Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Diseño e implementación de un protocolo de comunicación para entornos multi-robot

Autor: Diego Vigneron Olmos
Tutor: Javier de Lope Asiaín

Madrid, Enero 2025

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería Informática

Título: Diseño e implementación de un protocolo de comunicación para entornos multi-robot

Enero 2025

Autor: Diego Vigneron Olmos
Tutor: Javier de Lope Asiaín
Departamento de Inteligencia Artificial
Escuela Técnica Superior de Ingenieros Informáticos
Universidad Politécnica de Madrid

Resumen

En este Trabajo de Fin de Grado se diseña e implementa un protocolo de comunicación para la comunicación en entornos multi-robot con el objetivo de mejorar la eficiencia y la escalabilidad. El diseño del protocolo se aborda intentando mantener la transmisión de datos simple para obtener el mayor rendimiento con la menor latencia posible.

El protocolo ha sido desarrollado en el lenguaje C, utilizando el framework de ZeroMQ para la gestión de transmisión de mensajes por sockets. Se ha establecido un sistema de paquetes estructurado con mecanismos de serialización y deserialización que permite la transmisión de comandos entre los usuarios del protocolo.

Además, se ha desarrollado un servidor que se conecta con el simulador CoppeliaSim y actúa como traductor entre este entorno de simulación y clientes que se conectan con el servidor con el objetivo de modificar este entorno y tener un control sobre los robots representados en este entorno.

Posteriormente, se han implementado varios clientes para comprobar el funcionamiento esperado tanto del protocolo de comunicación como del servidor encargado de la comunicación con el simulador.

Abstract

This Final Degree Project explains the design and implementation of a communication protocol for communication in multi-robot systems with the objective of improving efficiency and scalability. The design focuses on maintaining simple data transmissions with the objective of obtaining the highest performance with the lowest possible latency.

The protocol has been developed in C, using the ZeroMQ framework for message transmission management via sockets. A structured packet system with serialization and deserialization mechanisms has been established, allowing the transmission of commands between the users of this protocol.

Additionally, a server has been developed to connect with the CoppeliaSim simulator, acting as a translator between this simulation environment and clients that connect to the server to modify the environment and control the robots represented within it.

Subsequently, several clients have been implemented to verify the expected operation of both the communication protocol and the server responsible for communication with the simulator.

Tabla de contenidos

Capítulo 1

Introducción

1.1. Motivación

La robótica ha experimentado un crecimiento significativo en las últimas décadas, impulsado por la madurez de las tecnologías de comunicación y la disponibilidad de potentes entornos de simulación [1]. Este auge ha propiciado la aparición de **sistemas multi-robot**, donde múltiples agentes cooperan para cumplir tareas más complejas de lo que podría abordar un robot en solitario. Ejemplos de estas aplicaciones abarcan desde logística industrial y exploración de entornos desconocidos, hasta asistencia en entornos hospitalarios o colaboración en procesos de manufactura [1].

En este contexto, la manera de comunicarse entre sí adquiere un papel crucial, ya que dependiendo del caso, un robot puede necesitar *recibir* información del estado del mundo generalmente suministrada por sus sensores o por datos aquiridos por otros agentes y además *enviar* estos datos ya sea a un servidor que se haga cargo de repartirlos o hacerla llegar a otros robots.

En escenarios con numerosos agentes, la eficiencia y la robustez del *protocolo de comunicaciones* encargado de realizar este intercambio de datos determina en gran medida el rendimiento global del sistema y la viabilidad de la coordinación, dejando en duda las tareas que puedan realizar [2].

Además, la elección de la **arquitectura de control** afecta de manera determinante a la complejidad de la infraestructura de comunicación [1]. Entre los modelos clásicos se distinguen:

- **Arquitecturas centralizadas:** Un único nodo o controlador maestro reúne la información de todos los robots y emite órdenes para toda la flota [1].
- **Arquitecturas jerárquicas:** En esta aproximación, cada robot (o grupo de robots) se encarga de supervisar un subconjunto específico del equipo, de forma análoga a una estructura de mando militar [1].

- **Arquitecturas descentralizadas:** Son las más comunes en la literatura de robótica multi-robot. Cada robot toma sus decisiones basándose únicamente en la información local (estado propio y datos recibidos de sus vecinos), sin un nodo maestro [1].
- **Arquitecturas híbridas:** Pretenden combinar la robustez de un control local con la capacidad de imponer objetivos generales a la flota [1].

Con una dependencia cada vez mayor en el uso de robots de diversos tipos desde robots industriales y logísticos hasta robots de servicio y autónomos en entornos domésticos y la necesidad de intercambiar datos a alta frecuencia (por ejemplo, información de proximidad o de posicionamiento), se requiere un *protocolo de comunicación* capaz de ofrecer ligereza y escalabilidad.[3]. De lo contrario, la infraestructura podría verse limitada por la sobrecarga de mensajes en un sistema centralizado, o por la dificultad de mantener coherencia global en uno descentralizado.

Bajo estas premisas, el presente trabajo se centra en la creación de un **protocolo de comunicaciones para sistemas multi-robot** capaz de abarcar los posibles problemas mencionados.

1.2. Objetivos y alcance

El objetivo principal de este proyecto es diseñar e implementar un protocolo de comunicaciones para un entorno multi-robot, integrando la comunicación entre un distintos procesos encargados de la conexión con un simulador o del control de robots. Los objetivos específicos son:

1. **Revisión bibliográfica de sistemas multi-robot:** Investigar el estado del arte en arquitecturas de control, metodologías de comunicación y problemáticas principales.
2. **Evaluación de tecnologías de comunicación:** Explorar el funcionamiento de tecnologías de comunicación y valorar su adecuación en escenarios de alta frecuencia de mensajes.
3. **Explorar el entorno de simulación de CoppeliaSim:** Estudiar el uso de este simulador y los usos que se le pueden dar.
4. **Diseño del protocolo:** Especificar el diseño de un protoclo de comunicación que permita transportar datos de manera ligera contemplando mecanismos de serialización.
5. **Desarrollo del sistema:** Implementar la lógica de control y traducción entre el simulador CoppeliaSim y clientes externos que permita el manejo de robots haciendo uso de la implementación protocolo de comunicación mencionado en el punto anterior.
6. **Desarrollo de comportamientos básicos de robots:** Demostrar tareas colaborativas (evitar colisiones, sincronizar movimientos) y de agentes solitarios para evidenciar la utilidad del protocolo.

7. **Analizar posibles mejoras:** tanto en el protocolo como en el desarrollo del sistema.

1.3. Estructura de la memoria

- **Capítulo 1:** Se introduce el proyecto, la motivación y problemas que plantea solucionar además de los objetivos a cumplir y la estructura de la memoria.
- **Capítulo 2:** Se revisan las tecnologías, librerías de comunicación y otros conceptos que se mencionarán a lo largo de este trabajo, haciendo especial hincapié en las utilizadas en la implementación.
- **Capítulo 3** Se describe la base histórica y práctica de la robótica multirobot, desde sistemas teleoperados hasta escenarios donde múltiples robots cooperan mediante protocolos de red avanzados.
- **Capítulo 4:** Se detalla el protocolo de comunicaciones, su estructura de paquetes, su uso en la integración con CoppeliaSim y su correspondiente implementación.
- **Capítulo 5:** Se discuten las contribuciones del proyecto, sus limitaciones, y se plantean mejoras potenciales.
- **Capítulo 6:** Finalmente se analiza el impacto de la investigación y el desarrollo, teniendo en cuenta los Objetivos de Desarrollo Sostenible de la agenda 2030.

Capítulo 2

Contexto

2.1. Arquitecturas de Control en Sistemas Multi-Robot

En el capítulo anterior se hecho mención a la configuración de cómo los robots se organizan y comparten información, es decir a la aquitectura de control que siguen [1]. Entre los modelos clásicos de estas arquitecturas se distinguen:

- **Centralizadas:** Un único nodo (o controlador maestro) concentra la información y toma decisiones para toda la flota de robots. Es sencillo de implementar cuando se dispone de un *vantage point* [1] , pero puede volverse un cuello de botella en sistemas con un número grande de robots. Además, sufre de vulnerabilidad ante fallos del nodo central.

- **Jerárquicas:** Organizan a los robots en niveles o grupos, de modo que cada líder coordina un subconjunto. Se inspira en estructuras de mando y control (por ejemplo, organizaciones militares). Aunque ofrece mejor escalabilidad que la arquitectura centralizada, puede presentar puntos críticos de fallo en los nodos superiores de la jerarquía dando lugar al mismo problema que en el caso anterior.

- **Descentralizadas:** Cada robot toma sus decisiones de forma autónoma en base a la información obtenida localmente o gracias a la comunicación con sus vecinos. Ganan en robustez, pues no dependen de un nodo central, sin embargo, el reto principal radica en asegurar la coherencia global, ya que si las metas cambian todo el equipo debe enterarse y adaptarse al nuevo contexto. El uso de algoritmos como el utilizado en uno de los primeros estudios de conductas sociales en entornos multi-robot por Maja J. Matarić denominado *Nerd Herd* [4] puede atenuar esta dificultad [1].

- **Híbridas:** Mezclan control local y global. Por ejemplo, cada robot opera de forma autónoma en situaciones normales, pero existe un mecanismo de coordinación superior que unifica los objetivos generales o resuelve conflic-

[1] Un vantage point se refiere a una ubicación estratégica desde la cual el controlador puede monitorear eficazmente a todos los robots.

tos. Esta aproximación busca un equilibrio entre la robustez descentralizada y la dirección unificada de un controlador.

La elección de la arquitectura depende de factores como la necesidad de escalabilidad del sistema, la complejidad de tareas que necesitan realizar o la infraestructura de comunicación disponible.

2.2. Simulador CoppeliaSim

CoppeliaSim es un entorno de simulación robótica desarrollado por Coppelia Robotics, introducido originalmente bajo el nombre V-REP (Virtual Robot Experimentation Platform) alrededor de 2010. Desde sus inicios, se ha utilizado ampliamente en ámbitos académicos, de investigación y en proyectos industriales debido a su versatilidad a la hora de permitir el uso de diferentes motores de física [5], [6].

En cuanto al acceso a esta plataforma de simulación, el entorno tiene diferentes versiones accesibles para el público, la primera es su versión `edu`, la cual da acceso a todo el software a estudiantes siempre y cuando el objetivo de su uso no sea comercial. Las otras dos versiones son `player` y `pro`, la primera da acceso a las funciones de simulación pero no es posible editar, mientras que la última version da acceso completo a la plataforma a cambio de un precio, el cual puede llegar a los `$4,270` por una licencia de por vida.

Además de ser multiplataforma, CoppeliaSim está basada en una arquitectura de control distribuida, y cada objeto puede controlarse a través de scripts, un plugin o una API remota (ZeroMQ RemoteAPI) [5], [7].

La API remota permite controlar y monitorear la simulación desde lenguajes de programación populares como C/C++, Python, Java o MATLAB. Esta característica resulta esencial para integrar algoritmos de coordinación y protocolos de comunicación externos en un contexto multi-robot, ya que cada robot puede operar sus propios procesos de control y enviar órdenes al simulador de forma asíncrona.

Sus simulaciones permiten el uso de diferentes motores de física ya integrados como Bullet, ODE, Vortex o Newton, dejando a sus desarrolladores tener el control sobre el tipo de interacciones físicas realistas como colisiones, fricción o dinámica de cuerpos rígidos que necesitan para el proyecto que deseen desarrollar. Asimismo, el simulador incluye una biblioteca de modelos realistas de componentes, equipamiento, infraestructura, objetos de naturaleza y robots, los cuales sirven para generar una escena ideal de acuerdo con el proyecto que se quiera realizar, se muestran ejemplos en la Figura 2.1.

Entre los robots que se pueden escoger, hay 2 tipos, `robots no móviles` los cuales están anclados al suelo `móviles`, los cuales tienen ruedas u otro tipo de motor que permita si cambio de posición sobre el suelo, come el Pioneer 3-DX [8], un robot serpiente o un robot con forma humanoide. La mayoría de estos modelos de robots vienen equipados con sensores virtuales (cámaras RGB, sensores láser, ultrasonidos, etc.) que permiten su entendimiento del entorno. Gracias a

todos estos modelos, los usuarios pueden recrear escenarios de diversa complejidad y llevar a cabo experimentos sobre algoritmos de localización, mapeo, navegación y coordinación sin poner en riesgo hardware real.

Otra de las características destacadas de CoppeliaSim es su modo de stepping, que posibilita avanzar la simulación *tick a tick* [2] , sincronizándola con la lógica externa. Esto da paso a que el entorno de simulación creado por un usuario pueda ser utilizado para investigaciones donde el tiempo de procesamiento, o la precisión de los algorítmos sea crítico.

En el contexto de sistemas multi-robot, CoppeliaSim constituye una plataforma idónea para la experimentación y validación de algoritmos, analizando tanto la eficacia de las estrategias de comunicación como el desempeño de los robots en entornos virtuales de distinta complejidad. El uso del simulador permite replicar escenarios complicados por ejemplo, entornos laberínticos, despliegues masivos de robots, o simulaciones con múltiples sensores, sin los riesgos ni los costos del equipamiento físico durante la etapa de investigación y desarrollo de un proyecto.

2.3. Pioneer 3-DX

El Pioneer 3-DX es el robot móvil inteligente más utilizado globalmente en educación e investigación [8]. Es un robot muy sencillo, de dos ruedas, y dos motores asociados a cada una de esas ruedas. Además está equipado con tecnología sonar lo cual permite que sean utilizados en aplicaciones de mapeado, monitorización del entorno y trabajos más sencillos [9].

En su versión más común, tiene 16 sensores sonar, 8 en la parte frontal y 8 en la trasera, los círculos en la imagen del robot mostrada representan estos sensores en el simulador de CoppeliaSim.

El Pioneer 3-DX es el robot móvil inteligente más utilizado globalmente en educación e investigación [8].

Es un robot de dos ruedas con motores independientes, lo que le permite una maniobrabilidad eficiente y un amplio rango de aplicaciones en robótica móvil. Su versión estándar incluye 16 sensores sonar (8 frontales y 8 traseros), los cuales proporcionan información para evitar obstáculos y realizar tareas de navegación autónoma [9]. Además se le pueden añadir distintos sensores adicionales como cámaras RGB, sensores LIDAR o módulos IMU.

Todas estas características hacen que su en control de velocidades, mapeados de entorno, análisis de algoritmos de circulación y monitorización del entorno sean comunes.

[2] Un tick representa un pequeño intervalo de tiempo en el cual se ejecutan las actualizaciones del simulador.

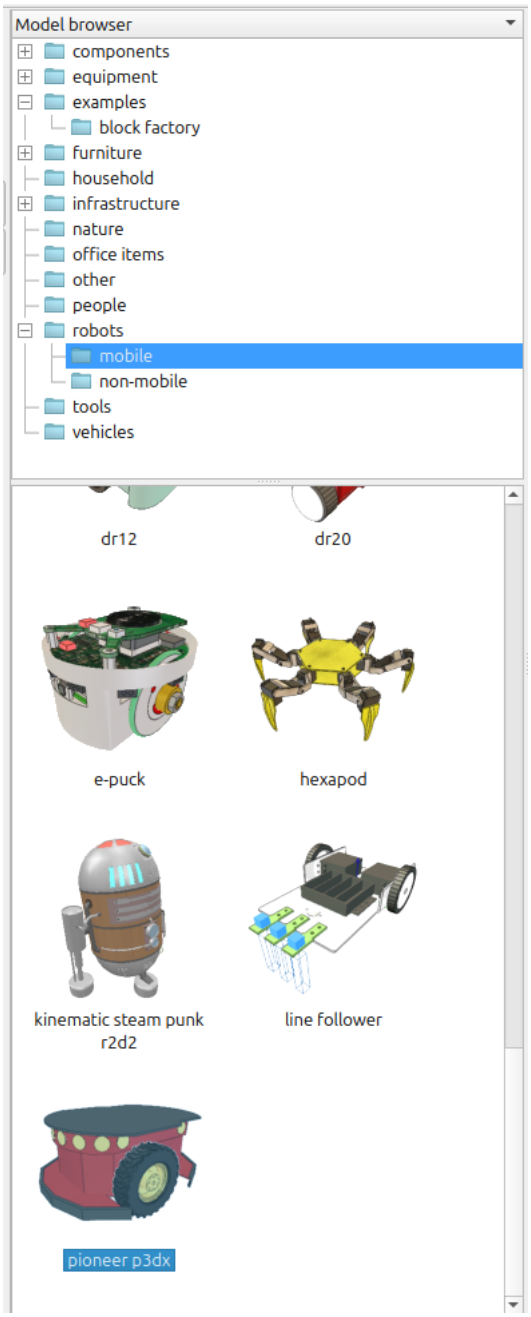

Figura 2.1: Distintos modelos en el simulador.

Figura 2.2: Representación del Pioneer 3-DX en CoppeliaSim.

2.4. Protocolos de Comunicación

Un protocolo de comunicación establece las reglas y formatos para el intercambio de datos entre agentes (robots, servidores, sensores, etc.). En robótica, estos protocolos deben responder a requisitos específicos:

- **Ligereza y eficiencia:** minimizan la sobrecarga en la transmisión, relevante en entornos con ancho de banda limitado o con muchos agentes que generan datos.
- **Escalabilidad:** la capacidad de incluir más robots sin penalizar en exceso el rendimiento global.
- **Flexibilidad:** adaptación a distintas topologías de red (Wi-Fi, 4G, 5G, etc.) y patrones de uso (publicación/suscripción, request-reply, etc.).
- **Robustez y tolerancia a fallos:** ante caídas de nodos o enlaces, el sistema debe reconfigurarse o recuperarse sin pérdida completa de funcionalidad.
- **Seguridad:** con la popularización de robots conectados a Internet, medidas como TLS, autenticación y control de acceso fino se vuelven indispensables.

La evolución de la comunicación en la web y la necesidad de interoperabilidad han impulsado la adopción de modelos asíncronos (por ejemplo, publish-subscribe) y la integración con middleware especializados para optimizar el *throughput* [3] y reducir la latencia [10].

2.5. Middleware de Comunicación: ZeroMQ (ZMQ)

ZeroMQ (ZMQ) es un framework de mensajería que ofrece patrones de comunicación (*publish-subscribe, request-reply, push-pull, etc.*) sobre sockets. A diferencia de middleware más completos como ROS o DDS, ZeroMQ se centra en proveer una capa de transporte ligera y flexible, que:

[3] La cantidad de datos que pueden ser transmitidos a través de un canal de comunicación.

- Permite **múltiples patrones de comunicación**, adaptándose a sistemas con nodos que se comunican *uno a uno* o *uno a muchos*.
- Resulta *sencillo de integrar* en C/C++, Python y otros lenguajes, facilitando la incorporación a proyectos de todo tipo.

2.5.1. CZMQ: Un Enfoque de Alto Nivel para ZeroMQ

Para simplificar aún más el uso de ZeroMQ, se desarrolló CZMQ, una capa de abstracción de alto nivel que proporciona una API más amigable y funcional para trabajar con ZeroMQ que además es recomendada para el desarrollo en C [10], [11].

CZMQ facilita la implementación de patrones de mensajería complejos mediante el uso de funciones de alto nivel que encapsulan la lógica algo repetitiva de ZeroMQ, además proporciona mecanismos más sencillos para el manejo de errores y recuperación de conexiones.

2.5.2. Conexión con la lógica multi-robot

Como se describe en la guía oficial de ZMQ [10], el diseño asíncrono de ZeroMQ encaja bien con las arquitecturas distribuidas, permitiendo que un robot publique lecturas de sensores sin bloquear la ejecución de sus demás subsistemas. Además, puede coexistir con otras tecnologías de middleware, dándole al desarrollador libertad para componer soluciones híbridas. Además, como ya se ha comentado en la sección 2.2, CoppeliaSim trabaja con esta librería para el uso de su RemoteAPI.

2.6. Serialización de Datos

La serialización de datos define cómo se empaquetan y desempaquetan los mensajes intercambiados entre endpoints. En el contexto de este TFG, esto puede ser los paquetes que intercambien distintos robots entre sí o con un servidor dependiendo de la arquitectura. Este proceso impacta directamente el rendimiento de la comunicación. Entre los formatos más utilizados se destacan JSON, XML o Protobuf [12].

Al diseñar un protocolo de comunicaciones en robótica, es fundamental equilibrar la legibilidad y la eficiencia, considerando la frecuencia de envío de datos, la importancia de la información y la capacidad de cómputo de los agentes.

Capítulo 3

Estado del Arte

3.1. Origen e Historia de la Robótica Multi-Robot

El trabajo pionero de Maja J. Matarić a mediados de los años noventa [4] supuso un hito en la comprensión de comportamientos colectivos en robótica. En su estudio Nerd Herd, se mostraba cómo múltiples robots podían cooperar basándose en reglas de interacción local, inspiradas en series vivos como abejas u hormigas, las cuales consiguen objetivos complejos separando tareas simples entre todo el enjambre u colonia. Estas ideas sentaron las bases de dos grandes paradigmas en sistemas multi-robot [1]:

- **Robótica de Enjambre (Swarm Robotics)**: Centrada en la cooperación de un gran número de robots principalmente homogéneos, que interaccionan mediante comunicaciones locales mínimas y generan comportamientos colectivos emergentes [1].
- **Sistemas Cooperativos Intencionales**: Fomentan la heterogeneidad y la comunicación explícita con el fin de resolver tareas más complejas de forma coordinada. Con frecuencia, requieren mayor intercambio de datos y algoritmos avanzados de planificación global o consenso distribuido.

Con el crecimiento de las redes inalámbricas y el auge de Internet, los sistemas multi-robot pasaron a ser cada vez más distribuidos y sofisticados. Surgieron aplicaciones en la industria manufacturera, logística y exploración, así como en el ámbito de la asistencia sanitaria, donde varias máquinas cooperan para aumentar la productividad y reducir riesgos [13], [14].

3.2. Uso Actual de Tecnologías Multi-Robot

La idea de que robots trabajen conectados entre sí se ha extendido a numerosas industrias, motivada por la convergencia de sensores avanzados, redes inalámbricas y algoritmos de inteligencia artificial [1]. Algunos ejemplos destacados incluyen:

- **Logística y Almacenes**: Empresas como Amazon Robotics coordinan flotas de robots para mover mercancías y optimizar recorridos. Suelen emplear sensores de proximidad para evitar colisiones y protocolos de comunicación de baja latencia.
- **Aplicaciones Domésticas y de Servicio**: Robots aspiradores (por ejemplo, `Roomba`) o asistentes de limpieza y mapeo de entornos, muchos equipados con sonar o infrarrojos para la detección de obstáculos. Asimismo, se han introducido agentes capaces de manejar la domótica en casa [1].
- **Agricultura de Precisión**: Drones y vehículos terrestres coordinados para tareas de siembra, fumigación y recolección. Ajustan su posición y acciones en tiempo real, basándose en datos de GPS y metereológicos.
- **Monitoreo y Rescate**: Enjambres de robots acuáticos y aéreos para misiones de búsqueda y respuesta ante desastres [1], aprovechando sensores de sonar para mapear entornos subacuáticos o áreas con poca visibilidad.
- **Militar y Defensa**: Drones y submarinos no tripulados se coordinan, intercambiando datos de radar, sonar y cámara infrarroja en tiempo real. En conflictos recientes, como las guerra en Ucrania, los drones han demostrado ser una herramienta clave tanto para el reconocimiento como para ataques de precisión. La guerra ha evidenciado el papel de los drones autónomos y semiautónomos en la recopilación de inteligencia, el reconocimiento de posiciones enemigas y la ejecución de ataques de alta precisión con costos reducidos [15].

En muchos de estos casos, la arquitectura descentralizada cobra relevancia para gestionar grandes flotas y operar en zonas con conectividad inestable o de alto riesgo como por ejemplo entornos afectados por desastres naturales o conflictos armados.

3.3. Inteligencia Artificial y Tareas Avanzadas en Multi-Robot

3.3.1. Pathfinding y Coordinación Distribuida con Sonar

Distintas aplicaciones industriales o domésticas integran robots con sonar para mejorar la detección de obstáculos en entornos con baja iluminación o superficies reflectantes [16]. Se combinan algoritmos de pathfinding (*A*, D* Lite, RRT, etc.*) con señales de proximidad, reduciendo colisiones y optimizando trayectorias en almacenes densos o pasillos estrechos.

3.3.2. Aprendizaje por Refuerzo

El aprendizaje por refuerzo (en inglés, `Reinforcement Learning`, RL) consiste en que un agente que interactúa con un entorno decida sus acciones con el fin de maximizar una señal de recompensa [17]. A diferencia de los métodos puramente supervisados, el agente no recibe ejemplos correctos de la acción a

realizar, sino que explora opciones, y el entorno le proporciona recompensas (positivas o negativas) en función de su desempeño a la hora de lograr las metas propuestas.

Según Richard S. Sutton [17], un sistema de aprendizaje por refuerzo se compone de:

- **Política**: Regla de mapeo desde estados a acciones. Determina el comportamiento del agente.
- **Señal de recompensa**: Indica el objetivo inmediato. El agente trata de maximizar estas recompensas cumpliendo objetivos o pasos para lograrlos.
- **Función de valor**: Decide lo correcto a largo plazo en cada estado en el que el agente se encuentre, es decir, permite a un agente medir no solo la recompensa que va a conseguir, sino también las consecuencias futuras de conseguirla. Por ejemplo, si un robot se desplaza a un punto donde va a obtener una recompensa alta, pero no podrá volver a recoger otras recompensas, esta función provocará que éste no se desplace a ese punto.
- **Modelo del entorno**: Ofrece una proyección interna de cómo evoluciona el entorno ante distintas acciones. Su uso se asocia con métodos de planificación en los que se decide un posible futuro sin haberlo realizado todavía.

En el aspecto del aprendizaje por refuerzo multi-agente, cuando varios agentes comparten un mismo entorno y tienen metas a conseguir, éstos pueden beneficiarse de la experiencia compartida, ajustando sus políticas en función de una recompensa global o de recompensas individuales pero correlacionadas [2]. Las políticas también pueden ser competitivas: cada agente trabaja para conseguir recompensas para sí mismo, pero recibe información de otros agentes o del entorno.

3.3.3. Visión y Reconocimiento de Entornos Complejos

Las técnicas de `computer vision` o visión por ordenador han evolucionado desde algoritmos basados en extracción de características clásicas (por ejemplo, SIFT o SURF) hasta enfoques modernos con redes neuronales convolucionales (`CNNs`) [18]. Si bien las CNN se propusieron en la década de 1980, su adopción masiva ocurrió a partir de 2012, cuando la disponibilidad de tarjetas gráficas con capacidad de procesamiento de datos y la necesidad de obtención de datos a gran escala las impulsó de manera determinante [19].

En contextos multi-robot, la visión por ordenador puede permitir la cobertura cooperativa de zonas extensas, la detección de objetos, así como la creación de mapas 3D compartidos [1], [20]. En sistemas de este tipo, los robots son utilizados para compartir perspectivas, donde varios robots envían imágenes con el objetivo de crear una representación virtual de la realidad. Esto resulta útil en situaciones como la detección de fuegos, y la búsqueda de animales, o personas desaparecidas en entornos complejos o poco iluminados.

3.4. Tendencias en Comunicación y Middleware

3.4.1. Protocolos de Comunicación en la Actualidad

La complejidad y la escala de los sistemas multi-robot demandan `middleware` eficientes y escalables. Entre las opciones más populares:

- **ZeroMQ**: Ofrece patrones de comunicación (`Req-Rep`, `Pub-Sub`, `Push-Pull`, `etc.`) con bajo overhead, facilitando la creación de protocolos a medida [10].
- **ROS 2**: Brinda una infraestructura más orientada a la robótica, con descubrimiento automático y comunicación basada en `topics`, aunque introduce mayor `overhead` y dependencias [21].
- **MQTT y AMQP**: Conocidos en aplicaciones IoT [1], permiten comunicación publish-subscribe, pero requieren un broker central ya que son protocolos pensados para ambientes relativamente pequeños, lo que no siempre resulta apropiado en sistemas multi-robot móviles o en misiones de alta autonomía.

Aspectos Críticos: Latencia, Robustez y Escalabilidad

En escenarios multi-robot, la latencia es esencial para evitar colisiones o coordinar movimientos. La robustez se logra mediante `mesh networks`, tolerancia a fallos y reintento de mensajes, mientras que la escalabilidad favorece arquitecturas con descubrimiento automático y modos asíncronos, facilitando la integración de nuevos robots.

3.5. Serialización de Datos: Técnicas y Avances

3.5.1. Formatos Binarios vs. Humanamente Legibles

Aunque `JSON` o `XML` siguen siendo populares por su legibilidad, la frecuencia alta de transmisión en escenarios multi-robot impulsa el uso de formatos binarios como `Protobuf`, `CBOR` o `MessagePack`, que ofrecen mejor rendimiento [12].

3.5.2. Soluciones Ad-hoc y Personalizadas

En la investigación multi-robot, a menudo se opta por formatos propios de serialización (ad-hoc) para reducir overhead y controlar la estructura de los datos [10]. Esto puede complicar la extensibilidad futura, pero garantiza un control absoluto sobre el coste computacional.

3.6. Simuladores y Entornos de Prueba en la Actualidad

La validación temprana de arquitecturas y algoritmos en entornos de simulación virtuales es un estándar hoy en día, por eso hay varios simuladores aparte

[1] Internet of Things o el Internet de las cosas, refiere en este caso al área de la domótica.

de CoppeliaSim refsec:coppeliasim que son utilizados con frecuencia con este propósito:

- **Gazebo/Ignition**: Muy vinculado a ROS, con realismo en colisiones, dinámica de cuerpos, y amplia biblioteca de robots y sensores. Su extensa biblioteca de robots y sensores facilita la experimentación en proyectos académicos e industriales.
- **Webots**: Orientado a la docencia y la investigación, con amplia base de dispositivos simulados y curva de aprendizaje menor.

Aun así, la verificación final en prototipos físicos sigue siendo esencial para comprobar la fiabilidad de la comunicación y la gestión energética en condiciones reales [2].

3.7. Aplicaciones Domésticas y Militares

3.7.1. Robots de Limpieza y Asistentes para el Hogar

La adopción de **robots domésticos** ha crecido sustancialmente, especialmente en tareas de limpieza (aspiradores, fregadoras). Incorporan sonar, infrarrojos u otros sensores para mapear el entorno y esquivar obstáculos, integrándose a menudo con aplicaciones en la nube y redes Wi-Fi [1]. Se han propuesto también robots de compañía y telepresencia para atender a población anciana o enferma en entornos residenciales.

3.7.2. Aplicaciones Militares y de Seguridad

En la industria de la defensa, proyectos como `Future Combat Systems` aplican un enfoque de `network-centric warfare`, donde UAVs, UGVs y vehículos submarinos cooperan para reconocimiento, cartografiado y asistencia en misiones críticas [1]. Muchos integran sonar, radares y sensores infrarrojos, comunicándose a través de redes malla resistentes a fallos de enlace. Si bien han demostrado eficacia, sigue siendo común la necesidad de múltiples operadores humanos por cada UAV o UGV avanzado, lo que impulsa nuevas investigaciones en autonomía y protocolos más confiables.

3.8. Retos Principales y Oportunidades de Futuro

En base a los estudios actuales y del crecimiento en el uso de sistemas multirobot, se plantean desafíos fundamentales:

1. **Coordinación Distribuida con IA**: Perfeccionar algoritmos de IA multiagente que reduzcan la dependencia de nodos centrales y optimicen la colaboración en escenarios cambiantes [2].
2. **Comunicación Resiliente**: Asegurar protocolos que se reconfiguren automáticamente ante fallos de red o ataques cibernéticos, usando cifrado robusto y autenticación [1].

3. **Eficiencia Energética**: Al incrementar la complejidad sensorial y el uso de redes inalámbricas, se requiere equilibrar el consumo energético y la calidad de los datos.
4. **Interoperabilidad y Estándares**: La ausencia de estándares unificados dificulta la integración de robots y sensores heterogéneos. Iniciativas como el `Network Robot Forum` [1] buscan unificar interfaces para la robótica doméstica e industrial.
5. **Reconfiguración en Entornos Dinámicos**: Cambios drásticos derrumbes, explosiones, fallos de red demandan capacidades de reorganización rápida de la flota, sin comprometer la misión principal.

Capítulo 4

Desarrollo

El objetivo de este capítulo es describir la ruta que se ha tomado a la hora de diseñar e implementar un protocolo de comunicación ligero con la ayuda del framework de ZeroMQ.

Una vez diseñado este protocolo, se describirá la implementación de un servidor que se conecte con el simulador CoppeliaSim a través de una API. Una vez esté conectado, podrá actuar como traductor entre clientes que se conectan con este servidor a través de sockets y que representan robots y el simulador CoppeliaSim. Estos clientes podrán hacer uso del protocolo de comunicación diseñado previamente para tener un control sobre los la simulación y los robots P3DX que representan.

4.1. Diseño e implementación del Protocolo

El protocolo de comunicación está desarrollado en C pero está adaptado para ser utilizado por C++, y se apoya en la biblioteca ZeroMQ para el envío y la recepción de mensajes.

La implementación se divide en dos ficheros principales:

- **`commProt.h`**: Contiene las estructuras de datos, enumeraciones y declaraciones de funciones.
- **`commProt.c`**: Implementa la lógica de serialización, deserialización y envío/recepción de mensajes.

4.1.1. Estructura de los mensajes del protocolo

Los mensajes de este protocolo están definidos como paquetes. Un paquete puede utilizar un máximo de 255 bytes y estos sirven para describir la acción, o respuesta que el remitente o el receptor quieren realizar o transmitir junto a los parámetros que acompañan a esta acción, que pueden ser de distintos tipos:

Listing 4.1: Tipos aceptados como parámetros

```
typedef enum
{
    ZCOM_TYPE_UINT8 = 0x01,
    ZCOM_TYPE_INT   = 0x02,
    ZCOM_TYPE_LONG  = 0x03,
    ZCOM_TYPE_FLOAT = 0x04,
    ZCOM_TYPE_DOUBLE = 0x05,
    ZCOM_TYPE_UNDEFINED = 0xFF
} ZCOM_ParamType;
```

Dejando esto claro, un paquete de datos en el protocolo está representado por la siguiente estructura:

Listing 4.2: Estructura principal `zcom_packet` en `commProt.h`

```
typedef struct
{
    uint8_t **parameters;
    size_t *param_sizes;
    ZCOM_ParamType *param_types;
    uint8_t action;
} zcom_packet;
```

Donde el parámetro `action` representa la acción o respuesta, `parameters` representa un array de bytes que contiene los parámetros asociados a dicha acción y el resto de parámetros son utilizados para asegurar que éstos parámetros de distintos tipos puedan ser reconstruidos a su tipo original.

La un paquete de este tipo en forma serializada, se representa como un array de bytes, organizados de la siguiente manera:

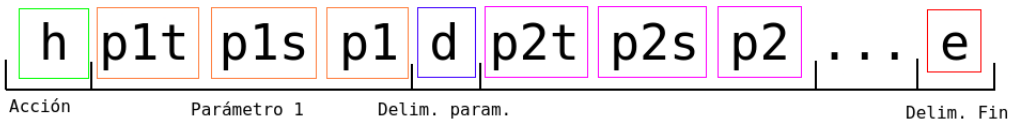

Figura 4.1: Descripción de un paquete serializado.

Donde cada una de éstas partes es:

- **Acción (h)**: 1 byte (8 bits guardados en un tipo `uint8_t`.).
- **Parámetro (p1t, p1s, p1)**: Mínimo 3 bytes, con 2 de cabecera y el resto siendo el parámetros.
  - Tipo del parámetro.
  - Tamaño del parámetro.
  - Representación hexadecimal en bytes del parámetro.

- **Delimitador de parámetro (d)**: Para representar cuándo termina el último parámetro, se utiliza el caracter '|' o 0x7C en hexadecimal como delimitador.
- **...** : Parámetros adicionales si existieran.
- **Delimitador de fin de mensaje (e)**: Un paquete serializado termina cuando aparece el caracter NULL.

En caso de que una acción no requiera de parámetros, el paquete estará formado por la acción y delimitador de fin de paquete, y podría tener la siguiente forma:

0x01 0x00

Mientras que un paquete que contenga 3 variables de tipo `uint8_t`, `double` y `double` se representaría como una serie de bits así:

05 01 01 03 **7C** 05 08 C3 F5 28 5C 8F C2 F1 3F **7C** 05 08 C3 F5 28 5C 8F C2 01 40 **00**

El primer byte 0x05, representa la acción que se quiere realizar, la cuál requiere de 3 parámetros.

- El primero 0x01 (`uint8_t` de acuerdo con el Listing 4.1), tiene longitud 0x01 y valor 0x03.
- El segundo 0x05, tiene 8 bytes de longitud y termina cuando aparece el caracter 0x7C.
- El segundo 0x05, tiene 8 bytes de longitud y termina cuando aparece el caracter 0x00 indicando que el paquete ha terminado.

4.1.2. Tipos de Acciones o Respuestas

Las acciones que pueden ser transmitidas por este protocolo se definen con el tipo `uint8_t` en una enumeración `ZCOM_Action`, donde el valor de cada acción es distinto para poder diferenciarlas.

Listing 4.3: Definición de acciones en el protocolo

```
typedef enum {
    ZCOM_ACTION_INIT = 0x00,
    ...
    ZCOM_ACTION_SET_MOTOR,
    ZCOM_ACTION_RESPONSE_OK,
    ...
    ZCOM_ACTION_UNSET = 0xFF
} ZCOM_Action;
```

Cada acción definida dentro de esta enumeración puede esperar una cantidad específica de parámetros, los cuales deben estar definidos en el array

`ZCOM_ACTION_PARAMS`:

Listing 4.4: Número de parámetros esperados por acción

```
static const uint8_t ZCOM_ACTION_PARAMS[] = {
    [ZCOM_ACTION_INIT]        = 0,
    ...
    [ZCOM_ACTION_SET_MOTOR]   = 3,
    ...
    [ZCOM_ACTION_RESPONSE_OK]= 0
};
```

Se observa que para las acciones de ejemplo mostradas en `ZCOM_Action` existe un valor asociado, a ésta en `ZCOM_ACTION_PARAMS` donde 0 indica que no requiere de parámetos y 3 que requiere de este número de parámetros.

Ejemplo de acción con que utiliza 0 parámetros

`ZCOM_ACTION_INIT` indica que no se esperan parámetros, por lo que el buffer podría ser:

```
0x00 0x00
```

(mostrando la *acción* y el *delimitador de fin de paquete*).

Ejemplo de acción usando 3 parámetros

`ZCOM_ACTION_SET_MOTOR` requiere tres parámetros:

- Un identificador de robot (1 byte).
- Una velocidad para la rueda izquierda en doble precisión (8 bytes).
- Una velocidad para la rueda derecha en doble precisión (8 bytes).

Si se asigna `robot_id = 0x01`, `left_speed = 1.11` y `right_speed = 2.22`, el buffer de ejemplo sería:

```
05 01 01 03 7C 05 08 C3 F5 28 5C 8F C2 F1 3F 7C 05 08 C3 F5 28 5C
                        8F C2 01 40 00
```

Igual que en el ejemplo anterior, el campo de acción en el struct tendrá el valor de `ZCOM_ACTION_SET_MOTOR` o 0x05 como vemos en el primer byte del buffer, mientras que los valores de los campos asignados a los parámetros se rellenan de manera organizada, dejando un paquete que impreso tiene la siguiente forma:

Listing 4.5: Paquete impreso en consola

```
--- PACKET ---
Action: ZCOM_ACTION_SET_MOTOR
Parameter 0: Type = 0x01, Size = 1, Value = 0x03

Parameter 1: Type = 0x05, Size = 8,
Value = 0xC3 0xF5 0x28 0x5C 0x8F 0xC2 0xF1 0x3F

Parameter 2: Type = 0x05, Size = 8,
Value = 0xC3 0xF5 0x28 0x5C 0x8F 0xC2 0x01 0x40
--- PACKET END ---
```

4.1.3. Serialización y Deserialización

Serialización

La función de serialización empaqueta toda la información en un buffer binario de tamaño máximo `ZCOM_MAX_PACKET_SIZE` (255), el cuál puede ser modificado para soportar tamaños superiores a este número si así se requiriera.

Esta función recibe:

- Un buffer: Array de bytes que se debe rellenar y dejar con el formato visto en la imagen 4.1 que no podrá tener un tamaño mayor a 255 bytes.
- Un puntero a un paquete: Este debe estar relleno con los datos que se quieren transmitir, incluyendo su acción, el tipo de los parámetos, la longitud de cada parámetro y por supuesto el array de parámetros. Hay que tener en cuenta que si alguno de estos datos no es correcto, o el número de parámetros esperado por la acción es distinto, la serialización será errónea.
- Una estructura resultado: Almacenará el resultado de la operación de serialización junto al número de bytes utilizado en el buffer. Estos datos serán útiles no solo para saber si el resultado de la función es correcto, sino para posteriormente saber el número de bytes que debemos enviar a través de un socket. La estructura está representado por siguiente tipo: `ser_deser_result`.

El Listing 4.6 muestra un fragmento con los apartados importantes de la función de serialización descrita, omitiendo de este framento todo tipo de comprobación de errores.

Listing 4.6: Fragmento de `zcom_serialize` en `commProt.c`

```
void zcom_serialize(uint8_t buffer[ZCOM_MAX_PACKET_SIZE],
                    const zcom_packet *packet,
                    ser_deser_result *result)
{
    // Inserta la acción en la primera dirección del buffer
    buffer[offset++] = (uint8_t)packet->action;

    // Inserta los parámetros requeridos por la acción en
    // el buffer
    for (size_t i = 0; i < required_params; i++)
    {
        buffer[offset++] = (uint8_t)packet->param_types[i];
        buffer[offset++] = (uint8_t)packet->param_sizes[i];

        memcpy(&buffer[offset],
               packet->parameters[i],
               packet->param_sizes[i]);

        offset += packet->param_sizes[i];

        // Inserta el delimitador de parámetros
        if (i < required_params - 1)
            buffer[offset++] = ZCOM_PARAMETER_DELIMITER;
    }

    // Inserta el delimitador de fin de paquete
    buffer[offset++]   = ZCOM_PACKET_DELIMITER;

    result->status     = ZCOM_STATUS_OK;
    result->used_bytes = offset;
}
```

Deserialización

Para la deserialización de un buffer a un paquete, se realiza el proceso inverso al visto anteriormente, es decir partimos de un array de bytes recibido por un socket, sin tener información externa a la escrita sobre éste.

Esta función recibe:

- Un buffer: Array de bytes relleno que deberá desglosar y almacenar en orden en un paquete como el del Listing 4.2.
- Un puntero a un paquete: Éste es el paquete que debe ser rellenado por esta función para que pueda ser utilizado por el receptor.
- Una estructura resultado: Almacenará el resultado de la operación de deserialización junto al número de bytes leídos sobre el buffer para comprobar si

éste es igual al número de bytes recibidos por el socket.

El siguiente Listing muestra un fragmento con los apartados importantes de la función de deserialización descrita:

Listing 4.7: Fragmento de `zcom_deserialize` en `commProt.c`

```
void zcom_deserialize(const uint8_t buffer[ZCOM_MAX_PACKET_SIZE],
                      zcom_packet *packet,
                      ser_deser_result *result)
{
    // Se extrae la acción de la primera dirección del buffer
    packet->action = buffer[offset++];

    uint8_t required_params =
            zcom_get_action_params((ZCOM_Action)packet->action);
    if (required_params == 0) {
        // La acción la acción recibida no requiere parámetros
        ...
        result->status      = ZCOM_STATUS_OK;
        return;
    }

    // Reserva memoria para almacenar los parámetros necesarios
    packet->parameters  =
    (uint8_t **)calloc(required_params + 1,
                       sizeof(uint8_t *));

    packet->param_sizes =
    (size_t *)calloc(required_params,
                       sizeof(size_t));

    packet->param_types =
    (ZCOM_ParamType *)calloc(required_params,
                       sizeof(ZCOM_ParamType));

    for (uint8_t i = 0; i < required_params; i++)
    {
        // Copia los parámetros del buffer a su
        // espacio correspondiente en el paquete
        ...
    }
    result->status     = ZCOM_STATUS_OK;
    result->used_bytes = offset;
}
```

4.1.4. Envío y Recepción de Paquetes

Para enviar y recibir los paquetes a través de la red y de esta manera enviar la información necesaria entre los agentes, se utilizan las funciones:

- `zcom_send_packet`
- `zcom_receive_packet`

éstas son capaces de transportar cualquier paquete creado usando este protocolo sobre cualquier socket de tipo `zsock_t`, utilizados por la librería de ZeroMQ en C.

Descripción del envío de paquetes

La función `zcom_send_packet` recibe un puntero a un socket y un puntero a un paquete. Este paquete se serializa en un buffer con la ayuda de la función `zcom_serialize` vista anteriormente y éste se envía sobre del socket propuesto como un array de bytes o frame `"b"` con ayuda de la función `zsock_send` descrita en la guía de ZeroMQ [10].

Descripción de la recepción de paquetes

De manera similar al envío de paquetes, la función `zcom_receive_packet` recibe un puntero a un socket, y un puntero a un paquete. Lo primero que hace tras asegurar que los argumentos de entrada son válidos, es esperar de manera bloqueante a recibir un array de bytes igual al que se envía, haciendo uso de la función `zsock_recv` [10]. Seguidamente, llama a la función `zcom_deserialize` la cuál se encarga de construir el paquete recibido como argumento en la función.

4.2. Implementación del Servidor intermediario

Como se ha comentado anteriormente, el diseño de este servidor tiene como objetivo ser un intermediario entre CoppeliaSim y los clientes responsables de la lógica aplicada por los robots u otros agentes en una simulación. Esta intermediación se consigue haciendo uso de la RemoteAPI proporcionada por Coppelia [7].

La implementación está separada en 3 secciones:

- Una clase Coppelia encargada de llamar a las funciones de la RemoteAPI.
- Una clase P3DX que representa a un robot de este tipo.
- La lógica principal del servidor, encargada de recibir y enviar paquetes a los distintos clientes.

4.2.1. Clase Coppelia

Esta clase gestiona la conexión con CoppeliaSim. Antes de nada, hay que asegurarse de que el simulador está abierto y el socket que utiliza el RemoteAPI está activo:

```
[sandboxScript:info] Simulator launched, welcome!
[Connectivity >> WebSocket remote API server@addOnScript:info] WebSocket Remote API server starting (port=23050)...
```

Este mensaje de la terminal integrada en el simulador nos indica que la inicialización del socket que utiliza el RemoteAPI está funcionando en el puerto 23050, y por tanto podemos utilizar la RemoteAPI.

La creación de un objeto de tipo Coppelia inicia la conexión con este socket. En primer lugar, crea un objeto de tipo `RemoteAPIClient` y luego otro de tipo `RemoteAPIObject::sim`, los cuales se inicializan en el constructor de este objeto de la siguiente manera: `client.getObject().sim()`.

A partir de aquí esta clase solo tendrá uso cuando un cliente se conecte y envíe paquetes que contengan acciones que requieran un cambio sobre Coppelia. Algunos ejemplos de estas acciones son:

Listing 4.8: Ejemplos de funciones que implementa la clase Coppelia

```
void clib_start_sim();
int clib_is_running();
...
void clib_stop_sim(bool wait);
void clib_set_motor_speed(P3DX &robot,
        double left_motor_speed, double right_motor_speed);
```

El código de la lógica principal ejecuta de manera bastante rápida, y existe la posibilidad de que una llamada a la RemoteAPI no haya terminado y ya se esté realizando otra llamada, de forma que para evitar errores las llamadas a RemoteAPI siempre irán acompañadas de un mutex `pthread_mutex_t api_mutex` para prevenir estos errores.

A modo de ejemplo, ésta es la implementación de la función `clib_start_sim()`, la cual se encarga de iniciar la simulación en modo stepping 2.2:

Listing 4.9: Función `clib_start_sim()`

```
void Coppelia::clib_start_sim()
{
    client.setStepping(true);
    pthread_mutex_lock(&api_mutex);
    sim.startSimulation();
    pthread_mutex_unlock(&api_mutex);
}
```

4.2.2. Clase P3DX

Como se ha comentado, esta clase se encarga de representar a un robot de tipo P3DX.

Al ser un objeto, podemos almacenar los datos necesarios para controlar sus sensores y motores en las ruedas a través de sus `"handles"`, o identificadores de sí mismo o de sus componentes. El único realmente necesario para obtener estos datos es un String que referencia al nombre que tiene el robot en el simulador. De esta manera, cuando creamos una clase de este tipo, en el constructor le pasamos el objeto `RemoteAPIObject::sim` creado en la clase de Coppelia junto al nombre del robot al que nos referimos en formato String, y éste se va a encargar de conseguir los identificadores de su sonar y de su motor en ambas ruedas.

Al igual que en la clase vista en el apartado anterior, las funciones declaradas en éste apartado solo son utilizadas en caso de que un cliente quiera conocer el valor de cada uno de los 16 sensores sonar que tiene el robot o si quiere conocer su posicion exacta en el mapa.

4.2.3. Lógica principal

Una vez definidas las clases auxiliares, se explica la implementación del servidor encargado de la recepción y el envío de paquetes con los clientes.

En primer lugar, se definen 2 sockets, uno de tipo Reply y otro de tipo Publish. Los clientes podrán conectarse a estos sockets si utilizan sockets de tipo Request y de tipo Subscribe respectivamente. Éstos sockets son creados utilizando la librería de ZeroMQ de la siguiente forma:

```
zsock_t *rep = zsock_new_rep("tcp://*:5555");

zsock_t *pub = zsock_new_pub("tcp://*:5556");
```

Esto implica que si un cliente quiere establecer una conexión con el socket Reply deberá hacerlo creando un socket de tipo Request y unlazarlo sobre el puerto 5555.

Seguidamente, dado que se asume que los mapas de CoppeliaSim utilizados en este conexto siempre tendrán robots de tipo P3DX con nombre `PioneerP3DX`, se buscarán los robots que tengan este nombre o un número en orden `PioneerP3DX1`, `PioneerP3DX2`, ... hasta que no existan más sobre el mapa de CoppeliaSim seleccionado. Una vez se han creado todos los objetos de tipo P3DX, se añaden a un mapa de robots de este tipo `std::map<uint8_t, P3DX *>g_robots;`, donde el Key para el robot es un número del 0 al 255 y será el identificador que el cliente utilice para referirse a un robot específico.

En siguiente lugar, se declara `pthread_t sonar_thread;` que podrá ser utilizado en un futuro para emitir los valores del sonar de todos los robots en el mapa. Si el servidor recibe la acción `ZCOM_ACTION_START_SONAR_FEED`, utilizará este thread para recoger los datos de todos los robots contenidos en el mapa de robots y transmitirlos de manera asíncrona a través del socket `Publisher`.

Una vez toda la inicialización ha acabado, inicia un bucle que terminará si el flag de tipo `zsys_interrupted` es activado. Este flag, de acuerdo con la guía de ZeroMQ será activado si las señales Ctrl-C o SIGTERM son recibidas [22], en cuyo caso el servidor terminará de manera segura, asegurando de liberar toda la memoria utilizada y de cerrar las conexiones abiertas con los sockets.

El bucle principal para de manera bloqueante y espera a recibir un paquete haciendo uso de la función `zcom_receive_packet`. Al recibir un paquete, una función `handleAction` es llamada para comprobar la validez de acción recibida y, en caso de formar parte de las declaradas en el enum `ZCOM_ACTION`, se hace cargo de completar los pasos necesarios además de rellenar el paquete de respuesta que será enviado al cliente.

La acción de respuesta que va a recibir el cliente suele ser informando de que la operación se ha completado con éxito o si algo ha fallado, pero también puede devolver datos importantes. Por ejemplo, cuando una acción `ZCOM_ACTION_SIM_STATE` es recibida por el servidor, el cliente espera que se le conteste con el estado de la simulación, entre las que se debe encontrar una de las vistas en la Figura 4.2.

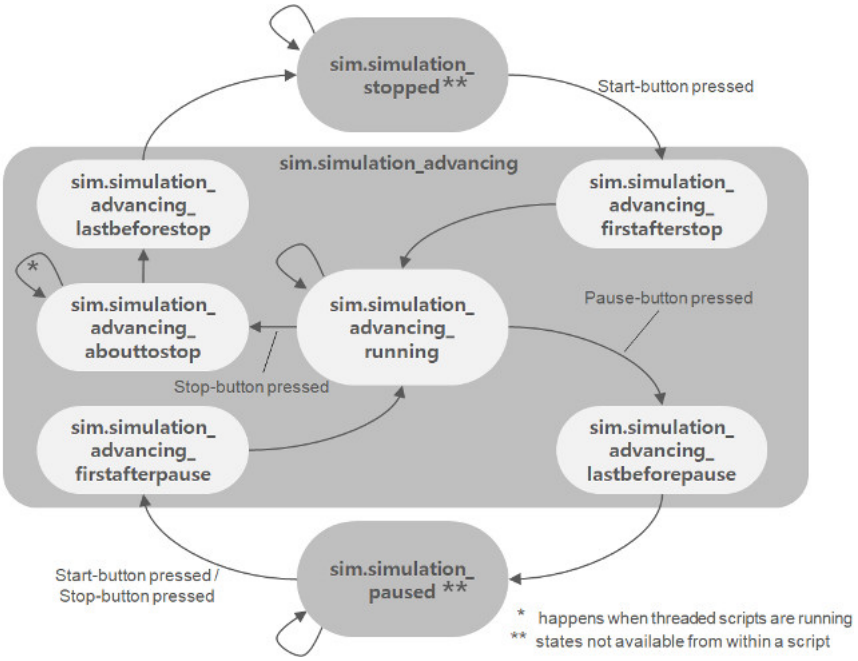

Figura 4.2: Mapa de estados posibles de la simulación.

Cuando el servidor recibe un paquete con la acción `ZCOM_ACTION_INIT`, algún cliente nos quiere indicar que quiere que se de inicio a la simulación. Con el objetivo de crear un entorno lo más real posible, se utiliza el modo stepping 2.2 y hará que la simulación avance una cantidad de tiempo fija si se ejecuta el comando client.step(). Para ello, cuando se recibe esta acción, aparte de hacer una llamada a CoppeliaSim para informarle del inicio de la simulación en este modo, el servidor crea un thread encargado de ejecutarse en bucle hasta que el cliente pida el fin de la simulación. Se ha decidido que este bucle se ejecute cada

5ms, por mantener un tiempo de ejecución bajo pero no lo suficientemente bajo como para hacer ejecuciones a máximo rendimiento. La velocidad de ejecución puede ser moficada según sea necesario utilizar mayor o menos precisión.

A modo de resumen, si el estado de la simulación es `advancing_running`, la simulación se está ejecutando, mientras que si está en el estado `stopped`, la simulación estará parada y reiniciada a su estado inicial.

4.3. Implementación de Distintos Clientes

Una vez definido el servidor y el protocolo de comunicación, muestra el desarrollo de varios clientes para ilustrar el intercambio de datos y la ejecución de acciones sobre el simulador. Cada cliente crea un `socket` de tipo **Request** para comunicarse con el **Reply** del servidor y, opcionalmente, tiene la posibilidad de subscribirse con un `socket` **Subscribe** al socket `Publisher` del servidor para recibir flujos de datos, como de los valores de sensores sonar del robot P3DX al que estén conectados.

4.3.1. Cliente avoid

El **cliente `avoid`** (Listado 4.10) demuestra cómo un robot P3DX puede desplazarse por el entorno de forma autónoma, evitando obstáculos gracias a los valores de los sensores sonar.

- **Inicio de sockets**: Se crean dos conexiones:
  - `req` (`ZMQ_REQ`) en `tcp://localhost:5555`, para enviar paquetes de tipo *request* y recibir paquetes de tipo *reply*.
  - `sub` (`ZMQ_SUB`) en `tcp://localhost:5556`, suscrito al tema `"SONAR"` en caso de ser un único robot y `"SONARn"`, donde n es el número del robot del que se quiere recibir datos periódicos de los sensores.
- **Arranque del feed de sonar** (`ZCOM_ACTION_START_SONAR_FEED`): Una vez se ha contectado a ese socket, el cliente envía un paquete que solicita el inicio de la emisión de datos sonar. Si el servidor responde con

  `ZCOM_ACTION_RESPONSE_SONAR_FEED_STARTED`, el cliente sabe que podrá recibir mediciones en la suscripción corresponspondiente.
- **Arranque de la simulación** (`ZCOM_ACTION_INIT`): El cliente envía un paquete que solicita el inicio de la simulación, provocando que CoppeliaSim active el modo stepping y que el servidor cree un hilo que ejecutará un paso de la simulación cada 5ms. En caso de que todo haya funcionado correctamente, el servidor enviará un paquete `ZCOM_ACTION_RESPONSE_OK`, con el cliente sabe la ejecución ha iniciado correctamente.
- **Función `avoid`**: En el código se define la función `avoid(...)` (Listado 4.10), que calcula las velocidades de la rueda izquierda y derecha en función de las distancias obtenidas por los sensores. Se evalúan umbrales de

detección frontal y lateral, aplicando reglas sencillas para girar y eludir obstáculos.

- **Bucle principal**:
  1. El cliente recibe datos de los sensores como un array de 16 posiciones.
  2. Llama a la función `avoid(...)` para computar las velocidades de las ruedas dependiendo de los valores medidos por el sensor.
  3. Construye un paquete `ZCOM_ACTION_SET_MOTOR` con las tres posiciones (ID del robot, velocidad izquierda, velocidad derecha) y lo envía al servidor para que éste cambie el valor de las velocidades en las ruedas del robot pedido.
  4. Espera la respuesta del servidor (`ZCOM_ACTION_RESPONSE_MOTOR_COMMAND_ACK`), asegurando que los comandos se hayan aplicado correctamente.
  5. Periódicamente verifica el estado de la simulación (`ZCOM_ACTION_SIM_STATE`) y la inicia de nuevo con la acción (`ZCOM_ACTION_INIT`) en caso de hallarla detenida.
  6. Permite al usuario pulsar `'q'` en cualquier momento para enviar un paquete de tipo `QUIT_ALL` o `QUIT_CLIENT` en caso de que se le quiera informar al servidor de que debe terminar su ejecución o en caso de que el cliente desee desconectarse.
- **Finalización**: Si el usuario pulsa `q`, el programa envía la acción correspondiente de `QUIT_ALL` o `QUIT_CLIENT` destruyendo los sockets, liberando memoria utilizada y termina.

Listing 4.10: Cliente `avoid` que recibe datos de sonar y establece velocidades

```
void avoid(double readings[], double *lspeed, double *rspeed) {
    // Lógica de evasión de obstáculos usando sensores
    ...
}

int main() {
    // Creación de sockets REQ/SUB, arranque de feed sonar,
    ...
    while (!zsys_interrupted) {
        // bucle principal donde se recibe SONAR y
        // se envía SET_MOTOR
        // Recibir SONAR, calcular velocidades con avoid(),
        // enviar SET_MOTOR
        // Leer ACK motor, verificar estado sim, posible reinit
    }
    // Liberación de recursos
    return EXIT_SUCCESS;
}
```

4.3.2. Cliente controlado por teclado

A diferencia del cliente anterior, el cliente controlado por teclado permite a un usuario dirigir manualmente las ruedas de un robot, ajustando sus velocidades con las teclas `W, A, S, D`. El esquema de comunicación sigue siendo **REQ/-REP**, pero en este caso no se suscribe a datos sonar, sino que continuamente envían mensajes de tipo `SET_MOTOR` al servidor para que éste actualice los valores de las velocidades en las ruedas del robot P3DX.

- **Captura de teclado con `ncurses`**: Esta librería permite configurar la terminal en modo no bloqueando, lo que deja detectar pulsaciones de forma asíncrona. Cada pulsación se traduce en un incremento o disminución de la velocidad de cada rueda (`W`=avanzar, `S`=retroceder, `A`=girar izquierda, `D`=girar derecha).
- **Envío de `ZCOM_ACTION_SET_MOTOR`**: Cada ciclo del bucle, el cual está marcado por el servidor, ya que el cliente debe esperar a recibir la acción de ACK en su socket REQ bloqueante, y tras leer el teclado, el cliente construye un paquete `SET_MOTOR` usando las velocidades calculadas.
- **Comprobación del estado de la simulación**: Periódicamente se envía `ZCOM_ACTION_SIM_STATE` para confirmar que la simulación siga en marcha. Si está detenida (`COPPELIA_SIM_STATE_STOPPED`), se reenvía `ZCOM_ACTION_INIT` para relanzarla.
- **Salida del programa**: Si el operador pulsa `q`, el cliente destruye los sockets y finaliza.

Este enfoque manual es útil para depurar la lógica de control o para controlar directamente el robot en escenarios específicos. Resulta también una base para desarrollar usos más sofisticados del robot.

4.3.3. Cliente multi-robot

En los dos ejemplos de clientes anteriores, vemos como cada cliente se comunica con el servidor haciendo uso del protocolo de comunicación desarrollado. Hemos comentado previamente que el servidor es capaz de tratar con varios clientes del mismo tipo comunicando Clientes(n) ->Servidor, pero que pasa si queremos hacer esto mismo, y al mismo tiempo comunicar Cliente ->Cliente.

Con el objetivo de contemplar como se desentendería este protocolo a la hora de comunicar varios clientes entre sí mientras reciven y envían datos con el servidor, se ha desarrollado el siguiente cliente sencillo.

El cliente `joint_avoid` está formado por dos ficheros, el primero conteniendo un socket de tipo Request (Cliente 1) y el segundo de tipo Reply (Cliente 2). El flujo de ejecución es igual que el que hemos visto en avoid para ambos, pero en este caso, el Cliente 1 adquiere los datos sonar del Cliente 2 y a su vez el Cliente 2 recoge los datos del Cliente 1. De esta manera, ambos se intercambiarán los datos sonar a través del socket `Request-Reply` y harán ejecutar la función avoid con sus propios datos para evitar colisiones.

A modo de ejemplo, estos son los datos que se reciben por consola en el cliente 1:

```
Datos avoid cliente 1 ->cliente 2:

0.43 0.59 1.00 1.00 1.00 0.35 0.27 ... 0.50 0.40 0.38

Datos avoid cliente 1 ->cliente 2:

0.43 0.55 1.00 1.00 1.00 0.37 0.27 ... 0.55 0.42 0.39
```

Y estos los datos desde recibidos en el cliente 2:

```
Datos avoid cliente 2 ->cliente 1:

0.34 0.33 0.39 0.91 1.00 1.00 0.91 ... 1.00 1.00 1.00

Datos avoid cliente 2 ->cliente 1:

0.33 0.33 0.40 0.88 1.00 1.00 0.55 0.36 ... 1.00 1.00
```

4.4. Ejemplos Creación y Envío de Paquetes

Para ilustrar el proceso de creación y envío de paquetes en el protocolo de comunicación desarrollado, se presentan dos ejemplos. El primero inicializará la transmisión de datos del sonar del robot que se pide y el segundo le pide al servidor que cambie las velocidades de las ruedas en el robot.

4.4.1. Ejemplo 1: Inicialización del sonar

El siguiente fragmento de código muestra como un cliente puede solicitar el inicio de la transmisión de datos del sonar de un robot que se indica con el parámetro `robot_id`.

Listing 4.11: Inicialización del Sonar

```
zcom_packet start_feed_packet;
zcom_packet_factory(&start_feed_packet);
// Anade la acción al paquete después de asegurarnos de que
// está listo para usarse
start_feed_packet.action = ZCOM_ACTION_START_SONAR_FEED;
// Reserva memoria necesaria para la acción que quiere
// transmitir
zcom_initialize_parameters(&start_feed_packet);

// Mete el id del robot en el hueco del primer parámetro
uint8_t robot_id = 1;
zcom_set_parameter_with_type(&start_feed_packet, 0, &robot_id,
            sizeof(robot_id), ZCOM_TYPE_UINT8);

// Envía el paquete a través de su socket REQUEST
zcom_send_packet(req, &start_feed_packet);

// Espera de forma síncrona a la respuesta
zcom_packet response_packet;
zcom_packet_factory(&response_packet);
zcom_receive_packet(req, &response_packet);
```

4.4.2. Ejemplo 2: Cambio de velocidad en motores

El siguiente fragmento de código ilustra como un cliente avoid (4.3.1) le pide al servidor que cambie la velocidad de las ruedas izquierda y derecha de un robot P3DX.

Listing 4.12: Cambiar velocidas motores

```
// Se encarga de decidir la velocidad de los
// motores según los datos del sonar
avoid(arr, &lspeed, &rspeed);
zcom_packet set_motor_packet;
zcom_packet_factory(&set_motor_packet);
// Inicializa la acción
set_motor_packet.action = ZCOM_ACTION_SET_MOTOR;
zcom_initialize_parameters(&set_motor_packet);
// Mete 3 parámetros de distintos tipos en el paquete
zcom_set_parameter_with_type(&set_motor_packet, 0, &robot_id,
                sizeof(uint8_t), ZCOM_TYPE_UINT8);
zcom_set_parameter_with_type(&set_motor_packet, 1, &lspeed,
                sizeof(double), ZCOM_TYPE_DOUBLE);
zcom_set_parameter_with_type(&set_motor_packet, 2, &rspeed,
                sizeof(double), ZCOM_TYPE_DOUBLE);
// Serializa y despues envía el paquete
zcom_send_packet(req, &set_motor_packet);
```

Capítulo 5

Conclusiones y Trabajo Futuro

Este trabajo de fin de grado ha descrito y desarrollado un protocolo de comunicación eficiente y ligero, diseñado para facilitar la transmisión de datos a alta velocidad entre agentes en entornos multi-robot. Además, se ha desarrollado un servidor intermediario entre el simulador CoppeliaSim y varios clientes, demostrando que el protocolo cumple con los objetivos esperados.

El protocolo implementado en C y basado en ZeroMQ, permite una comunicación estructurada y de baja latencia, por lo que podría ser utilizado junto a otros simuladores mientras que el servidor intermediario sea modificado para ajustar las llamadas a la RemoteAPI de CoppeliaSim a unas de otro simulador. Por ende, se considera que el objetivo principal de este trabajo ha sido conseguido.

Trabajo Futuro

A pesar de que el protocolo de comunicación desarrollado ha demostrado ser funcional y eficiente en entornos simulados, hay distintas áreas que pueden ser abordadas en trabajos futuros.

En primer lugar, no se ha probado que la transmisión de datos entre el servidor y los distintos clientes sea estable en máquinas distintas, el proyecto ha sido desplegado dentro de un mismo ordenador, lo que provoca que las conexiones hayan sido siempre internas. Aunque se espera que el comportamiento sea el mismo, no se puede afirmar.

Ligado al primer punto, la seguridad a la hora de enviar y recibir paquetes es importante en caso de que no se quiera que alguna persona con malas intenciones genere paquetes falsos que distraigan al correcto funcionamiento de la simulación, además que si este protocolo fuera utilizado para controlar robots que utilicen datos sensibles, podría presentarse una grave vulnerabilidad. De modo que puede ser interesante implementar técnicas de cifrado y de autenticación para garantizar comunicaciones seguras.

Actualmente el servidor intermediario está diseñado para comunicarse con los clientes diseñados y para traducir algunas funciones adicionales de la RemoteAPI, pero como mejora se podría ampliar su compatibilidad con otros proyectos o

clientes si se añadieran todas las funciones ofrecidas por dicha API, permitiendo la creación de entornos más diversos desde un cliente.

Siguiendo con la necesidad de un mayor desarrollo, en un trabajo futuro se pueden implementar clientes que tengan funcionalidades más avanzadas, haciendo así un uso completo del servidor actual o del servidor planteado en el desarrollo futuro. Además, se podrían crear clientes que controlen otro tipo de agentes como brazos robóticos o robots que tengan sensores más avanzados.

Capítulo 6

Análisis de Impacto

Este trabajo de fin de grado influye sobre varios factores dentro del desarrollo sostenible que la agenda 2030 impulsa.

Después de haber indagado sobre las tecnologías de protocolos de comunicación y de robótica, se puede decir que este mundo sólo va a más y gracias a la accesibilidad que empresas como Coppelia Robotics, y a proyectos de código abierto en el que todo el mundo puede contribuir como ZeroMQ, cualquier estudiante tiene accesibles los recursos necesarios para aprender.

El diseño de un protocolo de comunicación ligero y eficiente favorece al desarrollo de tecnológias que tengan menos requisitos computacionales, lo que permite la reutilización de equipos electrónicos que puedan resultar obsoletos o dispositivos que tengan menor capacidad computacional o de memoria puedan resultar útiles con un uso en aplicaciones domésticas como sistemas automatizados para la gestión del hogar permitiendo la comunicación entre distintos agentes. La posibilidad de extender la vida útil de dispositivos contribuye a la reducción de residuos electrónicos y promueve un uso más sostenible de nuestros recursos tecnológicos.

Finalmente, se espera que los resultados obtenidos en este trabajo sirvan como base para futuras investigaciones y aplicaciones en el ámbito de la transmisión de mensajes y de la robótica, facilitando el desarrollo de sistemas más accesibles, sostenibles y eficientes, en beneficio de un progreso tecnológico responsable y equitativo afectando a los objetivos 7 (Energía asequible y no contaminante), 9 (Innovación e infraestructura), 12 (Producción y consumo resopnsable).

Bibliografía

[1] L. E. Parker, D. Rus y G. S. Sukhatme, *Multiple Mobile Robot Systems*, 2nd. Springer, Berlin, 2016, cap. 53, págs. 1335-1366.

[2] J. Gielis, A. Shankar y A. Prorok, «A Critical Review of Communications in Multi-Robot Systems», ago. de 2022.

[3] «Overcoming Wireless Communication Challenges in Robotics». (2024), dirección: https://www.zettascale.tech/news/overcoming-wireless-communication-challenges-in-robotics (visitado 16-11-2024).

[4] M. J. Matarić, «Issues and Approaches in the Design of Collective Autonomous Agents», *Robotics and Autonomous Systems*, vol. 16, págs. 321-331, 1995.

[5] C. Robotics. «CoppeliaSim User Manual». (2024), dirección: https://manual.coppeliarobotics.com/.

[6] «Robot Platforms and Simulators», págs. 131-137, 2022.

[7] F. Ferri, S. Cavalcanti y M. Louis. «ZeroMQ remote API». (2024), dirección: https://manual.coppeliarobotics.com/en/zmqRemoteApiOverview.htm.

[8] A. Technology. «Pioneer3-DX». (2011), dirección: https://www.generationrobots.com/media/Pioneer3DX-P3DX-RevA.pdf (visitado 28-11-2024).

[9] C. Team. «Adept's Pioneer 3-DX». (2021), dirección: https://www.cyberbotics.com/doc/guide/pioneer-3dx?version=R2021a (visitado 28-11-2024).

[10] P. Hintjens. «ZeroMQ - The Guide». (2024), dirección: https://zguide.zeromq.org/.

[11] C. Developers. «CZMQ - High-level C Binding for ZeroMQ». (2024), dirección: http://czmq.zeromq.org/.

[12] A. Sumaray y S. K. Makki, «A comparison of data serialization formats for optimal efficiency on a mobile platform», en *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ép. ICUIMC '12, Kuala Lumpur, Malaysia: Association for Computing Machinery, 2012, ISBN: 9781450311724. DOI: 10.1145/2184751.2184810. dirección: https://doi.org/10.1145/2184751.2184810.

[13] D. Song, K. Goldberg y N.-Y. Chong, *Networked Robots*, 2nd. Springer, Berlin, 2016, cap. 44, págs. 1109-1125.

[14] C. Müller, N. Kutzbach y A. Jurkat. «World Robotics Reports». (2024), dirección: https://ifr.org/worldrobotics/ (visitado 28-11-2024).

[15] D. Beres. «The Ethical Case For Killer Robots». (2016), dirección: https://www.huffpost.com/entry/lethal-autonomous-weapons-ronald-arkin_n_574ef3bbe4b0af73af95ea36.

[16] A. M. Tipanguano Astudillo, «Aplicación de un algoritmo basado en consenso para un sistema multi-agente robótico simulado en CoppeliaSim y comandado desde Matlab», oct. de 2022.

[17] R. S. Sutton y A. G. Barto, *The Reinforcement Learning Problem*, 2nd. The MIT Press, London, 2015, cap. 53, págs. 1-25.

[18] L. Zheng, Y. Yang y Q. Tian, «SIFT Meets CNN: A Decade Survey of Instance Retrieval», 2012.

[19] A. Krizhevsky, I. Sutskever y G. E. Hinton, «ImageNet Classification with Deep Convolutional Neural Networks», ago. de 2015.

[20] X. Chen y H. M. et. al, «Multi-View 3D Object Detection Network for Autonomous Driving», 2017.

[21] O. Robotics. «ROS 2 Documentation». (2024), dirección: https://docs.ros.org/en/humble/index.html (visitado 28-11-2024).

[22] P. Hintjens. «CZMQ Manual - CZMQ/3.0.1 - zsys». (2024), dirección: http://czmq.zeromq.org/manual:zsys (visitado 30-10-2024).

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Tue Jan 14 20:07:03 CET 2025
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)