UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISEÑO E IMPLEMENTACIÓN DE MEMORIA DE LARGO PLAZO PARA ROBOTS DE SERVICIO

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELÉCTRICO E INGENIERO CIVIL EN COMPUTACIÓN

MATÍAS FERNANDO PAVEZ BAHAMONDES

PROFESOR GUÍA:
JAVIER RUIZ DEL SOLAR
PROFESOR GUÍA 2:
JOCELYN SIMMONDS WAGEMANN

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULÉ
ANDRÉS CABA RUTTE

SANTIAGO DE CHILE
2018

RESUMEN DE LA MEMORIA PARA OPTAR AL TÍTULO DE: INGENIERO CIVIL ELÉCTRICO E INGENIERO CIVIL EN COMPUTACIÓN
POR: MATÍAS FERNANDO PAVEZ BAHAMONDES
FECHA: 2018
PROF. GUÍA: JAVIER RUIZ DEL SOLAR
Y JOCELYN SIMMONDS WAGEMANN

DISEÑO E IMPLEMENTACIÓN DE MEMORIA DE LARGO PLAZO PARA ROBOTS DE SERVICIO

El objetivo de este trabajo de título es el diseño e implementación de un sistema de memoria episódica de largo plazo para robots de servicio domésticos. Además, el sistema debe considerar componentes emocionales y ser integrado en el software del robot Bender, perteneciente al Laboratorio de Robótica del Departamento de Ingeniería Eléctrica de la Universidad de Chile.

Un sistema de estas características es esencial para mejorar el desempeño de un robot doméstico, especialmente en el ámbito de la interacción humano-robot. Sin embargo, tras la revisión del estado del arte, se encuentran pocos trabajos relacionados y que aún no existe un consenso en el tema.

El sistema es diseñado para cumplir con un conjunto de requerimientos episódicos mínimos para este tipo de memorias. Además, el diseño es agnóstico del robot objetivo, permitiendo la definición de estructuras de datos genéricas para la representación episódica, las que son manejadas por un sistema de plugins. De esta forma, el sistema puede ser integrado en otras plataformas robóticas basadas en Robot Operating System (ROS).

El sistema implementado es capaz de recolectar episodios automáticamente desde las máquinas de estado que definen el comportamiento del robot. Utiliza la base de datos MongoDB para el almacenamiento de episodios y está programado en C++ y Python, solo utilizando paquetes estándar en ROS. Además, el sistema provee una API ROS capaz de responder consultas sobre estos episodios, las que permiten realizar búsquedas mediante combinaciones de condiciones lógicas.

La implementación es evaluada de manera cuantitativa, mediante experimentos de escalabilidad y eficiencia. Los resultados indican que el sistema se adapta al caso de uso esperado para el robot Bender. Sin embargo, la formulación de consultas al sistema tiene un alto impacto en su desempeño. Por esto, es importante seleccionar adecuadamente las operaciones, para que el uso de recursos no comprometa la interacción humano-robot. Desde otro enfoque, la integración del sistema LTM en Bender es validada mediante sesiones de demostración, donde el robot genera memorias sobre los humanos con quien interactúa.

Se concluye que el sistema cumple con la mayoría de los requerimientos establecidos, teniendo que acotar el proyecto para dejar algunos aspectos como trabajo futuro. Particularmente, queda propuesta la implementación de un sistema emocional para el robot.

A mi familia que me ha apoyado para crecer y convertirme en el que soy.

"Las preguntas que no podemos contestar son las que más nos enseñan. Nos enseñan a pensar. Si le das a alguien una respuesta, lo único que obtiene es cierta información. Pero si le das una pregunta, él buscará sus propias respuestas. Así, cuando encuentre las respuestas, las valorará más. Cuanto más difícil es la pregunta, más difícil la búsqueda. Cuanto más difícil es la búsqueda, más aprendemos. Una pregunta imposible . . . "

— Patrick Rothfuss, El Temor de un Hombre Sabio

Agradecimientos

En primer lugar, agradezco a mi familia: padres, hermanos, tíos y abuelos, que siempre me han apoyado en este largo camino. Especialmente, me gustaría agradecer a mis ninos, Pamela y Mario, por todo el cariño que me han brindado siempre, por el espacio donde me alojé en su casa durante mis primeros años de universidad y porque me siempre me han querido como un hijo más.

Además, debo agradecer a mis amigos, los que siempre han estado ahí en las buenas y en las malas. Por las horas y horas dedicadas al HOTS, en sacar adelante al equipo. Por los muchos años de amistad, muchas gracias Michi, Claudio y Carla.

Gracias a todos los miembros del equipo de robótica del DIE, con quienes he trabajado durante casi todos mis estudios, por todo el esfuerzo que le pusimos a nuestros robots. Gracias a quienes me guiaron en los primeros años de trabajo y a los conocí en el proceso. Gracias a Bender.

Debo agradecer a la profesora Jocelyn Simmonds por sus minuciosas correcciones a este extenso informe, sumado a sus consejos que me permitieron sacar adelante este trabajo de título. También, agradezco al profesor Javier Ruiz del Solar y el resto de la comisión por su tiempo y correcciones.

Finalmente, agradezco a mi pareja Luz, que me ha acompañado, ayudado y brindado su cariño durante estos últimos meses. Estamos iniciando una nueva etapa, con nuevas metas y desafíos, estoy feliz por poder compartirlos contigo. Muchas gracias lucy, te amo.

Tabla de Contenido

Índice de Tablas

Índice de Figuras

1 Introducción

En este capítulo se expone una introducción al trabajo de título a realizar: el diseño e implementación de una memoria de largo plazo episódica, semántica y emocional para el robot Bender, desarrollado en el laboratorio de robótica del Departamento de Ingeniería Eléctrica de la Universidad de Chile. Se da el contexto en el que se enmarca, la motivación para su desarrollo y los objetivos del trabajo. Se revisan brevemente conceptos sobre robótica doméstica, la memoria humana y el equipo de trabajo donde se implantará el sistema. Luego se explica el problema a tratar y la oportunidad de desarrollo. Finalmente se formalizan los objetivos del trabajo así como sus alcances.

1.1. Antecedentes Generales

A continuación, el lector encontrará una breve introducción a los temas requeridos para contextualizar este trabajo: La robótica de servicio doméstica y el equipo de trabajo donde se implantará la solución. Además, se introduce el tema de la memoria humana, requerido para entender el trabajo y su relación con la robótica. Estos temas serán tratados en mayor profundidad más adelante, en los Capítulos 2 y 3.

1.1.1. Robots de servicio domésticos

La robótica de servicio es un área enfocada en asistir a los seres humanos en tareas repetitivas y comunes. Para completar una tarea, el robot requiere cierto grado de autonomía, que le permita actuar en ambientes no controlados y utilizando sus sensores para responder correctamente a los cambios del entorno [1].

A grandes rasgos, los robots de servicio se categorizan en robots para el transporte, seguridad y domésticos. A su vez, los robots de servicio domésticos se caracterizan por realizar tareas de asistencia en el hogar y de compañía para humanos. Algunas de las tareas típicas que deben realizar son ayudar a ordenar, preparar comida u ofrecer bebestibles. Algunos se

enfocan en el cuidado de adultos mayores, en mascotas de compañía, salud o educación.

1.1.2. Liga RoboCup@Home

RoboCup es una competencia internacional cuyo objetivo es ser un vehículo para el desarrollo de la robótica y la inteligencia artificial. Está compuesta de variadas ligas: Rescue, Soccer, Simulation, @Home, Industrial y Junior, cada una con diversas subligas orientadas a fomentar la investigación de distintos aspectos del campo. El sueño de sus organizadores, es que para mediados del siglo 21 e incorporando los desarrollos de todas las ligas, un equipo de fútbol robótico completamente autónomo sea capaz de vencer al campeón de la última copa mundial, siguiendo las reglas de la FIFA [2].

Las pruebas de la liga @Home se desarrollan en escenarios que imitan ambientes domésticos reales, como un hogar o un restaurante. Las capacidades generalmente evaluadas y potenciadas son de visión computacional, navegación autónoma, manipulación de objetos y reconocimiento de voz. Sin embargo, constantemente se revisan nuevas capacidades deseables en un robot doméstico. Por otro lado, la competencia funciona como un espectáculo para público general, por lo que se priorizan pruebas y demostraciones interesantes para los espectadores.

1.1.3. Equipo de trabajo: UChile Homebreakers

El Laboratorio de Robótica del Departamento de Ingeniería Eléctrica de la Universidad de Chile alberga dos equipos de robótica: *UChile Robotics Team*, dedicado al fútbol robótico y *UChile Homebreakers Team*, enfocado en robótica de servicio. Ambos son conformados por alumnos de pregrado y postgrado de diversas especialidades, y liderados por el profesor Javier Ruiz del Solar [3]. UChile Homebreakers existe desde el año 2007 y actualmente cuenta con alrededor de 12 miembros. Desde sus inicios, el equipo participa en la competencia RoboCup@Home.

El equipo trabaja en dos plataformas humanoides de tipo doméstico, Bender y Maqui. Bender es un robot construido en el mismo laboratorio y con el objetivo de ser un mayordomo para el hogar. Maqui, desarrollado por SoftBank Robotics [4], está diseñado para ser un robot de compañía. Ambos basan su desarrollo de software en ROS, un framework para facilitar la comunicación entre componentes robóticos, y con miles de usuarios alrededor del mundo [5]. Ambos robots comparten la misma arquitectura de software y prácticamente todo su código, exceptuando los *drivers* para acceder al hardware respectivo.

1.1.4. La memoria humana

Según Eichenbaum [6], la memoria hace relación al almacenamiento de experiencias en el cerebro, mediante múltiples sistemas de memoria independientes y sustentados por distintas estructuras cerebrales. A grandes rasgos, la memoria se puede dividir en de corto plazo

"STM" (Short-Term Memory) y de largo plazo "LTM" (Long-Term Memory). La STM maneja información muy detallada, es de poca capacidad y permite un rápido acceso, mientras que la LTM maneja información sobre experiencias y entidades, es menos detallada, de mayor capacidad y de acceso más lento.

Eichenbaum divide la LTM en una componente explícita (consciente) y una implícita (inconsciente). La parte explícita se subdivide en 2 categorías: memoria episódica y memoria semántica. La primera se encarga de almacenar episodios, para responder a las preguntas "Qué sucedió", "Dónde " y "Cuándo". La Memoria semántica almacena hechos y conceptos, como el lenguaje o personas y sus características. Además, la memoria explícita almacena las conexiones entre la información semántica y los episodios relacionados. Por otro lado, la memoria implícita codifica habilidades, hábitos y preferencias.

La memoria emocional es una forma de memoria implícita que genera reacciones emocionales y sentimientos. Según los estímulos a los que se enfrente, permite modular el proceso de consolidación de la STM en LTM, modificando el nivel de relevancia de los eventos, pudiendo generar memorias muy fuertes y hábitos arraigados. Ejemplos de esto son los flashbacks y las memorias asociadas a eventos importantes.

1.2. Motivación

La LTM es una habilidad cognitiva esencial para que un humano sienta continuidad en su vida. Al interactuar con otras personas o el ambiente les permite recordar experiencias pasadas y sus detalles. Luego, es de esperar que un robot doméstico posea una memoria que le permita potenciar sus capacidades de interacción con los humanos que ayudará [7]. Una LTM permitiría, por ejemplo, generar diálogos interesantes sobre eventos pasados o inferir aspectos del comportamiento humano, por otro lado, también permitiría la generalización de las tareas que tiene que llevar a cabo.

Durante la última década han habido algunos intentos de implementar LTM en robots domésticos, sin embargo, no existe una solución estándar y aún quedan muchas consultas sin responder [8].

La LTM, al ser una capacidad esencial para robots domésticos, es esperable que sea agregada en el corto plazo como uno de los requisitos para RoboCup@Home. Más aún, dado el enfoque de las plataformas disponibles, Bender cómo robot mayordomo y Maqui cómo robot social, se espera que ambos posean capacidades avanzadas de interacción con los humanos, para lo que se requiere este tipo de memoria.

1.2.1. Un primer acercamiento

El año 2015 se desarrolló una LTM episódica para el robot Bender, orientada a la interacción con personas y objetos [9]. El trabajo consideraba métodos para almacenar, adquirir y manejar la información episódica, sumado a un proceso simple de consolidación de memoria.

Actualmente la memoria desarrollada no está operativa, ni es factible habilitarla. Se identificaron 4 causas del problema, de carácter técnico y humano:

Integración incompleta La memoria fue integrada parcialmente al software del robot, solo con motivos de ejemplificar su funcionamiento. Por un lado, la rutina implementada requiere una pauta específica de interacción con las personas, solo considerando la primera interacción con cada entidad, por lo que no existe actualización de los datos. Por otro lado, su desarrollo no consideró una integración con las rutinas de comportamiento utilizadas por el robot, y su estudio no indica cómo puede ser adaptado a tales rutinas de manera no intrusiva. Entonces, en la práctica no se recopila ni provee información continuamente mientras el robot está en funcionamiento.

Además, la implementación del sistema LTM no provee una API ROS para acceder a sus funcionalidades, según el estándar de los desarrollos del equipo.

Pocas entidades El modelo de datos está definido de forma estática y no queda claro como puede ser modificado para manejar nuevas entidades. Particularmente, la integración solo considera 2 entidades: Persona y Objeto, para los cuales solo se almacena información de nombre, nacionalidad e imagen. Es esperable que una memoria considere más entidades (como lugares, robots, animales u objetos) y más características para cada uno de estas (como nombre, hobbies, trabajo o edad, para el caso específico de un humano).

A pesar de considerar una entidad para objetos, en la práctica no se integró con los módulos relacionados que recopilan la información, por lo que realmente la memoria solo funciona para entidades de tipo Persona.

Eficiencia y escalabilidad El trabajo es una prueba de concepto, por lo que solo presenta una evaluación subjetiva del sistema, a partir de un conjunto de preguntas predefinidas: el robot hace 5 consultas al usuario, para luego responder 5 consultas hechas por el usuario. Tras la interacción, el usuario responde una encuesta sobre la calidad percibida de la interacción. Por lo tanto, no existen pruebas de escalabilidad ni eficiencia de la implementación.

Prioridades de UChile Homebreakers Cada año el equipo planifica sus desarrollos de acuerdo a los requerimientos de la liga @Home, por lo que todo trabajo fuera de las áreas evaluadas no son considerados una prioridad. Luego, ya que las funcionalidades LTM no son evaluadas en la competencia, no existe un incentivo para seguir desarrollando, finalizar la integración ni mantenerla.

Los primeros 3 puntos son un indicador de que el diseño considerado no se ajusta a los requerimientos mínimos para un sistema LTM con memoria episódica, como los propuestos por Stachowicz [10] (éstos requerimientos son extensos y se describen en detalle en la Sección 2.4.1). Particularmente, no cumple con los siguientes requisitos: las estructuras semánticas deben poder ser definidas por el usuario, la implementación debe ser no intrusiva, eficiente en el uso de recursos del robot, y escalable para soportar el almacenamiento de miles de episodios, sin afectar los tiempos de procesamiento.

En resumen, de acuerdo a los problemas identificados, se considera que el sistema implementado es incompleto y no se ajusta a los requerimientos reales de un sistema LTM. Más importante aún, se cree el trabajo no se ajustaba a las necesidades del equipo UChile Homebreakers, pues no existió una integración que considerara las rutinas de comportamiento del robot. Por lo anterior, el sistema LTM no es utilizado y su código ha quedado obsoleto debido a la falta de mantención.

1.2.2. Oportunidad

Existe un vasto desarrollo respecto a la memoria y los procesos cognitivos, sin embargo, la investigación se concentra en campos como psicología, neurología y ciencias cognitivas. Los estudios de LTM para robots de servicio son muy acotados y no existe una solución estándar a implementar [8]. Algunos robots, como la versión comercial de Maqui, proveen aplicaciones con características de un sistema LTM, pero el código asociado no es libre, ni está basado en ROS.

El uso de LTM no está en las prioridades del equipo, sino que es algo útil para demostraciones y para potenciar la interacción humano-robot. Por ello, se considera que no basta con desarrollar un módulo capaz de recopilar información inteligentemente, sino que es necesario atacar los 4 problemas identificados del sistema LTM implementado por Sánchez et al.

1.3. Objetivos del Trabajo de Título

En esta sección se presentan los objetivos del trabajo de título, a modo general y en un desglose por aspectos específicos requeridos. Luego se describen los alcances y la contribución del trabajo.

1.3.1. Objetivo general

El objetivo general es el diseño e implementación de una LTM para robots de servicio domésticos, que considere componentes episódicos, semánticos y emocionales. La LTM debe ser integrada en Bender, recopilando recuerdos constantemente y con una API acorde a los desarrollos de UChile Homebreakers. Además, se debe proveer una demostración de las funcionalidades introducidas.

1.3.2. Objetivos específicos

A continuación se presentan los objetivos específicos del trabajo de título, derivados a partir del objetivo general, y que representan los requerimientos clave del trabajo.

- Definir un conjunto de requisitos para el diseño del sistema y las validaciones para verificar el cumplimiento de tales requisitos.
- Diseñar un sistema LTM para almacenamiento de memoria episódica, semántica y emocional, enfocada en robots domésticos.
- Diseñar el sistema LTM de manera agnóstica al robot objetivo y proveer una interfaz ROS a sus funcionalidades.
- Implementar el sistema LTM e integrarlo en el robot Bender.
- Implementar una demostración del sistema LTM en funcionamiento.

1.3.3. Alcances y contribución del trabajo

En primer lugar, se espera que este trabajo de título sirva como base para la implementación de sistemas LTM y funcionalidades derivadas en robots domésticos. Por lo tanto, el foco del trabajo es proveer un diseño robusto, basado en los objetivos propuestos, sumado a una implementación inicial que soporte el diseño.

Este trabajo provee un sistema LTM que considera la memoria emocional en su diseño. Sin embargo, Bender no dispone de módulos emocionales que puedan ser utilizados como fuente de datos para el sistema. Se considera que el desarrollo de tales módulos es un trabajo extenso y se escapa de los objetivos expuestos. Considerando esto, el sistema LTM provee una interfaz emocional, en caso de que a futuro se implementen módulos emocionales en Bender.

Existen otras cuestiones deseables de tratar en una memoria LTM, pero que no son abordados en este trabajo. Dos ejemplos son los aspectos éticos de almacenar información sensible de usuarios, y el cómo compartir la memoria con otros robots. Estos se describen en detalle en el Capítulo 2.

1.4. Estructura de la Memoria

En este primer capítulo se hizo una descripción breve de los conceptos necesarios para introducir el trabajo, su motivación y sus objetivos. El Capítulo 2 hace una revisión de los aspectos teóricos necesarios para comprender el trabajo realizado, y el Capítulo 3 describe los componentes de software y hardware requeridos. Luego, el Capítulo 4 presenta el diseño del sistema LTM, a partir de sus requisitos y validaciones, mientras que en el Capítulo 5 se describe su implementación y funcionamiento. En el Capítulo 6 se muestran los resultados de las validaciones y su análisis. Finalmente, se presentan las conclusiones y el trabajo futuro propuesto.

2 Marco Teórico

En este capítulo se revisan los temas y conceptos teóricos relevantes para el desarrollo del trabajo. Se formaliza la definición de robot doméstico y sus alcances. Se describe la memoria humana, sus categorías, funcionamiento y procesos cerebrales relevantes. Basado en los temas anteriores, se revisa la relación entre robótica y la memoria humana: se describen algunos enfoques existentes y las reglas generales para su implementación.

2.1. Robots de Servicio Domésticos

La Federación Internacional de Robótica (IFR) [1] define *robot* como:

> "Un mecanismo actuado y programable en dos o más ejes y con un cierto grado de autonomía, que se mueve en su entorno para realizar tareas previstas. En este contexto, autonomía se refiere a la habilidad de realizar tareas previstas, basado en el estado actual y lo sensado, sin intervención humana."

Asimismo, la IFR define un *robot de servicio* como un robot "que realiza tareas útiles para humanos o equipamiento, excluyendo aplicaciones de automatización industrial". Así, un robot de servicio debe trabajar en ambientes no controlados y con la autonomía suficiente que le permita llevar a cabo su cometido. Generalmente, la robótica de servicio se enfoca en asistir a los seres humanos en tareas repetitivas y comunes.

Según su área de aplicación, un robot de servicio se clasifica en *de uso personal* o *de uso profesional*. Los primeros son utilizados en ambientes no comerciales y por personas comunes; como por ejemplo, un robot sirviente o una silla de ruedas autónoma. Un robot de servicio profesional se utiliza en ambientes comerciales, usualmente operados por alguien entrenado; un ejemplo son los robots de entrega de paquetes o para cirugía.

Según la recopilación de datos realizada por la IFR durante el 2016, este tipo de robots es utilizado en diversas áreas para tareas domésticas, entretenimiento, educación, investigación, asistencia a ancianos y discapacitados, transporte, seguridad y vigilancia. Finalmente, existen

otros robots de servicio que no pueden ser clasificados en las categorías anteriores.

El foco de este trabajo son los robots de servicio personales, dedicados a tareas domésticas, clasificación a la que en adelante se referirá como *robots domésticos*.

2.2. Memoria Humana

La memoria es un elemento fundamental para los humanos en su día a día, es parte integral de su existencia. Permite recordar quién, qué, cómo, dónde y cuándo. En términos psicológicos, es la habilidad para codificar, almacenar y luego obtener información sobre eventos pasados, en el cerebro. Los pensamientos son parte de la memoria de corto plazo, mientras que eventos pasados son almacenados en una memoria de largo plazo. Existen muchos estudios en el área de la psicología cognitiva con diversas descripciones y modelos teóricos de cada tipo de memoria [7].

Desde el punto de vista de la información procesada, la memoria es vista como una facultad humana consistente en procesos para el manejo de información. Los 3 componentes principales son:

- **Codificación**: En este paso, se adquiere nueva información desde los sentidos humanos. Los datos son convertidos a un formato que pueda ser almacenado en la estructura cerebral correspondiente.

- **Almacenamiento**: Consiste en la creación de registros permanentes de información. Es un proceso pasivo, de continuo procesamiento para clasificar datos nuevos y los ya existentes en el cerebro.

- **Adquisición**: Hace referencia al acceso de datos almacenados. El proceso se realiza en respuesta a una pista, que permita obtener una reconstrucción aproximada de la información, a partir de elementos repartidos en distintas partes del cerebro.

La memoria puede ser dividida en múltiples sistemas independientes, con funcionalidades bien definidas y sustentados por distintas estructuras cerebrales [11]. La primera diferenciación define dos tipos de memoria: la de corto y la de largo plazo, STM (Short-Term Memory) y LTM (Long-Term Memory), por sus siglas en inglés [12]. En el diagrama de la Figura 2.1 se muestra una separación clásica utilizada en el área de las ciencias cognitivas [6], explicada en las siguientes subsecciones.

2.2.1. Memoria de corto plazo (STM)

En el ámbito cognitivo, la STM se refiere a la habilidad de estar atento, recopilar información y memorias, para luego utilizarlas dentro de un corto periodo de tiempo [6]. Es responsable de almacenar información constantemente y de decidir que parte será transferida a la LTM. El término de *Memoria de Trabajo* suele ser utilizado de manera intercambiable

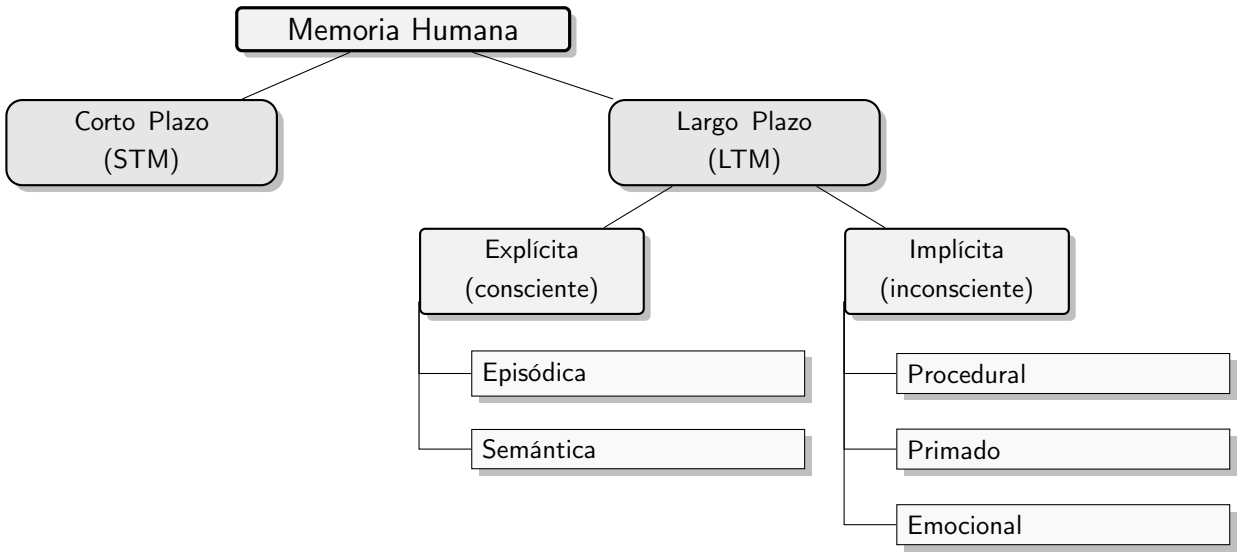


Figura 2.1: Clasificaciones de la memoria humana. Adaptado de [7].

con el de STM.

Se caracteriza por manejar información muy detallada, ser de poca capacidad y permitir un rápido acceso a estos datos. Permite recordar rápidamente y con gran detalle experiencias ocurridas hace pocos segundos, pero con dificultad creciente a medida que avanza el tiempo.

Se sustenta principalmente en la corteza prefrontal del cerebro. Las neuronas involucradas son capaces de mantener información relevante de corto plazo, la que es combinada con información sensorial entrante y áreas que manejan la toma de decisiones [6].

En los humanos esta área presenta gran activación durante procesos de codificación, acceso y manipulación de memorias.

2.2.2. Memoria de largo plazo (LTM)

La LTM se asocia al almacenamiento permanente de información en el cerebro. Se caracteriza por manejar una gran cantidad de experiencias y entidades, ser menos detallada y proveer un acceso más lento a los recuerdos, respecto a la STM [6]. Cierta información de la STM eventualmente es transferida a la LTM. De acuerdo a la Figura 2.1, sus dos principales categorías son la *Memoria Implícita* y la *Memoria Explícita*.

2.2.2.1. LTM explícita

La memoria explícita suele ser denominada *memoria consciente* o *memoria declarativa*, pues maneja conocimientos relacionados a hechos y eventos adquiridos de forma consciente. Según las estructuras cerebrales involucradas, se conforma de la *memoria episódica* y de la *memoria semántica* [6].

La memoria episódica, por primera vez definida por Tulving [13], es de carácter autobiográfico y almacena detalles de eventos y experiencias pasadas. Permite responder a las preguntas "Qué sucedió", "Dónde ocurrió" y "Cuándo ocurrió" [14]. Particularmente, se utiliza la noción de memoria episódica descrita por Clayton y Russel [15], que incorpora el concepto de perspectiva episódica de los recuerdos. Un humano puede acceder a esta memoria si es capaz de decir: "recuerdo que". Este tipo de memoria da al ser humano la sensación de continuidad en el tiempo.

La memoria semántica almacena el conocimiento de hechos, significados, categorías y proposiciones. Un humano puede acceder a esta memoria si es capaz de decir: "sé que". Esta memoria se abstrae de perspectiva e información situacional.

Las estructuras cerebrales que soportan la memoria explícita son el hipocampo, encargado de manejar la memoria episódica, junto a la corteza cerebral, en donde se distribuyen los conocimientos de la memoria semántica. En el hipocampo se mantienen conexiones neuronales a los sectores de interés de la corteza, en donde se alojan conocimientos semánticos asociados a cada episodio.

Un ejemplo de uso de memoria episódica es el recuerdo de una graduación escolar, el lugar y la fecha donde ocurrió. La memoria semántica podría responder en que consiste una graduación y describir la ropa que se suele ocupar en ellas.

2.2.2.2. LTM implícita

La memoria implícita abarca la capacidad de aprender habilidades, hábitos y preferencias, caracterizados por ser mejorados o adquiridos sin una recolección consciente. Así, también suele ser denominada *memoria inconsciente* o *memoria no declarativa*, pues comprende acciones que pueden ser realizadas sin pensar en ellas. Ejemplos de esto, son el andar en bicicleta o tocar un instrumento musical [6].

Dos de sus componentes son la *memoria procedural* y la *memoria de primado*. La primera ayuda a realizar tareas sin pensar en ellas, es decir, maneja el conocimiento del *Cómo*; Ejemplos de esto son comer y caminar. La memoria de primado hace referencia a la predisposición para recordar hechos o información a la que un sujeto es expuesto con anterioridad; Ejemplos de esto son la facilidad para recordar canciones escuchadas hace poco tiempo, o el uso de palabras e ideas vistas recientemente.

Se ha mostrado que la memoria procedural se sustenta en el cerebelo, mediante la activación de este durante el uso de habilidades motoras.

Un tercer componente de la memoria implícita es la *memoria emocional*. Esta se encarga de dar significado afectivo a ciertos estímulos, que de otra forma serían neutrales. Las estructuras cerebrales involucradas son la amígdala, las áreas corticales y subcorticales. Esta memoria se expresa mediante la activación del hipotálamo, en conjunto al sistema nervioso simpático, generando reacciones emocionales y sentimientos [16].

2.2.3. Plasticidad sináptica y modulación

Se denomina *consolidación* de memoria al proceso de transición de conocimiento desde la STM a la LTM [17]. Durante la consolidación se generan conexiones neuronales entre la memoria episódica y la respectiva zona semántica. Para activar la consolidación se requiere de un estímulo relevante, sumado a la cadena de eventos para el almacenamiento [6].

Se denomina *deterioro* de memoria u "olvido" al proceso de debilitamiento de las conexiones neuronales establecidas por los procesos de consolidación. Está en constante funcionamiento, degenerando las asociaciones entre la memoria episódica y la semántica. Por lo tanto, en este contexto, el olvido no significa una eliminación de los datos en el cerebro, sino que estos siguen ahí, pero la conexión requerida es inexistente o es demasiado débil para poder utilizarla.

Existen procesos químicos a nivel cerebral que afectan la consolidación y el deterioro de la LTM. Hay evidencia de que estos están en continuo funcionamiento. Estos eventos celulares ocurren en una escala de segundos a minutos, y son esenciales para la mantención de la LTM.

Es posible modular ambos procesos. Las experiencias repetidas potencian la consolidación de la memoria, lo que fortalece las conexiones neuronales. Por otro lado, la memoria emocional es capaz de potenciar o deprimir las reacciones químicas requeridas; según los estímulos a los que se enfrente, modifica el nivel de relevancia de los eventos, pudiendo generar memorias muy fuertes y hábitos arraigados. Ejemplos de esto, son la memorización por repetición, los flashbacks y las memorias asociadas a eventos importantes como cumpleaños.

2.3. Memoria y Robótica

En esta sección se revisan trabajos que buscan implementar LTM en robots. Se inicia haciendo énfasis en la importancia de esta en el campo de la robótica doméstica. Se hace una comparación entre la memoria humana y el manejo de información en robots. Luego, se revisan sistemas LTM encontrados en la literatura. Finalmente, se detallan las bases teóricas utilizadas para el diseño del trabajo, seleccionadas a partir de los estudios más relevantes revisados.

2.3.1. Relevancia de la LTM en robótica

La LTM es una habilidad cognitiva esencial para cualquier ser social, otorgando una sensación de continuidad a la vida [7]. Durante una interacción social, permite recordar experiencias pasadas y relacionarlas con información actual, lo que genera interacciones interesantes y no monótonas. Lo mismo se aplica al caso de un robot doméstico, donde es esperable que posea una memoria que le permita establecer una relación de largo plazo con humanos, y potenciar sus capacidades de interacción con ellos.

Para los desarrolladores, un problema común es que los usuarios tienden a perder el interés

rápidamente en sus robots, pues existen expectativas no cumplidas respecto a la inteligencia y capacidad de socializar de la máquina [18]. El problema se potencia con el paso del tiempo, donde la motivación por interactuar disminuye y se genera frustración, a medida que el robot continua repitiendo los mismos comportamientos predefinidos. De manera similar para un humano, cuando la LTM no funciona correctamente debido a una enfermedad (por ejemplo, Alzheimer), la habilidad de interacción con otros humanos se ve dañada severamente [8].

Si se desea mejorar la interacción humano-robot, entonces se requiere que el robot se comporte de manera más natural. Los mejores agentes robóticos sociales deberían satisfacer las necesidades cognitivas y sociales humanas; mientras más familiar sea la interacción, serán más efectivos en su propósito. Así, la LTM es una habilidad crucial si se espera que el robot sea capaz de aprender y adaptarse a su entorno. Una LTM permitiría, por ejemplo, generar diálogos interesantes sobre eventos pasados o inferir aspectos del comportamiento humano, por otro lado, también permitiría la generalización de las tareas que tiene que llevar a cabo.

Durante la última década han habido algunos intentos de implementar LTM en robots domésticos, sin embargo, no existe una solución estándar y aún quedan muchas consultas sin responder [8]. Por ejemplo, no existe una estrategia clara para el almacenamiento de eventos y experiencias en la forma de episodios en una memoria LTM. Tampoco está claro como almacenar el estado emocional del robot y utilizarlo para modulación de la relevancia episódica. Otros puntos en investigación son la forma de consolidar la STM en LTM, la implementación de mecanismos de olvido y represión, y como manejar los aspectos éticos de adquirir y almacenar información personal de usuarios.

Por otro lado, desde un punto de vista práctico, se ha mostrado que el concepto de memoria LTM aplicada a robots es beneficioso. Salgado et al. [19] ocupan memoria procedural para mejorar el desempeño de un robot en ambientes dinámicos, logrando acelerar el proceso de adaptación al entorno y la toma de decisiones.

2.3.2. Revisión de sistemas LTM

A continuación se presenta una revisión de sistemas LTM basados en la memoria humana. En primer lugar se describen otras implementaciones de estos sistemas. Luego, se revisan trabajos con aspectos de interés para un sistema LTM. Para concluir, se muestran algunos estudios enfocados en aspectos interesantes, pero que escapan de los alcances de este trabajo.

2.3.2.1. Frameworks LTM

SOAR-EM (2004): Nuxoll y Laird presentaron SOAR-EM [20], una extensión con memoria episódica para la arquitectura SOAR [21]. Ellos desarrollaron un sistema similar a PACMAN, donde la máquina debe moverse a través de una grilla para buscar "comida" lo más rápido posible. Utilizaron la memoria episódica para dar apoyo en la planificación de los movimientos del agente, mostrando que aquellos equipados con memoria episódica tienen un mejor desempeño que los que carecen de ella.

En SOAR-EM un episodio es creado con datos de la STM por cada acción del agente, los que pueden ser recolectados a partir de una pista episódica. Sin embargo, más adelante se mostró que el mecanismo de recolección es ineficiente y difícil de optimizar [22]. Por otro lado, el trabajo no considera aspectos como el olvido episódico, ni la interacción de emociones con la LTM.

ISAC (2005): En el año 2005 Ratanaswasd et al. [23] agregan memoria episódica al robot humanoide ISAC (Intelligent Soft Arm Control), enfocado en la manipulación de objetos. Los episodios son almacenados por su fecha y hora, combinados con todos los contenidos semánticos que aparecen en ese lapso. Luego, estos son recolectados para soportar la toma de decisiones en la planificación de trayectorias robóticas, bajo la suposición de que siempre existirá un episodio lo suficientemente similar a la situación actual. Más tarde, Dodd y Gutierrez [24] incorporaron el concepto de relevancia emocional e histórica, con el objetivo de optimizar la búsqueda y eliminación de eventos según su intensidad. La relevancia emocional se obtiene directamente desde el sistema emocional de ISAC. Para la relevancia histórica, los eventos recientes son más importantes que los pasados, y experiencias novedosas tienen mayor puntaje que las comunes.

Los datos almacenados cubren el Qué, Cuándo y Dónde requeridos por Clayton [15]. Sin embargo, el diseño no provee la flexibilidad esperada en un sistema de memoria episódica: Delimita los episodios por cambios en los objetivos generales de planificación del robot. No soporta el traslape ni la anidación de eventos. No queda claro si es suficientemente eficiente o escalable para cumplir con los requerimientos de un sistema episódico [10].

EPIROME (2008): TASER es un robot de servicio enfocado a ambientes reales de oficina [25]. Jockel et al. [26] proponen el uso de memoria episódica para apoyar con tareas y evitar trabajo innecesario. Por ejemplo, al revisar oficinas para ayudar buscando personas, podría evitar las que usualmente están vacías. Para ello, Jockel et al. desarrollaron el framework EPIROME. Sus simulaciones mostraron que los agentes con memoria episódica se adaptan más rápido a nuevas situaciones que aquellos que carecen de ella.

A partir sus características, EPIROME no cumple con los requerimientos de un sistema de memoria episódica: No dispone de almacenamiento de largo plazo, mecanismos de recolección de episodios, ni representaciones de los eventos almacenados [10].

ALIZ-E (2014): ALIZ-E (Adaptative Strategies for Sustainable Long-Term Social Interaction) es un sistema cognitivo enfocado en la interacción natural con niños. Vijayakumar desarrolló una extensión de memoria episódica para ALIZ-E, basado en el funcionamiento de la memoria humana [7]. Su módulo construye episodios en formato RDF (Resource Description Framework), a partir de los archivos de *log* que genera el sistema tras cada interacción.

Vijayakumar muestra que su sistema cumple con los requisitos de eficiencia esperados para su plataforma objetivo. Sin embargo, no aborda el diseño de metodologías para adquisición de episodios, ni la definición de información semántica, pues se enfoca en adaptar los archivos de *log* preexistentes. Por otro lado, propone un caso de uso interesante para un robot social, a través de la adaptación de sus frases, según la información episódica almacenada, e.g., cambiando la frase "Estoy feliz de verte" a "Estoy feliz de verte *nuevamente*".

Bender: El año 2015 Sanchez et al. [9] diseñaron un sistema de memoria episódica para el robot Bender (plataforma de este trabajo). Este primer acercamiento es descrito en detalle en la Sección 1.2.1 y es utilizado como parte de la motivación para este trabajo.

Otros trabajos: Existen otros esfuerzos en equipar robots con memoria episódica, por ejemplo, MINERVA2 [27], LIDA [28], Rity [29], Homer [30] y personajes de historias virtuales [31]. De manera similar a los trabajos descritos anteriormente, la mayoría de estos sistemas ha sido diseñado para satisfacer un contexto particular, por lo que no son útiles para el diseño de nuevos sistemas cognitivos robóticos [10].

2.3.2.2. Aspectos fuera del alcance del trabajo

A continuación se presentan algunos estudios relacionados con aspectos de una LTM que escapan de los requerimientos para este trabajo o que simplemente no son basados en la taxonomía de la memoria humana. Estos estudios solo se presentan a modo de completitud, pues no permiten resolver el objetivo general, sino que solo comprender otros enfoques y acercamientos a la solución.

Thorsten et al. [32] proponen una memoria LTM para el robot BIRON. En su trabajo, se abstraen de la clasificación entre memorias episódica y semántica, pues todos los datos de largo plazo almacenados por el robot son considerados LTM. La memoria almacena solo datos de alto nivel, obtenidos tras el procesamiento de streams de datos básicos, como cámaras, micrófonos o actuadores. Los datos almacenados corresponden a un historial de percepciones y acciones de alto nivel realizadas, como: detecciones de objetos, interacciones verbales o la descripción de movimientos realizados. A pesar de su simplicidad, esta arquitectura centralizada permite reducir las dependencias entre cada componente y el ancho de banda utilizado para retransmitir la información entre procesos.

El sistema propuesto por Ho et al. [18] busca modelar la memoria de forma suficientemente general, como para permitir el traspaso de los recuerdos de un robot a otro, independientemente de que el hardware sea distinto. El costo de esto, es que se reduce la personalización de cada robot. Por otro lado, Ho et al. además aplican la teoría *Roboética*, sugerida por Veruggio y Operto [33], de donde derivan restricciones de diseño, relativas al manejo de información privada de los usuarios.

La LTM procedural es estudiada por Salgado et al. [19], donde demuestran que a través de ella se puede mejorar el desempeño del robot en tareas repetitivas. En 2014 Winkler et al. [34] desarrollaron CRAMm, un sistema para el manejo de memoria semántica y procedural basado en KnowRob [35], con el fin de dar soporte a las tareas de manipulación de su plataforma robótica. Mediante CRAMm, el robot puede tener conocimiento de los objetos del entorno, sus posiciones habituales y las rutinas de manipulación apropiadas para cada caso. Por ejemplo, puede saber que la leche es un objeto perecible, por lo que probablemente pueda ser encontrada en el refrigerador, más aún, CRAMm provee procedimientos para abrir ese contenedor y obtener el objeto.

La LTM también ha sido implementada mediante estrategias que no siguen las estructuras

de procesamiento utilizadas por la memoria humana. Por ejemplo, Kim et al. [36] plantean el uso de Deep Learning para modelar la memoria episódica y la planificación de acciones de manera holística. En su implementación, los procesos de codificación, almacenado y recuperación de episodios son manejados como uno. Así, el sistema LTM y la aplicación objetivo comparten la misma implementación, y todas las interacciones son manejadas automáticamente por la red.

2.4. Aspectos Relevantes para el Diseño

Esta sección presenta en detalle el marco teórico requerido para el diseño del sistema LTM. Se detallan aspectos de diseño para la memoria episódica y semántica. Además se revisan implementaciones de memoria emocional y relevancia histórica para modular el almacenamiento, deterioro y la adquisición de episodios.

2.4.1. Memoria explícita

La implementación de un sistema de memoria episódica no puede ser completamente separada de la memoria semántica. La primera almacena eventos y su contexto espacio-temporal, a la vez que la otra almacena información sobre cada entidad del episodio. Así, es posible recordar información de un objeto y su evolución histórica ligada a los eventos almacenados.

No existe un consenso sobre los contenidos, el formato o las herramientas para implementar una memoria explícita [8]. Sin embargo, si existe una aceptación generalizada sobre los requerimientos mínimos y deseables para su diseño [7, 18, 26]. Particularmente, Stachowicz en su trabajo del año 2012 [10] deriva un conjunto de 11 requisitos mínimos para la construcción de una memoria episódica y semántica, a partir de características cognitivas de sistemas episódicos naturales, extendidas con propiedades deseables desde un punto de vista técnico. Estos requisitos ($\mathcal{R}_1, \ldots, \mathcal{R}_{11}$), descritos a continuación, permiten dar una base para el diseño del sistema LTM.

2.4.1.1. Aspectos de diseño requeridos:

($\mathcal{R}_1$) **Contenido:** La información de eventos pasados debe ser recolectada e indexada respecto a su contexto espacio-temporal: Qué, dónde y cuándo pasó. El campo "Qué" debe permitir almacenar información variable, organizada en estructuras de datos que no se conocen de antemano y que se ajustan a diversos módulos de procesamiento.

($\mathcal{R}_2$) **Estructura:** Cada evento en conjunto con su contexto espacio-temporal forman una única representación integrada, que debe ser recolectada como un todo, en caso de recolectar cualquiera de las características del evento.

($\mathcal{R}_3$) **Flexibilidad:** La información almacenada es declarativa por naturaleza, y puede ser flexiblemente almacenada. Particularmente, puede interactuar con conocimiento semántico, incluso si este fue obtenido con posterioridad a la codificación del episodio.

($\mathcal{R}_4$) **Unicidad:** La memoria episódica cuenta con solo una instancia de cada evento para su entrenamiento, pues cada evento tiene características específicas a la situación. Es decir, cada episodio es único y no puede ser generalizado o entrenado a partir de una secuencia de episodios.

($\mathcal{R}_5$) **Introspección:** Recuerdos episódicos pueden ser almacenados por periodos de segundos, minutos, días o años. Es posible hablar sobre eventos asociados y acceder a ellos para introspección.

($\mathcal{R}_6$) **Perspectiva:** La memoria episódica debe lidiar con datos específicos al evento, lo que implica una perspectiva. Es decir, eventos recordados deben mantener la misma perspectiva que se tenía en la experiencia original [15]. Por otro lado, la memoria semántica generalmente se puede abstraer de la perspectiva y aspectos situacionales.

($\mathcal{R}_7$) **Anidamiento:** Los eventos almacenados en la memoria episódica pueden variar en tiempo y extensión. Particularmente, pueden ocurrir eventos dentro del lapso temporal de otro, ambos asociados al mismo contexto episódico.

($\mathcal{R}_8$) **Transposición:** Los eventos almacenados en la memoria episódica pueden variar en tiempo y extensión. Particularmente, un evento A puede iniciar antes B, pero terminar durante la vida de B.

2.4.1.2. Aspectos de diseño deseables:

($\mathcal{R}_9$) **No intrusivo:** Se espera que la LTM no requiera dependencias de módulos externos para poder funcionar y representar los datos. A la vez, no puede depender en que los otros módulos no cambien la representación de sus datos.

($\mathcal{R}_{10}$) **Eficiente:** El sistema debe ser lo suficientemente eficiente para tolerar el manejo de una alta tasa de eventos, sin degradar el funcionamiento del robot. Es decir, todos los eventos deben ser procesados eventualmente, aún cuando el robot esté ocupando gran parte de sus recursos, y sin generar hambruna de CPU ni de ancho de banda (de disco y red) al resto de los procesos.

($\mathcal{R}_{11}$) **Escalable:** Los costos asociados al manejo de la información en la memoria (agregar, eliminar, actualizar y buscar datos) no deben exceder un límite superior, respecto a la cantidad creciente de datos almacenados. La memoria debe mantener los costos acotados, dentro de un rango que no entorpezca su uso.

2.4.2. Modulación de procesos cognitivos

El almacenamiento, deterioro y recolección de episodios puede ser afectado por diversos factores. A continuación se detallan estudios que influencian el diseño, en cuanto a la modulación de tales procesos cognitivos, a partir de la memoria emocional y la relevancia histórica de los episodios. En otras palabras, los estudios que aquí se exponen son la base para el cálculo de las relevancias episódicas de la memoria LTM.

2.4.2.1. Memoria emocional

La importancia de un evento se ve fuertemente influenciada por el estado emocional de una persona, o en este caso, el robot. Así, se puede dar prioridad a unos eventos por sobre otros, afectando los procesos cognitivos. Particularmente, un evento importante emocionalmente podría ser más fácilmente recordado, o ser almacenado por un periodo más largo [14]. A continuación se presenta un acercamiento a la generación de reacciones emocionales en un robot, a partir de los estímulos que este percibe.

La implementación de la memoria emocional requiere como mínimo de un mapeo entre los estímulos percibidos por el robot y las sensaciones emocionales que estos generan. Dood et al. [24] proponen el uso de la teoría emocional de reacciones de Haikonen [37], que considera a una emoción como una combinación de estímulos básicos. Las sensaciones elementales son: bienestar, malestar, dolor, placer e interés.

Dood et al. proponen implementar las sensaciones a partir de distintos estímulos medidos en un robot:

- Según las entradas de sensores. Por ejemplo, actuador que se aproxima a sus límites de movimiento físico o de fuerza, y el nivel de iluminación o ruido acústico percibido.
- Según el nivel de interacción con humanos. Por ejemplo, cantidad de interacciones en un periodo, ausencia de humanos o cantidad de humanos en el entorno.
- Según el cumplimiento de objetivos y expectativas. Por ejemplo, de acuerdo a las tareas que se lograron cumplir y los errores cometidos.

Sistemas más avanzados incluso pueden considerar la generación de reacciones emocionales, basándose en las sensaciones derivadas anteriormente. En la Tabla 2.1 se muestran las reacciones generadas según el modelo de Haikonen. Además, estas se podrían reflejar en la personalidad del robot, por ejemplo, mediante gestos, vocabulario o nivel de aceptación para realizar una acción. Kasap et al. [38] utilizan un sistema llamado *Emotion Engine*, para generar reacciones emocionales y simular cambios de personalidad de un robot, según las sensaciones percibidas.

Para su uso efectivo dentro de un esquema LTM, se espera que las sensaciones reportadas incluyan un nivel de intensidad. Según el nivel percibido en cada episodio, es posible clasificarlos entre eventos muy o poco relevantes. Los más relevantes tendrán mayor probabilidad

Sensación Elemental	Reacción
Bueno: gusto, aroma	Aceptación, Acercar
Malo: gusto, aroma	Rechazo, Alejar
Dolor: autoinfligido	Alejar, Desistir
Dolor: agente externo	Agresión
Dolor: sobre esfuerzo	Sumisión
Placer	Mantener, Acercar
Acierto	Mantener atención
Desacierto	Migrar atención
Novedad	Enfocar atención

Tabla 2.1: Sensaciones elementales y sus reacciones correspondientes, según el modelo de Haikonen. Obtenido de [24].

de ser recuperados al recordar. Deutsch et al. [14] consideran que la intensidad de las sensaciones es importante, pues permite evitar costos de búsqueda lineales dentro de todos los episodios almacenados.

En el año 1980 Plutchik [39] presenta la Rueda de las Emociones, que se muestra en la Figura 2.2. Su trabajo es uno de los más influyentes para la clasificación de las emociones y sus respuestas. Plutchik propone 8 emociones primarias, cada una es de carácter bipolar y con un nivel de intensidad asociado: ira - miedo, alegría - tristeza, confianza - aversión, sorpresa - anticipación. La rueda además muestra 8 emociones avanzadas de carácter bipolar, formadas a partir de las 2 emociones primarias adyacentes.

2.4.2.2. Relevancia histórica

La relevancia histórica es un indicador que representa el olvido o pérdida de interés sobre un episodio en la memoria. Está directamente relacionado a la edad de un episodio, a menor antigüedad, mayor es su importancia, lo que se traduce en una mayor probabilidad de recordar ese episodio.

La implementación de relevancia histórica es estudiada por Dood et al., donde presenta una estrategia para su implementación y su interacción con la memoria emocional [24]. Propone el uso de un indicador de relevancia generalizado, obtenido al combinar la intensidad de los dos indicadores. Además, propone un modelamiento matemático para representar el decaimiento de la relevancia histórica en el tiempo.

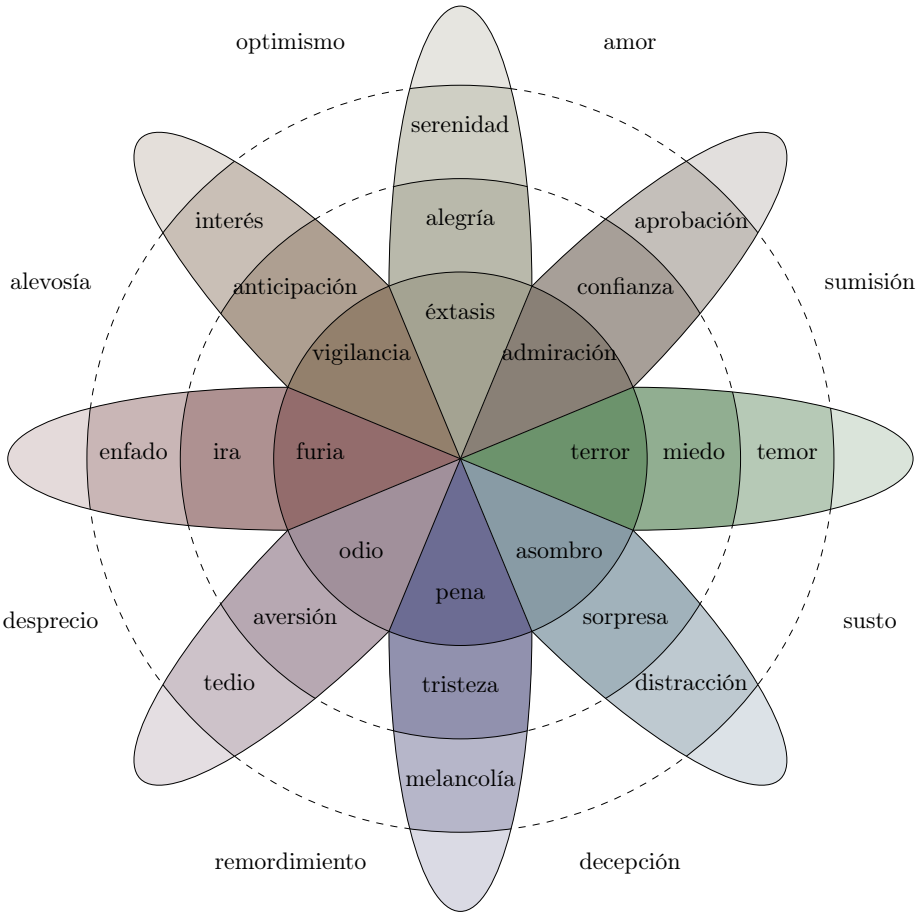


Figura 2.2: Rueda de las Emociones de Plutchik. Cada rama representa una de las 8 emociones primarias y sus 3 niveles de intensidad.

3 Aspectos Técnicos

En este capítulo se describen los componente de software y hardware utilizados para comprender el diseño e implementación del trabajo. Se revisan los conceptos utilizados sobre el framework robótico ROS. Se hace una revisión breve de la base de datos MongoDB. Se describe la librería SMACH, utilizada para la creación de máquinas de estado. Finalmente, se presenta a Bender junto a los aspectos requeridos de su sistema operativo.

3.1. ROS

ROS, acrónimo para Robot Operating System, es un trabajo que funciona como *middleware* para aplicaciones robóticas, y permite resolver el problema de la comunicación entre procesos. Es una colección de herramientas, librerías y convenciones que buscan simplificar la tarea de crear comportamientos robóticos complejos y robustos, sin importar la plataforma robótica [5].

Fue originalmente creado por la organización WillowGarage en el 2008, y es mantenido actualmente por la Open Source Robotics Foundation (OSRF). Existe un ecosistema ROS mantenido por la comunidad y con cientos de módulos de software con soluciones a problemas específicos, los que pueden interconectarse para construir comportamientos más complejos. Por lo anterior, su uso se ha convertido en una práctica mundial, siendo adoptado incluso en soluciones industriales.

A continuación se introducen los conceptos de interés, necesarios para entender el diseño e implementación del trabajo. En primer lugar, se presenta la estructuración jerárquica de los componentes en ROS. Luego, se detalla el protocolo de comunicación utilizado, basado en *tópicos*, *servicios* y *acciones*. Finalmente, se presentan herramientas que provee ROS para soportar el desarrollo y ejecución de programas.

3.1.1. Intraestructura

En esta sección se presentan los 3 componentes sobre los que se construye la estructura jerárquica para agrupar programas en ROS.

Nodo: Es un programa ejecutable en ROS, enfocado en una tarea específica. Cada nodo puede establecer comunicación con otros, a través de una interfaz ROS especializada, presentada más adelante. ROS provee librerías oficiales para escribir nodos en los lenguajes C++ (`roscpp`[1]), Python (`rospy`[2]) y Common Lisp (`roslisp`[3]), junto a librerías experimentales para otros lenguajes.

Paquete: Corresponde a un conjunto de nodos, junto a todos sus archivos necesarios para su compilación y ejecución. Cada paquete tiene asociado un nombre único en el sistema, junto a una lista de dependencias de otros paquetes. ROS provee conjunto estándar de paquetes y librerías enfocados en diversos aspectos de la robótica, que en conjunto a los paquetes mantenidos por la comunidad, permiten simplificar la construcción y mantenimiento del software un robot.

Distribuciones: Una distribución es un conjunto versionado de paquetes ROS. Su propósito es permitir que los desarrolladores trabajen en un conjunto relativamente estable de dependencias, hasta que puedan avanzar a la siguiente versión. Se libera una distribución anualmente, enfocada a una versión LTS[4] de Ubuntu, y las versiones pares de ROS también se consideran LTS y mantienen soporte por 5 años.

3.1.2. Comunicación

En esta sección se detallan los componentes del protocolo de comunicación utilizado por los nodos en ROS.

ROS master: La ejecución de uno o más nodos requiere la presencia del programa *ROS master*, un servidor que provee servicios de registro y búsqueda al resto de los nodos. Mantiene una lista de los tópicos, servicios y parámetros disponibles, que han registrado los nodos activos. Su principal funcionalidad es permitir que cada nodo pueda encontrar a los otros. Una vez establecida la comunicación, los nodos se comunican en una red P2P[5]. Comúnmente la comunicación en ROS se realiza mediante el protocolo HTTP, bajo un transporte TCP, sin embargo, los nodos pueden ser configurados para utilizar un transporte basado en UDP. Por lo tanto, ROS permite la comunicación de nodos que se encuentren en distintas máquinas, siempre que sea posible establecer una comunicación HTTP entre ellos y el ROS master.

[1]Documentación oficial de `roscpp`: http://wiki.ros.org/roscpp.
[2]Documentación oficial de `rospy`: http://wiki.ros.org/rospy.
[3]Documentación oficial de `roslisp`: http://wiki.ros.org/roslisp.
[4]LTS es un acrónimo para Long-Term-Support. Una versión LTS de un software indica que este contará con soporte a largo plazo, es decir, durante un periodo mayor a su periodo normal de vigencia.
[5]En una red P2P (peer-to-peer) los participantes se comunican total o parcialmente sin requerir un servidor intermediario.

Mensaje: Es una estructura de datos utilizada para comunicar un nodo con otro. Se definen en un archivo de tipo `.msg`, cuyos campos se construyen a partir de tipos dato primitivos en ROS[6] y otros mensajes preexistentes. Durante la compilación se genera el código para poder utilizarlos mediante las librerías ROS de cada lenguaje de programación disponible.

Tópico: Permite que los nodos de comuniquen a través de mensajes, y está basado en el patrón de diseño observador: cada tópico es un canal de comunicación que transmite mensajes ROS de un tipo predefinido; cada nodo puede publicar simultáneamente en uno o más tópicos; cada nodo puede suscribirse a los tópicos que desee, siendo notificado por cada mensaje entrante. Esto se describe en el diagrama de la Figura 3.1.

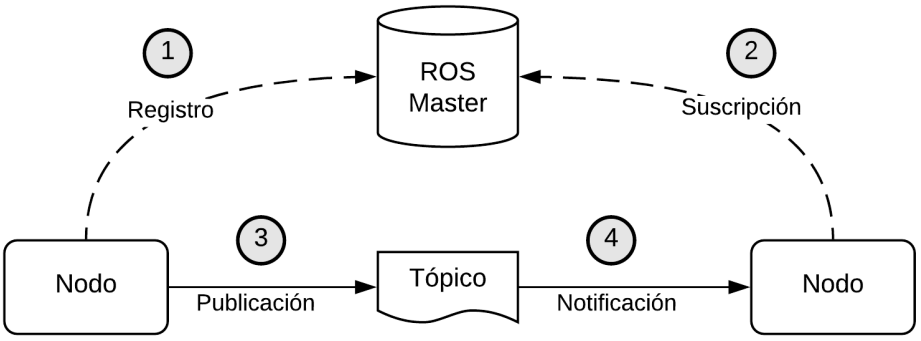


Figura 3.1: Diagrama de la comunicación mediante tópicos en ROS. Primero, el ROS master se encarga de establecer el proceso de comunicación entre los nodos involucrados. Luego, los nodos se comunican **unidireccionalmente** usando el tópico establecido.

Servicios: Es una forma de comunicación cliente-servidor de carácter síncrono. Se construyen a partir de una estructura de datos de tipo `.srv`, que especifica mensajes ROS de petición y respuesta, disponible tras la compilación. Cada nodo (servidor) puede proveer uno o más servicios. Cada nodo (cliente) puede enviar peticiones al servidor. Las peticiones son de carácter bloqueante, hasta que una respuesta desde el servidor sea recibida.

Acciones: Esta forma de comunicación se construye a partir de tópicos y servicios, mediante la librería `actionlib`[7]. Está enfocada en la resolución de tareas de largo plazo, mediante la definición de objetivos, su ejecución y la notificación de su evolución al cliente. El servidor mantiene una máquina de estados que recibe peticiones (servicio) e informa constantemente sobre el estado del proceso y su resultado (tópicos). La estructura de datos utilizada es de tipo `.action`, que especifica mensajes ROS para la petición, el *feedback* y el resultado.

[6]ROS provee alrededor de 15 tipos primitivos, que permiten representar valores de tipo booleano, entero, números de punto flotante, texto y tiempo. Además, el formato para mensajes permite definir arreglos de cada tipo de dato y de otros mensajes. Documentación oficial: http://wiki.ros.org/msg

[7]Documentación oficial de `actionlib`: http://wiki.ros.org/actionlib.

3.1.3. Herramientas

En esta sección se describen herramientas que provee ROS para la implementación de nodos y su ejecución.

Servidor de parámetros: ROS provee un servidor para almacenar, modificar y obtener parámetros compartidos entre los nodos. Estos permiten la configuración del programa sin tener que recompilar el código, y funcionan como el estándar para el manejo de parámetros. La configuración por defecto puede ser especificada en archivos usando el formato YAML [40].

Pluginlib: Es un paquete oficial de ROS[8] que provee herramientas para escribir y cargar plugins dinámicamente en nodos C++. La librería permite delegar la implementación de funcionalidades específicas a los usuarios. El nodo que ocupará el plugin debe definir una clase abstracta a ser implementada. Mientras que los paquetes proveedores deben registrar las librerías con sus implementaciones, para que los plugins queden disponibles en el sistema.

Roslaunch: La librería `roslaunch`[9] es una herramienta que permite la ejecución y configuración de múltiples nodos, a través de solo un comando en la consola del sistema. Utiliza archivos en formato XML y con extensión `.launch`, donde se define un árbol de nodos y otros archivos *launch* a ser ejecutados. Esta herramienta permite modularizar la ejecución de un sistema robótico complejo, compuesto generalmente por decenas de nodos.

3.2. MongoDB

MongoDB es una base de datos no relacional orientada a documentos tipo JSON [41], donde los campos pueden variar de documento a documento y la estructura de datos puede ser modificada en el tiempo [42]. MongoDB está diseñada para funcionar de manera distribuida, proveer alta disponibilidad, escalamiento horizontal y distribución geográfica. Es una base de datos gratis y de código libre, publicada bajo la licencia GNU AGPL.

3.2.1. Conceptos

MongoDB representa documentos JSON en un formato binario denominado BSON. BSON extiende JSON a través de nuevos tipos de datos, introduce campos ordenados y está diseñado para proveer eficiencia al codificar y decodificar la información en distintos lenguajes.

Los documentos son organizados en *colecciones*, cada una destinada a almacenar información asociada a un concepto particular. MongoDB provee funcionalidades para consultas, indexado y realizar agregaciones de los datos almacenados.

MongoDB provee drivers para diversos lenguajes de programación, entre ellos C++, Java

[8] Documentación oficial de `pluginlib`: http://wiki.ros.org/pluginlib.
[9] Documentación oficial de `roslaunch`: http://wiki.ros.org/roslaunch.

y Python. Para este trabajo, el driver de interés es el de C++[10].

En su configuración por defecto, el servidor de MongoDB soporta documentos de hasta 16MB de tamaño, lo que restringe la información a almacenar, pero permite mantener acotados los costos de las operaciones sobre la base de datos. MongoDB permite remover este límite mediante la estrategia de almacenamiento GridFS, la que divide documentos grandes en subdocumentos menores a 16MB, para ser tratados normalmente.

3.2.2. Interfaz ROS

MongoDB es una de las dependencias de *MoveIt!*[11], librería estándar utilizada para la manipulación de objetos, de uso extendido en la comunidad robótica. Por lo tanto, MongoDB ya está instalada y funcionando en muchos robots de servicio basados en ROS.

El equipo desarrollador de MoveIt! y sus paquetes ROS relacionados mantiene el paquete `warehouse_ros_mongo`[12], que provee una interfaz ROS al driver en C++ para MongoDB. MoveIt! y sus dependencias son consideradas estables, al menos hasta la distribución ROS *melodic*, con fecha de caducidad para el 2023.

`warehouse_ros_mongo` permite definir colecciones destinadas a almacenar mensajes ROS, de manera binaria. Cada documento es construido a partir del mensaje ROS serializado (mediante su API ROS), sumado a un conjunto de metadatos utilizados para la definición de índices y consultas. Actualmente, la API solo soporta la definición de metadatos de los tipos `bool`, `int`, `double` y `std::string` en C++.

3.3. SMACH

SMACH es una librería en Python para la creación y ejecución de máquinas de estado. Permite crear rápidamente comportamientos robóticos complejos, basados en capacidades de alto nivel de un robot. Está diseñada para ser independiente de ROS, pero provee una interfaz para su uso. Se puede encontrar en el conjunto de paquetes ROS denominado `executive_smach`[13].

En SMACH una máquina de estado está compuesta por un conjunto de estados y sus transiciones. La librería soporta anidamiento, por lo que cualquier estado puede ser reemplazado por otra máquina de estado disponible. Cada estado debe proveer una función `execute` a ser ejecutada cuando esté activo. En la Figura 3.2 se presenta máquina de estado construida en SMACH, que ejemplifica el concepto de anidación de estados.

[10]Documentación del driver de MongoDB para C++: http://mongodb.github.io/mongo-cxx-driver/.
[11]Documentación oficial de MoveIt!: https://moveit.ros.org/.
[12]El paquete ROS `warehouse_ros_mongo` está disponible en: https://github.com/ros-planning/warehouse_ros_mongo.
[13]Documentación oficial de SMACH disponible en: http://wiki.ros.org/smach.

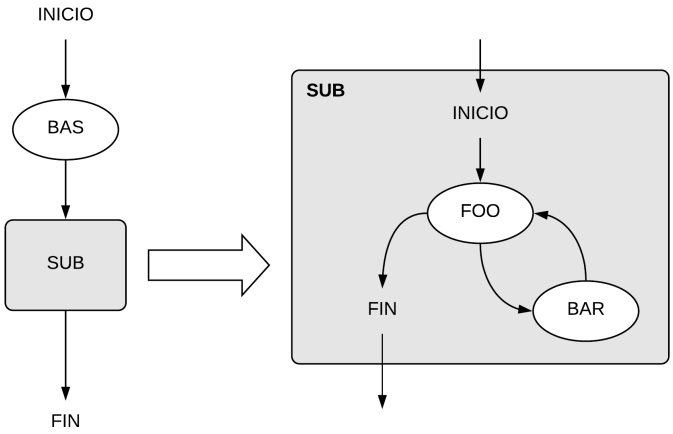


Figura 3.2: Máquina de estado construida en SMACH para ejemplificar el concepto de anidación de estados. A la izquierda, se muestra la máquina principal con los estados "BAS" y "SUB". A la derecha, se muestra que "SUB" es una máquina compuesta por los estados "FOO" y "BAR".

3.4. UChile ROS Framework

UChile ROS Framework (URF) hace referencia al sistema de software desarrollado en el Laboratorio de Robótica del Departamento de Ingeniería Eléctrica de la Universidad de Chile, para sus robots de servicio. El sistema cuenta con 10 años de desarrollo y está orientado a cumplir los requisitos de la competencia RoboCup en su categoría @Home.

3.4.1. Conceptos

URF está construido sobre ROS y utiliza una estructura de 4 capas, cómo se presenta en el diagrama de la Figura 3.3:

1. La primera capa de software contiene todas las dependencias del sistema, ya sean de ROS o no. Es la única capa que solo posee código externo.

2. Sobre la capa de dependencias se monta una capa ROS de bajo nivel, con herramientas y librerías comunes, sumado a los drivers necesarios para manejar el hardware de cada robot.

3. La siguiente capa alberga capacidades robóticas avanzadas, relacionadas a la percepción robótica, manipulación de objetos, navegación autónoma e interacción humano-robot.

4. Finalmente, la capa de alto nivel cumple dos propósitos. En primer lugar, provee una interfaz Python para el uso de las capacidades de menor nivel, la que es utilizada principalmente para la elaboración de máquinas de estado y comportamientos robóticos complejos mediante SMACH. Segundo, almacena los archivos de configuración y los *launchfiles* para adecuar URF al robot objetivo. Además, provee herramientas para

visualizar la ejecución del sistema robótico.

Todos los módulos de URF son de código libre, a excepción de los algoritmos relacionados con percepción y la interfaz de alto nivel. El código se almacena públicamente en la organización *uchile-robotics* en GitHub[14].

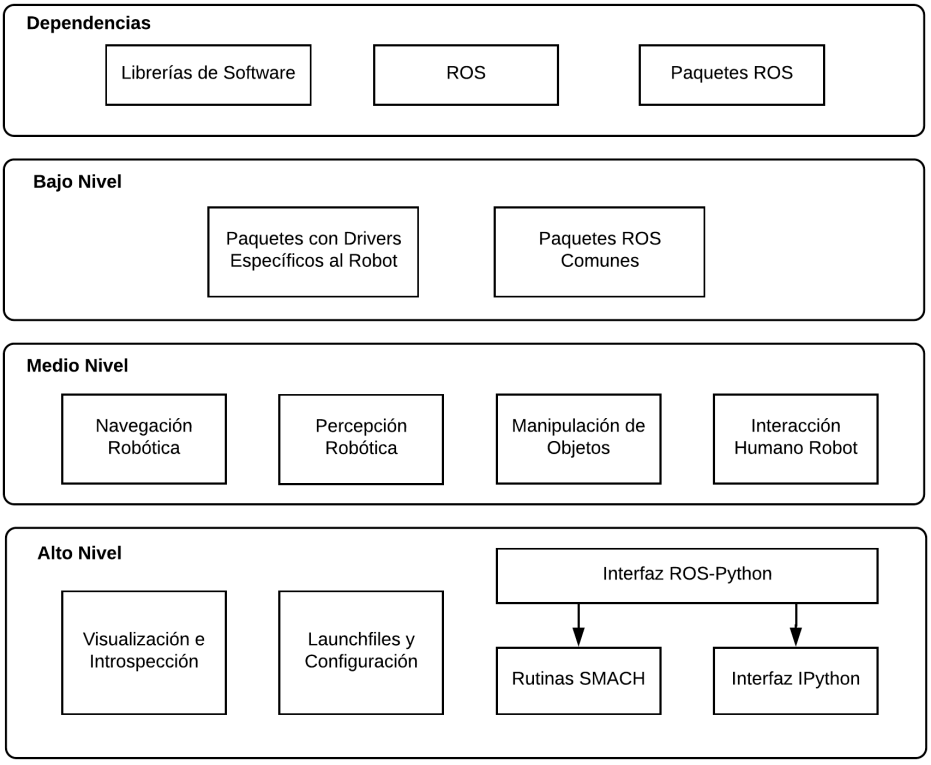


Figura 3.3: Diagrama general con componentes de software de UChile ROS Framework. Cada bloque de software depende de los módulos mostrados en las capas de menor nivel.

3.4.2. Memoria a corto y largo plazo

En un robot implementado sobre URF es posible acceder a la información compartida por sus procesos. Cualquier módulo ROS en el sistema tiene acceso a los datos extraídos desde sensores y luego generados en postprocesamientos, junto al acceso para controlar el hardware.

Existen algunas formas de memoria implementadas en URF, comparables a los conceptos definidos para la memoria humana. También se pueden dividir en de corto y largo plazo, los que se describen a continuación.

Como STM, se puede definir como memoria de trabajo a todo el flujo de información presente durante la ejecución del robot. Lo que incluye datos sensados, procesamientos y

[14]Organización *uchile-robotics* y URF en GitHub: https://github.com/uchile-robotics

acciones realizadas. Generalmente, tales datos solo están disponibles durante la ejecución de una rutina y no son almacenados para posteriores ejecuciones.

A manera de LTM, se puede encontrar una memoria procedural, relacionada con todo el conocimiento almacenado que posee el robot para cumplir ciertas tareas. Caen en esta categoría: modelos para percepción robótica, modelos para reconocimiento de voz y patrones, bases de datos de movimientos precalculados para manipular objetos y acciones predefinidas que se utilizan para controlar el robot.

También se pueden encontrar especializaciones de memoria LTM semántica. Ejemplos de esto son: el mapa que se conoce del entorno, junto a los lugares y objetos anotados en él; diccionarios con información anotada sobre entidades y sus características, cómo personas y objetos; bases de datos con imágenes anotadas para el reconocimiento de objetos y personas.

Sin embargo, en URF no existen formas de memoria emocional ni episódica de largo plazo. Luego, toda interacción realizada por los robots está limitada a la información obtenida desde el inicio al fin de cada rutina.

3.5. Bender

Bender, presentado en la Figura 3.4, es un robot humanoide de tipo doméstico, cuyo objetivo es ser un robot mayordomo para el hogar. Fue creado el año 2007 en el laboratorio de robótica del Departamento de Ingeniería Eléctrica de la Universidad de Chile, por el equipo UChile Homebreakers [3].

En cuanto a actuadores, el robot cuenta con 2 brazos antropomórficos de 6 grados de libertad cada uno, una base móvil diferencial Pioneer 3-AT, un cuello que permite rotaciones en dos ejes cartesianos; pudiendo imitar gestos de asentimiento y negación, y finalmente, una cabeza que puede mostrar expresiones faciales mediante movimientos de su boca, orejas, cejas y cambios de colores alrededor de los ojos.

El robot cuenta con los siguientes sensores: un láser Hokuyo UTM-30LX, un laser Hokuyo URG-04LX-UG01, un micrófono M-Audio Producer USB y una cámara de profundidad ASUS Xtion Pro.

El software de Bender está basado completamente en el framework URF, de acuerdo a lo descrito en la sección anterior. Bender cuenta con alrededor de 80 paquetes ROS mantenidos por el equipo, sumado a un conjunto de dependencias en paquetes ROS de la comunidad.

Figura 3.4: Robot Bender en una prueba de la competencia RoboCup@Home del año 2015.

4 Diseño

En este capítulo se revisa el diseño del sistema LTM implementado en el trabajo. El capítulo es extenso y algunos puntos son muy detallados, por lo que se inicia con una revisión general del sistema LTM diseñado, con el fin de guiar su exposición. Considerando esto, se describe el diseño del trabajo, iniciando desde sus requerimientos hasta su integración al robot. Primero se presentan todos los requerimientos de diseño y se elabora un conjunto de validaciones que permitirán guiar el diseño, implementación y posterior evaluación del trabajo. En las siguientes 4 secciones se describe la arquitectura del sistema LTM: se revisa el diseño de los episodios, su estructura de datos y limitantes; se estudia el diseño del modelo de datos para la memoria episódica y su relación con los componentes semánticos; en tercer lugar, se presenta el diseño del servidor LTM; finalmente, se definen la interfaz episódica y los componentes específicos a implementar para el robot Bender.

4.1. Revisión General

A continuación se presenta una revisión general del sistema LTM diseñado, junto a algunas consideraciones relevantes para la comprensión de este.

4.1.1. Conceptos

Acá se presentan algunas revisiones sobre conceptos utilizados en el capítulo y aclaraciones sobre la terminología empleada.

Usuario versus operador: Primero, se hará la distinción entre el usuario del sistema y el operador de este. En adelante, se utilizará el término *usuario* para indicar a la(s) persona(s) encargadas de utilizar y modificar el software LTM para un robot en particular, es decir, un usuario es un *desarrollador* de software. Por otro lado, los términos *operador* y *humano* serán utilizados para referirse a alguien que posiblemente interactuaría con el robot. Por lo general, en casos de prueba, un usuario hace el papel de operador, sin embargo, notar esta diferencia

es relevante. Finalmente, se utilizará el término *robot* para referirse al robot objetivo donde se implementa el sistema LTM. A modo de ejemplo, el equipo UChile Homebreakers sería un **usuario** del sistema LTM implementado en el **robot** Bender, mientras que una persona del público con quien interactúa el robot es llamada **operador**.

Memoria episódica versus semántica: Ya que ambos son términos recurrentes en este informe, merece la pena la siguiente aclaración. Un *episodio* almacena la información de qué, dónde y cuándo sucedió un evento. La *memoria episódica* es una colección de episodios. La *memoria semántica* es una colección de información sobre *entidades* particulares. La *memoria LTM* se compone de las memorias episódica y semántica, relacionadas por el qué sucedió, respecto a las entidades conocidas.

Tareas y comportamientos: Una *tarea* hace referencia a toda actividad realizada por el robot durante su funcionamiento normal, para cumplir con sus obligaciones como robot doméstico. Ejemplos de esto son la interacción con humanos y el aseo del hogar. En adelante, se utilizarán los términos de *tarea* y *comportamiento* para indicar este concepto.

4.1.2. Sistema LTM

Según se detalla en las siguientes secciones del capítulo, el sistema LTM diseñado emula una memoria LTM humana con componentes episódicos, semánticos y emocionales. A modo general, en el diagrama de la Figura 4.1 se presenta el funcionamiento del sistema LTM diseñado. En primer lugar, se recopila información a partir de las tareas que realiza el robot y su memoria de trabajo, para generar los episodios a almacenar. Luego, el sistema se encarga de mantener los datos y responder las consultas episódicas o semánticas que se requieran.

Tarea: Las tareas realizadas por el robot son utilizadas para indicar el inicio y término de un episodio y conocer la actividad realizada por el robot. Una tarea puede ser dividida en diversas subtareas más detalladas. La interfaz de tarea es utilizada para acceder a la información anterior e iniciar la generación de nuevos episodios.

Memoria de trabajo: Cada episodio generado debe poder responder a las consultas dónde y qué sucedió, para lo que se recopila información de localización y de las entidades de interés. Además, la información del estado emocional del robot permite asignar una relevancia a los episodios almacenados. Todos los datos anteriores son recopilados a partir de la memoria de trabajo del robot, durante la ocurrencia del episodio.

Generador de episodios: La generación de episodios utiliza la interfaz de tarea para reconocer nuevos episodios, y la memoria de trabajo del robot, para recopilar los datos necesarios para crear el episodio. Un episodio es generado por cada tarea y subtarea registrada.

Servidor LTM: El servidor LTM hace referencia al programa encargado de almacenar, mantener y proveer interfaces de consulta para la memoria a largo plazo generada. El servidor utiliza la base de datos MongoDB para almacenar las memorias episódica y semántica. Es diseñado para funcionar en ROS y proveer una API de registro y consulta de episodios.

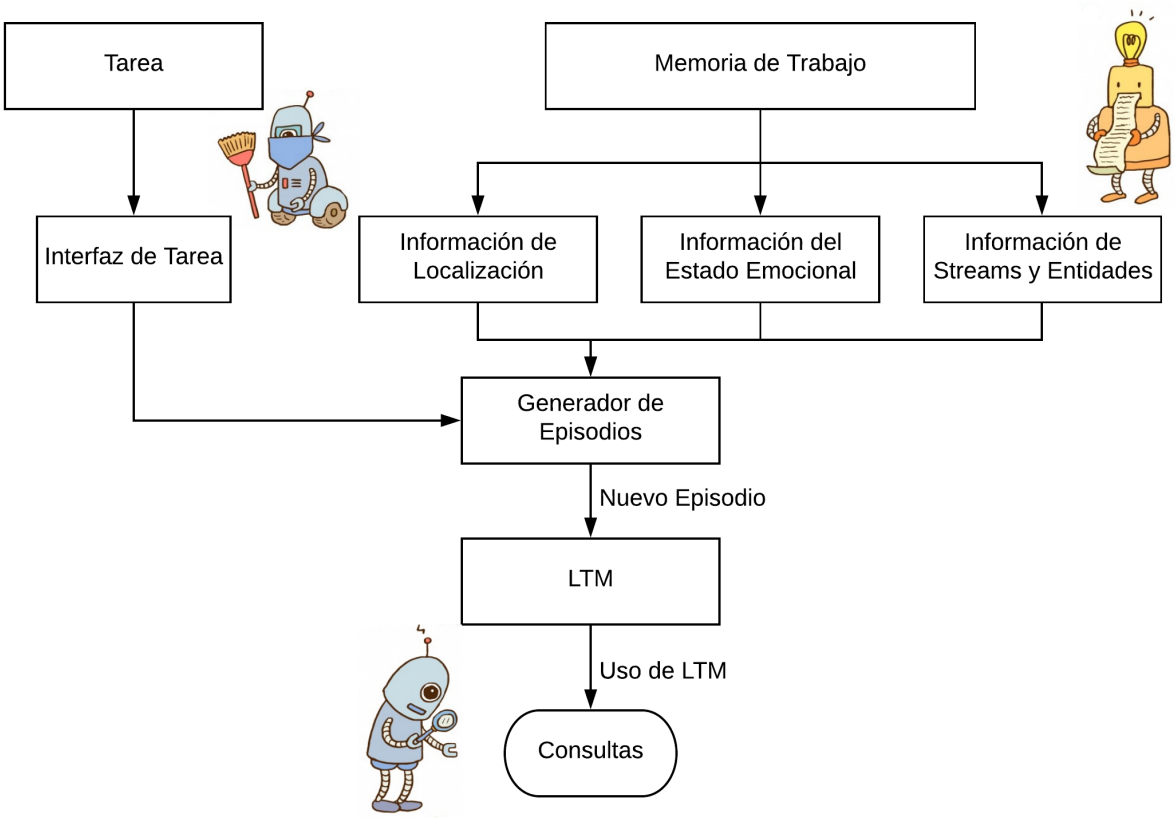


Figura 4.1: Diagrama general del sistema LTM diseñado. Los episodios son generados a partir de información extraída sobre la tarea que ejecuta el robot y su memoria de trabajo disponible en el momento. La información almacenada puede ser consultada posteriormente.

Consultas: A través de la API de consulta del servidor LTM, es posible adquirir información episódica y semántica. Las consultas se construyen utilizando el lenguaje de consultas que provee MongoDB en formato JSON. El sistema es capaz de responder consultas como: qué sucedió en cierta fecha o hablar sobre una persona en particular, sus datos y cómo se ha modificado su conocimiento sobre ella en el tiempo.

Generalidad y sistema de plugins: El sistema LTM está diseñado para ser utilizado en Bender y otros robots, mediante el uso de un sistema de plugins. Ya que la información semántica a almacenar depende del robot objetivo, el sistema permite la definición de sus contenidos mediante cualquier estructura de datos representable en un mensaje ROS. Luego, la interfaz de tarea y las interfaces para acceder a la memoria de trabajo deben ser implementadas por el usuario, mediante una serie de plugins adecuados al robot. Particularmente, el sistema LTM requiere la implementación de 4 plugins, para recopilar información sobre la ubicación del robot, emociones del robot, streams de datos percibidos y entidades.

Integración en Bender: Para el caso de Bender, este trabajo provee implementaciones de cada uno de los plugins requeridos. Por un lado, las tareas son recopiladas desde máquinas de estado en SMACH. Por otro lado, se implementan plugins para acceder al sistema de localización del robot, recopilar información emocional y para recopilar datos sobre las

entidades modificadas durante la tarea.

4.2. Requerimientos y Validaciones

A continuación se presentan los requisitos de diseño sobre los que se construye el trabajo y las validaciones generadas a partir de ellos. Primero, se da una descripción breve sobre el origen y razón de los requisitos. Luego, se presenta un listado formal de todos los requerimientos, escritos de una forma clara y verificable. Para concluir, se presenta un conjunto de pruebas que busca validar cada requerimiento expuesto.

4.2.1. Objetivos del sistema

El primer conjunto de requisitos se deriva a partir de los objetivos del trabajo y sus alcances, presentados en la Sección 1.3. En adelante se referirá a estos como *requisitos de proyecto*.

($\mathcal{RP}_1$): Diseño de LTM episódica y semántica para robots de servicio domésticos, debe basarse en los 11 requerimientos para una memoria episódica planteados por Stachowicz [10] y presentados en la Sección 2.4.1.

($\mathcal{RP}_2$): Este diseño debe considerar la modulación de los procesos cognitivos mediante memoria emocional.

($\mathcal{RP}_3$): El diseño del sistema debe soportar el concepto de relevancia histórica.

($\mathcal{RP}_4$): Además, los episodios deben manejar un indicador de relevancia generalizada, que encapsule todos los demás indicadores en solo uno.

($\mathcal{RP}_5$): Diseño del sistema debe ser agnóstico del robot a utilizar.

($\mathcal{RP}_6$): El trabajo debe ser integrado en Bender.

4.2.2. Requerimientos de sistema

Se desarrolló un documento con 26 requisitos de sistema, derivados a partir de los objetivos revisados en la sección anterior. Para evitar extender innecesariamente este informe, el documento desarrollado es de carácter semiformal, es decir, por cada requisito de sistema solo se presenta: su descripción, su numeración y el requisito de proyecto asociado. En adelante, cada requisito será indicado por su prefijo y numeración de la forma RSXX. El listado de requisitos es extenso y puede ser encontrado en el Anexo A.1 junto con la matriz de trazabilidad entre requisitos de proyecto y sistema.

Los requisitos se pueden separar en 4 categorías de acuerdo a su finalidad:

1. RS01: Requisito sobre diseño agnóstico del robot objetivo. Ver $\mathcal{RP}_5$.

2. RS02-RS18: Requisitos derivados de reglas episódicas de Stachowicz. Ver $\mathcal{RP}_1$.

3. RS19-RS22: Requisitos sobre relevancias episódicas. Ver $\mathcal{RP}_2$, $\mathcal{RP}_3$, $\mathcal{RP}_4$.

4. RS23-RS26: Requisitos sobre la integración en Bender. Ver $\mathcal{RP}_6$.

4.2.3. Validaciones

También se desarrolló un documento (semiformal) con validaciones para cada uno de los requisitos de sistema en el documento. Las validaciones sirven como guía para la implementación del trabajo y el resultado de su ejecución es presentado en el Capítulo 6. El listado completo es extenso y se encuentra en el Anexo A.2.

El documento consta con alrededor de 37 validaciones, separadas en 3 categorías. Cada validación se identifica por un prefijo y numeración de la forma VTXX (V: Validación, T: Tipo, XX: numeración). Además, cada una indica su requisito de sistema asociado. Las categorías son las siguientes:

1. VAXX: Validaciones funcionales. Tienen como objetivo verificar que el sistema LTM cumple con todos los requisitos mínimos de funcionalidad, exceptuando los de integración con el robot Bender. En este caso, las validaciones pueden ser ejecutadas mediante simulaciones.

2. VBXX: Validaciones de integración con Bender. Tienen como objetivo verificar que el sistema LTM cumple con los requisitos de integración con el robot Bender. Para esto, se debe integrar el sistema al robot y ejecutar las pruebas en sus computadores.

3. VCXX: Validaciones de desempeño. Tienen como objetivo medir el desempeño del sistema LTM en términos de escalabilidad y eficiencia de este, para el caso de uso del robot Bender.

Las validaciones VCXX de eficiencia tienen por objetivo medir el uso de recursos en término de la tasa de operaciones que debe soportar el sistema LTM en Bender. Se construyen a partir del requisito de sistema RS15. Las validaciones de escalabilidad se construyen a partir del requisito de sistema RS16. Estas tienen por objetivo medir el costo en tiempo de las operaciones de inserción y búsqueda de episodios, en función de la cantidad de episodios almacenados en la base de datos.

En el Anexo A.2 se presenta la matriz de trazabilidad indicando la relación entre cada requisito de sistema y las validaciones asociadas.

4.3. Diseño de Episodios

A continuación presentan las decisiones de diseño seguidas para la representación de los datos episódicos mediante mensajes de ROS. Se estudia el formato a utilizar para el almacenamiento de cada episodio, seguido de la estructura de datos tipo árbol que soporta sus anidamientos. Luego se explica el diseño de los campos episódicos (*What, When, Where*), la representación de la memoria semántica y el manejo de relevancias episódicas. Finalmente, se introducen los metadatos asociados a cada episodio y las limitantes del diseño episódico desarrollado.

4.3.1. Formato y representación

Mensaje ROS: Se decidió utilizar mensajes de ROS para la representación de episodios. Esto se justifica en la amplia gama de mensajes preconstruidos que provee, los que permiten generar un mensaje adecuado a cualquier necesidad. Además, al utilizar este sistema de mensajes se evita definir una formato de información particular para la representación episódica, y la comunicación de episodios mediante la API ROS del servidor se simplifica. Más aún, como se explica en la Sección 3.2, la interfaz ROS para MongoDB permite almacenar cualquier estructura de datos, mientras esta esté encapsulada en un mensaje ROS.

Unicidad: El requisito de sistema RS08 exige que cada episodio sea único. Para esto, el mensaje ROS debe contener un campo que identifique al episodio. El servidor debe encargarse de que cada nuevo episodio cuente con un identificador único.

Tags: Además, para simplificar la búsqueda y reconocimiento de episodios, se incluye un campo de *tags*. Este corresponde a un vector de *strings* y permite almacenar descripciones breves sobre la naturaleza del episodio. Por ejemplo, un episodio podría contener los siguientes *tags*: "cumpleaños", "fiesta", "atender invitados" y "torta".

4.3.2. Árboles episódicos

Anidamiento episódico: A partir de los requisitos de sistema RS09 y RS11, se debe soportar el concepto de anidación episódica. Esto permite expresar cualquier episodio en términos de 1 o más subepisodios más específicos, como se presenta en la Figura 4.2. Para ello, se definen los *árboles episódicos*, utilizando una estructura de datos de árbol: los nodos externos (en adelante, *hojas*) contienen todo episodio que el sistema identifica como no divisible, mientras que cada padre es una representación menos específica de lo sucedido. Los hijos pueden ser ordenados temporalmente por su instante de inicio, para obtener una representación de su padre.

Temporalidad: Para el árbol episódico es fundamental que cada episodio padre inicie antes que cualquiera de sus hijos, y termine después que todos ellos. Lo anterior es presentado en el diagrama de la Figura 4.2, donde el episodio A cubre el rango temporal de sus

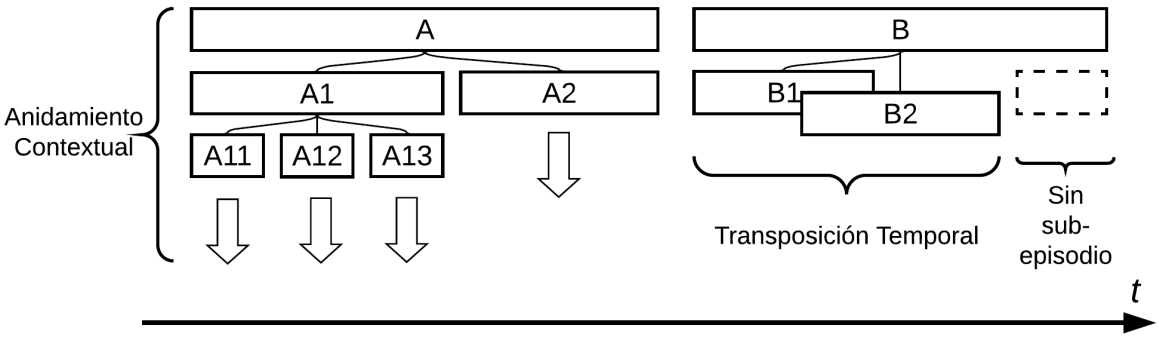


Figura 4.2: Diagrama temporal ejemplificando nociones de anidamiento y transposición episódica. Cada cuadro es un episodio identificado por el nombre indicado (*A, B, A1*, ...). El tiempo transcurre horizontalmente hacia la derecha, por lo que *A2* ocurre después de *A1*, y *B2* inicia después que *B1*, pero ocurren simultáneamente por un periodo. Los episodios pueden ser anidados indefinidamente.

hijos. Además, el diagrama ejemplifica un caso particular, denominado *transposición temporal episódica*, donde dos episodios (*B1* y *B2*) ocurren de manera simultánea, cada uno representando eventos distintos. Finalmente, el diagrama muestra un caso de borde, donde un padre puede tener lapsos de tiempo sin hijos asociados (cuadro con bordes punteados bajo el episodio *B*).

Estrategia de almacenamiento: Ya que un padre puede ser representado lógicamente por su conjunto de hijos, se decide utilizar una metodología de almacenamiento que no genere redundancia de información. Solo las hojas del árbol son las encargadas de manejar la información semántica del episodio (datos para el campo *What*), lo que potencialmente corresponde a una gran cantidad de datos. Asimismo, las hojas son encargadas de manejar la mayor cantidad de información episódica que sea posible (datos para los campos *When*, *Where* y las relevancias episódicas). Por lo tanto, los nodos internos del árbol (en adelante, *nodos*) manejan poca información y la mayor parte de sus datos puede ser calculada a partir de sus hijos y de manera automática por el servidor. Las estrategias de almacenamiento y reconstrucción para cada tipo de dato serán explicadas en las siguientes secciones.

Noción de contexto: Es importante enfatizar que no basta que un episodio esté embebido temporalmente en otro para ser considerado un hijo. Pues a pesar de cumplir la condición de temporalidad, puede ocurrir el caso de que ambos episodios sean simultáneos (ver RS12), pero estén ligados a un contexto distinto. Por ejemplo, durante los días de la competencia RoboCup, el robot podría ser utilizado para una tarea de uno de los estudiantes; a pesar de que los eventos son simultáneos, se puede decir que el último pertenece a otro contexto y debe ser almacenado en un árbol distinto. Esta noción de contexto es importante, y es la razón por la que el servidor debe manejar una cantidad indefinida de árboles episódicos, cada uno identificado por su raíz. Debido lo anterior, cada episodio debe ser almacenado indicando quien es el padre asociado.

4.3.3. Contexto temporal: *When*

Requisitos: De acuerdo al requisito RS03, los episodios deben almacenar información que indique su contexto temporal (*When*), es decir, cuándo sucedió. Para esto, la única información de interés a almacenar son los instantes de tiempo que indican el fin y el inicio del episodio.

Estrategia de almacenamiento: En el caso de las hojas, basta utilizar los instantes de tiempo obtenidos al iniciar y finalizar el episodio. Sin embargo, para el caso de los nodos, se pueden seguir dos estrategias: Utilizar los instantes de tiempo iniciales y finales percibidos, o calcular automáticamente los tiempos de inicio y fin para ajustarse a los hijos. A pesar de que la segunda opción suena razonable, está asumiendo que siempre existirá un hijo para cada instante de tiempo de vida del padre, lo que puede no ser cierto. Por lo tanto, se deben manejar cuidadosamente los valores de tiempo de los padres, para asegurar que no son sobreescritos por sus hijos, y para asegurar que los padres siempre contienen temporalmente a cada hijo.

4.3.4. Contexto espacial: *Where*

Requisitos: De acuerdo al requisito RS04, los episodios deben almacenar información que indique su contexto espacial (*Where*), es decir, dónde sucedió. A continuación se explican las decisiones de diseño consideradas.

Desplazamiento durante episodios: En primer lugar, dependiendo de la duración del episodio, el robot puede haber estado en más de un lugar durante ese periodo. Esto no es válido solo para episodios nodo, sino que también para hojas, cuando estas representan acciones de desplazamiento del robot. Luego, es importante que cada episodio registre cada uno de los lugares en los que ha estado el robot. El listado de lugares debe ser almacenado ordenadamente y con la hora asociada a cada desplazamiento, lo que es importante por dos motivos: permite calcular la secuencia de movimientos para los padres, y permite reconstruir la ruta del robot durante cada episodio. Para ello, la posición del robot puede ser consultada periódicamente en busca de cambios, para almacenar solo los instantes que indiquen movimiento.

Estrategias de almacenamiento: La información espacial puede ser manejada de dos maneras: se pueden almacenar los nombres de los lugares en donde el robot estuvo durante el episodio, o se puede almacenar la ubicación precisa del robot durante el episodio, mediante posiciones relativas a un sistema coordenado predefinido. La decisión sobre cual sistema utilizar depende mucho del contexto en el cual funciona el robot. Por ejemplo, en el caso de Bender, hay ocasiones en donde solo es posible conocer su ubicación respecto a un sistema coordenado, mientras que en otras oportunidades, solo se tiene conocimiento verbal de esta. Luego, ya que no existe un consenso sobre que opción es la mejor, y para evitar limitar la usabilidad del sistema LTM, se decide dar soporte a ambas alternativas.

Estrategia 1 - Descripción: Los nombres de lugares en los que estuvo el robot durante el episodio pueden ser almacenados utilizando vectores de *strings*. Considerando el ambiente

en donde puede desempeñarse un robot de servicio doméstico, se decide almacenar esta información en dos niveles de profundidad. El primer nivel corresponde a una descripción específica de la ubicación del robot (e.g., "Comedor"), mientras que el segundo sirve para almacenar el nombre del área donde se encuentra la primera ubicación (e.g., "Casa de John"), junto a otras de similar orden semántico. En el caso de los episodios nodo, se ha decidido calcular automáticamente toda esta información a partir de sus hijos.

Estrategia 2 - Coordenadas: Similarmente al caso anterior, las coordenadas de lugares son almacenadas en vectores de puntos, junto con su hora y sistema de coordenadas asociados. Además, para cada punto se debe almacenar el nombre del mapa en donde se define el sistema coordenado. Ya que el trabajo está orientado a robots de servicio doméstico, se ha decidido implementar solamente ubicaciones en 2 dimensiones, es decir, cada punto solo considera 2 valores numéricos. En el caso de los padres se sigue la siguiente estrategia: las posiciones son obtenidas a partir de sus hijos, se agrega un campo para almacenar la envoltura convexa de todas las posiciones, y se agrega un campo indicando el centroide de esta. La envoltura convexa es agregada por motivos prácticos, pues puede ser utilizada para visualización y análisis del árbol episódico.

Limitantes: En la Sección 4.3.10 se describen dos limitantes del diseño realizado: la posible redundancia de información, y la utilidad de esta.

4.3.5. Memoria semántica: *What*

Requisitos: De acuerdo a los requisitos RS01, RS02, la memoria semántica (*What*) ligada a un episodio permite describir "qué" sucedió en él. Esta debe poder contener cualquier dato, por lo que no es posible establecer de antemano qué información y estructura debe ser utilizada, sino que esto debe poder ser definido por el usuario. El requisito RS07 (regla de flexibilidad episódica) establece que cada episodio debe permitir crear nuevos datos semánticos o actualizar los ya existentes. El requisito RS10 (regla de perspectiva episódica) exige que al leer un episodio, este pueda describir sus datos semánticos asociados, de acuerdo a la misma perspectiva que se tenía de ellos cuando el episodio fue ejecutado.

Streams y entidades: La memoria semántica maneja entidades conocidas por el robot e información sobre ellas. Por ejemplo, se puede definir la entidad "persona" y almacenar datos como su nombre, nacionalidad o la fecha de la última interacción registrada. Sin embargo, existe un tipo de información que no está asociada a alguna entidad en particular, correspondiente a datos en forma de flujos de información (*streams*). Estos permiten registrar secuencias de imágenes, sonido y otros conceptos basados en flujos de datos percibidos por el robot, especialmente los obtenidos directamente desde sus sensores. Otra característica de los *streams*, es que a diferencia de las entidades, representan datos que no se desea actualizar posteriormente, pues almacenan una fotografía de lo percibido en el instante, la que generalmente es invariable. Entonces, se decide separar la memoria semántica en dos componentes, *streams* y entidades, lo que permite diseñar estrategias de procesamiento adecuadas al funcionamiento de cada uno.

Generalización: Ya que los contenidos a almacenar deben poder ser definidos por el

usuario, se establecen las siguientes decisiones de diseño. En primer lugar, el usuario debe definir qué información utilizará, mediante uno o más mensajes de ROS. Segundo, a partir del requisito RS18, debe haber una conexión bidireccional entre los mensajes y los episodios, esto es válido tanto para *streams* como entidades:

- Mensajes deben tener una referencia al identificador del episodio relacionado.
- Mensajes deben definir un campo para almacenar su identificador único, el que debe ser utilizado por el episodio.

Sistema de plugins: Se decidió utilizar un sistema basado en plugins para manejar los mensajes ROS definidos por el usuario. De esta manera, se delega al usuario la implementación de los algoritmos para adquisición de la información semántica y procesamiento de esta. El diseño del servidor LTM en la Sección 4.5 especifica qué funcionalidades deben ser implementadas por plugins destinados a *streams* o entidades.

Datos semánticos - Streams: Los *streams* corresponden a flujos de datos generalmente invariables, relacionados a un episodio según sus tiempos de inicio y fin. Existen 3 metodologías para almacenar los *streams*, con ventajas y desventajas. En cada caso, no se requiere almacenar datos cuando no hay episodios activos:

1. Cada *stream* contiene solo 1 dato semántico, por lo que no tiene duración y es asociado a un instante de tiempo. Esta estrategia es ideal funcionalmente, pero ya que la tasa de mensajes a almacenar puede ser alta, en la práctica esta solución es costosa en términos de consultas a la base de datos.

2. Mantener 1 *stream* por cada episodio, con tiempos de inicio y fin idénticos a los del episodio relacionado. Tiene la desventaja de almacenar información duplicada cuando existe traslape temporal de episodios.

3. Similar a la estrategia anterior, pero subdividiendo el *stream* cuando hay traslape de episodios. Evita la duplicación de información, pero complica la mantención de la base de datos.

Para acotar el trabajo, y bajo la suposición de que en la práctica existen pocos episodios traslapados, se escoge la segunda estrategia. Entonces, cada episodio puede estar asociado a un *stream* de cada tipo, y a su vez, cada *stream* puede contener uno o más datos. El servidor debe proveer notificaciones para indicar el inicio y fin de un episodio, mientras que el usuario debe definir su estrategia para adquirir datos durante ese lapso de tiempo. Por ejemplo, en el caso de un *stream* de imágenes representando la visión del robot, y cuyo fin solo sea de visualización a futuro, el usuario podría almacenar una imagen cada 3 segundos.

Según la cantidad de plugins a utilizar y la periodicidad de recopilación de información, el espacio de disco requerido para almacenar los mensajes puede ser agotado rápidamente. Para solucionar esto hay 3 alternativas. En primer lugar, se puede expandir la memoria disponible, lo que no es una solución real. En segundo lugar, está el "olvido" de información, mediante la eliminación de datos antiguos o de baja relevancia episódica (posiblemente previo respaldo en otra máquina) o eliminar datos intercaladamente. La tercera opción es la degradación

de información, para lo que cada usuario debe implementar un algoritmo que disminuya el tamaño de sus mensajes (e.g., disminuyendo la resolución o cantidad de las imágenes del episodio). Ya que las dos primeras opciones pueden ser ejecutadas manualmente por el usuario, se ha decidido utilizar la degradación de mensajes. Esta estrategia puede ser aplicada automáticamente a mensajes antiguos y de baja relevancia episódica.

Datos semánticos - Entidades: El usuario debe definir cada tipo de entidad disponible a través un mensaje ROS. Cada tipo (e.g., "persona") puede tener muchas instancias (e.g., "Carl" y "Lenny"). A diferencia de los *streams*, las entidades manejan datos modificables, lo que introduce dos condiciones:

- Su modelo de datos debe soportar el requisito RS07 al crear nuevas instancias o modificar una ya existente.
- El modelo de datos debe soportar el requisito RS10, por lo que cada episodio debe almacenar referencias a las instancias modificadas y los cambios aplicados.

Las instancias de una entidad y sus modificaciones no están ligadas a un episodio particular, sino que a los episodios activos durante el registro de la modificación. Entonces, es posible registrar cambios a entidades a pesar de no haber episodios activos.

4.3.6. Relevancia generalizada

Requisitos: De acuerdo a los requisitos RS19-RS20-RS21, los episodios deben almacenar información de relevancia episódica. Estos indicadores son importantes al momento de buscar episodios, para poder ordenarlos de acuerdo a una medida de importancia, y así tener una pista sobre en cuales enfocar la atención. El sistema debe soportar, al menos, la noción de relevancia histórica y emocional, considerando que en un futuro se podrían agregar otros indicadores de relevancia. También, el sistema debe proveer un indicador generalizado, capaz de representar la relevancia global del episodio, mediante un solo indicador numérico que unifique las subrelevancias.

Estrategia de almacenamiento: Entonces, se define la siguiente estrategia de almacenamiento. Cada tipo de indicador debe proveer un valor numérico en el rango $[0, 1]$, que represente la importancia del episodio de acuerdo a su perspectiva. Un valor de 0 significa que el episodio no es relevante, mientras que el valor 1 es utilizado para indicar que el episodio es muy importante. Utilizando el formato anterior, se construye el indicador de relevancia generalizada a partir de todas las subrelevancias definidas para el episodio. Este indicador debe ser actualizado cada vez que una subrelevancia sea modificada o ingresada.

Cómputo: Este trabajo solo considera dos subindicadores, histórico y emocional, por lo que se decidió utilizar la metodología propuesta por Dood et al. y explicada en la Sección 2.4.2.

4.3.7. Relevancia emocional

Requisitos: El requisito de sistema RS19 exige el manejo del concepto de relevancia emocional, como un medio para almacenar las emociones "que siente" el robot durante un episodio, en conjunto a un indicador de la intensidad de estas.

Abstracción del sistema emocional: Según se revisó en la Sección 2.4.2, existen diversos sistemas para la generación de emociones y no existe una metodología única para su implementación: Cada sistema es alimentado con datos diferentes, utilizan algoritmos distintos y proveen salidas distintas. Debido a la generalidad esperada para el trabajo, se propone la siguiente estrategia para la abstracción del sistema emocional a utilizar, que permite delegar la implementación al usuario:

- Se define un conjunto de 8 emociones base, a partir de las emociones primarias propuestas por Plutchik (ver Sección 2.4.2).
- Se debe almacenar el nivel de intensidad percibido durante un episodio, para cada una de las 8 emociones.
- Se utilizará un sistema de plugins, donde el usuario debe implementar el mapeo entre las salidas de su sistema emocional, a la combinación de emociones base disponibles para un episodio.

Estrategia de almacenamiento: Las 8 emociones definidas deben seguir el estándar definido para la representación de relevancias, con valores en el rango $[0, 1]$ indicando la intensidad asociada. La relevancia emocional de cada episodio se asocia a la emoción cuya intensidad sea la de mayor valor. Para el caso de episodios nodo, el registro emocional es calculado automáticamente a partir de los hijos, almacenando los valores máximos para cada emoción.

Metadatos: Además, para simplificar la introspección de episodios a futuro, se decide agregar metadatos que indican el software emocional utilizado y su versión. Es importante enfatizar que cada metadato solo debe ser utilizado para introspección y no tienen influencia en el cálculo de las relevancias.

4.3.8. Relevancia histórica

Requisitos: El requisito RS20 exige el manejo del concepto de relevancia histórica, como un medio para indicar la importancia de un episodio, según su edad en la base de datos.

Estrategia de almacenamiento: Siguiendo el estándar definido para representación de relevancias, se utiliza un valor numérico en el rango $[0, 1]$, que indica el estado de envejecimiento del episodio. Este valor siempre debe ser inicializado en 1, para decaer hacia 0 con el paso del tiempo.

Actualización automática: El valor de la relevancia debe ser manejado automáticamen-

te por el sistema LTM. Se propone disminuir el indicador, según el algoritmo de decaimiento presentado por Dood et al. (ver Sección 2.4.2). Ya que se espera almacenar una gran cantidad de episodios, se propone discretizar la función de decaimiento. Los episodios deben ser actualizados en lapsos cada vez más espaciados en el tiempo (e.g., primero cada 1 día, luego 1 vez a la semana, cada 1 mes, cada 1 año), lo que permite acotar el esfuerzo computacional dedicado a las actualizaciones.

4.3.9. Datos para introspección

Metadatos: El último conjunto de datos considerado para un episodio es denominado *Metadatos*. Estos no están ligados a algún requerimiento de software, por lo que no son formalmente necesarios, sin embargo, permiten almacenar datos útiles para la implementación LTM, y sirven para conocer información sobre el software utilizado durante el almacenamiento de cada episodio. Los *Metadatos* tienen como finalidad ser de utilidad al resolver problemas relacionados al sistema LTM, y para cuando los usuarios deseen conocer el contexto de software sobre el cual fue generado el episodio.

4.3.10. Limitantes y trabajo futuro

A continuación se presenta un conjunto de limitantes conocidas del diseño episódico desarrollado, las que pueden ser consideradas como parte del trabajo futuro.

Ausencia de hijos: Este problema fue explicado parcialmente en la Sección 4.3.3, y aparece con padres cuyo rango temporal no es cubierto completamente por sus hijos. Esto puede suceder cuando el usuario decide no almacenar algunos episodios hoja, pues los considera poco relevantes. El efecto de esto, es que los padres podrían contar con información incompleta al calcular datos a partir de sus hijos. Se puede argumentar que, ya que el usuario considera que los subepisodios eran irrelevantes, entonces la información perdida también lo es. Sin embargo, por motivos de introspección, puede ser de interés almacenar los datos de *streams* asociados a los periodos temporales perdidos.

Episodios de larga duración: A pesar de que el diseño permite episodios de duración indefinida, en la práctica es raro que el robot funcione por periodos prolongados. Luego, los episodios cuyo fin es dar un contexto temporal a un árbol episódico no pueden ser almacenados directamente (e.g., cuando el robot está participando en una competencia de 4 días de duración). A pesar de lo anterior, el diseño permite la introducción manual de episodios de larga duración, sin tener que mantener en funcionamiento el robot durante todo el periodo.

Memoria emocional detallada: Ya que el diseño apunta a ser lo suficientemente genérico, la información emocional que se puede exigir es acotada. El formato permite almacenar los datos justos para la asignación de relevancias, pero a la vez, limita las consultas posibles sobre las emociones del robot en un episodio. Particularmente, en episodios de larga duración sería interesante conocer la secuencia (ordenada) de emociones, o las causas de estas. A pesar de que el diseño no almacena información al respecto, si permite definir una forma alternativa

de almacenar tales datos. Se puede crear una entidad "Robot" con campos asociados a sus emociones, para así, llevar un registro detallado de lo que se necesite.

Ubicación del robot duplicada o no útil: Se cree que el formato propuesto para el campo *Where* se ajusta a los requerimientos de un robot doméstico, sin embargo, se identifican dos problemas. Por un lado, la naturaleza de la ubicación del robot no depende de un episodio, sino que es información desacoplada de este, similar al funcionamiento de una entidad. Esto genera duplicación de la información en caso de traslape episódico. Por otro lado, se cree que el formato se adapta a los requerimientos para Bender, pero pueden haber usuarios que requieran almacenar otra información (e.g., GPS) o utilizar mayores niveles de profundidad semántica (e.g., Chile → Santiago → Laboratorio → Mesón A). En un trabajo futuro se podría modificar el diseño, para manejar el campo *Where* de manera similar al caso de las entidades, donde cada usuario defina los datos que desea almacenar y su metodología de adquisición mediante un plugin asociado, y asimismo, evitando la redundancia de información.

4.4. Diseño del Modelo de Datos

En esta sección se presentan las decisiones sobre el diseño del modelo de datos a utilizar para el trabajo. En primer lugar, se revisan las consideraciones sobre la base de datos escogida: *MongoDB*. Luego, se describe el manejo de mensajes episódicos y sus datos semánticos asociados: *streams* y entidades.

4.4.1. Base de datos

MongoDB: De acuerdo al diseño de los mensajes episódicos, la implementación del sistema LTM requiere el manejo de datos no conocidos de antemano, sino que representados mediante mensajes de ROS que debe definir el usuario. Además, el sistema debe permitir modificar la representación de tales mensajes a futuro y soportar el almacenamiento de datos binarios. Por lo tanto, se decidió utilizar la base de datos no relacional *MongoDB*, presentada en la Sección 3.2.

La decisión sobre el uso *MongoDB* se basa en tres razonamientos principales. Primero, *MongoDB* ya es de uso extendido en la comunidad ROS, por lo que ya ha sido probada y está disponible en muchos robots de servicio domésticos, evitando agregar dependencias extra a tales trabajos. Además, la interfaz ROS para esta es considerada estable, al menos hasta el año 2023. Finalmente, *MongoDB* se ocupa actualmente en el robot Bender. Así, se minimizan los problemas de dependencias y compatibilidad a futuro.

Colecciones: Debido al funcionamiento de `warehouse_ros_mongo`, visto en la Sección 3.2, cada mensaje ROS debe ser asociado a una colección distinta. Particularmente, una colección será destinada al almacenamiento de episodios, mientras que cada mensaje definido por el usuario será asociado a una colección diferente. En primera instancia, la base de datos no

relacional permitiría almacenar todos los mensajes en la misma colección, pero la separación de cada tipo de dato en colecciones diferentes es conveniente por las siguientes razones:

- Permite optimizar las consultas simples a la base de datos, que no requieran acceder a toda la memoria semántica asociada.
- Se evita propagar problemas al registro de episodios, cuando se presenten fallos en los plugins de memoria semántica implementados por el usuario.
- Ya que el sistema debe soportar una cantidad indefinida de mensajes ROS elegidos por el usuario, se evita saturar el registro de episodios con mensajes que, una vez agrupados, superen el límite de tamaño (16MB) permitido a cada elemento de una colección.

Modificación de mensajes semánticos: El requisito RS14 especifica que el servidor debe permitir modificar la representación de los mensajes episódicos almacenados. Esto puede suceder porque el usuario decide agregar/quitar campos al mensaje, o porque una dependencia del mensaje fue actualizada. La implementación debe soportar la migración desde la estructura antigua a la nueva.

4.4.2. Colección de episodios

Todos los episodios, definidos por su mensaje ROS, serán almacenados en la colección `episodes`. De acuerdo a los requisitos RS05 y RS22, cada episodio debe ser indexado al menos por los campos *What*, *When*, *Where*, las relevancias emocionales y sus campos internos. En el caso de *What*, las entidades y *streams* también deben ser indexados, y queda a criterio del usuario cuales campos son de interés para cada concepto.

4.4.3. Colecciones de streams

Colección: Según se explicó anteriormente, cada *stream* definido por el usuario es asociado a una colección de *MongoDB* propia. El nombre de cada colección lleva el prefijo `stream:`, seguido de un identificador definido por el usuario.

Estrategia: Por cada episodio a almacenar se genera un *stream* de cada tipo disponible, para ser almacenado en su colección respectiva. Esto se ejemplifica en la Figura 4.3, mediante un diagrama comparando la relación entre un episodio y los datos semánticos percibidos, de acuerdo al diseño episódico descrito en la Sección 4.3.5.

Indexado: Por lo general, los campos de un *stream* no requieren ser indexados, ya que es conveniente almacenarlos de manera binaria. Sin embargo, durante la implementación del plugin asociado, el usuario puede decidir añadir metadatos para indexar campos convenientes. La utilidad de esto, es que permitirá agregar nuevos términos de comparación para las consultas a la base de datos. Luego, se tienen dos formas de acceder a un *stream* en particular; mediante su mensaje episódico asociado, o a través de los metadatos definidos por el usuario.

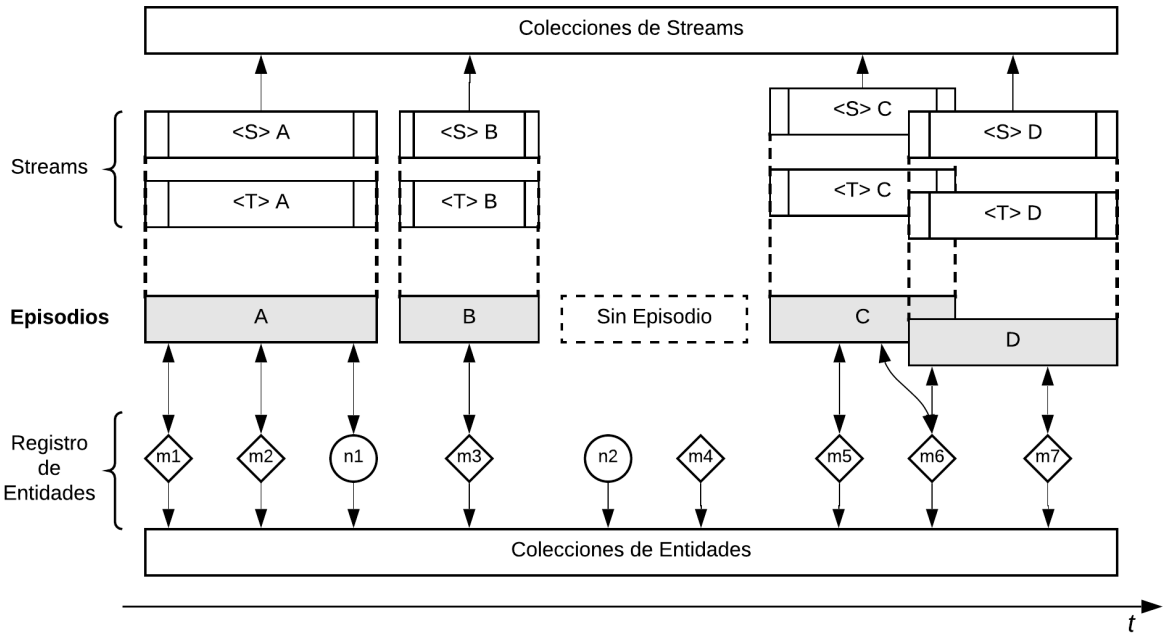


Figura 4.3: Funcionamiento de *streams* y entidades respecto a los episodios. `<S>` y `<T>` hacen referencia a 2 tipos de *streams*, mientras que $m_1 \dots m_7$ (rombos) y n_1, n_2 (círculos) son notificaciones sobre cambios a las entidades m y n. Un *stream* de cada tipo es asociado a cada episodio, incluso en caso de transposición. Cada notificación sobre entidades es asociada a todo episodio activo. En caso de no haber episodio, no se almacenan *streams*, pero si se registran los cambios en entidades.

Tamaño del mensaje: El límite de tamaño para cada mensaje ROS a almacenar es de $16MB$, lo que impone una cota superior a la cantidad de datos que pueden ser almacenados por cada episodio. Según se describe en la Sección 3.2, *MongoDB* provee estrategias de almacenamiento para sobrepasar este límite.

4.4.4. Colecciones de entidades

Colección: Cada mensaje ROS definido por el usuario para representar una entidad de la memoria semántica, es asociado a una colección de *MongoDB* propia. El nombre de cada colección lleva el prefijo `entity:`, seguido de un identificador definido por el usuario.

Flexibilidad: El modelo debe considerar un identificador único para indicar la instancia de cada entidad. Así, cuando un episodio registra actualizaciones de una instancia particular, modifica su entrada en la colección correspondiente.

Perspectiva: Para soportar este concepto se requiere poder reconstruir el estado de una instancia en cada instante de tiempo desde su creación. Este es un requisito común para una base de datos, denominado *Audit Trail*. La solución propuesta para el problema es la siguiente:

- Por cada entidad (mensaje ROS) se deben mantener 2 colecciones en la base de datos.
- Las primera colección almacena un historial de las modificaciones a cada instancia de la entidad. Cada entrada debe indicar el tiempo de ocurrencia, tener referencias a los episodios asociados e indicar los cambios modificados.
- La otra colección almacena el estado actual de cada instancia de la entidad, lo que es equivalente a aplicar su historial de cambios desde el comienzo. Esta colección tiene el propósito de agilizar consultas que solo requieran los conocimientos actuales de cada instancia.

Ya que `warehouse_ros_mongo` solo permite almacenar un tipo de mensaje ROS por colección, la destinada a almacenar los cambios es subdividida en dos colecciones:

- La primera almacena un mensaje ROS del mismo tipo que la entidad, y sirve para indicar los cambios registrados. Su nombre utiliza el prefijo `entity:` y el sufijo `.trail`.
- La segunda almacena un mensaje ROS con los metadatos necesarios para mantener el historial. Su nombre utiliza el prefijo `entity:` y el sufijo `.meta`.

Estrategia: Con la finalidad de mantener la consistencia del historial de entidades, se deben almacenar todos los cambios recolectados para una entidad, a pesar de que no haya un episodio activo. Sin embargo, cada registro debe tener referencias a los episodios activos durante el instante de su recolección, y a su vez, cada episodio debe tener referencias a tales registros. Lo anterior es ejemplificado en el diagrama de la Figura 4.3.

Indexado: Se delega al usuario la definición de los campos de interés para el indexado de su mensaje. Estos campos estarán disponibles para realizar consultas episódicas al servidor. Sin embargo, para enriquecer las búsquedas de episodios, los metadatos del registro histórico deben tener vectores con los nombres de los campos modificados, campos creados, y campos eliminados. Los 3 vectores deben estar indexados.

4.4.5. Consultas al modelo

De acuerdo a los requisitos RS06 y RS17, el modelo debe soportar operaciones CRUD sobre los episodios y mensajes semánticos almacenados. Las operaciones deben permitir filtrar los episodios mediante condiciones de {igualdad, mayor/menor que, pertenencia a arreglos} sobre los campos indexados. Se debe dar soporte para comparaciones entre los siguientes tipos de dato primitivos de ROS: {`bool`, `int`, `float`, `string`} y condiciones de pertenencia a arreglos de esos tipos.

A continuación se describe la estrategia de procesamiento para las consultas CRUD que debe soportar el sistema. En cada caso se enfatizan las reglas a seguir para mantener la consistencia del modelo de datos.

4.4.5.1. Operaciones de inserción (Create)

Episodios: La inserción de episodios se hace en un proceso de dos etapas:

1. Al inicio del episodio se declara la creación de este al servidor, el que proveerá un identificador disponible para su registro. Si el usuario indica que se trata de un episodio hoja, el servidor notifica a los plugins relacionados para que recopilen la información episódica y semántica requerida.

2. Al finalizar el episodio, el usuario debe indicar la referencia al padre. Este proceso notifica a los plugins para que finalicen la recopilación de información y se almacene el mensaje episódico y el *stream* asociado.

Tras la inserción del episodio se deben actualizar todos los padres, hasta llegar a la raíz.

Los requisitos no lo exigen, pero en caso de dar soporte para la inserción de episodios pasados, no se deben ejecutar los plugins episódicos ni semánticos. La información sobre relevancia emocional y posición debe ser entregada por el usuario. El *stream* debe ser insertado manualmente por el usuario. Se deben registrar automáticamente las entidades que calcen con el lapso de tiempo designado. Análogamente, se debe actualizar el árbol episódico, hasta llegar a la raíz.

Streams: Para la inserción de *streams* se debe indicar el episodio asociado e incluir la referencia del *stream* al episodio. En caso de ya existir un *stream* del mismo tipo asociado al episodio, se sobrescribe la referencia y se elimina el antiguo de su colección. Además, es necesario que los tiempos de inicio y fin del mensaje estén dentro de los límites temporales definidos por el episodio.

Entidades: Los requisitos RS07 (flexibilidad) y RS10 (perspectiva) establecen dos operaciones de inserción sobre entidades: creación de una nueva instancia y la modificación de una instancia ya existente, mediante la inserción de un nuevo registro en el historial. Ambas operaciones son requeridas para registros obtenidos en el instante actual. Ya que la creación de una nueva instancia se puede ver como un caso particular de modificación, solo se detallará la estrategia del último caso:

- El nuevo registro debe indicar la fecha de creación, un identificador único del historial y una referencia a la instancia que modifica/crea.

- Se debe modificar/crear la instancia en la colección que mantiene los datos actualizados.

- La inserción del registro solo debe almacenar los datos que fueron modificados, agregados o eliminados. La información que se mantuvo estática no debe ser registrada.

- Se deben agregar referencias bidireccionales a los episodios cuyo lapso de vida contenga al registro.

No es necesario dar soporte para crear nuevos registros del historial de entidades, asociados a un tiempo pasado. En caso de hacerlo, se debe seguir una estrategia similar a la

anterior, pero además modificando recursivamente los registros adyacentes (temporalmente), para mantener la consistencia del historial.

API ROS: Las operaciones de inserción de *streams* y entidades son manejadas automáticamente por el sistema LTM, por lo que no es necesario proveer una API para ello. Por otro lado, si se requiere una API para las operaciones de registro e inserción de episodios.

4.4.5.2. Operaciones de lectura (Read)

Etapas de consulta: Para evitar sobrecargar el ancho de banda de red, con respuestas a consultas que retornen muchos episodios o datos semánticos, se diseñan las operaciones de lectura en 2 etapas. Primero, se obtienen los identificadores de los mensajes de interés mediante una condición que provee el usuario. Luego, se puede acceder a los episodios, *streams* y entidades mediante servicios ROS, encargados de retornar mensajes para los identificadores escogidos.

Filtro de mensajes: La búsqueda de mensajes según condiciones es considerada por el requisito RS06. Para evitar limitar las consultas que puede realizar el usuario sobre la base de datos y para simplificar el diseño, se decide utilizar el sistema de condiciones nativo de MongoDB. Así, es posible aplicar filtros complejos a las consultas, basados en todos los campos disponibles de cada documento. Sin embargo, las consultas solo pueden ser aplicadas a una colección a la vez. Algunas de las condiciones que provee MongoDB son: igualdad, mayor qué (y derivados), pertenencia a un arreglo y agrupamiento lógico mediante condiciones *AND* y *OR*.

Episodios: Se debe proveer un servicio para filtrar episodios de acuerdo a una condición basada en los metadatos disponibles. El servicio debe retornar los identificadores de los episodios que calcen con la condición, junto a los identificadores de los *streams* y entidades asociadas. Además, se debe proveer un servicio para obtener episodios de acuerdo a su identificador.

Streams: De manera similar, se debe proveer un servicio para filtrar *streams* de acuerdo a una condición basada en sus metadatos. El servicio debe retornar los identificadores de los mensajes que cumplan la condición y sus episodios asociados. Además, se debe proveer un servicio para obtener *streams* mediante su identificador.

Entidades: En el caso de las entidades, el filtro debe poder ser aplicado a la colección de entidades actuales y al historial de cambios. La consulta al historial debe permitir filtrar mediante los nombres de los campos modificados, añadidos u olvidados en cada registro. En ambos casos, se deben retornar los identificadores de las entidades o registros que calcen con la condición, sumado a los de los episodios asociados.

Además, se debe proveer un servicio para obtener una entidad según su identificador. Sin embargo, para cumplir con el requisito RS10 (perspectiva), la consulta debe permitir indicar una fecha, para retornar la entidad de acuerdo al conocimiento obtenido hasta el momento.

API ROS: De acuerdo a lo anterior, se debe proveer una API ROS para realizar consultas

con condiciones en el formato de MongoDB, para las colecciones de episodios, *streams* y entidades. Además, se deben proveer servicios ROS para obtener episodios, entidades y *streams* según sus identificadores.

4.4.5.3. Operaciones de actualización (Update)

Episodios: De acuerdo al requisito RS08, los episodios son únicos y solo tienen una instancia para ser aprendidos. Entonces, a pesar de ser útil, no es necesario proveer servicios para la actualización de episodios. En el caso de hacerlo, se deben modificar todos los nodos padres, hasta llegar a la raíz. En caso de modificar el intervalo de tiempo, se deben actualizar las referencias bidireccionales con las entidades ya existentes. Al acortar el intervalo de tiempo, se debe modificar el *stream* asociado, para eliminar mensajes sobrantes.

Streams: De la misma manera, ya que los *streams* hacen referencia a datos estáticos, no es necesario proveer esta funcionalidad para ellos. Solo tiene sentido actualizar el intervalo de tiempo del mensaje cuando el de su episodio relacionado sea modificado.

Entidades: Por otro lado, el requisito de flexibilidad RS07 exige que las entidades deban soportar operaciones de actualización. Estas ya son manejadas por el modelo de datos mediante inserciones en las 3 colecciones correspondientes a la entidad, por lo que no hay necesidad de proveer una API con esta funcionalidad.

En caso de proveer una API para actualizar mensajes del historial, las reglas son las siguientes: se deben modificar recursivamente los registros adyacentes del historial, para considerar los cambios; se debe actualizar la colección que mantiene la imagen actual de la instancia; se deben actualizar las referencias bidireccionales con los episodios que ocurrieron en ese lapso de tiempo.

API ROS: Como se revisó, no es un requisito proveer servicios ROS para la actualización de los episodios y la memoria semántica. De todas formas, tales funcionalidades son soportadas por el modelo de datos, y pueden ser consideradas como parte del trabajo futuro.

4.4.5.4. Operaciones de eliminación (Delete)

A partir de los requisitos de sistema, no es necesario proveer operaciones de eliminación de mensajes episódicos. Sin embargo, esta operación puede ser de utilidad para pruebas o liberación de memoria.

Episodios: En el caso de eliminar un episodio, se deben quitar sus referencias de las entidades y eliminar el *stream* relacionado. Además, se deben actualizar todos los padres hasta llegar a la raíz, para quitar la información asociada al hijo modificado.

Streams: Al eliminar un *stream* se debe quitar su referencia del episodio relacionado.

Entidades: Al eliminar algún mensaje del historial de una entidad, se deben quitar sus

referencias de los episodios relacionados, es decir, todo episodio cuyo periodo de vida contenga el instante asociado al mensaje. Se debe quitar el registro del historial y actualizar los historiales adyacentes para mantener su consistencia. Se deben recalcular los datos asociados a la entidad actual.

API ROS: No es necesario proveer una API ROS para la eliminación de información episódica o semántica. Sin embargo, tales funcionalidades son soportadas por el modelo de datos, y pueden ser consideradas como parte del trabajo futuro.

4.5. Diseño del Servidor LTM

A continuación se describe el diseño del servidor LTM, a partir de las consideraciones para el modelo de datos presentadas en las secciones anteriores. En primer lugar, se revisa el diseño operativo para el manejo de la memoria episódica: la adquisición de episodios, la recolección de información emocional y de ubicación, y el manejo de relevancias episódicas. Luego, se revisa el diseño de la memoria semántica (*streams* y entidades), el manejo de las colecciones, la API ROS y el flujo de información.

El sistema LTM diseñado se presenta en la Figura 4.4. A modo general, se indican las 3 etapas del flujo de información en el sistema LTM: recopilación desde la memoria STM del robot, manejo y almacenamiento por el servidor, y uso de la información por las aplicaciones del usuario. Los componentes del diagrama son explicados en detalle a continuación.

4.5.1. Memoria episódica

En esta sección se describe el diseño del mecanismo para la recolección de episodios, sus datos a partir de plugins y la actualización de relevancias episódicas.

4.5.1.1. Recopilación de episodios

Como se muestra en la Figura 4.4, el servidor depende de una API episódica para la adquisición de episodios. El diseño propuesto almacena episodios en dos etapas: registro y término, mediante una API ROS basada en servicios. Como lo indica el diagrama, el servidor se comunica con cada plugin para recopilar la información episódica y semántica necesaria.

Registro: En esta etapa, el usuario debe indicar el inicio de un nuevo episodio. Luego, el servidor se encarga de registrar el episodio, generando un identificador único, el que es retornado al usuario. Ya que el sistema debe soportar anidamiento y transposición, pueden haber muchos episodios registrados simultáneamente.

Durante el registro se debe indicar el tipo de episodio a introducir (hoja o nodo). Si el episodio es de tipo hoja, se notifica a todos los plugins, para que consideren el nuevo episodio

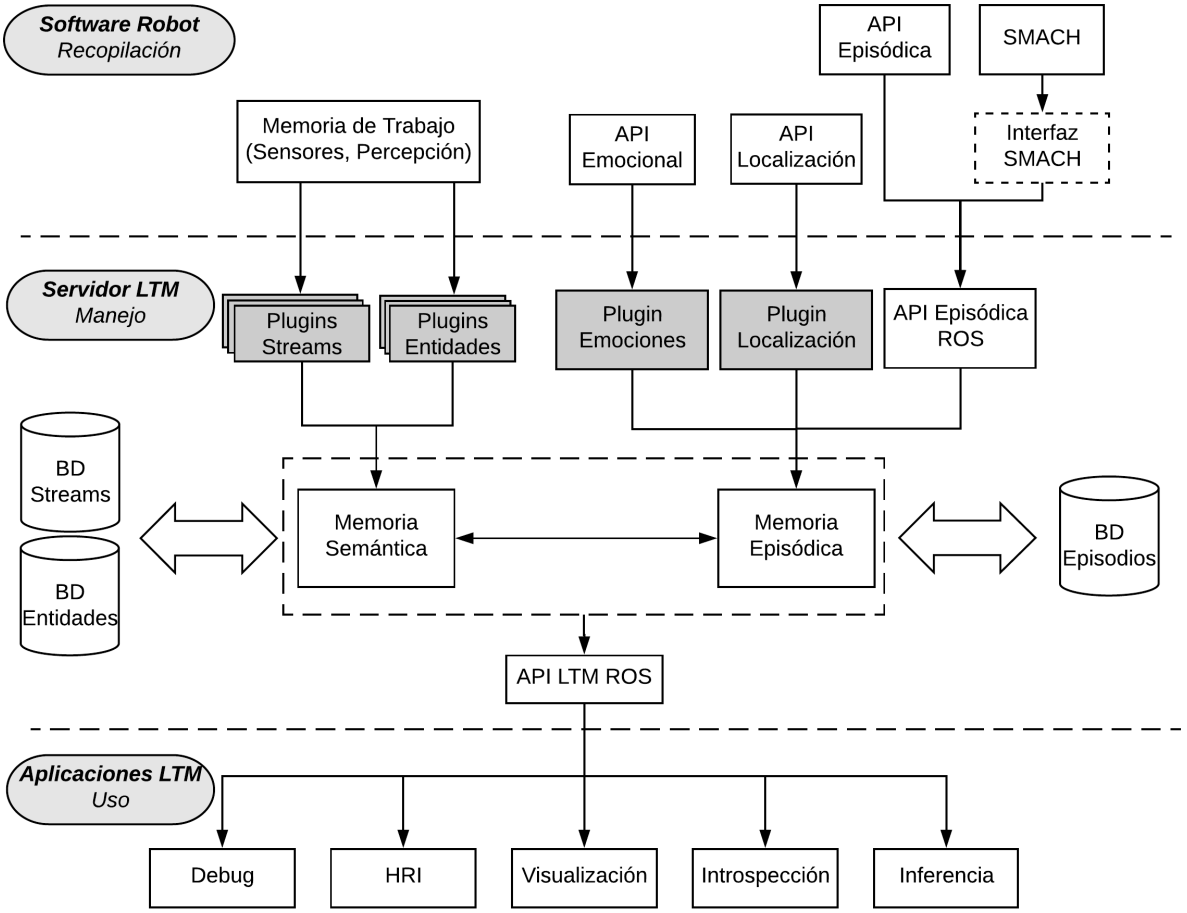


Figura 4.4: Diagrama con el diseño del sistema LTM y los módulos de software involucrados. La zona superior indica todos los módulos de software del robot necesarios para proveer información al sistema LTM. En plomo se presentan los componentes que debe implementar el usuario, que sirven como interfaz entre el robot y el servidor, para la recopilación de información. En la zona central se realiza el manejo y almacenamiento de la información episódica. En la zona inferior, el servidor se comunica mediante una API ROS con las aplicaciones LTM implementadas por el usuario.

como activo e inicien las tareas de recopilación de información. De acuerdo al diseño episódico, no se debe recopilar información para episodios nodo.

Término: Una vez que el episodio ha concluido, el usuario debe notificar al servidor sobre esto, indicando el identificador del padre, los *tags* asociados y los metadatos episódicos que desee almacenar.

Tras recibir la notificación, el servidor recolecta información episódica desde cada plugin, almacena el episodio en la base de datos y lo marca como inactivo. En caso de ya no haber episodios activos, cada plugin es notificado para dejar de recopilar información.

4.5.1.2. Plugin: *Where*

En cuanto a la recopilación de la ubicación del robot, dato episódico *Where*, el cómo se obtiene la información depende totalmente del usuario y la API de localización utilizada en el robot objetivo (ver diseño episódico en la Sección 4.3.4). Sin embargo, de acuerdo a la sección anterior, el plugin debe proveer las funcionalidades de registro y adquisición de información.

Se propone la siguiente estrategia para la implementación del plugin: recopilar periódicamente el posicionamiento del robot, mientras hayan episodios activos; cuando el servidor necesite la información, se debe proveer un listado con cada ubicación y su instante de tiempo, según el formato estudiado en el diseño episódico del campo *Where*; la lista debe estar restringida a los tiempos de inicio y fin del episodio en consideración.

4.5.1.3. Plugin: Emociones

Los requerimientos y estrategia de funcionamiento propuesta para este plugin son los mismos que para el plugin *Where*. La información recolectada debe cumplir con el formato propuesto durante el diseño episódico de la Sección 4.3.7.

4.5.1.4. Relevancia histórica y generalizada

La implementación del servidor debe contar con un proceso automático de actualización de relevancias episódicas, particularmente relevancia histórica y generalizada, según la estrategia descrita en la Sección 4.3.8. Ya que la cantidad de episodios a actualizar puede ser alta, se debe ejecutar el proceso de actualización cuando el robot no esté en funcionamiento, lo que se traduce en un umbral sobre el porcentaje de recursos disponibles en el sistema. El umbral puede ser escogido por el usuario.

4.5.2. Memoria semántica

En esta sección se describe el diseño de los mecanismos para la recolección de información sobre entidades y *streams*, a través de los plugins respectivos. Se revisan los requerimientos de cada plugin, las estrategias de implementación, el manejo de las colecciones de mensajes y la API ROS que provee el servidor.

4.5.2.1. Plugins: *Streams*

Requerimientos: De acuerdo al diseño para el campo *What*, revisado en la Sección 4.3.5, el primer requisito para la definición de un *stream* es la asignación de un mensaje ROS que lo represente. El mensaje además debe incluir un campo para almacenar metadatos episódicos

utilizados por el servidor (id del mensaje, id del episodio y lapso de tiempo). Luego, cada plugin debe indicar el mensaje ROS sobre el que operará.

Este tipo de plugins debe implementar 4 funcionalidades. Proveer un método para el registro de un nuevo episodio y uno para la recolección de la información por el servidor. Además, debe indicar cuales de los campos del mensaje deben ser considerados para ser indexados por la base de datos, pues el resto del mensaje se almacena como objeto binario. Finalmente, debe proveer un método para la degradación del mensaje, por motivos de olvido o escasez de recursos.

Estrategia: El servidor notificará a los plugins por cada nuevo episodio. La estrategia propuesta es mantener un *buffer* circular con información reciente, indexada por su instante de tiempo. Cuando el servidor solicite los datos de un episodio, el plugin genera un *stream* que contiene un listado con mensajes que sucedieron en el periodo del episodio. En el caso de no haber episodios activos, el plugin debe cerrar sus conexiones ROS, para restringir el uso de recursos.

Base de datos: La implementación del sistema LTM debe manejar automáticamente la colección de mensajes de cada *stream*, por lo que el usuario no debe tener acceso a la base de datos desde el plugin. Así, el servidor se encarga de mantener la consistencia de la información de forma centralizada, evitando delegar ese trabajo a cada usuario y limitando los posibles errores.

API ROS: De la misma forma, el servidor debe proveer automáticamente una API ROS para las operaciones CRUD de cada plugin. Así, se asegura la homogeneidad de la API y se evita agregar otra responsabilidad al usuario.

4.5.2.2. Plugins: Entidades

Requerimientos: De manera similar al caso de los *streams*, el usuario debe definir un mensaje ROS que represente la entidad de interés. El mensaje debe incluir un campo para almacenar metadatos episódicos utilizados por el servidor. Luego, cada plugin debe indicar el mensaje ROS sobre el que operará.

Ya que el registro de entidades es manejado de manera desacoplada respecto a los episodios (ver Figura 4.3), los métodos que debe proveer el usuario difieren de los utilizados para *streams*. Se deben implementar 4 funcionalidades:

- Se debe indicar cuales son los campos de interés a ser considerados para ser indexados por la base de datos, pues el resto es almacenado como un objeto binario.
- Debe proveer un método que permita identificar los campos que han sido modificados entre 2 entidades. El servidor lo utilizará para generar un registro de cambios y mantener el historial.
- Debe notificar al servidor por cada nuevo registro de una entidad. El servidor verificará los cambios y actualizará el registro de ser necesario.

- Debe proveer un método para actualizar una entidad, con los campos de otra más actual. El servidor utilizará esta funcionalidad para reconstruir entidades a partir del registro histórico.

Cada una de las funcionalidades requeridas es simple de implementar y mantener, las que se reducen a la asignación y comparación de los campos de la entidad definida por el usuario.

Estrategia: El plugin debe estar siempre en funcionamiento, recopilando información a partir de la STM del robot, a pesar de que no existan episodios activos. Cada nueva entidad, o cambio en una ya existente debe ser notificada al servidor, para actualizar los registros de la base de datos.

En este caso, el registro de nuevos episodios y la recolección de cambios del historial es manejada automáticamente por el servidor, sin requerir intervención del usuario. La estrategia a utilizar es la siguiente. En caso de haber episodios activos, el servidor crea un listado con los identificadores de las entidades modificadas y los registros correspondientes del historial. Cuando el servidor solicita la recolección, se entregan todos los cambios ocurridos en el rango temporal del episodio.

Base de datos: Análogamente al caso de los *streams*, el servidor debe manejar automáticamente la colección de mensajes de cada entidad, por lo que el usuario no debe tener acceso a la base de datos desde el plugin. De esta forma, el servidor se encarga de mantener la consistencia de los registros y el historial de entidades.

API ROS: Asimismo, el servidor debe proveer automáticamente una API ROS para las operaciones CRUD de cada plugin, asegurando la homogeneidad de la API y evitando agregar otra responsabilidad al usuario.

4.6. Diseño de Módulos Específicos para Bender

Según especifican los requisitos RS25 y RS26, el trabajo debe ser integrado al robot Bender. Para esto, se debe implementar un módulo que permita recolectar episodios desde SMACH, e implementaciones de cada plugin requerido por el servidor. A continuación se describe el diseño de los componentes a implementar: la interfaz para recolectar episodios, la posición del robot y sus emociones, junto a los plugins para la recolección de *streams* y entidades a partir de la memoria de trabajo de Bender.

Cada uno de los módulos presentados a continuación cumple una doble funcionalidad. En primer lugar, cumplen con el requisito del diseño de componentes orientados a robots de servicio doméstico, enfocados en el robot Bender. Además, sirven como ejemplo para el diseño e implementación de nuevas funcionalidades requeridas a futuro para otras plataformas.

4.6.1. Recolección de episodios: SMACH

SMACH: El módulo específico más importante a implementar corresponde al encargado de generar episodios a partir del funcionamiento del robot, para luego entregarlos al servidor LTM. Como se explica en la Sección 3.4, en Bender se utiliza la librería SMACH para la definición y ejecución de las máquinas de estado encargadas de encadenar rutinas simples, para generar comportamientos complejos.

Información episódica: A partir de SMACH se puede obtener parte del conocimiento episódico requerido para la definición de un episodio. Desde el punto de vista de sus transiciones, las máquinas de estado están estructuradas en forma de grafo, estableciendo un inicio y fin temporal para cada estado. Conceptualmente, las máquinas se estructuran en forma de árboles, donde cada máquina de estado puede ser contenida por otra, lo que representa el anidamiento episódico. Luego, SMACH provee una forma de conocimiento episódico, capaz de representar información temporal (*When*) y las relaciones de anidamiento episódico (referencias padre-hijo). Finalmente, ya que cada máquina de estado está asociada a una capacidad o contexto, se puede utilizar esa información para obtener los *tags* requeridos para marcar cada episodio.

Funcionamiento: La interfaz episódica diseñada tiene 2 etapas importantes. En primer lugar, durante la programación de la máquina, el desarrollador debe indicar que estados considera almacenar como episodio, en caso que estos sean ejecutados. Además, para cada estado marcado se debe indicar una lista de *tags* que describan brevemente el estado. La segunda etapa ocurre durante el funcionamiento de la máquina de estado, y realiza las tareas de registrar e insertar el episodio en el servidor.

Estabilidad de la implementación: La ejecución ininterrumpida de SMACH durante una rutina es crucial para el desempeño del robot. Si la librería deja de funcionar repentinamente o se detiene continuamente, el robot se detendrá o tomará mucho tiempo en realizar sus tareas, lo que durante una competencia puede tener pésimas consecuencias para el equipo. Es por esto, que la implementación de la interfaz entre el servidor LTM y SMACH debe ser estable. Particularmente, es de vital importancia que el funcionamiento de SMACH no se vea afectado por la implementación, independientemente de si esta tiene problemas en su ejecución.

Intrusividad: Por otro lado, la implementación debe minimizar las modificaciones a la librería SMACH y a las máquinas de estado ya implementadas. En primer lugar, si se modifica la librería SMACH, el equipo tendrá que agregar una copia de esta y mantenerla a futuro, para lo que no hay personal ni tiempo suficiente. Segundo, el robot dispone de muchas máquinas de estado, las que deberán ser actualizadas para agregar la funcionalidad de la interfaz episódica. Por lo tanto, se toman las siguientes decisiones:

- La librería SMACH no puede ser modificada para la implementación de la interfaz.
- Las modificaciones a las máquinas de estado deben ser mínimas y opcionales. Esto permitirá ir agregando soporte LTM de manera gradual al robot.

4.6.2. Recolección de dato episódico: *Where*

Propuesta: Se debe implementar un plugin para Bender capaz de proveer la posición del robot durante cada episodio, según los requerimientos especificados en la Sección 4.5. Por motivos prácticos, se consideran dos versiones a implementar, una con información falsa, y otra capaz de obtener datos reales desde el robot.

Versión de prueba: Debe proveer datos generados aleatoriamente para cada uno de los campos del mensaje *Where*. La motivación para esto, es poder realizar pruebas durante la implementación del sistema y poder ejecutar validaciones que no precisen de la ubicación real del robot. Además, el plugin permite minimizar el uso del robot real para la implementación y validación del sistema, pues el robot es un recurso compartido en el Laboratorio de Robótica y su uso implica costos altos en términos de tiempo.

Versión en Bender: Debe ser integrada en el software del robot, a través de la API para la obtención de su ubicación. Se utilizará la estrategia de funcionamiento recomendada en la Sección 4.3.4. El plugin deberá recopilar información periódicamente para cada uno de los campos del mensaje *Where*. Se deberá proveer un listado temporalmente ordenado de las ubicaciones recopiladas. Para disminuir el uso de recursos, solamente se deben recopilar posiciones cuando hayan episodios registrados como activos en el servidor.

4.6.3. Recolección de dato episódico: Emociones

Propuesta: Este plugin está encargado de proveer las emociones registradas por el robot durante cada episodio, según los requerimientos especificados en la Sección 4.5. En este caso, también se consideran dos versiones a implementar, una que provee datos falsos, y otra capaz de obtener datos reales desde el robot. Ambas versiones deben recopilar datos para cada uno de los campos del mensaje emocional.

Versión de prueba: Debe proveer datos generados aleatoriamente para cada uno de los campos emocionales. Similarmente al plugin para *Where*, este permite realizar pruebas durante la implementación del sistema y ejecutar validaciones que no precisen de la emoción real del robot, mientras se minimiza el uso del robot. Sin embargo, la razón principal para su implementación es que actualmente el robot no cuenta con un sistema de emociones.

Versión en Bender: Esta versión debe ser integrada con el software del robot, una vez que se haya implementado el sistema emocional. Para esto, se utilizará la estrategia de funcionamiento recomendada en la Sección 4.3.7. El plugin deberá recopilar información periódicamente sobre las emociones del robot. Una vez que el servidor notifique la finalización del episodio, el plugin debe proveer un mensaje emocional con los valores máximos percibidos para cada entidad. Además, para disminuir el uso de recursos, solamente se debe recopilar información cuando hayan episodios registrados como activos en el servidor.

4.6.4. Plugin para streams: Imágenes

Propuesta: Para los plugins encargados de proveer *streams* de datos se escogió solamente la recopilación de imágenes. Bender dispone de variados datos en su memoria de trabajo que son candidatos para ser almacenados en la memoria semántica, como por ejemplo: el sonido percibido en su micrófono, las nubes de puntos 3D percibidas por su sensor de profundidad, o los posiciones de sus efectores. Sin embargo, el almacenamiento de imágenes es muy útil, pues sirve para la demostración final, permite visualizar lo sucedido en el episodio, y la implementación puede servir para otras plataformas, pues las cámaras de video son un sensor muy común en las plataformas robóticas domésticas. Por motivos prácticos, se consideran dos versiones a implementar, una con información ficticia y otra que recopila datos desde el robot real.

Estrategia: Se utilizará la estrategia de funcionamiento propuesta en la Sección 4.3.5, mediante el uso de un *buffer* para almacenar las últimas imágenes percibidas. Luego, el plugin deberá entregar un vector con las imágenes asociadas al lapso episódico requerido.

Versión de prueba: Debe proveer imágenes obtenidas a partir de un archivo de video. Análogamente a los plugins anteriores, la motivación para esto, es poder acelerar la implementación del trabajo y poder ejecutar validaciones sin requerir el robot real.

Versión en Bender: La segunda versión debe ser integrada en el software robot, utilizando su API ROS para leer imágenes desde sus cámaras de video.

4.6.5. Plugins para entidades

Propuesta: Se decidió implementar la entidad "Humano", pues es un concepto muy recurrente para el robot Bender. Otras entidades de alta relevancia que pueden ser consideradas como trabajo futuro, son las de "Objeto", "Lugar" y "Robot". De manera similar al caso de los *streams*, se consideran dos versiones a implementar, una capaz de proveer información ficticia sobre cada entidad, y otra encargada de proveer información real recopilada por el robot.

Estrategia: Se utilizará la estrategia de funcionamiento propuesta en la Sección 4.3.5. Para ello, el plugin se debe subscribir a notificaciones sobre cambios en las entidades conocidas. Luego, se deben registrar los cambios percibidos en el historial de la colección, a pesar de que no hayan episodios activos.

Versión de prueba: Se debe implementar un nodo que publique mensajes aleatorios sobre las entidades definidas. El plugin debe subscribirse al tópico asociado, para almacenar registros de las entidades.

Versión en Bender: Se debe implementar un módulo capaz de notificar cambios en las entidades conocidas por el robot, a partir de sus módulos de percepción disponibles. El plugin se debe suscribir al tópico asociado.

5 Implementación

Es este capítulo se describe la implementación del trabajo, a partir de las decisiones de diseño expuestas en el Capítulo 4. Primero, se presenta la estructuración del software en términos de archivos y paquetes ROS. Luego se describe la implementación del sistema LTM: el modelo episódico, el servidor, el sistema de plugins y la API ROS para su utilización. En tercer lugar, se presentan módulos de software implementados para acelerar la implementación y validar el trabajo. Finalmente, se presenta la integración del sistema LTM en el robot Bender.

5.1. Estructura del Software

Esta sección presenta la estructuración del software implementado, en términos de sus archivos y paquetes ROS involucrados. Se describen las dependencias del trabajo, la estructura elegida para el sistema LTM, y la estructura para los plugins encargados de la recolección de información episódica.

5.1.1. Dependencias

A continuación se describen todas las dependencias de software utilizadas para la implementación del trabajo. Estas se dividen en las siguientes categorías: de sistema, del lenguaje C++, del lenguaje Python y de ROS.

5.1.1.1. Sistema

El trabajo fue desarrollado en Linux, Ubuntu 16.04. Las dependencias del sistema LTM son las siguientes:

- **ROS kinetic**: Se utilizó la distribución *kinetic* de ROS, la que cuenta con soporte hasta Abril del año 2021.
- **mongodb-server**: Paquete de software que contiene el servidor de MongoDB para Ubuntu 16.04.

La implementación del sistema debería ser compatible con versiones más recientes de ROS y Ubuntu, siempre que estén disponibles las dependencias mostradas en las siguientes secciones.

5.1.1.2. Dependencias C++

El servidor fue implementado completamente utilizando C++, bajo el estándar C++03, que es soportado por la mayoría de los compiladores actuales y es el utilizado por defecto en ROS kinetic. A continuación se listan las dependencias de C++ utilizadas para la implementación del servidor.

- **mongo-cxx-driver**: Driver oficial para mongodb en C++[1].
- **Boost Geometry**: Utilizado para el cómputo de la envoltura convexa y para el cálculo del centroide de un polígono.

5.1.1.3. Dependencias Python

Se utiliza Python en su versión 2.7, que es el estándar utilizado para ROS kinetic. Las siguientes dependencias son solo para los módulos específicos para la integración con Bender o para el software demostrativo.

- **cv2**: Librería OpenCV 2. Utilizada para insertar imágenes ficticias en entidades[2].
- **faker**: Librería utilizada para la generación de datos ficticios para entidades[3].

5.1.1.4. Dependencias ROS

A continuación se listan las dependencias de paquetes ROS utilizados para la implementación del servidor.

- Suite estándar de mensajes y servicios: std_srvs, std_msgs, geometry_msgs, sensor_msgs.
- **pluginlib**: Librería estándar para la implementación de plugins en ROS.

[1]Repositorio oficial de `mongo-cxx-driver`: https://github.com/mongodb/mongo-cxx-driver.git.
[2]Página oficial de OpenCV: https://opencv.org/
[3]Repositorio oficial de librería `faker` para Python: https://github.com/joke2k/faker

Las siguientes dependencias son paquetes ROS utilizados para la implementación de componentes específicas para el robot Bender y para el software demostrativo.

- **smach, smach_ros**: Librería SMACH. Utilizada para la implementación de la interfaz episódica con máquinas de estado.
- **cv_bridge**: Interfaz ROS con librería OpenCV. Utilizado para el manejo de imágenes en los plugins.

5.1.2. Paquetes de software desarrollados

Con el motivo de separar conceptos e implementar un software independiente del robot objetivo, se dividió la implementación en 6 paquetes ROS. El primero provee la conexión a la base de datos. El siguiente contiene solamente la implementación del servidor, otro provee plugins e interfaces episódicas genéricas, otro contiene código de ejemplo, para pruebas y validación. Los dos últimos contienen implementaciones específicas para el robot Bender.

Todos los paquetes de ROS implementados cuentan con un repositorio de software en GitHub, de carácter público.

5.1.2.1. Paquete ROS: `ltm_db`

El paquete se puede considerar un *fork* de `warehouse_ros_mongo`. Extiende la versión original con nuevas funcionalidades para el manejo de colecciones en MongoDB, requeridas para la implementación del servidor LTM.

Particularmente, se agregaron las siguientes funcionalidades:

- Soporte para agregar arreglos como campos para ser indexados en colecciones. Los arreglos se pueden construir para los siguientes tipos de datos en C++: `std::string`, `int`, `double` y `bool`.
- Soporte para metadatos anidados.
- Soporte para arreglos de objetos como metadatos de una colección.
- Soporte para búsquedas de mensajes en colecciones, mediante condición de pertenencia de valor a un arreglo.
- Soporte para consultas genéricas basadas en el sistema de consultas de MongoDB. Se puede realizar cualquier consulta y combinación de condiciones expresable en formato JSON, utilizando el estándar de consultas de MongoDB.

El paquete ROS se encuentra en un repositorio público en GitHub: https://github.com/mpavezb/ltm_db.

5.1.2.2. Paquete ROS: `ltm`

Contiene solamente la implementación del servidor, excluyendo plugins e interfaces episódicas, pues son consideradas como software no necesario o muy específico para un robot, como para ser considerado genérico. El paquete se puede encontrar en el siguiente repositorio público de software: https://github.com/mpavezb/ltm.

5.1.2.3. Paquete ROS: `ltm_addons`

Este paquete ROS implementa un plugin para adquisición de *streams* de imágenes, junto a las interfaces para obtener episodios desde JSON y SMACH. Contiene software genérico, agnóstico del robot a utilizar. Los tres componentes anteriores pueden ser considerados herramientas útiles para algún robot, pero que no son estrictamente necesarios para el funcionamiento del sistema LTM. El repositorio asociado se puede encontrar en: https://github.com/mpavezb/ltm_addons.

5.1.2.4. Paquete ROS: `ltm_samples`

Este paquete contiene implementaciones de ejemplo de algunos plugins episódicos capaces de generar información ficticia. Además, el paquete contiene software útil para pruebas del funcionamiento del sistema y código para su validación. Su repositorio de software se encuentra en: https://github.com/mpavezb/ltm_samples.

5.1.2.5. Paquete ROS: `bender_emotion`

Ya que Bender no dispone de un software generador de emociones, este paquete tiene como objetivo albergar la implementación del sistema emocional descrito en la Sección 4.6. El repositorio se encuentra alojado en la cuenta GitHub del equipo encargado de Bender: https://github.com/uchile-robotics/bender_emotion.

5.1.2.6. Paquete ROS: `bender_ltm`

Este paquete tiene por objetivo almacenar la implementación de los plugins específicos para el uso del sistema LTM en el robot Bender, y es considerado el punto de integración del trabajo con el robot. Similarmente al paquete `bender_emotion`, su repositorio se encuentra alojado en: https://github.com/uchile-robotics/bender_ltm.

5.1.3. Líneas de código

En la Tabla 5.1 se presenta una tabla con el conteo de líneas de código implementadas para el trabajo, ordenadas por el lenguaje utilizado. Se utilizó el programa `cloc`[4] para realizar el cómputo.

Lenguaje	Archivos	Líneas en blanco	Comentarios	Código
C++	37	831	575	3896
Python	57	1078	792	3646
C/C++ Header	51	716	435	2397
CMake	8	135	294	380
XML	29	190	178	373
JSON	12	0	0	321
Markdown	10	145	0	245
ROS msg/srv	34	43	90	217
BASH	5	37	22	183
JavaScript	1	30	45	133
YAML	5	43	70	111
Total	249	3248	2501	11902

Tabla 5.1: Líneas de código del sistema LTM por lenguaje. Para cada lenguaje utilizado, se indican la cantidad de archivos asociados y un conteo de la cantidad de líneas en blanco, líneas utilizadas para comentarios y líneas de código.

5.2. Sistema LTM

En esta sección se describe la implementación del sistema LTM desarrollado. Primero, se presenta el modelo de datos para la definición de episodios, *streams* y entidades. Luego, se describe la implementación del servidor LTM, basada en la librería `pluginlib`. Se presenta el sistema de plugins desarrollado y sus requisitos para los usuarios. Finalmente, se presenta la API ROS que provee el sistema LTM, para su configuración, ejecución, recolección de episodios y realización de consultas al servidor.

5.2.1. Modelo de datos

A continuación se presenta el modelo de datos, construido a partir de mensajes ROS, utilizado para manejar los conceptos de episodios, entidades y *streams*. Los mensajes implementados son la base para la construcción del sistema LTM y definen la estructura episódica a utilizar para la comunicación entre los clientes y el servidor en ROS.

Todos los mensajes presentados a continuación se encuentran en el paquete ROS `ltm`.

[4]Repositorio del programa `cloc` en GitHub: https://github.com/AlDanial/cloc

5.2.1.1. Episodios

A partir del diseño propuesto en la Sección 4.3, se construye el mensaje ROS `ltm/Episode.msg`[5], presentado en el Código 5.1. El mensaje tiene campos para manejar la estructura de árbol episódico, la información sobre *What*, *When* y *Where*, las relevancias episódicas, y la información para introspección.

Código 5.1: ltm/Episode.msg

```
# Identification
uint32 uid
uint8 type
string[] tags
ltm/Info info

# Tree Information
uint32 parent_id
uint32[] children_ids
string[] children_tags

# Episode Information
ltm/What what
ltm/When when
ltm/Where where

# Relevance
ltm/Relevance relevance
```

Para ordenar la información, el mensaje se construye a partir de otros mensajes ROS definidos en el mismo paquete:

- `ltm/When.msg`: Mantiene el contexto temporal del episodio, indicando tiempos de inicio y fin, mediante campos del tipo `time`[6]. La información temporal descrita en otro formato, por ejemplo, cuando un humano dice "este semestre" o "la semana pasada", debe ser convertida en rangos temporales del tipo `time`.

- `ltm/Where.msg`: Almacena el contexto espacial del episodio, utilizando dos estrategias.

  - Permite almacenar un conjunto de coordenadas, mediante mensajes del tipo `geometry_msgs/Point.msg`[7], sumado al nombre del mapa y del sistema coordenado utilizado. Además, mantiene la envoltura convexa de las posiciones de sus hijos.

  - Permite almacenar la ubicación del robot en formato textual.

- `ltm/What.msg`: Almacena referencias a cada pieza de memoria semántica asociada al episodio. Se construye a partir de un mensaje para el manejo de *streams* y otro para el

[5]El dominio `ltm/` hace referencia al nombre del paquete ROS donde se implementa el servidor y que contiene la definición de todos los mensajes relacionados.

[6]El tipo de dato `time` es estándar en ROS y permite indicar un instante de tiempo mediante un contador de segundos transcurridos desde el tiempo cero (Jueves 1 de Enero de 1970 a las 00:00:00). Tiene una resolución de nanosegundos.

[7]El paquete de ROS `geometry_msgs` proporciona mensajes para manejar primitivas geométricas y sus transformaciones. Pertenece al conjunto de paquetes para mensajes `common_msgs`, el cual se considera estable. Más información en la web oficial: http://wiki.ros.org/geometry_msgs

manejo de entidades.

  - `ltm/StreamRegister.msg`: Utilizado para mantener un listado de cada *stream*, mediante su identificador único y el tipo de mensaje ROS registrado por su plugin.

  - `ltm/EntityRegister.msg`: Utilizado para mantener un listado de cada entidad asociada al episodio, mediante el tipo de mensaje ROS registrado por su plugin, su identificador único, y un listado de los registros ingresados en el historial de la entidad.

- `ltm/Relevance.msg`: Almacena datos sobre la relevancia episódica generalizada y las subrelevancias.

  - La relevancia generalizada es construida mediante un indicador numérico y la fecha de su última actualización.

  - `ltm/EmotionalRelevance.msg`: Almacena la relevancia emocional del episodio. Mantiene un indicador numérico para la emoción más relevante, los valores de cada una de las 8 emociones percibidas, sumado a información sobre el software emocional utilizado.

  - `ltm/HistoricalRelevance.msg`: Almacena la relevancia histórica del episodio. Mantiene un indicador numérico, la fecha de la última actualización y la fecha de la siguiente actualización agendada.

- `ltm/Info.msg`: Utilizado para almacenar metadatos del episodio, útiles para una introspección posterior por parte del usuario. Mantiene información como la fecha de creación y de último acceso al episodio, e información sobre el sistema operativo, la versión de ROS y la versión del sistema LTM utilizada.

Todos los mensajes ROS descritos anteriormente se pueden encontrar en el Anexo B.1.

5.2.1.2. Construcción de *streams*

Metadatos: Los *streams* se definen a partir de mensajes ROS construidos por el usuario, los que pueden contener cualquier estructura soportada por ROS para sus mensajes. La implementación impone solamente un requisito sobre su definición. Se debe añadir un campo de tipo `ltm/StreamMetadata.msg` al mensaje, bajo el nombre `meta`. El tipo `ltm/StreamMetadata.msg`, presentado en el Código 5.2, contiene información manejada por el servidor, indicando los tiempos de inicio y fin del *stream*, sumado a los identificadores del mensaje y su episodio asociado.

Código 5.2: ltm/StreamMetadata.msg

```
uint32 uid # stream uid in database
uint32 episode # episode uid in database

# time span
```

```
time start
time end
```

5.2.1.3. Construcción de entidades

Metadatos: En el caso de las entidades, se deben almacenar distintas instancias de estas, cada una asociada a su historial de cambios. Una entidad es definida a través de un mensaje ROS construido por el usuario, y puede contener cualquier estructura soportada por ROS para sus mensajes. De manera similar al caso de los *streams*, cada mensaje debe contener un campo de nombre `meta` y tipo `ltm/EntityMetadata.msg`. Este campo es manejado por el servidor y permite identificar la instancia de la entidad asociada al mensaje. El tipo `ltm/EntityMetadata.msg` se presenta en el Código 5.3.

Código 5.3: ltm/EntityMetadata.msg

```
uint32 uid # entity uid in database

# log uids
uint32 log_uid # entity register uid at recall time
uint32 init_log # first entity register uid in database
uint32 last_log # last entity register uid in database

# log stamps
time stamp # recall time (log stamp)
time init_stamp # first registered stamp
time last_stamp # last registered stamp
```

Historial: Como se describe en el diseño del modelo de datos (ver Sección 4.4), el historial se construye utilizando dos colecciones, de sufijos `.trail` y `.meta`. Cada mensaje almacenado en una colección tiene asociado un registro en la otra, bajo el mismo identificador.

La colección de sufijo `.meta` se construye a partir de mensajes ROS de tipo `ltm/EntityLog.msg`, presentado en el Código 5.4. Este tipo de mensajes es manejado automáticamente por el servidor, y permite indicar lo siguiente: Identificadores de la instancia, del registro y de los episodios asociados; Instante de tiempo del registro; Registro que precede al mensaje (lista enlazada de registros); Nombres de campos que fueron agregados, modificados o eliminados.

Código 5.4: ltm/EntityLog.msg

```
# keys
uint32 entity_uid
uint32 log_uid

# who
uint32[] episode_uids

# when
time timestamp
uint32 prev_log
uint32 next_log

# what
string[] new_f
```

```
string[] updated_f
string[] removed_f
```

La colección de sufijo `.trail` se construye a partir de mensajes ROS del mismo tipo definido por el usuario. El campo `meta` es utilizado para asociar bidireccionalmente cada entrada del historial a su mensaje mensaje de tipo `ltm/EntityLog.msg`, en la colección de sufijo `.meta`.

5.2.2. Servidor

El servidor es implementado de acuerdo a las decisiones de diseño estudiadas en el Capítulo 4, y su estructura se basa en el diagrama de la Figura 4.4. La implementación se realiza completamente en el lenguaje C++ y se encuentra en el paquete ROS `ltm`. El servidor solo depende de tipos de dato primitivos y paquetes estándar disponibles en todas las distribuciones ROS (e.g. `geometry_msgs` y `std_msgs` y `pluginlib`), el módulo geométrico de la librería Boost, y el driver para la base de datos.

La implementación se puede separar lógicamente en 2 componentes: memoria episódica y memoria semántica. La primera se encarga de adquirir episodios a partir de la API episódica que provee el servidor, mientras que el componente semántico se encarga de los *streams* y entidades, por medio de los plugins que define el usuario.

La base de datos es manejada automáticamente por el servidor, por lo que el usuario no tiene un acceso directo a esta, sino que solo puede realizar operaciones de escritura a través de la API ROS del servidor y de las facilidades que proveen los plugins. Todas las funcionalidades para el manejo de MongoDB se encuentran en el paquete ROS `ltm_db`.

5.2.3. Plugins

El diseño del sistema LTM requiere que el usuario implemente un conjunto de plugins para la adquisición de información episódica y semántica. Por un lado, para la memoria episódica se debe implementar un plugin para la adquisición del estado emocional del robot, y un plugin para obtener la ubicación del robot durante cada episodio. Por otro lado, el usuario puede implementar una cantidad indefinida de plugins para la recopilación de información semántica, en forma de *streams* y entidades; Según lo requiera su aplicación robótica, pueden haber M plugins para *streams* y N para entidades, por lo que el sistema utilizaría $M + N + 2$ plugins en total.

La implementación utiliza el paquete ROS `pluginlib` para la definición y carga dinámica de plugins definidos en paquetes ROS externos. Además, ya que los mensajes ROS definidos por el usuario para *streams* y entidades no son conocidos en tiempo de compilación, se hace uso intensivo de tipos de dato genéricos en C++ para su representación y el manejo de la base de datos. A continuación, se presentan las consideraciones para la implementación de cada tipo de plugin requerido.

5.2.3.1. Plugins episódicos: *Where* y Emociones

Estos dos plugins son necesarios para obtener información episódica dependiente del robot objetivo. El único requisito impuesto para el usuario es la implementación de cada plugin mediante el estándar utilizado en `pluginlib`, para proveer cada dato episódico cuando el servidor lo requiera. Se aconseja seguir la estrategia de funcionamiento descrita en la Sección 4.5.

El plugin de emociones debe heredar la clase `ltm::plugin::EmotionBase`, definida en `#include <ltm/plugin/emotion_base.h>`. Se deben implementar los *métodos virtuales* declarados en la clase padre.

De la misma manera, el plugin para recolectar el campo *Where* debe heredar la clase `ltm::plugin::LocationBase`, definida en `#include <ltm/plugin/location_base.h>`. El plugin debe implementar los *métodos virtuales* declarados por la clase padre.

5.2.3.2. Plugins: *Streams*

Plugin a implementar: Se debe implementar un plugin por cada *stream* definido por el usuario. Cada plugin debe heredar de la clase `ltm::plugin::StreamBase`, definida en `#include <ltm/plugin/stream_base.h>`. Se deben implementar los *métodos virtuales* declarados por la clase padre. Las funcionalidades requeridas se describen en la Sección 4.5.2 y se recomienda utilizar la estrategia de procesamiento propuesta en la misma sección.

Servicio ROS: Además, ya que los plugins son cargados dinámicamente, el mensaje ROS definido por el usuario no es conocido en tiempo de compilación. Es por esto, que para proveer una API CRUD mediante servicios ROS, se requiere que el usuario defina un servicio adecuado a su mensaje. El archivo `.srv` debe tener el formato presentado en el Código 5.5, donde `<user_package>` y `<stream_msg>` hacen referencia al paquete ROS y al tipo del mensaje definido por el usuario, respectivamente.

Código 5.5: SampleStreamSrv.srv

```
1 uint32[] uids
2 <user_package>/<stream_msg>[] msgs
3 ---
4 <user_package>/<stream_msg>[] msgs
```

Funcionalidades heredadas: Dado que `pluginlib` no soporta clases base genéricas de C++, el plugin también debe heredar de la clase `ltm::plugin::StreamDefault<MsgType, SrvType>`, definida en `#include <ltm/plugin/stream_default.h>`, donde `MsgType` y `SrvType` deben ser reemplazados por la clase del mensaje y servicio ROS manejados por el plugin. Lo anterior, permite al plugin manejar automáticamente la base de datos y proveer una API CRUD mediante servicios ROS, adecuada al mensaje en cuestión y simplificando la implementación del plugin.

5.2.3.3. Plugin: *Entidades*

Plugin a implementar: De manera análoga al caso de los *streams*, se debe implementar un plugin por cada entidad definida por el usuario. Cada plugin debe heredar de la clase `ltm::plugin::EntityBase`, definida en `#include <ltm/plugin/entity_base.h>`. Se deben implementar los *métodos virtuales* declarados por la clase padre. Las funcionalidades requeridas se describen en la Sección 4.5.2 y se recomienda utilizar la estrategia de procesamiento propuesta en la misma sección.

Servicio ROS: Para este tipo de plugin, también se requiere que el usuario defina un servicio adecuado a su mensaje. El archivo `.srv` debe tener el formato presentado en el Código 5.6, donde `<user_package>` y `<entity_msg>` hacen referencia al paquete ROS y al tipo del mensaje definido por el usuario, respectivamente.

Código 5.6: SampleEntitySrv.srv

```
uint32[] uids
time[] stamps
<user_package>/<entity_msg>[] msgs
---
<user_package>/<entity_msg>[] msgs
```

Funcionalidades heredadas: La implementación provee automáticamente funcionalidades para el manejo de la base de datos y una API CRUD mediante servicios ROS. Para esto, el plugin a implementar debe heredar la clase `ltm::plugin::EntityDefault<MsgType, SrvType>`, definida en `#include <ltm/plugin/stream_default.h>`, donde `MsgType` y `SrvType` deben ser reemplazados por la clase del mensaje y servicio ROS manejados por el plugin.

5.2.4. API ROS

A continuación se presenta la API ROS para el uso de las funcionalidades del sistema LTM implementado. Se explica el uso del servidor de parámetros de ROS para la configuración del servidor LTM, y el uso de la herramienta `roslaunch` para su ejecución. Luego, se presenta la API para recolección de episodios, a partir de servicios ROS. Finalmente, se presenta la API para realizar consultas episódicas y semánticas al sistema LTM.

Es importante destacar que a pesar de que el sistema se implemente en C++, ROS permite utilizar todas las funcionalidades de la API a través de diversos lenguajes de programación: Nativamente en C++(`roscpp`), Python (`rospy`) y CommonLisp (`roslisp`), y mediante clientes externos, implementados para otros lenguajes.

5.2.4.1. Configuración y ejecución

Configuración: El servidor LTM puede ser configurado de manera estática, previo a su iniciación, a través del servidor de parámetros de ROS. La configuración puede ser escrita

en un archivo en formato YAML, lo que permite modificar parámetros del sistema LTM, sin tener que compilar nuevamente el código.

En el Código B.11, presentado en el Apéndice de Implementación, se presenta un archivo YAML de ejemplo con la configuración del sistema. Es posible modificar los siguientes conjuntos de parámetros:

- Base de datos: Se puede indicar el nombre de la base de datos y opciones de conectividad al servidor de MongoDB. Parámetros: `db`, `host`, `port` y `timeout`.

- Colección de episodios: Se puede indicar el nombre a utilizar para la colección de episodios.

- Plugin emocional: Se puede indicar el plugin emocional a ocupar, mediante su nombre exportado para `pluginlib`. Se pueden agregar parámetros extra, requeridos por el desarrollador del plugin. En el caso de ejemplo, se ocupa un plugin de clase `Emotion Plugin`, definido en el paquete ROS `ltm_samples`.

- Plugin *Where*: Se puede indicar el plugin para el campo *Where* a ocupar, mediante su nombre exportado para `pluginlib`. Se pueden agregar parámetros extra, requeridos por el desarrollador del plugin. En el caso de ejemplo, se ocupa un plugin de clase `LocationPlugin`, definido en el paquete ROS `ltm_samples`.

- Plugins de *streams*: Se puede indicar un listado de plugins a cargar para el manejo de *streams*. Cada plugin debe indicar al menos 3 parámetros:

  - `class`: Clase exportada por `pluginlib`. En el ejemplo se carga un plugin de clase `ltm_addons::ImageStreamPlugin`.

  - `type`: Tipo del plugin. Utilizado para construir la API ROS para *streams* y para indicar el tipo del *stream* al episodio relacionado.

  - `collection`: Nombre de la colección de MongoDB a utilizar para el plugin. Este nombre será modificado con el prefijo `stream:`.

  Además, el desarrollador de cada plugin puede requerir la introducción de parámetros extra. En el código de ejemplo, se anexan los parámetros: `topic`, `buffer_frequency` y `buffer_size`.

- Plugins de *Entidades*: Se puede indicar un listado de plugins a cargar para el manejo de entidades. En el ejemplo se configuran dos plugins: `people` y `objects`. De la misma forma que para los *streams*, cada plugin debe indicar al menos los siguientes parámetros:

  - `class`: Clase exportada por `pluginlib`. En el ejemplo se cargan plugins de las clases `ltm_samples::PeopleEntityPlugin` y `ltm_samples::ObjectsEntityPlugin`.

  - `type`: Tipo del plugin. Utilizado para construir la API ROS para entidades y para indicar el tipo de entidad al episodio relacionado.

– `collection`: Nombre de la colección de MongoDB a utilizar para el plugin. Este nombre será modificado con el prefijo `entity:`.

El desarrollador de cada plugin puede requerir la introducción de parámetros extra. En el código de ejemplo, ambos plugins anexan solo un parámetro extra: `topic`.

En resumen, el código de ejemplo realiza las siguientes configuraciones para el sistema LTM: Define una conexión a una base de datos específica. Configura plugins emocionales y de ubicación específicos, definidos en el paquete `ltm_samples`. Configura solo un plugin para `streams`, implementado en el paquete `ltm_addons`. Configura 2 plugins para entidades, ambos implementados en el paquete `ltm_samples`. Luego, en caso de implementar un nuevo plugin, basta modificar el archivo de configuración para anexarlo a la lista respectiva.

Ejecución: Finalmente, una vez construido el archivo de configuración adecuado para el robot, basta utilizar la herramienta `roslaunch` para ejecutar el sistema LTM. En el Código 5.7 se muestra un archivo `.launch` de ejemplo donde se asocia indica la configuración para el servidor. Para su ejecución, se puede utilizar el comando de consola: `$ roslaunch ltm_samples server.launch`.

Código 5.7: server.launch

```
<launch>

    <!-- LTM server -->
    <include file="$(find ltm)/launch/ltm.launch" ns="robot">
        <arg name="config_file" value="$(find ltm_samples)/config/server.yaml"/>
    </include>

</launch>
```

5.2.4.2. API Episódica

La API episódica es el medio para que el usuario pueda generar episodios para el sistema LTM. Funciona según el diseño propuesto en la Sección 4.5, mediante servicios ROS para registro e inserción de episodios.

Registro: En primer lugar, se debe utilizar el servicio ROS `~/ltm/episode/register` para registrar el inicio de un episodio en el sistema. El servicio de tipo `ltm/RegisterEpisode.srv`, presentado en el Código 5.8, permite que el servidor genere un identificador disponible para el episodio. Es necesario indicar si el episodio a introducir es de tipo hoja, para iniciar los procesos de recopilación de información. Además, el servidor permite registrar un episodio con un identificador predefinido, lo que es útil para realizar pruebas del sistema.

Código 5.8: ltm/RegisterEpisode.srv

```
# Use random UID or fixed one.
bool generate_uid # Whether to generate a random UID for the episode or not.
uint32 uid # This field is used instead of the automatic default uid.
bool replace # Replace episode if it already exists (same uid).
```

```
---
uint32 uid
```

Término: Al finalizar el episodio, se debe utilizar el servicio `~/ltm/episode/add` para que el servidor recolecte información de cada plugin y almacene el episodio en la base de datos. El servicio de tipo `ltm/AddEpisode.srv` se presenta en el Código 5.9. Es necesario indicar el identificador del episodio y del padre. Además, el servicio permite introducir un episodio completo, sin necesidad de su registro, lo que es útil para la inserción de episodios pasados.

Código 5.9: ltm/AddEpisode.srv

```
# target episode
ltm/Episode episode

# replace if already exists
bool replace
bool logging
---
bool succeeded
```

Actualización: Finalmente, el sistema provee el servicio `~/ltm/episode/update_tree`, de tipo `ltm/UpdateTree.srv`, que permite recomputar recursivamente todos los datos episódicos de un nodo y sus hijos. Este servicio es útil cuando se insertan episodios pasados. Es necesario indicar el identificador de la raíz del árbol a actualizar. Sus campos se presentan en el Código 5.9.

Código 5.10: ltm/UpdateTree.srv

```
# target tree root uid
uint32 uid
---
bool succeeded
```

5.2.4.3. API LTM

De acuerdo al diseño de las operaciones CRUD, revisado en la Sección 4.4, se implementaron 2 etapas de consulta.

Consultas JSON: La primera etapa consiste en realizar una consulta al sistema, mediante filtros sobre los campos episódicos, de *streams* o de entidades. Se debe utilizar el servicio `~/ltm/db/query`, de tipo `ltm/QueryServer.srv`, presentado en el Código 5.11. El formato permite indicar si la consulta se realizará sobre la colección de episodios, o sobre un *stream* o entidad de tipo particular.

Código 5.11: ltm/QueryServer.srv

```
# target LTM data: episode, entity, stream
string target
```

```
3
4  # semantic type
5  # only valid when 'target' in [entity, entity_trail, stream]
6  string semantic_type
7
8  # json query based on MongoDB style
9  string json
10
11 bool logging
12 ---
13 # matching uids
14 uint32[] episodes
15 ltm/QueryResult[] streams
16 ltm/QueryResult[] entities
17 ltm/QueryResult[] entities_trail
```

El campo `json` permite definir una consulta compleja, utilizando el mismo formato que en una consulta de MongoDB. Luego, la respuesta indica los identificadores de los episodios, *streams* y entidades que calzan con el filtro. El formato de respuesta utiliza el tipo `ltm/QueryResult.msg`, mostrado en el Anexo B.2.

El Código 5.12 muestra algunos ejemplos de consultas JSON válidas para realizar búsquedas de episodios, entidades y *streams*. Las condiciones de búsqueda presentadas son de igualdad, orden, pertenencia a un arreglo y anidamiento mediante condiciones lógicas AND y OR. Sin embargo, el servidor permite el uso de cualquier consulta JSON soportada por MongoDB.

Código 5.12: Consultas JSON de ejemplo

```
1  # Search by specific field
2  { "uid": 10 }
3  { "where.map_name": "a_map" }
4
5  # String array contains a string
6  { "tags": "a_tag" }
7
8  # String in Array
9  { "where.location": { "$in": [ "location_a", "location_b" ] } }
10
11 # Search by range (<, <=, >, >= operators)
12 { "relevance.emotional.value": { "$gt": 0.6 } }
13
14 # Logical AND
15 { "type": 0, "tags": "a_tag" }
16
17 # Logical OR
18 { "$or": [
19     { "relevance.emotional.value": { "$gt": 0.3 } },
20     { "tags": "a_tag" }
21   ]
22 }
```

Adquisición de episodios: Una vez seleccionados los episodios de interés, el servicio `~/ltm/episode/get` permite recolectarlos desde el servidor LTM. El servicio de tipo `ltm/GetEpisodes.srv` se presenta en el Código 5.13.

Código 5.13: ltm/GetEpisodes.srv

```
1  # target uids
```

```
uint32[] uids
---
ltm/Episode[] episodes
uint32[] not_found
```

Adquisición de *streams*: De la misma forma, una vez seleccionados los *streams* de interés, se puede utilizar un servicio ROS para recolectar los mensajes. Existe un servicio con el nombre `~/ltm/stream/<type>/get` para cada plugin, donde `<type>` corresponde al tipo definido en la configuración. Cada servicio tiene el formato presentado en el Código 5.5.

Adquisición de entidades: Una vez seleccionadas las entidades de interés, se puede utilizar un servicio ROS para recolectar sus mensajes. Cada plugin provee un servicio de nombre `~/ltm/entity/<type>/get`, donde `<type>` corresponde al tipo definido durante la configuración. Cada servicio tiene el formato presentado en el Código 5.6. Además, el servicio permite indicar un instante de tiempo para cada instancia a recuperar. Las entidades serán reconstruidas según la perspectiva que se tenía de ellas en ese instante de tiempo.

5.2.4.4. API LTM: Otros servicios

El sistema LTM además provee otros servicios mediante su API ROS. Cada uno de los servicios mostrados a continuación no es parte de los requisitos del trabajo, pero es de utilidad para la ejecución de pruebas y validaciones.

Base de datos: El sistema provee servicios para eliminar todas las colecciones de la base de datos actual, y para cambiar la base de datos por otra. El primero, se provee mediante el servicio `~/ltm/db/drop`, de tipo `DropDB.srv`, mostrado en el Código 5.14. Se pide confirmación, pues es una operación peligrosa. El cambio de base de datos se provee en el servicio `~/ltm/db/switch`, de tipo `SwitchDB.srv`, el que solo requiere indicar el nombre de la base de datos alternativa. Esta funcionalidad es útil para ejecutar pruebas sin afectar la base de datos del robot.

Código 5.14: ltm/DropDB.srv

```
# As this service is potentially dangerous
# the server requires some kind of user confirmation
#
# Also, as we cannot depend on default values for ROS msg/srv fields
# (they are language dependent), you must set BOTH fields
bool i_understand_this_is_a_dangerous_operation
bool html_is_a_real_programming_language
---
```

Operaciones CRUD para *streams*: Cada plugin para *streams* provee servicios ROS para insertar (`~/ltm/stream/<type>/add`) y eliminar (`~/ltm/stream/<type>/delete`) mensajes de su colección. Ambos servicios utilizan el formato presentado en el Código 5.5.

Operaciones CRUD para entidades: De la misma manera, cada plugin para entidades provee funcionalidades para insertar y eliminar mensajes de su colección, mediante los servicios `~/ltm/entity/<type>/add` y `~/ltm/entity/<type>/delete`, respectivamente. Ambos

servicios utilizan el formato presentado en el Código 5.6.

5.3. Componentes para Validación

En esta sección se presentan tres herramientas implementadas para la validación del trabajo y la realización de experimentos. En primer lugar, se revisa una interfaz episódica entre el servidor LTM y la librería SMACH, cuyo objetivo es la generación de episodios a partir de máquinas de estado. Luego, se presenta un plugin para la adquisición de *streams* de imágenes, a partir del estándar ROS para la transmisión de video. Tanto la librería SMACH como el plugin de video son agnósticos de la plataforma objetivo. Finalmente, se presenta la construcción de un robot simulado, con todos los plugins requeridos para validar el sistema LTM.

5.3.1. Interfaz SMACH para recolección de episodios

De acuerdo a las consideraciones expresadas en la Sección 4.6, se implementó una librería Python para la generación de episodios a partir de máquinas de estado expresadas mediante SMACH. Por cada ejecución de un estado marcado como candidato, se almacena un episodio en el servidor LTM. El proceso es automático y solo impone unos pocos requerimientos al desarrollador.

Implementación: Se siguió la siguiente estrategia para reducir los requerimientos para el usuario de la librería. Se utilizó la API SMACH para la lectura de máquinas de estado y la creación de *callbacks* ante el inicio y término de un estado. El desarrollador de la máquina debe indicar que estados son candidatos para ser convertidos en episodios, y debe indicar un listado de *tags* opcionales que identifiquen al estado. Todos los procesos de registro e inserción de episodios son llevados a cabo automáticamente por la librería. Ya que las máquinas de estado en SMACH siguen una estructura de anidada, por cada máquina se crea un árbol episódico con cada episodio candidato.

La implementación solo utiliza la API SMACH y la API ROS para el servidor LTM, siendo agnóstica del robot. Luego, la librería puede ser utilizada por otros robots basados en SMACH. La implementación se encuentra en el paquete ROS `ltm_addons`.

Uso de la interfaz: En el Código 5.15 se presenta una máquina de estado completamente funcional, implementada en SMACH y utilizando la interfaz desarrollada. La máquina consta de dos estados `FOO` y `BAR`. Los requisitos para el uso de la interfaz son 3:

1. Importar la interfaz LTM/SMACH (ver línea 4 del código).

2. Registrar los estados candidatos, junto con sus tags. En el ejemplo se presentan dos alternativas de registro:

    - Se puede marcar un estado desde su constructor (línea 9), para que toda instancia

creada sea registrada automáticamente.

- Se puede marcar manualmente una instancia particular de un estado (línea 19), cuando solo se desea registrar unos pocos estados, o no es posible modificar el constructor.

3. Se debe ejecutar el método `ltm.setup()` (línea 27), para configurar los callbacks de todo estado registrado e inicializar la conexión con el servidor LTM.

Código 5.15: Ejemplo de uso de la interfaz LTM/SMACH

```
1  #!/usr/bin/env python
2  import rospy
3  import smach
4  import ltm_addons.smach as ltm
5
6  class SampleState(smach.State):
7      def __init__(self, text=""):
8          smach.State.__init__(self, outcomes=['succeeded'])
9          ltm.register_state(self, ["sample_tag"])
10
11     def execute(self, userdata):
12         return 'succeeded'
13
14 if __name__ == '__main__':
15     rospy.init_node('sample_machine_node')
16
17     # Build machine
18     sm = smach.StateMachine(outcomes=['succeeded'])
19     ltm.register_state(sm, ["a_tag", "other_tag"])
20     with sm:
21         smach.StateMachine.add('FOO', SampleState(),
22             transitions={'succeeded': 'BAR'})
23         smach.StateMachine.add('BAR', SampleState(),
24             transitions={'succeeded': 'succeeded'})
25
26     # Setup LTM callbacks
27     ltm.setup(sm)
28
29     # RUN
30     sm.execute()
```

Consideración 1: La interfaz SMACH/LTM implementada ha mostrado ser funcional, sin embargo, podría no ajustarse a las necesidades de otros robots. En tal caso, el desarrollador puede modificar o implementar una versión adecuada de la interfaz, siempre que utilice los servicios LTM para el registro e inserción de episodios.

Consideración 2: No se han desarrollado pruebas sobre el impacto de la interfaz en una máquina de estado, en términos del costo temporal requerido para satisfacer los callbacks utilizados. Esto puede ser importante para usuarios que registren estados consecutivos de muy corta duración, pues se deben ejecutar servicios de registro e inserción por cada ejecución de un estado. Más aún, es esperable que el costo de inserción se incremente según se almacenan más episodios en la base de datos.

5.3.2. Plugin para *streams* de imágenes

De acuerdo a lo revisado en la Sección 4.6, se ha implementado un plugin LTM para la adquisición de *streams* a partir de imágenes. El plugin recolecta imágenes mediante la suscripción a un tópico con mensajes de tipo `sensor_msgs/Image`, el cual es el estándar utilizado en ROS para la lectura de imágenes desde cámaras de video.

Implementación: El plugin se implementó en el paquete ROS `ltm_addons`, en la clase `ltm_addons::ImageStreamPlugin`. Puede ser configurado mediante el servidor de parámetros de ROS, para ajustar la frecuencia de adquisición de imágenes y el tamaño del buffer. En los Códigos 5.16 y 5.17 se presenta la firma del mensaje ROS utilizado por el plugin y del servicio ROS para su interfaz con el servidor.

Código 5.16: ltm_addons/ImageStream.msg

```
ltm/StreamMetadata meta
sensor_msgs/Image[] images
```

Código 5.17: ltm_addons/ImageStreamSrv.srv

```
uint32[] uids
ltm_addons/ImageStream[] msgs
---
ltm_addons/ImageStream[] msgs
```

Configuración: Para utilizar el plugin, basta anexar su configuración a la lista de plugins utilizados en el servidor LTM. En el Código 5.18 se presenta un ejemplo de configuración del servidor, donde se utiliza este plugin y se configuran los 3 parámetros de interés: `topic`, para indicar el tópico de lectura de imágenes, sumado a `buffer_frequency` y `buffer_size` para ajustar el buffer.

Código 5.18: server.yaml

```yaml
plugins:
  streams:
    include: ["image"]
    image:
      class: "ltm_addons::ImageStreamPlugin"
      type: "images"
      collection: "images"
      topic: "/robot/fake/sensors/camera/image_raw"
      buffer_frequency: 3.0
      buffer_size: 100
```

5.3.3. Robot simulado

Para acelerar el desarrollo del trabajo y realizar validaciones de funcionalidad, se implementó un robot simulado capaz de responder a todas las consultas necesarias para el servidor

LTM. La implementación se encuentra en el paquete ROS `ltm_samples`, donde se implementaron plugins para la recolección del campo *Where*, las emociones y las entidades simuladas.

El robot implementado responde a todas las consultas mediante la generación de campos aleatorios que sean requeridos. Los plugins implementados son los siguientes:

- Ubicación: Implementado en la clase `ltm_samples::LocationPlugin`.
- Emociones: Implementado en la clase `ltm_samples::EmotionPlugin`.
- Entidades: Se implementaron plugins para las entidades personas y objeto. Las clases correspondientes son `ltm_samples::PeopleEntityPlugin` para personas, y la clase `ltm_samples::ObjectsEntityPlugin` para representar objetos.

5.4. Integración en Bender

En esta sección se describe la integración del sistema LTM en el robot Bender, a partir de los módulos implementados en las secciones anteriores. En primer lugar, se da una descripción general de la integración. Luego, se presentan los plugins específicos implementados para esto. Finalmente, se explican las máquinas de estado utilizadas para demostrar la integración del sistema en el robot.

5.4.1. Descripción general

Por un lado, la integración con Bender requiere el desarrollo de plugins para la adquisición de datos desde el robot. Por otro lado, se deben elaborar máquinas de estado que permitan recopilar información episódica y acceder a ella. Todos los componentes específicos para Bender fueron implementados en el paquete ROS `bender_ltm`.

Se desarrollaron plugins para recolectar datos desde la memoria de trabajo del robot. Particularmente, la ubicación del robot se obtiene desde su sistema de localización; las secuencias de imágenes se obtienen desde su cámara de video; y la información sobre entidades se recopila desde sus capacidades de percepción robótica y reconocimiento de voz.

A modo de demostración, se implementó una máquina de estado capaz de recolectar datos de personas. Además, se utiliza otra máquina de estado es para que el robot describa los conocimientos adquiridos previamente, a partir de consultas a la memoria LTM.

5.4.2. Plugins del sistema LTM

El proceso de integración del sistema LTM en el robot Bender requiere el desarrollo de plugins específicos para sus necesidades. A continuación se detallan los aspectos importantes de cada plugin elaborado.

Plugin para el campo *where*: La ubicación del robot se obtiene a partir del sistema de localización del robot. URF provee un tópico que indica la ubicación del robot en un mapa predefinido, en cada instante de tiempo y respecto a un sistema coordenado. Además, cada mapa tiene zonas predefinidas, identificadas por una etiqueta indicando el nombre de cada lugar en el mapa. Conociendo la ubicación coordenada del robot, se puede saber el nombre de la zona actual. Por ejemplo, este plugin permitiría saber que el robot se encuentra en la coordenada X,Y de un mapa, la que se puede asociar al "dormitorio del hogar".

Plugin para el estado emocional: Según se describe en capítulos anteriores, el desarrollo de un plugin emocional requiere la existencia de un sistema emocional en el robot Bender. Tal sistema aún no existe en URF y su desarrollo se escapa del alcance de este trabajo de título. Por lo tanto, este plugin queda propuesto como trabajo futuro.

Plugin para *streams*: A modo de demostración funcional, se considera el concepto de *streams* de imágenes. Para esto, se utiliza el plugin implementado en la Sección 5.3, donde solamente se debe indicar el tópico para acceder a los datos de la cámara de video de Bender.

Plugins para entidades: Se consideró la entidad "humano". A modo de demostración, solamente se consideraron los siguientes datos de interés: nombre, edad, género, emoción e imagen de la cara. A excepción del nombre, todos los datos se pueden obtener desde el módulo de percepción del robot. Además, por simplicidad, cada persona es identificada por su nombre, por lo que estos se consideran únicos. Formalmente, cada entidad se describe mediante mensajes ROS de tipo `HumanEntity`, el que se presenta en el Código 5.19.

Código 5.19: bender_ltm_plugins/HumanEntity.msg

```
ltm/EntityMetadata meta

string name
time last_seen
uint8 age_bottom
uint8 age_top
uint8 age_avg
uint8 live_phase
uint8 genre
string emotion
sensor_msgs/Image face
```

5.4.3. Sesiones de demostración

Las sesiones de demostración se dividen en dos etapas: adquisición de datos episódicos y consultas episódicas al sistema. Para esto, se implementaron máquinas de estado en SMACH para ambos propósitos.

En la sesión de aprendizaje, el robot es presentado ante una cantidad indefinida de humanos, sobre los cuales debe recopilar su nombre y sus características faciales. Un mismo operador puede ser presentado nuevamente, para recopilar datos nuevos o actualizar los ya existentes. La máquina de estado desarrollada se resume en la imagen de la Figura 5.1. Esta máquina hace uso de los módulos de localización y percepción del robot.

A partir de los estados de la máquina utilizada en la sesión se aprendizaje se recopilan episodios para el sistema LTM, cada uno asociado al humano con el que interactúa el robot. Para esto, se utiliza la interfaz SMACH desarrollada en la Sección 5.3. Ya que cada estado se compone otras máquinas de menor jerarquía, cada sesión de aprendizaje genera un árbol episódico.

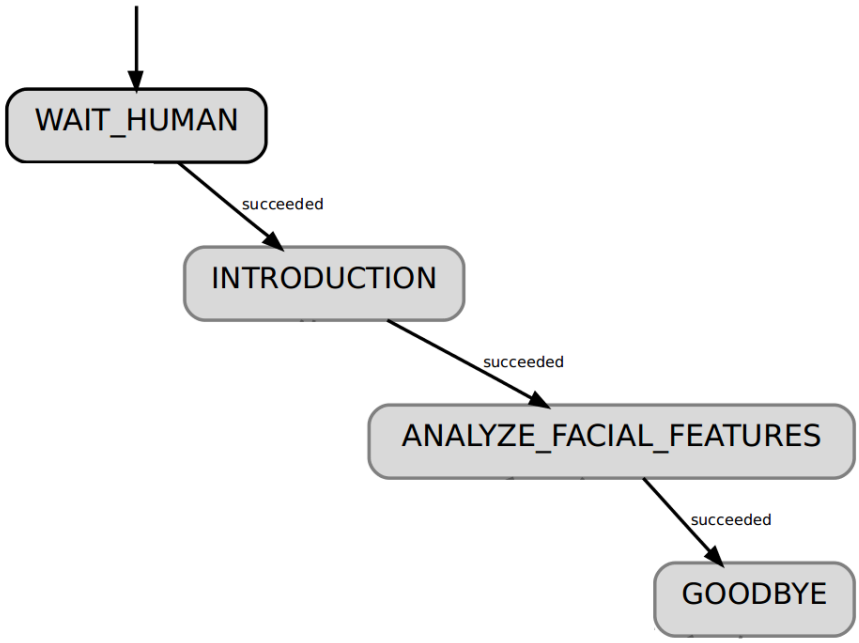


Figura 5.1: Máquina de estado construida en SMACH para que el robot Bender adquiera datos episódicos sobre humanos. En cada sesión de aprendizaje el robot espera la presencia de un humano, para luego consultar su nombre y analizar sus características faciales. Finalmente, el robot resume los datos recopilados y se despide.

Finalmente, se implementó una máquina de estado que permite acceder a los datos almacenados en las sesiones de aprendizaje. En este caso, el robot es sometido a distintas preguntas episódicas sobre los datos almacenados. Por ejemplo, respondiendo consultas como: "¿qué sucedió en el intervalo temporal T1 - T2?", "¿qué lugares conoce el robot?", "describir al humano H" o "describir al humano H, con el conocimiento del instante de tiempo T", entre otras consultas.

6 Resultados y Análisis

En este capítulo se presentan los resultados y análisis de las validaciones propuestas en el Capítulo 4 y presentadas en el Anexo A.2. En primer lugar, se revisa el cumplimiento de las funcionalidades del sistema por medio de pruebas en un robot simulado. Luego, se revisa el estado de la integración en el robot Bender, de acuerdo a las validaciones asociadas. Finalmente, se presentan los experimentos de escalabilidad y eficiencia utilizados para medir el desempeño del sistema LTM implementado.

6.1. Funcionalidad del Sistema LTM

A continuación se presentan los resultados de las validaciones de funcionalidad del sistema LTM. El listado se encuentra en el Anexo A.2 y consta de 28 validaciones, cada una con el prefijo VAXX. A modo general, cada una busca validar el cumplimiento de requisitos pertenecientes a alguna de las siguientes categorías: reglas episódicas de Stachowicz, generalidad de la implementación y el funcionamiento de las relevancias episódicas.

Para las pruebas se utilizó un robot simulado, de acuerdo a lo presentado en la Sección 5.3. El sistema LTM fue configurado para utilizar el plugin de *streams* de imágenes, el plugin de entidades tipo personas y objeto, y la interfaz episódica para SMACH. Además, los episodios son obtenidos a partir de una máquina de estado implementada mediante SMACH, que simula el flujo de estados utilizados en una prueba de la competencia RoboCup.

6.1.1. Generalidad del sistema LTM

Uno de los énfasis del diseño del sistema LTM es la generalidad del mismo. El objetivo es que el trabajo pueda ser utilizado por otros robots además de Bender, como por ejemplo, Maqui, otro robot del mismo laboratorio. A continuación se describen los conceptos de interés para lograr este objetivo, y cómo se relacionan con las validaciones.

En primer lugar, el sistema LTM soporta la definición de plugins para el manejo de estructuras episódicas de acuerdo a los requerimientos del usuario (VA02). Esto es fundamental para poder utilizar el sistema LTM en cualquier plataforma, pues evita forzar una representación de la información, dejando la adquisición de esta al usuario.

Además, la implementación fue separada en 3 grupos de paquetes ROS para desacoplar el sistema (VA03). El primero contiene el manejo de la base de datos y la implementación del servidor genérico a base de plugins. El segundo provee el plugin para streams de imágenes y la interfaz episódica para SMACH, ambos son componentes genéricos, pero no son esenciales para el funcionamiento del sistema LTM. El último provee implementaciones de plugins específicos para el robot Bender.

La implementación del servidor solo requiere un mínimo de dependencias (VA16) y no necesita módulos de Bender para su funcionamiento (VA01). Además, el servidor provee una API ROS para el manejo de episodios (VA26, 27).

Todas las validaciones asociadas se pueden verificar a partir de la implementación de plugins para el robot simulado, el que define mensajes episódicos para los streams y entidades que utiliza.

Debido a los puntos anteriores, el sistema LTM implementado se puede ajustar a las necesidades de otros trabajos robóticos, sin requerir software de Bender.

6.1.2. Reglas episódicas de Stachowicz

El otro foco del trabajo está en el cumplimiento de las reglas episódicas propuestas por Stachowicz. A modo de resumen, todas las validaciones asociadas se cumplen, a excepción de VA17. A continuación se describen los conceptos asociados a cada una de las validaciones de interés.

Los episodios son representados por un mensaje ROS con componentes genéricos. El usuario puede definir las estructuras de datos para la información semántica de interés utilizada en el campo *What*: streams y plugins. El campo *When* solo considera tiempos de inicio y fin, sin imponer restricciones sobre el lapso de tiempo. Además, no existe una restricción temporal entre un episodio y otro, lo que permite generar episodios traspuestos en el tiempo, siempre que uno no sea descendiente del otro. El campo *Where* permite almacenar la posición del robot durante cada episodio, mediante una representación literal o a través de una posición dentro de un sistema coordenado de 3 dimensiones.

La implementación permite recolectar episodios y sus componentes semánticos a partir de consultas en formato JSON, aplicando condiciones booleanas de búsqueda, a través de una API ROS. El diseño soporta la actualización de entidades y la perspectiva episódica de estas, pudiendo reconstruir el estado de una entidad en distintos instantes de tiempo. Los episodios son únicos y mantienen referencias bidireccionales a las unidades semánticas relacionadas. Los episodios se estructuran a modo de árboles, permitiendo el anidamiento episódico.

Por un lado, todos los puntos anteriores se validan mediante los conceptos descritos en el diseño (Capítulo 4) y la implementación del sistema LTM (Capítulo 5). Por otro lado, de una manera práctica, es posible validar las reglas anteriores mediante la ejecución del sistema LTM para el robot simulado, ejecutando una máquina SMACH para generar episodios, para finalmente realizar consultas episódicas a través del servicio ROS que provee el servidor LTM.

La validación VA17 no se cumple. Está relacionada al requisito de sistema RS14, el que indica que el usuario debe poder cambiar la representación del episodio en el futuro, sin sacrificar la información de episodios que utilizan la versión anterior. Particularmente, el usuario debe poder modificar los campos del mensaje ROS de los streams o entidades definidas para sus plugins. Se cree que el diseño soporta esta funcionalidad, pero no fue implementada por falta de tiempo. Queda propuesta la implementación de una funcionalidad capaz de migrar todos los episodios de la base de datos a la nueva representación episódica.

6.1.3. Relevancias episódicas

Se requiere la implementación de la relevancia emocional, histórica y generalizada. A continuación se revisan los puntos necesarios asociados a las validaciones requeridas.

El sistema LTM implementado soporta el concepto de relevancia episódica, almacenando sus datos mediante el mensaje ROS `ltm/Relevance.msg`, embebido en el mensaje `ltm/Episode.msg`. Debido a esto, es posible incluir cualquiera de los campos en las condiciones de búsqueda, permitiendo filtrar episodios por cualquiera de las relevancias (VA25).

La relevancia emocional fue implementada en el mensaje `ltm/EmotionalRelevance.msg` (VA21, 22). Sus datos son recopilados mediante un plugin implementado por el usuario. El paquete `ltm_samples` provee un plugin de ejemplo, capaz de adquirir el estado emocional del robot simulado.

En cuanto a la relevancia histórica (VA23), esta ha sido parcialmente implementada. Utiliza el mensaje ROS `ltm/HistoricalRelevance.msg` para su representación, pero no se ha implementado la rutina para actualizar automáticamente el indicador de relevancia con el paso del tiempo. Sin embargo, en el Capítulo 4 se describe su diseño y una metodología para su implementación.

Ya que la relevancia histórica aún está incompleta, también se postergó la implementación de la relevancia generalizada (VA24). Asimismo, el capítulo de diseño describe una metodología para su implementación y actualización.

Tanto la relevancia histórica como la generalizada pueden ser consideradas parte del trabajo futuro, donde solo queda implementar el algoritmo de actualización.

6.1.4. Revisión

A modo de resumen, el foco del trabajo estuvo en el diseño e implementación de un sistema LTM genérico, basado en las reglas episódicas de Stachowicz. El objetivo fue diseñar un sistema LTM para el robot Bender, pero que pueda ser integrado en otras plataformas robóticas. Las validaciones asociadas a estos conceptos se cumplen todas, a excepción de VA17, relacionada con la migración de la base de datos al modificar la representación episódica.

En cuanto a las relevancias episódicas, solo queda pendiente la implementación del algoritmo para la actualización automática de la relevancia histórica y la genérica.

Luego, las únicas validaciones de funcionalidad pendientes son: VA17, VA23 y VA24.

6.2. Integración en Bender

A continuación se describe es estado de la integración del sistema LTM en el robot Bender. Las 6 validaciones de interés se presentan en el Anexo A.2.

Durante la mayor parte del periodo de implementación del trabajo, el robot Bender no ha estado operativo. El equipo encargado ha decidido reestructurar el hardware del robot y además, está en el proceso de migrar URF a ROS kinetic. Por lo anterior, se ha retrasado la integración del sistema LTM en el robot y solo algunos de los componentes requeridos se han integrado.

La integración requiere la implementación de plugins para el sistema LTM, la generación de episodios a partir de las máquinas de estado del robot y la integración de tal código en URF.

6.2.1. Módulos integrados en URF

A continuación se describen los componentes integrados en Bender. Estos corresponden a piezas de software que no requieren el uso del robot para su integración, o son suficientemente genéricos como para utilizar una versión simulada.

Plugin para streams: Debido al diseño genérico del plugin para la recolección de streams de imágenes, este componente fue integrado mediante la selección del tópico ROS de la cámara de video del robot.

Plugin de para campo *Where*: Se creó un plugin para la recolección de la ubicación del robot según las consideraciones de diseño descritas en el capítulo 4. Esta integración fue posible, debido a que Bender utiliza un conjunto estándar de paquetes ROS para navegación robótica (ROS navigation stack). Su funcionamiento fue verificado mediante la simulación de un robot que utiliza estos paquetes para navegar y localizarse.

Plugin para entidades: Se implementó un plugin para la recopilación de información facial sobre humanos. Todos los datos son obtenidos a partir del sistema de percepción del robot Bender. Esto se puede validar mediante la demostración implementada durante la integración.

Interfaz LTM-SMACH: Según se presentó en el Capítulo 6, la interfaz del sistema LTM con las máquinas de estado en SMACH se implementó de manera genérica y funciona de acuerdo a los requerimientos. Para validar esta integración, se implementó una máquina de estado similar a una prueba de la competencia RoboCup, con la que el sistema LTM puede recolectar episodios. Además, esto se valida mediante las sesiones de aprendizaje y consultas utilizadas para la demostración implementada.

Integración en URF: Todos los componentes descritos anteriormente fueron integrados en URF mediante el paquete ROS `bender_ltm`. El proceso de instalación fue modificado para incluir los paquetes ROS del sistema LTM y la instalación de sus dependencias. Se añadieron archivos `launch` y de configuración para ejecutar el sistema LTM con los plugins integrados.

6.2.2. Módulo pendiente

Plugin emocional: Este plugin es el único módulo que no ha sido implementado ni integrado a Bender, por dos razones. Como se explicó anteriormente, no se pudo acceder al robot para la integración. Sin embargo, el problema de fondo es que Bender no dispone de un motor emocional, capaz de generar emociones a través de los estímulos que percibe. La implementación de un sistema emocional escapa del alcance del trabajo de título. Sin ese requerimiento es imposible implementar el plugin requerido.

Por lo tanto, la integración del sistema LTM en Bender es parcial. Queda propuesta la implementación del plugin para entidades, el motor emocional para el robot y el plugin asociado.

6.2.3. Demostración del sistema en Bender

Todos los módulos descritos anteriormente son validados simultáneamente mediante la demostración implementada para Bender. Esta permite almacenar secuencias de episodios, en los cuales el robot se encuentra en un lugar (*Where*) y tiempo (*When*) específicos, interactuando y recolectando datos (*What*) sobre humanos.

6.3. Desempeño del Sistema

A continuación se presentan los resultados para las validaciones de eficiencia y escalabilidad definidas en el Capítulo 4, cuyo objetivo es medir el desempeño del sistema para un conjunto de consultas de interés. En primer lugar, se describen los procesos a ser estudiados

y la máquina objetivo. Luego, se identifica el conjunto de operaciones de lectura y escritura a ser utilizado para la evaluación. Tercero, se hace una estimación sobre la cantidad de episodios y la tasa de consultas a evaluar según el caso de estudio. Más adelante, se presenta la metodología de medición utilizada, para finalmente, presentar los resultados y análisis de las pruebas realizadas.

6.3.1. Consideraciones

En esta sección se evalúa el desempeño del sistema LTM, de acuerdo a las validaciones VCX presentadas en el Anexo A.2.

Procesos: Una vez activo, el sistema LTM se compone de dos procesos ejecutados simultáneamente en la máquina objetivo. Por un lado, se ejecuta el proceso correspondiente al servidor LTM (en adelante, LTMp), encargado de procesar las consultas ROS, ejecutar los plugins definidos por los usuarios y de mantener la consistencia de la base de datos. Por otro lado, se requiere que el servidor de MongoDB (en adelante MongoDBp) esté activo para responder a las consultas del LTMp. En adelante, se hará la distinción entre LTMp y MongoDBp al hablar de las validaciones a realizar. Además, se hablará de *sistema LTM* para referirse a la combinación de ambos procesos.

La distinción anterior es importante, pues permitirá entender donde se encuentran los puntos críticos donde el procesamiento se estanca e identificar las responsabilidades de cada proceso. Además, se considera necesario enfatizar que todas las consultas de lectura en el sistema LTM son entregadas directamente a la librería de MongoDB para C++, utilizada por LTMp, por lo que se espera que el mayor costo de recursos se atribuya a MongoDBp, a tal librería y al formato de las consultas. Por otro lado, se espera que LTMp incremente el uso de recursos, de acuerdo a la cantidad de plugins utilizados y la calidad de su implementación.

Finalmente, tanto LTMp como MongoDBp son procesos que solo ocupan 1 núcleo de la máquina en donde residen. Por lo tanto, en el peor caso, las pruebas de eficiencia indicarán el uso de dos núcleos del CPU. Sin embargo, es posible que algunos plugins LTM, implementados por el usuario, requieran del uso de más de 1 núcleo.

Especificaciones de la máquina: En el caso de Bender, URF se ejecuta en 3 computadores simultáneamente. A pesar esto, las pruebas se realizaron en solo uno de los computadores, el que tiene las siguientes especificaciones de interés:

- CPU: Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz
- Memoria RAM: 15.6 GB (Memoria primaria).
- Disco Duro: HDD con 400GB de capacidad (Memoria secundaria).

6.3.2. Consultas de interés

De acuerdo a lo revisado en la sección 4.4, las operaciones CRUD más relevantes son las de creación y lectura de episodios.

Inserción de episodios: La operación de creación se realiza en un proceso de dos etapas, registro e inserción. Durante el registro, LTMp se encarga de generar un identificador único para el episodio y avisar a los plugins. Este proceso es rápido y no se considera necesario realizar una medición de su impacto en el sistema. Por otro lado, la etapa de inserción se encarga de actualizar el árbol episódico y de almacenar los cambios en la base de datos. Este proceso puede ser costoso y se considera interesante para el estudio.

Luego, la consulta de interés es la de inserción de episodios, la que se realiza a través del servicio ROS `~/ltm/episode/add`. En los gráficos de resultados esta operación siempre se presentará mediante una línea negra, marcada con símbolos `+` por cada muestra obtenida.

Búsqueda de episodios: Asimismo, la operación de lectura de episodios se realiza a través de un proceso de dos etapas, búsqueda en la base de datos y recopilación de documentos con los episodios de interés. En este caso, la operación de interés es la de búsqueda de episodios dada una consulta expresada mediante distintas condiciones booleanas. Según la cantidad de episodios y las condiciones de búsqueda, esta consulta puede ser más o menos costosa.

Luego, la consulta de interés es la de búsqueda de episodios, la que se realiza a través del servicio ROS `~/ltm/db/query`. Las condiciones de búsqueda se expresan en formato JSON, siguiendo la sintaxis para consultas utilizado por MongoDB.

Ya que las condiciones de búsqueda pueden afectar el desempeño del sistema, se define un conjunto de 5 consultas JSON, con el fin de evaluar el sistema LTM bajo distintos casos de uso. Las consultas se presentan en el Código 6.1 y se describen a continuación:

- Q_1: Consulta que filtra episodios a partir de una condición de igualdad entre un campo y un valor numérico de tipo entero.
- Q_2: Consulta que filtra episodios donde un valor particular se encuentra dentro de un campo de tipo lista de *string*.
- Q_3: Consulta que filtra episodios donde un campo de tipo *string* coincide con alguno de los que se provee en la lista que se provee.
- Q_4: Consulta que filtra episodios mediante dos condiciones unidas por el operador lógico OR. Una de las consultas evalúa una condición de orden (mayor qué) para un valor de tipo decimal. La otra consulta es idéntica a Q_2.
- Q_5: Consulta que filtra episodios mediante 3 condiciones de búsqueda, anidadas por operadores lógicos OR y AND. Las consultas internas son similares a Q_2 y a la de orden utilizada en Q_4.

En adelante, se referirá a cada consulta por su abreviación en el formato Q_x, donde x co-

rresponde al número de la consulta presentado anteriormente. En los gráficos, se utilizará siempre el mismo color y estilo de línea para representar a una misma consulta.

Código 6.1: Consultas JSON utilizadas para validación

```
# Q1
{ "uid": 10 }

# Q2
{ "children_tags": "a_tag" }

# Q3
{ "where.location": { "$in": [ "location_a", "location_b" ] }}

# Q4
{
  "$or": [
    { "relevance.emotional.value": { "$gt": 0.3 } },
    { "children_tags": "a_tag" }
  ]
}

# Q5
{
  "$or": [
    { "children_ids": 10 },
    { "uid": { "$gte": 15, "$lte": 20 }}
  ],
  "relevance.emotional.value": { "$gt": 0.2 }
}
```

6.3.3. Estimaciones

Para poder ejecutar las validaciones de escalabilidad y eficiencia es necesario conocer el uso esperado del sistema LTM en el robot objetivo. Acá se presentará una estimación del orden de magnitud para la cantidad de episodios y la tasa de consultas necesarias para las validaciones, dado un caso de uso esperado.

Caso de uso: El robot objetivo de este trabajo es Bender, el que se entiende como un robot de laboratorio, con un tiempo de operación diario muy bajo. Se tienen las siguientes consideraciones:

- El equipo suele trabajar solamente en los tiempos libres durante los días de clases. Es decir, 5 días a la semana, durante 32 semanas de clases al año.
- El robot no es utilizado todos los días, sino que de manera intermitente, siendo encendido por cortos periodos de tiempo.
- De las veces que el robot está en funcionamiento, solo muy pocas veces se encuentra ejecutando una máquina de estado.

Por lo anterior, los requerimientos impuestos para las validaciones son bajos. Sin embargo, es deseable que el sistema LTM sea capaz de soportar una carga más exigente de episodios y consultas, por el ejemplo, asumiendo que el robot funciona durante todo el horario laboral

de manera ininterrumpida.

Cantidad de episodios: Según el caso de uso esperado, se hace una estimación de la cantidad de episodios a recolectar, para soportar el uso del sistema LTM de manera continua durante un periodo de 1, 5 y 10 años.

Para determinar la cantidad de episodios, se estima que en una hora el robot es capaz de realizar 3 sesiones de trabajo. Por cada sesión, se almacena un árbol con 10 episodios.

En la Tabla 6.1 se presentan los valores de cada concepto utilizado para la estimación. Se debe enfatizar que los valores seleccionados son solo válidos para este estudio y la cantidad real de episodios dependerá del uso dado al sistema LTM.

Concepto	Valor Estimado
Semanas / Año	32
Días / Semana	5
Horas / Día	3
Árboles / Hora	3
Episodios / Árbol	10
Episodios en 1 año	14.400
Episodios en 5 años	72.000
Episodios en 10 años	144.800

Tabla 6.1: Estimación de la cantidad de episodios necesarios para las evaluaciones de escalabilidad, bajo el caso de uso del sistema LTM en el robot Bender.

Cantidad de consultas por minuto: La cantidad de consultas por minuto esperadas para el sistema LTM, en adelante *CPM*, dependerá mucho del uso del robot. A continuación se hace una estimación gruesa sobre las CPM requeridas para las operaciones de inserción y búsqueda de episodios, se acuerdo a la experiencia que el autor tiene con el robot.

Para el caso de la inserción de episodios, siguiendo las estimaciones de escalabilidad, con 3 árboles/hora y 10 episodios/árbol se estima 1 CPM para consultas de inserción. Sin embargo, es posible que los procesos de inserción se ejecuten de manera consecutiva en un corto intervalo de tiempo, por lo que se propone un valor de 60 CPM.

Para el caso de las búsquedas, la tasa de consultas depende completamente del uso que le de el desarrollador. A modo de experimento, se asumirá que es deseable una tasa de 10 consultas por segundo, es decir, 600 CPM. Sin embargo, una tasa de 10 CPM puede ser aceptable en la mayoría de los casos, asumiendo que el robot debe responder consultas realizadas por un humano durante una conversación.

6.3.4. Metodología de medición

A continuación se detalla la metodología para la generación de episodios, consultas y la medición de las variables de interés para las validaciones requeridas.

Generación de episodios: Tanto las validaciones de escalabilidad como las de eficiencia requieren que la base de datos ya contenga un conjunto de árboles con episodios. La estrategia utilizada es la siguiente:

1. Se crea una base de datos vacía para cada prueba.

2. Se inserta un árbol de 10 episodios (tamaño esperado). El árbol es de profundidad 3, tiene 1 raíz, 3 nodos internos y 6 hojas. Todos los episodios son generados con campos aleatorios para cada tipo de dato requerido.

3. Según se requiera se repite el paso 2, hasta obtener la cantidad de episodios deseada.

Generación de consultas: Con el fin de evitar sesgar las mediciones por motivos del caché de búsqueda, los valores de cada campo de las consultas Q_x son generados al azar. Todos los valores generados corresponden al rango de valores posibles para el campo en cuestión. Por ejemplo, para el campo `children_tags`, los episodios generados contienen valores del conjunto $\{tag_1, \ldots, tag_{1000}\}$, entonces la consulta Q_2 solo utilizará valores al azar obtenidos desde el conjunto correspondiente. Para cada campo se tienen al menos 100 valores y a lo más 1.000 valores posibles.

Escalabilidad: Las validaciones de escalabilidad buscan medir el tiempo promedio que toma una consulta en ser ejecutada. Para esto, cada consulta es repetida 100 veces, donde se mide el tiempo que el programa queda bloqueado durante la llamada al servicio ROS correspondiente. Las mediciones son repetidas cada intervalos de 10.000 episodios ingresados al sistema LTM.

Además, se mide la cantidad de memoria secundaria requerida para soportar los episodios en la base de datos. Para esto, se realiza una consulta a MongoDB mediante la librería `pymongo 3.6.1` para Python 2.7. Todos los valores se obtienen través del comando `dbStats`[1]. Se realizan las siguientes mediciones:

- Uso total de memoria secundaria. Se obtiene del comando `dbStats.fileSize`.

- Espacio total reservado para documentos, es decir: episodios, streams y entidades. Se obtiene del comando `dbStats.storageSize`.

- Espacio utilizado por los episodios. Se obtiene del comando `dbStats.dataSize`.

- Espacio utilizado para el indexado de los documentos. Se obtiene del comando `dbStats.indexSize`.

Eficiencia: Las validaciones de eficiencia buscan medir el uso de recursos del sistema (CPU y RAM) al variar las CPM, dada una base de datos con una cantidad fija de episodios. De la misma forma, cada consulta es repetida al menos 5 veces y durante al menos 1 minuto. Además, la medición del uso de recursos se realiza cada 3 segundos, para obtener el promedio de las muestras medidas. Para mantener las CPM, el programa de medición es pausado tras

[1]En MongoDB, el comando `dbStats` permite obtener estadísticas generales de la base de datos. Más información en la página oficial: https://docs.mongodb.com/manual/reference/command/dbStats/

cada consulta, durante el tiempo requerido para mantener la tasa de consultas deseada.

La mediciones sobre el uso de CPU y RAM para cada proceso fueron obtenidas utilizando la librería `psutil 3.4.2`[2] para Python 2.7.

6.3.5. Validaciones de Escalabilidad

A continuación se presentan los resultados de las mediciones de escalabilidad y el análisis relacionado a cada uno. Estas mediciones buscan medir los costos temporales de las operaciones y el uso de memoria secundaria, según la cantidad de episodios almacenados en el sistema LTM.

Ejecución de mediciones: Se implementó un nodo ROS para las mediciones de escalabilidad del sistema LTM. Este se encarga de crear la base de datos, generar episodios, consultas y realizar las mediciones. El nodo almacena los resultados en un archivo en formato CSV. El nodo se encuentra en archivo `scalability_test.py` del paquete ROS `ltm_samples`, por lo que puede ser ejecutado mediante el siguiente comando: `$ rosrun ltm_samples scalability_test.py`.

A partir de los resultados, un nodo ROS es utilizado para la generación de los gráficos presentados. De la misma manera, este nodo se puede ejecutar mediante el siguiente comando: `$ rosrun ltm_samples graphs_scalability.py`.

Muestras recopiladas: Según se estimó, para un lapso de 10 años el robot Bender recopilaría cerca de 144.000 episodios. Para establecer un factor extra de seguridad en la estimación, se realizaron mediciones con hasta 500.000 episodios, por lo que cada gráfico está compuesto de 50 muestras.

Uso de memoria secundaria: Los resultados son presentados en el gráfico de la Figura 6.1. Por un lado, se ve que el espacio total de memoria secundaria utilizado por MongoDB, es reservado en porciones cada vez mayores. Por otro lado, del espacio reservado, la porción utilizada para el almacenamiento de episodios e índices crece de manera lineal, respecto a la cantidad de episodios almacenados.

En sus inicios, el sistema LTM utiliza hasta 0.25 GB para almacenamiento, mientras que a los 10 años (200.000 episodios) llega a utilizar 1 GB en documentos, indices y espacio reservado.

Se considera que tal cantidad de memoria secundaria no es un problema para las cantidades de espacio disponibles en los computadores actuales. Sin embargo, esta evaluación solo considera el almacenamiento de episodios, sin streams ni las entidades asociadas. Se espera que el uso de disco aumente de acuerdo a los tipos de dato definidos por el usuario para cada uno de los plugins.

[2]La librería `psutil` permite el monitoreo de procesos y del sistema en múltiples sistemas operativos. Más información en su web oficial: https://pypi.org/project/psutil/

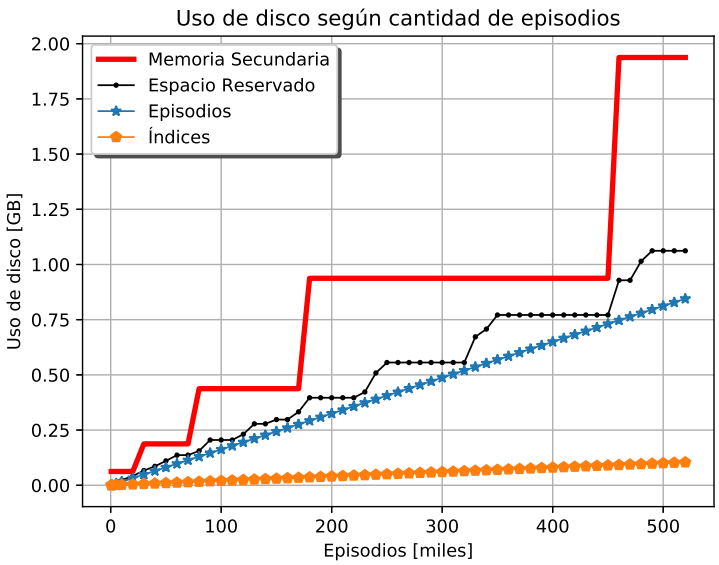


Figura 6.1: Prueba de escalabilidad sobre el uso de memoria secundaria según la cantidad de episodios en la base de datos.

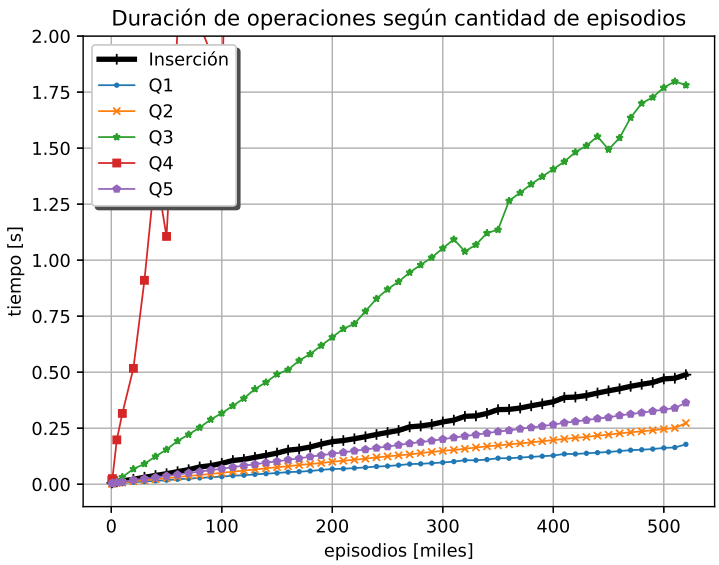
Particularmente, en el caso de los streams, si se almacenan imágenes en alta resolución a una alta frecuencia, es esperable que el consumo de memoria secundaria sea mucho mayor al mostrado en este experimento. En el caso de las entidades, si estas no contienen imágenes u otros datos binarios, es esperable que tengan un comportamiento similar al de los episodios.

Duración de consultas: Los resultados son presentados en los gráficos de la Figura 6.2. Ambos gráficos presentan las mismas mediciones y solo difieren en la escala y límites del eje temporal.

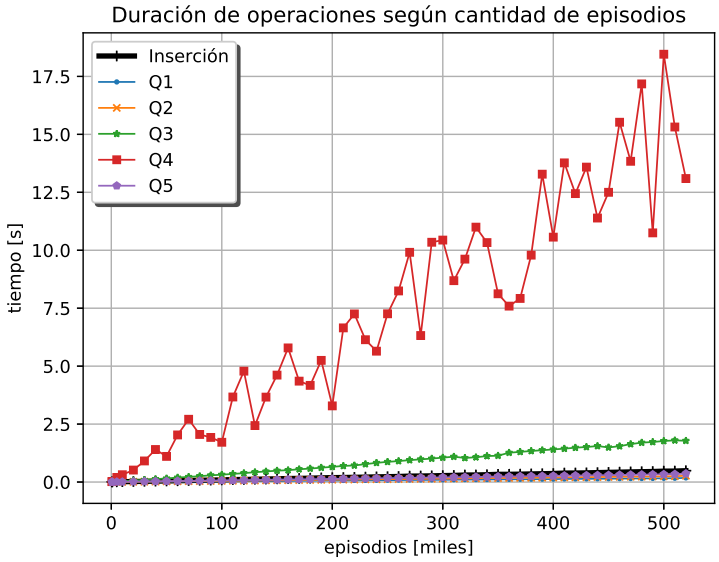
En el caso de la inserción, se puede ver que el costo temporal crece de manera lineal respecto a la cantidad de episodios almacenados. A los 300.000 episodios la consulta requiere alrededor de 0.25[s], mientras que a los 500.000 la consulta toma 0.5[s]. Dado los costos descritos, se impone una cota inferior para la duración de los episodios a generar, pues no será posible almacenar una secuencia de episodios de 0.1[s] cada uno, cuando el sistema LTM requiere más tiempo para su procesamiento. De todas formas, el costo temporal de la inserción es considerado bajo para el caso de uso de Bender.

Para el caso de la búsqueda, se puede ver que todas las consultas, a excepción de Q_4, tienen un costo temporal que crece de manera lineal respecto a la cantidad de episodios almacenados. Las consultas Q_1, Q_2 y Q_5 siempre se mantienen bajo los 0.25[s], mientras que Q_3 tiene un costo de 1.0[s] a los 300.000 episodios. En cuanto a Q_4, se podría decir que el costo se comporta de manera lineal, pero las mediciones toman demasiado tiempo en ser ejecutadas y tal vez se requieran más muestras para obtener un valor promedio menos variable. Además, Q_4 toma 10[s] a los 300.000 episodios, 10 veces más que Q_3 y cerca de 40 veces más que Q_1, Q_2 y Q_5.

De lo anterior se puede ver que distintas consultas tienen costos de búsqueda muy distintos, los que pueden llegar a degradar el funcionamiento del robot, como es el caso de Q_4. Se cree



(a)



(b)

Figura 6.2: Prueba de escalabilidad sobre la duración de las operaciones de inserción y búsqueda de episodios en la base de datos según la cantidad de episodios almacenados. Ambos gráficos presentan las mismas mediciones, pero para distintas escalas del eje temporal.

que se debe estudiar en más detalle el costo de todas las operaciones disponibles en el lenguaje de consulta de MongoDB y su interacción con los operadores lógicos, para así disponer de medidas empíricas sobre las cuales diseñar consultas de menor costo.

En cuanto al impacto en la interacción humano robot, es esperable que Bender tenga un funcionamiento ininterrumpido. En este caso, una demora de 0.5[s] en la inserción o búsqueda de episodios puede no ser problema, mientras que en el caso de la consulta Q_4, una demora

de 10[s] puede comprometer el interés y las expectativas del humano con quien se interactúa. Por lo tanto, en el caso de realmente requerir una consulta compleja como lo es Q_4, se recomienda avisar a la persona que el robot deberá procesar la consulta (o "pensar") durante unos segundos, para así evitar comprometer la interacción.

Finalmente, una manera simple y alternativa para disminuir los costos de tiempo para las operaciones del sistema es adquirir componentes de hardware con mejor desempeño. Por ejemplo, se podría considerar el uso de memoria secundaria de tipo SSD, un CPU con mayor caché y frecuencia de procesamiento, y memoria primaria más veloz.

Límite para las CPM: Finalmente, a partir de los costos en tiempo para cada operación, revisados en la Figura 6.2, se puede calcular la tasa máxima de CPM para cada una de las operaciones. Los resultados se presentan en el gráfico de la Figura 6.3.

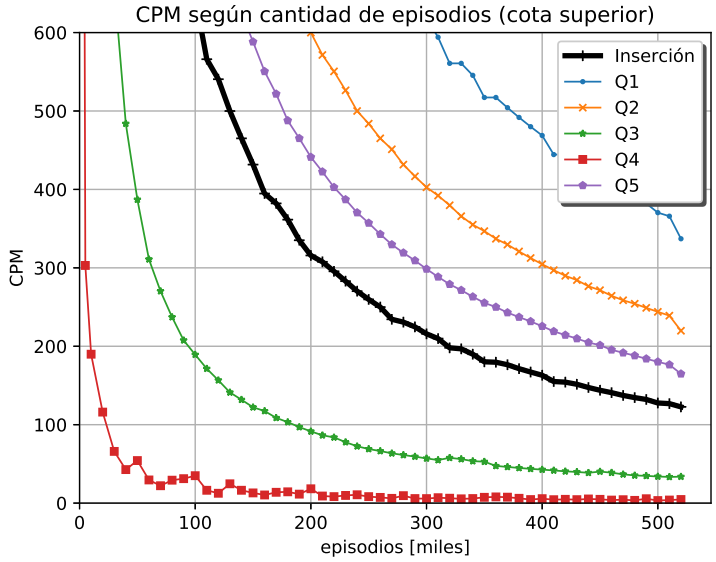


Figura 6.3: El gráfico muestra la cota superior para las CPM posibles para cada operación según la cantidad de episodios almacenados en la base de datos.

Estos resultados son importantes para entender como decrece la tasa de CPM a medida que aumenta la cantidad de episodios almacenados. Además, sirven para comprender el comportamiento de los resultados de eficiencia para distintas cantidades de episodios.

6.3.6. Validaciones de Eficiencia

A continuación se presentan los resultados de las mediciones de eficiencia y el análisis relacionado a cada uno. Estas mediciones buscan medir el uso de RAM y de CPU según la cantidad de consultas por minuto (CPM) a efectuar.

Ejecución de mediciones: Se implementó un nodo ROS para las mediciones de eficiencia del sistema LTM. Este se encarga de crear la base de datos, generar episodios, consultas y

realizar las mediciones. El nodo almacena los resultados en un archivo en formato CSV. Su implementación se encuentra en el archivo `efficiency_test.py` del paquete ROS `ltm_samples`, por lo que puede ser ejecutado mediante el siguiente comando: `$ rosrun ltm_samples efficiency_test.py`.

A partir de los resultados un nodo ROS es utilizado para la generación de los gráficos presentados. Puede ser ejecutado mediante el siguiente comando: `$ rosrun ltm_samples graphs_efficiency.py`.

Presentación de resultados: En estos experimentos se mide el uso de recursos según la cantidad de CPM utilizadas para distintas cantidades de episodios. Además, ya que se tienen 5 operaciones de búsqueda ($Q_1, \ldots, Q_5$), los resultados se dividirán en dos conjuntos. Primero, se presentan gráficos con una cantidad fija de episodios y los resultados para cada operación. Segundo, se presentan gráficos particulares para cada consulta, con curvas para distintas cantidades de episodios. Finalmente, ya que existen dos procesos de interés, cada conjunto muestra el uso de recursos de MongoDBp, LTMp y del sistema LTM (suma de MongoDBp y LTMp).

Muestras recopiladas: En este caso se presentan mediciones con hasta 100.000 episodios. En cuanto a las CPM, se estimó que se requieren hasta 600 CPM, pero para establecer un factor extra de seguridad en la estimación se realizaron mediciones con hasta 1.000 CPM, espaciadas cada 25 CPM. Por lo tanto, cada gráfico dispone de 40 muestras.

Uso de RAM según CPM: Se midió el uso de RAM según la cantidad de CPM para distintas cantidades de episodios y para cada consulta. De los resultados se observó que el uso de RAM no es afectado por la consulta efectuada ni la cantidad de CPM, sino que solo por la cantidad de episodios. El gráfico correspondiente se presenta en la Figura 6.4, el que muestra el uso de RAM en el sistema LTM para cada consulta evaluada.

De los experimentos se entiende que el consumo de RAM debe ser visto como parte de las medidas de escalabilidad y no de eficiencia. Para los 100.000 episodios (alrededor de 5 años de uso) el sistema LTM consume aproximadamente 500 MB de RAM. Sin embargo, se cree necesario conocer la curva de escalabilidad del consumo de RAM, para poder entender el uso de recursos en mayor detalle.

Uso de CPU según CPM, cantidad fija de episodios: En la Figura 6.5 se presentan 6 gráficos con las mediciones de eficiencia del uso de CPU según la cantidad de CPM, para cada consulta y manteniendo fija la cantidad de episodios. Se hace una separación por cada proceso y la suma de ambos, y para 10.000 y 100.000 episodios.

En el peor caso, LTMp utiliza $\sim 10\,\%$ del CPU y MongoDBp el $25\,\%$ del CPU. Es decir, MongoDBp utiliza completamente uno de los 4 núcleos disponibles, mientras que LTMp utiliza el $\sim 40\,\%$ de otro. Sin embargo, ambos límites no se alcanzan de manera simultánea, sino que en el peor caso el sistema consume hasta un $\sim 27\,\%$ del total del CPU.

El máximo uso de CPU solo se alcanza a una tasa muy alta de CPM (cerca de 1.000 CPM) o para los 100.000 episodios. En el último caso, el CPU se satura con menos de 100 CPM para las consultas Q_3 y Q_4. El resto de las consultas soportan una alta tasa de CPM

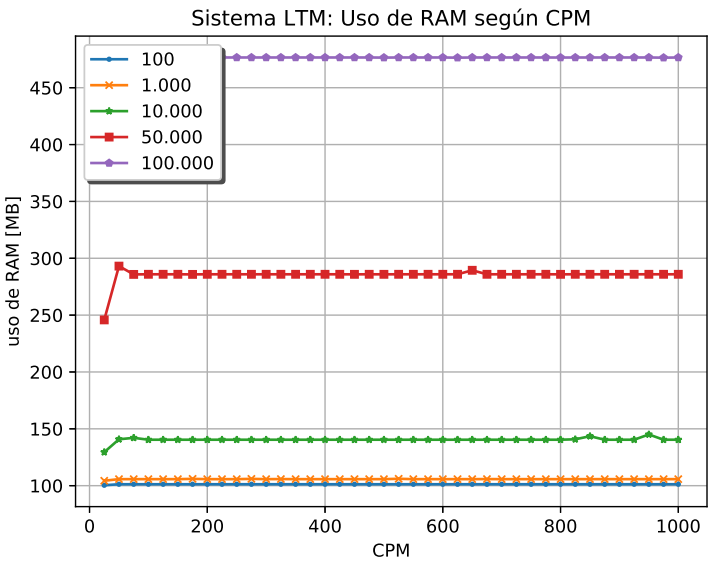


Figura 6.4: Prueba de eficiencia sobre el uso de RAM en el sistema LTM según la cantidad de CPM, para distintas cantidades de episodios. Los resultados presentados son independientes de la consulta efectuada.

sin llegar a saturar uno de los núcleos.

Del total de CPU utilizado en cada consulta, el mayor costo se atribuye al proceso MongoDBp, el que llega a saturar el núcleo utilizado. El proceso LTMp requiere hasta un $\sim 10\,\%$ de CPU, el que se atribuye al procesamiento de la consulta realizado por la librería de MongoDB para C++, pues la implementación del servicio bloquea el programa hasta que la consulta es respondida por la librería. Entonces, el costo de CPU se debe principalmente a la base de datos, a la librería de MongoDB y a la estructura de la consulta efectuada.

Finalmente, es posible ver que el costo de CPU incrementa de manera lineal respecto a la cantidad de CPM, a excepción de Q_4. Además, se ve una relación directa entre el costo de CPU y el tiempo de consulta de las pruebas, presentado en los resultados de escalabilidad. Las consultas de menor costo en duración son las menos costosas en términos de CPU, mientras que Q_3 y Q_4 son más costosas en ambos ámbitos.

Uso de CPU según CPM, revisión por cada consulta: En la Figura 6.6 se presenta un gráfico por cada consulta estudiada. En cada caso se evalúa el costo de CPU según la cantidad de CPM para distintas cantidades de episodios. Solo se presentan los resultados para la suma de ambos procesos de interés, pues ya se ha evaluado el impacto de cada proceso por separado. Estas visualizaciones se obtienen de las mismas mediciones que las de la Figura 6.5, pero permiten evaluar la eficiencia desde otro punto de vista.

Las visualizaciones reafirman que el uso de CPU para las consultas Q_1, Q_2, Q_3 y Q_5 crece de manera lineal con la cantidad de CPM y para distintas cantidades de episodios. Ya que MongoDBp solo utiliza un núcleo del computador, tal crecimiento se estanca al saturar su capacidad.

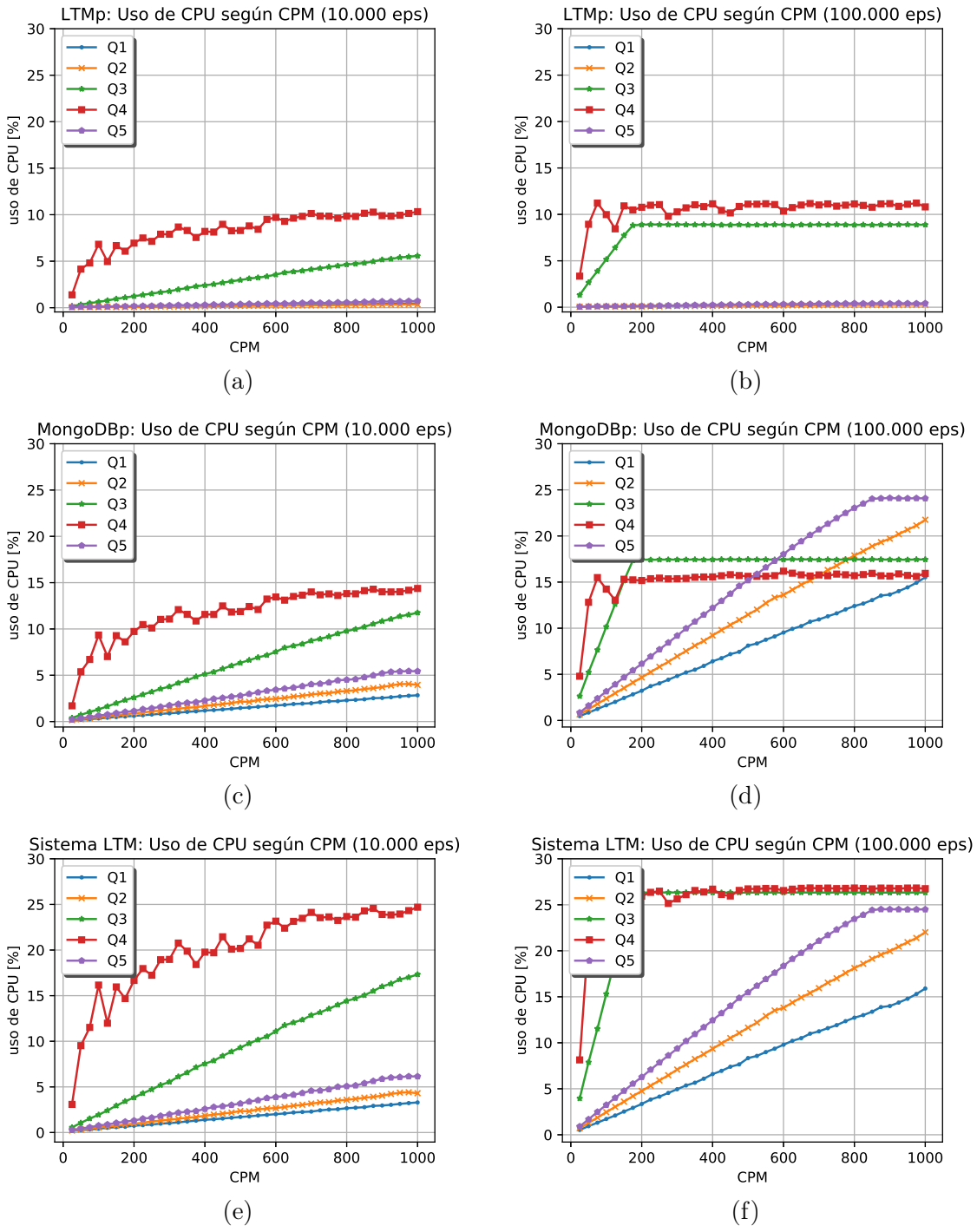
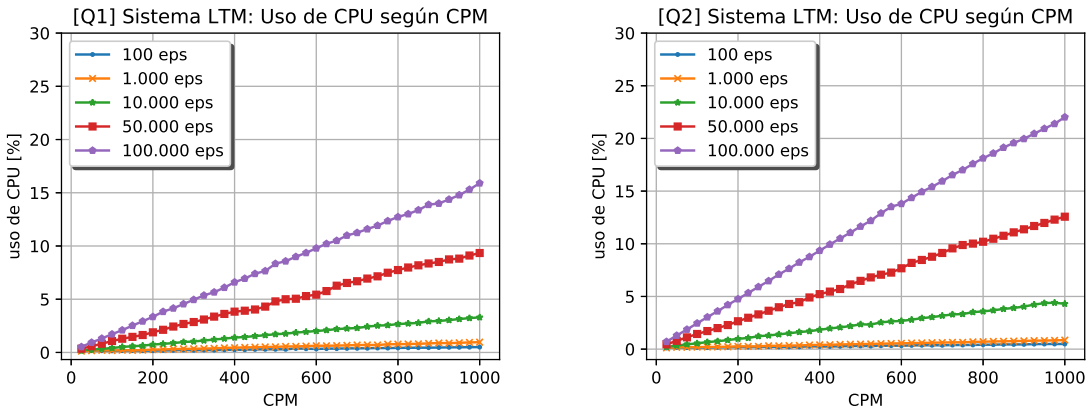


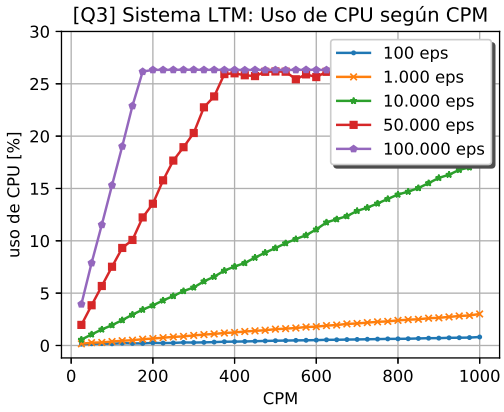
Figura 6.5: Pruebas de eficiencia sobre el uso de CPU según la cantidad de CPM, bajo una cantidad fija de episodios. Se presentan 6 gráficos para distintas configuraciones, cada uno con curvas para cada consulta Q_x. A la izquierda hay pruebas con 10.000 episodios, a la derecha con 100.000 episodios. La primera fila muestra el uso de CPU del proceso LTMp, la segunda corresponde al uso de CPU del proceso MongoDBp y la tercera muestra la suma de ambos procesos.

Además, se reafirma que las consultas Q_1, Q_2 y Q_5 tienen bajo costo de CPU respecto a las otras consultas y escalan linealmente incluso para los 100.000 episodios. Por otro lado, Q_3 y Q_4 son consultas costosas, y particularmente, Q_4 ya presenta problemas a los 10.000

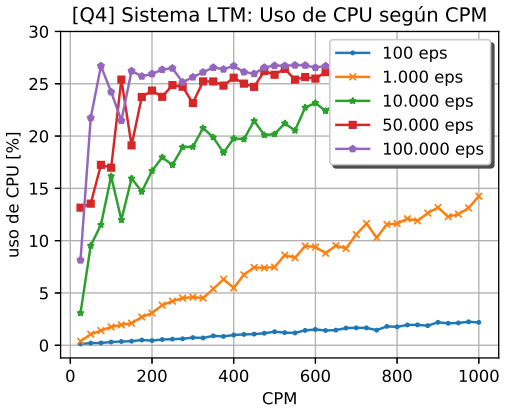


(a)

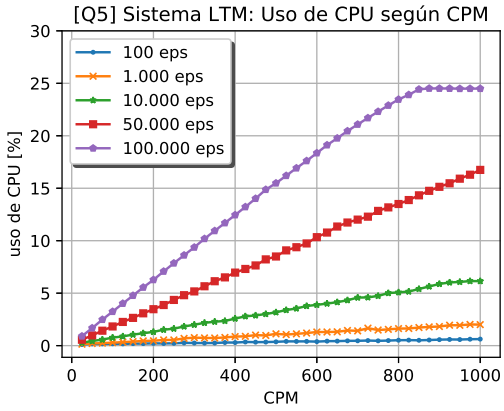
(b)



(c)



(d)



(e)

Figura 6.6: Pruebas de eficiencia sobre el uso de CPU según la cantidad de CPM. Se presenta un gráfico para cada consulta Q_x. Cada uno muestra mediciones para distintas cantidades de episodios. Los valores presentados corresponden a la suma del uso de recursos por los procesos LTMp y MongoDBp.

episodios, el equivalente a 4 meses según la estimación.

El experimento de eficiencia realizado muestra que la selección de operaciones lógicas para una consulta es importante y tiene un impacto en el desempeño del sistema. Esto

puede marcar la diferencia entre la utilidad del sistema LTM y su impacto en la interacción humano-robot.

Finalmente, se cree que la evaluación de eficiencia realizada puede ser mejorada con la consideración de hasta 300.000 episodios. Además, se puede complementar el experimento con la evaluación de la consulta de inserción y la consideración de distintos casos de uso para streams y entidades.

Conclusión

En este trabajo se diseñó e implementó una memoria episódica de largo plazo para robots domésticos basados en ROS. El trabajo estuvo enfocado en el diseño de un sistema LTM adecuado a Bender, pero capaz de ser integrado en otras plataformas robóticas que utilizan ROS.

En primer lugar, se estudió la relevancia de una memoria a largo para un robot doméstico. Este concepto es esencial para mejorar el desempeño de las tareas que debe realizar el robot. Más aún, esta capacidad tiene un alto impacto en la interacción humano-robot, permitiendo comportamientos interesantes y evitando la pérdida de interés del operador. Es decir, la memoria LTM es esencial para un robot doméstico.

Según se investigó, hay pocos trabajos enfocados en este tema y aún no existe un consenso respecto a la implementación de un sistema LTM. Sin embargo, algunos trabajos siguen una serie de requerimientos básicos para un sistema de este tipo. El diseño estuvo fuertemente influenciado por los requerimientos episódicos propuestos por Stachowicz, que rigen el comportamiento del sistema y las cualidades esperadas del mismo.

De los objetivos del trabajo se derivaron requisitos de sistema y validaciones con las que guiar el diseño e implementación. El sistema LTM diseñado permite recolectar episodios a través de máquinas de estado, las que son utilizadas comúnmente para la implementación de comportamientos en robots domésticos. Cada episodio está asociado a información semántica, en forma de streams y entidades, cuya representación debe ser definida por el usuario y sus datos recopilados mediante un sistema de plugins implementados por él. De esta forma, el sistema LTM puede ser integrado en Bender, pero también en otros robots basados en ROS. El costo de la generalidad que provee el sistema es que se dejan muchas decisiones de implementación al usuario, lo que puede ser problemático cuando hay errores en un plugin, o para asegurar un desempeño similar en cada caso de uso.

El sistema LTM provee una API ROS para su ejecución, configuración y recolección de episodios. Utiliza la base de datos MongoDB para el almacenamiento de episodios. Utilizando el lenguaje de consulta de MongoDB, el sistema permite realizar búsquedas de episodios

mediante cualquier combinación de condiciones lógicas.

Para el caso de uso de Bender, los experimentos muestran que el costo temporal de las consultas episódicas escala linealmente respecto a la cantidad de episodios almacenados, siendo capaz de responder un conjunto de consultas de interés en menos de 0.25[s] a los 10 años de uso. En términos de eficiencia, el sistema LTM es capaz de mantener una tasa de 600 consultas por minuto, llegando a utilizar el 20 % de CPU y de 200 consultas por minuto, utilizando el 5 %. En ambos experimentos, el sistema LTM cumple los requerimientos para el caso de uso estudiado, sin embargo, queda claro que la consulta a utilizar debe ser elegida adecuadamente, pues la duración de esta puede comprometer la fluidez de una interacción humano-robot y además, puede saturar el núcleo de CPU asignado.

El sistema LTM cumple con todos los requisitos funcionales establecidos, a excepción de 2 puntos. Primero, aún no se implementa la capacidad de migrar la base de datos a una nueva estructura semántica. Segundo, se requiere un algoritmo para la actualización automática de la relevancia histórica y generalizada, esto se revisa en el capítulo de diseño, pero no ha sido implementado. Ambos puntos permitirían concluir la implementación y se consideran parte del trabajo futuro.

Respecto a la integración del sistema LTM en el robot Bender, esta es parcial. Se integraron correctamente los plugins para recolección de imágenes y de la localización del robot, junto a la interfaz para recopilar episodios a partir de máquinas de estado en SMACH. Además, Bender es capaz de recopilar datos episódicos sobre interacciones con humanos, mediante una demostración exitosa del sistema LTM. Por otro lado, se realizaron las modificaciones requeridas en URF para instalar y ejecutar el sistema junto al robot. Sin embargo, ya que no se pudo acceder al robot para completar la integración, y debido a un mayor entendimiento del sistema emocional, se tuvo que acotar el trabajo de título para dejar la recopilación del estado emocional como una tarea propuesta.

Finalmente, se considera este trabajo como una oportunidad de promover la inclusión de desafíos basados en sistemas LTM en la liga RoboCup@Home. Para ello, a partir de las investigaciones y diseño realizado, se ha elaborado un estudio [8] motivando el uso de LTM y una metodología para su inclusión en la competencia. Así, se espera que el desarrollo de LTM y capacidades asociadas deje de ser postergado y pase a ser una de las prioridades para los equipos participantes.

Trabajo Futuro

El trabajo realizado provee un servidor LTM genérico para robots domésticos. El sistema cumple con los requisitos básicos que una memoria episódica debiera cumplir, sin embargo, hay muchos aspectos pendientes o no cubiertos por este trabajo.

En primer lugar, el trabajo futuro más importante es la implementación de los aspectos dejados como pendientes. Es decir, la migración de la base de datos, el algoritmo para la actualización automática de relevancias y el sistema emocional de Bender.

Ya que la selección de una consulta episódica puede tener un alto impacto en el desempeño del sistema, se considera prioritario realizar un estudio sobre el costo de cada operación lógica que provee MongoDB y sus interacciones. Esto, con el fin de optimizar las consultas y asegurar una medida de desempeño del sistema LTM. Por otro lado, los experimentos de desempeño pueden ser complementados con casos de uso de streams y entidades, y mediciones sobre la tasa de lectura y escritura de memoria secundaria.

La inferencia de información a partir de los recuerdos también es un aspecto deseable. Esto fue considerado en primera instancia para el desarrollo del trabajo, pero tras algunas pruebas preliminares no se pudo satisfacer este objetivo secundario, debiendo acotar el alcance del trabajo. La propuesta consideraba el uso del framework KnowRob [34, 35, 43], que implementa memoria semántica y permite inferir información a partir de los datos almacenados.

Bibliografía

[1] IFR, International Federation of Robotics. [Online]. Available: https://ifr.org/

[2] M. Matamoros, C. Rascon, J. Hart, D. Holz, and L. van Beek, "RoboCup@Home 2018: Rules and Regulations," http://www.robocupathome.org/rules/2018_rulebook.pdf, 2018.

[3] UChile Robotics. [Online]. Available: http://robotica-uchile.amtc.cl/

[4] SoftBank Robotics. [Online]. Available: http://www.softbank.jp/en/robot/

[5] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Mg, "ROS: an open-source Robot Operating System," in *Icra*, vol. 3, 2009, p. 5.

[6] H. Eichenbaum, *Learning & Memory*. New York: W. W. Norton & Company, 2008.

[7] S. Vijayakumar, "Long-Term Memory in Cognitive Robots," Ph.D. dissertation, Universitaet des Saarlandes, 2014.

[8] M. Pavez, J. Ruiz del Solar, V. Amo, and F. Meyer, "Towards Long-Term Memory for Social Robots: Proposing a New Challenge for the RoboCup@Home League," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018.

[9] M. L. Sánchez, M. Correa, L. Martínez, and J. Ruiz-Del-Solar, "An episodic long-term memory for robots: The bender case," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9513, pp. 264–275, 2015.

[10] D. Stachowicz and G. J. M. Kruijff, "Episodic-like memory for cognitive robots," *IEEE Transactions on Autonomous Mental Development*, 2012.

[11] L. R. Squire, "Memory systems of the brain: a brief history and current perspective,"

Neurobiol Learn Mem, vol. 82, no. 3, pp. 171–177, Nov 2004.

[12] R. G. Morris and D. O. Hebb, "D.O. Hebb: The Organization of Behavior, Wiley: New York; 1949," *Brain Res. Bull.*, vol. 50, no. 5-6, p. 437, 1999.

[13] E. Tulving, "Episodic and semantic memory," *Organization of Memory (Tulving E, Donaldson W, eds).*, pp. 381–403. Academic Press, New York., 1972.

[14] T. Deutsch, A. Gruber, R. Lang, and R. Velik, "Episodic memory for autonomous agents," in *2008 Conference on Human System Interactions*, May 2008, pp. 621–626.

[15] N. S. Clayton and J. Russell, "Looking for episodic memory in animals and young children: Prospects for a new minimalism," *Neuropsychologia*, vol. 47, no. 11, pp. 2330 – 2340, 2009, episodic Memory and the Brain. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0028393208004132

[16] P. A. Allen, K. P. Kaut, and R. R. Lord, "Chapter 1.8 emotion and episodic memory," in *Handbook of Episodic Memory*, ser. Handbook of Behavioral Neuroscience, E. Dere, A. Easton, L. Nadel, and J. P. Huston, Eds. Elsevier, 2008, vol. 18, pp. 115 – 132.

[17] C. H. Bailey, D. Bartsch, and E. R. Kandel, "Toward a molecular definition of long-term memory storage," *Proceedings of the National Academy of Sciences*, vol. 93, no. 24, pp. 13 445–13 452, 1996. [Online]. Available: http://www.pnas.org/content/93/24/13445

[18] W. C. Ho, K. Dautenhahn, M. Y. Lim, P. A. Vargas, R. Aylett, and S. Enz, "An initial memory model for virtual and robot companions supporting migration and long-term interaction," in *RO-MAN 2009 - The 18th IEEE International Symposium on Robot and Human Interactive Communication*, Sept 2009, pp. 277–284.

[19] R. Salgado, F. Bellas, P. Caamano, B. Santos-Diez, and R. J. Duro, "A procedural Long Term Memory for cognitive robotics," *2012 IEEE Conference on Evolving and Adaptive Intelligent Systems*, pp. 57–62, 2012.

[20] A. Nuxoll and J. E. Laird, "A cognitive model of episodic memory integrated with a general cognitive architecture," in *ICCM*, 2004.

[21] J. E. Laird, A. Newell, and P. S. Rosenbloom, "Soar: An architecture for general intelligence," *Artificial Intelligence*, vol. 33, no. 1, pp. 1 – 64, 1987. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0004370287900506

[22] A. Nuxoll and J. Laird, "Extending cognitive architecture with episodic memory," in *AAAI-07/IAAI-07 Proceedings: 22nd AAAI Conference on Artificial Intelligence and the 19th Innovative Applications of Artificial Intelligence Conference*, vol. 2, 2007, pp. 1560–1565.

[23] P. Ratanaswasd, S. Gordon, and W. Dodd, "Cognitive control for robot task execution," in *ROMAN 2005. IEEE International Workshop on Robot and Human Interactive Communication, 2005.*, Aug 2005, pp. 440–445.

[24] W. Dodd and R. Gutierrez, "The role of episodic memory and emotion in a cognitive robot," *Proceedings - IEEE International Workshop on Robot and Human Interactive Communication*, vol. 2005, pp. 692–697, 2005.

[25] S. Jockel, D. Westhoff, and J. Zhang, "Epirome - a novel framework to investigate high-level episodic robot memory," in *2007 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Dec 2007, pp. 1075–1080.

[26] S. Jockel and M. Weser, "Towards an episodic memory for cognitive robots," *European Conference on Artificial Intelligence*, pp. 68–74, 2008.

[27] D. Hintzman, "Judgments of frequency and recognition memory in a multiple-trace model," vol. 95, pp. 528–551, 10 1988.

[28] S. F. W. H. Feinstone and F. G. Patterson, "The lida architecture : Adding new modes of learning to an intelligent , autonomous , software agent," in *Integrated Design and Process Technology (IDPT-2006), San Diego, CA*, 2006.

[29] N. Kuppuswamy, S.-H. Cho, and J.-H. Kim, "A cognitive control architecture for an artificial creature using episodic memory," pp. 3104 – 3110, 11 2006.

[30] S. Vere and T. Bickmore, "A basic agent," *Comput. Intell.*, vol. 6, no. 1, pp. 41–60, Jan. 1990. [Online]. Available: https://doi.org/10.1111/j.1467-8640.1990.tb00128.x

[31] C. Brom, K. Pešková, and J. Lukavský, "What does your actor remember? towards characters with a full episodic memory," in *Virtual Storytelling. Using Virtual Reality Technologies for Storytelling*, M. Cavazza and S. Donikian, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 89–101.

[32] T. P. Spexard, F. H. K. Siepmann, and G. Sagerer, "Memory-based Software Integration for Development in Autonomous Robotics," *Intelligent Autonomous Systems 10*, pp. 49–53, 2008.

[33] G. Veruggio and F. Operto, "Roboethics: A Bottom-Up Interdisciplinary Discourse in the Field of Applied Ethics in Robotics," *International Review of Information Ethics*, vol. 6, pp. 2–8, 2006.

[34] J. Winkler, M. Tenorth, A. K. Bozcuoglu, and M. Beetz, "CRAMm - Memories for Robots Performing Everyday Manipulation Activities," *Advances in Cognitive Systems*, vol. 3, pp. 47–66, 2014.

[35] M. Tenorth and M. Beetz, "KnowRob: A knowledge processing infrastructure for cognition-enabled robots," *The International Journal of Robotics Research*, vol. 32, no. 5, pp. 566–590, apr 2013.

[36] M.-J. Kim, S. H. Baek, S. H. Cho, and J. H. Kim, "Approach to integrate episodic memory into cogency-based behavior planner for robots," in *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, oct 2016, pp. 4188–4193.

[37] P. O. Haikonen, *The cognitive approach to conscious machines.* Charlottesville, VA: Imprint Academic, 2003.

[38] Z. Kasap and N. Magnenat-Thalmann, "Towards episodic memory-based long-term affective interaction with a human-like robot," in *Proceedings - IEEE International Workshop on Robot and Human Interactive Communication*, 2010.

[39] R. Plutchik and H. Kellerman, "Emotion: Theory, research, and experience," in *Theories of Emotion*, R. Plutchik and H. Kellerman, Eds. Academic Press, 1980, p. ii. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780125587013500016

[40] Página web oficial de YAML. [Online]. Available: http://yaml.org/

[41] Página web oficial de JSON. [Online]. Available: https://www.json.org/

[42] Página web oficial de MongoDB. [Online]. Available: https://www.mongodb.com/

[43] M. Tenorth and M. Beetz, "KNOWROB - knowledge processing for autonomous personal robots," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, oct 2009, pp. 4261–4266.

A

Anexo: Diseño

En este anexo se presentan los requisitos y validaciones del sistema LTM, utilizados para el diseño de éste, como se describe en el Capítulo 4. Las validaciones son revisadas nuevamente en el Capítulo 6, para estudiar el cumplimiento de los objetivos del trabajo.

A.1. Requisitos de sistema

A continuación se presenta un listado con los requisitos que debe cumplir el sistema LTM. Cada requisito indica el requisito de proyecto asociado y es identificado por una abreviación de la forma RSXX.

En la Tabla A.1 se presenta la matriz de trazabilidad de requisitos de proyecto y sistema, indicando la relación entre cada uno.

Requisito RS01 *(Derivado de $\mathcal{RP}_5$)*
El sistema debe ser agnóstico respecto a la plataforma objetivo. No se puede asumir que estructuras de datos semánticas serán almacenadas para cada episodio. Debe permitir que usuarios definan las estructuras de datos semánticas a almacenar.

Requisito RS02 *(Derivado de $\mathcal{RP}_1$)*
▷*Véase requisito* ($\mathcal{R}_1$).
Episodios deben contener campo *What*, para almacenar memoria semántica. Ya que la memoria semántica es definida por el usuario, éste campo debe poder almacenar cualquier tipo de información.

Requisito RS03 *(Derivado de $\mathcal{RP}_1$)*
▷*Véase requisito* ($\mathcal{R}_1$).
Episodios deben contener campo *When*, para almacenar tiempos de inicio y finalización del episodio. Debe poder manejar intervalos de tiempo desde segundos a años.

Requisito RS04 *(Derivado de $\mathcal{RP}_1$)*

▷ *Véase requisito* ($\mathcal{R}_1$).
Episodios deben contener campo *Where*, para almacenar información sobre la ubicación del robot durante su ocurrencia.

Requisito RS05 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_2$).
Episodios deben ser indexados al menos por los campos *What*, *When* y *Where*.

Requisito RS06 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_2$).
El servidor debe permitir realizar búsquedas de episodios con condiciones sobre la información almacenada para *What*, *When*, *Where*. Las condiciones de búsqueda deben permitir comparaciones igualdad, mayor que, pertenencia entre tipos de dato básicos strings, números, bool, asociados a los campos del episodio.

Requisito RS07 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_3$).
Los episodios pueden crear entidades semánticas o actualizar las ya existentes.

Requisito RS08 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_4$).
Cada episodio ingresado a la memoria es único, a pesar de que existan similitudes con otros episodios.

Requisito RS09 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_5$).
A partir de un episodio se puede acceder a episodios padre e hijos.

Requisito RS10 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_6$).
Las lecturas de episodios deben mostrar la memoria semántica como se conocía en ese instante. Es decir, el resultado de una consulta debe mostrar las entidades según la perspectiva del momento.

Requisito RS11 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_7$).
Episodios pueden estar compuestos de sub-episodios.

Requisito RS12 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_8$).
Pueden existir episodios simultáneos, y con distintos tiempos de inicio y fin.

Requisito RS13 *(Derivado de* $\mathcal{RP}_1$*)*
▷ *Véase requisito* ($\mathcal{R}_9$).
Se deben minimizar las dependencias de módulos de software externos para el funcionamiento del servidor. Particularmente:

- El sistema debe ser implementado en ROS y solamente utilizando las librerías estándar de Python y C++. La única excepción corresponde a la librería con el driver de la base

de datos a utilizar.

- No deben existir dependencias extra para los módulos de memoria semántica definidos por el usuario, ni para la representación de la memoria semántica.

Requisito RS14 *(Derivado de $\mathcal{RP}_1$)*
▷*Véase requisito* ($\mathcal{R}_9$).
En caso que ya exista información semántica almacenada, los usuarios deben poder modificar su representación.

Requisito RS15 *(Derivado de $\mathcal{RP}_1$)*
▷*Véase requisito* ($\mathcal{R}_{10}$).
Eficiencia: El servidor debe tolerar una alta tasa de generación de eventos, sin degradar el funcionamiento del robot. Específicamente, se debe realizar una estimación de la cantidad de consultas por minuto que se espera para cada operación de lectura y escritura para el robot objetivo. A partir de ésto, se deben cumplir los siguientes requerimientos:

- El funcionamiento base del sistema, es decir, con el servidor en ejecución y sin consultas funcionando, no debe exceder el uso del 10 % del CPU y 10 % de RAM.

- El servidor debe poder trabajar bajo la tasa de consultas esperadas, sin exceder un aumento en el uso de recursos de un 15 %, relativo al costo de su funcionamiento base.

Requisito RS16 *(Derivado de $\mathcal{RP}_1$)*
▷*Véase requisito* ($\mathcal{R}_{11}$).
Escalabilidad: Los costos de operaciones CRUD deben escalar bien, respecto a la cantidad de datos almacenados. Particularmente, el costo en tiempo de las operaciones de escritura y lectura de episodios debe estar acotado en $O(n)$ respecto a la cantidad de episodios almacenados. Para esto, se debe realizar una estimación de la cantidad de episodios que debe manejar el robot objetivo.

Requisito RS17 *(Derivado de $\mathcal{RP}_1$)*
El servidor debe proveer servicios para las operaciones CRUD sobre los episodios y unidades semánticas manejadas.

Requisito RS18 *(Derivado de $\mathcal{RP}_1$)*
Los episodios almacenados deben estar conectados bidireccionalmente con los datos semánticos relacionados.

Requisito RS19 *(Derivado de $\mathcal{RP}_2$)*
Episodios deben tener, al menos, la siguiente información sobre la emoción del robot asociada al episodio: nombre de la emoción principal, e indicador numérico que identifique la intensidad de la emoción. La estructura de datos y formato utilizado no debe depender de algún software emocional externo.

Requisito RS20 *(Derivado de $\mathcal{RP}_3$)*
Episodios deben tener un indicador numérico de relevancia histórica, el que debe ser degradado automáticamente con el paso del tiempo.

Requisito RS21 *(Derivado de $\mathcal{RP}_4$)*
Episodios deben tener un indicador numérico de relevancia episódica generalizado, que una los subsistemas de relevancia en sólo uno. El indicador debe ser actualizado automáticamente cuando cualquiera de los subindicadores sea actualizado.

Requisito RS22 *(Derivado de $\mathcal{RP}_4$)*
Episodios deben estar indexados por sus indicadores numéricos de relevancia, sumado a la descripción de la emoción. Se deben poder realizar búsquedas utilizando éstos índices.

Requisito RS23 *(Derivado de $\mathcal{RP}_6$)*
El servidor debe ser compatible con ROS, mediante una API ROS para acceder a todas sus funcionalidades.

Requisito RS24 *(Derivado de $\mathcal{RP}_6$)*
El robot Bender no dispone de módulos para el cómputo de emociones. Se deben implementar al menos 3 sistemas de generación de emociones para el robot, los que deben ser utilizados para la demostración.

Requisito RS25 *(Derivado de $\mathcal{RP}_6$)*
Se deben implementar instancias de cada componente genérico, enfocadas en el robot Bender.

- Implementación de la interfaz para adquisición de episodios basados en librería SMACH.
- Implementación de memoria semántica adecuada al robot: Información sobre personas y objetos.
- El software dedicado al robot debe ser implementado en paquetes de ROS distintos del utilizado para el servidor LTM.

Requisito RS26 *(Derivado de $\mathcal{RP}_6$)*
El servidor debe ser implantado en el robot Bender. Debe ser agregado al proceso de instalación y su documentación. Debe ser configurado para ser ejecutado simultáneamente a los otros módulos.

RSX	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
$\mathcal{RP}_1$		X	X	X	X	X	X	X	X	X	X	X	X	X	X
$\mathcal{RP}_2$															
$\mathcal{RP}_3$															
$\mathcal{RP}_4$															
$\mathcal{RP}_5$	X														
$\mathcal{RP}_6$															

RSX	16	17	18	19	20	21	22	23	24	25	26	/	/	/	/
$\mathcal{RP}_1$	X	X	X												
$\mathcal{RP}_2$				X											
$\mathcal{RP}_3$					X										
$\mathcal{RP}_4$						X	X								
$\mathcal{RP}_5$															
$\mathcal{RP}_6$								X	X	X	X				

Tabla A.1: Matriz de trazabilidad de requisitos de proyecto y sistema.

A.2. Listado de validaciones

A continuación se presenta un listado con las validaciones a realizar para estudiar el cumplimiento de los requisitos del trabajo. Las validaciones son separadas en 3 categorías: funcionalidad, integración y desempeño. Cada validación indica el requisito de sistema asociado y es identificada por una abreviación de la forma VTXX. Finalmente, en la Tabla A.2 se presenta la matriz de trazabilidad entre las validaciones definidas y los requisitos de sistema asociados.

A.2.1. Validaciones de funcionalidad: VAXX

Validación VA01 *(Derivado de* RS01)
Verificar que el que el trabajo no tiene dependencias en software de Bender.

Validación VA02 *(Derivado de* RS01)
Verificar que el servidor admite la definición de estructuras de datos genéricas para la información semántica.

Validación VA03 *(Derivado de* RS01)
Verificar que módulos de software implementados sean separados en al menos 2 grupos de paquetes ROS. El primero debe contener el servidor LTM sin ninguna dependencia al robot. El resto debe contener los módulos extra implementados, que deben depender del primero.

Validación VA04 *(Derivado de* RS02)
Verificar que el campo *What* de un episodio pueda almacenar cualquier estructura de datos que ROS sea capaz de manejar.

Validación VA05 *(Derivado de* RS03)
Verificar que episodios poseen campo *When* para almacenar tiempo de inicio y finalización del episodio.

Validación VA06 *(Derivado de* RS03)
Verificar que el servidor permita almacenar episodios con duraciones de segundos, horas, días y años.

Validación VA07 *(Derivado de* RS04)
Verificar que episodios contengan campo *Where*. Debe permitir almacenar la ubicación del robot en términos de coordenadas, relacionadas a un mapa y sistema coordenado.

Validación VA08 *(Derivado de* RS05)
Recolectar episodios desde el servidor, utilizando cada campo perteneciente a *What*, *When* y *Where*.

Validación VA09 *(Derivado de* RS06)
Recolectar episodios desde el servidor, utilizando condiciones de {igualdad, mayor/menor que, pertenencia a un arreglo}, para datos de tipo {entero, double, string, bool}. Consultas deben

ser realizadas sobre campos pertenecientes a *What*, *When* y *Where*. Repetir, considerando 2 o más condiciones en cada consulta.

Validación VA10 *(Derivado de* RS07)
Para una entidad semántica de tipo A, se hacen las siguientes validaciones:

- Datos aprendidos posteriormente. Realizar en orden:
  1. Ingresar episodio con nueva instancia (a) de A.
  2. Ingresar nuevo episodio con datos nuevos sobre (a).
  3. Consultar (a), debe incluir el dato nuevo.
- Datos modificados posteriormente. Realizar en orden:
  1. Ingresar episodio con nueva instancia (a) de A, con campo F.
  2. Ingresar nuevo episodio donde se modifica el valor de F en (a).
  3. Consultar (a), debe mostrar el dato modificado.

Validación VA11 *(Derivado de* RS08)
Verificar que cada nuevo episodio disponga de un ID único entre el resto de los episodios ya almacenados.

Validación VA12 *(Derivado de* RS09)
Obtener un episodio desde el servidor. A partir de sus datos consultar por su episodio padre y episodios hijos.

Validación VA13 *(Derivado de* RS10)
Para una entidad semántica de tipo A, se realiza en orden:

1. Se ingresa un episodio E1 con instancia (a) de A y campo F1.
2. Se ingresa nuevo episodio E2 que modifica el valor F1 de (a) por F2.
3. Consultar por episodio E1. Debe mostrar a (a) con valor F1.
4. Consultar por episodio E2. Debe mostrar a (a) con valor F2.

Validación VA14 *(Derivado de* RS11)
Recolectar episodios anidados a partir de máquina de estado anidada en SMACH. Verificar lo siguiente:

- Recolectar episodio raíz. Verificar que el episodio tiene campo que identifica a sus hijos. Verificar que es posible consultar por ellos al servidor.
- Recolectar episodio hoja. Verificar que el episodio tiene campo que identifica a su episodio padre. Verificar que es posible consultar por el episodio padre.

Validación VA15 *(Derivado de* RS12)
Generar dos eventos transpuestos en tiempo (campo `When`). Verificar que es posible almacenarlos y consultar por ellos. Verificar que sus datos sean correctos y tengan referencias a los datos semánticos modificados en su rango temporal. Verificar que datos semánticos asociados no sean duplicados.

Validación VA16 *(Derivado de* RS13)
Verificar que el paquete de ROS que implementa el servidor sólo tenga como dependencias: librerías estándar de C++ y Python, driver de la base de datos, paquetes estándar de ROS.

Validación VA17 *(Derivado de* RS14)
Crear entidad semántica de tipo A. Crear e ingresar M episodios que ingresan datos sobre instancias de A. Mutar estructura de A a estructura B. Ingresar N episodios que ingresen datos de estructura B. Verificar que es posible consultar sobre entidades semánticas definidas en episodios de M y N.

Validación VA18 *(Derivado de* RS17)
Verificar que el sistema provee servicios ROS para operaciones CRUD sobre episodios y unidades semánticas definidas por los usuarios. Verificar que servicios CRUD funcionan correctamente.

Validación VA19 *(Derivado de* RS18)
Obtener un episodio desde el servidor. A partir de él, acceder a datos semánticos definidos. Verificar que se pueda obtener una referencia al episodio, a partir de los datos semánticos obtenidos.

Validación VA20 *(Derivado de* RS18)
Obtener una entidad desde el servidor. A partir de ella, listar todos los episodios relacionados.

Validación VA21 *(Derivado de* RS19)
Consultar algún episodio de la base de datos. Verificar que dispone de campos de relevancia emocional: Indicador numérico para la intensidad de la emoción, y descripción de la emoción.

Validación VA22 *(Derivado de* RS19)
Verificar que estructura de datos emocional sólo está compuesta por tipos de datos disponibles entre los del estándar en ROS. Verificar que el sistema LTM sólo dependa de la intensidad de la emoción y nombre asociado para la implementación de sus funcionalidades.

Validación VA23 *(Derivado de* RS20)
Generar episodios con antigüedades en el rango de 1 segundo hasta 5 años, espaciados por intervalos de horas (para los 24 primeros) y días, para los (365) siguientes y semanas para el resto. Forzar actualización de relevancias, leer todos los episodios y graficar su relevancia histórica, para verificar que es decreciente.

Validación VA24 *(Derivado de* RS21)
Consultar algún episodio de la base de datos. Verificar que dispone de un indicador numérico con la relevancia generalizada.

- Verificar que al disminuir la relevancia histórica del episodio, la relevancia generalizada

también disminuya.

- Manteniendo las otras relevancias fijas. Verificar que episodios con mayor/menor relevancia emocional muestran una mayor/menor relevancia generalizada.

- Manteniendo las otras relevancias fijas. Verificar que episodios con mayor/menor relevancia histórica muestran una mayor/menor relevancia generalizada.

Validación VA25 *(Derivado de* RS22)
Generar episodios con distintos índices de relevancia histórica, emocional y generalizada. Verificar que se puedan realizar búsquedas de episodios mediante cada uno de los indicadores de relevancia, sumado a la descripción de la emoción.

Validación VA26 *(Derivado de* RS23)
Verificar que el servidor provee API ROS para configurar sus parámetros.

Validación VA27 *(Derivado de* RS23)
Verificar que el servidor activo provee API ROS para agregar, buscar, actualizar y borrar episodios.

Validación VA28 *(Derivado de* RS25)
Verificar que servidor almacena episodios generados a partir de la interfaz con SMACH. Se debe verificar que un set de máquinas de estado ejecutadas sea procesado correctamente.

A.2.2. Validaciones de integración: VBXX

Validación VB01 *(Derivado de* RS24)
Verificar que existan implementados al menos 3 módulos generadores de emociones para el robot Bender. Éstos deben utilizar sensores disponibles en el robot. Los módulos deben pertenecer a un paquete ROS dedicado.

Validación VB02 *(Derivado de* RS24)
Generar episodios en donde se ocupe cada uno de los módulos generadores de emociones implementados para el robot Bender.

Validación VB03 *(Derivado de* RS25)
Verificar que módulos específicos para Bender provean implementaciones de entidades semánticas para personas y objetos.

Validación VB04 *(Derivado de* RS26)
Verificar que el sistema LTM se instala correctamente en conjunto con la instalación del robot.

Validación VB05 *(Derivado de* RS26)
Verificar que el trabajo provee archivos *launch* para ejecutar el software integrado en Bender.

Validación VB06 *(Derivado de* RS26)

Verificar que tras iniciar el software base del robot, se encuentre activo el servidor LTM y sus API ROS.

A.2.3. Validaciones de desempeño: VCXX

Se debe determinar un conjunto de operaciones de lectura y escritura de interés para el uso del servidor LTM en el robot objetivo. Luego, se deben realizar las siguientes mediciones para determinar el uso de recursos del sistema:

1. Uso de CPU y RAM del servidor LTM y la base de datos cuando el sistema está activo, pero no responde consultas (estado base).

2. Uso de CPU y RAM para cada operación del conjunto elegido, sin considerar información semántica.

3. Uso de CPU y RAM para cada operación del conjunto elegido, considerando información semántica para la inserción y búsqueda de episodios.

4. Repetir pruebas para distintas cantidades de episodios almacenadas en el sistema.

VC01 (RS15) Se debe validar que en estado base, el uso de CPU y RAM se comporte de acuerdo a los límites impuestos por el requisito RS15.

VC02 (RS15) Se debe verificar que las operaciones definidas no sobrepasen el límite de uso de recursos impuesto por el requisito RS15.

Cada una de las siguientes pruebas busca validar que la complejidad temporal de las operaciones se comporte de acuerdo al requisito RS16, al aumentar la cantidad de episodios almacenados en la base de datos.

Se debe definir un conjunto de operaciones de lectura y escritura de interés para el uso del servidor LTM en el robot objetivo. Luego, se debe medir el tiempo promedio que toma cada operación, para una cantidad creciente de episodios. La cantidad de episodios a evaluar se debe obtener a partir de las estimaciones de uso del sistema LTM en el robot objetivo.

VC03 (RS16) Se debe validar que los tiempos de cada operación de interés escalen de acuerdo al comportamiento definido en el requisito RS16.

RSX	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
VA01	X														
VA02	X														
VA03	X														
VA04		X													
VA05			X												
VA06			X												
VA07				X											
VA08					X										
VA09						X									
VA10							X								
VA11								X							
VA12									X						
VA13										X					
VA14											X				
VA15												X			
VA16													X		
VA17														X	
VC01															X
VC02															X

RSX	16	17	18	19	20	21	22	23	24	25	26	/	/	/	/
VC03	X														
VA18		X													
VA19			X												
VA20			X												
VA21				X											
VA22				X											
VA23					X										
VA24						X									
VA25							X								
VA26								X							
VA27								X							
VA28										X					
VB01									X						
VB02									X						
VB03										X					
VB04											X				
VB05											X				
VB06											X				

Tabla A.2: Matriz de trazabilidad entre validaciones y los requisitos de sistema asociados. Se omiten filas vacías.

B

Anexo: Implementación

En este capítulo se presentan bloques de código utilizados para la implementación del trabajo, los que complementan las explicaciones dadas en el Capítulo 5. En primer lugar, se presentan todos los mensajes ROS que conforman la definición de un episodio. Luego, se presenta código utilizado para la API ROS del servidor LTM.

B.1. Modelo de Datos

A continuación se muestran todos los mensajes que definen la estructura de un episodio, los que son embebidos en el mensaje `ltm/Episode.msg`.

Código B.1: ltm/When.msg

```
# timestamps in UTC (as handled by ROS)
time start
time end
```

Código B.2: ltm/Where.msg

```
# numerical information
string frame_id # related to pose
string map_name # where the pose was recorded
geometry_msgs/Point position # 2D position in map

# semantic location information
string location # e.g., Living Room
string area # e.g., Arena 1

# children information
geometry_msgs/Point[] children_hull # Convex Hull for parents
                                    # only valid if all children
                                    # share frame_id and map_name
string[] children_locations
string[] children_areas
```

Código B.3: ltm/What.msg

```
ltm/StreamRegister[] streams
ltm/EntityRegister[] entities
```

Código B.4: ltm/StreamRegister.msg

```
string type # stream type
uint32 uid # stream uid in database
```

Código B.5: ltm/EntityRegister.msg

```
string type
uint32 uid
uint32[] log_uids
```

Código B.6: ltm/Info.msg

```
int32 n_usages # e.g. 10
string source # e.g. "text-json"
time creation_date # (in UTC)
time last_access # (in UTC)

# LTM software
string ltm_version # e.g. "0.0.0"
string ros_version # e.g. "kinetic"
string os_version # uname -a information
```

Código B.7: ltm/Relevance.msg

```
ltm/EmotionalRelevance emotional
ltm/HistoricalRelevance historical
```

Código B.8: ltm/EmotionalRelevance.msg

```
# Information provided by the emotion engine.
string software # e.g. "fake"
string software_version # e.g. "0.0.0"

# Registered Emotions
# Do not impose names, quantity or bounds for values. However, it is
# adviced to align the emotion names to the ones provided below
# and to keep values in the range [0, 1].
string[] registered_emotions # e.g. ["happy","surprised","angry","sad", ...]
float32[] registered_values # e.g. [0.6, 0.2, 0.1, 0.1, ...]

# Registered emotion and value for the episode.
# - emotion is specified by the integer associated to one emotion below.
# - value is a floating number between 0 (minimum) and 1 (max emotion strength).
int32 emotion # e.g. 3 (SURPRISE)
float32 value # e.g. 0.5
int32[] children_emotions
float32[] children_values


# Available emotions (Plutchik's Wheel of emotions).
uint8 JOY=0
```

```
uint8 TRUST=1
uint8 FEAR=2
uint8 SURPRISE=3
uint8 SADNESS=4
uint8 DISGUST=5
uint8 ANGER=6
uint8 ANTICIPATION=7
```

Código B.9: ltm/HistoricalRelevance.msg

```
# Historical relevance strength.
# Values between 0 (no relevance) and 1 (max relevance).
float32 value

# date of the last applied update for "value" field.
ltm/Date last_update

# next scheduled update for "value" field.
ltm/Date next_update
```

Código B.10: ltm/Date.msg

```
uint16 day # e.g. 17 - range [1-31]
uint16 month # e.g. 12 - range [1-12]
uint16 year # e.g. 1991
```

B.2. Interfaz ROS

A continuación se presenta un ejemplo de configuración del sistema LTM, para su uso con el robot simulado. Luego se presenta el mensaje `ltm/QueryResult.msg`, utilizado por la API ROS de consulta para entregar los resultados de las búsquedas episódicas.

Código B.11: server.yaml

```
# ============================================
# Sample configuration file for the LTM server
# ============================================

# Server parameters
db: "ltm_db"
host: "localhost"
port: 27017
timeout: 60.0
collection: "episodes"

# LTM plugins and parameters.
# Each plugin must define the pluginlib class and its parameters
plugins:

  # Robot emotion plugin
  emotion:
    class: "ltm_samples::EmotionPlugin"

  # Robot location plugin
  location:
    class: "ltm_samples::LocationPlugin"
```

```yaml

  # Stream plugins
  streams:
    # List of streams to consider.
    include: ["image"]

    # ImageStream
    image:
      class: "ltm_addons::ImageStreamPlugin"
      type: "images"
      collection: "images"
      topic: "/robot/fake/sensors/camera/image_raw"
      buffer_frequency: 3.0
      buffer_size: 100

  # Entity plugins
  entities:
    # List of entities to consider.
    include: ["people", "objects"]

    # PeopleEntityPlugin
    people:
      class: "ltm_samples::PeopleEntityPlugin"
      type: "people"
      collection: "people"
      topic: "/robot/fake_short_term_memory/person/updates"

    # ObjectsEntityPlugin
    objects:
      class: "ltm_samples::ObjectsEntityPlugin"
      type: "objects"
      collection: "objects"
      topic: "/robot/fake_short_term_memory/object/updates"
```

Código B.12: ltm/QueryResult.msg

```
string type
uint32[] uids
```