Pruebas de aceptación:* el cliente define si esl sistema está listo para ser aceptado por el desarrollador y distribuirlo al cliente o usuario final.

CAPÍTULO 4

MARCO METODOLÓGICO

En el presente capítulo se aborda la metodología utilizada para el desarrollo del proyecto descrito en este documento. Por la naturaleza del proyecto, se plantea utilizar un modelo de desarrollo de software claramente establecido en la bibliografía correspondiente.

4.1 Metodología seguida

Las etapas definidas para el desarrollo del proyecto se basan en el modelo V de desarrollo de software detallado en la sección 3.5.1. Dichas etapas se detallan a continuación:

1. **Identificación del problema**: esta etapa consiste en determinar la problemática que se desea resolver. En el presente proyecto el problema por resolver consiste en la falta de un sistema integrado con la *LTM* que permita a los investigadores del *GII* tener mayor control e interactividad sobre ella.

2. **Identificación de necesidades**: etapa donde se aborda el problema con el cliente (*GII*), con la finalidad de definir los requerimientos específicos que el sistema por desarrollar debe cumplir. Los resultados de esta etapa se deben reportar en un documento de requerimientos. Los resultados de esta etapa están sujetos a cambios, al seguir un modelo de desarrollo incremental, conforme se desarrolle versiones y estas sean presentadas al cliente, los requerimientos pueden aumentar o cambiar.

3. **Revisión bibliográfica**: etapa donde se realiza un estudio de la bibliografía derivada de la arquitectura cognitiva Multilevel Darwinist Brain, de la *LTM* y de sistemas de visualización. La finalidad de esta etapa es comprender en detalle el funcionamiento de la *LTM* y realizar un estudio de las herramientas de software disponibles para la solución del problema.

4. **Diagnóstico**: esta etapa consiste en analizar los componentes de visualización utilizados previo a la realización de este proyecto. Esto se hará para determinar los procesos que pueden mantenerse, así como los que pueden ser mejorados o desechados.

5. **Estrategia de generación de datos**: esta etapa consiste en definir una estrategia de generación de datos para realizar el programa de visualización previo a la integración de con la *LTM*.

6. **Diseño del sistema**: tras el estudio sobre sistemas de visualización se deberá definir el lenguaje de programación por utilizar, así como las bibliotecas necesarias tanto para la construcción de grafos como para su visualización. Además se debe definir el flujo del programa, así como las clases y funciones para la visualización e interacción con el programa.

7. **Diseño detallado**: en esta etapa se redefine el flujo del programa, las funciones y

clases en función del lenguaje de programación, la biblioteca para la construcción de grafos y para su visualización.

8. **Módulo y código de unidad y prueba**: En esta etapa se empleará el modelo de desarrollo incremental para el desarrollo del código para la visualización del grafo que representa la *LTM*. Este modelo será empleado para generar diversas versiones que serán construidas sobre versiones previas, hasta que el código de visualización cumpla con las necesidades del cliente.

9. **Prueba de integración del subsistema**: en esta etapa se solicitará al cliente que defina un conjunto de datos de prueba, que pueda ser utilizado en el código de visualización y cuyo resultado sea conocido; con la finalidad de garantizar el funcionamiento correcto del código. Se ejecutará principalmente pruebas de unidad.

10. **Prueba de integración del sistema**: en esta etapa se realizará la integración del código de visualización, por medio de *ROS*, con la arquitectura cognitiva *MDB*. Para esta etapa se deberá definir un protocolo de comunicación, que permita obtener información sobre los nodos de conocimiento de la *LTM*, que se han creado tanto en la memoria como en otros subsistemas del *MDB*. Se utilizará el mismo conjunto de datos de la etapa anterior, esta vez siguiendo la estructura de *topics*, *nodos* y *mensajes* de *ROS*. En esta etapa se definirán pruebas de sistema para evaluar la interacción del sistema de visualización con el resto de subsistemas del *MDB*.

11. **Prueba de aceptación**: finalmente se probará el sistema de visualización con el *MDB* en un robot Baxter. Las pruebas para validar esta etapa consisten en pruebas de usuario y pruebas de sistema; para verificar el correcto funcionamiento según las necesidades de los investigadores del GII.

Ahora bien, en la figura 4.1 se detallan las etapas mencionadas siguiendo el modelo V definido en la sección 3.2.1; especificando gráficamente para cada objetivo definido en la sección 2.1.2, las etapas para su cumplimiento.

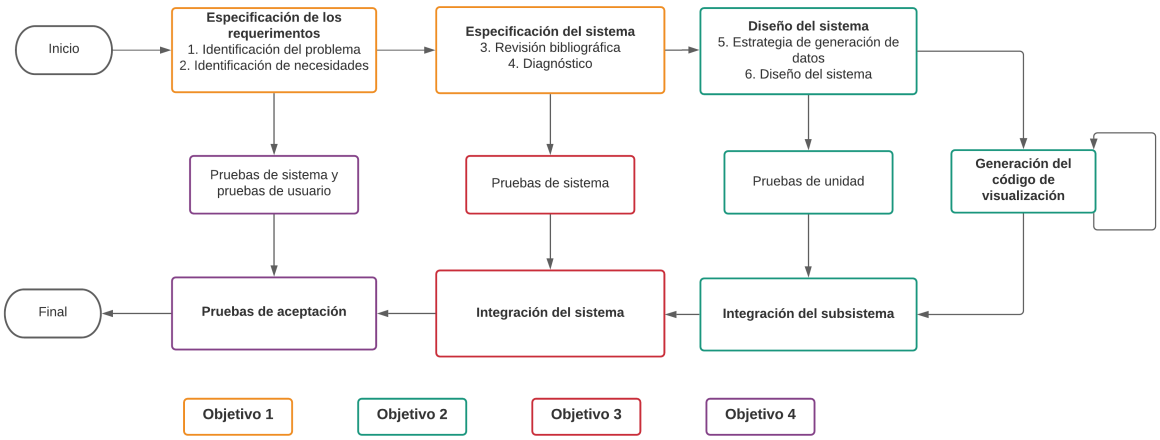

Figura 4.1: Diagrama de la metodología propuesta. Fuente: elaboración propia.

CAPÍTULO 5

PROPUESTA DE DISEÑO

En este capítulo se detalla el proceso de diseño que se siguió en el desarrollo del proyecto. En la sección 5.1 se definen los requerimientos del sistema, mientras que en la sección 5.2 se inicia el diseño general. En la sección 5.3 se inicia el diseño detallado, haciendo una revisión de las herramientas seleccionadas y definiendo parte del código que se implementará en la sección 5.4 donde se utilizan datos generados artificialmente para definir la estructura del código de visualización. En la sección 5.5 se detalla un experimento que permitirá medir la funcionalidad del código de visualización desarrollado. En la sección 5.6 se inicia la implementación del código de visualización en conjunto con la *LTM*, mediante la adaptación del código definido en la etapa 5.4. Finalmente, en la sección 5.7 se detalla la validación del sistema de visualización con un experimento con un robot Baxter.

5.1 Especificación de los requerimientos

La especifiación de los requerimientos se debe basar en el problema que se desea resolver, por lo que se procederá en primer lugar a definirlo. El problema por resolver consiste en la falta de una herramienta de visualización que sirva de interfaz para los investigadores del GII y que les permita extraer información relevante en tiempo aproximadamente real de la LTM.

Una vez definido el problema, se procedió a conversar con el cliente acerca de los requerimientos que debe cumplir la herramienta de software que solucionará el problema. Estos se detallan a continuación.

1. El primer requerimiento es poder observar la estructura de la LTM. Esto significa que se debe poder traducir dicha estructura en una gráfica y desplegarla en algún medio. De esta manera, se tiene que el producto debe ser capaz de seguir los cambios que se dan de la LTM. Dichos cambios pueden ser la aparación de un nuevo nodo, de modo que se debe extraer el tipo al que pertenece, con quién se encuentra conectado, su activación y su nombre; el número de iteraciones de la LTM, la actualización de las activaciones de los nodos, el período actual, el modelo de mundo actual, la recompensa actual y la última *policy* ejecutada.

2. El segundo requerimiento definido es cúales aspectos de la visualización -como la definición de posiciones, colores, tamaños e intensidades- deben ser resueltos en la herramienta de visualización y no en la LTM (como se hacía previo al desarrollo del proyecto).

3. El tercer requerimiento es que la herramienta debe actualizarse periódicamente, de manera que se pueda observar la evolución de la LTM. El cliente ha definido que la actualización de la visualización se debe realizar cada 2 segundos. Sin embargo, la actualización de la estructura de la LTM no tiene porqué realizarse en el mismo período (este tema se tratará en la sección 5.6).

4. El cuarto requerimiento que debe cumplir la herramienta es que sea interactiva. Esto puede incluir, pero no limitarse, a poder hacer *zoom* en la gráfica, mover los nodos, resaltar un nodo, desplegar información del nodo al hacer *click* sobre este o pasar el cursor por encima.

5. El quinto requerimiento que debe cumplir la herramienta es que se pueda pausar y reanudar la ejecución de la LTM. De este modo, la LTM debe ser capaz de recibir comandos de la herramienta de visualización, por lo que se deberá definir un protocolo de comunicación entre la visualización y la LTM.

6. El sexto requerimiento que debe cumplir, es que se pueda guardar el gráfico donde se representa la estructura de la LTM desde la herramienta.

7. El séptimo requerimiento es poder graficar puntos y antipuntos de los *PNodes*, por lo que la herramienta debe ser capaz de seguir la creación de un puntos y antipuntos, además de los nodos a los que pertenecen. Este requerimiento implica que el usuario deber ser capaz de seleccionar el nodo del que se desea obervar la información. Además, al igual que con el gráfico de la red, debe ser capaz de almacenar el gráfico y poder interaccionar con este.

5.2 Diseño del sistema

El diseño del sistema consistirá en definir los elementos básicos de la herramienta de software como lo son el lenguaje de programación, las bibliotecas y el tipo de aplicación que se desarrollará que permitan solucionar el problema.

Para desarrollar esta etapa se realizará en primer lugar un diagnóstico del sistema de visualización previo. A continuación, se procede a definir una estrategia de generación de datos para realizar pruebas de unidad en el código de visualización. Tras esta definición, se definirán las bibliotecas que se utilizarán y en último lugar, se realizará un diagrama

con el flujo del programa, para tener una guía para el trabajo futuro.

Para definir los elementos anteriores se procede a realizar un diagnóstico y evaluación de la LTM con la finalidad de extraer elementos que se pueden mantener, modificar o desechar en el desarrollo de la herramienta de visualización, así como los nuevos elementos que deberán definirse.

5.2.1 Diagnóstico

El diagnóstico se realiza analizando el código de la implementación actual de la LTM. De este análisis se desea extraer: el lenguaje de programación así como las bibliotecas que se usan para graficar y almacenar la red, además de los puntos y antipuntos.

A partir de este análisis, se observó que la LTM se desarrolló en *Python*, que se ejecuta como un nodo de ROS, la biblioteca utilizada para almacenar la red de nodos es *Networkx* y que para graficarla posteriormente se utiliza Matplotlib. Ahora bien, tras conversaciones con el cliente, se observó que la utilización de *Networkx* para almacenar la estructura de la red se puede desechar, pues esta se utiliza únicamente para poder visualizarla posteriormente con *Matplotlib*. De este modo, para el desarrollo del resto del proyecto asumirá que la implementación de *Networkx*en la *LTM* no es útil, y no se contará con ella.

Otra observación importante es que los colores especificados para cada tipo de nodo, así como para sus conexiones y el posicionamiento de los nodos en la gráfica, se pueden mantener. El posicionamiento de los nodos se puede observar en el código del apéndice 8.3 mientras que el esquema de colores para los nodos se puede observar en el cuadro 5.1 y para las conexiones se tiene el esquema de colores del cuadro 5.2.

Cuadro 5.1: Esquema de colores utilizados para dibujar los nodos

Tipo de nodo	**Color**
CNode	Azul
PNode	Morado
Perception	Gris
Policy	Rojo
Forward Model	Naranja
Goal	Verde

Cuadro 5.2: Esquema de colores utilizados para dibujar las aristas

Conexión	**Color**	**Grosor**
CNode - PNode	Morado	1
CNode - Goal	Verde	1
CNode - Policy	Rojo	1
CNode - Forward Model	Naranja	1
Otro	Negro	0.2

Del diagnóstico se observó que la opacidad de cada nodo en la gráfica será definida por su activación, que es un número entre 0 y 1. Además, el grosor de las aristas también depende de los nodos que conecten, de manera que si la arista conecta un *CNode* con algún otro nodo, será más gruesa que si no conectara este tipo de nodos. Un mecanismo similar se puede implemantar en el sistema de visualización por desarrollar.

Sobre el posicionamiento de los nodos es importante notar que su definición no sigue las recomendaciones propuestas en la sección 3.3, sin embargo, es la que el cliente ha definido previamente para efectos de publicaciones pues permite ver claramente la agrupación entre nodos del mismo tipo. Con el sistema de visualización previo, se obtenían gráficas como la que se muestra en la figura 5.1.

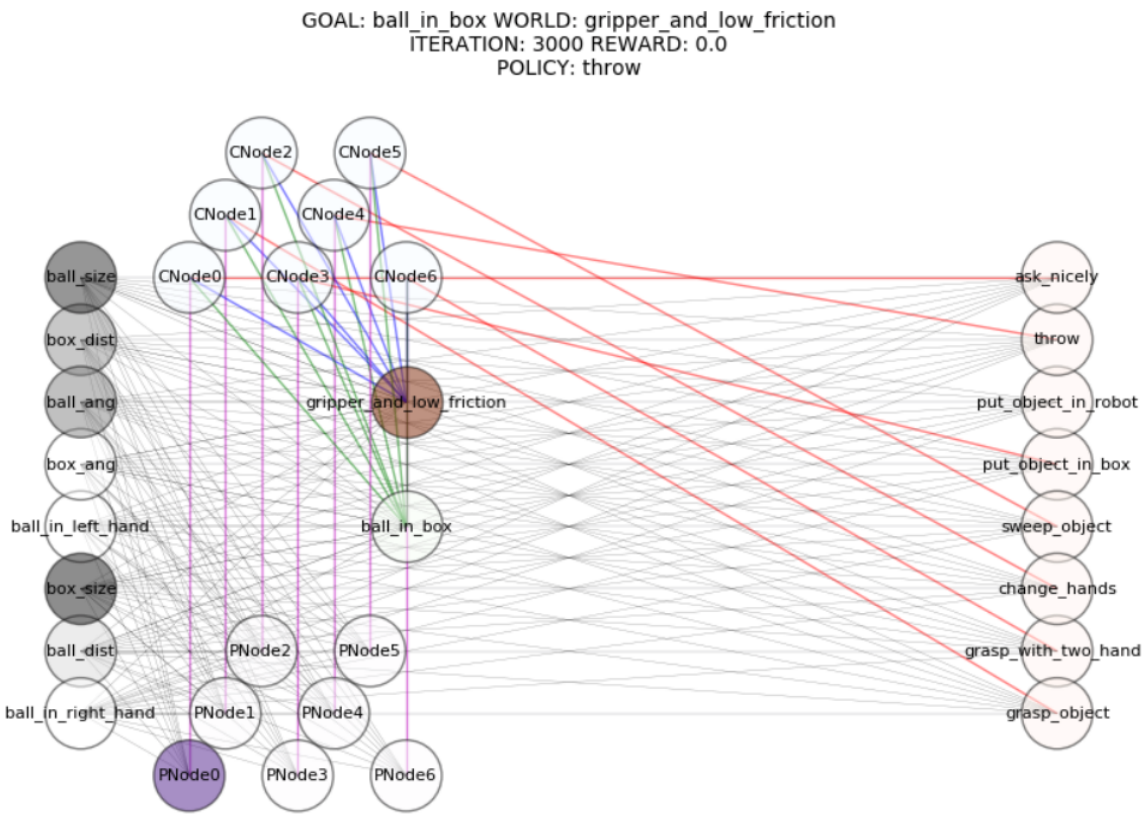

Figura 5.1: Gráfica obtenida con el sistema de visualización previo. Recuperado de [27]

En la figura anterior se logra observar la iteración 3000 de un experimento con la *LTM*. El robot se coloca en una mesa frente a una bola y una caja.

A la izquierda se aprecian las percepciones. De estas, las que terminan en *dist* y *ang*, provienen de una cámara RGB tras realizar una redescripción de la imagen, y representan la distancia y el ángulo en el que se encuentran los objetos. El identificador *size* representa su tamaño y también proviene de la cámara. Las percepciones restantes son discretas y permiten determinar la presencia o ausencia de la bola en un *gripper* del robot. A la derecha se observan 9 *policies*, que describen acciones físicas que el robot puede ejecutar para cumplir con la meta.

Se observa que el modelo de mundo se llama *gripper_and_low_friction* y significa que el robot tiene *grippers* con los que pueden levantar objetos pequeños. Además, la meta

del agente es colocar la bola en una caja.

Finalmente, se observa que en las 3000 iteraciones del experimento se han creado 7 *CNodes* y sus respectivos *PNodes*.

Del título se puede observar que para la última *policy* ejecutada corresponde a *throw*, lo cual significa que el robot lanzó la bola lejos de su área de alcance. Sin embargo, esta acción no logró cumplir con la meta, pues la recompensa obtenida por el robot fue de 0.

5.2.2 Estrategia de generación de datos

Además del diagnóstico del sistema de visualización utilizado actualmente en la LTM se plantea que en el diseño del sistema se deberá definir una estrategia de generación de datos para poder realizar pruebas sobre el código que definirá la herramienta de visualización. Esto se realizará pues, inicialmente, no se integrará el sistema de visualización de forma directa con la LTM.

La estrategia de generación de datos se definió con el cliente de la siguiente manera: para evitar complicaciones en el diseño se simulará la LTM según el pseudocódigo detallado en el cuadro de texto 5.1 con lo que se respetarán las reglas de conexión entre los nodos y la generación de puntos y antipuntos, sin embargo no es necesario replicar los mecanismos con los que se leen las percepciones, se calculan las activaciones de los nodos y se ejecuta una policy. Por esta razón se decidió, en conjunto con el cliente, generar dichos mecanismos de forma aleatoria.

```
Read initial perceptions.
while alive or last iteration not reached
    Update activation of every node
    Select most active policy to be executed
    Execute policy
    Read new perceptions
    Get reward
    if (reward < threshold)
        for activated C-node connected to policy
            for activated P-node connected to C-node
                Add perception to P-node as anti-point
```

```
    else
        if (C-nodes connected to policy)
            for activated C-node connected to policy
                for activated P-node connected to C-node
                    Add perception to P-node as point
        else
            Create new C-node and P-node using perception
        end if
    end if
end while
```

Listing 5.1: Pseudocódigo de la implementación más reciente de la LTM del MDB.

5.2.3 Selección de conceptos

Por la naturaleza de los requerimientos, en específico por la fuerte componente de interactividad que el cliente solicita, se plantea el desarrollo de una aplicación web.

La selección de conceptos se basará en la información recabada del diagnóstico del sistema y de los requerimientos definidos por el cliente. Se debe seleccionar el lenguaje de programación, y en base a esta decisión se escogerá las bibliotecas que cumplan las siguientes tareas: construir la página web tanto en apariencia como en funcionalidad, graficar de forma interactiva la red de la LTM, graficar los puntos y antipuntos de los *PNodes*, almacenar la red de nodos y realizar operaciones sobre ella y almacenar los puntos y antipuntos.

Como se mencionó previamente la *MDB* se desarrolló en *Python* en casi toda su totalidad. Para mantener la modularidad de la arquitectura, se plantea el uso de este mismo lenguaje de programación para el desarrollo del sistema de visualización. Además, presenta la gran ventaja de que existen muchas bibliotecas de *Python* para visualización de datos, como lo son *Matplotlib* [28], *Seaborn* [29] o *Plotly* [30].

La selección de la biblioteca para la construcción de la página web se basará en el tipo de aplicación que se requiere. Por las características de la herramienta, se tiene que la aplicación tiene un alto enfoque a la visualización de datos. De este modo, se plantea el uso de bibliotecas como *Bokeh* o *Dash*, diseñadas para desarrollar aplicaciones de tipo *dashboard*.

Un estudio en detalle sobre *Bokeh* [31] y *Dash* [32], refleja que, si bien *Bokeh* lleva más tiempo en desarrollo, ambas herramientas son respaldadas por un número similar de usuarios en Github. Ambos incorporan elementos con un alto grado de interactividad, pues pueden ser usados tanto como entradas o como salidas a *callbacks* del programa. Además, ambas bibliotecas incorporan herramientas para graficar redes de forma interactiva. Para el caso de *Bokeh*, se tiene *GraphRenderer* y para *Dash* se tiene *Cytoscape*. Ambas herramientas con funcionalidades similares, en las que se puede extraer información o resaltar un nodo o arista al hacer *click* o pasar el cursor por encima y hacer *zoom in* o *zoom out* . Sin embargo, *Cytoscape* es la que más grado de interactividad presenta, pues a diferencia de *GraphRenderer*, permite la selección de un nodo y desplazarlo de su posición actual. Sin embargo, el factor por el que se escogerá *Dash* sobre *Bokeh*, es que el primero tiene soporte para desarrollar aplicaciones multi-página, mientras que el segundo no. La importancia de este hecho, es que dadas las características de la herramienta que se solicita, puede llegar a ser necesario definir más de una página donde se muestren diferentes tipos de información.

Para resumir lo explicado en el párrafo anterior se puede emplear la matriz de selección descrita en el cuadro 5.3.

Cuadro 5.3: Matriz de selección para escoger la biblioteca para construir la aplicación web

Criterio	**Peso**	**Bokeh**	**Dash**
Respaldo de la comunidad	1	1	1
Tiempo de desarrollo	2	2	1
Elementos de interactividad	5	5	5
Paquetes disponibles para graficar redes	5	5	5
Interactividad de paquetes para graficar redes	4	2	4
Soporte para aplicaciones multipágina	5	1	5
Total	22	16	21

Con la matriz de selección anterior, se puede observar que para los criterios definidos, la mejor opción es *Dash*

Derivado de la selección de *Dash* , se escoge *Plotly* para graficar los puntos y an-

tipuntos de los *PNodes*. Además, se escoge *Networkx* [33] como la biblbioteca para almacenar la red de la *LTM*. Si bien es cierto, esta no es la que mejor se desempeña a nivel de cálculo o carga de datos en comparación con otras (*igraph* o *graph-tool*), estos dos factores no son críticos para el desarrollo de la aplicación. La justificación para escoger *Networkx* por encima de otras bibliotecas más eficientes, se basa en cuatro factores.

1. El tamaño de la red de la *LTM* no es relativamente grande (como es el caso del análisis de redes sociales por ejemplo).

2. No se necesita realizar ningún cálculo sobre la red. Por cálculo entiendase, definición de las posiciones (pues como se verá más adelante, estas las define el cliente), cálculo del camino más corto. medidas de centralidad de los nodos, etc.

3. *Dash*, en específico Cytoscape, tiene un soporte amplio para la graficación de redes almacenadas en *Networkx*. Y en cuarto lugar, *Networkx* tiene funciones que permiten la conversión a las estructuras de datos usadas por *Dash* y *Cytoscape*.

Finalmente, para almacenar los puntos y antipuntos de los *PNodes* se utilizará *Pandas* [34], pues provee estructuras de datos como los *DataFrames* que permiten realizar análisis, filtrado y manejo de conjuntos extensos de datos.

5.2.4 Definición del flujo del programa

El sistema de visualización, planteado como una aplicación web, deberá monitorear dos tipos de procesos: interacciones del usuario con la aplicación y la recepción de datos nuevos desde la *LTM*. Ahora bien, estos son los procesos base, de ellos pueden derivar procesos más complejos que se definirán en etapas posteriores del proceso de diseño. El flujo del programa que permitirá solucionar el problema planteado se describe gráficamente en la figura 5.2.

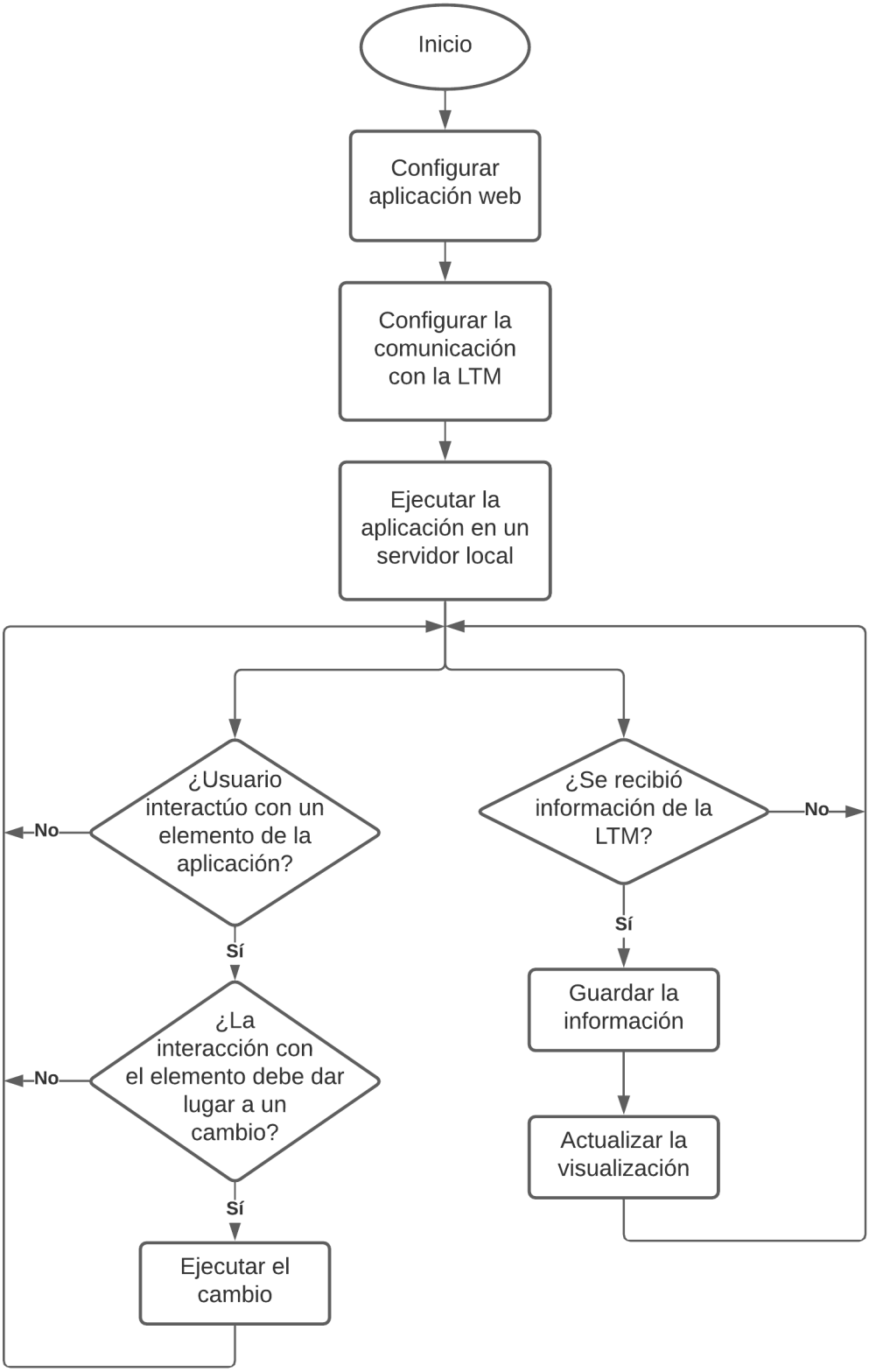

Figura 5.2: Diagrama de flujo del sistema de visualización

Se observa que en el ramal derecho del diagrama presentado, cómo debe responder el sistema ante nuevos datos de la *LTM*. Estos datos pueden ser nuevos nodos, sus activaciones, puntos o antipuntos, información del experimento, entre otros; y deben ser almacenados y procesados para modificar la visualización.

Al finalizar esta etapa, se han definido que el sistema de visualización será una aplicación web, desarrollada con *Python* y *Dash*. La red se almacenará en *Networkx* y para su visualización *Cytoscape*. En cuanto al almacenamiento de puntos y antipuntos, se definió el uso de *Pandas* por su capacidad de almacenamiento y filtrado de los datos.

5.3 Diseño detallado

El diseño detallado consiste en definir de manera más detallada el código que solucionará el problema planteado. Para esta etapa, se procede a realizar un estudio de la documentación de *Dash* y *Networkx*, con el fin de determinar qué elementos de la programación se pueden definir en esta etapa, qué propiedades tienen y cómo se manejarán en la implementación del programa. Además, se abordarán aspectos relevantes a *Cytoscape* y la integración de este paquete con *Networkx*. Finalmente se definirá el método por el cual se agregará un estilo definido a la aplicación.

5.3.1 Aspectos relevantes a Networkx

En esta sección se analizarán las características de la red de la *LTM*, con la finalidad de determinar el objeto del paquete *Networkx* que mejor se adapte a las necesidades del funcionamiento de la aplicación.

De *Networkx* se pueden extraer estructuras que permiten definir redes en las que la dirección de las aristas importa, es decir la arista tiene un origen y un final definido; o redes en las que la dirección de las aristas no importa. Para la visualización de la *LTM* se considera que la red es del segundo tipo; es decir, se clasifica como una red

no dirigida, pues el sentido de las aristas no es importante. Las aristas en este caso se emplean únicamente para visualizar las conexiones entre un par de nodos.

Otro aspecto importante, es que *Networkx* también soporta redes donde la conexión entre un par de nodos se puede dar por múltiples aristas. Sin embargo, para el caso de la *LTM*, estas no serán necesarias, pues la red no presenta esta característica.

De este modo se tiene que el objeto de *Networkx* que mejor se ajusta a las necesidades de la herramienta es el objeto *Graph*, pues este permite almacenar nodos y aristas en una red no dirigida, y donde los nodos se conectan únicamente por una arista.

El siguiente aspecto a considerar, es que el objeto *Graph* permite añadir nodos y además una serie de atributos asociados con la función *add_nodes()*. El nodo se podrá identificar como un número, *string* o cualquier objeto que sea *hashable* de *Python*, y dicho identificador será el primer argumento de la función previamente mencionada. Además del identificador, se pueden definir una serie de atributos que representan el nodo. De este modo, para la aplicación que se desarrollará, es necesario definir tanto el tipo de identificador como los atributos que se almacenarán en el nodo y para las aristas, se deben definir los atributos que se almacenarán en él.

Las siguientes propiedades del nodo son candidatos a atributos que pueden almacenarse en la red: el nombre, el tipo de nodo, la activación, la posición definida y el color. Por otro lado, las aristas pueden almacenar como atributos, el color de la línea y su grosor.

5.3.2 Aspectos releventes a Dash

Dash, en específico los *frameworks* de desarrollo web sobre los que se desarrolló como lo son *Flask*, *Plotly.js* y *React.js*; se encarga de abstraer los protocolos y tecnologías necesarias para construir aplicaciones web, en estructuras de *Python* mucho más fáciles de implementar.

Las aplicaciones web de *Dash* se componen de dos partes principales: del *layout* y de

los *callbacks*. La primera define la estructura de la aplicación web, es decir, los elementos que observa el usuario al utilizarlo como lo pueden ser títulos, textos, imágenes, gráficas, botones, links, entre otros. El *layout* se puede definir como un árbol jerárquico de componentes tal y como se observa en el siguiente código, donde se realiza la definición del *layout* de una aplicación con tres componentes. En la figura 5.14 se observa la jerarquización de estos elementos.

```
app.layout = html.Div([
                        dcc.Location(id='url', refresh=False),
                        html.Div(id='page-content')
                    ])
```

Listing 5.2: Ejemplo de la definición del layout de una aplicación

Del código mostrado en el cuadro de texto 5.2 se puede observar que un elemento, por ejemplo *dcc.Location*, se compone de un identificador con el nombre de *id*, y de una propiedad que para este elemento corresponde a *refresh*. Estos conceptos serán necesarios para definir la estructura y definición de los *callbacks*.

Los *callbacks* son llamados a funciones que se realizan cuando la propiedad de un elemento definido cambia de valor. De la investigación preliminar sobre *Dash*, se observó que algunas de estas propiedades disponibles para usarse en la aplicación pueden ser, por ejemplo, un contador que maneja el número de veces que el usuario ha hecho *click* en un botón, un contador que lleva el número de veces que ha pasado un intervalo de tiempo o el valor escogido en un selector de opciones. Ahora bien, estas funciones se activan con los cambios a las entradas definidas, y deben producir una salida que cambiará algún elemento de la aplicación. Ejemplo de estas salidas pueden ser una gráfica o un cuadro de texto.

Para definir la primera parte de la aplicación, es decir el *layout*, *Dash* provee dos componentes principales con elementos que pueden formar la estructura de la aplicación: *Dash Core Components* y *Dash HTML Components*. A continuación se analiza cada uno de estos componentes y algunos elementos básicos que contiene y que podrían usarse en la definición de la aplicación.

5.3.2.1 Dash HTML Components

Dash HTML Components (o *html*) permite al diseñador definir estructuras de *HTML* mediante estructuras de *Python*. Algunos elementos de este paquete se analizarán a continuación, pues pueden utilizarse en el diseño de la aplicación web. Dichos elementos son botones, cuadros de texto, encabezados y elementos de división.

A continuación se analizan las posibles aplicaciones de dichos elementos dentro de la aplicación web y algunas propiedades necesarias para la definición de *callbacks*.

html.Button

Este elemento define un botón en la aplicación web. Se plantea usar este elemento para pausar o reanudar el experimento, así como guardar la gráfica de la red. Este elemento puede usarse para llamar un *callback*, mediante una de sus propiedades llamada **n_clicks**, que es un número entero con el que se cuenta el número de veces que se ha pulsado el botón.

html.Pre

Este elemento permite agregar cuadros de texto, con la peculiaridad de que el texto se presenta en la aplicación web en el mismo formato con el que se ha definido en el código. Este elemento se puede usar para desplegar información relevante de un nodo, como lo puede ser su nombre, activación, tipo y los nodos con los que se conecta. El texto que despliega este elemento en la aplicación web se puede modificar al acceder a la propiedad llamada *children*, por lo esta propiedad debería ser la salida de un **callback** que se encargue de modificar el texto del elemento *html.Pre*.

html.Div

Este elemento permite agregar divisiones entre estructuras de la aplicación web. Se utiliza para que todos los elementos dentro de esta división tengan el mismo estilo. Se plantea usarlo para agrupar secciones de la aplicación web que se relacionen entre sí, o que cumplan un objetivo en común. Por ejemplo, se podría usar para agrupar los dos botones definidos tanto para el pausado como para el guardado.

html.H1, html.H2, html.H3

Estos elementos se encargan de definir encabezados en la aplicación web, por lo que podrían utilizarse para definir títulos o instrucciones. No se platea el uso de este elemento para definir los *callbacks*. Para modificar secciones de texto, se prefiere el uso de los elementos de tipo *Pre*

5.3.2.2 Dash Core Components

Por otro lado, *Dash Core Components* (o *dcc*) contiene estructuras que permiten generar aplicaciones web interactivas. Por ejemplo, se tienen *check lists*, deslizadores, entradas de texto o pestañas para la selección de datos, o contenedores gráficos para renderizar figuras de *Plotly*. Se plantea el uso de este paquete para aumentar el grado de interactividad de la aplicación, mediante la implementación de selectores de datos y de contenedores gráficos.

dcc.Markdown

Este tipo de elemento permite tener una lista desplegable de opciones, de donde el usuario puede escoger una opción que se almacenará en la propiedad **value**. Dichas opciones se almacenan en una lista de diccionarios, en donde se almacena en cada diccionario un **label**, que almacena el nombre que se observa en la lista, y un **value** que será el que identificará la opción seleccionada. De este modo, se puede implementar su propiedad **value** para definir un llamado que se ejecute cuando esta cambie. Una implementación en código del elemento *Dropdown* se detalla en el cuadro de texto 5.3 y su respectivo resultado se presentan en la figura 5.3.

```
dcc.Dropdown(
        className="eight columns",
        id='option-dropdown',
        options=[
            {'label': 'LTM', 'value': 'ltm'},
            {'label': 'PNode 1', 'value': 'pn1'},
            {'label': 'PNode 2', 'value': 'pn2'}
        ],
        style = {'width': '100%', 'color': 'black'},
        value='LTM'
```

```
11      )
```

Listing 5.3: Implementación de un elemento de tipo Dropdown

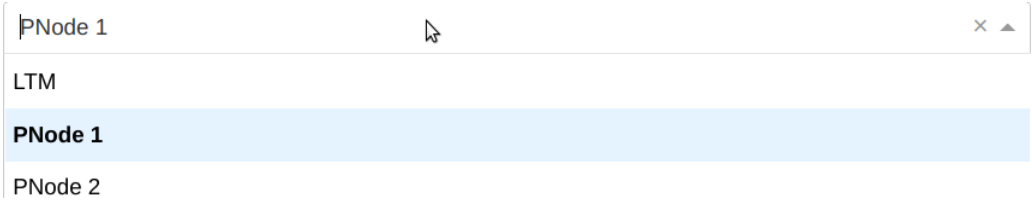

Figura 5.3: Elemento de tipo dropdown. Elaboración propia

dcc.Graph

Este elemento es un contenedor gráfico que permite renderizar figuras de *Plotly*. Como propiedad principal se tiene **fig**, donde se almacena la figura que se desea renderizar. Además, posee dos propiedades importantes como lo son **clickData** y **hoverData**, que contienen la información relevante al último evento de *click* y de pasar el cursor por encima, respectivamente. De este modo, se podría utilizar estas últimas propiedades para hacer llamados cada vez que se pase el cursor por encima de un dato graficado.

Los elementos anteriores, representan la base sobre la que se construirá la aplicación. Es importante notar, que según el desarrollo del proyecto, las necesidades que la herramienta debe cumplir pueden cambiar, con lo que se tiene que pueden ser necesarios nuevos elementos no mencionados en esta sección. En dicho caso, se explicará el funcionemiento principal del elemento y la funcionalidad que agrega a la aplicación.

Ahora bien, dentro de los elementos primordiales para el desarrollo de la aplicación web es que debe actualizarse periódicamente. Para lograrlo es necesario definir un elemento de tipo **dcc.Interval**, que hará un llamado a un *callback* de manera periódica. Para esto es necesario definir la propiedad que se llama **interval** que actualizará un contador llamado **n_intervals** cada vez que hayan pasado **interval** milisegundos. De este modo, se plantea utilizar este elemento para actualizar las gráficas, tanto de la *LTM* como de los *PNodes*.

5.3.3 Aspectos relevantes a Dash Cytoscape

Otro elemento primordial, pues así se definió en la sección anterior, es *Dash Cytoscape*. Como se mencionó previamente, esta herramienta está diseñada para la visualización e interacción de redes en *Dash* y se basa en *Cytoscape.js*.

Para utilizar esta herramienta en la aplicación web se debe importar el paquete *dash_cytoscape*. Dentro de este módulo se encuentra el objeto *Cytoscape*. Un ejemplo de la declaración de este tipo de objeto se muestra en el cuadro de texto 5.4.

```
cyto.Cytoscape(
        id='cytoscape-two-nodes',
        layout={'name': 'preset'},
        style={'width': '100%', 'height': '400px'},
        elements=[
            {'data': {'id': 'PNode 1', 'label': 'PNode 1'}, 'position'
    : {'x': 75, 'y': 75}},
            {'data': {'id': 'CNode 1', 'label': 'CNode 1'}, 'position'
    : {'x': 200, 'y': 200}},
            {'data': {'source': 'PNode 1', 'target': 'CNode 1'}}
        ]
    )
```

Listing 5.4: Declaración de un elemento de Cytoscape

Del código anterior se observan 4 propiedades principales. La primera es el *id* del elemento, que lo identifica dentro del *layout* de la aplicación web. El segundo es el *layout* que definirá las posiciones de los nodos en la gráfica. En este caso, dado que el valor asociado a la llave *name* es *preset*, la posición de los nodos vendrá dada por la posición especificada en la tecer propiedad, *elements*. Esta última contiene la información de los nodos y aristas de la red. Por último, se tiene la propiedad *style*, que definirá el estilo del componente dentro de la aplicación.

Dicho código se traduce a una gráfica como la que se muestra en la figura 5.4.

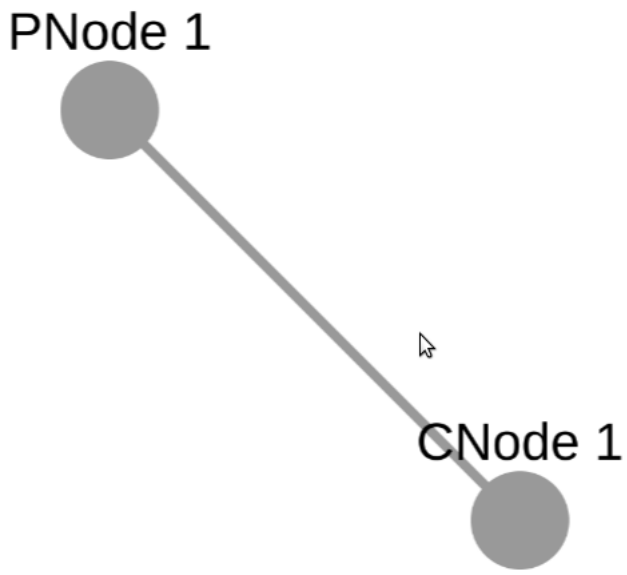

Figura 5.4: Ejemplo de red con *Cytoscape*. Elaboración propia

Ahora bien, el propósito de esta introducción a *Cytoscape* es comprender cuatro aspectos principales de cara a la definición del código de visualización. El primero de ellos es definir el formato en el que se almacenan los nodos y aristas de la red, para así definir las operaciones que se deben efectuar sobre la red almacenada en un objeto de *Networkx*. El segundo, consiste en comprender de qué forma se definen aspectos como tamaño, forma y color de los nodos, aristas y texto que aparezcan en la gráfica. El tercero se basa en definir las propiedades del objeto *Cytoscape* que van a dar lugar a *callbacks*, así como el formato en el que están definidos. Y el cuarto, se fundamenta en el guardado de la gráfica como una imagen.

Almacenamiento de la red

En *Cytoscape* los elementos que componen la red (nodos y aristas) se definen como una lista de diccionarios, tal y como se observa en el código del cuadro de texto 5.4, en la propiedad *elements*.

Cada diccionario presente en la lista representa un nodo o una arista de la red y se compone de las siguientes llaves:

- **data**: un diccionario con la información del elemento. A su vez este diccionario puede contener las siguientes llaves:
  - **id**: un *string* con el identificador del nodo. Servirá para referenciar el elemento.
  - **label**: un *string* con el nombre del nodo. Este se emplea por convención, para definir el texto que aparecerá asociado al nodo en el gráfico.
  - **source**: especico para las aristas, es un *string* que define su origen.
  - **target**: especico para las aristas, es un *string* que define su destino.
  - **Otra**: se pueden definir llaves adicionales, como lo puede ser color u opacidad, que serán utilizadas como selectores para definir el estilo de la red.
- **position**: específica para los nodos, es a su vez un diccionario con la coordenada x y y del nodo.

Estilo de la red

Para definir el estilo de una red con *Cytoscape* se debe definir una variable llamada *stylesheet*. Al igual que *elements*, esta variable es una lista de diccionarios en donde cada diccionario está definido por dos variables:

- **selector**: un *string* que define el elemento por estilizar. Este selector puede identificar un grupo (de nodos o aristas) o un solo elemento. Además se pueden definir atributos de los elementos para realizar un filtrado. Por ejemplo, para el código del cuadro de texto 5.5. Se podría definir un selector que tome solo los elementos cuyo tipo de nodo es *PNode*.

```
elements=[
    {'data': {'id': 'PNode 1', 'label': 'PNode 1', 'node_type' : '
   PNode', 'color': 'purple', 'activation': '0.5', 'name': 'PNode 1'},
    'position': {'x': 75, 'y': 75}},
```

```
    {'data': {'id': 'CNode 1', 'label': 'CNode 1', 'node_type' : '
    CNode','color': 'blue', 'activation': '0.5', 'name': 'CNode 1'}, '
    position': {'x': 200, 'y': 200}},
    {'data': {'id': 'title', 'label': 'title', 'node_type' : 'title','
    color': '', 'activation': '0', 'name': 'GOAL: WORLD: \n POLICY: '},
     'position': {'x': 137.5, 'y': 60}},
    {'data': {'source': 'PNode 1', 'target': 'CNode 1', 'color' : '
    purple', 'width' : '8'}}
]
```

Listing 5.5: Ejemplo de elementos de *Cytoscape* con atributos

- **style**: esta variable corresponde a un diccionario donde se especifica la propiedad que se desea modificar y su valor. Por ejemplo algunas propiedades son: la forma, tamaño, color y opacidad de un nodo, que son características importantes de modificar para efectos del sistema de visualización.

Para el caso de los elementos del cuadro de texto 5.5, se puede definir un *stylesheet* que seleccione los nodos y aristas y defina propiedades como el color, opacidad o grosor, según los propios valores almacenados en la red. Dicho *stylesheet* se muestra en el cuadro de texto 5.6. Otros valores pueden tomar valores constantes, implicando que para todos los elementos del grupo seleccionado, esa propiedad tendrá el mismo valor.

```
stylesheet = [
            {
                'selector': 'node',
                'style': {
                    'label': 'data(name)',
                    'size' : '20%',
                    'font-size' : '18',
                    'background-opacity' : 'data(activation)',
                    'background-color': 'data(color)'
                }
            },
            {
                'selector': 'edge',
                'style': {
                    'width' : 'data(width)',
                    'curve-style' : 'bezier',
                    'line-color' : 'data(color)'
                }
            },
            {
                'selector': '[node_type *= "title"]',
                'style': {
```

```
                    'size' : '1%',
                    'text-wrap' : 'wrap',
                    'font-weight': 'bold'
                }
            },
        ]
```

Listing 5.6: *Stylesheet* para definir el estilo de la red

También es importante notar, que el último selector hace un filtrado de los elementos. Sólo aquellos cuyo tipo de nodo empiece con la palabra *title*, serán seleccionados para aplicarles el estilo definido.

Una comparación entre elementos sin estilo y con estilo definido para los elementos del cuadro de texto 5.5 se detalla a continuación. En la figura 5.5 se muestra una red sin un *stylesheet* definido y en la figura 5.6 con uno que permite definir el color de los *PNodes* y *CNodes*, así como el de la arista que los conecta.

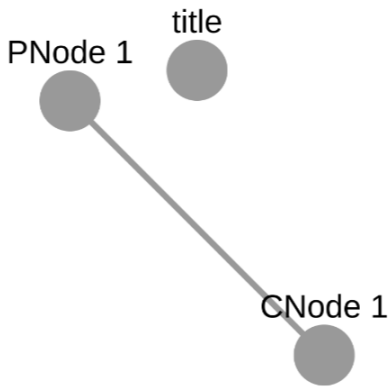

Figura 5.5: Red sin estilo definido.

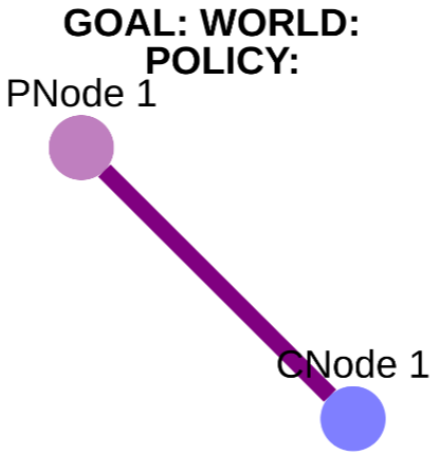

Figura 5.6: Red con estilo definido.

Guardado de la red

Para almacenar la red en una imagen se debe modificar el valor de la propiedad *generateImage*. Esta corresponde a un diccionario con las siguientes llaves:

- **type**: define en un *string* el formato de la imagen. En el caso del sistema de visualización se plantea guardar la imagen en formato *png*.

- **action**: define la acción para guardar la imagen. Este puede tomar el valor *store* con lo que almacena la información de la imagen en otra propiedad llamada *imageData*; o *download*, con la que se abre una ventana del navegador para definir la dirección donde se almacenará la imagen y después descargarla.

- **filename**: variable que almacenará el nombre de la imagen.

Una vez dicho esto, se determina que sea cual sea el método definido en la aplicación web, para guardar la imagen se necesita un callback que modifique el diccionario almacenado en *generateImage*. Este diccionario se puede definir en esta etapa, tal y como se muestra en el cuadro de texto 5.7.

```
generateImage = {
            'type': 'png',
            'action' : 'download',
            'filename': 'ltm_1'
        }
```

Listing 5.7: Diccionario para generar imagen de la red.

Con este diccionario, la imagen se descargará en formato *png* con el nombre *ltm_1*.

Interacción con la red

Como se mencionó previamente, una ventaja de esta herramienta es que permite muchas formas de interaccionar con la red, requisito esencial para cumplir con el cuarto requerimiento. En dicho enunciado se planteaba desplegar información cuando se pasa el cursor por encima o cuando se presiona un nodo. *Cytoscape* tiene la ventaja de que cuando el primer evento ocurre en la aplicación, el diccionario *tapNodeData* almacena el diccionario correspondiente a dicho nodo en la lista *elements*; y cuando ocurre el segundo evento, se almacena en el diccionario *mouseoverNodeData*.

5.3.4 Integración entre *Networkx* y *Cytoscape*

Una vez abordados los principales aspectos de *Networkx* y *Cytoscape*, se procederá a plantear el código que permita traducir la red almacenada en una estructura de tipo

Graph, a un formato procesable en *Cytoscape*. Además, se definirán los atributos que finalmente serán almacenados en cada nodo y arista, para así definir el *stylesheet* que dará el estilo a la red.

Una de las limitaciones de *Cytoscape* es que no es posible realizar anotaciones en la red. Por anotaciones se entiende texto que aparece en la gráfica de la red y que no corresponda a una etiqueta asociada a un nodo o arista. Por esta razón se plantea utilizar un nodo 'fantasma', es decir con él se mostrará el valor del título pero no el nodo en sí. Para ello se debe definir su opacidad inicial como 0 y nunca cambiar este valor, tal y como se mostró en el cuadro de texto 5.5.

Para integrar *Networkx* y *Cytoscape*, se deberá almacenar para cada nodo los siguientes atributos: el nombre, el tipo de nodo, la activación, la posición y su color; mientras que para las aristas se guardará el color y el grosor. Se debe recordar que el color, tanto de los nodos como de las aristas, está definido por el tipo al que pertenece y por los nodos que conectan, respectivamente. De este modo, se plantea el uso de la plantilla de estilos definida en el cuadro de texto 5.6

Los elementos de la red se traducirán al formato de *Cytoscape* usando la función *json_graph.cytoscape.cytoscape_data()* del paquete *Networkx.readwrite*.

5.3.5 Definición de estilos dentro de la aplicación

La definición de los estilos consiste en especificar la apariencia que cada componente tendrá dentro de la aplicación. Dicha apariencia indicará algunos aspectos visuales como lo son el color del fondo, la fuente, el tamaño y el color de las letras o el ancho y largo del componente. Para cada componente se puede definir un estilo en forma de diccionario en la propiedad *style* o se puede asignar una clase de una plantilla de *CSS* a la propiedad *className*, y *Dash* asignará el estilo de la plantilla a dicho componente.

Ahora bien, en el desarrollo del *layout* se utilizará tanto una plantilla como la definición del estilo por medio de diccionarios, según el caso. En el código que se colocará en

la sección de apéndices, se podrán observar ambos casos.

5.4 Módulo y código de unidad y prueba

En esta etapa se detalla la implementación de la estrategia de generación de datos definida en la subsección 5.2.2 y la implementación de los *callbacks* y el *layout* que deberá tener la aplicación web que se desarrollará con *Dash* y *Python*. Además, se abordará el diseño evolutivo del *layout* de la aplicación y la definición general de los *callbacks* necesarios para añadir la interactividad planteada. Finalmente, se detallan los *callbacks* específicos a cada *layout* definido, especificando sus entradas y salidas.

5.4.1 Simulación de la *LTM*

La simulación de la *LTM* corresponde a la implementación en código de la estrategia de generación de datos definida en la subsección 5.2.2. Este código depende de la biblioteca *Networkx* para la construcción y almacenamiento de la red, de *Pandas* para el almacenamiento y clasificación de los puntos y antipuntos, de *random* para generar número aleatorios y *yaml* y *yamlloader*loader para cargar archivos de configuración iniciales.

Para desarrollar el código con la simulación de la *LTM* se creó una clase llamada *LTM* que contiene los atributos y los métodos necesarios para poder emular la implemantación de la *LTM*.

El código con la implementación del simulador de la *LTM* se puede observar en el código del apéndice 8.5.

5.4.2 Desarrollo del *layout* de la aplicación web

Como se mencionó previamente, las aplicaciones de *Dash* consisten en un *layout* que define la apariencia y estructura de la aplicación y de *callbacks*, que son funciones que se

ejecutan cuando el usuario interacciona con algún elemento del *layout* de la aplicación. En esta sección se detalla el proceso para definir el leyout final de la aplicación web que solucionará el problema planteado.

El desarrollo del *layout* siguió un proceso de desarrollo incremental. Es decir, se sigue un diseño iterativo, donde la colaboración con el cliente es crucial para definir la aplicación que se ajuste mejor a sus necesidades. La información que se comparte con el cliente será una imagen del *layout* y una explicación del posible funcionamiento del *layout*.

Los *layouts* que se presentan a continuación se desarrollaron utilizando las bibliotecas de *Dash dash-core-components* (o dcc) y *dash-html-components*. Para definir el estilo de los elementos que componen el *layout* de la aplicación se utilizó una plantilla de CSS.

5.4.2.1 Primera versión

En primer lugar se programó un bosquejo que consistía en los posibles elementos que podría tener la aplicación. Dicho bosquejo se puede apreciar en la figura 5.7.

Figura 5.7: Primera versión del *layout* de la aplicación. Elaboración propia

La figura anterior presenta la interfaz que un usuario de la herramienta se podría encontrar. Consiste de 2 botones: uno para el guardado y otro para el pausado; cuadros de texto donde se pueda observar información relevante al nodo, como lo puede es su activación, el tipo de nodo y los nodos con los que se conecta y una ventana donde se puede graficar tanto la red como los puntos y antipuntos de un nodo en específico.

El funcionamiento que se propone para esta primera versión del *layout* se enlista a continuación:

- Cuando el usuario presiona el botón de Pausa, se pausa o se reanuda la ejecución de la LTM.
- Cuando el usuario presiona el botón de Guardar, se almacena la red en formato de texto y una gráfica de la red.
- Al hacer *click* sobre un nodo o pasar el cursor por encima de este, se despliega información relevante en los cuadros de texto presentados a la derecha.
- La ventana con el gráfico se actualiza cada 2 segundos.

Al exponer esta primera versión al cliente, este dio su visto bueno para usarla como base, pues el funcionamiento propuesto cumple con los requerimientos 1,3,4,5 y 6. Sin embargo planteó la observación de que no se propone el mecanismo por el cual se pueda observar los puntos y antipuntos de un nodo en específico.

El código que define la primer versión del *layout* se puede observar en la sección 8.6

5.4.2.2 Segunda versión

La segunda versión se basa en la estructura de la primera y se propone el mismo funcionamiento. A la estructura inicial se agrega un elemento de tipo *Dropdown* encima de la ventana donde se grafica la red. Este elemento permite desplegar una lista desplegable de opciones, de donde el usuario puede seleccionar solamente una. Dicho

elemento desplegará una lista con los *PNodes* de la red y la *LTM*. La figura 5.8 contiene la segunda versión del layout expuesta al cliente.

Figura 5.8: Segunda versión del *layout* de la aplicación. Elaboración propia

De este modo, se propone añadir el siguiente funcionamiento al de la versión base:

- Cuando el usuario escoge una opción del elemento *Dropdown*, la ventana con el gráfico se actualiza.
- Las opciones del *Dropdown* se actualizan cada 2 segundos. Esto se debe realizar pues los *PNodes* se van añadiendo a la red. No tiene sentido desplegar un *PNode* que aún no ha sido creado.

El código que define la segunda versión del *layout* se puede observar en la sección 8.7

5.4.2.3 Tercera versión

Similar a la segunda versión, esta se basa en la estructura y funcionamiento de la primera. Sin embargo, añade la posibilidad de seleccionar la información que se desea

graficar. En esta versión se emplea un elemento de tipo Tabs, que consiste en pestañas que muestran los *PNodes* que han sido creados y la *LTM*. De este modo, el usuario escoge la información por graficar, al hacer *click* sobre alguna pestaña. El código que define la Tercera versión del *layout* se puede observar en la sección 8.8 y en la figura 5.9 se observa la imagen expuesta al cliente.

Figura 5.9: Tercera versión del *layout* de la aplicación. Elaboración propia

Ahora bien, la segunda y tercera versión del *layout* se desarrollaron en paralelo, ambas fueron conceptos que se presentarían al cliente de manera simultánea para que fuera este el que escogiera la mejor opción o desechara ambas.

El cliente prefirió la segunda versión, pues luego de revisar el código determinó que era más fácil de actualizar los valores mostrados en el *Dropdown* que en el Tabs. Además, planteó la idea de desplegar la aplicación en más de una pantalla, de manera que en una se pueda observar el progreso de la *LTM* y en las demás, algunos de los *PNodes*. Esta observación, sumada al soporte de *Dash* para desarrollar aplicaciones multi página, dio lugar a la siguiente versión del *layout*.

5.4.2.4 Cuarta versión

La cuarta versión del *layout* de la aplicación web se basa en el concepto de aplicación multi-página. Este concepto se refiere a la capacidad de *Dash* de desplegar diferentes *layouts* en función de la dirección URL que se escriba en el navegador. De este modo, se puede desplegar una página con un *layout* específico en una pestaña del navegador, y otra página con un *layout* completamente distinto en otra pestaña.

Al utilizar el concepto de aplicación multi-página, la cuarta versión del *layout* será definido como un conjunto de *layouts* en lugar de uno solo. El primero de ellos será un *layout* inicial en blanco, que el usuario nunca observará, el segundo será el *layout* de la página de inicio que se cargará por defecto al iniciar la aplicación, el tercero será el *layout* de la *LTM* y el cuarto será el *layout* para graficar los puntos y antipuntos de los *PNodes*.

El *layout* de la página de inicio se accede al ingresar en la barra de búsqueda del navegador la siguiente dirección: *http://127.0.0.1:8050/*. Ahora bien, la apariencia del *layout* de inicio se muestra en la figura 5.10.

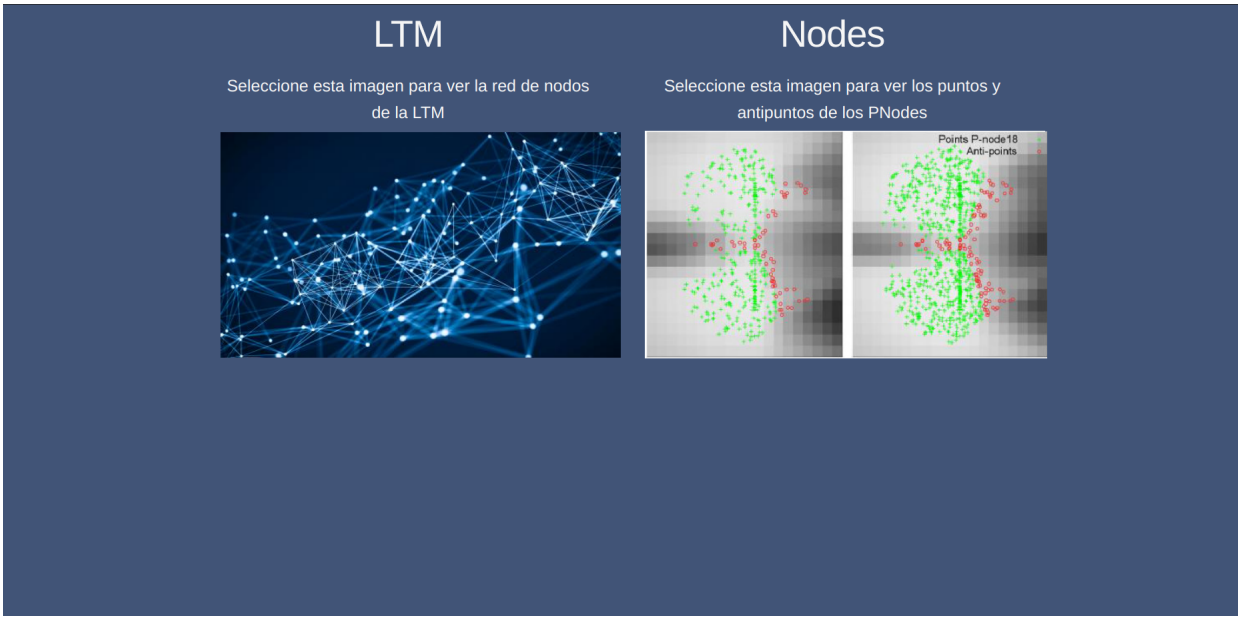

Figura 5.10: Página de inicio de la cuarta versión del *layout*. Elaboración propia

El funcionamiento propuesto de la página de inicio consiste en dirigir al usuario a la página de la *LTM* o de los *PNodes* el hacer *click* sobre una imagen. Esta funcionalidad se aprovecha de tres elementos de *Dash*: *dcc.Location*, *html.A* y *dcc.Link*. El primero almacena la dirección de la barra de búsqueda del navegador en una propiedad llamada *pathname*, el segundo permite modificar la dirección del navegador y actualiza la página, mientras que el tercero modifica la dirección pero no la actualiza. De este modo, cuando se selecciona un *html.A* o un *dcc.Link*, el *pathname* de un *dcc.Location* definido cambia de valor, con lo que podría activar un *callback* que modifica el *layout* que el usuario observa.

En el caso de la página de inicio de la figura 5.10, se tienen dos elementos de tipo *html.A* que contienen una imagen que al ser presionada por el usuario, redirige a la siguiente dirección: *http://127.0.0.1:8050/ltm* en caso de presionar la que se encuentra bajo el título *LTM* o, en caso de presionar la imagen bajo0*Nodes*, a la dirección *http://127.0.0.1:8050/nodes*.

Si la dirección del navegador es *http://127.0.0.1:8050/ltm* el layout al que se dirige al usuario se presenta en la figura 5.11.

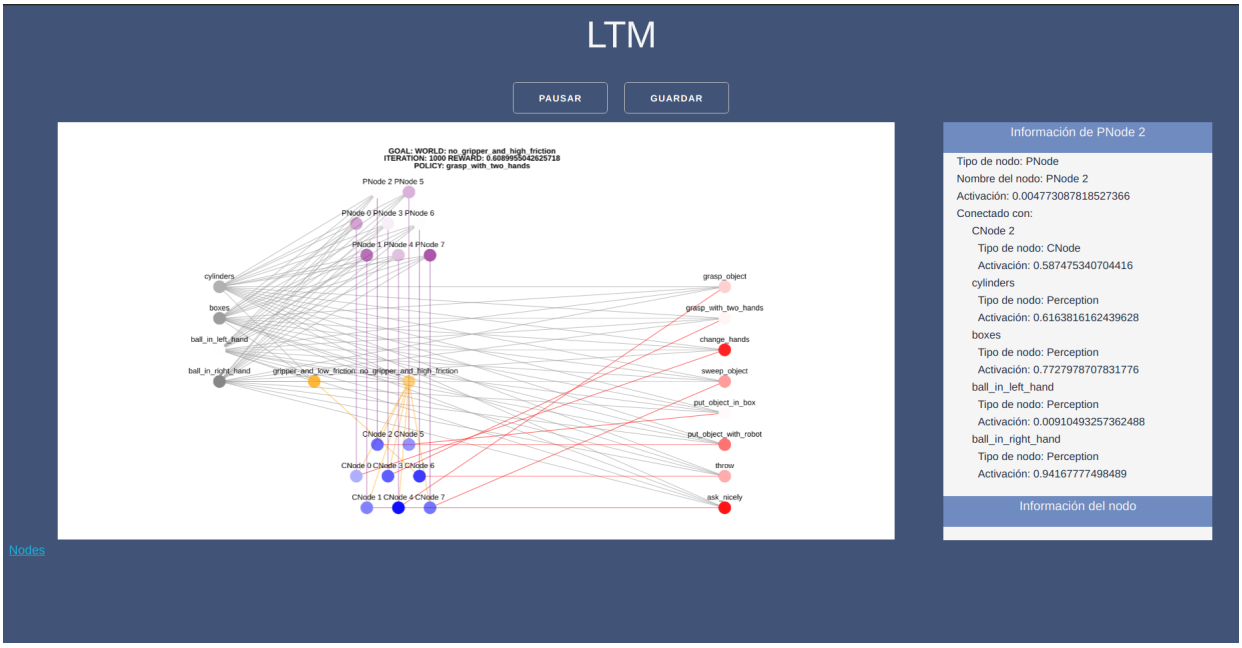

Figura 5.11: Página de la *LTM* de la cuarta versión del *layout*. Elaboración propia

El funcionamiento de esta página es el mismo que se propuso para la versión inicial del *layout*. Además, no debe añadirsele la funcionalidad de seleccionar el PNode al que se le desea observar los puntos y antipuntos, pues esta funcionalidad se abarca en otra página. Lo único que se adicionó con respecto a la versión inicial, es un elemento de tipo dcc.Link ubicado bajo la gráfica, con el nombre *Nodes*. Se plantea que al ser presionado, el usuario sea dirigido a la página donde se despliegan los *PNodes*.

El *layout* de la página de los *PNodes* consiste en dos elementos de tipo dcc.Graph, donde se muestra una figura de *Plotly* que puede ser de tipo polar o rectangular. Encima de cada uno de estos elementos se tiene un *Dropdown* con los *PNodes* creados hasta el momento. A la derecha de la página se tienen diversos elementos para configurar las gráficas mostradas. En primer lugar se tiene un elemento de tipo dcc.RadioItems con el que se selecciona el tipo de gráfica (polar o rectangular), en segundo lugar se tienen dos *Dropdown*, donde se escogen el par de percepciones que se desean graficar. Además, debajo de las ventanas para graficar, se tiene un elemento dcc.Link que al ser pulsado, dirige al usuario a la página de la *LTM*. La selección del tipo de gráfica (polar o rectangular) es una funcionalidad que le permite al usuario definir la forma en la que se observarán las relaciones entre dos sensorizaciones definidas en el *PNode*

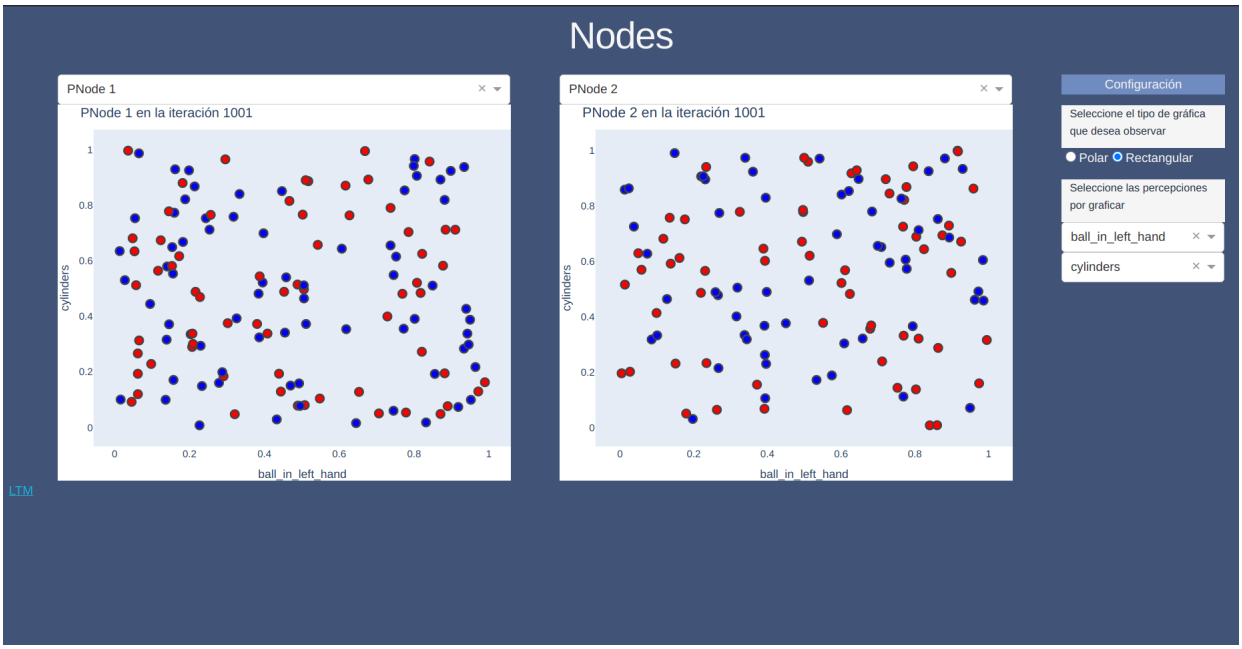

Figura 5.12: Página de los *PNodes* de la cuarta versión del *layout*. Elaboración propia

En la figura 5.12 se logra apreciar dos gráficas de puntos y antipuntos (generados aleatoriamente). A la izquierda se aprecia el conjunto de datos para el *PNode 1* y a la derecha el *PNode 2*. Se observa además que el experimento está corriendo la iteración 1001, que el tipo de gráfica es rectangular y que las percepciones graficadas son si la bola se encuentra en la mano izquierda y la distancia de los cilindros. Si bien es cierto, la primera percepción es de tipo discreta, el objetivo de esta imagen es servir de ejemplo para el cliente de lo que podría observar en este *layout* con el sistema desarrollado.

Es importante describir la funcionalidad de este nuevo *layout*. A este se accede si la barra del buscador contiene la dirección *http://127.0.0.1:8050/nodes*. Además, se propone que las dos gráficas se actualicen cuando se de alguno de los siguientes casos: han pasado 2 segundos desde la última actualización, el valor seleccionado en el elemento dcc.RadioItemas ha cambiado, el PNode seleccionado para graficar ha cambiado o alguna de las dos percepciones de los elementos *dcc.Dropdown* se ha modificado. Ahora bien, el cambio en alguno de esos valores por si solo no garantiza que la gráfica se pueda observar, todos los valores de los elementos anteriores deben estar definidos.

Tras mostrar esta versión al cliente, este determina que dado el funcionamiento propuesto, el *layout* de la aplicación es adecuado, con lo que se toma esta versión como la final. Sin embargo, es importante notar que está sujeta a cambios en caso de que se requiera para el correcto funcionamiento de los *callbacks* que se definirán a continuación.

5.4.3 Definición de *callbacks* de la aplicación web

Una vez definido el *layout* de la aplicación web se procede a definir los *callbacks* que añadirán la interactividad definida por los requerimientos del cliente. El llamado de la aplicación a un *callback* se da cuando una o más propiedades de los elementos definidos en el *layout* sufre una modificación.

Es importante notar que se deben definir los *callbacks* para cada *layout* definido, es decir el *layout* de la figura 5.10 tendrá *callbacks* diferentes a los *layout* de las figuras 5.11 y 5.12. El único *callback* común a los tres *layout* es el que permite dirigir al usuario de una página a otra y esto se debe a que las tres páginas comparten el *layout* inicial sobre el que se cargan. De este modo se pueden listar los siguientes tipos *callbacks*:

- **Callback para el pausado:** este *callback* se encargará de pausar o reanudar la simulación de la *LTM* cuando se pulse el botón de pausado, además debe modificar el valor mostrado en el botón de modo que si el sistema está ejecutándose el botón debe mostrar la palabra *Pausar* y si el sistema está pausado *Reanudar* . Es importante notar que su funcionamiento está sujeto a cambios, pues más adelante debe ser capaz de enviar el comando de pausado a la *LTM* real. Este *callback* se define solo para el *layout* de la visualización *LTM*.

- **Callback para el guardado:** este *callback* se encargará de guardar la representación gráfica actual de la *LTM* cuando se pulse el botón de guardado. En este *callback* solo se debe implementar el guardado del gráfico hecho con *Cytoscape*, ya que la figura de *Plotly* para los *PNodes* contiene un botón que permite almacenar

su contenido. Este *callback* se define solo para el *layout* de la visualización *LTM*.

- **Callbacks para desplegar información:**en este tipo de *callbacks* se tiene el *desplegar información* al hacer *click* sobre un nodo y el desplegar información al pasar el cursor por encima del nodo. La diferencia entre ambos es el tipo de entrada, pues el tipo de salidas y el procesamiento de las entradas son del mismo tipo. Este *callback* se define solo para el *layout* de la visualización de la *LTM*.

- **Callback para actualizar el *layout*:** este *callback* se encargará de modificar el *layout* que el usuario visualiza en un momento dado. Este *callback* se define para todos los tipos de *layouts* disponibles.

- **Callback para actualizar gráficos:** se debe definir un *callback* para el *layout* de la *LTM* y otro para los *PNodes*. El funcionamiento del de la *LTM* consiste en actualizar los elementos del elemento *cyto.Cytoscape*, mientras que el de los *PNodes* consiste en modificar las figuras de los elementos *dcc.Graph*.

A continuación, se realiza una definición detallada de los *callbacks* necesarios en cada *layout*. En dichas definiciones se especificarán los elementos que realizan el llamado al *callback*, así como las propiedades de la aplicación que se modificará.

5.4.4 *Callbacks* del *layout* base

Se plantea la representación del *layout* por medio de un diagrama de elementos jerárquico. Estos diagramas permiten observar la agrupación de los diversos elementos dentro del *layout*. Además serán útiles para describir las relaciones entre las entradas y las salidas de los *callbacks* que se definirán en la siguiente sección.

El primer *layout* que se aprecia es el de la base y se aprecia en la figura 5.13. Sobre esta base se albergarán los demás *layouts* según las interacciones del usuario con la aplicación. En específico el elemento que servirá de contenedor es el elemento de tipo *html.Div* llamado *page-content*, bajo su propiedad *children*.

La entrada al *callback* es el cambio en el *pathname* del elemento *dcc.Location*, cuyo funcionamiento se explicó previamente, mientras que su salida se almacenará como el *children* del elemento *html.Div* en blanco.

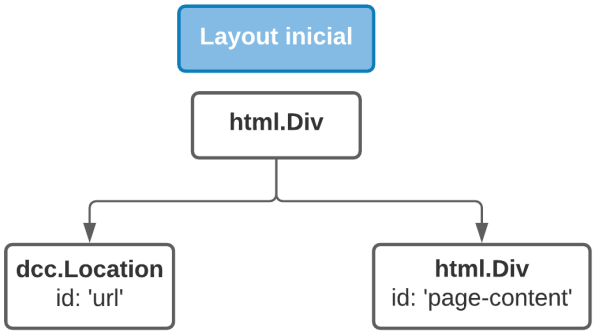

Figura 5.13: Árbol jerárquico para el *layout* de la página inicial. Elaboración propia

Junto con este *layout* base se puede analizar el de la página de inicio, que es la que el usuario ve por defecto cuando abre el *localhost* (dirección *127.0.0.1* en el puerto 8050).

El esquema de elementos jerárquicos para la página de selección se muestra en la figura 5.14. Su funcionamiento se explicó previamente, pero en resumen redirige al usuaio a otras páginas en función de la imagen seleccionada. Para esto utiliza el *callback* descrito para la página base.

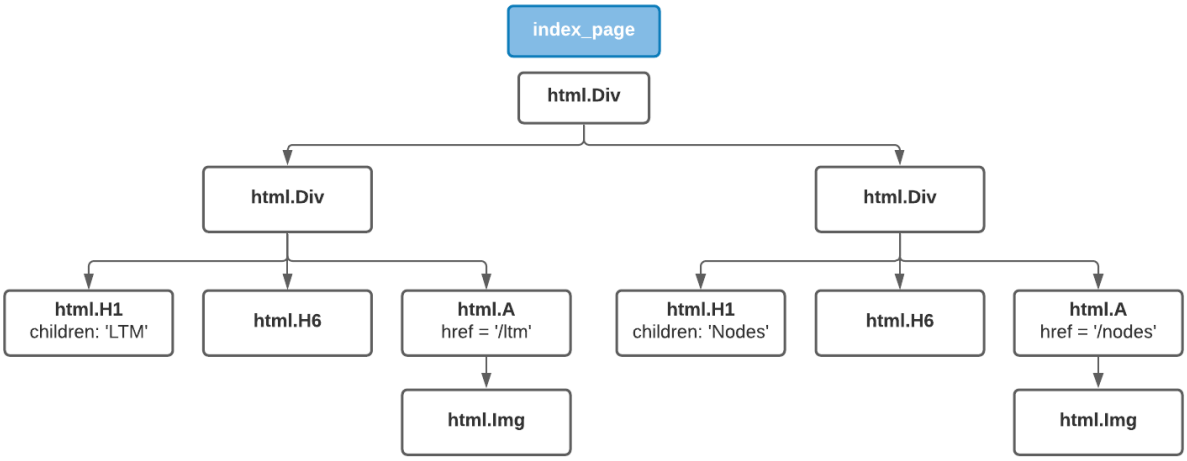

Figura 5.14: Árbol jerárquico para el *layout* de la página de selección. Elaboración propia

5.4.5 *Callbacks* del *layout* de la *LTM*

El esquema de la figura 5.15 muestra las agrupaciones de la página de la *LTM*. Se observan 5 grupos principales. El primero, de izquierda a derecha, correspone al encabezado de la página, el segundo corresponde a los dos botones de la sección superior, el tercero muestra el objeto de *Cytoscape* que permite observar la red, el cuarto se compone de los 4 cuadros de texto donde se despliega la información de un nodo seleccionado. El quinto se basa en un link que permite dirigir al usuario a la página donde se observan los puntos y antipuntos; y un temporizador que permite actualizar periódicamente la gráfica de la red.

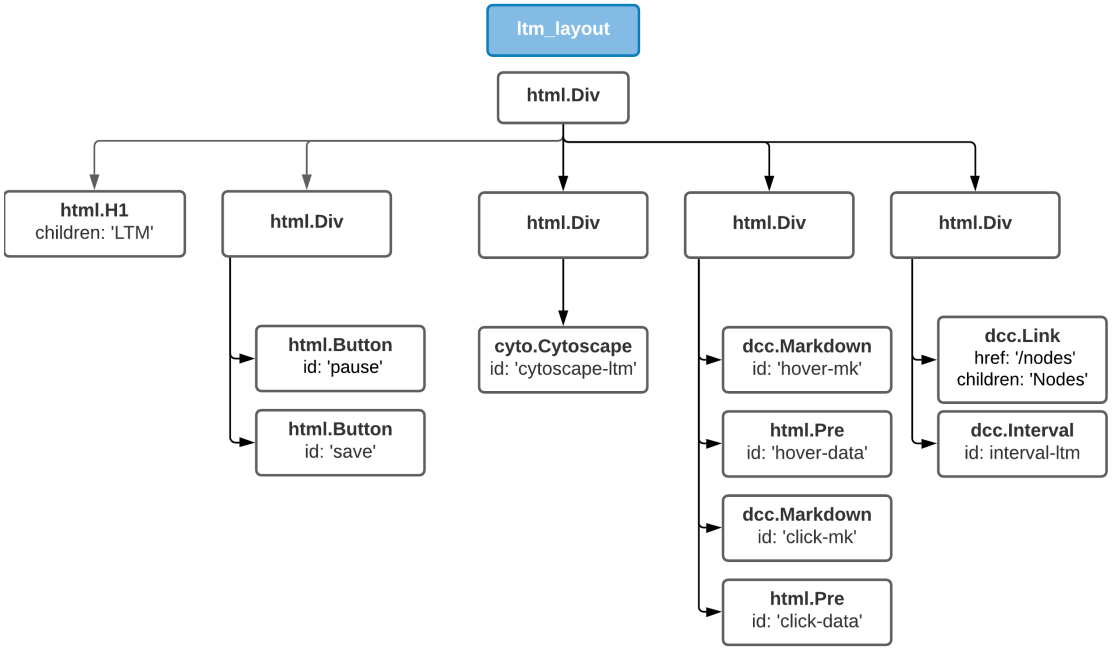

Figura 5.15: Árbol jerárquico para el *layout* de la página de la *LTM*. Elaboración propia

A continuación se hará un análisis de cada *callback* necesario, pasando por las entradas y salidas que la definen y detallando la función del código final con el que se implementa.

Actualización de la gráfica

Este es el *callback* más importante, pues es el que actualiza la visualización de la red. Tiene una única entrada, que es un tipo de temporizador que se activa cada 2 segundos. Esta entrada se describe en el cuadro 5.4.

Cuadro 5.4: Entradas del *callback* de actualización de la gráfica

Entrada		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
interval-ltm	n_intervals	La propiedad aumenta en una unidad cada vez que transcurre un período de tiempo definido por otra propiedad llamada *interval*. Inicialmente, interval tiene un valor de 2000, representando que el callback se ejecuta cada 2000 milisegundos.

La salida de este *callback* será una lista de diccionarios, como la que se representa en el cuadro de texto 5.5. De esta manera, la función que se ejecuta debe realizar la traducción para pasar de una red de *Networkx* a un formato procesable en *Cytoscape*. Dicha salida se describe en el cuadro 5.5

Cuadro 5.5: Salidas del *callback* de actualización de la gráfica

Salida		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
cytoscape-ltm	elements	La propiedad se modificará para que contemple la actualización del número de nodos en la red así como de sus activaciones. Se debe recordar que esta es una lista de diccionarios, donde cada diccionario representa una arista o un nodo de la red.

La implementación de este *callback* en el código del apéndice 8.11 se detalla en la función **update_fig_ltm**.

Pausado o reanudado

Este *callback* debe pausar o reanudar el experimento en la *LTM*. Por esta razón único donde se deberá definir, en secciones posteriores, un método por el cual se puedan enviar comandos. De momento, para efectos del código de visualización, la función

que se ejecutará modificará el valor de una variable de tipo booleana que representa si el sistema está pausado o no. Además, se ha definido cambiar el texto del botón, de modo que si se ha pausado, el botón debe indicar *Reanudar* y en caso contrario *Pausar*. Además, dado que el sistema está pausado, la *LTM* no debería enviar nueva información, por lo que se debe definir que el temporizador deba esperar un período muy largo de tiempo antes de volver a actualizar la gráfica.

De este modo, se define la entrada descrita en el cuado 5.6 para el pausado del sistema.

Cuadro 5.6: Entradas del *callback* de pausado o reanudado

Entrada		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
pause	n_clicks	La propiedad aumenta en una unidad cada vez que el botón es presionado. De este modo, el callback se ejecuta cada vez que se presiona el botón de pausa.

Y las salidas del mecanismo de pausado se definen en el cuadro 5.7.

Cuadro 5.7: Salidas del *callback* de pausado o reanudado

Salida		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
pause	children	La propiedad almacena el texto que se muestra en el botón de pausado. Si el sistema está en pausa, despliega Reanudar y si está en ejecución despliega Pausar
interval-ltm	interval	La propiedad almacena el período con el que se actualiza la gráfica con la red. Si el sistema está en pausa, esta actualización también, por lo que el período debe ser alto.

En el código del apéndice 8.11, la función que implementa este *callback* se llama **update_pause**.

Guardado

El guardado funciona de manera similar al pausado, en el hecho de que se ejecuta

cuando se pulsa un botón, pero a diferencia del primero, sólo debe cambiar el valor de una variable en vez de 2. Dicha variable y el valor que debe tomar, se detallaron en el cuadro de texto 5.7

De este modo la entrada del *callback* de guardado se detalla en el cuadro 5.8.

Cuadro 5.8: Entradas del *callback* de guardado

Entrada		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
save	n_clicks	La propiedad aumenta en una unidad cada vez que el botón es presionado. De este modo, el *callback* se ejecuta cada vez que se presiona el botón de guardado.

Y, como se detalló en el cuadro de texto 5.7, la salida debe ser la que se detalla en el cuadro 5.9.

Cuadro 5.9: Salidas del *callback* de guardado

Salida		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
cytoscape-ltm	generateImage	La propiedad se modificará para que se almacene la red como una imagen en formato png. Recordar que este es un diccionario donde se especifica el formato de la imagen, el nombre y una acción.

La implementación de este *callback* en el código del apéndice 8.11 se detalla en la función **save_graph**.

Desplegar información con click sobre nodo

Esta función se implementa para poder visualizar información en forma de texto sobre las conexiones de un nodo en específico y no sólo de forma visual. A veces esto puede resultar útil, si la cantidad de aristas y nodos en la red es excesiva y dificultan su comprensión.

El abordaje de *Cytoscape* realizado en secciones anteriores, permitió definir el modo en el que se obtiene la información de un nodo al mover o hacer click con el cursor

sobre un nodo. Con la información del nodo, se puede recorrer luego sus nodos vecinos para poder leer y desplegar la siguiente información: nombre, activación y tipo. De esta manera se puede definir la entrada descrita en el cuadro 5.10.

Cuadro 5.10: Entradas del *callback* de despliegue de información con click

Entrada		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
cytoscape-ltm	tapNodeData	La propiedad cambia de valor cada vez que se hace click sobre un nodo. Almacena un diccionario, sólo de lectura, que contiene todala información del nodo almacenada en la propiedad *elements* del objeto de *Cytoscape*.

Ahora bien, la salida de este cursor es la información que se despliega en dos cuadros de texto. En el primero se muestra el título indicando el nombre del nodo y en el segundo se detallan las activaciones, los tipos y los nombres del nodo y de sus vecinos. Estas salidas se describen en el cuadro 5.11

Cuadro 5.11: Salidas del *callback* de despliegue de información con click

Salida		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
click-data	children	La propiedad almacena el texto que se muestra en el cuadro de texto. Este despliega la información con el nombre, la activación y tipo del nodo y de sus vecinos.
click_mk	children	La propiedad almacena el texto que se muestra en el cuadro de texto. Este despliega el nombre del nodo seleccionado.

Este *callback* se define en la función **display_info** del código presente en el apéndice 8.11. Esta función es la misma que se emplea para el siguiente callback.

Desplegar información con cursor encima del nodo

La funcionalidad es la misma que en el anterior, Lo que cambia es el nombre de la entrada y de las salidas. Las entradas de este *callback* se definen en el cuadro 5.12.

Cuadro 5.12: Entradas del *callback* de despliegue de información con cursor encima del nodo

Entrada		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
cytoscape-ltm	mouseoverNodeData	La propiedad cambia de valor cada vez que se pasa el cursor encima del nodo. Almacena un diccionario, sólo de lectura, que contiene toda la información del nodo almacenada en la propiedad *elements* del objeto de *Cytoscape*.

La salida, puesto que proviene de la misma función, no cambia en tipo y funcionalidad, pero si en nombre, y se describe en el cuadro 5.13.

Cuadro 5.13: Salidas del *callback* de despliegue de información con cursor encima del nodo

Salida		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
hover-data	children	La propiedad almacena el texto que se muestra en el cuadro de texto. Este despliega la información con el nombre, la activación y tipo del nodo y de sus vecinos.
hover_mk	children	La propiedad almacena el texto que se muestra en el cuadro de texto. Este despliega el nombre del nodo seleccionado.

5.4.6 *Callbacks* del *layout* de los *PNodes*

De igual manera que para los *layouts* anteriores, se desarrolló un esquema de elementos jerárquicos para la página que muestra los puntos y antipuntos de los *PNodes*. En la figura 5.16 se observan 5 grupos principales, que se describirán recorriéndolos de izquierda a derecha. El primer grupo contiene el encabezado de la página. El segundo y tercer grupo se forman de los mismos tipos de elementos, y se emplean para mostrar las gráficas con los puntos y antipuntos, además de seleccionar el nodo por observar. El cuarto grupo se utiliza para definir el tipo de gráficas por mostrar, así como para filtrar

las percepciones que se grafican. El quinto y último grupo, se compone de un elemento que permite cambiar a la página de la *LTM* y un temporizador.

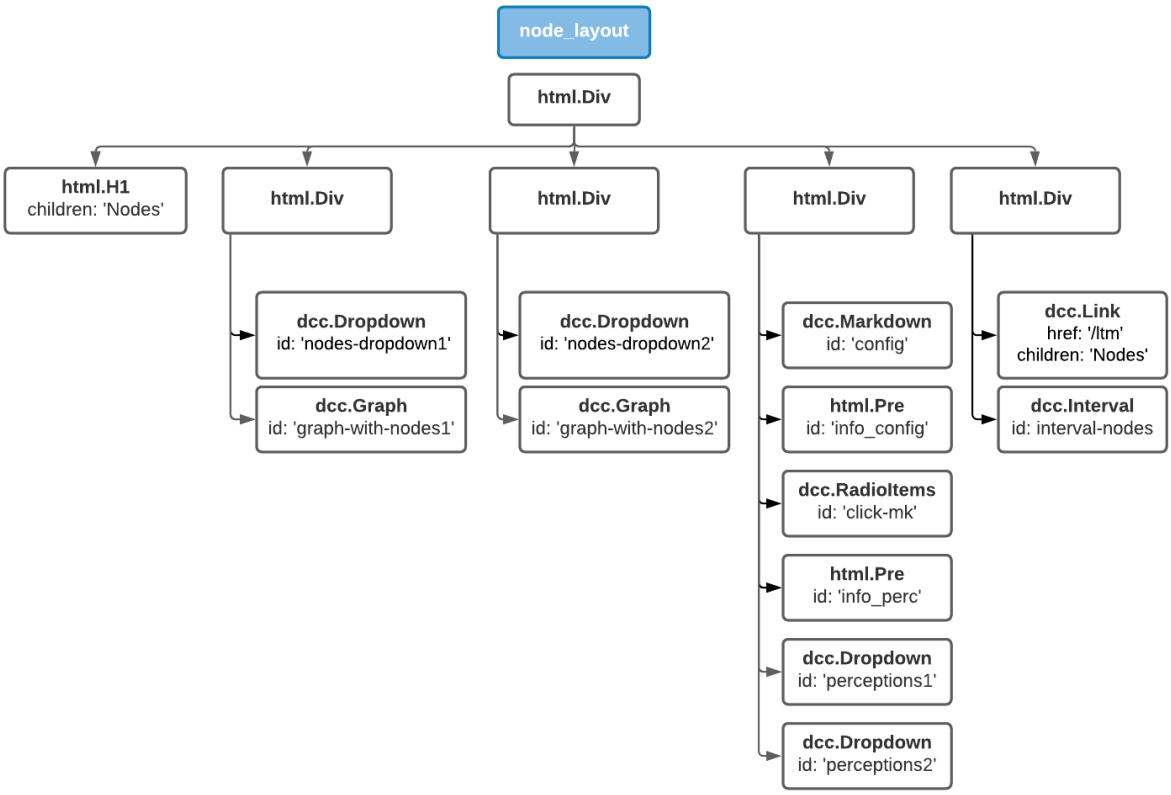

Figura 5.16: Árbol jerárquico para el *layout* de la página de los *PNodes*. Elaboración propia

En esta página solo se tiene un *callback*, que se emplea para actualizar las gráficas mostradas. Dichas gráficas se deben actualizar si se cumple una de las siguientes condiciones: han pasado 2 segundos desde la última actualización, se ha cambiado una de las percepciones por graficar, se ha cambiado uno de los *PNodes* por graficar o se ha cambiado el tipo de gráfica por mostrar (polar o rectangular).

Guiándose en el esquema de elementos jerárquicos de la figura 5.16, se plantean las entradas del cuadro 5.14 que activarán el *callback*, indicando el nombre del elemento, su propiedad y una explicación de cómo cambia esta propiedad.

Cuadro 5.14: Entradas del *callback* para actualizar las gráficas de los *PNodes*

Entrada		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
cytoscape-ltm	n_intervals	La propiedad aumenta en una unidad cada vez que transcurre un período de tiempo definido por otra propiedad llamada *interval*. Inicialmente, *interval* tiene un valor de 2000, representando que el *callback* se ejecuta cada 2000 milisegundos.
nodes-dropdown1	value	La propiedad contiene el nombre del *PNode* que se observará en el gráfico de la izquierda. Cambia de valor cuando se selecciona una opción de la lista desplegable.
nodes-dropdown2	value	La propiedad contiene el nombre del *PNode* que se observará en el gráfico de la derecha. Cambia de valor cuando se selecciona una opción de la lista desplegable.
perceptions1	value	La propiedad contiene el nombre de una de las percepciones por graficar. Cambia de valor cuando se selecciona una opción de la lista desplegable.
perceptions2	value	La propiedad contiene el nombre de una de las percepciones por graficar. Cambia de valor cuando se selecciona una opción de la lista desplegable.
radio_type	value	La propiedad contiene el tipo de gráfica por observar. Cambia de valor cuando se selecciona una opción de un *radio item*

Así mismo, se definen las salidas del *callback* en el cuadro 5.15. Se observa que no sólo se actualizan las figuras que se muestran en pantalla, sino que también se deben actualizar las opciones de percepciones y *PNodes* por graficar. Las primeras porque inicialmente no se conocen, y las segundas porque se van creando nuevos nodos conforme avance el experimento en la *LTM*.

Cuadro 5.15: Salidas del *callback* para actualizar las gráficas de los *PNodes*

Salida		**Funcionamiento**
Nombre del elemento	**Nombre de la propiedad**	
nodes-dropdown1	options	La propiedad se modificará para aumentar el número de opciones en caso de que se haya agregado un nuevo *PNode*. Esta propiedad es una lista de diccionarios.
nodes-dropdown2	options	La propiedad se modificará para aumentar el número de opciones en caso de que se haya agregado un nuevo *PNode*. Esta propiedad es una lista de diccionarios.
perceptions1	options	La propiedad se modificará para actualizar las percepciones que se pueden graficar. Esta propiedad es una lista de diccionarios.
perceptions2	options	La propiedad se modificará para actualizar las percepciones que se pueden graficar. Esta propiedad es una lista de diccionarios.
graph-with-nodes1	figure	La propiedad contiene la figura que se despliega con los puntos y antipuntos. La figura dependerá del nodo seleccionado en *nodes-dropdown1*, el tipo de gráfica y las percepciones escogidas.
graph-with-nodes2	figure	La propiedad contiene la figura que se despliega con los puntos y antipuntos. La figura dependerá del nodo seleccionado en *nodes-dropdown2*, el tipo de gráfica y las percepciones escogidas.

Las funciones que llevan a cabo este *callback* se pueden observar en el código del apéndica 8.11 con el nombre de **update_pnode_figure** y **update_fig_nodes**.

La función **update_fig_nodes**, es donde se lleva a cabo toda la funcionalidad del *callback*. En ella se actualizan las opciones que puede escoger el usuario y las gráficas mostradas. En caso de que alguna opción escogida sea inválida o no está definida, se presentará una gráfica en blanco.

La función **update_pnode_figure** retorna una traza de *Plotly* polar o rectangular, con los datos almacenados en un diccionario a los que se accede a través de la llave definida por el nombre del *PNode*. Para esta traza, se definió que el color de los puntos sea azul y el de los antipuntos sea rojo.

5.5 Validación del sistema de visualización

La etapa de validación del sistema de visualización consiste en definir las pruebas que permitan determinar si la funcionalidad del código de visualización, definido en la etapa anterior, satisfacen las necesidades del cliente.

Para esta etapa se plantea el uso del simulador de la *LTM* desarrollado en la sección 5.4.1 en conjunto con el código de visualización. Es importante recordar que esto se hace pues no se tiene acceso a la *LTM* real.

Ahora bien, es necesario definir las condiciones en las que operará la simulación de la *LTM*. La simulación planteada se basará en el experimento descrito en [8]. En dicho experimento se utiliza un robot Baxter de dos brazos, frente a una mesa donde se coloca un objeto largo (un cilindro de 7 cm de radio y altura de 14.7 cm), un objeto pequeño (un cilindro de 3 cm y altura de 6 cm) y una canasta. El objetivo del experimento es que el robot desarrolle una serie de conocimientos por la interacción con los dos objetos y una recompensa negativa o positiva tras ejecutar una acción. Es decir, se busca que el robot genere nuevos conocimientos por refuerzo. Dichos conocimientos serán contextos representados por *CNodes*, que relacionarán una *policy* o acción, con una meta y un modelo de mundo definido. El mecanismo de la generación de estos nodos se definió en la sección 3.2.1.

El robot no es capaz de cubrir toda la mesa, como se observa en la figura 5.17, de manera que si necesita levantar un objeto fuera de su alcance, debe solicitarle primero al experimentador que se lo coloque más cerca. Además, si el robot desea colocar un objeto en la canasta, y esta se encuentra fuera de alcance, debe lanzar el objeto a la canasta.

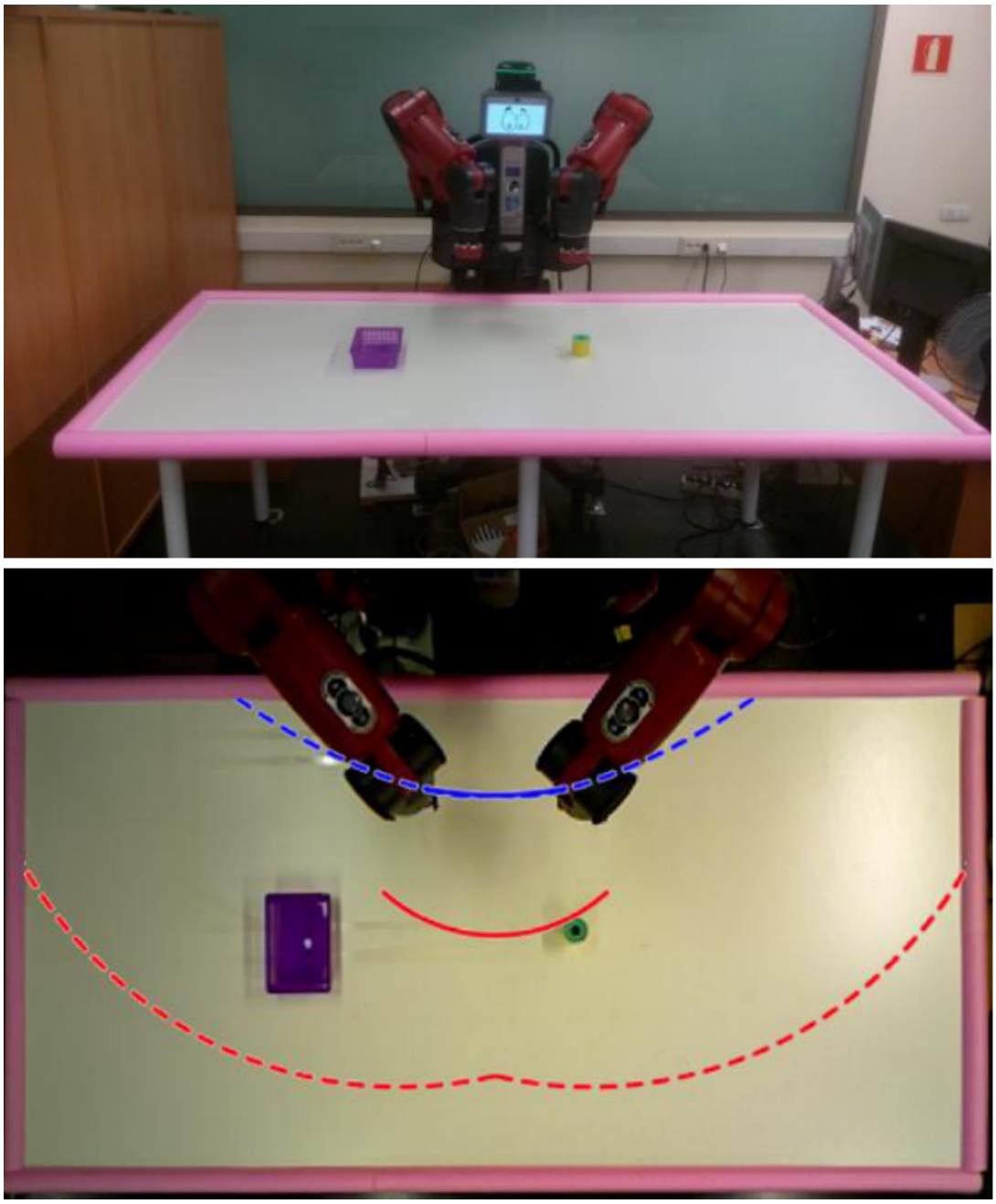

Figura 5.17: Configuración del robot Baxter para el experimento. Recuperado de [8]

Los sensores usados en el experimento son una cámara RGB en la cabeza del robot y sensores de fuerza asociados a los motores de sus articulaciones. Para disminuir la dimensionalidad del espacio perceptual se proce a redescribir la información de la cámara en forma de la distancia, el ángulo al centro y el radio de dos objetos, es decir alguno de los dos cilindros y la canasta. Además, se redescribe la información de los sensores

de fuerza, en la forma de una señal binaria que representa la presencia o ausencia de un objeto en el gripper del robot. De este modo el espacio perceptual de la *LTM* para este experimento se compone de 6 dimensiones continuas y 2 discretas. De este modo, se define el primer conjunto de nodos básicos de conocimiento que se cargan al inicio del experimento, y que corresponden a los nodos de tipo *percepción*. Una descripción de los nodos de percepción se presenta en el cuadro 5.16.

Cuadro 5.16: Nodos de percepción para experimento de validación del sistema de visualización

Nombre	**Tipo**	**Descripción**
basket size	Continua	Tamaño de la canasta donde se podría introducir un objeto.
basket distance	Continua	Distancia entre el centro de la cámara y el centro de la canasta.
basket angle	Continua	Ángulo de la canasta con respecto a una referencia definida por el experimentador.
object size	Continua	Tamaño del objeto.
object distance	Continua	Distancia entre el centro de la cámara y el objeto.
object angle	Continua	Ángulo del objeto con respecto a una referencia definida por el experimentador.
object in left hand	Discreta	Presencia o ausencia de un objeto en el gripper izquierdo.
object in right hand	Discreta	Presencia o ausencia de un objeto en el gripper derecho.

El otro conjunto de nodos de conocimiento iniciales que el experimentador definió es el de *policies*. Estos se detallan en el cuadro 5.17.

Cuadro 5.17: Nodos de *policies* para experimento de validación del sistema de visualización

Nombre	Descripción
Grasp_object	Usar un gripper para coger un objeto.
Ask_nicely	Solicitar al experimentador que acerque el objeto al alcance del robot.
Change_hands	Mover el objeto de un gripper a otro.
Put_object_in	Colocar el objeto en un receptáculo.
Sweep_object	Desplaza el objeto a la línea central de la mesa.
Throw_object	Lanza un objeto hacia una posición.
Hold_with_two_hands	Usa ambos brazos para agarrar el objeto. Sin grippers
Put_objects_near	Acerca el objeto cerca de la base del robot.

Además, se definen dos metas que el robot busca cumplir. Una de ellas es colocar el objeto en la canasta o colocar el objeto cerca de la base. Para el experimento definido en [8] el experimentador es quien activa una meta o la otra, mientras que en las pruebas de validación del sistema de visualización, esto se hará de manera aleatoria pues así es como se ha definido que funcione el simulador. Aparte de las metas, se definen dos modelos de mundo. En el primero de ellos el robot tiene ambos *grippers*, con los que puede levantar objetos pequeños, mientras que en el segundo el robot no cuenta con ningún gripper, de manera que debe usar ambos brazos para levantar objetos.

Esta definición de *policies*, metas y modelos de mundo; permite la definición de los contextos que se presentan en el cuadro 5.18.

Cuadro 5.18: Posibles contextos aprendidos por el agente para el experimento de integración del sunsistema

CN	**Modelo de mundo**	**Meta**	***Policy***
0	*W1: Grippers low friction*	*G1: Object in basket*	*Grasp_object*
1			*Ask_nicely*
2			*Change_hands*
3			*Put_object_in*
4			*Sweep_object*
5			*Hold_with_two_hands*
6			*Throw_object*
7		*G2: Object near robot*	*Ask_nicely*
8			*Grasp_object*
9			*Put_object_near*
10			*Sweep_object*
11			*Hold_with_two_hands*
12	*W2: No grippers high friction*	*G1: Object in basket*	*Ask_nicely*
13			*Sweep_object*
14			*Hold_with_two_hands*
15			*Put_object_in*
16			*Throw_object*
17		*G2. Object near robot*	*Ask_nicely*
18			*Sweep_object*
19			*Hold_with_two_hands*
20			*Put_object_in*

Es importante notar que los contextos mostrados en el cuadro anterior pueden no llegar a cumplirse en la simulación de la *LTM*, pues esta se basa en un funcionamiento aleatorio, en donde no se analiza si la *policy* ejecutada tiene sentido para el mundo y la meta actual. Sin embargo, esto no representa un problema, pues el simulador solo se diseñó para generar una red similar a la LTM, donde las activaciones de los nodos se actualicen en cada iteración, y donde se crean nodos de contexto (*CNodes*) cuando las condiciones del experimento lo ameriten. Este problema se resolverá con la integración con la *LTM*, pues en esta si se cumplen dichos contextos.

Por otro lado, para simular las percecpiones que el robot se podrían encontrar en el experimento real, el cliente ha facilitado archivos en formato *csv*, donde se almacena para cada *PNode* los puntos y antipuntos derivados de un experimento real. De este

modo, se configura el simulador de la *LTM* para que los puntos y antipuntos sean leídos de estos archivos con *Pandas*.

Una vez definido el experimento, se procede a definir lo que se desea extraer de este. Puesto que se desea analizar la funcionalidad del código de visualización, se plantea guardar imágenes de la red y de gráficas de puntos y antipuntos de PNodes definidos, para diferentes números de iteraciones. Para observar los resultados de esta etapa se puede observar la sección 6.2.

5.6 Integración del sistema

Esta etapa consiste en definir la forma en la que el código de visualización accederá la información de la red de nodos de la *LTM* y cómo enviará el comando de pausado o reanudado. Además, se deberá realizar cambios en el código de visualización definido en la etapa anterior, en función de los nuevos requerimientos que salgan de esta etapa.

Inicialmente se debe definir qué información se desea extraer de la *LTM*. En primer lugar, para poder construir la red del lado del código de visualización, es necesario saber el momento en el que se ha creado un nodo nuevo en la *LTM*. Esto implica conocer además su activación inicial, el tipo de nodo, el nombre que se le ha asignado, con qué nodos está conectado y el tipo de nodos al que se ha conectado; y en caso de ser un *PNode*, el nombre de las percepciones que lo definen. Además, es necesario saber cuando un nodo ya existene actualiza el valor de su activación.

En segundo lugar, se debe saber cuándo se ha generado una nueva percepción, si es un punto o antipunto y a qué *PNode* está asociado.

En tercer lugar, el código de visualización debe estar al tanto de la iteración actual del experimento, de la recompensa actual, de la *policy* que generó dicha recompensa, del modelo de mundo en el que opera el agente y de la meta actual. Esta información es relevante pues es la que se despliega en el título de la gráfica.

Para esta etapa se podría realizar un guardado de los datos desde la *LTM*, para luego ser leídos por el código de visualización. Esto sería guardando la red construida con *Networkx* en la *LTM*, con algún método que permita almacenar en memoria la red en formato de texto. Sin embargo, esta solución es ineficiente pues agota el recurso de memoria en el disco duro del ordenador, además de aumentar el tiempo de ejecución de la *LTM*.

Ante esta situación, se plantea integrar el sistema de visualización como un nodo de *ROS*. Con esta propuesta, el sistema se suscribiría a una serie de canales de comunicación donde la *LTM* publicaría la información detallada previamente, cada vez que sea necesario. Esta solución implica un cambio tanto en el sistema de visualización como en la propia *LTM*, pues antes no debía publicar dicha información por medio de *ROS*, pues no era necesaria para ninguno de los demás nodos que componen la *MDB*.

Además de comportarse como suscriptor, el nodo de visualización debe actuar como *publisher* para poder enviar un mensaje a la *LTM* indicando que debe pausarse o reanudarse el experimento.

El siguiente paso a seguir es definir los canales de comunicación entre la *LTM* y el sistema de visualización, por donde se hará el intercambio de la información. Seguidamente se deberán definir los mensajes que serán enviados por estos canales, indicando en estos el tipo de variable que se envía y el identificador asociado, además de indicar cúando se envían estos datos por parte de la *LTM*. Tras estas definiciones, se abordará el diseño de los *callbacks* de *ROS* para transmitir datos entre *LTM* y sistema de visualización.

5.6.1 Definición de *topics*

Tras conversar con el cliente sobre la implementación del sistema de visualización como un nodo de *ROS*, este comentó que previamente se habían definido una serie de canales de comunicación para la publicación de información relacionada a la creación

de nuevos nodos. Esto se deba a que en la implementación de la *MDB* se plantea que nodos distintos a la *LTM* pueden crear nuevos nodos de conocimiento (como por ejemplo nuevas metas).

El cliente definió un canal de comunicación específico para cada tipo de nodo. De este modo, el sistema de visualización deberá suscribirse a los siguientes canales donde la *LTM* publicará un mensaje cada vez que un nodo ha sido creado o se ha actualizado el valor de su activación.

- /mdb/ltm/perception
- /mdb/ltm/p_node
- /mdb/ltm/c_node
- /mdb/motiven/goal
- /mdb/stm/forward_model
- /mdb/stm/policy

Para la publicación de nuevos puntos y antipuntos se definió un canal llamado */mdb/ltm/p_node_update*. En este canal se deberá publicar el nombre del *PNode* al que se deben agregar las percepciones, el punto como una lista y un identificador que permita determinar si se trata de un punto o un antipunto. En este canal se publicará desde la *LTM*, cada vez que se añada una nueva percepción a un *PNode*.

Finalmente, el nodo de visualización se debe suscribir a un canal donde la *LTM* publique la información del experimento. Para ello, se definió el canal llamado */mdb/ltm/info*. En la *LTM*, tras la finalización de una iteración del experimento, se publicará en un mensaje la iteración actual del experimento, la recompensa actual, la *policy* que generó dicha recompensa, el modelo de mundo en el que opera el agente y la meta actual.

Adicionalmente, el nodo de visualización deberá publicar en un canal que le permita enviar comandos de pausado o reanudado al nodo de la *LTM*. Al conversar con el cliente sobre este requerimiento, indicó que se podría utilizar un canal ya definido, llamado */mdb/baxter/control*, en donde se publican comandos hacia la *LTM*.

5.6.2 Definición de mensajes

La definición de mensajes se realizó en paralelo a la definición de los canales de comunicación.

De igual forma como sucedió con los canales de comunicación, el cliente había definido previamente un conjunto de mensajes que se publicarían con la creación de un nuevo nodo. Inicialmente, el cliente había definido un mensaje diferente para cada canal, pero la información que se publicaba para cada tipo de nodo era la misma, por lo que se definió un único mensaje que se publicaría en cada canal, exceptuando al de los *PNodes*. Este mensaje se puede observar en el cuadro de texto 5.8.

```
1 string command
2 string id
3 string[] neighbor_ids
4 string[] neighbor_types
5 float64 activation
6 string execute_service
7 string get_service
8 string class_name
9 string language
```

Listing 5.8: Mensaje NodeMsg.

Para efectos de la construcción de la red en el sistema de visualización, es necesario comprender el significado de los primeros 5 campos del mensaje anterior. El *string* ***command*** indicará al sistema de visualización si la información que se publica es de un nodo nuevo, o si el mensaje se debe usar para actualizar la activación del nodo. Para el primer caso, el valor de **command** será *new* y para el segundo será *update*.

El *string* ***id*** representa el nombre que la *LTM* ha asignado al nodo. El campo ***neighbor_ids*** contiene una lista con los nombres de los nodos con los que se encuentra

conectado, y se usa en conjunto con el campo ***neighbor_types*** para definir las aristas en la red, así como el color y grosor con el que se observarán.

Finalmente, el campo ***activation*** contendrá el valor de la activación actual del nodo.

Ahora bien, para la publicación de los *PNodes* se definió un mensaje similar al anterior, con la diferencia de que se agrega un campo llamado ***names***, que contendrá una lista con los nombres de las sensorizaciones que definen al *PNode*. Dicho mensaje se presenta en el cuadro de texto 5.9.

```
string command
string id
string[] names
string[] neighbor_ids
string[] neighbor_types
float64 activation
string execute_service
string get_service
string class_name
string language
```

Listing 5.9: Mensaje PNodeMsg.

Para la publicación de los puntos y antipuntos se definió el siguiente mensaje en donde el primer campo que es relevante para el sistema de visualización es ***id***, pues representa el *PNode* al que se debe agregar el punto o antipunto. El segundo campo de interés es ***point*** pues contiene una lista con las percepciones que se deben agregar al nodo y finalmente el campo ***confidence*** permite determinar si la percepción es un punto o antipunto. Este mensaje se describe en el cuadro de texto 5.10.

```
string command
string id
float64[] point
float64 confidence
```

Listing 5.10: Mensaje PointMsg.

Para obtener la información del experimento se definió un mensaje con tres campos de tipo *string* que representan la *policy* , el modelo de mundo y la meta actuales, un campo de tipo entero con la información de la iteración actual y un campo de tipo *float*

con la recompensa actual. Este mensaje se describe en el cuadro de texto 5.11.

```
int32 iteration
string current_world
string current_policy
float64 current_reward
string current_goal
```

Listing 5.11: Mensaje InfoMsg.

Finalmente se debe definir el mensaje para enviar los comandos desde el sistema de visualización a la *LTM*. Este mensaje ya estaba definido por el cliente, y el único campo que interesa para efectos del sistama es el ***command*** de tipo *string*. El sistema de visualización podrá enviar 3 comandos diferentes: *pause* para pausar el experimento en la *LTM*, *continue* para reanudar el experimento y *publish_ltm* para solicitar a la *LTM* que envíe la información de todos los nodos. Este último comando se utilizará cuando se inicializa el nodo de visualización, pues puede suceder que el nodo de la *LTM* inicie antes. El mensaje que permite cumplir con esta funcionalidad se detalla en el cuadro de texto 5.12.

```
string command
string world
bool reward
```

Listing 5.12: Mensaje ControlMsg.

5.6.3 Definición de los *callbacks* de *ROS*

La tercera etapa de la integración del sistema consiste en definir los *callbacks* del nodo de *ROS*. Estos son distintos de los definidos en la sección 5.4.3, pues son llamados a funciones que se dan cuando se ha publicado un mensaje nuevo en alguno de los canales a los que el nodo está suscrito.

De este modo, será necesario definir un *callback* para cada uno de los mensajes que se detallaron en la subsección anterior. Es importante notar que algunas de estas funciones se encuentran definidas en el código de visualización desarrollado en la sección 5.4, simplemente se deben adaptar a la nueva implementación en *ROS*.

El primer callback por definir es el que manejará la creación y actualización de los nodos dentro del sistema de visualización. Este se activará cada vez que la *LTM* publique en alguno de los siguientes canales:

- /mdb/ltm/perception
- /mdb/ltm/p_node
- /mdb/ltm/c_node
- /mdb/motiven/goal
- /mdb/stm/forward_model
- /mdb/stm/policy

Se basará de la definición de los mensajes definidos en los cuadros de texto 5.8 y 5.9. El *callback* deberá, en primer lugar, determinar según el comando publicado, si el nodo que se publica es nuevo o si ya existe en la red almacenada en la variable *graph*. En caso de que el nodo sea nuevo, se debe analizar el tipo de nodo para determinar dos variables nuevas: la posición y el color del nodo. Además, en caso de ser un *PNode*, se debe agregar un nuevo *DataFrame* al diccionario *pnode_dict*, cuyas columnas serán el nombre de las percepciones asociadas al nodo. Seguidamente, se deben añadir las aristas según el nombre de sus vecinos y el tipo de cada uno de ellos. En caso de que el comando indique que se debe actualizar la activación del nodo, la función debe modificar dicho valor dentro de la variable *graph*. El pseudocódigo de esta función se muestra en el cuadro de texto 5.13.

```
Leer el comando del mensaje
Leer activacion del nodo en el mensaje
Leer id del nodo en el mensaje
if comando es 'new'
    Definir el color y la posicion segun el tipo de nodo
    if tipo de nodo es 'PNode'
        Agregar un nuevo DataFrame al diccionario pnode_dict
```

```
8       Agregar el nodo a la red, con los atributos: color, posicion,
    activacion y nombre
9
10      Leer los vecinos del nodo
11      Leer el tipo de nodo de los vecinos del nodo
12      for nodo in vecinos
13          Definir el color y el grosor de la arista segun el tipo del
    nodo y el tipo de su vecino
14          Agregar arista entre el nodo y su vecino, con los atributos:
    color y grosor
15
16  elif comando es 'update':
17      Actualizar la activacion del nodo en la red
```

Listing 5.13: Pseudocódigo del *callback* para añadir nodos.

Para observar la implementación real del pseudocódigo anterior se puede observar la función **add_node_callback()** en el apéndice 8.11.

El segundo callback por definir es el que se encarga de añadir puntos y antipuntos a los *PNodes* de la red. Este debe leer en primer lugar el identificador del nodo, pues esta es la llave para acceder al nodo dentro del diccionario *pnode_dict*. Es importante observar que el *DataFrame* asociado a dicha llave, ya ha sido creado al recibir el nodo de la *LTM* y crearlo dentro del sistema de visualización. A continuación, se debe añadir la percepción definida por el punto del campo *point* y su *confidence*, como una nueva fila del *DataFrame* del *PNode* asociado. De este modo, se puede definir el pseudocódigo para esta función en el cuadro de texto 5.14.

```
1 Leer el id del nodo
2 Leer las percepciones en points
3 Leer la activacion en confidence
4 Unir en una lista las percepciones y la activacion
5 Agregar la lista de puntos como una nueva fila del DataFrame asociado
    al nodo
```

Listing 5.14: Pseudocódigo del *callback* para añadir puntos y antipuntos.

Para observar la implementación real del pseudocódigo anterior se puede observar la función **add_point_callback()** en el apéndice 8.11. Esta función seré ejecutada cada vez que se publique un mensaje nuevo en el topic */mdb/ltm/p_node_update*.

El tercer y último *callback* por definir, corresponde a la actualización de la información del experimento de la *LTM*. Esta información es importante para el sistema de

visualización porque definen el título de la red. Se debe recordar que por las limitaciones de *Cytoscape*, el título será el texto que aparece encima de un nodo de la red, cuyo tamaño es bastante pequeño en comparación a los demás y cuya opacidad se mantiene en 0 para no observarlo en la gráfica. El pseudocódigo de este *callback* se define en el cuadro de texto 5.15.

```
1
2 Leer y almacenar la iteracion del campo iteration
3 Leer y almacenar el modelo mundo actual del campo current_world
4 Leer y almacenar la meta actual del campo current_goal
5 Leer y almacenar la recompensa actual del campo current_reward
6 Leer y almacenar la policy actual del campo current_policy
7 Modificar el titulo de la grafica
```

Listing 5.15: Pseudocódigo del *callback* para actualizar información del experimento.

Finalmente, se deben definir las funciones que enviarán comandos a la *LTM* desde el sistema de visualización. Para ello, el sistema de visualización debe configurarse como un *publisher* que enviará mensajes al canal */mdb/ltm/info*. En la siguiente subsección se abordará el proceso de inicialización del nodo de visualización, donde se especificará la forma en la que el nodo se suscribe y publica a los nodos correspondientes.

Se debe recordar que desde el sistema de visualización se pueden enviar tres comandos: *pause*, *continue* y *publish_ltm*. Para los primeros dos se plantea modificar el *callback* de la aplicación web destinado a la pausa del experimento, para que pueda enviar dichos comandos, tal y como se muestra en las líneas 4 y 9 del código del cuadro de texto 5.16.

```
1 def update_pause(self, n_clicks):
2         self.paused = not(self.paused)
3         if(self.paused):
4             self.control_publisher.publish(command = "pause", world =
    "", reward = False)
5             rospy.loginfo('Se ha pausado el proceso')
6             return 'Reanudar', 1*1000*60*60
7         else:
8             rospy.loginfo('Se ha reanudado el proceso')
9             self.control_publisher.publish(command = "continue", world
     = "", reward = False)
10            return 'Pausar', 1000
```

Listing 5.16: Función para el envío del pausado y continuación del experimento.

Y para enviar el mensaje solicitando a la *LTM* que publique todos sus nodos se define la función descrita en el cuadro de texto 5.17.

```
def ask_for_ltm(self):
    rospy.logdebug('Asking for current nodes in LTM')
    self.control_publisher.publish(command = "publish_ltm", world = ""
   , reward = False)
```

Listing 5.17: Función para solicitar todos los nodos en la *LTM*.

La función anterior se ejecutará cada vez que se inicializa el sistema de visualización.

5.6.4 Consideraciones de la inicialización del nodo de ROS

Todos los nodos de *ROS* dentro de la *MDB* necesarios para la ejecución de un experimento, se inicializan desde un archivo de tipo *launch*, Dicho archivo permite configurar algunos parámetros de *ROS*, y permiten ejecutar un conjunto de nodos de diversos paquetes, además de pasar argumentos para la inicialización de cada uno de ellos.

Para el caso del sistema de visualización, el paquete que permite su implementación se llamará *mdb_view*. La estructura de este paquete se aborda en la siguiente subsección. Dicho paquete contiene un nodo llamado *mdb_view*, que será ejecutado desde el archivo de tipo *launch*. A este nodo se le pasarán 3 argumentos: la dirección en el ordenador donde se desea almacenar los productos de la herramienta, la dirección en el ordenador de un archivo de configuración en formato *yaml* y el nivel de registro.

La función del archivo de configuración es inicializar el nodo de visualización, ya que contiene información que el nodo puede interpretar para determinar el nombre de los canales a los que debe suscribirse y en los que debe publicar y el nombre de los mensajes, para que pueda importarlos del paquete donde se encuentran. Además, en dicho archivo se especifica el nombre de los *callbacks* asociados a cada canal, por lo que en la implementación del sistema, se deben respetar estos nombres.

Como el nodo de *ROS* debe cargar información de un archivo de configuración,

se definieron dos funciones que se encargan de esto: **setup_topics**, que se utiliza para suscribir el nodo a los canales de información y de actualización de los nodos; y **setup_-control_channel**, que se utiliza para definir el *publisher* al canal de control. Estas funciones se pueden apreciar en el apéndice 8.11.

Además de configurar los canales de información, al inicializar el nodo de visualización se debe definir la aplicación web, lo que implica que se deben establecer tanto sus *layouts* como sus *callbacks*. Finalmente, el último paso en la inicialización del nodo, es ejecutar la aplicación web. Para este paso, se debe tener el siguiente cuidado. Si la aplicación se corre en modo *debug* se pueden utilizar algunas herramientas del desarrollador como una gráfica de monitoreo de los *callbacks*. Sin embargo, al usarla en este modo, al iniciar el código este se correrá dos veces. Al correrse dos veces, el nodo se inicializa de nuevo, provocando que se cierre, porque no pueden haber dos nodos con el mismo nombre, y no se tenga comunicación con la *LTM*.

Si se desea utilizar el modo *debug* se debe especificar que la propiedad *use_reloader* sea falsa, de manera que el código no se ejecute dos veces.

Una vez inicializado el nodo de manera correcta, se procede a abrir un navegador web y especificar en la barra de búsqueda la dirección *http://127.0.0.1:8050/*, donde se aloja la aplicación web local desarrollada.

5.6.5 Estructura de las carpetas del paquete *mdb_view*

El sistema de visualización se entregará como un paquete de *ROS* que tiene la estructura descrita en la figura 5.18.

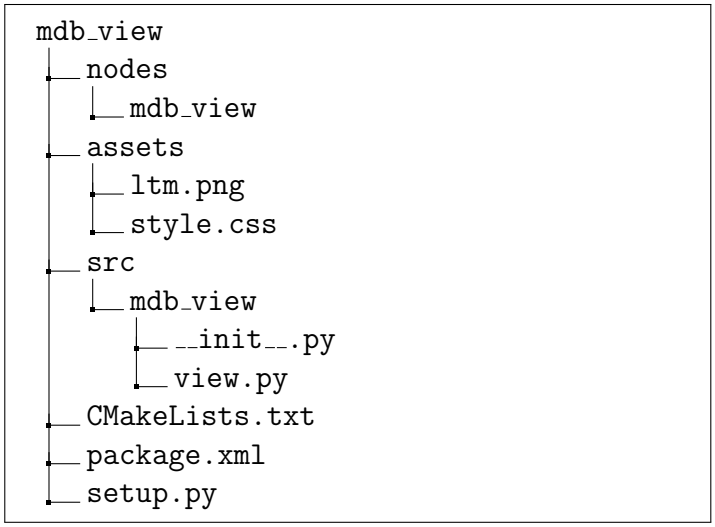

Figura 5.18: Estructura de las carpetas del paquete *mdb_view*

En la carpeta *assets* se tiene la plantilla de CSS y la imagen que se visualiza en la página de inicio. Por otro lado, en la carpeta *nodes* se tiene el código que inicializa el nodo de *ROS*, que importa el código fuente de la carpeta *src*, donde se tiene la clase donde se implementan tanto la aplicación web como sus *callbacks* y los de *ROS*.

5.7 Validación del sistema

Para la validación del sistema se integrará el sistema de visualización desarrollado, en un experimento con un robot Baxter como el que se muestra en la figura 5.19.

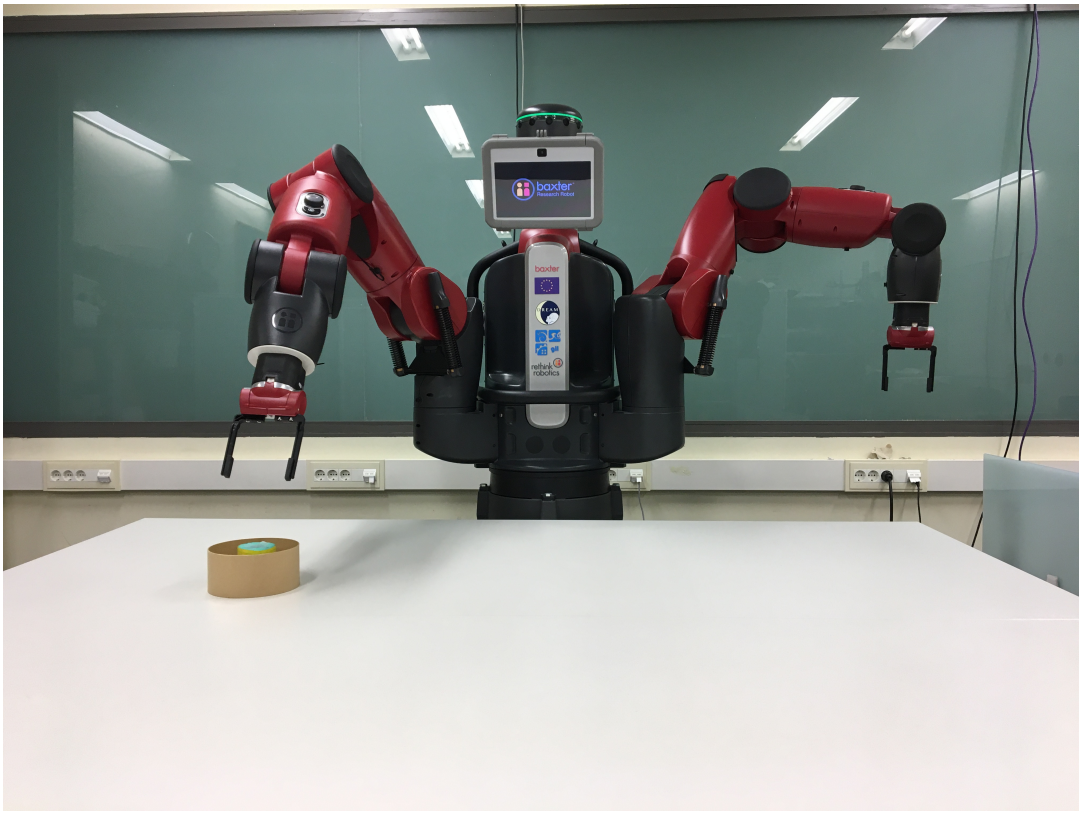

Figura 5.19: Robot Baxter para la validación del sistema de visualización. Fuente: GII

Se ejecutarán 1000 iteraciones del experimento detallado en la sección 5.5. En resumen, en dicho experimento el robot deberá aprender los contextos necesarios para ejecutar acciones, como levantar una bola o pedirla al experimentador, que le permitan cumplir satisfactoriamente un objetivo, como lo puede ser colocar una bola en una canasta. En este experimento se busca validar la comunicación por medio de *ROS*. Para ello se utilizarán los registros tanto de la *LTM* como del sistema de visualización, para determinar los instantes en donde se publica un dato desde la primera y cuando se termina de procesar el mensaje en el segundo. Además, se utilizará el canal de estadisticas de *ROS*, llamado */statistics*, para determinar si hay pérdida de mensajes durante la comunicación.

Lo que se busca validar con la medición de los tiempos, es que la actualización de la red y las gráficas de puntos y antipuntos en la visuzalización, refleje todos los cambios

que se dan dentro de la LTM.

CAPÍTULO 6

RESULTADOS Y ANÁLISIS

En este capítulo se presentan los resultados obtenidos del proyecto, detallados por algunas de las etapas definidas en el capítulo 5. Se iniciará por los resultados de la validación de la etapa de validación del código de visualización, seguidos por los resultados de la validación de la integración del sistema y finalizando con los resultados de las pruebas de validación realizadas con un robot simulado.

6.1 Resultados de la etapa de validación del código de visualización

En esta sección se procederá a detallar los resultados obtenidos del sistema de visualización previo a la integración con la *LTM*. Estos resultados se orientan a definir si la aplicación cumple con las necesidades del cliente, por lo que se evaluará que todos los *callbacks* definidos previamente funcionen del manera esperada.

Para la corrida del código, las percepciones, *policies*, metas y modelos de mundo iniciales son extraídos de un archivo tipo *yaml* con el que se cargan los nodos iniciales

de la *LTM*. Es importante mencionar, que para garantizar la reproducibilidad de los resultados expuestos en esta sección, se debe definir el *seed* del paquete *random* en 0 para que los números aleatorios generados sean los mismos en cada corrida del código.

6.1.1 Evaluación de la visualización de la *LTM*

El primer resultado que se mostrará es la evolución de la *LTM* en el tiempo. Para ello, se presentan dos figuras, la primera muestra la *LTM* para la iteración 100 y la segunda muestra la *LTM* en la iteración 1000.

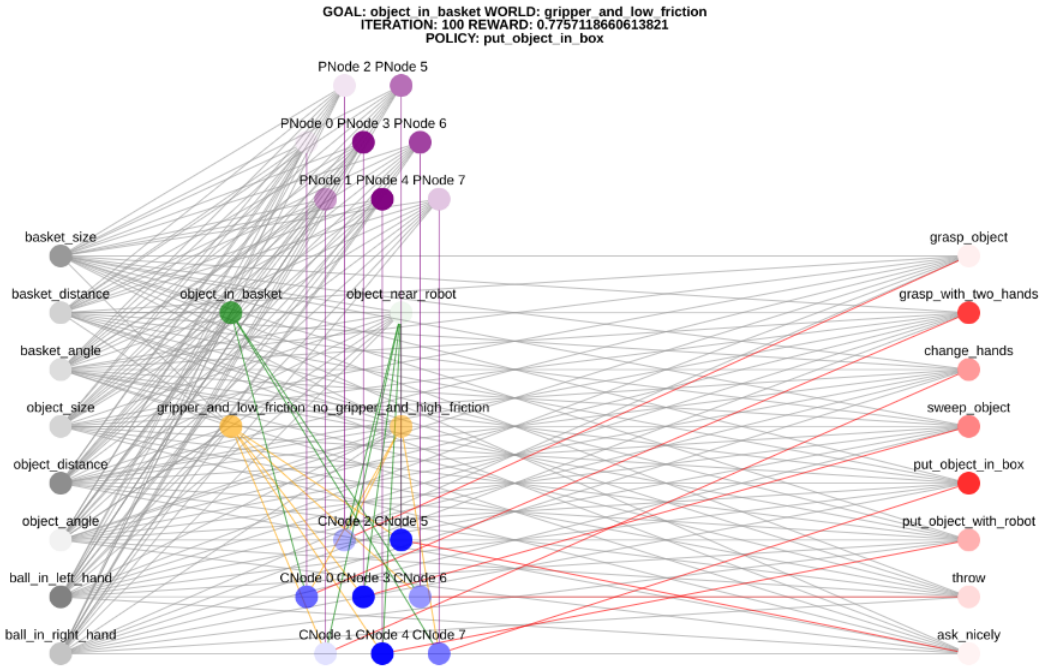

Figura 6.1: *LTM* simulada para la iteración 100. Elaboración propia

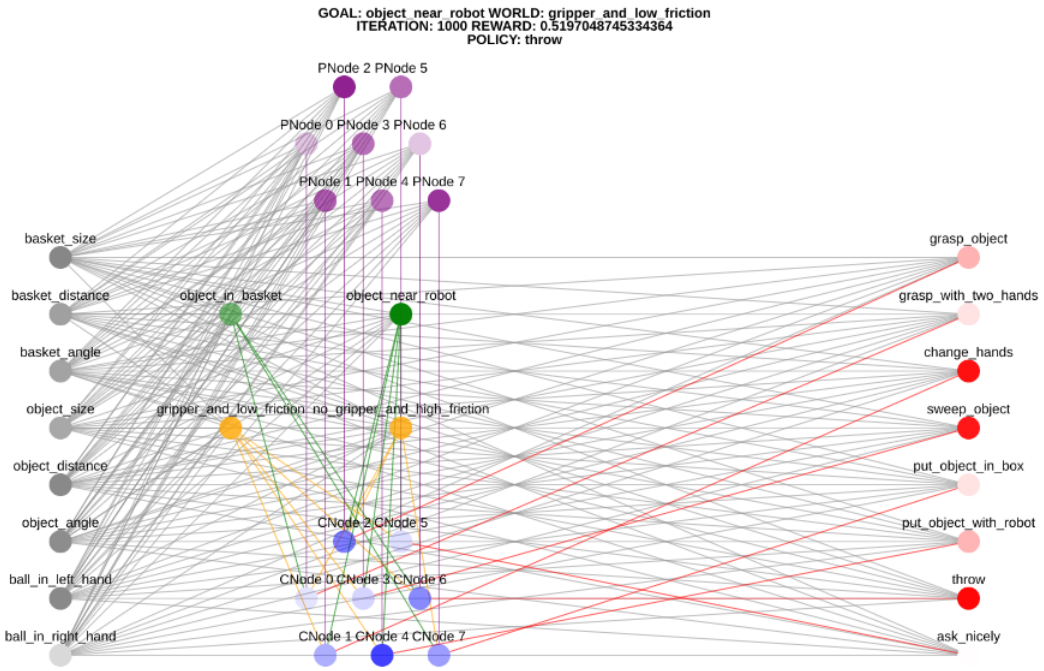

Figura 6.2: *LTM* simulada para la iteración 1000. Elaboración propia

La figura 6.1 muestra la estructura de la *LTM* para la iteraciòn 100, mientras que para la figura 6.2 se presenta la iteraciòn 1000. Con estas dos imágenes, obtenidas con el botón de guardado, se puede comprobar que esta funcionalidad trabaja de manera correcta.

Ahora bien, un investigador que utilice esta herramienta podrá deducir de la primera imagen que se han generado 8 *CNodes* y *PNodes*, con lo que el agente ha aprendido 8 contextos diferentes. Por ejemplo, el primer contexto que el robot aprendió fue el modelo de mundo *no_gripper_and_high_friction* , la meta *object_in_basket* y la *policy grasp_with_-two_hands*. Dicho contexto corresponde al número 14 detallado en el cuadro 5.18. Así, como se pudo establecer este contexto a partir de la gráfica, se pueden determinar los demás contextos aprendidos por el agente a partir de un estudio de la gráfica mostrada. Otra forma de obtener esta información, es mediante los cuadros de texto. Por ejemplo al hacer click en el primer *CNode*, se despliega la información de la figura 6.3

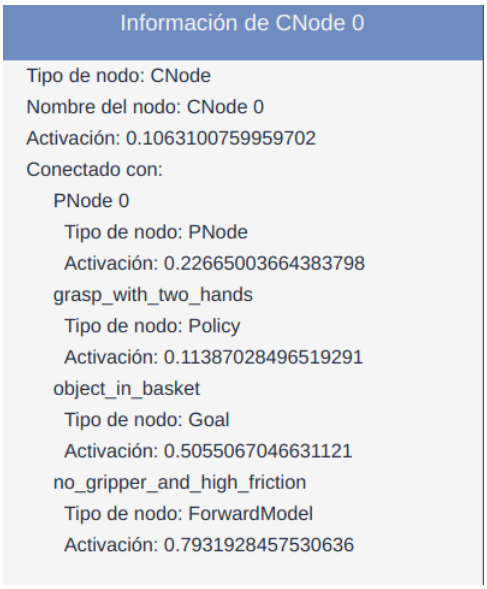

Figura 6.3: Información del *CNode 0* mostrada en el cuadro de texto. Elaboración propia

Con las figuras 6.1, 6.2 y 6.3 se puede observar el correcto funcionemiento del *callback* para desplegar información al seleccionar un nodo con el cursor, pues coinciden los vecinos a los que se conectan con los que se visualizan en la gráfica y el tipo de nodos, representado en el cuadro de texto como una palabra y en la figura como un color.

De la figura 6.2, se puede determinar que la última *policy* ejecutada por el agente fue la de lanzar el objeto, que derivó en una recompensa media para la meta que se tenía, que era acercar el objeto al robot. Onservese que la recompensa tiene un valor bajo, pues el objetivo actual del robot es colocar el objeto cerca de él, y al ejecutar la *policy* de tirar el objeto se logra todo lo contrario.

Caso opuesto se da en la figura 6.1, donde la última *policy* ejecutada fue poner el objeto en la canasta y esta generó una recompensa moderadamente alta pues la meta que se tenía era de poner el objeto en la canasta.

Por las conclusiones que el investigador puede obtener de las imágenes anteriores, se puede comprobar la importancia y utilidad que tiene el sistema de visualización desarrollado. Además de que desacopla esta tarea de la LTM, lo cual representa una ventaja pues permite que la LTM se utilice únicamente para establecer conexiones entre

los nodos de conocimiento que le permitan maximizar la recompensa ante la ejecución de una acción; y no para visualizar su estructura.

Otro tipo de resultados importantes de analizar en esta sección es el de la actualización entre iteraciones consecutivas. Para ello se utilizará la figura 6.4 que representa la memoria en la iteración 7 y la figura 6.5 que representa la iteración 8.

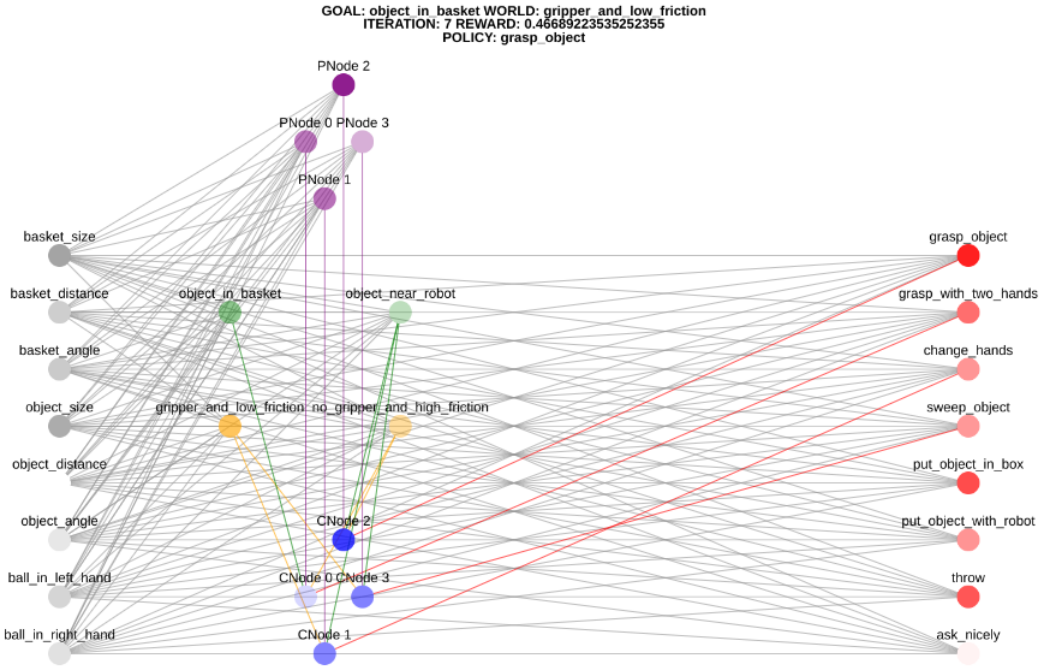

Figura 6.4: *LTM* simulada para la iteración 7. Elaboración propia

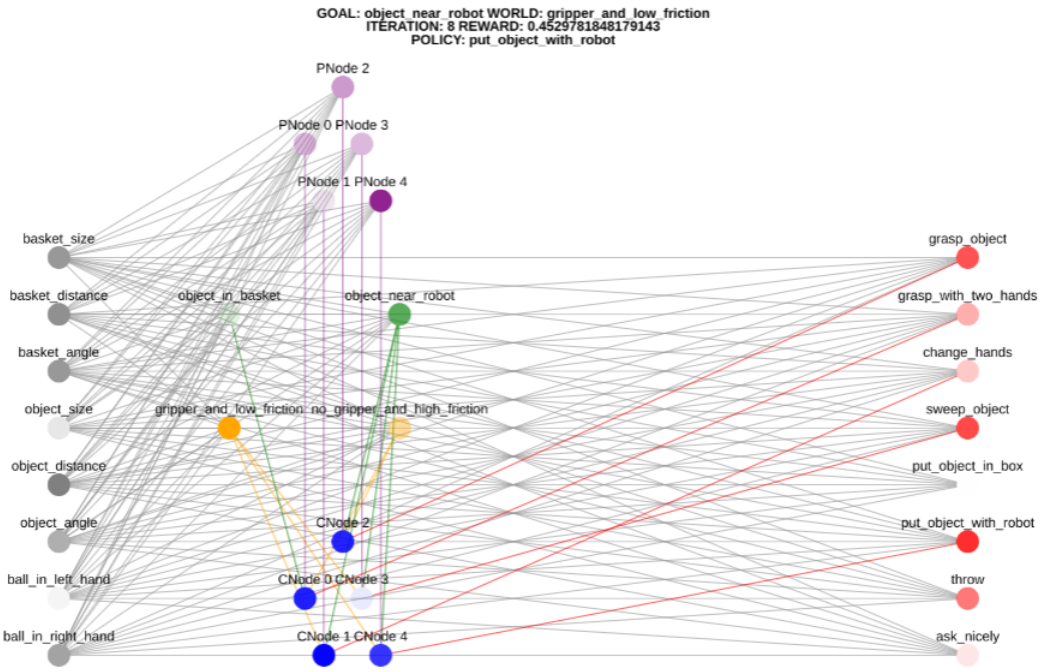

Figura 6.5: *LTM* simulada para la iteración 8. Elaboración propia

Las figuras anteriores muestran la red en la iteración 7 y 8. En la primera se observan sólo 4 *CNodes*, mientras que en la segunda se observa la adición de un nuevo *CNode* a la red. Observando la figure 6.4 se puede notar que la *policy* *put_object_with_robot* no está conectada a ningún nodo. Justo esa misma *policy* se vuelve la más activa en la siguiente iteración, provocando así su ejecución y tras recibir la recompensa de aproximadamente 0.4529, se crea un nuevo *CNode*. Además, se puede observar cómo se actualizan las activaciones de los nodos por el cambio en su opacidad. Por ejemplo, se observa el *PNode 2* de la iteración 7 con una activación que provoca que el nodo se vea bastante opaco, mientras que en la 8 se observa un poco transparente.

6.1.2 Evaluación de la visualización de los *PNodes*

Para esta etapa, se debe recordar que el cliente brindó -para la parte de visualización de los puntos y antipuntos- un conjunto de datos que son resultado de le ejecución del experimento con un agente real operando con la *LTM*. Puesto que se trabajaban con

resultados conocidos, el cliente indicó para cada uno de ellos cuáles percepciones eran relevantes de graficar. Con base en esto, se definieron 4 nodos a los que se les daría seguimiento: el *PNode 1* y el *PNode 7*, con los que se graficaría la distancia y el ángulo del objeto; y del *PNode 3* y *PNode 6*, para los que se graficaría la distancia y el ángulo de la canasta.

En la figura 6.6 se presentan los puntos y antipuntos de los *PNodes* 1 y 7. Se logra observar que con el avance del experimento, los puntos y antipuntos definen una frontera. Con la iteración 100, dicha frontera aún no es clara, pero para las iteraciones 1000 y 10005 se va definiendo aún mejor.

Figura 6.6: Puntos y antipuntos para el PNodes 1 y el PNode 7 para diversas iteraciones de la simulación de la *LTM*

Ahora bien, si se define la activación de un nodo por la cercanía de un punto nuevo con los puntos y antipuntos existentes, se podría decir que entre más lejos se encuentre el objeto, mejor la recompensa por ejecutar el *policy* al que está asociado. En el cuadro 5.18, se observa que para los dos nodos que se están estudiando, la *policy* asociada es la de pedir al experimentador que acerque el objeto a la base. Una vez dicho esto,

la conclusión sobre la distancia del objeto y la activación del nodo adquiere sentido, porque para objetos colocados cerca de la base del robot, solicitar que se le acerque el objeto no tiene mucho sentido.

La figura 6.7 muestra la evolución de los puntos y antipuntos de los *PNodes* 3 y 6.

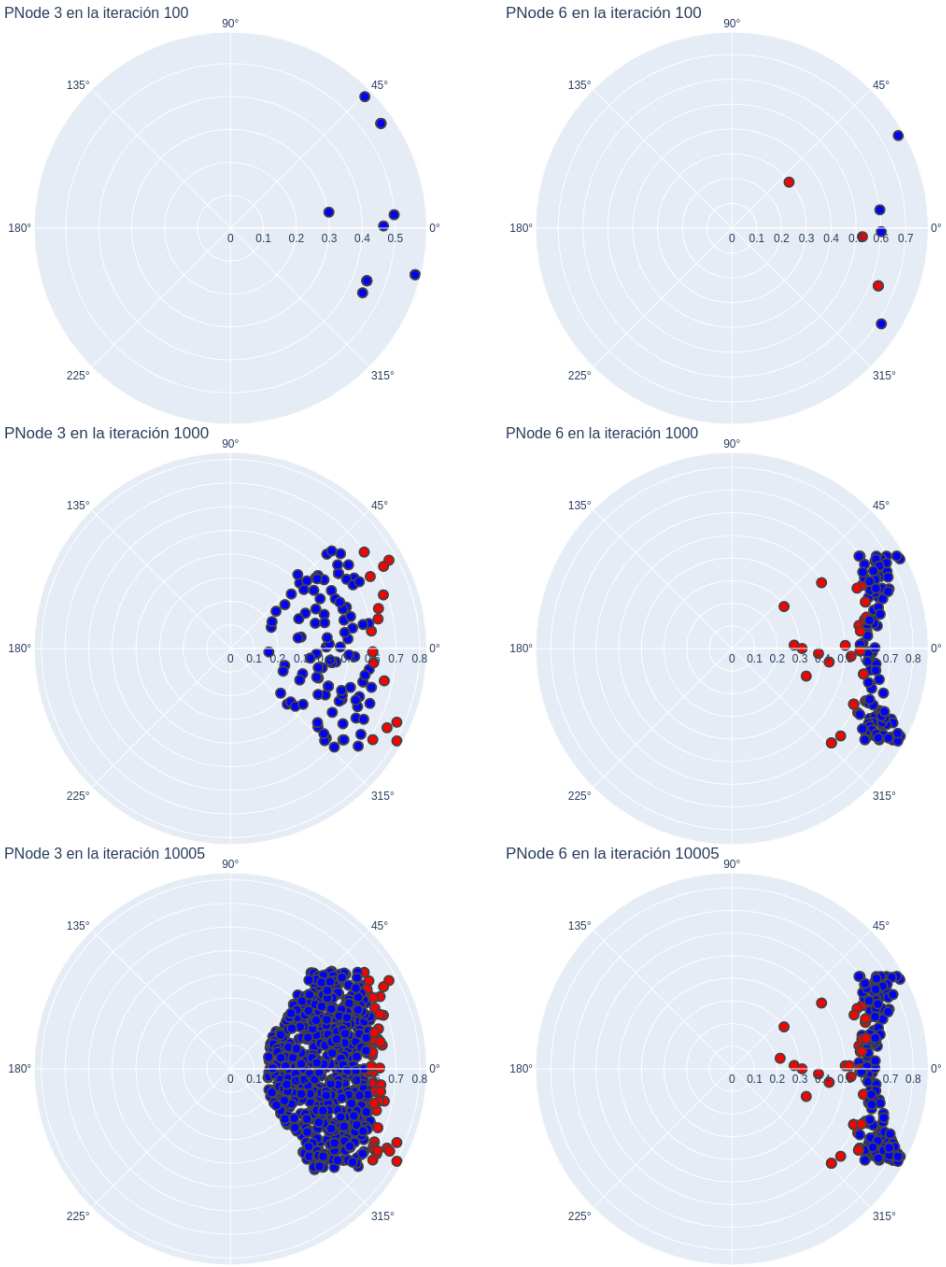

Figura 6.7: Puntos y antipuntos para el PNodes 3 y el PNode 6 para diversas iteraciones de la simulación de la *LTM*

Para las gráficas mostradas en la figura anterior, se detalla la distancia y el ángulo

de la canasta. Volviendo a revisar el cuadro *contexts*, se puede observar que el *PNode 3* se asocia a la *policy* de colocar el objeto en la canasta para el objetivo o meta de colocar el objeto en la canasta. Por esta razón es que se observa tan poblada la región cercana a la base del robot, porque entre más cerca la canasta de la base del robot, más sencillo será para el robot colocar el objeto dentro. Por otro lado, el *PNode 6*, se asocia con la *policy* de tirar el objeto para la misma meta que el nodo anterior. De esta manera, se le puede dar sentido a las regiones que se van formando. Entre más lejos se encuentre la canasta, simplemente agarrar el objeto y colocarlo va a ser imposible por estar fuera del alcance de los brazos del robot. Por esto, es que entre más lejos se encuentre la canasta, la *policy* con la que se logra introducir el objeto es tirándolo.

Sobre el análisis de los contextos de los *PNodes* estudiados en esta sección y los resultados esperados que se reportan en el cuadro 5.18, es importante observar que la visualización de la *LTM* no cumple con los datos de dicho cuadro, a diferencia de los puntos de los *PNodes*. Esto se debe a que el primero se basa en un funcionamiento que replica el pseudocódigo de la *LTM*, pero con mecanismos de activación aleatorios y el segundo se basa en datos de experimentos reales. Se aclara así, que la parte de visualización de la *LTM* y la de los nodos funcionan con datos desacoplados. Esto no significa que el sistema funcione incorrectamente, pues su objetivo no es replicar de manera perfecta la *LTM*, si no definir una estructura del programa que permita observar la información necesaria.

Con el análisis hecho en esta sección sobre el significado de las regiones que se pueden observar con los puntos y antipuntos en base al contexto en el que están definidos; se puede observar la utilidad que la herramienta de visualización puede aportar a los investigadores. Al tratarse de una aplicación que se actualiza de manera constante, el investigador será capaz de formular y comprobar hipótesis de manera más rápida pues puede observar en la ejecución del experimento cómo se comportan los nodos.

6.1.3 Evaluación de los *callbacks*

Como resultado adicional, se presentan las relaciones entre los elementos de la aplicación web. Además, gracias al modo *debug* de las aplicaciones de Dash, se puede extraer información de los *callbacks*, como el tiempo de ejecución aproximado. Para comprender los siguientes diagramas se debe definir los siguientes aspectos: los cuadros en gris representan un elemento del *layout*, el nombre ubicado en su parte superior representa su *id*, el cuadro en azul dentro del cuadro gris representa una propiedad del elemento. Si la flecha sale del cuadro azul, significa que es la entrada del *callback*, si por el contrario la flecha entra al cuadro, la propiedad es la salida del *callback*. En el cuadro verde, entre entrada y salida se muestra en la parte superior el número de veces que el *callback* ha sido activado y en la parte inferior el tiempo medio que tarda la ejecución de dicho *callback*.

De este modo, en la figura 6.8 se observa el *callback* para el cambio de *layout*. Se debe recordar que este se activa cuando el *pathname* del elemento llamado *url* cambia por haber seleccionado una imagen en la página de inicio, o por haber seleccionado el link *LTM* o *Nodes*.

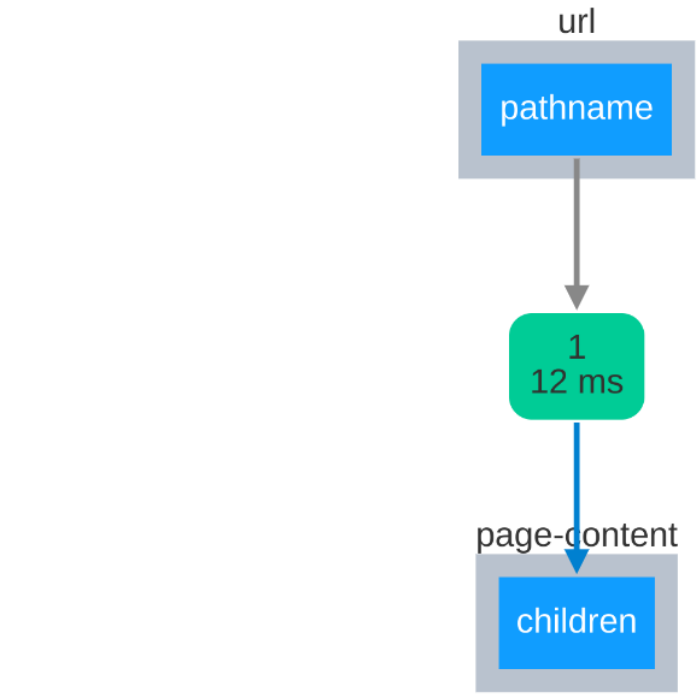

Figura 6.8: Diagrama del callback de cambio de *layout* y tiempo medio de ejecución. Elaboración propia

El diagrama de la figura 6.9 se representan las relaciones entre elementos de la página de la visualización de la *LTM*.

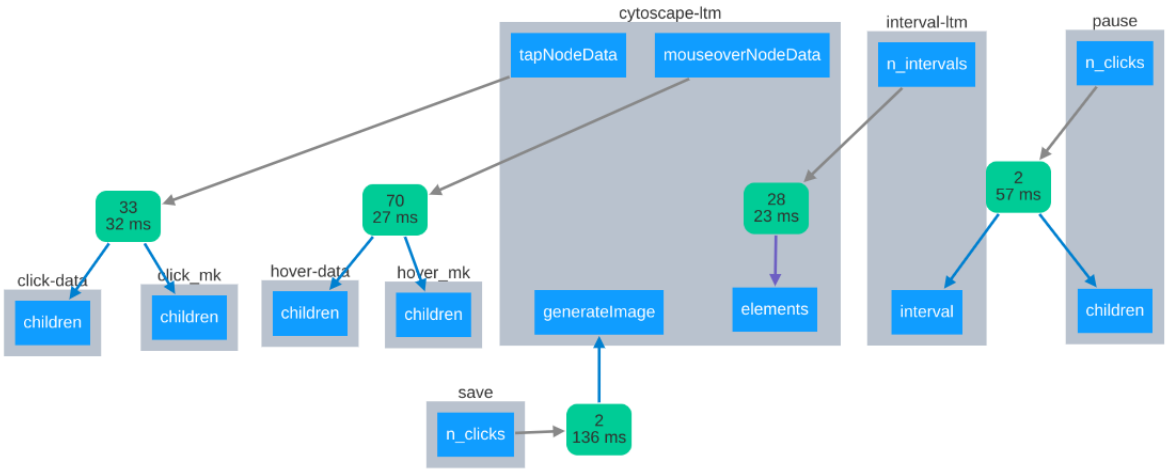

Figura 6.9: Diagrama de *callbacks* de la visualización de la *LTM*. Elaboración propia

Por último se presenta en la figura 6.10 el diagrama de *callbacks* de la página de visualización de los *PNodes*.

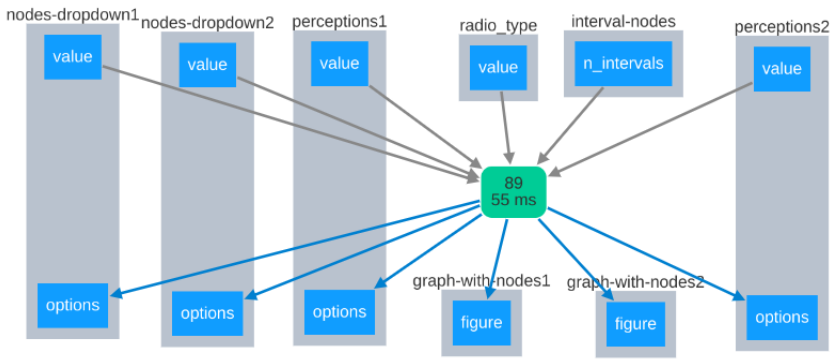

Figura 6.10: Diagrama de *callbacks* de la visualización de los *PNodes*. Elaboración propia

Con las figuras anteriores, se puede concluir que las relaciones establecidas en el código entre los elementos de la aplicación, se cumplen de manera adecuada. Además, los tiempos medios de ejecución de los **callbacks** no exceden el período de actualización de la red, con lo que se garantiza un mayor período de tiempo para el almacenamiento y procesamiento de los datos que provienen de la *LTM*.

6.2 Resultados de la etapa de integración del sistema

Para esta etapa los resultados que se obtuvieron provienen de pruebas de unidad de la integración entre la *LTM* y el código de visualización. Para esta etapa, se corrieron los nodos de visualización y de visualización de forma simultánea.

La *LTM* por si sola no es capaz de generar percepciones, por lo que su ejecución se limita a cargar los nodos iniciales. Por esta razón, el resultado esperado es que en el nodo de visualización se logre visualizar por lo menos los nodos que la *LTM* carga inicialmente. Si esta parte funciona, significa que la publicación de la información de los nodos desde la *LTM* funciona de manera correcta, al igual que la lectura del mensaje en el nodo de visualización. Sin embargo, no se podrá probar la publicación del resto de información desde la *LTM*. Para ello, se debe realizar la validación final con un agente

que genere percepciones que la *LTM* pueda leer e interpretar.

A pesar de esta limitante, de igual forma se puede determinar si el nodo responde adecuadamente a los mensajes, publicándolos desde la terminal y observando si hay un cambio en la red o en los puntos y antipuntos. Con esto, además, se genera un conjunto de pruebas que permitirán a futuros usuarios si la funcionaliad del nodo de visualización se ha visto perjudicada.

Tras probar la ejecución simultánea de ambos nodos se tiene que en el sistema de visualización se observa la estructura detallada en la figura 6.11.

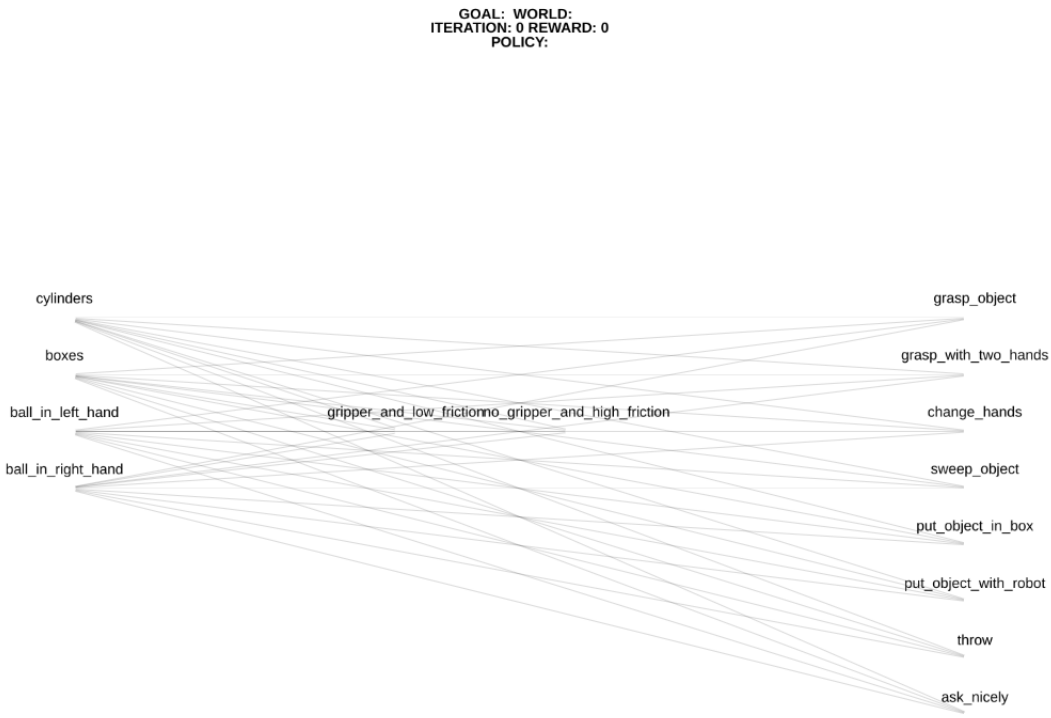

Figura 6.11: Red inicial de la *LTM* vista desde el nodo de visualización. Elaboración propia

Revisando el archivo de configuración utlizado para cargar los nodos iniciales de la *LTM* se concluye que los vistos en la figura anterior coinciden con los de dicho archivo. Por esta razón se determina que la *LTM* sí reacciona a la solicitud del sistema de visualización para que envíe la información de todos sus nodos. Por otro lado, se concluye que el sistema de visualización responde de manera adecuada a la publicación

de nuevos nodos.

Para probar la respuesta del nodo de visualización ante el resto de mensajes que de la *LTM* puede publicar, se definió la lista de comandos que deben ejecutarse en orden desde la terminal y que se detallan en el cuadro de texto 6.1.

```
Actualizar la activacion de una policy

rostopic pub /mdb/stm/policy mdb_common/NodeMsg '{command : 'update',
    id : "grasp_object", neighbor_ids : [], neighbor_types : [],
    activation : 0.7, execute_service : '', get_service: '', class_name
     : '', language: 'Python'}'

Enviar un nuevo CNode

rostopic pub /mdb/ltm/c_node mdb_common/NodeMsg '{command : 'new', id
    : "CNode1", neighbor_ids : [], neighbor_types : [], activation :
    0.7, execute_service : '', get_service: '', class_name : '',
    language: 'Python'}'

Enviar un nuevo PNode

rostopic pub /mdb/ltm/p_node mdb_common/PNodeMsg '{command : 'new', id
     : "PNode1", names : ['ball_distance', 'ball_angle'], neighbor_ids
    : ['CNode1'], neighbor_types : [CNode], activation : 0.7,
    execute_service : '', get_service: '', class_name : '', language: '
    Python'}'

Enviar variables de informacion

rostopic pub /mdb/ltm/info mdb_common/InfoMsg '{iteration : 32,
    current_world: 'no_gripper_high_friction', current_policy: '
    grasp_object', current_reward: 0.3256, current_goal: '
    object_in_basket'}'

Publicacion de punto en PNode 1

rostopic pub /mdb/ltm/p_node_update mdb_common/PointMsg '{command : '
    new', id : "PNode1", point : [0.2464, 0.014], confidence: 1}'
rostopic pub /mdb/ltm/p_node_update mdb_common/PointMsg '{command : '
    new', id : "PNode1", point : [0.2464, -1.114], confidence: 0}'
```

Listing 6.1: Comandos para validar el funcionamiento correcto de los *callbacks* de *ROS* del nodo de visualización

Tras ejecutar los comandos descritos, se obtiene como resultado la siguiente figura para la red de la *LTM*. Se observa que la activación de la *policy grasp_object* se actualizó pues ahora se logra ver en color rojo. Además, se añadió un nuevo nodo de tipo *CNode*, uno de tipo *PNode* y una arista que los conecta a ambos. Además, al *PNode 1*

se le definieron el nombre de las percepciones (*ball_distance y ball_angle*). Finalmente, se observa un cambio en el título, lo que significa que si responde bien a los mensajes publicados en el canal de información. Estos cambios se ven reflejados en la figura 6.12

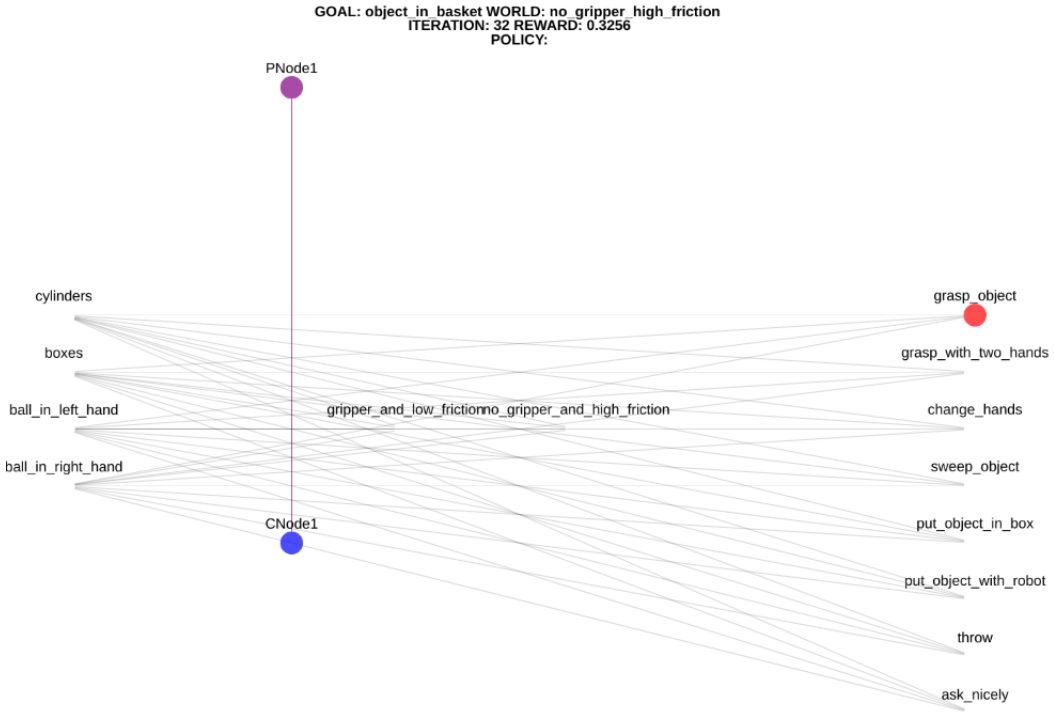

Figura 6.12: Red vista desde el nodo de visualización tra publicar comandos en la terminal. Elaboración propia

Además, se añadieron dos percepciones al *PNode*, la primera correspondiente a un punto y la segunda a un antipunto. Se observa el resultado en la figura 6.13.

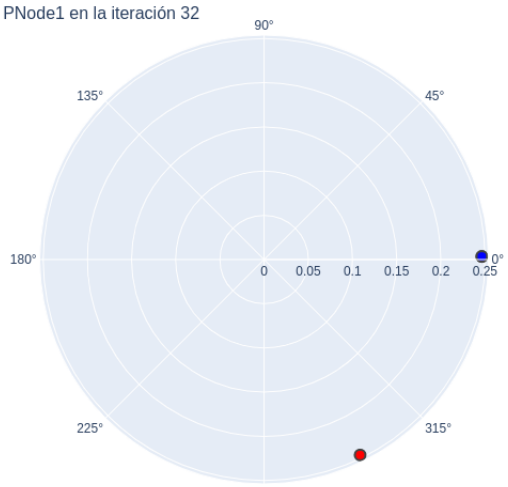

Figura 6.13: Punto y antipunto tras publicarlos desde la terminal. Elaboración propia

Finalmente, con la herramienta *rqt_graph*, se puede visualizar el nodo de visualización y los canales en los que publica y a los que está suscrito, resultando en el diagrama de *topics* descrito en la figura 6.14.

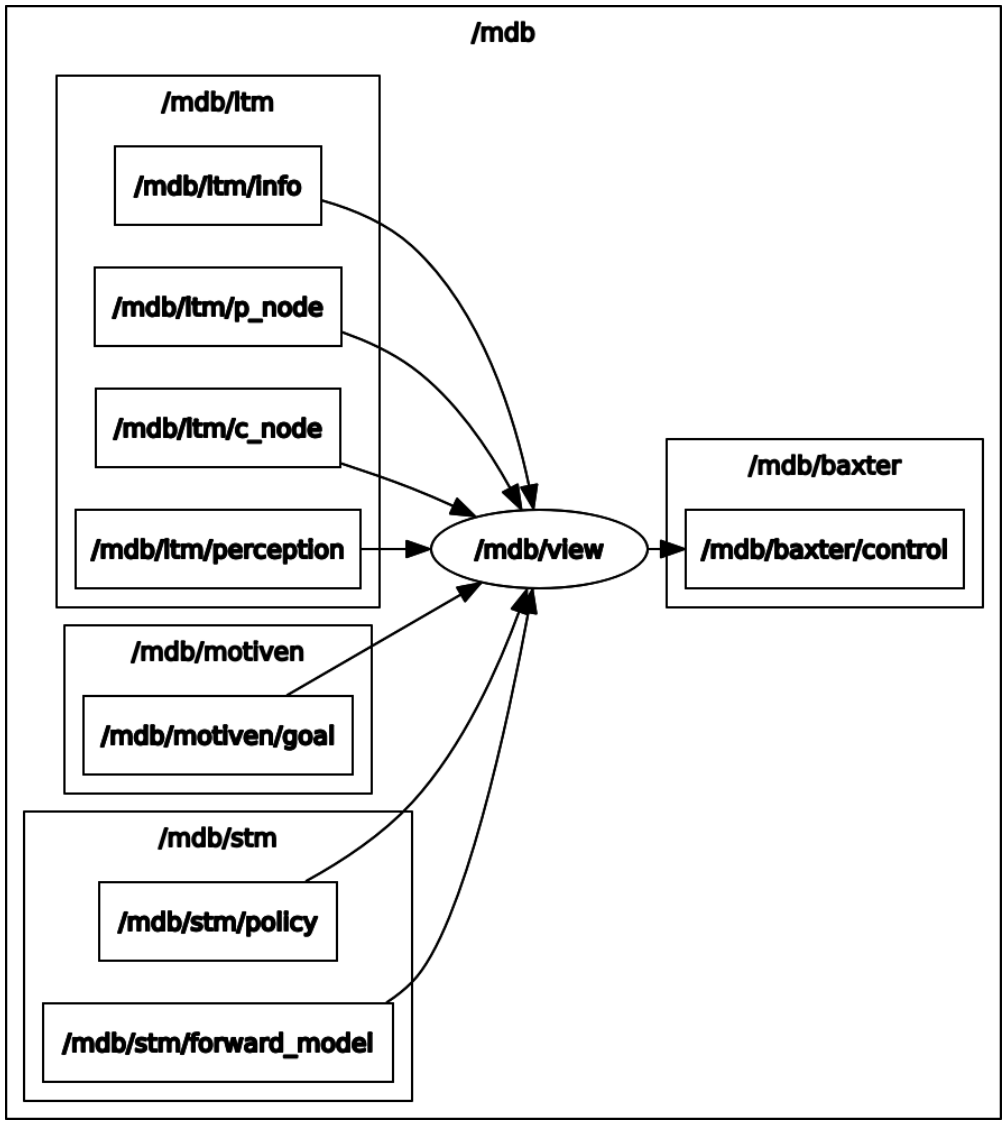

Figura 6.14: Canales conectados al nodo de visualización. Elaboración propia

6.3 Resultados de la validación del sistema

Se debe recordar que esta etapa consiste en garantizar que los datos publicados por la *LTM*, son recibidos por el sistema de visualización, y que el tiempo en el que se actualizan estos datos no exceda el período de actualización de las gráficas. Para ello se medirá el tiempo que tarda un mensaje en ser enviado y procesado y se compararán el número de mensajes enviados con el número de mensajes procesados.

Como se mencionó en el capítulo anterior, en esta etapa se implementó la *LTM* en un robot Baxter. Para medir el tiempo que transcurre desde que se envía un mensaje, hasta que es procesado en el sistema de visualización, se hace un análisis de los registros que cada uno almacena en el ordenador. Dichos archivos registran un mensaje, el tipo de mensaje y el tiempo en el que se ha publicado en el canal *rosout*, con algún comando como *rospy.loginfo*. De este modo, previo a ejecutar un experimento, se pueden definir mensajes con indicadores específicos, que se envían antes de publicar en un canal en la *LTM* y después de procesar en el sistema de visualización. Y después de finalizado el experimento, se hace un análisis con *Python* de los registros, para buscar mensajes con indicadores que coinciden en ambas partes, y se comparan los tiempos de publicación. En la figura 6.15 se muestra la utilización del sistema de visualización con el robot Baxter.

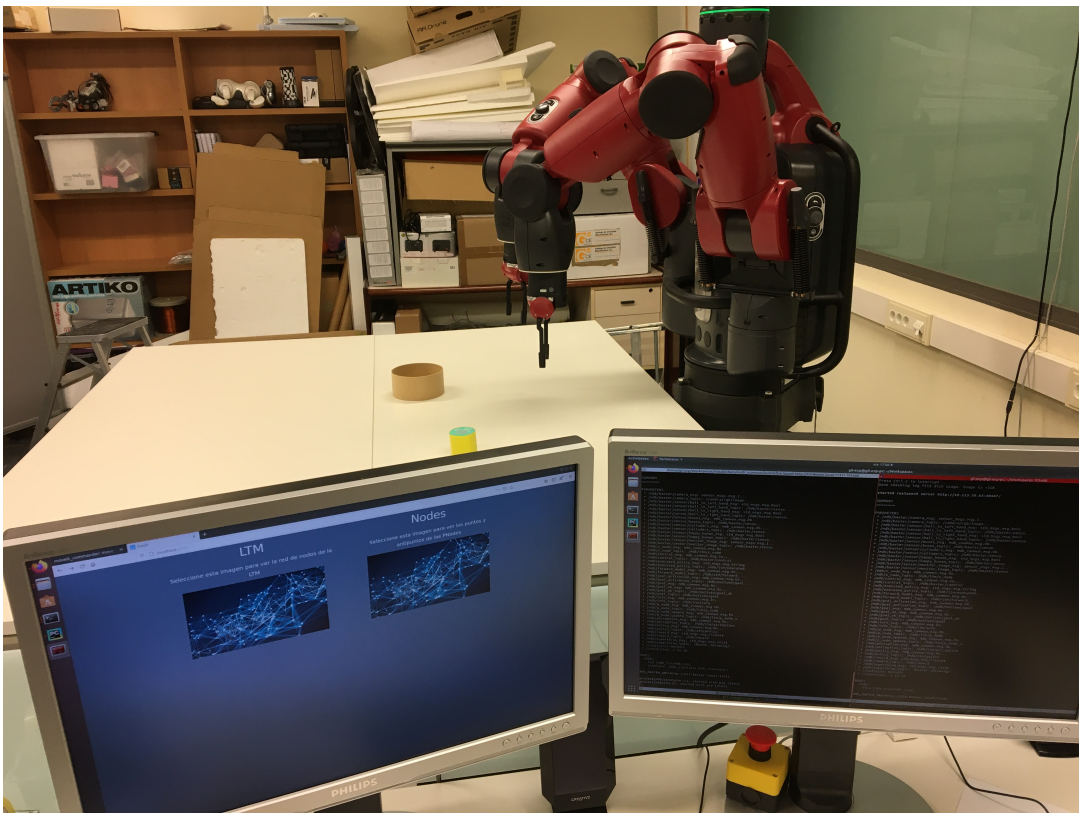

Figura 6.15: Implementación del sistema de visualización con el robot Baxter. Fuente: G11

Además, se puede medir el tiempo por iteración para cada tipo de mensaje: creación de un nodo, actualización de la activación, actualización de la información del experimento y publicación de un punto o antipunto, agrupándolos por iteración.

De este modo, se logra obtener la gráfica de la figura 6.16 donde se observa el tiempo de envío y de procesado total, para 1000 iteraciones del experimento.

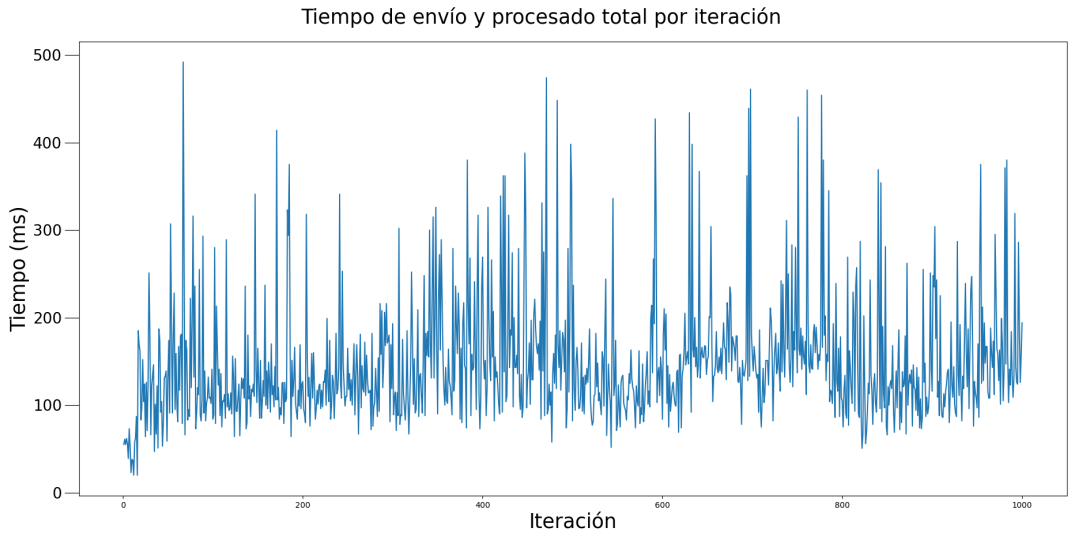

Figura 6.16: Tiempo de envío y procesado por iteración. Elaboración propia

Se logra observar de la figura anterior, que el tiempo de envío y procesado de los mensajes se mantiene, para todas las iteraciones, por debajo del tiempo de actualización de la gráfica que es de 2 s. Se puede observar cómo inicialmente el tiempo inicia en un valor relativamente pequeño. Esto se debe a que los únicos nodos en la red son los iniciales, pues no se han creado nuevas metas ni *PNodes* o *CNodes*. En estas iteraciones iniciales, los únicos mensajes que se reciben son los de la actualización de la activación y de la información del experimento. Además, se puede comparar estos tiempos con el número de mensajes por iteración.

Durante el experimento, de 1000 iteraciones, se enviaron desde la *LTM* 59734 mensajes. La composición de estos mensajes y el tiempo medio de procesado de cada uno de ellos se detalla en el cuadro 6.1.

Cuadro 6.1: Distribución de los mensajes en el experimento y tiempo medio de procesado

Tipo de mensaje	**Número de ocurrencias**	**Tiempo medio de envío y procesado (ms)**
Creación de nuevo nodo	44	2.692
Actualización nodo	57457	2.42
Actualización de la información	1000	3.161
Nuevo punto o antipunto	1233	12.839

Se procede a realizar un anślisis de los tiempos específicos de cada tipo mensaje. Se inicia analizando la creación de nuevos nodos. Para ello, se hará uso de la figura 6.17, donde se observa la relación entre el número de mensajes (en azul) y el tiempo por iteración (en rojo).

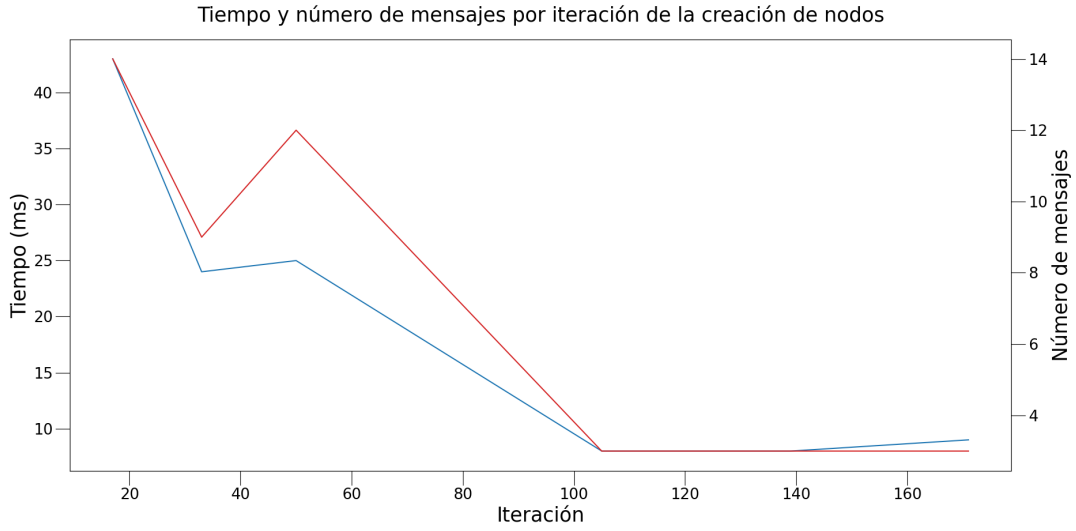

Figura 6.17: Tiempo de envío y procesado por iteración de la creación de nuevos nodos

Un análisis de los datos de la figura anterior, permite deducir que el primer nodo, o grupo de nodos creados en la *LTM*, se da en la iteración 17, mientras que el último se da en la 171. Además, se logra observar una relación casi lineal entre el número de mensajes y el tiempo de actualización. El tiempo medio de actualización por cada mensaje es de 2.692 ms con una desviación estándar de 0.35 ms.

El segundo tipo de mensajes que se puede analizar, es el de la actualización de la

información. Para este se analiza únicamente el tiempo y no el número de mensajes, puesto que este es constante, ya que la actualización de la información se da solo una vez por iteración. De este modo solo se analizará el tiempo de envío y procesado en la figura 6.18.

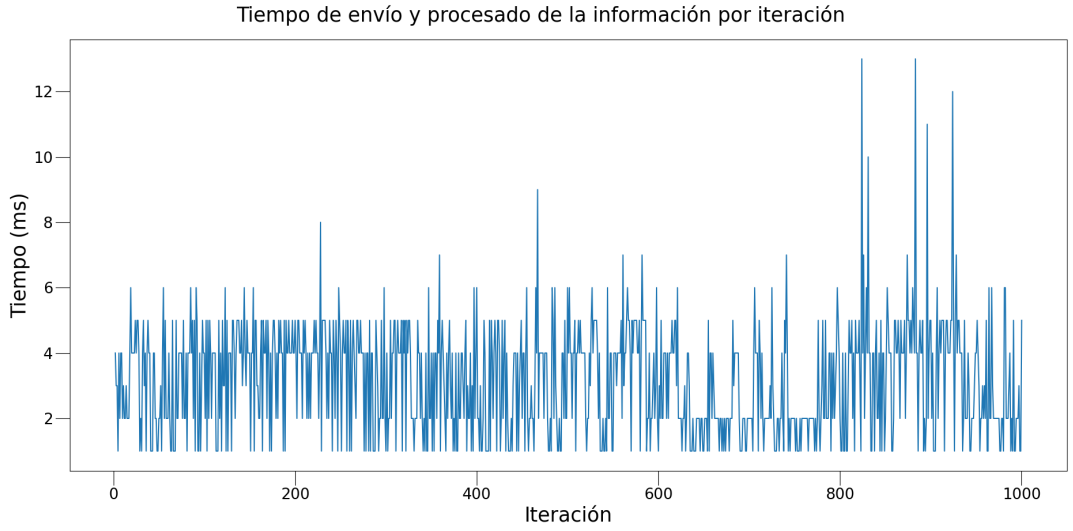

Figura 6.18: Tiempo de envío y procesado por iteración de la actualización de la información del experimento. Elaboración propia

Se observa en la figura anterior, que la actualización de la información del experimento es un proceso poco demandante, y que se mantiene en la mayoría de las iteraciones dentro de límites claros. El mensaje es procesado rápidamente en el sistema de visualización, pues, a diferencia de los demás, los datos no deben ser modificados ni utilizados para buscar alguna variable almacenada. El proceso únicamante asigna a variables globales del sistema, los valores del mensaje.

El proceso de actualización de las activaciones es el que más se da durante el experimento, tal y como se muestra en el cuadro 6.1. Los resultados obtenidos para este proceso se muestran en la figura 6.19. En comparación con la creación de nodos nuevos, la activación de todos los nodos de la red se da en cada iteración del experimento. La integridad de los datos es fundamental en este proceso, pues se debe reflejar el cambio en la opacidad de los nodos en el sistema de visualización. Un monitoreo con un *script*

de *Python* del canal de estadísticas, permitió concluir que en los canales dedicados a la publicación de nodos no hubo pérdida de mensajes en todo el experimento.

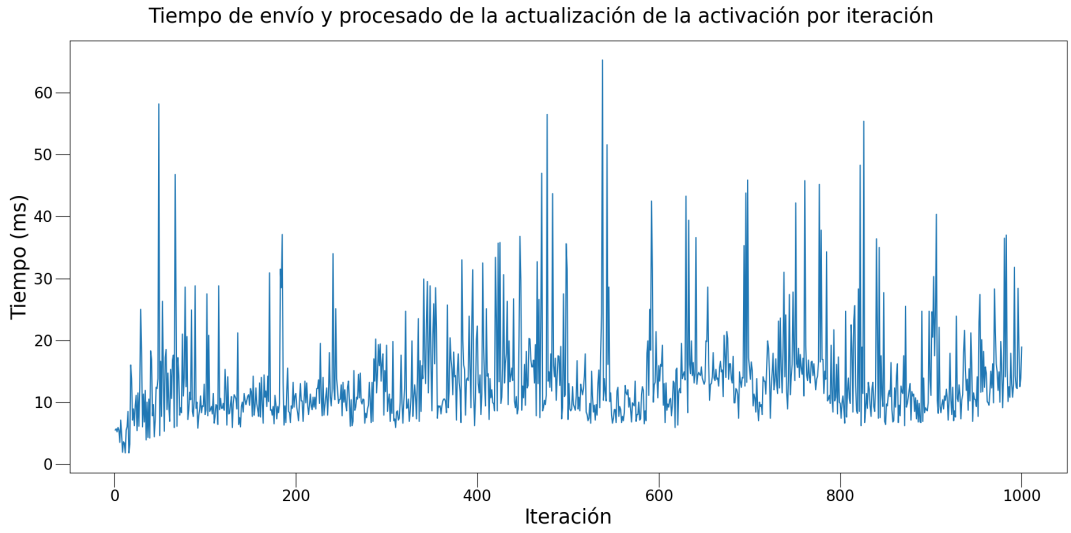

Figura 6.19: Tiempo de envío y procesado por iteración de la activación de los nodos. Elaboración propia

Finalmente, se observa el tiempo medido para la actualización de puntos y anti-puntos, que según el cuadro 6.1 es el proceso más demandante en cuanto a tiempo de actualización, pues requiere mayor procesado que el resto de mensajes. Este resultado se observa en la figura 6.20.

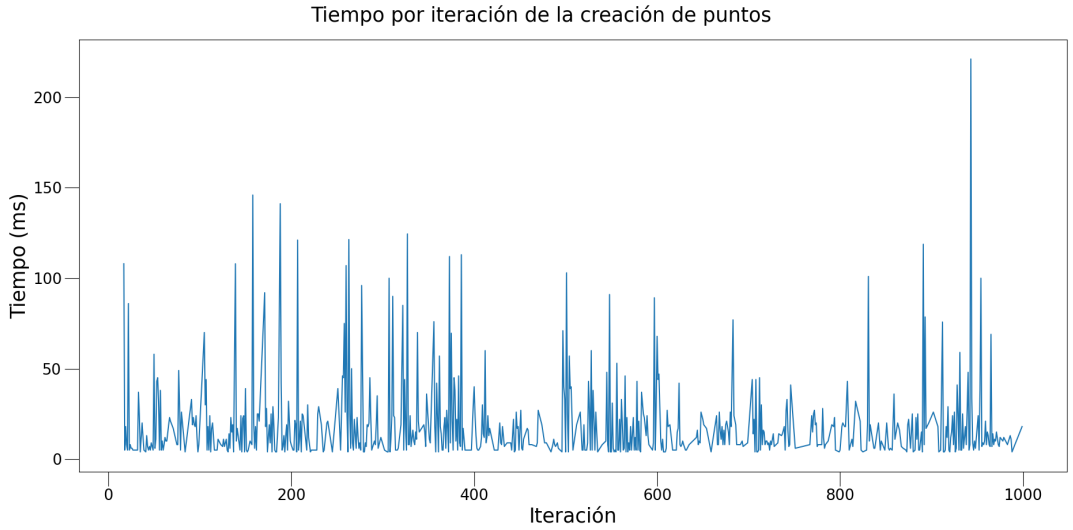

Figura 6.20: Tiempo de envío y procesado por iteración de la generación de puntos nuevos. Elaboración propia

De este modo, dado que los tiempos de actualización de los datos no exceden el período de actualización de la red en el sistema de visualización, el sistema se encuentra validado.

6.4 Análisis económico

El análisis económico del proyecto iniciará haciendo un estudio del mercado. Esta etapa se basa en la definición de los requerimientos planteados en la sección 5.1. En primer lugar, se debe realizar una descripción del producto desarrollado. Este es un sistema de visualización interactivo que forma parte de una aplicación robotica para la memoria a largo plazo de la arquitectura cognitiva *MDB*. El producto desarrollado es único, pues es para una aplicación específica en el dominio de la robótica cognitiva. En cuanto a la demanda, sólo se tiene un cliente, que corresponde a los investigadores del Grupo Integrado de Ingeniería. Con respecto a la oferta, puesto que es un producto para una aplicación muy específica, no hay ningún producto en el mercado que cumpla las necesidades del cliente.

A continuación, se realizó un estudio técnico, donde se estudió el comportamiento de la *LTM*, la tecnología que utiliza, las bases teóricas sobre las que se fundamenta y cómo se integra posteriormente con un agente, como lo puede ser un robot. Los resultados de esta parte se relacionan con los aspectos detallados en el marco teórico.

El siguiente aspecto por definir es los recursos del proyecto. Dentro de la categoría del recurso material se tiene la computadora de trabajo utilizada. Sobre el activo fijo que es la computadora se puede realizar el cálculo de una depreciación lineal como la que se muestra a continuación.

Cuadro 6.2: Valores para el cálculo de depreciación de la computadora utilizada

Valor de compra	**Valor residual**	**Tiempo aprximado de vida útil**
550000	300000	5 años

Con los datos del cuadro anterior, y siguiendo una depreciación lineal, el valor de la computadore personal utilizada disminuye en 50000 colones por año. Traduciendo esto a semanas, la computadora se deprecia a 962 colones por semana. Para el tiempo que transcurre el proyecto, el activo fijo se habrá depreciado en 15392 colones.

El segundo recurso por considerar, es el tiempo. Para completar el proyecto, se tienen 16 semanas, donde se trabajó 10 horas en cada una. De esas 10 horas, se dedicaron 2 horas para documentar el proyecto, 1 hora para comunicarse con el cliente y 7 horas para desarrollar el producto. Con esto, de las 160 horas dedicadas al proyecto, 16 horas fueron dedicadas a consultas con el cliente, 32 a la documentación y 112 a la construcción del producto. Además se tiene el recurso humano. Para el desarrollo del proyecto, se contó con la asesoría de un investigador del GII y del profesor Juan Luis Crespo Mariño. Este recurso se usará principalmente para consultar aspectos técnicos del proyecto.

Para finalizar el análisis económico, se analizan los resultados obtenidos, la rentabilidad de la solución y los beneficios que le dará al cliente. El resultado del proyecto es una herramienta de experimentación para la robótica cognitiva. Se determinó que con esta, la visualización se actualiza periódicamente, y en ella se observa la estructura de

la *LTM* y de los puntos y antipuntos de los *PNodes*. Además, tras la incorporación de la herramienta, el trabajo posterior al experimento se reduce, pues gráficas como las de los *PNodes* se logran extraer directamente de la aplicación, sin necesidad de realizar un procesado posterior de algún archivo. Además, la capacidad de pausar el experimento desde la aplicación, permitirá al investigador realizar un análisis en detalle del experimento, extraer conclusiones y determinar hipótesis, que pueden ser posteriormente aceptadas o refutadas.

Finalmente, sobre la rentabilidad del producto es importante notar la solución usa software libre, por lo que no se requirió la inversión de capital en software de paga. Los posibles costos en los que debe incurrir el cliente en un futuro, corresponden al mantenimiento del código. Sin embargo, por el contenido del presente documento, se tiene una documentación amplia del código, en donde se detalla las principales consideraciones de diseño. Por la amplia documentación presente, el mantenenimiento no debería representar un costo significativo. Además, el resultado de implementar este sistema aumentará el nivel de productividad de los investigadores, con lo que se ahorrarán el recurso del tiempo necesario para el procesado posterior de los datos del experimento.

CAPÍTULO 7

CONCLUSIONES Y RECOMENDACIONES

Tras el desarrollo del proyecto, se logró solucionar el problema de la falta de una herramienta de visualización para la memoria a largo plazo de la arquitectura cognitiva *MDB*. Dicha herramienta permite desacoplar la visualización de la memoria, permitiéndole enfocarse únicamente en el aprendizaje para llevar a cabo una serie de tareas definidas. El resultado del proyecto, brindará a

- Se logró diagnosticar correctamente el sistema de visualización utilizado previamente, al determinar sus componentes principales, el funcionamiento específico de cada uno y sus limitaciones.
- Se logró desarrollar un sistema de visualización en forme de una aplicación web multi-página que permite cumplir con los requerimientos del cliente de la siguiente manera:
  - Se logra visualizar la estructura de la *LTM* mediante la implementación del

paquete *Cytoscape* de *Dash*. Además, se da un seguimiento a los datos de la *LTM* al definir el sistema de visualización a través de la definición de mensaje, canales y *callbacks* de *ROS*, descritos en la sección 5.6.

  - Se definió que todos los aspectos de la visualización se resuelvan en el sistema y no en la *LTM* como se hacía previamente. Esto se logra asignando las características del nodo cuando se recibe de la *LTM*.
  - Se logró la actualización de la visualización cada 2 segundos con la incorporación de un elemento en el *layout* de tipo *dcc.Interval*. Además se midió el tiempo de actualización de los datos para 1000 iteraciones, y se mantuvo en todas por debajo del período mencionado.
  - La interactividad se logra mediante la incorporación de botones, cuadros de texto y gráficas que ejecutan una acción cuando su valor cambia.
  - Se logró desarrollar un sistema de pausado mediante la utilización de un botón en la aplicación web y el envío de comandos a la *LTM* a través de *ROS*.
  - Se logró implementar el guardado del gráfico de la red con un botón disponible en el *layout* de la aplicación web.
  - Se logró graficar los puntos y antipuntos de los *PNodes* en dos gráficas donde se obervan los primeros en color azul y los segundos en rojo, y permiten el filtrado de las percepciones, el número de nodo y el tipo de gráfica. Estos datos además se almacenan cada vez que son publicados desde la *LTM*.

- Se logró integrar el sistema de visualización por medio de nodos de *ROS* que se comunican por medio de canales de comunicación donde publican información relevante al experimento.

- Se logró validar el correcto funcionamiento de la herramienta con un robot Baxter, en donde se midió la integridad de la transferencia de los datos y el tiempo

aproximado desde que se envían hasta que se procesan, resultando en una actualización de los datos en un período menor al de la actualización de la visualización y donde no se pierde información del experimento.

7.1 Recomendaciones

El proyecto desarrollado puede mejorarse y otorgar a los investigadores del GII una herramienta que les permita visualizar más información de la *LTM*. Además se pueden hacer mejoras a las definiciones del *layout* de la aplicación wbe actuales. Con respecto a esto se plantean las siguientes recomendaciones.

- Se puede añadir una nueva gráfica en alguno de los *layouts* definidos o en uno nuevo, que permita visualizar el número de nodos de la red, incluso se podría filtrar esta información para mostrar únicamente nodos de tipos específicos.
- A las gráficas de los *PNodes* se puede agregar el mapa de activación (como los que se ven en la figura 3.2
- Se puede añadir un botón que permita almacenar los puntos y antipuntos como un archivo ya sea en formato *txt, csv* o de Excel, con la finalidad de hacer un análisis posterior de los datos.
- Implementar un botón de pausa en el *layout* de los *PNodes*, de modo que el experimento se pueda pausar o reanudar desde cualquier página.
- Desacoplar las percepciones que se grafican en el *layout* de los *PNodes*. Actualmente se grafican el mismo para de percepciones en ambas y puede resultar útil graficar un par en un gráfico y otro par en el otro.
- Añadir la funcionalidad de que los *PNodes* puedan visualizarse en otros tipos de representaciones. Se debe recordar que se pueden representar por medio de redes neuronales por medio de la distancia a puntos y antipuntos.

- Añadir la funcionalidad, para que aparte de visualizar otras representaciones de los *PNodes*, se puede realizar el cambio entre una y otra en medio de un experimento.

REFERENCIAS BIBLIOGRÁFICAS

[1] Grupo Integrado de Ingeniería. Presentación, 2014.

[2] Grupo Integrado de Ingeniería. Mecanismos Cognitivos Grupo Integrado de Ingeniería, 2014.

[3] John F Kihlstrom. Unconscious Cognition. In *Reference Module in Neuroscience and Biobehavioral Psychology*. Elsevier, 2018.

[4] S.J Shettleworth. *Cognition, Evolution and Behavior*. Oxford University Press, 1998.

[5] Paul Thagard. Cognitive Science. In Edward N Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 201 edition, 2019.

[6] Iuliia Kotseruba and John K. Tsotsos. *40 Years of Cognitive Architectures: Core Cognitive Abilities and Practical Applications*, volume 53. Springer Netherlands, 2020.

[7] Ron Sun. The importance of cognitive architectures: An analysis based on CLARION. *Journal of Experimental and Theoretical Artificial Intelligence*, 19(2):159–193, 2007.

[8] Richard J Duro, Jose A Becerra, Juan Monroy, and Francisco Bellas. Perceptual Generalization and Context in a Network Memory Inspired Long-Term Memory for Artificial Cognition. *International Journal of Neural Systems*, 29(06):1850053, 2019.

[9] Antonio Bandera, Jorge Dias, Markus Vincze, and Luis J. Manso. Special issue on cognitive robotics. *Cognitive Processing*, 19(2):231–232, 2018.

[10] F Bellas, R J Duro, A Faina, and D Souto. Multilevel Darwinist Brain (MDB): Artificial Evolution in a Cognitive Architecture for Real Robots. *IEEE Transactions on Autonomous Mental Development*, 2(4):340–354, 12 2010.

[11] Alejandro Romero, Francisco Bellas, Jose A. Becerra, and Richard J. Duro. Studying How Innate Motivations Can Drive Skill Acquisition in Cognitive Robots. *Proceedings*, 21(1):2, 2019.

[12] Richard J. Duro, Jose A. Becerra, Juan Monroy, and Luis Calvo. Context nodes in the operation of a long term memory structure for an evolutionary cognitive architecture. *GECCO 2017 - Proceedings of the Genetic and Evolutionary Computation Conference Companion*, (July):1172–1176, 2017.

[13] Ilir Jusufi. *Multivariate Networks : Visualization and Interaction Techniques*. PhD thesis, Linnaeus University, 2013.

[14] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, 1994.

[15] Weidong Huang, Jing Luo, Tomasz Bednarz, and Henry Duh. Making graph visualization a user-centered process. *Journal of Visual Languages and Computing*, 48(March):1–8, 2018.

[16] Juan Gómez-Romero, Miguel Molina-Solana, Axel Oehmichen, and Yike Guo. Visualizing large knowledge graphs: A performance analysis. *Future Generation Computer Systems*, 89:224–238, 2018.

[17] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An Open Source Software for Exploring and Manipulating Networks, 2009.

[18] Paul Shannon, Andrew Markiel, Owen Ozier, Nitin S Baliga, Jonathan T Wang, Daniel Ramage, Nada Amin, Benno Schwikowski, and Trey Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome research*, 13(11):2498–2504, 11 2003.

[19] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[20] T M J Fruchterman and E M Reingold. Graph Drawing by Force-directed Placement. *Software-Practice and Experience*, 21(11):1129–1164, 1991.

[21] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[22] P Sahoo. 6 - Tribological performance of electroless Ni-P coatings. In J Paulo Davim, editor, *Materials and Surface Engineering*, Woodhead Publishing Reviews: Mechanical Engineering Series, pages 163–205. Woodhead Publishing, 2012.

[23] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*, volume 51. Springer US, 1 edition, 2001.

[24] Jiju Antony. *Design of Experiments for Engineers and Scientists*. Elsevier, Oxford, second edi edition, 2014.

[25] Ian Sommerville. *Ingeniería de Software*. Pearson Educación, México, 9 edition, 2011.

[26] Adedeji Badiru. *Systems Engineering Models: Theory, Methods, and Applications*. 2019.

[27] Jose A. Becerra, Richard J. Duro, and Juan Monroy. A Redescriptive Approach to Autonomous Perceptual Classification in Robotic Cognitive Architectures. *Proceedings of the International Joint Conference on Neural Networks*, 2018-July(640891):1–8, 2018.

[28] John Hunter, Darren Dale, Eric Firing, Michael Droettboom, and Matplotlib development team. Matplotlib, 2020.

[29] Malcom Waskom. seaborn, 2020.

[30] Plotly. Plotly, 2020.

[31] Bokeh Contributors. bokeh, 2020.

[32] Plotly. Dash, 2020.

[33] Networkx. Networkx network analysis in python, 2020.

[34] Pandas. oandas-python data analysis library, 2020.

ANEXOS

En esta sección se presentan documentos importantes para el desarrollo del proyecto como lo son secciones de código o programas completos, que permiten cumplir con las diferentes etapas del proyecto. De este modo en la sección 8.1 se detalla el código del sistema de visualización previo, en la sección

8.1 Código del sistema de visualización previo

En esta sección se presentan las cuatro componentes principales del sistema de visualización utilizado previo a la realización de este proyecto.

```
def draw_nodes(self, node_type):
    """Draw nodes of a given type using a specified color map."""
    graph_node_color = []
    nodes = [node for node in self.graph.nodes() if node.type ==
   node_type]
    for node in nodes:
        graph_node_color.append(node.activation)
    circles = networkx.draw_networkx_nodes(
        self.graph,
        self.graph_node_position,
        nodelist=nodes,
        node_size=1200,
        node_color=graph_node_color,
        alpha=0.5,
        cmap=NodeColorMap[node_type].value,
        vmin=0.0,
        vmax=1.0,
        label=node_type,
    )
    if circles:
```

```
        circles.set_edgecolor("k")
```

Listing 8.1: Función para definir las propiedades de los nodos de la LTM

```
@staticmethod
def graph_set_edge_properties(edges):
    """Set edge width and color for graphical representation."""
    graph_edge_width = []
    graph_edge_color = []
    for edge in edges:
        if edge[0].type == "CNode" or edge[1].type == "CNode":
            graph_edge_width.append(1)
            if edge[0].type == "PNode" or edge[1].type == "PNode":
                graph_edge_color.append("m")
            elif edge[0].type == "Goal" or edge[1].type == "Goal":
                graph_edge_color.append("g")
            elif edge[0].type == "ForwardModel" or edge[1].type == "
    ForwardModel":
                graph_edge_color.append("b")
            elif edge[0].type == "Policy" or edge[1].type == "Policy":
                graph_edge_color.append("r")
        else:
            graph_edge_width.append(0.2)
            graph_edge_color.append("k")
    return graph_edge_width, graph_edge_color
```

Listing 8.2: Función para definir las propiedades de las aristas de la LTM

```
def graph_set_perception_nodes_coordinates(self):
    """Set node position in LTM graph for a Perception node."""
    for idx, same_blood in enumerate(self.nodes["Perception"]):
        self.graph_node_position[same_blood] = [0, idx + 1]

def graph_set_p_node_nodes_coordinates(self):
    """Set node position in LTM graph for a P-node node."""
    for idx, same_blood in enumerate(self.nodes["PNode"]):
        posx = idx // 3
        posy = idx % 3
        self.graph_node_position[same_blood] = [1 + posx + posy / 3.0,
    posy]

def graph_set_goal_nodes_coordinates(self):
    """Set node position in LTM graph for a Goal node."""
    for idx, same_blood in enumerate(self.nodes["Goal"]):
        self.graph_node_position[same_blood] = [3 * (idx + 1), 4]

def graph_set_forward_model_nodes_coordinates(self):
    """Set node position in LTM graph for a Forward Model node."""
    for idx, same_blood in enumerate(self.nodes["ForwardModel"]):
        self.graph_node_position[same_blood] = [3 * (idx + 1), 6]

def graph_set_policy_nodes_coordinates(self):
    """Set node position in LTM graph for a Policy node."""
```

```
    for idx, same_blood in enumerate(self.nodes["Policy"]):
        self.graph_node_position[same_blood] = [9, idx + 1]

def graph_set_c_node_nodes_coordinates(self):
    """Set node position in LTM graph for a C-node node."""
    for idx, same_blood in enumerate(self.nodes["CNode"]):
        posx = idx // 3
        posy = idx % 3
        self.graph_node_position[same_blood] = [1 + posx + posy / 3.0,
    posy + 8]
```

Listing 8.3: Funciones para definir las coordenadas de cada tipo de nodo de la LTM

```
def show(self):
    """Plot the LTM structure."""
    pyplot.get_current_fig_manager().window.geometry("800x600-0-0")
    pyplot.clf()
    pyplot.ioff()
    pyplot.axis("off")
    pyplot.subplots_adjust(left=0.0, right=1.0, bottom=0.0, top=0.9)
    for node_type in self.nodes:
        if node_type != "Goal":
            self.draw_nodes(node_type)
    graph_edge_width, graph_edge_color = self.
   graph_set_edge_properties(self.graph.edges())
    networkx.draw_networkx_edges(
        self.graph, self.graph_node_position, width=graph_edge_width,
   edge_color=graph_edge_color, alpha=0.5
    )
    networkx.draw_networkx_labels(self.graph, self.graph_node_position
   , labels=self.graph_node_label, font_size=8)
    if not self.current_reward:
        pyplot.title(
            "GOAL: None WORLD: "
            + self.current_world
            + "\nITERATION: "
            + str(self.iteration)
            + " REWARD: "
            + str(self.current_reward)
            + "\nPOLICY: "
            + self.current_policy.ident,
            fontsize=10,
        )
    else:
        pyplot.title(
            "GOAL: "
            + self.current_goal.ident
            + " WORLD: "
            + self.current_world
            + "\nITERATION: "
            + str(self.iteration)
            + " REWARD: "
            + str(self.current_reward)
            + "\nPOLICY: "
```

```
            + self.current_policy.ident,
            fontsize=10,
        )
    # Legend is disable due to all nodes have the same color due to,
    when we paint nodes, we change color map.
    # pyplot.legend(
    #     loc="bottom right", shadow=True, fancybox=True, fontsize=10,
    labelspacing=3, bbox_to_anchor=(1.30, 0.5))
    # You will not get any graph on the screen without this line
    pyplot.pause(0.0001)
```

Listing 8.4: Función show de la LTM

8.2 Código empleado para la simulación de la LTM

En el siguiente código se implementó la simulación del funcionamiento de la LTM y que se emplea para la generación de datos de prueba del sistema de visualización.

```
#!usr/env/python3
import networkx as nx
import random
import pandas as pd
import yaml
import yamlloader

class LTM():
    def __init__(self):
        self.threshold = 0.5
        self.graph = nx.Graph()
        self.iteration = 0
        self.perceptions = []
        self.pnode_dict = {}
        self.nodes_config = {'PNode': 0,
                             'CNode': 0,
                             'Goal':  0,
                             'ForwardModel': 0,
                             'Goal': 0,
                             'Perception': 0,
                             'Policy': 0}

    def set_pos(self, name):
        idx = self.nodes_config[name]

        if(name == "PNode"):
            posx = idx // 3
            posy = idx % 3
            posx = 1 + posx + posy / 3.0
            color = 'purple'
        elif(name == "Goal"):
```

```
            posx, posy = 3*idx, 4
            color = 'green'
        elif(name == "ForwardModel"):
            posx, posy = 3 * idx, 6
            color = 'orange'
        elif(name == "Policy"):
            posx, posy = 13, idx + 3
            color = 'red'
        elif(name == "CNode"):
            posx = idx // 3
            posy = idx % 3
            posx, posy = 1 + posx + posy / 3.0, posy + 8
            color = 'blue'
        elif(name == "Perception"):
            posx, posy = -3, idx + 3
            color = 'grey'

        return 75*posx, 75*posy, color

    def load_graph(self, filename):
        print('Loading graph')
        configuration = yaml.load(open(filename, "r", encoding="utf-8"
    ), Loader=yamlloader.ordereddict.CLoader)
        nodes = configuration["LTM"]["Nodes"]
        for node_type, node_list in nodes.items():
            print("Loading %s...", node_type)
            for element in node_list:
                ident = element["id"]
                posx, posy, color = self.set_pos(node_type)
                self.graph.add_node(
                    ident,
                    node_type = node_type,
                    name = ident,
                    activation = random.random(),
                    position =  {'x': posx, 'y': posy},
                    color = color
                )
                print('Added ' + ident)
                if(node_type == 'Perception'):
                    self.perceptions.append(ident)
                else:
                    for perception in self.perceptions:
                        self.graph.add_edge(ident, perception)

                self.nodes_config[node_type] += 1

        self.graph.add_node('title',
                            node_type = 'title',
                            name = '',
                            activation = 0,
                            position =  {'x': 5*75, 'y': -0.5*75})

    def read_perceptions(self):
        perceptions = {}
```

```
85          for node in self.graph.nodes:
86              if(self.graph.nodes()[node]['node_type'] == 'Perception'):
87                  perceptions[node] = [self.graph.nodes()[node]['
    activation']]
88          perc_df = pd.DataFrame(perceptions)
89          return perc_df
90
91      def update_activations(self):
92          for node in self.graph.nodes:
93              if(self.graph.nodes()[node]['node_type'] != 'title'):
94                  self.graph.nodes()[node]['activation'] = random.random
    ()
95
96      def select_policy(self):
97          max_act = 0
98          policy = ''
99          for node in self.graph.nodes:
100             if(self.graph.nodes()[node]['node_type'] == 'Policy'):
101                 if(self.graph.nodes()[node]['activation'] > max_act):
102                     max_act = self.graph.nodes()[node]['activation']
103                     policy = self.graph.nodes()[node]['name']
104         return policy
105
106     def select_forward(self):
107         max_act = 0
108         forward = ''
109         for node in self.graph.nodes:
110             if(self.graph.nodes()[node]['node_type'] == 'ForwardModel'
    ):
111                 if(self.graph.nodes()[node]['activation'] > max_act):
112                     max_act = self.graph.nodes()[node]['activation']
113                     forward = self.graph.nodes()[node]['name']
114         return forward
115
116     def select_goal(self):
117         max_act = 0
118         goal = ''
119         for node in self.graph.nodes:
120             if(self.graph.nodes()[node]['node_type'] == 'Goal'):
121                 if(self.graph.nodes()[node]['activation'] > max_act):
122                     max_act = self.graph.nodes()[node]['activation']
123                     goal = self.graph.nodes()[node]['name']
124         return goal
125
126     def get_reward(self):
127         return random.random()
128
129     def create_cnode(self, perceptions, policy,  goal, point):
130         print('Creating CNode at iteration: ' + str(self.iteration))
131         name_c = 'CNode ' + str(self.nodes_config['CNode'])
132         name_p = 'PNode ' + str(self.nodes_config['PNode'])
133
134         self.nodes_config['CNode'] += 1
135         self.nodes_config['PNode'] += 1
```

```
136
137         posx, posy, color = self.set_pos('CNode')
138         self.graph.add_node(
139             name_c,
140             node_type = 'CNode',
141             name = name_c,
142             activation = random.random(),
143             position =  {'x': posx, 'y': posy},
144             color = color
145         )
146         perceptions['color'] = 'blue' if point == 1 else 'red'
147         self.pnode_dict[name_p] = perceptions
148
149         posx, posy, color = self.set_pos('PNode')
150
151         self.graph.add_node(
152             name_p,
153             node_type = 'PNode',
154             name = name_p,
155             activation = random.random(),
156             position =  {'x': posx, 'y': posy},
157             color = color
158         )
159         self.graph.add_edge(name_c, name_p, color = 'purple')
160         self.graph.add_edge(name_c, policy, color = 'red')
161         self.graph.add_edge(name_c, goal, color = 'green')
162         forward = self.select_forward()
163         self.graph.add_edge(name_c, forward, color = 'orange')
164         for perception in self.perceptions:
165             self.graph.add_edge(name_p, perception)
166
167     def update_cnodes(self, perceptions, policy, goal, point):
168         ntps = self.graph.neighbors(policy)
169         cnode_con = False
170         for ntp in ntps:
171             node_type = self.graph.nodes()[ntp]['node_type']
172             if(node_type == 'CNode'):
173                 ntcs = self.graph.neighbors(ntp)
174                 cnode_con = True
175                 for ntc in ntcs:
176                     node_type_2 = self.graph.nodes()[ntc]['node_type'
    ]
177                     if(node_type_2 == 'PNode'):
178                         print('Adding point or antipoint to ' + ntc)
179                         perceptions['color'] = 'blue' if point == 1
    else 'red'
180                         self.pnode_dict[ntc] = pd.concat([self.
    pnode_dict[ntc], perceptions], ignore_index = True)
181         if(not(cnode_con)):
182             self.create_cnode(perceptions, policy, goal, point)
183
184     def set_title(self, reward, policy, world, goal):
185         self.graph.nodes()['title']['name'] = "GOAL: " + goal + "
    WORLD: " + world + '\n' + "ITERATION: " + str(self.iteration) +  "
```

```
    REWARD: "+ str(reward) + '\n' + "POLICY: " + policy

    def ltm_sim(self):
        self.update_activations()
        policy = self.select_policy()
        perceptions = self.read_perceptions()
        reward = self.get_reward()
        world = self.select_forward()
        goal = self.select_goal()
        if(reward < self.threshold):
            self.update_cnodes(perceptions, policy, goal, 0)
        else:
            self.update_cnodes(perceptions, policy, goal, 1)
        self.iteration += 1
        self.set_title(reward, policy, world, goal)
```

Listing 8.5: Código para definir el simulador de la LTM

8.3 Código empleado para la definición del layout

En esta sección se detalla el código de cada uno de los layouts presentados en la sección 5.4.3.

```
import dash
import dash_cytoscape as cyto
import plotly.graph_objects as go
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash(__name__, suppress_callback_exceptions=True)

app.layout = html.Div([
            html.H1('LTM', style={'textAlign': 'center'}),
            html.Div(
                style = {'width': '100%', 'display': 'flex', 'align-
    items': 'center', 'justify-content': 'center'},
                children = [
                    html.Button(
                        children = 'Pausa',
                        title = 'Inicio',
                        id = 'pause',
                        n_clicks = 0,
                        style={'margin':'10px'}
                    ),
                    html.Button(
                        children = 'Guardar',
                        title = 'Guardar',
                        id = 'save',
                        style={'margin':'10px'}
```

```
                    )
                ]
            ),
            html.Div(
                className="eight columns",
                style = {'background-color' : 'white'},
                children=[
                    dcc.Graph(
                        className="eight columns",
                        id="graph-with-update"
                        )
                ]
            ),
            html.Div(
                style = {'background-color': '#6f8bc0'},
                className="three columns",
                children=[
                    html.Div(
                        className='twelve columns',
                        children=[
                            dcc.Markdown(children = "Informacion del
    nodo", id = 'hover_mk', style={'textAlign': 'center'}),
                            html.Pre(
                                id='hover-data',
                                children = 'Nombre del node: \nTipo de
     nodo: \nActivacion: \n ')
                        ],
                        style={'height': '100%'}),
                    html.Div(
                        className='twelve columns',
                        children=[
                            dcc.Markdown(children = "Informacion del
    nodo", id = 'click_mk', style={'textAlign': 'center'}),
                            html.Pre(
                                id='click-data',
                                children = 'Nombre del node: \nTipo de
     nodo: \nActivacion: \n ')
                        ],
                        style={'height': '100%'}),
                    ]
            ),
            dcc.Interval(
                id = 'interval-ltm',
                interval = 1*1000,
                n_intervals = 0)
            ])

if __name__ == '__main__':
    app.run_server(debug = False)
```

Listing 8.6: Código para definir la primera versión del layout de la aplicación web

```
import dash
import dash_cytoscape as cyto
```

```
import plotly.graph_objects as go
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash(__name__, suppress_callback_exceptions=True)

app.layout = html.Div([
            html.H1('LTM', style={'textAlign': 'center'}),
            html.Div(
                style = {'width': '100%', 'display': 'flex', 'align-
    items': 'center', 'justify-content': 'center'},
                children = [
                    html.Button(
                        children = 'Pausa',
                        title = 'Inicio',
                        id = 'pause',
                        n_clicks = 0,
                        style={'margin':'10px'}
                    ),
                    html.Button(
                        children = 'Guardar',
                        title = 'Guardar',
                        id = 'save',
                        style={'margin':'10px'}
                    )
                ]
            ),
            html.Div(
                className="eight columns",
                style = {'background-color' : 'white'},
                children=[
                    dcc.Dropdown(
                        className="eight columns",
                        id='option-dropdown',
                        options=[
                            {'label': 'LTM', 'value': 'ltm'},
                            {'label': 'PNode 1', 'value': 'pn1'},
                            {'label': 'PNode 2', 'value': 'pn2'}
                        ],
                        style = {'width': '100%', 'color': 'black'},
                        value='LTM'
                    ),
                    dcc.Graph(
                        className="eight columns",
                        id="graph-with-update"
                        )
                ]
            ),
            html.Div(
                style = {'background-color': '#6f8bc0'},
                className="three columns",
                children=[
                    html.Div(
                        className='twelve columns',
```

```
                        children=[
                            dcc.Markdown(children = "Informacion del
    nodo", id = 'hover_mk', style={'textAlign': 'center'}),
                            html.Pre(
                                id='hover-data',
                                children = 'Nombre del node: \nTipo de
     nodo: \nActivacion: \n ')
                        ],
                        style={'height': '100%'}),
                    html.Div(
                        className='twelve columns',
                        children=[
                            dcc.Markdown(children = "Informacion del
    nodo", id = 'click_mk', style={'textAlign': 'center'}),
                            html.Pre(
                                id='click-data',
                                children = 'Nombre del node: \nTipo de
     nodo: \nActivacion: \n ')
                        ],
                        style={'height': '100%'}),
                    ]
            ),
            dcc.Interval(
                id = 'interval-ltm',
                interval = 1*1000,
                n_intervals = 0)
            ])

if __name__ == '__main__':
    app.run_server(debug = False)
```

Listing 8.7: Código para definir la segunda versión del layout de la aplicación web

```
import dash
import dash_cytoscape as cyto
import plotly.graph_objects as go
import dash_core_components as dcc
import dash_html_components as html

app = dash.Dash(__name__, suppress_callback_exceptions=True)

app.layout = html.Div([
            html.H1('LTM', style={'textAlign': 'center'}),
            html.Div(
                style = {'width': '100%', 'display': 'flex', 'align-
    items': 'center', 'justify-content': 'center'},
                children = [
                    html.Button(
                        children = 'Pausa',
                        title = 'Inicio',
                        id = 'pause',
                        n_clicks = 0,
                        style={'margin':'10px'}
                    ),
```

```
                    html.Button(
                        children = 'Guardar',
                        title = 'Guardar',
                        id = 'save',
                        style={'margin':'10px'}
                    )
                ]
            ),
            html.Div(
                className="eight columns",
                style = {'background-color' : 'white'},
                children=[
                    dcc.Dropdown(
                        className="eight columns",
                        id='option-dropdown',
                        options=[
                            {'label': 'LTM', 'value': 'ltm'},
                            {'label': 'PNode 1', 'value': 'pn1'},
                            {'label': 'PNode 2', 'value': 'pn2'}
                        ],
                        style = {'width': '100%', 'color': 'black'},
                        value='LTM'
                    ),
                    dcc.Graph(
                        className="eight columns",
                        id="graph-with-update"
                        )
                ]
            ),
            html.Div(
                style = {'background-color': '#6f8bc0'},
                className="three columns",
                children=[
                    html.Div(
                        className='twelve columns',
                        children=[
                            dcc.Markdown(children = "Informacion del
nodo", id = 'hover_mk', style={'textAlign': 'center'}),
                            html.Pre(
                                id='hover-data',
                                children = 'Nombre del node: \nTipo de
 nodo: \nActivacion: \n ')
                        ],
                        style={'height': '100%'}),
                    html.Div(
                        className='twelve columns',
                        children=[
                            dcc.Markdown(children = "Informacion del
nodo", id = 'click_mk', style={'textAlign': 'center'}),
                            html.Pre(
                                id='click-data',
                                children = 'Nombre del node: \nTipo de
 nodo: \nActivacion: \n ')
                        ],
```

```
                        style={'height': '100%'}),
                    ]
            ),
            dcc.Interval(
                id = 'interval-ltm',
                interval = 1*1000,
                n_intervals = 0)
            ])

if __name__ == '__main__':
    app.run_server(debug = False)
```

Listing 8.8: Código para definir la tercer versión del layout de la aplicación web

8.4 Código de visualización de la integración del subsistema

En esta sección se presenta el código desarrollado para implementar los callbacks de la aplicación web y la versión final del layout.

```
#!usr/env/python3
import random
import dash
import dash_core_components as dcc
import dash_cytoscape as cyto
import dash_html_components as html
from dash.dependencies import input, Output, State
import plotly.graph_objects as go
import math
import networkx as nx
import os
import pandas as pd
from ltm_sim import LTM

random.seed(0)

class ViEW():
    def __init__(self):
        self.ltm = LTM()
        self.save_dir = ''
        self.paused = False
        self.pnode_temp_trace = go.Scatter()
        self.app = dash.Dash(__name__, suppress_callback_exceptions=
    True)
        self.stylesheet=[
            {
                'selector': 'node',
```

```
                'style': {
                    'label': 'data(name)',
                    'size' : '20%',
                    'font-size' : '18',
                    'background-opacity' : 'data(activation)',
                    'background-color': 'data(color)'
                }
            },
            {
                'selector': 'edge',
                'style': {
                    'width' : '1%',
                    'curve-style' : 'bezier',
                    'line-color' : 'data(color)'
                }
            },
            {
                'selector': '[node_type *= "title"]',
                'style': {
                    'size' : '1%',
                    'text-wrap' : 'wrap',
                    'font-weight': 'bold'
                }
            },
        ]
        self.index_page = html.Div(
                classNamе = 'twelve columns',
                style = {
                    'width': '100%',
                    'display': 'flex',
                    'align-items': 'center',
                    'justify-content': 'center',
                    'textAlign': 'center'
                    },
                children = [
                    html.Div(
                        classNamе = 'four columns',
                        children = [
                            html.H1('LTM'),
                            html.H6('Seleccione esta imagen para ver
la red de nodos de la LTM'),
                            html.A([
                                html.img(
                                    src= self.app.get_asset_url('ltm.
jpg'),
                                    )
                                ], href='/ltm'),
                            ]
                    ),
                    html.Div(
                        classNamе = 'four columns',
                        children = [
                            html.H1('Nodes'),
```

```
                            html.H6('Seleccione esta imagen para ver los puntos y antipuntos de los PNodes'),
                            html.A([
                                html.img(
                                    src= self.app.get_asset_url('ltm.jpg'),
                                    )
                                ], href='/nodes'),
                            ]
                    )
                ])

        self.ltm_layout = html.Div([
                html.H1('LTM', style={'textAlign': 'center'}),
                html.Div(
                    style = {'width': '100%', 'display': 'flex', 'align-items': 'center', 'justify-content': 'center'},
                    children = [
                        html.Button(
                            children = 'Pausa',
                            title = 'inicio',
                            id = 'pause',
                            n_clicks = 0,
                            style={'margin':'10px'}
                        ),
                        html.Button(
                            children = 'Guardar',
                            title = 'Guardar',
                            id = 'save',
                            style={'margin':'10px'}
                        )
                    ]
                ),
                html.Div(
                    className="eight columns",
                    style = {'background-color' : 'white'},
                    children = [
                        cyto.Cytoscape(
                            id='cytoscape-ltm',
                            layout={'name': 'preset'},
                            style={'width': '100%', 'height': '500px'},
                            elements = [],
                            stylesheet = self.stylesheet
                        )
                    ]
                ),
                html.Div(
                    style = {
                        'background-color': '#6f8bc0',
                        'height': '100%'
                        },
                    className="three columns",
                    children=[
```

```
                        html.Div(
                            className='twelve columns',
                            children=[
                                dcc.Markdown(
                                    children = "informacion del nodo",
                                    id = 'hover_mk',
                                    style={
                                        'textAlign': 'center'
                                        }
                                    ),
                                html.Pre(
                                    id='hover-data'
                                    ),
                                dcc.Markdown(
                                    children = "informacion del nodo",
                                    id = 'click_mk',
                                    style={
                                        'textAlign': 'center'
                                        }
                                    ),
                                html.Pre(
                                    id='click-data'
                                    )
                            ]
                        )
                    ]
                ),
                html.Div(
                    className = 'twelve columns',
                    children = [
                        dcc.Link('Nodes', href='/nodes'),
                        dcc.interval(
                            id = 'interval-ltm',
                            interval = 1*1000,
                            n_intervals = 0)
                            ]
                        )
                ])

        self.node_layout = html.Div([
                html.H1('Nodes', style={'textAlign': 'center'}),
                html.Div(
                    className = 'five columns',
                    style = {
                        'width': '37%',
                        'display': 'inline-block',
                        'align-items': 'center',
                        'justify-content': 'center'
                        },
                    children = [
                        dcc.Dropdown(
                            id='nodes-dropdown1',
                            options=[],
                            value='',
```

```
                            style = {'width': '100%', 'color': 'black'
    }
                            ),
                        dcc.Graph(
                            id="graph-with-nodes1"
                            )
                    ]
                ),
                html.Div(
                    className = 'five columns',
                    style = {
                        'width': '37%',
                        'display': 'inline-block',
                        'align-items': 'center',
                        'justify-content': 'center'
                        },
                    children = [
                        dcc.Dropdown(
                            id='nodes-dropdown2',
                            options=[],
                            value='',
                            style = {'width': '100%', 'color': 'black'
    }
                            ),
                        dcc.Graph(
                            id="graph-with-nodes2"
                            )
                    ]
                ),
                html.Div(
                    className="two columns",
                    style = {
                        'align-items': 'center',
                        'height': '100%'
                        },
                    children=[
                            dcc.Markdown(
                                children = "Configuracion",
                                id = 'config',
                                style={
                                    'textAlign': 'center',
                                    'background-color': '#6f8bc0'
                                    }
                            ),
                            html.Pre(
                                id='info_config',
                                children = ['Seleccione el tipo de
    grafica \nque desea observar']
                            ),
                            dcc.Radioitems(
                                id = 'radio_type',
                                options=[
                                    {'label': 'Polar', 'value': 'pol'
    },
```

```
                                    {'label': 'Rectangular', 'value':
'rect'},
                                ],
                                value='rect',
                                labelStyle={'display': 'inline-block'}
                            ),
                            html.Pre(
                                id='info_perc',
                                children = ['Seleccione las
percepciones \npor graficar']
                            ),
                            dcc.Dropdown(
                                id = 'perceptions1',
                                options=[],
                                value='',
                                style = {'width': '100%', 'color': '
black'}
                            ),
                            dcc.Dropdown(
                                id = 'perceptions2',
                                options=[],
                                value='',
                                style = {'width': '100%', 'color': '
black'}
                            )
                        ],
                ),
                html.Div(
                    className = 'twelve columns',
                    children = [
                        dcc.Link('LTM', href='/ltm'),
                        dcc.interval(
                            id = 'interval-nodes',
                            interval = 1*1000,
                            n_intervals = 0
                        )
                    ]
                ),
                ])

    def define_app(self):
        self.app.layout = html.Div([
            dcc.Location(id='url', refresh=False),
            html.Div(id='page-content')
        ])

        self.app.callback(
            Output('nodes-dropdown1', 'options'),
            Output('nodes-dropdown2', 'options'),
            Output('graph-with-nodes1', 'figure'),
            Output('graph-with-nodes2', 'figure'),
            Output('perceptions1', 'options'),
            Output('perceptions2', 'options'),
            [input('interval-nodes', 'n_intervals'),
```

```
            input('nodes-dropdown1', 'value'),
            input('nodes-dropdown2', 'value'),
            input('perceptions1', 'value'),
            input('perceptions2', 'value'),
            input('radio_type', 'value')]
            )(self.update_fig_nodes)

        self.app.callback(
            Output('cytoscape-ltm', 'elements'),
            [input('interval-ltm', 'n_intervals')]
            )(self.update_fig_ltm)

        self.app.callback(
            Output('pause', 'children'),
            Output('interval-ltm', 'interval'),
            [input('pause', 'n_clicks')]
            )(self.update_pause)

        self.app.callback(
            Output("cytoscape-ltm", "generateimage"),
            [input('save', 'n_clicks')]
            )(self.save_graph)

        self.app.callback(
            Output('click-data', 'children'),
            Output('click_mk', 'children'),
            [input('cytoscape-ltm', 'tapNodeData')]
        )(self.display_info)

        self.app.callback(
            Output('hover-data', 'children'),
            Output('hover_mk', 'children'),
            [input('cytoscape-ltm', 'mouseoverNodeData')]
        )(self.display_info)

        self.app.callback(
            Output('page-content', 'children'),
            [input('url', 'pathname')]
        )(self.display_page)

    def update_fig_nodes(self, n_intervals, value1, value2, value3,
value4, value5):
        options = [{'label': self.ltm.graph.nodes()[node]['name'], '
value': self.ltm.graph.nodes()[node]['name']} for node in self.ltm.
graph.nodes() if self.ltm.graph.nodes()[node]['node_type'] == '
PNode']
        options2 = [{'label': self.ltm.graph.nodes()[node]['name'], '
value': self.ltm.graph.nodes()[node]['name']} for node in self.ltm.
graph.nodes() if self.ltm.graph.nodes()[node]['node_type'] == '
Perception']
        try:
            trace1 = self.update_pnode_figure(value1, value3, value4,
value5)
```

```
            trace2 = self.update_pnode_figure(value2, value3, value4, value5)
        except:
            trace1 = self.pnode_temp_trace
            trace2 = self.pnode_temp_trace
        fig1 = go.Figure(
                data = [trace1],
                layout = go.Layout(
                    title = value1 + ' en la iteracion ' + str(self.ltm.iteration),
                    xaxis_title = value3,
                    yaxis_title = value4,
                    titlefont_size = 16,
                    autosize = True,
                    showlegend = False,
                    hovermode ='closest',
                    margin = dict(b = 0,l = 0,r = 0,t = 30),
                    xaxis = dict(showgrid = False, zeroline = False, showticklabels = True),
                    yaxis = dict(showgrid = False, zeroline = False, showticklabels = True))
                )

        fig2 = go.Figure(
                data = [trace2],
                layout=go.Layout(
                    title = value2 + ' en la iteracion ' + str(self.ltm.iteration),
                    xaxis_title = value3,
                    yaxis_title = value4,
                    autosize = True,
                    showlegend = False,
                    hovermode ='closest',
                    margin = dict(b = 0,l = 0,r = 0,t = 30),
                    xaxis = dict(showgrid = False, zeroline = False, showticklabels = True),
                    yaxis = dict(showgrid = False, zeroline = False, showticklabels = True))
                )

        return options, options, fig1, fig2, options2, options2

    def update_pause(self, n_clicks):
        self.paused = not(self.paused)
        if(self.paused):
            print('Se ha pausado el proceso')
            return 'Reanudar', 1*1000*60*60
        print('Se ha reanudado el proceso')
        return 'Pausar', 1000

    def save_graph(self, n_clicks2):
        ftype = 'png'
        filename = 'ltm_' + str(n_clicks2)
        action = 'download'
```

```
374
375         if(n_clicks2 != None):
376             return {
377                     'type': ftype,
378                     'action' : action,
379                     'filename': filename
380                 }
381         else:
382             return {}
383
384     def display_page(self, pathname):
385         if pathname == '/ltm':
386             return self.ltm_layout
387         elif pathname == '/nodes':
388             return self.node_layout
389         else:
390             return self.index_page
391
392     def update_pnode_figure(self, value, xaxis, yaxis, type_plot):
393         if self.read_from_csv:
394             size = len(self.ltm.pnode_dict[value])
395             filename = os.getcwd() + '/minmax_normalized_output/' +
    value.replace(' ', '').lower() + '.csv'
396             df = pd.read_csv(filename, nrows = size)
397             df['color'] = ['blue' if activation == 1 else 'red' for
    activation in df['activation']]
398         else:
399             df = self.ltm.pnode_dict[value]
400         if(type_plot == 'pol'):
401             trace = go.Scatterpolar(
402                 r = [], theta = [],
403                 mode='markers',
404                 hoverinfo='text',
405                 marker=dict(
406                     showscale=False,
407                     reversescale=True,
408                     size = 10,
409                     sizemode = 'area',
410                     line_width = 2,
411                     opacity = 1)
412                 )
413             trace.r = df[xaxis]
414             trace.theta = df[yaxis]*180/math.pi
415             trace.marker.color = df['color']
416         else:
417             trace = go.Scatter(
418             x=[], y=[],
419             mode='markers+text',
420             hoverinfo='text',
421             textposition='bottom center',
422             marker=dict(
423                 showscale=False,
424                 reversescale=True,
425                 size= 10,
```

```
                sizemode = 'area',
                line_width = 2,
                opacity = 1)
            )
            trace.x = df[xaxis]
            trace.y = df[yaxis]
            trace.marker.color = df['color']
        return trace

    def display_info(self, data):
        display = ''
        title = 'informacion del nodo'
        node_name = 'Nodo'
        if data:
            node = data['id']
            node_name = data['name']
            display = 'Tipo de nodo: {}\n'.format(data['node_type'])
            display += 'Nombre del nodo: {}\n'.format(data['id'])
            display += 'Activacion: {}\n'.format(data['activation'])
            adj_nodes = [n for n in self.ltm.graph.adj[node].keys()]
            display += 'Conectado con: \n'
            for node in adj_nodes:
                display += ' '*5 + self.ltm.graph.nodes()[node]['name'] + '\n'
                display += ' '*7 + 'Tipo de nodo: {}\n'.format(self.ltm.graph.nodes()[node]['node_type'])
                display += ' '*7 + 'Activacion: {}\n'.format(self.ltm.graph.nodes()[node]['activation'])
            title = 'informacion de ' + node_name

        return display, title

    def update_fig_ltm(self, n_intervals):
        if(not(self.paused)):
            self.ltm.ltm_sim()

        elements = cyto_data = nx.readwrite.json_graph.cytoscape.cytoscape_data(self.ltm.graph)
        nodes = cyto_data['elements']['nodes']
        for node in nodes:
            node['position'] = node.get('data').get('position')
            del node['data']['position']
        edges = cyto_data['elements']['edges']
        elements = nodes + edges
        return elements

if __name__ == '__main__':
    view = ViEW()
    view.ltm.load_graph('config.yaml')
    view.read_from_csv = True
    view.define_app()
```

```
473     view.app.run_server(debug = True)
```

Listing 8.9: Codigo para implementar el sistema de visualizacion propuesto y la cuarta version del layout de la pagina web. Elaboracion propia

Además, se muestra el archivo de configuración utilizado para obtener los resultados expuestos en la sección 6.2.

```
LTM:
    Nodes:
        Perception:
            -
                id: basket_size
            -
                id: basket_distance
            -
                id: basket_angle
            -
                id: object_size
            -
                id: object_distance
            -
                id: object_angle
            -
                id: ball_in_left_hand
            -
                id: ball_in_right_hand

        ForwardModel:
            -
                id: gripper_and_low_friction
            -
                id: no_gripper_and_high_friction
        Policy:
            -
                id: grasp_object
            -
                id: grasp_with_two_hands
            -
                id: change_hands
            -
                id: sweep_object
            -
                id: put_object_in_box
            -
                id: put_object_with_robot
            -
                id: throw
            -
                id: ask_nicely
        Goal:
            -
                id: object_in_basket
```

```
46              -
47                  id: object_near_robot
```

Listing 8.10: Archivo de configuración en formato *yaml* para experimentos de integración del susbistema.

8.5 Archivo principal del paquete de *ROS mdb_view*

Como se mencionó en la sección 5.6.5 el resultado del proyecto se puede resumir en un paquete de *ROS* en donde se implementa un nodo con el sistema de visualización.

El siguiente código corresponde a la implementación de la clase *VIEW*, con la que se define el *layout* de la aplicación web, sus *callbacks*, el nodo de *ROS* y sus respectivos *callbacks*.

```
1  #!/usr/bin/env python3
2  import os
3  import dash
4  import json
5  import math
6  import rospy
7  import random
8  import sys
9  import yaml
10 import time
11 import yamlloader
12 import pandas as pd
13 import networkx as nx
14 import dash_cytoscape as cyto
15 import plotly.graph_objects as go
16 import dash_core_components as dcc
17 import dash_html_components as html
18 from collections import OrderedDict
19 from networkx.readwrite import json_graph
20 from dash.dependencies import Input, Output, State
21
22 class VIEW(object):
23     def __init__(self):
24         #INFO VARIABLES
25         self.iteration = 0
26         self.current_world = ""
27         self.current_policy = ""
28         self.current_reward = 0
29         self.current_goal = ""
```

```
30
31          #GRAPH CONSTRUCTION VARIABLES
32          self.graph = nx.Graph()
33          self.pnode_dict = {}
34          self.nodes_config = {'PNode': 0,
35                               'CNode': 0,
36                               'Goal':  0,
37                               'ForwardModel': 0,
38                               'Goal': 0,
39                               'Perception': 0,
40                               'Policy': 0}
41          self.init_graph()
42
43          #CONTROL VARIABLES
44          self.paused = True
45          self.save_dir = ""
46          self.default_class = OrderedDict()
47          self.default_ros_node_prefix = OrderedDict()
48          self.default_ros_data_prefix = OrderedDict()
49
50          #VISUALIZATION CONTAINERS AND STYLES
51          self.pnode_temp_trace = go.Scatter()
52          self.app = dash.Dash(__name__, suppress_callback_exceptions=
    True)
53          self.sens_names = []
54          self.stylesheet=[
55              {
56                  'selector': 'node',
57                  'style': {
58                      'label': 'data(id)',
59                      'size' : '20%',
60                      'font-size' : '18',
61                      'background-opacity' : 'data(activation)',
62                      'background-color': 'data(color)'
63                  }
64              },
65              {
66                  'selector': 'edge',
67                  'style': {
68                      'width' : 'data(width)',
69                      'curve-style' : 'bezier',
70                      'line-color' : 'data(color)'
71                  }
72              },
73              {
74                  'selector': '[node_type *= "title"]',
75                  'style': {
76                      'size' : '1%',
77                      'text-wrap' : 'wrap',
78                      'font-weight': 'bold'
79                  }
80              },
81          ]
82
```

```
83          self.index_page = html.Div(
84                  className = 'twelve columns',
85                  style = {
86                      'width': '100%',
87                      'display': 'flex',
88                      'align-items': 'center',
89                      'justify-content': 'center',
90                      'textAlign': 'center'
91                      },
92                  children = [
93                      html.Div(
94                          className = 'four columns',
95                          children = [
96                              html.H1('LTM'),
97                              html.H6('Seleccione esta imagen para ver
    la red de nodos de la LTM'),
98                              html.A([
99                                  html.Img(
100                                     src= self.app.get_asset_url('ltm.
    jpg'),
101                                     )
102                                 ], href='/ltm'),
103                             ]
104                     ),
105                     html.Div(
106                         className = 'four columns',
107                         children = [
108                             html.H1('Nodes'),
109                             html.H6('Seleccione esta imagen para ver
    los puntos y antipuntos de los PNodes'),
110                             html.A([
111                                 html.Img(
112                                     src= self.app.get_asset_url('ltm.
    jpg'),
113                                     )
114                                 ], href='/nodes'),
115                             ]
116                     )
117                 ])
118
119         self.ltm_layout = html.Div([
120                 html.H1('LTM', style={'textAlign': 'center'}),
121                 html.Div(
122                     style = {'width': '100%', 'display': 'flex', '
    align-items': 'center', 'justify-content': 'center'},
123                     children = [
124                         html.Button(
125                             children = 'Pausa',
126                             title = 'Inicio',
127                             id = 'pause',
128                             n_clicks = 0,
129                             style={'margin':'10px'}
130                         ),
131                         html.Button(
```

```
                                children = 'Guardar',
                                title = 'Guardar',
                                id = 'save',
                                style={'margin':'10px'}
                            )
                        ]
                    ),
                    html.Div(
                        className="eight columns",
                        style = {'background-color' : 'white'},
                        children = [
                            cyto.Cytoscape(
                                id='cytoscape-ltm',
                                layout={'name': 'preset'},
                                style={'width': '100%', 'height': '600px'
},
                                elements = [],
                                stylesheet = self.stylesheet
                            )
                        ]
                    ),
                    html.Div(
                        style = {
                            'background-color': '#6f8bc0',
                            'height': '100%'
                            },
                        className="three columns",
                        children=[
                            html.Div(
                                className='twelve columns',
                                children=[
                                    dcc.Markdown(
                                        children = "Informaci[U+FFFD]n del nodo"
,
                                        id = 'hover_mk',
                                        style={
                                            'textAlign': 'center'
                                            }
                                        ),
                                    html.Pre(
                                        id='hover-data'
                                        ),
                                    dcc.Markdown(
                                        children = "Informaci[U+FFFD]n del nodo"
,
                                        id = 'click_mk',
                                        style={
                                            'textAlign': 'center'
                                            }
                                        ),
                                    html.Pre(
                                        id='click-data'
                                        )
                                ]
```

```
                        )
                    ]
                ),
                html.Div(
                    className = 'twelve columns',
                    children = [
                        dcc.Link('Nodes', href='/nodes'),
                        dcc.Interval(
                            id = 'interval-ltm',
                            interval = 1*1000,
                            n_intervals = 0)
                            ]
                        )
                ])

        self.node_layout = html.Div([
                html.H1('Nodes', style={'textAlign': 'center'}),
                html.Div(
                    className = 'five columns',
                    style = {
                        'width': '37%',
                        'display': 'inline-block',
                        'align-items': 'center',
                        'justify-content': 'center'
                        },
                    children = [
                        dcc.Dropdown(
                            id='nodes-dropdown1',
                            options=[],
                            value='',
                            style = {'width': '100%', 'color': 'black'
    }
                            ),
                        dcc.Graph(
                            id="graph-with-nodes1"
                            )
                    ]
                ),
                html.Div(
                    className = 'five columns',
                    style = {
                        'width': '37%',
                        'display': 'inline-block',
                        'align-items': 'center',
                        'justify-content': 'center'
                        },
                    children = [
                        dcc.Dropdown(
                            id='nodes-dropdown2',
                            options=[],
                            value='',
                            style = {'width': '100%', 'color': 'black'
    }
                            ),
```

```
                        dcc.Graph(
                            id="graph-with-nodes2"
                            )
                    ]
                ),
                html.Div(
                    className="two columns",
                    style = {
                        'align-items': 'center',
                        'height': '100%'
                        },
                    children=[
                            dcc.Markdown(
                                children = "Configuraci[U+FFFD]n",
                                id = 'config',
                                style={
                                    'textAlign': 'center',
                                    'background-color': '#6f8bc0'
                                    }
                            ),
                            html.Pre(
                                id='info_config',
                                children = ['Seleccione el tipo de
gr[U+FFFD]fica \nque desea observar']
                            ),
                            dcc.RadioItems(
                                id = 'radio_type',
                                options=[
                                    {'label': 'Polar', 'value': 'pol'
},
                                    {'label': 'Rectangular', 'value':
'rect'},
                                ],
                                value='rect',
                                labelStyle={'display': 'inline-block'}
                            ),
                            html.Pre(
                                id='info_perc',
                                children = ['Seleccione las
percepciones \npor graficar']
                            ),
                            dcc.Dropdown(
                                id = 'perceptions1',
                                options=[],
                                value='',
                                style = {'width': '100%', 'color': '
black'}
                            ),
                            dcc.Dropdown(
                                id = 'perceptions2',
                                options=[],
                                value='',
                                style = {'width': '100%', 'color': '
black'}
```

```
                            )
                        ],
                ),
                html.Div(
                    className = 'twelve columns',
                    children = [
                        dcc.Link('LTM', href='/ltm'),
                        dcc.Interval(
                            id = 'interval-nodes',
                            interval = 1*1000,
                            n_intervals = 0
                        )
                    ]
                ),
                ])

    ########CALLBACK FUNCTIONS##########
    '''Functions used to add interactivity to the web app. Here
    are included the web app callbacks. They include the
    functions to update the PNodes graphs, the LTM network graph,
    functions to pause the simulation, to save the graphs, display
    info when hovering or clicking over a node and to update the
    web page layout'''

    def init_graph(self):
        self.graph.add_node('title',
                            node_type = 'title',
                            id = '',
                            activation = 0,
                            position =  {'x': 5*75, 'y': -0.5*75})
        self.set_title()

    def set_title(self):
        self.graph.nodes()['title']['id'] = "GOAL: {} WORLD: {} \
nITERATION: {} REWARD: {}\nPOLICY: {}".format(self.current_goal,
self.current_world, self.iteration, self.current_reward, self.
current_policy)

    # Functions to update the LTM figure
    def update_fig_ltm(self, n_intervals):
        elements = cyto_data = nx.readwrite.json_graph.cytoscape.
cytoscape_data(self.graph)
        nodes = cyto_data['elements']['nodes']
        for node in nodes:
            node['position'] = node.get('data').get('position')
            # del node['data']['position']
        edges = cyto_data['elements']['edges']
        elements = nodes + edges
        return elements
    # Function to update the node figure
    def update_pnode_figure(self, value, xaxis, yaxis, type_plot):
        df = self.pnode_dict[value]
        if(type_plot == 'pol'):
            print(value, xaxis, yaxis, type_plot)
```

```
333             trace = go.Scatterpolar(
334                 r = [], theta = [],
335                 mode='markers',
336                 hoverinfo='text',
337                 marker=dict(
338                     showscale=False,
339                     reversescale=True,
340                     size = 10,
341                     sizemode = 'area',
342                     line_width = 2,
343                     opacity = 1)
344                 )
345             trace.r = df[xaxis]
346             trace.theta = df[yaxis]*180/math.pi
347         else:
348             trace = go.Scatter(
349             x=[], y=[],
350             mode='markers+text',
351             hoverinfo='text',
352             textposition='bottom center',
353             marker=dict(
354                 showscale=False,
355                 reversescale=True,
356                 size= 10,
357                 sizemode = 'area',
358                 line_width = 2,
359                 opacity = 1)
360             )
361             trace.x = df[xaxis]
362             trace.y = df[yaxis]
363         trace.marker.color =['blue' if confidence == 1 else 'red' for
    confidence in df['confidence']]
364         return trace
365 
366     def update_fig_nodes(self, n_intervals, value1, value2, value3,
    value4, value5):
367         options = [{'label': self.graph.nodes()[node]['id'], 'value':
    self.graph.nodes()[node]['id']} for node in self.graph.nodes() if
    self.graph.nodes()[node]['node_type'] == 'PNode']
368         options2 = [{'label': column, 'value': column} for column in
    self.sens_names]
369         try:
370             trace1 = self.update_pnode_figure(value1, value3, value4,
    value5)
371             trace2 = self.update_pnode_figure(value2, value3, value4,
    value5)
372         except:
373             trace1 = self.pnode_temp_trace
374             trace2 = self.pnode_temp_trace
375 
376         fig1 = go.Figure(
377                 data = [trace1],
378                 layout = go.Layout(
379                     title = value1 + ' en la iteraci[U+FFFD]n ' + str(self.
```

```
    iteration),
380                     xaxis_title = value3,
381                     yaxis_title = value4,
382                     titlefont_size = 16,
383                     autosize = True,
384                     showlegend = False,
385                     hovermode ='closest',
386                     margin = dict(b = 0,l = 0,r = 0,t = 30),
387                     xaxis = dict(showgrid = False, zeroline = False,
    showticklabels = True),
388                     yaxis = dict(showgrid = False, zeroline = False,
    showticklabels = True))
389                 )
390 
391         fig2 = go.Figure(
392                 data = [trace2],
393                 layout=go.Layout(
394                     title = value2 + ' en la iteraci[U+FFFD]n ' + str(self.
    iteration),
395                     xaxis_title = value3,
396                     yaxis_title = value4,
397                     autosize = True,
398                     showlegend = False,
399                     hovermode ='closest',
400                     margin = dict(b = 0,l = 0,r = 0,t = 30),
401                     xaxis = dict(showgrid = False, zeroline = False,
    showticklabels = True),
402                     yaxis = dict(showgrid = False, zeroline = False,
    showticklabels = True))
403                 )
404 
405         return options, options, fig1, fig2, options2, options2
406 
407     # Function to display info on hover or click
408     def display_info(self, data):
409         display = ''
410         title = 'Informaci[U+FFFD]n del nodo'
411         try:
412             node_name = 'Nodo'
413             node = data['id']
414             node_name = data['name']
415             display = 'Tipo de nodo: {}\n'.format(data['node_type'])
416             display += 'Nombre del nodo: {}\n'.format(data['id'])
417             display += 'Activaci[U+FFFD]n: {}\n'.format(data['activation'])
418             adj_nodes = [n for n in self.graph.adj[node].keys()]
419             display += 'Conectado con: \n'
420             for node in adj_nodes:
421                 display += ' '*5 + self.graph.nodes()[node]['id'] + '\
    n'
422                 display += ' '*7 + 'Tipo de nodo: {}\n'.format(self.
    graph.nodes()[node]['node_type'])
423                 display += ' '*7 + 'Activaci[U+FFFD]n: {}\n'.format(self.
    graph.nodes()[node]['activation'])
424             title = 'Informaci[U+FFFD]n de ' + node_name
```

```
425         except:
426             pass
427         return display, title
428 
429     def define_app(self):
430         self.app.layout = html.Div([
431             dcc.Location(id='url', refresh=False),
432             html.Div(id='page-content')
433         ])
434 
435         self.app.callback(
436             Output('nodes-dropdown1', 'options'),
437             Output('nodes-dropdown2', 'options'),
438             Output('graph-with-nodes1', 'figure'),
439             Output('graph-with-nodes2', 'figure'),
440             Output('perceptions1', 'options'),
441             Output('perceptions2', 'options'),
442             [Input('interval-nodes', 'n_intervals'),
443             Input('nodes-dropdown1', 'value'),
444             Input('nodes-dropdown2', 'value'),
445             Input('perceptions1', 'value'),
446             Input('perceptions2', 'value'),
447             Input('radio_type', 'value')]
448             )(self.update_fig_nodes)
449 
450         self.app.callback(
451             Output('cytoscape-ltm', 'elements'),
452             [Input('interval-ltm', 'n_intervals')]
453             )(self.update_fig_ltm)
454 
455         self.app.callback(
456             Output('pause', 'children'),
457             Output('interval-ltm', 'interval'),
458             [Input('pause', 'n_clicks')]
459             )(self.update_pause)
460         self.app.callback(
461             Output("cytoscape-ltm", "generateImage"),
462             [Input('save', 'n_clicks')]
463             )(self.save_graph)
464 
465         self.app.callback(
466             Output('click-data', 'children'),
467             Output('click_mk', 'children'),
468             [Input('cytoscape-ltm', 'tapNodeData')]
469         )(self.display_info)
470 
471         self.app.callback(
472             Output('hover-data', 'children'),
473             Output('hover_mk', 'children'),
474             [Input('cytoscape-ltm', 'mouseoverNodeData')]
475         )(self.display_info)
476 
477         self.app.callback(
478             Output('page-content', 'children'),
```

```
479             [Input('url', 'pathname')]
480         )(self.display_page)
481 
482     def update_pause(self, n_clicks):
483         self.paused = not(self.paused)
484         if(self.paused):
485             self.control_publisher.publish(command = "pause", world =
    "", reward = False)
486             rospy.loginfo('Se ha pausado el proceso')
487             return 'Reanudar', 1*1000*60*60
488         else:
489             rospy.loginfo('Se ha reanudado el proceso')
490             self.control_publisher.publish(command = "continue", world
     = "", reward = False)
491             return 'Pausar', 1000
492 
493     def save_graph(self, n_clicks2):
494         ftype = 'png'
495         filename = 'ltm_' + str(n_clicks2)
496         action = 'download'
497 
498         nx.write_edgelist(self.graph, self.save_dir + 'graph_{}.txt'.
    format(n_clicks2))
499         nx.write_gexf(self.graph, self.save_dir + 'graph_{}.gexf'.
    format(n_clicks2))
500         rospy.loginfo('Se ha guardado el archivo graph_{}.txt'.format(
    n_clicks2))
501         if(n_clicks2 != None):
502             return {
503                     'type': ftype,
504                     'action' : action,
505                     'filename': filename
506                 }
507         else:
508             return {}
509 
510     def display_page(self, pathname):
511         if pathname == '/ltm':
512             return self.ltm_layout
513         elif pathname == '/nodes':
514             return self.node_layout
515         else:
516             return self.index_page
517 
518     ######ROS NODE CONFIGURATION FUCNTIONS#####
519     '''These functions are responsible of setting up the ROS node
    for visualization'''
520     @staticmethod
521     def class_from_classname(class_name):
522         """Return a class object from a class name."""
523         module_string, _, class_string = class_name.rpartition(".")
524         if sys.version_info < (3, 0):
525             # node_module = __import__(module_string, fromlist=[bytes(
    class_string, "utf-8")])
```

```
527             node_module = __import__(module_string, fromlist=[
    class_string])
528         else:
529             node_module = __import__(module_string, fromlist=[
    class_string])
530         # node_module = importlib.import_module('.' + class_string,
    package=module_string)
531         node_class = getattr(node_module, class_string)
532         return node_class
533 
534     def setup_control_channel(self, control_channel):
535         """Load simulator / robot configuration channel."""
536         topic = rospy.get_param(control_channel["control_prefix"] + "
    _topic")
537         self.control_message = self.class_from_classname(rospy.
    get_param(control_channel["control_prefix"] + "_msg"))
538         self.control_publisher = rospy.Publisher(topic, self.
    control_message, latch=True, queue_size=0)
539         topic = rospy.get_param(control_channel["info_prefix"] + "
    _topic")
540         message = self.class_from_classname(rospy.get_param(
    control_channel["info_prefix"] + "_msg"))
541         rospy.Subscriber(topic, message, callback=self.info_callback)
542 
543     def setup_topics(self, connectors):
544         """Load topic configuration for adding nodes."""
545         for connector in connectors:
546             self.default_class[connector["data"]] = connector.get("
    default_class")
547             self.default_ros_node_prefix[connector["data"]] =
    connector.get("ros_node_prefix")
548             self.default_ros_data_prefix[connector["data"]] =
    connector.get("ros_data_prefix")
549             if self.default_ros_node_prefix[connector["data"]] is not
    None:
550                 topic = rospy.get_param(connector["ros_node_prefix"] +
     "_topic")
551                 message = self.class_from_classname(rospy.get_param(
    connector["ros_node_prefix"] + "_msg"))
552                 callback = getattr(self, connector["callback"])
553                 callback_args = connector["data"]
554                 rospy.logdebug("Subscribing to %s...", topic)
555                 rospy.Subscriber(topic, message, callback=callback,
    callback_args=callback_args)
556         topic = '/mdb/ltm/p_node_update'
557         message = self.class_from_classname(rospy.get_param('/mdb/
    p_node_update_msg'))
558         callback = self.add_point_callback
559         rospy.logdebug("Subscribing to %s...", topic)
560         rospy.Subscriber(topic, message, callback=callback)
561 
562     def setup(self, log_level, file_name):
563         """Init VIEW: read ROS parameters, init ROS subscribers and
    load initial nodes."""
```

```
564         if file_name is None:
565             rospy.logerr("No configuration file specified!")
566         else:
567             if not os.path.isfile(file_name):
568                 rospy.logerr(file_name + " does not exist!")
569             else:
570                 rospy.init_node("mdb_view", log_level = getattr(rospy,
     log_level))
571                 rospy.loginfo("Loading configuration from %s...",
    file_name)
572                 configuration = yaml.load(open(file_name, "r"), Loader
    =yamlloader.ordereddict.CLoader)
573                 self.setup_topics(configuration["LTM"]["Connectors"])
574                 self.setup_control_channel(configuration["Control"])
575                 rospy.on_shutdown(self.shutdown)
576 
577     def run(self, seed = None, log_level = "INFO", save_dir = None):
578         """Start the visualization part"""
579         try:
580             self.define_app()
581             self.setup(log_level, seed)
582             self.save_dir = save_dir
583             self.ask_for_ltm()
584             self.app.run_server(use_reloader = False, debug = False)
585             rospy.spin()
586         except rospy.ROSInterruptException:
587             rospy.logerr("Exception caught! Or you pressed CTRL+C or
    something went wrong...")
588 
589     def shutdown(self):
590         rospy.loginfo('Closing port 8050')
591         os.system('kill -9 $( lsof -i:8050 -t )')
592 
593     ######ROS NODE CALLBACK FUCNTIONS#####
594     '''These functions are executed when a new message is
595     published in a topic to which the node is subscribed
596     and for sending commands to the control topic'''
597     def graph_set_edge_properties(self, node, node_type, neighbor_ids,
     neighbor_types):
598         for neighbor, neighbor_node_type in zip(neighbor_ids,
    neighbor_types):
599             if not self.graph.has_edge(node, neighbor):
600                 if neighbor_node_type == "CNode" or node_type == "
    CNode":
601                     width = 1
602                     if neighbor_node_type == "PNode" or node_type == "
    PNode":
603                         color = "purple"
604                     elif neighbor_node_type == "Goal" or node_type ==
    "Goal":
605                         color = "green"
606                     elif neighbor_node_type == "ForwardModel" or
    node_type == "ForwardModel":
607                         color = "blue"
```

```
608                     elif neighbor_node_type == "Policy" or node_type
    == "Policy":
609                         color = "red"
610                 else:
611                     color = "black"
612                     width = 0.2
613                 self.graph.add_edge(node, neighbor, color = color,
    width = width)
614                 # rospy.loginfo('Adding connection between {} and {}'.
    format(node, neighbor))
615 
616     def add_node_callback(self, data, node_type):
617         ident = data.id
618         if (data.command == "new"):
619             idx = self.nodes_config[node_type]
620             if(node_type == "PNode"):
621                 posx = idx // 3
622                 posy = idx % 3
623                 posx = 1 + posx + posy / 3.0
624                 color = 'purple'
625             elif(node_type == "Goal"):
626                 posx, posy = 3*idx, 4
627                 color = 'green'
628             elif(node_type == "ForwardModel"):
629                 posx, posy = 3 * idx, 6
630                 color = 'orange'
631             elif(node_type == "Policy"):
632                 posx, posy = 13, idx + 3
633                 color = 'red'
634             elif(node_type == "CNode"):
635                 posx = idx // 3
636                 posy = idx % 3
637                 posx, posy = 1 + posx + posy / 3.0, posy + 8
638                 color = 'blue'
639             elif(node_type == "Perception"):
640                 posx, posy = -3, idx + 3
641                 color = 'grey'
642             self.graph.add_node(ident,
643                                 id = ident,
644                                 position =  {'x': posx*75, 'y': posy
    *75},
645                                 node_type = node_type,
646                                 color = color,
647                                 activation = data.activation
648                                 )
649             self.nodes_config[node_type] += 1
650             self.graph_set_edge_properties(ident, node_type, data.
    neighbor_ids, data.neighbor_types)
651             rospy.logdebug("MENSAJE: Creating node {}".format(ident))
652 
653         elif(data.command == "update"):
654             if node_type == "PNode":
655                 columns = data.names
656                 columns.append("confidence")
```

```
657                 self.sens_names = columns
658                 if not ident in self.pnode_dict:
659                     self.pnode_dict[ident] = pd.DataFrame(columns =
    columns)
660                 else:
661                     self.pnode_dict[ident].columns = columns
662 
663             self.graph.nodes()[ident]["activation"] = data.activation
664             self.graph_set_edge_properties(ident, node_type, data.
    neighbor_ids, data.neighbor_types)
665             rospy.logdebug("MENSAJE: " + node_type + " activation for
    " + ident + " = " + str(data.activation))
666 
667     def add_point_callback(self, data):
668         new_data = list(data.point)
669         new_data.append(data.confidence)
670         if not data.id in self.pnode_dict:
671             self.pnode_dict[data.id] = pd.DataFrame(columns = ['']*len
    (new_data))
672         self.pnode_dict[data.id].loc[len(self.pnode_dict[data.id])] =
    new_data
673         rospy.logdebug("MENSAJE: Updating points of " + data.id)
674 
675     def info_callback(self, data):
676         # rospy.loginfo('Updating information from LTM')
677         self.iteration = data.iteration
678         self.current_world = data.current_world
679         self.current_policy = data.current_policy
680         self.current_reward = data.current_reward
681         self.current_goal = data.current_goal
682         rospy.logdebug("MENSAJE: Updated info of iteration: {}".format
    (self.iteration))
683         self.set_title()
684         rospy.loginfo("*** ITERATION: " + str(self.iteration) + " ***"
    )
685 
686     def ask_for_ltm(self):
687         rospy.logdebug('Asking for current nodes in LTM')
688         self.control_publisher.publish(command = "publish_ltm", world
    = "", reward = False)
```

Listing 8.11: Archivo view.py con la clase que permite implementar el nodo de visualización.