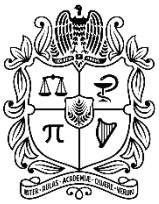


UNIVERSIDAD
NACIONAL
DE COLOMBIA

Arquitectura HW/SW para la aceleración de tareas de robots móviles mediante la integración de FPGA y ROS

Jairo David Cuero Ortega

Universidad Nacional de Colombia
Facultad de Ingeniería, Departamento de Ingeniería Eléctrica y Electrónica
Bogotá, Colombia
2022

Arquitectura HW/SW para la aceleración de tareas de robots móviles mediante la integración de FPGA y ROS

Jairo David Cuero Ortega

Tesis presentada como requisito parcial para optar al título de:

Magister en Ingeniería – Automatización Industrial

Director:

Ph.D. Pedro Fabián Cárdenas Herrera

Codirector:

Msc. Edwar Jacinto Gómez

Línea de Investigación:

Robótica

Universidad Nacional de Colombia

Facultad de Ingeniería, Departamento de Ingeniería Eléctrica y Electrónica

Bogotá, Colombia

2022

Has llegado a este mundo en un momento de convulsión para ser mi asidero y mi refugio, mi amparo y mi fortaleza, dándole vuelta a todo lo que conocía, creaste en mí miedos que no sabía. No obstante, también me hiciste conocer un amor distinto, por ahora silencioso, de solo sonrisas y miradas inocentes, ese amor que ya esperaba pero que no imaginaba, no imaginaba lo mucho que te amaría. Para ti Mariana.

Declaración de obra original

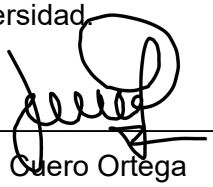
Yo declaro lo siguiente:

He leído el Acuerdo 035 de 2003 del Consejo Académico de la Universidad Nacional. «Reglamento sobre propiedad intelectual» y la Normatividad Nacional relacionada al respeto de los derechos de autor. Esta disertación representa mi trabajo original, excepto donde he reconocido las ideas, las palabras, o materiales de otros autores.

Cuando se han presentado ideas o palabras de otros autores en esta disertación, he realizado su respectivo reconocimiento aplicando correctamente los esquemas de citas y referencias bibliográficas en el estilo requerido.

He obtenido el permiso del autor o editor para incluir cualquier material con derechos de autor (por ejemplo, tablas, figuras, instrumentos de encuesta o grandes porciones de texto).

Por último, he sometido esta disertación a la herramienta de integridad académica, definida por la universidad.



Jairo David Cuero Ortega
Fecha 01/03/2022

Agradecimientos

A los directores de este trabajo, los ingenieros Pedro Fabian Cárdenas y Edwar Jacinto Gómez por el acompañamiento no solo académico sino profesional que ha contribuido en alto grado a mi crecimiento como persona y como ingeniero.

A Álvaro Andrés Montenegro Poches por el apoyo constante en la redacción y revisión de este documento, a Sebastián Cárdenas por su apoyo con los temas de teleinformática y a Santiago Penagos, que desde la distancia también ha sabido orientarme.

A Javier Andrés Vargas Guativa por haber sido un amigo, un guía y un ejemplo a seguir hasta el último momento en que la pandemia lo arrebatara de nuestras vidas. A él le debo gran parte de lo que soy; por sus enseñanzas, consejos y por impulsarme a ser cada vez mejor, estaré siempre agradecido.

Para mi esposa y mi familia, que han sabido apoyarme siempre todo mi amor y agradecimiento.

.

Resumen

Arquitectura HW/SW para la aceleración de tareas de robots móviles mediante la integración de FPGA y ROS

La complejidad inherente de los robots móviles ha requerido un trabajo interdisciplinario de varios campos de la ingeniería, como las tecnologías de la información y las comunicaciones, la inteligencia artificial y el desarrollo de software. A su vez, la comunidad de ingenieros y desarrolladores que utilizan ROS ha contribuido a que los robots sean mejores, más accesibles y estén presentes casi en cualquier tipo de aplicación. Cada vez más, los robots integran múltiples sensores y actuadores heterogéneos para realizar tareas más complejas que a menudo son computacionalmente intensivas o requieren procesamiento en tiempo real que los procesadores, sin importar cuántos núcleos tengan, no pueden cumplir. Para abordar este problema, se propone desarrollar, evaluar y demostrar en una plataforma robótica prototipo la integración de ROS con dispositivos SoC FPGA que combinan unidades de procesamiento y recursos de hardware reconfigurables en un chip. La plataforma robótica objetivo fue un robot móvil de tracción diferencial utilizado en educación, investigación y desarrollo llamado Turtlebot3 Burger. Este robot compatible con ROS incluye un sensor LiDAR y una placa Raspberry Pi como unidad de procesamiento. La Raspberry se reemplazó por el SoC FPGA Ultra96v2 para implementar una rutina de software para crear el mapa de ocupación a partir de los datos del sensor LiDAR. Se midió el tiempo de ejecución del algoritmo y luego se desarrolló un diseño de hardware para reemplazar parte de la rutina en el software. La partición y la asignación de tareas al hardware programable mejoraron el rendimiento al acelerar hasta diez veces la construcción del mapa de ocupación en el entorno del robot.

Palabras clave: ROS, FPGA, Robot móvil, Particionamiento de tareas, Codiseño HW/SW.

Abstract

HW/SW architecture for mobile robot task acceleration by integrating FPGA and ROS

The inherent complexity of mobile robots has required interdisciplinary work from several engineering fields, such as information and communications technology, artificial intelligence, and software development. In turn, the community of engineers and developers using ROS has contributed to making robots better, more accessible, and ubiquitous in almost any type of application. Robots increasingly integrate multiple heterogeneous sensors and actuators to perform more complex tasks that are often computationally intensive or require real-time processing that CPUs, no matter how many cores they have, cannot fulfill. To address this problem, we propose developing, evaluating, and demonstrating on a prototype robotic platform the integration of ROS with SoC FPGA devices that combine both processing units and reconfigurable hardware resources on a chip. The target robotic platform was a differential drive mobile robot used in education, research, and development called Turtlebot3 Burger. This ROS-enabled robot includes a LiDAR sensor and a Raspberry Pi board as a processing unit. A SoC FPGA Ultra96v2 replaced the Raspberry to implement a software routine to create the occupancy map from the LiDAR sensor data. Then, we measured the algorithm execution time and developed a hardware design to replace part of the routine in software. The partitioning and the task assignment to the programmable hardware improved the performance by speeding up the construction of the occupancy map in the robot environment by up to ten times.

Keywords: ROS, FPGA, Mobile Robot, Task Partitioning, HW/SW Codesign,

Contenido

Lista de figuras

Pág.

Lista de tablas

Pág.

Introducción

El diseño concurrente de componentes de hardware y software en el desarrollo de un dispositivo electrónico se conoce como codiseño. A través de esta sinergia de hardware/software (HW/SW), se optimizan y satisfacen las restricciones de diseño, como los costos, el rendimiento y el consumo de energía. El codiseño se centra principalmente en el diseño de sistemas en un chip (SoC) o el diseño de sistemas integrados que implican la integración de microprocesadores de propósito general, estructuras DSP y lógica programable (FPGA), núcleos ASIC, periféricos de bloque de memoria e interconexión de buses en un chip (Teich, 2012).

En la última década, los procedimientos de codiseño se han integrado en muchos dispositivos, especialmente en aquellos que realizan tareas complejas y múltiples como los robots móviles. Aunque esta integración fue ideada hace más de treinta años (Smith et al., 1985),), fue necesario el avance en el desarrollo de microprocesadores y el aumento de su poder computacional, para que se dieran los primeros intentos de integrar completamente las capacidades de HW y SW como el diseño de una arquitectura para la programación y particionamiento de tareas en sistemas embebidos en tiempo real (Azzedine et al., 2002) y una metodología de codiseño para aplicaciones de visión artificial (Albaladejo et al., 2004). Este último imaginó cómo los entonces nuevos FPGA mejorarían la metodología de codiseño al implementar las particiones de software y hardware de las tareas en el mismo sistema programable de campo en chip (FPSoC).

Asimismo, el inicio del siglo XXI marcó importantes avances en el campo aeroespacial motivados por el deseo de comprender el universo circundante, especialmente el planeta Marte. En 2003, llegaron a explorar dicho planeta dos robots Rover, el Spirit y el Opportunity, sucesores del Sojouner el cual en 1997 se convirtió en el primer vehículo con ruedas en Marte que, aunque perdió la comunicación a los pocos meses, supuso un hito en la exploración del planeta rojo. Una vez que los robots móviles llegaron a Marte,

surgieron desafíos debido al entorno hostil, impulsando el desarrollo de diversas técnicas para resolver las condiciones hostiles del planeta rojo utilizando elementos como FPGA y técnicas como el codiseño (Lentaris et al., 2016; Marosy et al., 2009), especialmente para la aceleración de algoritmos para visión por computador (Matthies et al., 2007) y navegación autónoma (Kostavelis et al., 2014).

El lanzamiento del primer sistema operativo robótico (ROS) en el 2010, permitió e hizo necesaria la implementación de técnicas de codiseño en aplicaciones robóticas avanzadas. ROS es un compendio de herramientas de software libre que ayuda a la creación de aplicaciones robóticas (Quigley et al., 2009) que van desde controladores primarios hasta algoritmos de última generación. Su uso generalizado y sus ventajas en comparación con otros middlewares han convertido a ROS en la plataforma de facto para estudiantes, aficionados y empresas que desarrollan aplicaciones robóticas.

ROS ha contribuido al avance de la robótica al ofrecer a los desarrolladores diversas herramientas base para poder crear prototipos de manera rápida y así concentrar los esfuerzos en el desarrollo de habilidades más complejas para el robot como la inteligencia artificial y la navegación autónoma. Estas nuevas tareas demandan bastante energía y representan un alto consumo de recursos computacionales, por ello se han explorado diversas técnicas con diferentes dispositivos para lograr el rendimiento deseado. Las ya conocidas ventajas que ofrecen las FPGA la convierten en una opción para superar los problemas de consumo de energía y de capacidad de procesamiento, sin embargo, no se puede dejar de lado las ventajas de ROS, así que se han hecho esfuerzos para lograr la integración de las FPGA junto con las herramientas de ROS.

Dentro de los primeros esfuerzos para lograr la integración de ROS con FPGA se cuenta con lo realizado por Yamashina et al., (2015) quién usó una metodología de desarrollo basado en componentes, que en este caso consistió en diseñar un nodo hardware en la FPGA con capacidad de suscripción y publicación de mensajes para que ROS pudiera reconocerlo e interactuar con él nodo de hardware como si fuera uno en software. Este prototipo de componente hardware compatible con ROS se desarrolló en una arquitectura Zynq, que posee tanto un sistema de Procesador ARM como una parte de lógica programable de FPGA en un solo chip.

Los mismos autores desarrollaron una mejora al prototipo inicial con la herramienta de software denominada CReComp (Yamashina et al., 2016) que genera automáticamente el código para la integración de la parte de hardware con la de software, lo que se traduce en una optimización del tiempo para crear componentes FPGA compatibles con ROS. Igualmente se aceleró el proceso de transmisión de mensajes de publicación y suscripción al separar la estructura de comunicación en registro (XMLRPC) y comunicación de datos (TCPROS).

A partir de las anteriores ideas, este proyecto plantea la integración de ROS con una tarjeta SoC-FPGA que gobierna una plataforma robótica con el fin de reducir el tiempo de ejecución de tareas específicas en el robot. Para ello se selecciona el TurtleBot3, un robot móvil de tracción diferencial que ofrece distintas ventajas para el desarrollo de prototipado y que posee una base con dos motores Dynamixel y una rueda libre tipo ball caster que le da estabilidad, una Raspberry Pi 3B+ en la cual se encuentra instalado ROS, un sensor LiDAR y una tarjeta OPENCR 1.0 para la navegación y la odometría. El TurtleBot3 permite realizar distintas tareas propias de un robot móvil, como navegación autónoma, construcción de mapa, detección de obstáculos, entre otros. Sin embargo, al poseer ROS las restricciones se presentan más en el sistema de hardware que en el de software, puesto que conforme aumenta la cantidad de tareas del robot, aumenta la demanda de energía y el consumo computacional del procesador. Para esto se propone reemplazar la tarjeta Raspberry Pi por un sistema en chip (SoC) que integre tanto la parte del procesador como la de lógica programable de tal manera que se puedan particionar las tareas y asignarlas al procesador o a la FPGA. Las rutinas que se implementan en hardware en un problema de codiseño actúan como aceleradores debido a su capacidad de operar de manera concurrente, es decir, en paralelo.

Este documento está estructurado en cuatro capítulos, en el primero de ellos se hizo la revisión de las tareas de navegación más comunes que desarrolla un robot móvil de tracción diferencial. Para ello se dividió la navegación en percepción, localización, planificación del movimiento y seguimiento de la trayectoria. En esta etapa se identificaron varias tareas que son críticas por su nivel de complejidad y/o por la cantidad de recursos que demandan del robot, de estas se eligió implementar la creación del mapa de ocupación del robot a partir de los datos del LiDAR con el fin de probar la arquitectura HW/SW planteada.

El segundo capítulo comprende la definición de los elementos que componen la arquitectura, como la plataforma robótica, la versión de ROS, las tarjetas de desarrollo que integran un SoC y las opciones de sistema operativo para integrar en el robot. Con respecto a la parte física, la evaluación de las características permitió seleccionar la plataforma de TurtleBot3 y la tarjeta de AVNET Ultra96V2. Para la sección de los elementos de software se presentan las herramientas que ofrece el fabricante Xilinx como vitis, vivado y petalinux. Las dos primeras son de diseño y la última permite la personalización del sistema operativo de la tarjeta. Finalmente, el capítulo presenta el framework de PYNQ como alternativa para optimizar el tiempo de diseño y despliegue de aplicaciones de codiseño.

La integración de los componentes se presenta en el capítulo tres, el cual se subdivide en hardware, software y red. En las modificaciones físicas al TurtleBot3 se cuentan las adecuaciones mecánicas básicas para que la Ultra96v2 ocupara el espacio de la Raspberry Pi, la sección de software comprende el proceso de integración de la tarjeta de desarrollo con el sistema operativo y el middleware Ros. Finalmente se presenta la estructura de la red que se configuró para la comunicación entre el robot y el pc desarrollador.

Con la plataforma de hardware lista, en el capítulo cuatro se desarrolla la implementación de la aplicación robótica: la construcción de un mapa de ocupación con los datos tomados por el LiDAR. La función para construir el mapa se desarrolló en el Framework PYNQ, inicialmente solo en lenguaje de programación Python y luego se describió de manera comportamental, utilizando el lenguaje de alto nivel HLS en Vitis y se integró el diseño con Vivado. Ambas implementaciones se integran en Jupyter Notebook con una clase principal que permite seleccionar la función en software para que la ejecute el procesador o la función descrita en hardware que se ejecuta en la FPGA. En el capítulo también se presenta la comparación de ambas implementaciones.

1. Navegación en robots móviles

La navegación en los robots móviles consiste en una serie de tareas cuya finalidad es guiar la locomoción del robot dentro de un ambiente con o sin obstáculos. La primera tarea de navegación que un robot móvil realiza es percepción del entorno, la cual por medio de sensores permite al robot crear una abstracción del mundo (Choset, 2005). Tradicionalmente se ha considerado la navegación como la respuesta a las siguientes preguntas ¿dónde estoy?, ¿dónde están los otros lugares relacionados conmigo? y ¿cómo llego a otros lugares desde aquí?

Al responder estas preguntas se determina la localización del robot, posteriormente se planifica la ruta para determinar cómo llegar a otros lugares desde donde se encuentra el robot, para finalmente realizar el seguimiento de la trayectoria hasta alcanzar el objetivo. Este enfoque ha gobernado el paradigma de la navegación del robot proponiéndose muchas soluciones exitosas, sin embargo, los avances técnicos en la robótica móvil han ampliado el problema de la navegación hacia las siguientes preguntas ¿cómo es este lugar? ¿Cuál es la forma del entorno en el que me encuentro? (Barber et al., 2018).

1.1. Percepción

La navegación autónoma de un robot móvil implica la realización de movimientos de manera segura conociendo completamente el entorno del robot, estos elementos definen las capacidades de acción e interacción entre el robot, el ambiente, los obstáculos, e incluso los humanos. Las tareas realizadas por los robots móviles conllevan a que estos se encuentren en constante movimiento, así que su entorno presenta constantes cambios y es necesario detectar e identificar elementos, objetos o escenas del ambiente. La percepción se define como el entendimiento del ambiente basado en mediciones necesarias para que el sistema responda de manera inteligente a lo que se encuentra a su alrededor (Kelly, 2013).

La información del entorno se adquiere a partir de la medición de varios sensores que pueden ser clasificados en exteroceptivos y propioceptivos (Klančar et al., 2017). La Tabla 1-1 muestra algunos de los sensores con mayor uso comercial en la robótica móvil.

Tabla 1-1: Clasificación de sensores en robot móviles.

Propioceptivos	Exteroceptivos
Sensor de batería	Sonar
Sensor de corriente de motor	LiDAR
Sensor de temperatura	Sensores de visión
Encoder absoluto e incremental	Sensores de distancia infrarrojo
IMU (giroscopio, acelerómetro)	Sensores de distancia ultrasónicos

1.1.1. Sensores propioceptivos

Los sensores propioceptivos brindan información de cuál es el estado interno del robot, permitiendo conocer cómo se encuentra en cada momento y la implementación de lazos de control. El monitoreo de esta clase de sensores puede indicar cuando es tiempo de recargar las baterías, cuando el motor se está sobrecalentando o cuando un componente interno del robot no está funcionando correctamente (Corrochano, 2020).

- **Detección del nivel de batería:** En la robótica móvil el sensado del nivel de voltaje de la batería es indispensable, ya que esto determina el tiempo de autonomía del robot y en qué momento este debe retornar a la estación de recarga, o por el contrario disminuir las operaciones que generan un alto consumo de energía.

- **Detección de la corriente de parada:** Este sensor se encarga de medir la corriente de los motores y determinar si el robot móvil se encuentra atascado en un obstáculo en el caso de que los demás sensores fallen en la detección de una colisión. Cuando el robot se encuentra obstruido por un obstáculo las llantas dejarán de girar mientras que la corriente en los motores crecerá a su máximo valor. De esta forma, la medición de la corriente de los motores servirá como detector de colisiones, como último recurso.

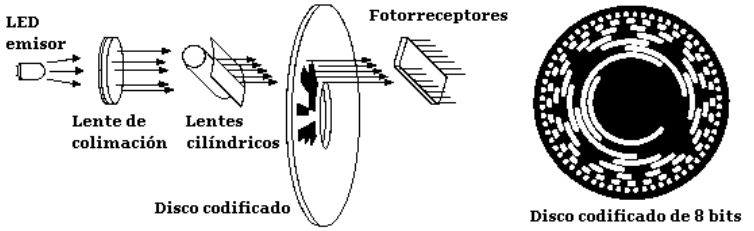
- **Temperatura:** La gran mayoría de robots móviles monitorean las temperaturas internas del robot para evitar que los circuitos electrónicos de potencia se calienten en

exceso y ocasionen daños a la tarjeta de desarrollo, la batería del robot, o impacten en la vida útil de los motores.

- **Sensores de posicionamiento:** Acá se incluyen los que proporcionan la orientación del robot y su posición. Existen de *medida absoluta* los cuales proporcionan la pose (posición y orientación) con respecto a un marco fijo del entorno, aunque presentan errores en la medición estos no se acumulan con el paso del tiempo. Hacen parte de este subgrupo el GPS, la brújula, el encoder absoluto y demás sensores que detectan las marcas que se hacen en el ambiente y cuya localización se sabe con precisión (Klančar et al., 2017). Los sensores de *medida incremental* dan la medida a partir de incrementos del movimiento del robot con respecto a un punto, son más precisos que los de medida absoluta, pero los errores de la medición se acumulan. Los sensores más comunes de este subgrupo son el encoder incremental, el giroscopio y el acelerómetro.

- **Encoder absoluto:** Este sensor provee un valor de posición único en cada punto de rotación a través de la lectura de un disco codificado por sectores, a cada sector le corresponde un código binario como el código gray. Su resolución es fija ya que su codificación es absoluta, de allí su nombre. La Figura 1-1 muestra los componentes de un encoder absoluto tipo óptico donde cada valor binario se representa con zonas transparentes u opacas dispuestas en forma radial; también existen sensores tipo magnéticos y capacitivos.

Figura 1-1: Encoder absoluto.



Fuente: Tomado de https://www.ingmecafenix.com/automatizacion/encoder/.

- **Encoder incremental:** Consiste normalmente en dos canales que proveen señales cuadradas desplazadas 90 grados a medida que el eje gira. A través de la cuenta de los pulsos de cualquiera de los canales se obtiene la velocidad de rotación y la posición

relativa, y con la lectura del desfase se encuentra el sentido de giro. Los encoder incrementales son menos complejos que los absolutos y normalmente también son menos costosos, pueden ser de tipo óptico o de efecto hall.

- **GPS (Global Positioning System):** El GPS no es necesariamente un sensor propioceptivo debido a que recibe de satélites externos la señal para funcionar, sin embargo, al brindar una estimación de la posición del robot puede ser considerado como uno. Su gran ventaja es que provee la posición absoluta del robot en casi cualquier entorno en el que se encuentre, a excepción de la planta baja de un edificio; su principal desventaja es que el valor de la posición tiene baja precisión.

- **Unidad de medida inercial (giroscopio y acelerómetro):** Son sensores compuestos por brújulas de medida incremental que detectan variaciones en la orientación del robot a partir de mediciones de su aceleración. Su principal limitación es que acumula error con el paso del tiempo, por lo cual suele ser implementado el giroscopio en conjunto con el acelerómetro.

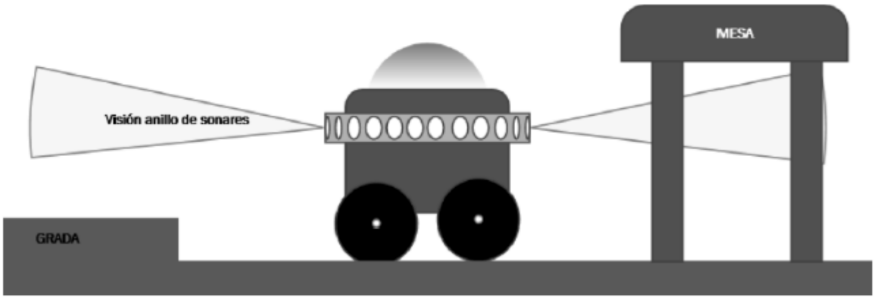
1.1.2. Sensores exteroceptivos

Los sensores exteroceptivos proporcionan información del exterior que rodea al robot, estos se dividen en sensores de proximidad y de visión. Aunque ambos cumplen la función de detección de obstáculos en la locomoción del robot móvil, los sensores de visión permiten además, reconocer objetos, personas y lugares (Corrochano, 2020).

- **Sonar:** El sonar es un sensor de ultrasonido que emite una señal de sonido en un rango de frecuencias de 50kHz a 250kHz, las cuales se encuentran por fuera de las frecuencias audibles por el ser humano. Es utilizado para calcular la distancia midiendo el tiempo de vuelo desde que la señal acústica es emitida hasta que rebota en un obstáculo y el eco vuelve al sensor. (Corrochano, 2020).

En la Figura 1-2 se muestra un robot móvil cuya circunferencia está cubierta por sensores ultrasónicos, en una configuración típica que utiliza 24 sensores donde cada uno mapea 15 grados, debido al estrecho cono de los sensores. Sin embargo, el problema más recurrente con este sistema de sensores son la reflexión y la interferencia.

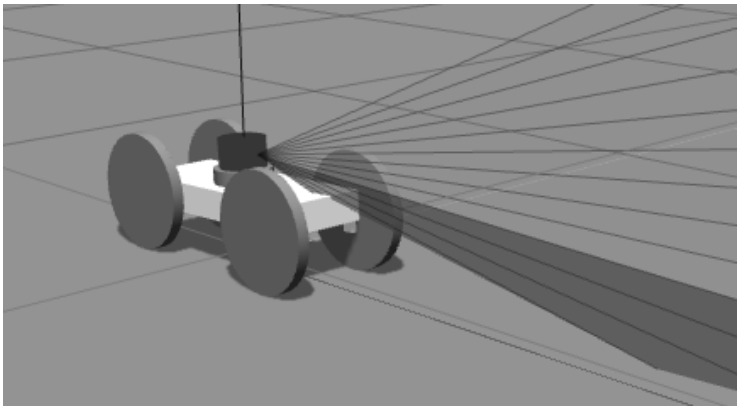
Figura 1-2: Aplicación de un sistema de sensores sonar en un robot móvil.



Fuente: Tomado de Corrochano, (2020).

- **LiDAR (Light Detection and Ranging):** El LiDAR es uno de los sensores más gutilizados en la robótica móvil ya que presenta mejoras significativas con respecto a los sonares, que funcionan con ultrasonido, y los radares que lo hacen con señales de radio. Este sensor consta de un transmisor que utiliza un láser para iluminar el objetivo u obstáculo con un haz de luz, cuyos rayos son paralelos entre sí, y un receptor que detecta un componente de luz que es coaxial con respecto al haz emitido (Li & Shi, 2019), como ilustra la Figura 1-3.

Figura 1-3: Sensor LiDAR en un robot móvil.



Fuente: Tomado de https://www.cplusgears.com/lesson-5-adding-a-LiDAR.html

Los sensores LiDAR realizan una estimación de la distancia con base en el tiempo que el haz de luz laser emplea para llegar al obstáculo y regresar al receptor, un arreglo mecánico junto a un espejo barre el haz de luz para abarcar el objeto requerido en dos dimensiones, o a través de un espejo giratorio genera una escena en 3D. Las desventajas del LiDAR

consisten en el costo elevado, el considerable consumo de energía, la técnica de detección y la mecánica (Li & Shi, 2019).

- **Kinect:** Es un sensor fabricado por Microsoft para la consola de juegos Xbox 360, el cual consta de una cámara RGB, un sensor de profundidad y un conjunto de giroscopios que proporcionan la orientación. El campo de visión va hasta los 45° verticalmente y 57° de manera horizontal y la distancia de trabajo se extiende desde los 1.2m hasta los 3.5m (Ruiz-Sarmiento et al., 2011). Inicialmente se desarrolló para su uso en videojuegos pero luego se adaptó para labores de desarrollo e investigación en el área de la robótica como el mapeo, la navegación, la visión artificial, la detección y reconocimiento de objetos, personas y lugares.

1.1.3. Evolución

La evolución de la percepción en robótica móvil se dio conforme se hizo necesaria una mayor autonomía en los mismos, puesto que inicialmente su actividad estaba limitada a un ambiente estático (robots industriales) que no contaban con un reconocimiento de su ambiente (Sosa Valverde, 2020). Tras la inclusión de sensores que permiten el conocimiento del entorno (radares, sonares, infrarrojos, visión) se desarrollaron y fabricaron desde robots de seguridad que patrullan áreas determinadas, robots enfermeros que distribuyen medicamentos y verifican el estado de los signos vitales de los pacientes, hasta robots para investigación espacial y actividades militares, dotados de cámaras, sensores de radiación, presión, temperatura, humedad y de contacto, entre muchos más (Colomer, 2018).

1.2. Localización

Para desarrollar una navegación exitosa, el robot móvil autónomo debe pasar a través de diferentes fases como la percepción, que como se explicó anteriormente, consiste en extraer información relevante de los sensores con los que cuenta el robot. Posteriormente, en la fase de localización el robot estima su posición y orientación actual con respecto a su entorno usando los datos de los sensores exteroceptivos con los que cuente (Panigrahi & Bisoy, 2021). Para cualquier tarea que desee desarrollar un robot móvil autónomo se debe conocer con cierto nivel de precisión su posición y orientación dentro de un marco de referencia global.

1.2.1. Problemas en la localización

Basándose en la posición inicial del robot, se pueden presentar tres tipos de problemas en la localización: seguimiento de la posición (position tracking), localización global (global localization) y la desaparición del robot (kidnapped robot). En el primer problema la posición inicial del robot es conocida y el objetivo es seguir al robot en cada instante de tiempo durante la navegación en su entorno, durante la navegación el algoritmo de localización utiliza la posición previa del robot para actualizar la posición actual (Ko & Kuc, 2015). Para lograr este objetivo, se aplica una técnica donde se fusiona la información del encoder y de los sensores exteroceptivos (Panigrahi & Bisoy, 2021).

En el problema de la localización global, el robot no tiene ninguna información sobre su posición inicial, esto significa que el robot debe ser capaz de localizarse a sí mismo de manera global dentro su entorno. En algunas situaciones, el robot es llevado a una posición arbitraria que no corresponde con la posición inicial del robot de la que se tenía información o en otros casos durante el seguimiento de la posición es llevado arbitrariamente a una posición indefinida, a dicho problema se le conoce como desaparición del robot (Marín Paniagua, 2014). De acuerdo con la revisión bibliográfica realizada en el artículo científico de (Panigrahi & Bisoy, 2021), el porcentaje en que se presenta cada problema de localización se visualiza en la Figura 1-4.

Figura 1-4: Problemas presentados en la localización de robots móviles.

Fuente: Tomado de Panigrahi & Bisoy (2021).

1.2.2. Estimación local

El método de localización menos complejo es el de localización local, el cual requiere conocer la posición y orientación inicial del robot móvil, al iniciar la navegación se estima la pose actual con la información de los sensores propioceptivos, es decir la información interna, o local del robot, a esto también se le conoce como odometría. La odometría calcula la ubicación del robot en un plano a partir de la velocidad de los ejes del robot junto con la velocidad angular. Este procedimiento tiene como ventaja bajos tiempos de respuesta y baja demanda de procesamiento, mientras que su principal inconveniente es que el error se acumula con el tiempo y la estimación de la pose puede diferir del valor real luego de recorrer una distancia considerable (Marín Paniagua, 2014).

1.2.3. Estimación global

La localización global comprende las técnicas para estimar la posición del robot dentro de un marco de referencia global, de ellas se destaca el uso de GPS que provee información de la longitud, latitud y altitud del robot móvil. Con esta técnica se reduce el error al estimar la pose del robot ya que no depende del tiempo ni requiere de valores iniciales de la posición y orientación (Espinoza & Zuñiga, 2021). No obstante, esta técnica presenta algunas desventajas relacionadas con una baja tasa de muestreo y la imposibilidad de utilizar la técnica en ambientes cerrados o grandes edificaciones que obstruyen las señales de GPS.

1.2.4. Estimación probabilística

Las técnicas probabilísticas utilizan datos de mediciones previas de la posición y orientación inicial del robot y/o del conocimiento del entorno, lo cual disminuye el error que se presenta en los métodos de estimación anteriores. El método más usado es el filtro de Kalman, que consiste en una sumatoria ponderada de las mediciones que hacen distintos sensores. (Espinoza & Zuñiga, 2021).

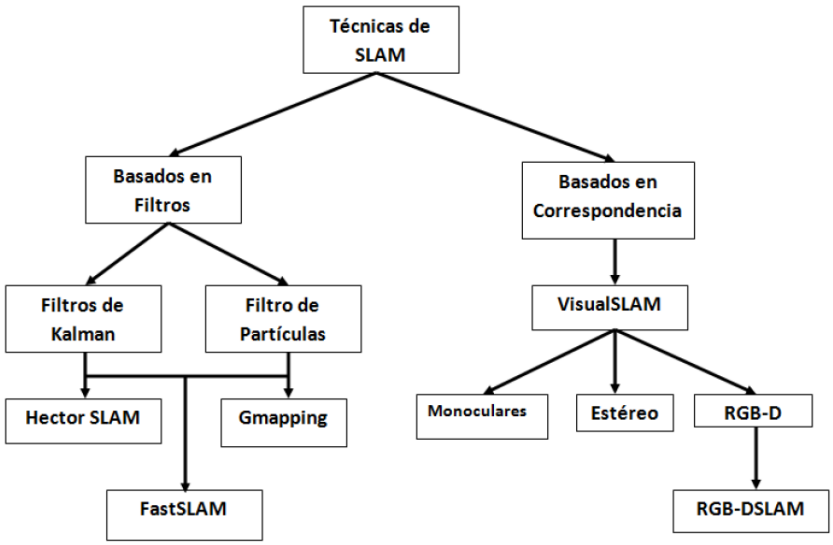
1.2.5. SLAM (Simultaneous Localization and Mapping)

Los robots móviles realizan una primera estimación de su pose a través de los sensores propioceptivos, no obstante, existen errores que afectan la precisión y exactitud de dicha estimación que hace necesario el uso de sensores exteroceptivos para obtener mayor

información del ambiente que les rodea y así contrastar y rectificar la primera estimación realizada (Gómez, 2015). Mediante la localización y mapeo simultáneo – SLAM, se construye una representación del entorno del robot móvil y a través de odometría se estima su posición y orientación en cada instante de su recorrido, es una técnica que se caracteriza por su amplio uso para el desarrollo de robots y vehículos autónomos (Velásquez, 2017).

Aunque el SLAM se basa en métodos probabilísticos, las técnicas pueden clasificarse en dos grupos: basados en filtros y basados en correspondencias. La Figura 1-5 muestra un mapa conceptual de las técnicas de SLAM y sus correspondientes algoritmos.

Figura 1-5: Técnicas de SLAM



Fuente: Tomado de Velásquez, (2017).

1.3. Planificación y seguimiento de la trayectoria

La navegación de un robot consiste en partir desde un punto inicial y pasar por posiciones sucesivas intermedias que lo conducen hasta una posición final (Wahab et al., 2020). Existen muchos métodos para la planificación de trayectorias, los cuales dependen del enfoque desde el cual se aborde el problema. Por ejemplo, en la robótica la planificación de trayectorias aborda el movimiento del robot desde un punto a otro encontrando la ruta más corta y óptima. En la inteligencia artificial, hace referencia a la búsqueda de una secuencia de acciones lógicas que transforman una posición inicial del robot en una

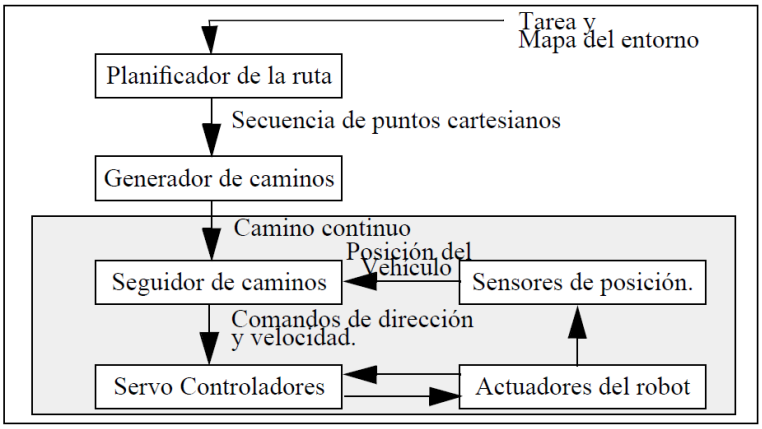
posición final deseada. Mientras que en la teoría de control, la planificación de trayectorias se ocupa de problemas como la estabilidad del sistema, la retroalimentación y la optimización del algoritmo de control (Acosta, 2018).

Una trayectoria óptima podría ser una ruta en la que se disminuyan la cantidad de giros y/o frenadas, para ello es indispensable contar con un mapa del entorno que se obtiene en la fase de percepción y conocer la posición y orientación del robot con respecto al mapa, las cuales se obtienen en la fase de localización.

1.3.1. El problema de la planificación de trayectoria

En la Figura 1-6 se plantea un esquema de la navegación de un robot móvil, el cual parte desde la generación del mapa del entorno y las particularidades de la tarea de navegación. A partir de esta información se planea un conjunto de submetas que se presentan como un grupo de coordenadas dispersas que definen la trayectoria. El robot adicionalmente debe asegurarse que la ruta asociada se encuentre libre de obstáculos. La planificación de la ruta o trayectoria proporciona las acciones para el control de la dirección y la velocidad de los motores del vehículo (Barber et al., 2018).

Figura 1-6: Estructura básica de navegación con planificador de trayectoria.



Fuente: Tomado de Wahab et al., (2020).

De forma general, se puede resumir el problema de la planificación del movimiento en robots móviles como la determinación de una trayectoria admisible que permite la evasión de los obstáculos y se ajusta a las restricciones cinemáticas del robot (Castellet, 2018).

1.3.2. Tipos de entorno y representaciones de mapa

Los entornos en los que se generan las trayectorias pueden ser estáticos o dinámicos. Un entorno estático es aquel que no se mueve y del que se tiene total conocimiento del espacio de trabajo, mientras que el entorno dinámico es aquel en que los obstáculos se mueven y por lo tanto no se conoce completamente el espacio de trabajo. Cuando el ambiente es estático se realiza planificación asíncrona, mientras que se utiliza una planificación síncrona para los entornos dinámicos ya que el espacio de trabajo cambia continuamente y lleva al algoritmo a generar nuevas rutas de planificación (Acosta, 2018).

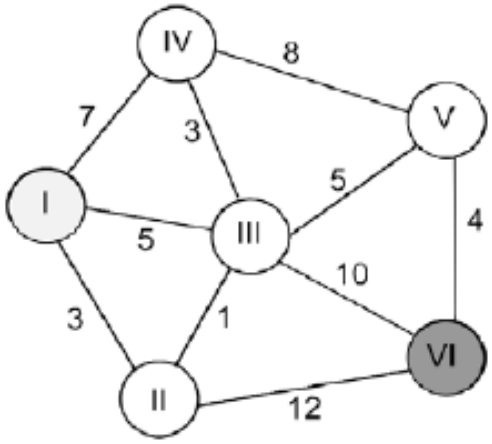
Para la planificación asíncrona se requiere de toda la información del entorno del robot, localizando con exactitud los obstáculos, realizando la planificación de la trayectoria una única vez. Para la planificación síncrona la información del entorno es incompleta, por lo que la planeación de la trayectoria es un proceso continuo, las acciones de control se generan a partir de la información del ambiente entregada por los sensores exteroceptivos, ya que no es necesario conocer la localización y forma de los obstáculos (Acosta, 2018).
Para planificar una trayectoria es necesario que se represente el ambiente por medio de mapas, para los cuales existen dos enfoques complementarios: mapas con aproximaciones discretas y mapas con aproximaciones continuas. En la aproximación discreta el mapa se subdivide en partes de igual o diferente tamaño, donde cada parte representa un vértice o nodo, que están conectados por los bordes permitiendo al robot móvil navegar por los vértices. En la aproximación continua se requiere la definición de límites interiores que representan los obstáculos y limites externos en formas de polígono, mientras que la trayectoria puede ser codificada como una secuencia de puntos cartesianos, a pesar de que utiliza menos recursos de memoria en la robótica móvil predominan los mapas discretos (Wahab et al., 2020).

1.3.3. Algoritmos geométricos

Los algoritmos geométricos buscan encontrar una trayectoria más corta de un vértice a otro a través de un gráfico conectado, la trayectoria más corta se obtiene cuando el costo de borde acumulativo es mínimo, en aplicaciones de robótica móvil indica la menor distancia entre nodos o submetas de la trayectoria. En la Figura 1-7 se muestra un ejemplo de planificación de trayectoria donde los nodos están identificados por los números romanos del I al VI, el punto inicial del robot móvil es el nodo I mientras que la meta está

identificada por el nodo VI, los números que aparecen en los vértices corresponde a la distancia que existe entre un nodo a otro.

Figura 1-7: Ejemplo de planificación de trayectoria.



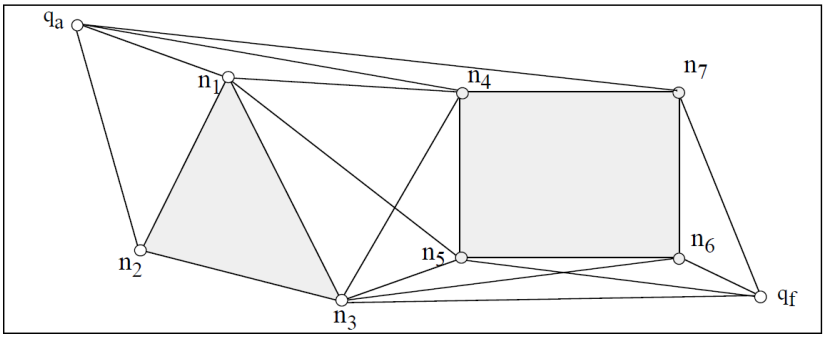
Fuente: Tomado de Acosta, (2018).

Existen muchas trayectorias que puede tomar el robot móvil, no obstante, la planificación de la trayectoria deseada es la que representa un menor costo de borde acumulativo, es decir que represente una menor distancia entre el punto inicial y la meta. Por ejemplo, las trayectorias I-II-VI y la I-III-VI tienen una distancia de 15 cada una, mientras que la trayectoria I-II-III-V-VI aunque tiene mayor número de submetas representa el camino más corto ya que tiene un costo de borde acumulativo de 13.

- **Grafos de Visibilidad:** Aborda de manera geométrica el problema de planificación de la ruta ya que consiste en un mapa en dos dimensiones donde los obstáculos se representan mediante polígonos; el mapa se genera con base en el concepto de visibilidad que permite clasificar como *visibles entre sí* a dos nodos si se pueden unir por una línea recta, sin intersecar ningún obstáculo. (Asqui, 2017).

De acuerdo con el grafo mostrado en la Figura 1-8 los nodos de visibilidad son el punto inicial (q_a), el punto final (q_f) y los puntos del n_1 al n_7 corresponden a los vértices de los obstáculos del entorno, la trayectoria resulta de unir con segmentos rectilíneos todos los nodos que son visibles, de esta manera se elige la trayectoria que une el nodo inicial y final minimizando el coste de borde acumulativo.

Figura 1-8: Grafo de visibilidad con dos obstáculos.



Fuente: Tomado de Asqui, (2017).

La versión presentada funciona para entornos totalmente conocidos, sin embargo, al igual que en aviación se puede realizar la planificación de la ruta a medida que se desplaza el robot aplicando la técnica LNAV (Lateral Navigation). Este algoritmo elige la trayectoria más cercana entre los nodos visibles, según la distancia euclídea entre ellos, a través de ella, se sitúa en el nodo seleccionado, lo marca como visitado y desde ahí se repite el proceso con los nodos que son visibles en la nueva posición, hasta llegar a la posición final.

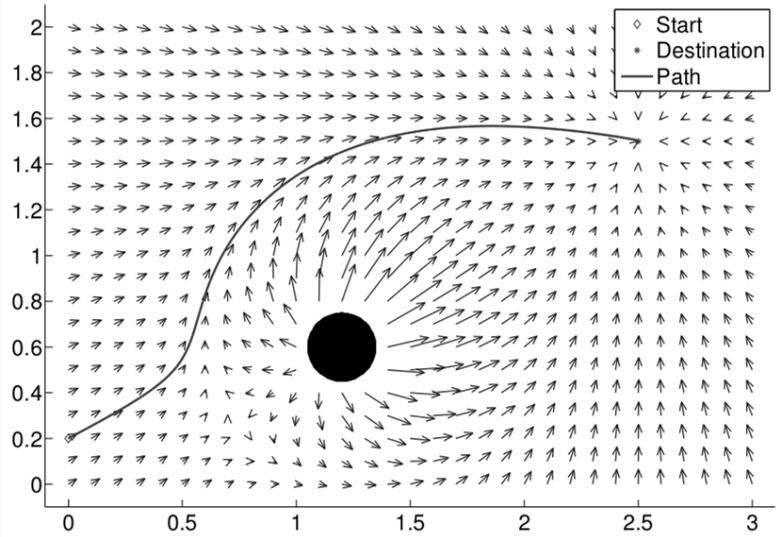
Aunque el algoritmo de grafos de visibilidad es muy implementado gracias a su bajo costo computacional, presenta un problema al considerar las dimensiones del robot, ya que este no es un punto en el espacio y al considerar el radio del robot se incrementa las dimensiones de los obstáculos y se disminuye la cantidad de trayectorias que permitan al robot navegar del nodo inicial al nodo final (Asqui, 2017).

- **Diagramas de Voronoi:** El algoritmo de diagramas de Voronoi ubica la trayectoria lo más apartada posible de los obstáculos para que el robot no colisione con ellos. Para ellos, se divide el espacio en zonas iguales llamadas regiones de Voronoi, las cuales están compuestas por los puntos más cercanos al punto p_i, si se consideran estos puntos como la posición de los obstáculos en el entorno, las líneas del diagrama son las trayectorias más seguras que debe seguir el robot móvil ya que son equidistantes de todos los obstáculos. Los vértices equidistan de dos o más obstáculos, en la planificación de la trayectoria estos vértices representan las submetas de la ruta (Asqui, 2017).

- **Descomposición de celdas:** La técnica de descomposición de celdas divide el espacio de configuraciones en regiones a las que etiqueta según contengan o no un obstáculo. Este método permite la solución de dos problemas, por un lado resuelve las trayectorias por zonas, y a su vez, encuentra secuencias de celdas vecinas libres de colisión, uniendo dos puntos cualesquiera en su interior para asegurar un camino razonable (Asqui, 2017).

- **Campos potenciales:** El enfoque del algoritmo de campos potenciales difiere bastante de los anteriores, ya que está basado en técnicas reactivas de navegación, útiles en la planificación local para entornos no conocidos o la evasión de obstáculos en tiempo real. La teoría de campos potenciales concibe al robot como una partícula inmersa en un campo potencial artificial generado por los obstáculos, los cuales empujan al robot hacia la posición objetivo (Castellet, 2018). La generación de trayectorias por el algoritmo de campos potencial atrae al robot hacia la posición final, mientras que los obstáculos repelen al robot. En este orden de ideas, en la navegación por campos potenciales se calcula el efecto del campo potencial en el robot, se determina el vector de fuerza artificial que actúa sobre el robot y finalmente se genera las ordenes de movimiento del robot. La Figura 1-9 presenta la simulación de una trayectoria con este método.

Figura 1-9: Simulación de trayectoria generada por algoritmo de potencial de campo.



Fuente: Tomado de Barber et al., (2018).

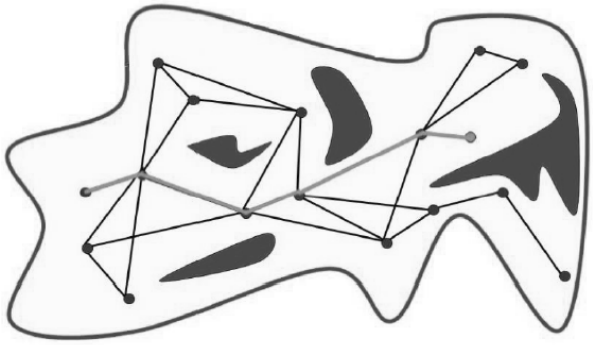
1.3.4. Algoritmos probabilísticos

Los algoritmos probabilísticos presentan una ventaja con respecto a los algoritmos geométricos ya que la implementación es menos compleja y pueden adaptarse a entornos dinámicos.

- **Mapas probabilísticos (PRMs):** Estos pueden ser multiconsulta o de consulta única, la principal característica de este algoritmo consiste en el cálculo eficiente de trayectorias para robots móviles con varios grados de libertad. Básicamente, el algoritmo de los mapas probabilísticos captura aleatoriamente muestras del entorno del robot, verifica si se encuentran en el espacio libre y luego conecta estas configuraciones con las configuraciones cercanas, posteriormente se agregan la información de posición y orientación inicial y final para determinar una ruta entre dichos puntos (Castellet, 2018).

En los mapas probabilísticos de consulta múltiple pueden coexistir varios grafos y la planificación de la trayectoria se realiza en dos etapas, la de construcción y la de consulta. En la primera etapa se construyen trayectorias interconectando los nodos que representan los movimientos que puede hacer el robot en el entorno. En la etapa de consulta se conectan al gráfico las configuraciones de inicio y final, y la trayectoria se consigue al evaluar la ruta más corta (Castellet, 2018), como se muestra en la Figura 1-10.

Figura 1-10. Mapa probabilístico de consulta múltiple.



Fuente: Tomado de Castellet, (2018).

Por su parte, los mapas probabilísticos de consulta única producen un solo grafo al limitar la búsqueda a las configuraciones accesibles. También existen derivaciones del algoritmo que construyen dos árboles al mismo tiempo cuyas raíces están en la configuración de

inicio o final y en donde el crecimiento de la trayectoria se detiene cuando se conectan estos dos árboles (Castellet, 2018). En el primer caso, cuando se genera un árbol único la búsqueda es unidireccional, mientras que cuando se generan dos árboles se realiza una búsqueda bidireccional.

- **RRT (Rapidly Exploring Random Tree):** Este algoritmo genera un árbol que se origina en la configuración inicial y que mediante el uso de muestras aleatorias del espacio de búsqueda determina nuevos nodos que se añaden al árbol siempre y cuando sea factible la conexión hasta ese nodo, es decir esté libre de obstáculos y restricciones (Asqui, 2017).

1.4. Tareas críticas

Esta sección presenta las tareas críticas en el desarrollo de robots móviles autónomos, entendiendo las tareas críticas como aquellas que generan un mayor costo computacional, tiempo de procesamiento y mayor consumo de energía en el hardware de la plataforma robótica. Para ello se tiene en cuenta trabajos recientes en aspectos de navegación como la construcción del mapa, localización, SLAM y el seguimiento de la trayectoria.

1.4.1. Mapeo

El mapeo consiste en crear un mapa, modelo o representación del entorno del robot móvil a partir de los datos obtenidos por los sensores propioceptivos y exteroceptivos de la plataforma robótica. Dichos modelos pueden ser utilizados secuencialmente por otras tareas, como la localización, la planeación y el seguimiento de la trayectoria. El mapeo se puede realizar para un entorno estructurado o uno totalmente desconocido, convirtiendo dicha información en una cuadrícula de ocupación, la cual es un arreglo de datos binarios, que asigna un “0” para las áreas libres de obstáculos y un “1” para para las áreas ocupadas con obstáculos o para áreas que no se mapearon debido a que no está en el rango de profundidad del sensor (Kundu et al., 2016).

La calidad del mapa de ocupación es clave para la navegación, para evitar que el robot se aproxime a áreas cercanas a la colisión con algún obstáculo del entorno es necesario mapear las fronteras de estos. Para ello es importante tener en cuenta las dimensiones físicas del robot en la elección de la resolución y tamaño de las celdas de ocupación del

mapa. Aunque este sea realizado a baja resolución, requiere el procesamiento de datos del sensor en cada pose del robot en la navegación, por tanto representa un alto costo computacional y se recomienda calcular únicamente para una serie de poses de los obstáculos más cercanos al robot (Vallvé & Andrade-Cetto, 2015).

En Thrun, (2003) se proponen dos marcos para el mapeo: los mapas métricos, los cuales representan el entorno con precisión métrica; y los mapas topológicos, los cuales describen el ambiente como grafos donde los puntos son representados con nodos y los lazos representan las posibles trayectorias del robot móvil. Para el primer tipo de mapa se requiere un gran costo computacional para la lectura de los sensores del robot, mientras que para los mapas topológicos se requiere un alto consumo de memoria para almacenar la información de la pose del robot para cada nodo de la representación (Cebollada et al., 2021).

1.4.2. Localización

La localización es la tarea que intenta estimar la posición y orientación actual, es decir, la pose del robot en su entorno, para llevar a cabo esto el modelo del entorno debe estar disponible antes de que inicie la tarea de localización. El sensor LiDAR es frecuentemente usado para automatizar el proceso de posición y orientación del robot, este sensor funciona independientemente de la locomoción del robot, por lo tanto, tiene la capacidad de mapear el ambiente mientras el robot está en movimiento.

La tarea se vuelve critica ya que el LiDAR presenta una resolución de 1° grado, lo cual implica que por cada revolución este sensor entregará un vector de 360 datos por cada instante de tiempo, lo cual indica que mientras el robot esté en movimiento se tendrá una matriz de 360xN datos, dependiendo del tiempo de navegación del robot (Gul et al., 2019), lo cual implica un alto costo computacional y consumo de memoria, sumado al consumo de energía adicional del sensor.

Autores como (Kendall et al., 2015; Neto, 2015) proponen algoritmos de localización a partir de sensores de visión los cuales realizan la tarea en tiempo real a través de redes neuronales de convolución con cámara monocular y RGB, las cuales aunque presentan mayor precisión que el LiDAR ya que este por cada instante de tiempo presenta un vector

de 360 datos, mientras que las cámaras de visión presentan matrices de información cuyo tamaño depende de la resolución del sensor, lo cual significa un aumento considerable en el tiempo de procesamiento y los costos computacionales.

1.4.3. SLAM

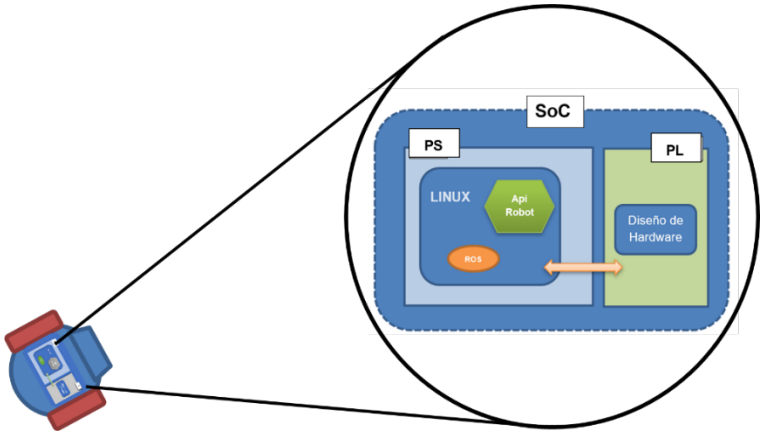
La tarea SLAM realiza de forma simultánea mapeo y localización del robot móvil, es una técnica basada en métodos probabilísticos que aunque permite la reducción del almacenamiento de memoria y cierto costo computacional al no tener que operar directamente las matrices de datos con información proveniente de los sensores del robot; se vuelve crítica al tener que validar la información de la pose inicial del robot, con el modelo cinemático y dinámico del robot y la información del exterior que proporcionan los sensores exteroceptivos como el LiDAR o demás sensores basados en visión (Gómez, 2015).

Los algoritmos como el filtro extendido de Kalman es una tarea crítica en robótica móvil debido a que los modelos de movimiento del robot y la percepción de obstáculos del entorno son ecuaciones no lineales; lo cual aumenta la complejidad computacional ya que la matriz de covarianza tiene que ser actualizada (Qin et al., 2020). La técnica Karto SLAM, por otro lado, permite el mapeo y localización del robot en áreas extensas, ya que por su representación en grafos, permite almacenar un nuevo nodo cuando la información de la pose del robot ha sido verificada y es consistente con el entorno; sin embargo, esta misma característica conlleva a un elevado consumo de memoria para almacenar la información de los nodos, y altos costos computacionales para procesar dichos datos vectoriales (Jaramillo, 2020).

2. Arquitectura HW/SW

Un robot es una máquina que está integrada por tres sistemas: sistema electrónico, sistema mecánico (hardware) y sistema de procesamiento de información (software). Para el desarrollo de un robot móvil se requiere integrar estos tres sistemas para tener la capacidad de navegación, interacción con el entorno y autonomía, a esta integración se le conoce como la arquitectura Hardware – Software (HW/SW) del robot (Florez, 2009). El diseño de software normalmente está compuesto por varias capas como la de aplicación, controladores, comunicación entre otras, y se ejecuta en uno o varios de los procesadores con los que cuenta el sistema; en contraste, el diseño de hardware se mapea y depende de los recursos disponibles del dispositivo. La Figura 2-1 ilustra la idea principal de la arquitectura HW/SW propuesta con el fin de acelerar las tareas de navegación en los Robots móviles. Esta consta de una estructura robótica gobernada por un SoC con un procesador ARM y un bloque de lógica programable (FPGA) que permite la aplicación de técnicas de codiseño con la implementación de funciones en hardware, de tal manera que se pueden particionar las tareas y asignarles su ejecución en el bloque de software o en el de hardware según la complejidad y/o necesidad del diseño.

Figura 2-1: Arquitectura Hardware - Software HW/SW



Fuente: El autor

A continuación, se presentan las diferentes plataformas robóticas que se evaluaron, la comparación entre las tarjetas de desarrollo disponibles, las distintas versiones del middleware de robótica ROS y las consideraciones para la construcción del sistema operativo que gobierna la tarjeta.

2.1. Plataformas robóticas para desarrollo e investigación

Una plataforma robótica brinda una colección de herramientas, librerías y convenciones para simplificar el desarrollo del software para la realización de tareas complejas por parte del robot móvil. Es importante considerar que las tareas que son triviales para el ser humano, se vuelven extremadamente complicadas para que un robot genérico las resuelva, en ese orden de ideas las plataformas robóticas integran de forma sencilla los módulos de un sistema robótico, para la realización de tareas complejas (Tsardoulias & Mitkas, 2017).

Para la selección de la plataforma robótica se definieron los siguientes requerimientos de hardware y software para delimitar la búsqueda.

- Arquitectura de hardware con suficiente soporte y pruebas de funcionamiento.
- Incluir componentes de hardware esenciales para la locomoción.
- Tener buen desempeño mecánico y bajo consumo de energía.
- Admitir la integración del hardware con ROS.
- Permitir el acceso a la información de odometría.
- Contar con superficies de montaje de componentes periféricos.
- Estar equipado con sensores de visión y/o LiDAR.

Preferiblemente, la plataforma robótica debe ser una de las más comerciales en el campo de investigación y desarrollo, ya que eso garantiza encontrar la suficiente información en foros especializados y contar con el soporte del fabricante para hardware y software. Además, permite el fácil mantenimiento y reemplazo de sus partes, un buen desempeño mecánico en diferentes entornos y la inclusión de herramientas, librerías y paquetes de software integradas con ROS. El cuarto requerimiento asegura que el fabricante haya realizado las suficientes pruebas durante el desarrollo de la plataforma robótica, por lo cual es indispensable tener acceso a la información de odometría para estimar la posición del

robot móvil durante la navegación. Finalmente, contar con las superficies para hacer montaje de componentes periféricos permite equipar al robot con sensores que minimizan las fuentes de error para el procesamiento de la información.

2.1.1. Plataformas consideradas

Los criterios técnicos establecidos anteriormente permitieron acotar el análisis de las plataformas robóticas dedicadas a actividades de desarrollo e investigación, aunque muchas plataformas cumplen uno o más requerimientos necesarios para este trabajo, en la Tabla 2-1 se presentan las cinco opciones más destacadas para ser consideradas.

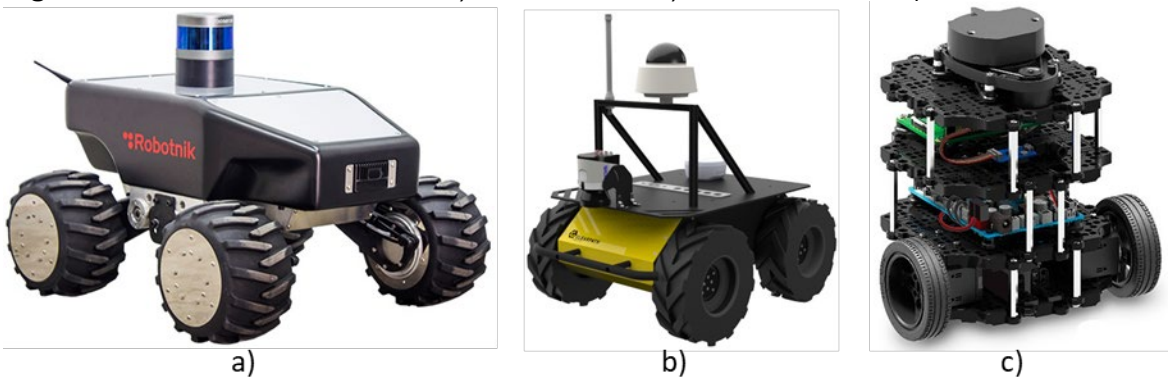
Tabla 2-1: Plataformas robóticas consideradas.

Plataforma	Husky A200 UGV	iRobot Create2	Summit - XL	Parallax Arlo	TurtleBot 3 Burger
Dimensiones	990mm x 670mm x 390mm	340mm (diámetro) x 92mm	720mm x 614mm x 416mm	450mm (diámetro) x 92mm	138mm x 178mm x 192mm
Controlador del motor	Construido por el fabricante	Construido por el fabricante	Servomotores sin escobillas, 4 x 500 W	Parallax DHB - 10	Dynamixel (XL430-W250-T)
Batería	24V a 5000mAh	14.4V a 3800mAh	48V a 15000mAh	12V a 7000mAh	11.8V a 1800mAh
Sistema de tracción	Sistema diferencial	Sistema diferencial	Sistema diferencial	Sistema diferencial	Sistema diferencial
Módulos de control	Velocidad cinemática, de rueda, control de voltaje	Velocidad cinemática y de rueda	Velocidad cinemática y de rueda	Velocidad cinemática y de rueda	Velocidad cinemática, de rueda, control de voltaje
Integración con ROS	ROS Melodic, ROS Kinetic.	ROS Kinetic	ROS Melodic, ROS Kinetic	ROS Kinetic	ROS Melodic, ROS Kinetic
Odometría	Encoder magnético	No	Encoder magnético	Encoder óptico	Encoder magnético
Sensores	LiDAR	Ultrasónico (proximidad)	LiDAR	Ultrasónico (proximidad)	LiDAR

Cada una de las plataformas anteriores se encuentra integrada con ROS y cuenta con módulos de control de velocidad cinemática y velocidad de cada una de las ruedas. Los

robots móviles Husky A200, Summit XL y el TurtleBot3 Burger (Figura 2-2) destacan por los sensores que integran para la navegación como lo es el LiDAR.

Figura 2-2: Plataformas móviles: a) SUMMIT-XL. b) HUSKY A2000. c) TurtleBot3.



a) b) c)

Fuente: Páginas web oficiales de los fabricantes

El robot Summit XL fue desarrollado por la empresa ROBOTNIK, cuenta con dos posibles cinemáticas: omnidireccional o convencional, cuenta con 4 servomotores de alta potencia y una odometría basada en encoder magnéticos para cada llanta y un sensor angular de alta precisión montado dentro del chasis del robot (*SUMMIT-XL Mobile Robot - Indoor & Outdoor | Robotnik®*, n.d.).

La plataforma robótica Husky A200 es desarrollada por CLEARPATH ROBOTICS, su cinemática es de tracción diferencial y cuenta con una estructura física muy robusta lo que le permite integrar sensores como cámaras estéreo, GPS, LiDAR y una gran variedad de manipuladores, cuenta con dos motores de alta potencia desarrollados por el fabricante (*Husky UGV - Outdoor Field Research Robot by Clearpath*, n.d.).

El TurtleBot3 Burger hace parte de la serie de robots TurtleBot3, su estructura física está diseñada a partir de plataformas modulares que permite a los desarrolladores modificar su forma teniendo disponible los tipos Burger, Waffle y Waffle Pi, los cuales constan de dos motores Dynamixel XL430-W250-T, una batería de 11.8v a 1800mAh lo que asegura su bajo consumo de energía, además de contar con la integración de un sensor LiDAR y un procesador Raspberry Pi 3 en su sistema de control (, n.d.).

La plataforma robótica más familiarizada con ROS es el TurtleBot3 Burger, a tal punto que el icono identificador del robot es utilizado como símbolo de ROS. Dicha plataforma tiene

la ventaja de contar con una gran cantidad de documentación y una comunidad de soporte de ROS.

2.1.2. TurtleBot

TurtleBot es una plataforma robótica personal, de bajo costo y código abierto basada en ROS, se deriva del robot Turtle que fue impulsado por el lenguaje educativo de programación LOGO en 1967. Originalmente, fue diseñado para facilitar el aprendizaje de quienes eran nuevos en la programación en ROS y LOGO. Desde que TurtleBot es la plataforma estándar de ROS se convirtió en la plataforma robótica más popular entre desarrolladores, investigadores y estudiantes (, n.d.).

- **TurtleBot1:** Creado por Melonee Wise y Tully Foote en el laboratorio de desarrollo tecnológico Willow Garage en el año 2010 y lanzado al público a partir del año siguiente. Está equipado con una batería de 3000mAh, dos motores Dynamixel, un computador portátil ASUS 1215N con un procesador de dos núcleos, sensores Kinect y un giroscopio, aunque su hardware permite adicionar otros sensores y actuadores (, n.d.).

- **TurtleBot2:** Lanzado en octubre de 2012 por la compañía Yujin Robot basado en la estructura física del robot iClebo Kobuki. Está equipado con una batería de 2200mAh, dos motores Dynamixel, se lanzaron tres versiones del TurtleBot2, la original contaba con un procesador de dos núcleos del portátil ASUS 1215N, mientras que la versión TurtleBot2e reemplaza el portátil por una tarjeta de computadora DB410c, finalmente, la versión TurtleBot2i extiende sus características con una estructura modular con la integración de un brazo robótico Pincher MK3 de cuatro grados de libertad. Los tres modelos cuentan con sensores Kinect y giroscopio, con la posibilidad de adicionar otros sensores y actuadores (, n.d.).

- **TurtleBot3:** Desarrollado por la compañía ROBOTIS fue lanzado en mayo del año 2017, con la meta de reducir considerablemente el tamaño y costo de la plataforma sin sacrificar su funcionalidad y calidad, de hecho, mejora algunas características de funcionamiento de sus predecesores. Su estructura se basa en superficies modulares que permiten al usuario modificar su forma y funcionalidad. La versión Burger está equipada con una batería de 1800mAh, dos servomotores Dynamixel XL-430-W350T, un procesador

integrado en la tarjeta Raspberry Pi 3, y un sensor LiDAR lo cual le permite ejecutar algoritmos de localización y mapeo simultáneo (SLAM), planeación de trayectoria y navegación autónoma. Las versiones Waffle y Waffle Pi están equipados con un procesador Intel Joule y con sensores laser Intel RealSense, además cuenta con una mayor superficie para integrar periféricos incluyendo la posibilidad de implementar un brazo robótico para tareas de manipulación de objetos (, n.d.).

2.2. Tarjetas de desarrollo

Los microcontroladores fueron los primeros dispositivos utilizados para el control de robots móviles, sin embargo, cuando estos mejoraron sus características y dispusieron de mayor cantidad de sensores para interactuar con su entorno, la velocidad y eficiencia de los microcontroladores fueron opacadas por carencias tales como el poco nivel de procesamiento, la ejecución de tareas de forma secuencial y su baja capacidad de almacenamiento. Posteriormente se recurrió a las GPU (Graphics Processing Unit) que aunque suplían las falencias de los microcontroladores, generaron nuevas desventajas como su gran tamaño físico, sus altos costos de adquisición y un alto consumo energético, deficiencias que se vuelven más marcadas en las aplicaciones de robótica móvil (Biookaghazadeh et al., 2018).

Las FPGA (Field Programmable Gate Array) son dispositivos basados en una matriz de bloques lógicos configurables (CLB) que se conectan a través de interconexiones programables. A diferencia de las CPU's y GPU's, la FPGA puede ser considerada como hardware programable que puede ser configurada para crear buses de instrucciones específicas para cada problema a resolver (Purwanto et al., 2017). En robótica móvil las FPGA ofrecen un excelente rendimiento ante las altas demandas de computación en paralelo, procesamiento digital de imágenes, bajos consumos energéticos requeridos y la posibilidad de reconfiguración dinámica para obtener paralelismo espacial o temporal en su lógica de programación.

Lo anterior, permite la adquisición de altas tasas de datos de los sensores y procesar tareas de forma paralela, además que su tamaño compacto y bajo consumo de energía las hacen adecuadas para su implementación en robots móviles (Gu et al., 2016). Los diseños de FPGA comúnmente se implementan en un lenguaje de descripción de hardware de bajo

nivel llamado VHDL, lenguaje de programación en paralelo que se distingue de lenguajes procedimentales como C que se ejecutan secuencialmente (Schenck et al., 2017), lo cual genera dificultades para los programadores y ha significado un impedimento para el desarrollo de algoritmos en robótica móvil.

Para solucionar este problema las FPGA en la actualidad integran sistemas SoC (System on Chip) que combinan diferentes componentes y funcionalidades dentro de un solo chip. Estos sistemas, además de la flexibilidad que brinda contar también con bloque procesador ARM, ofrecen un alto rendimiento y presentan un consumo de energía menor pues pueden procesar las operaciones con alta carga computacional en el hardware FPGA; su mayor desventaja es que los desarrolladores requieren un profundo conocimiento tanto del hardware como del software (Nitta et al., 2019).

2.2.1. Comparación entre tarjetas de desarrollo

Para solucionar la dificultad el desarrollo con el lenguaje de programación VHDL, las FPGA más recientes que han salido al mercado cuentan con soporte de Python para hardware (Bingo, 2018), además, gracias al SoC permiten implementar servidores integrados basados en ROS (Romanov et al., 2019). La Tabla 2-2 muestra la comparación de tres tarjetas de desarrollo que cumplen con los requerimientos explicados anteriormente.

Tabla 2-2: Tarjetas de desarrollo consideradas

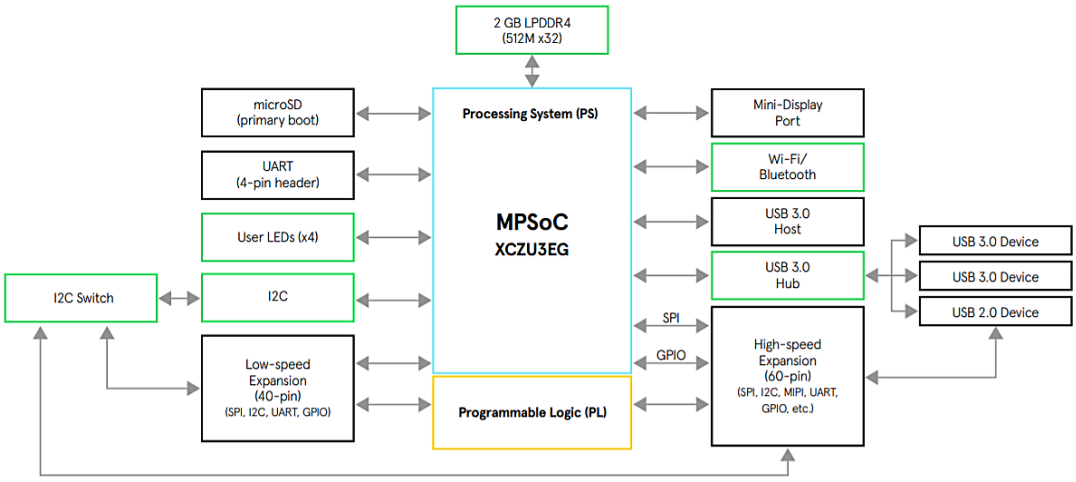
Tarjeta	PYNQ Z1	ZYBO Z7-10	ULTRA96v2
Alimentación	12V a 3A	5V a 2.5A	12V a 3A
Ethernet	Gigabit Ethernet	Gigabit Ethernet	Gigabit Ethernet
WiFi	NO	NO	SI
USB	1 USB 2.0, 2 USB OTG	1 USB 2.0, 1 USB OTG	1 USB 2.0, 3 USB 3.0
UART	1 controlador dedicado vía PL+GPIO	2 controladores dedicados vía PL+GPIO	2 controladores dedicados vía PL+GPIO
I2C	2 controlador dedicado vía PL+GPIO	2 controladores dedicados vía PL+GPIO	2 controladores dedicados vía PL+GPIO
GPIO	1 puerto PMOD	5 puertos PMOD	5 puertos PMOD
Procesador	650MHz de dos núcleos ARM Cortex-A9, 32 bits	667MHz de dos núcleos ARM Cortex-A9, 32 bits	1.5GHz de cuatro núcleos ARM Cortex-A53 MPCore, 64 bits
RAM	512MB DDR3	1GB DDR3	2GB DDR4
FPGA	Xilinx 7-series Artix	Xilinx XC7Z010-1CLG400C	Xilinx Zynq UltraScale+ MPSoC ZU3EG A484

La Digilent PYNQ-Z1 es una tarjeta de desarrollo FPGA SoC de propósito general para la programación de sistemas embebidos utilizando Python, su estructura de código abierto permite explorar las capacidades del SoC Xilinx sin tener que diseñar los circuitos lógicos

programables, los cuales son importados como librerías de hardware y son programados a través de sus API's (Application Programming Interfaces); cuenta con un procesador de dos núcleos ARM Cortex A9 de 32 bits con una frecuencia de operación de 650MHz, además de tener una RAM DDR3 de 512MB. La tarjeta ZYBO Z7-10 cuenta con los mismos propósitos y características que la PYNQ-Z1 diferenciándose por la capacidad de la tarjeta RAM DDR3 de 1GB y la frecuencia de 667MHz del procesador.

Por último, la tarjeta ULTRA96V2 fabricada por Avnet es una placa de desarrollo Xilinx de la familia Zynq que cuenta con arquitectura UltraScale+ MPSoC. El SoC posee un procesador (APU) de cuatro núcleos ARM Cortex A53 de 64 bits, un procesador de tiempo real (RPU) de dos núcleos Arm Cortex-R5F, una unidad de procesamiento gráfico y memoria RAM DDR4 de 2GB. Con respecto a los recursos de hardware de la parte de lógica programable, el SoC integra en un mismo chip una FPGA con 8820 bloques lógicos configurables (CLB) y hasta 360 segmentos DSP. Todas estas características son deseables para la integración hardware - software propuesta en el presente trabajo. Además, como se muestra en la Figura 2-3, la tarjeta Ultra96v2 cuenta con módulo bluetooth y WiFi que facilita la comunicación con otros dispositivos y el acceso inalámbrico a internet, también posee un puerto Mini Display para un monitor HDMI.

Figura 2-3: Diagrama de bloques de la tarjeta Ultra96V2.



Fuente: Ultra96-V2 Hardware User's Guide (https://www.avnet.com/wps/portal/us/).

Finalmente, la arquitectura de 64 bits cuenta con nivel de soporte uno para la instalación de Ros a diferencia del nivel tres de soporte para la arquitectura de 32 bits.

2.3. Middleware ROS

El middleware es un software que interactúa entre una aplicación para que establezca comunicación con programas, redes, sistemas operativos o hardware. En robótica, su fin es simplificar el trabajo de los programadores para generar conexiones y sincronizaciones necesarios entre el software y el hardware, además de encargarse de tareas de gestión de datos, de aplicaciones, servicios de mensajería, entre otros (Tsardoulias & Mitkas, 2017). Aunque un número considerable de sistemas robotizados no utilizan ningún tipo de middleware, existe una gran variedad de middleware orientados a aplicaciones de robótica, en la Tabla 2-3 se presenta una comparación de algunos de los más utilizados en desarrollo e investigación.

Tabla 2-3: Comparación de Middleware utilizados en plataformas robóticas

Middleware	Sistema operativo	Lenguaje de programación	Código abierto	Arquitectura distribuida	Interfaz de hardware	Algoritmos robóticos	Simulación
ROS	Unix	C++, Python, Lisp	**SI**	**SI**	**SI**	**SI**	**SI**
HOP	Unix, Windows	Scheme, JavaScript	**SI**	**SI**	**NO**	**NO**	**NO**
GAZEBO	Linux, Solaris, BSD	C++, Java, Python	**SI**	**NO**	**SI**	**SI**	**SI**
MSRS	Windows	C#	**NO**	**SI**	**NO**	**NO**	**SI**
OPRoS	Linux, Windows	C++	**SI**	**SI**	**SI**	**SI**	**SI**
MIRO	Linux	C++	**SI**	**SI**	**SI**	**NO**	**NO**
PYRO	Linux, Windows, OS/W	Python	**SI**	**NO**	**SI**	**SI**	**SI**
OROCOS	Linux, OS/W	C++	**SI**	**SI**	**SI**	**SI**	**NO**
RT Middleware	Linux, Windows, QNX	C++, Java, Python	**SI**	**SI**	**SI**	**NO**	**NO**
TeamBots	Linux, Windows	Java	**SI**	**NO**	**SI**	**SI**	**SI**

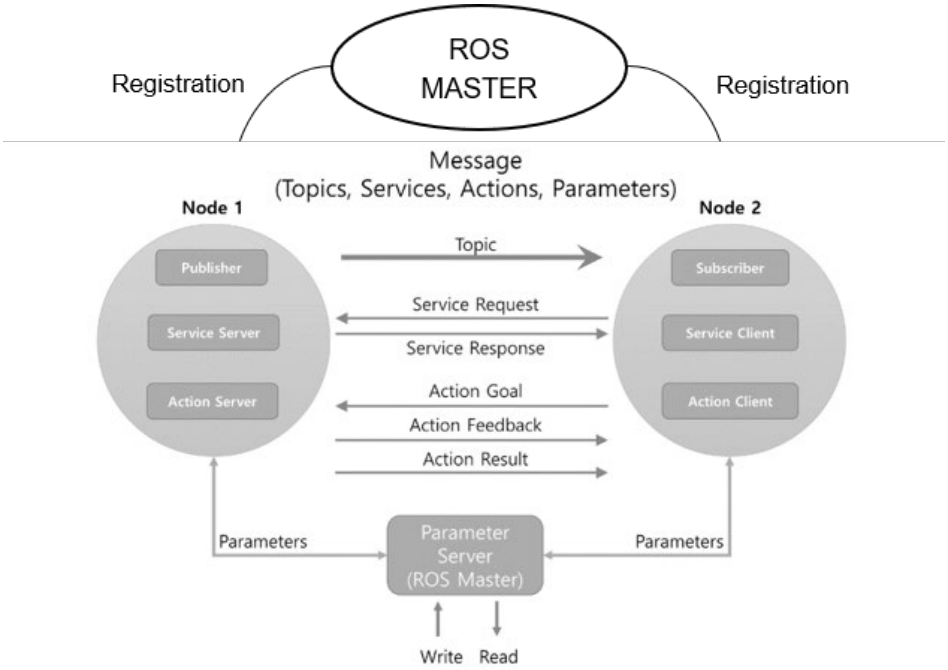
2.3.1. ROS (Robot Operating System)

ROS es un middleware orientado para la programación de robots, presentado en el año 2009 por medio del artículo *"ROS: an open-source Robot Operating System"* (Quigley et al., 2009). Su desarrollo fue impulsado por criterios de diseño que no tenían otros middlewares de la época, como la habilitación de topologías P2P (peer to peer) en las que los nodos se comportan como cliente y servidor simultáneamente, desarrollo en múltiples lenguajes de programación, separación de operadores complejos y librerías de algoritmos, herramientas especializadas para determinados ambientes de desarrollo y el hecho de ser un middleware de código abierto y libre de restricciones.

La topología P2P de ROS evita el problema de tráfico innecesario sobre las conexiones de red proporcionadas por el módulo maestro que maneja la búsqueda de servicios para procedimientos que requieren comunicación con otros servicios. Al ser multi lenguaje asegura que los programadores puedan utilizar el lenguaje que prefieran aprovechando los beneficios de las librerías de ROS. Finalmente, una característica importante es que los programadores pueden participar de una gran comunidad de desarrolladores para realizar revisiones y aportes en todos los niveles del software (Joseph & Whitaker, 2019).

- **Conceptos:** Las aplicaciones de ROS son modeladas como redes de nodos, los cuales son procesos que están a cargo de tareas particulares, como controlar actuadores y ejecutar algoritmos de navegación, procesamiento de imágenes, publicar y/o leer datos de los sensores. Estos nodos se comunican a través de canales unidireccionales asíncronos llamados tópicos. Siguiendo una arquitectura de publicación/suscripción como se muestra en la Figura 2-4, los nodos publican mensajes a través de los tópicos y se suscriben a los que contengan la información que ellos necesitan (Estefo et al., 2019).

Figura 2-4: Arquitectura de ROS



Fuente: Adaptado de *Distributions - ROS Wiki*, n.d.

Cualquier mensaje puede ser publicado en un tópico en particular para proporcionar un fácil manejo por parte de los nodos publicadores y los nodos suscriptores interesados en

dicha información. Las operaciones bidireccionales requieren tener las peticiones y los tipos de respuesta bien estructuradas, por lo tanto ROS proporciona servicios para las operaciones síncronas y tópicos para las operaciones asíncronas (Magyar et al., 2015). Finalmente, existe un nodo ROS Master responsable de la comunicación entre nodos, la administración del registro de los servicios y tópicos dentro del sistema.

- **Paquetes y distribuciones:** ROS se encuentra disponible en paquetes, los cuales son una pieza de software que encapsula sus funcionalidades para que sean sencillas de usar por aparte de los programadores, como es el caso de drivers de sensores, software de planeación y control de la navegación del robot, entre otros. Un paquete puede contener código fuente, librerías, nodos, archivos de compilación, ejecución, documentación y otros archivos relacionados con ROS (Estefo et al., 2019).

La arquitectura y el sistema de paquetes han llevado al éxito de ROS, el cual se ha convertido en la herramienta más utilizada para software robótico. La Tabla 2-4 muestra el número de paquetes de las diez últimas distribuciones de ROS, partiendo de la más reciente.

Tabla 2-4: Distribuciones de ROS

Distribución	Fecha de lanzamiento	# paquetes	Fin de vida útil
Noetic Ninjemys	23 de mayo, 2020	----	Mayo 2025
Melodic Morenia	23 de mayo, 2018	788	Mayo 2023
Lunar Loggerhead	23 de mayo, 2017	856	Mayo 2019
Kinetic Kame	23 de mayo, 2016	2509	Abril 2021
Jade Turtle	23 de mayo, 2015	1361	Mayo 2017
Indigo Igloo	22 de julio, 2014	3210	Abril 2019
Hydro Medusa	4 de septiembre, 2013	2110	Mayo 2015
Groovy Galapagos	31 de diciembre, 2012	2256	Julio 2014
Fuerte Turtle	23 de abril, 2012	2728	----
Electric Emys	30 de agosto, 2011	----	----

- **Características específicas:** ROS está equipado con varias características específicas para el desarrollo de aplicaciones robóticas, las cuales aceleran el desarrollo del software. Algunas de las más importantes son: definición de mensajes estándar para robots, librerías de geometrías del robot, lenguaje de descripción del robot, diagnóstico e

información de odometría, algoritmos de estimación de posición, módulos de localización, algoritmos de mapeo, módulos de navegación y creación de trayectoria, entre otros.

- **Herramientas:** Otra de las fortalezas de ROS es el potente conjunto de herramientas de desarrollo, varias de ellas incluyen depuración, visualización de variables, realización de gráficas, procedimientos e inspección del estado de la plataforma robótica, usando estas herramientas, el flujo de datos puede ser fácilmente visualizado y depurado. Las herramientas más populares son RVIZ que permite visualizar en 3D el espacio en el que los robots se desplazan y simular la información proporcionada por los sensores del robot. Por su parte, CATKIN es la herramienta de compilación que reemplaza al paquete rosbuild; es de código abierto, multiplataforma y se basa en CMake. La herramienta RQT es un módulo gráfico que permite visualizar la comunicación entre nodos, tópicos, servicios y el ROS Master (Tsardoulias & Mitkas, 2017). Finalmente, ROS permite integrar librerías de otros middlewares como el simulador 3D de Gazebo, OpenCV para el procesamiento de imágenes, Point Cloud Library (PCL) y MoveIt! ambas con propósitos de navegación.

2.4. Herramientas de diseño HW/SW

A medida que el uso plataformas con Linux embebido ha ganado popularidad entre los desarrolladores de hardware, muchos de los principales fabricantes de FPGA, como Xilinx, ofrecen herramientas para agilizar el proceso de diseño de sistemas embebidos con sus dispositivos, entre ellas se destacan Vitis y Vivado. Asimismo, ofrece PetaLinux para la personalización del sistema operativo que se aloja en el procesador del dispositivo, y cómo código abierto está el Framework PYNQ que reduce significativamente el tiempo de diseño y desarrollo de aplicaciones de hardware.

2.4.1. Vitis y Vivado

Vitis y Vivado son herramientas de diseño y desarrollo, la primera de ellas orientada al software y la segunda al hardware, las cuales desde la versión 2020 se distribuyen de manera unificada en Vitis Unified Software. Con Vivado la creación de IP’s (Intelectual Property) tradicionalmente se ha realizado en Verilog o VHDL a través de la captura de la intención del diseño a un nivel de transferencia de registros RTL, que luego se sintetiza para generar el archivo de configuración para programar la FPGA. Sin embargo, además de la utilización de otras IP’s propias de Xilinx es posible integrar en Vivado diseños de

terceros que hayan sido sintetizados con el motor HLS de Vitis. Esto significa un nivel de abstracción superior al convertir a RTL la intención del diseño que se ha escrito en un lenguaje de programación como C/C++ o SystemC. La utilización de C++ supone un avance en el propósito de reducir la curva de aprendizaje en la implementación de aplicaciones de hardware e igualmente les abre el campo a desarrolladores de otras áreas afines al desarrollo de software para que puedan construir sus propios IP's sin necesidad de tener un conocimiento avanzado en diseño de hardware.

2.4.2. PetaLinux

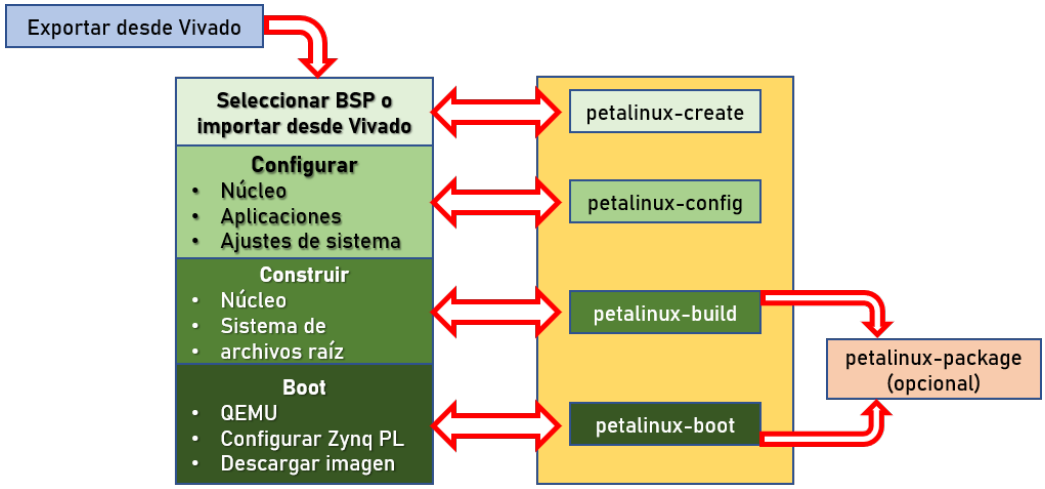
PetaLinux es una herramienta de Xilinx que simplifica el proceso de exportar el diseño de hardware en la FPGA al cargador de arranque Kernel y las aplicaciones de software, ya que incluye todo lo que se necesita para personalizar, crear e implementar Linux en plataformas Xilinx. PetaLinux requiere para su funcionamiento una descripción del dispositivo de hardware (BDF) creada con Xilinx Vivado, en esta descripción se encuentran entre otras, direcciones de memoria IP personalizadas, parámetros de configuración del sistema de hardware y los parámetros de configuración del procesador, mediante los cuales PetaLinux informa al Kernel de Linux sobre todos los componentes de hardware disponibles. Adicionalmente, El Kernel puede ser personalizado incluyendo módulos de arranque que permiten aumentar o restringir las capacidades de los procesos básicos de Linux.

Otra alternativa consiste en utilizar un paquete soporte del dispositivo BSP (Board Support Package), que es una colección de controladores esenciales que se han adaptado a la descripción del hardware proporcionada, y que normalmente distribuye el fabricante de la tarjeta de desarrollo. Un BSP es necesario para adaptar un sistema operativo específico y su entorno a un diseño de hardware específico (placa). Los BSP suelen incluir un sistema operativo de bajo nivel y el código del controlador de dispositivo que depende del hardware y sobre el que se asienta el resto del sistema operativo; también pueden incluir otros archivos que contienen directivas, parámetros de compilación y parámetros de hardware utilizados para configurar el sistema operativo. Para cada diseño de hardware se debe crear un BSP único para dicha configuración, ya que cada plataforma de hardware de Xilinx es configurable. El sistema de archivos raíz se construye y personaliza utilizando Yocto,

herramienta de código abierto de PetaLinux, que proporciona plantillas y métodos para crear sistemas integrados personalizados basados en Linux.

Las personalizaciones a nivel del sistema, que incluyen parámetros para modificar los comportamientos de arranque, las ubicaciones de las imágenes de arranque y el empaquetado de imágenes son las opciones finales abordadas por PetaLinux para generar un Kernel que pueda ser empaquetado con el cargador de arranque de primera etapa (FSBL) y el flujo de bits de hardware (Bitstream) en una imagen de Linux utilizando los parámetros de personalización. La Figura 2-5 presenta el flujo de los procesos para poder conseguir una imagen del sistema operativo, con los paquetes de personalización del usuario y el diseño del sistema de hardware.

Figura 2-5: Flujo del proceso para obtener una imagen del SO con PetaLinux

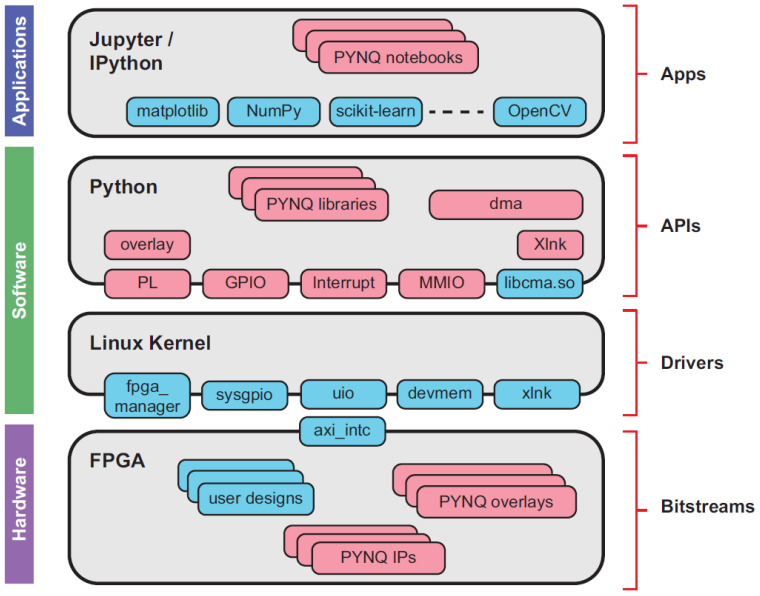


Fuente: El autor.

2.4.3. PYNQ

PYNQ es un Framework de código abierto para el diseño de sistemas embebidos, el cual mejora la productividad del sistema al integrar Python con las tarjetas de desarrollo Xilinx, inicialmente la PYNQ-Z1. Sin embargo, PYNQ no está orientado solo a unas tarjetas de desarrollo específicas, puesto que a través de PetaLinux se puede personalizar un BSP para hacer que un dispositivo oficialmente no soportado por PYNQ pueda ejecutar el Framework. PYNQ consta de tres capas: Aplicación, Software y Hardware como se observa en la Figura 2-6.

Figura 2-6: Capas de Aplicación, Software y Hardware de PYNQ.



Fuente: Tomado de Crockett et al., (2019, p527).

Mediante la capa superior de Aplicación el usuario interactúa con los periféricos y recursos del dispositivo a través de Jupyter Notebooks alojados en un servidor web que está dentro del procesador Arm ZYNQ, a este servidor se puede acceder a través de cualquier navegador con conexión de red. El entorno de Jupyter Notebooks provee de todas las funcionalidades para desarrollar código en Python, con librerías, consola y capacidad de acceder directamente a la capa de hardware del Zynq.

En la segunda capa de Software, se encuentran las librerías PYNQ y las API's que permiten la interacción entre las partes PS y PL del SoC; y también con el exterior a través de GPIO's. Allí también se aloja el sistema operativo basado en Linux, el servidor web para Jupyter y los controladores para interactuar con el hardware de Zynq. En la capa inferior se alojan los diseños de hardware llamados Overlays. Un Overlay consta de un archivo Bitstream y un archivo hardware *handoff* que son generados con las herramientas de Xilinx Vitis y Vivado. El usuario puede acceder a los Overlays con una línea de código desde la capa de Aplicación, como si fueran funciones, lo que permite el codiseño al integrar secciones de código en software con implementaciones en hardware para mejorar el rendimiento del sistema.

3. Integración

El codiseño (codesign) es una metodología más eficiente en términos de consumo energético y velocidad de procesamiento para la ejecución de tareas críticas. El codiseño se basa en la coordinación y paralelismo en las etapas de diseño hardware/software, lo que permite evaluar la concurrencia y sincronización de los subsistemas en las fases de diseño previas al desarrollo, esto con el fin de obtener sistemas que operen de forma correcta y sean altamente optimizados en cuestión de costos, potencia y rendimiento (Lentaris et al., 2016; Teich, 2012).

En la actualidad, el uso de codiseño como metodología para el desarrollo tecnológico ha permitido integrar arquitecturas hardware/software con base en FPGAs y CPU, con el fin de implementar estructuras conectadas con flujo horizontal, las cuales representan mayores rendimientos en la robótica (Lentaris et al., 2016). Lo anterior se logra ya que permite hacer uso del paralelismo para el procesamiento de información y/o desarrollo de algoritmos robustos en FPGAs, descargando así las tareas de la CPU, en la cual se pueden integrar sistemas operativos dedicados como ROS (Ohkawa et al., 2016; Sugata et al., 2017), facilitando el flujo de información entre el hardware y software, permitiendo obtener sistemas reconfigurables compatibles con ROS más robustos y con cierto grado de independencia secuencial para aplicaciones en la robótica (Nitta et al., 2018; Podlubne & Gohringer, 2019).

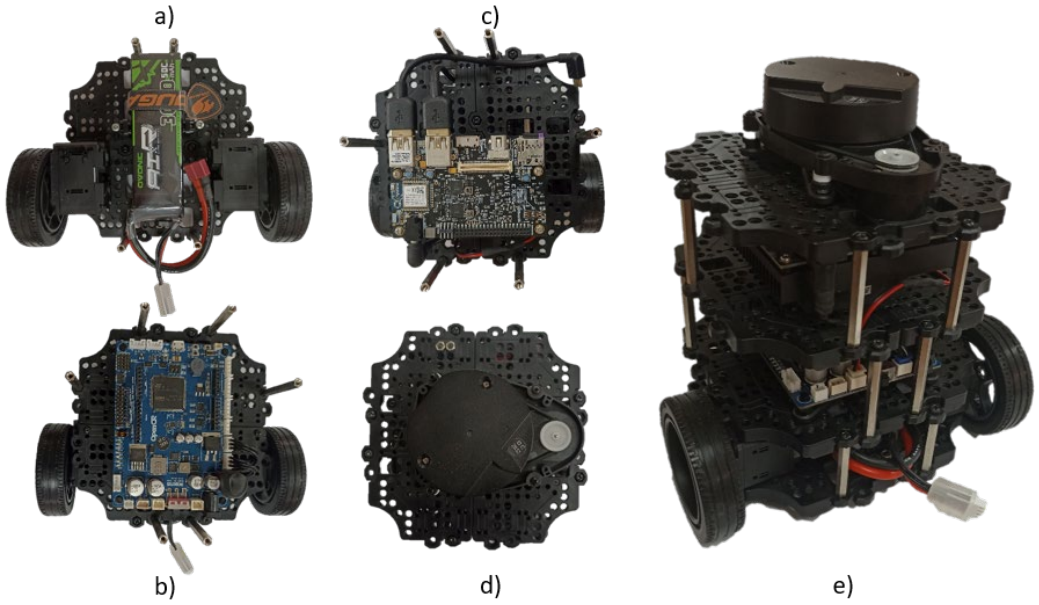
El proceso de integración se puede dividir en tres etapas: adecuaciones preliminares de hardware, construcción del sistema operativo e integración con ROS y la configuración de la red.

3.1. Hardware: adecuaciones preliminares

Como se estableció en el capítulo dos la plataforma robótica que mejor se ajusta a las necesidades del presente trabajo es la TurtleBot3 Burger, y por tanto el primer paso consiste en ensamblar el robot, ya que viene completamente desarmado de fábrica. Durante este proceso se establece que la batería LiPo de 11.1V y 1800mAh que viene en el plato inferior ofrece poca autonomía al robot, por tanto, se cambia por una nueva con mayor capacidad cuyas dimensiones permiten seguir ocupando el mismo espacio de la original, en este caso se elige una batería LiPo de 3 celdas en configuración 3S1P con capacidad de 3000mAh y que puede proveer 12.6V con carga completa.

En el tercer plato del robot se ubica la tarjeta Raspberry Pi, esta se remueve para darle espacio a la Ultra96V2, que relativamente tiene las mismas dimensiones. En la Figura 3-1 se observa el proceso de ensamble de los platos y la estructura final del robot con la nueva batería y el SoC-FPGA.

Figura 3-1: Modificación del robot a) adaptación de la batería b) conexión de motores y OpenCR c) Adaptación de la Ultra96V2 d) Instalación del LiDAR e) Estructura final.



Fuente: El autor

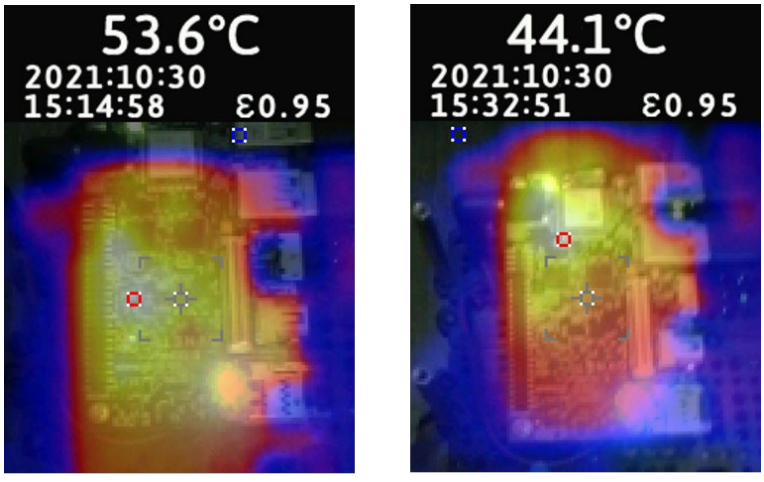
Para la integración de la Ultra96V2 se requiere partir del análisis del circuito esquemático para definir como se realiza la conexión eléctrica, ya que requiere 12VDC para su

funcionamiento, mientras que la Raspberry con la que cuenta la plataforma robótica desde fábrica se alimenta con apenas 5VDC. Originalmente en el TurtleBot3 la alimentación para la Raspberry se toma desde una de las fuentes externas de la tarjeta OpenCR, por tanto, se verifica cuáles eran los rangos de alimentación de la nueva tarjeta y si la OpenCR cumple con los valores de voltaje y corriente exigidos para proveer el voltaje de operación de la nueva tarjeta.

De acuerdo con el datasheet de la tarjeta Ultra96V2 y el circuito de alimentación principal, ésta tolera un rango de 8V a 16V. Por su parte, el OpenCR requiere una alimentación de 7V a 24V a través de batería o del cargador, y está programada para emitir una alarma sonora cuando el voltaje es menor a 11V, de igual manera, posee tres salidas de voltaje reguladas a saber 12V@1A, 5V@4A y 3.3V@800mA. Por tanto, se conectó a la fuente regulada de 12V.

Al hacer las primeras pruebas se mide la temperatura de funcionamiento de la Ultra96V2, alcanzando más de 53°C en unos pocos minutos de operación. Para reducir el sobrecalentamiento de la tarjeta de desarrollo se adapta un ventilador en la carcasa del disipador del SoC que mantuvo la temperatura por debajo de los 45°C, como se muestra en la Figura 3-2.

Figura 3-2: Imagen termográfica en el SoC a) sin ventilador b) con ventilador



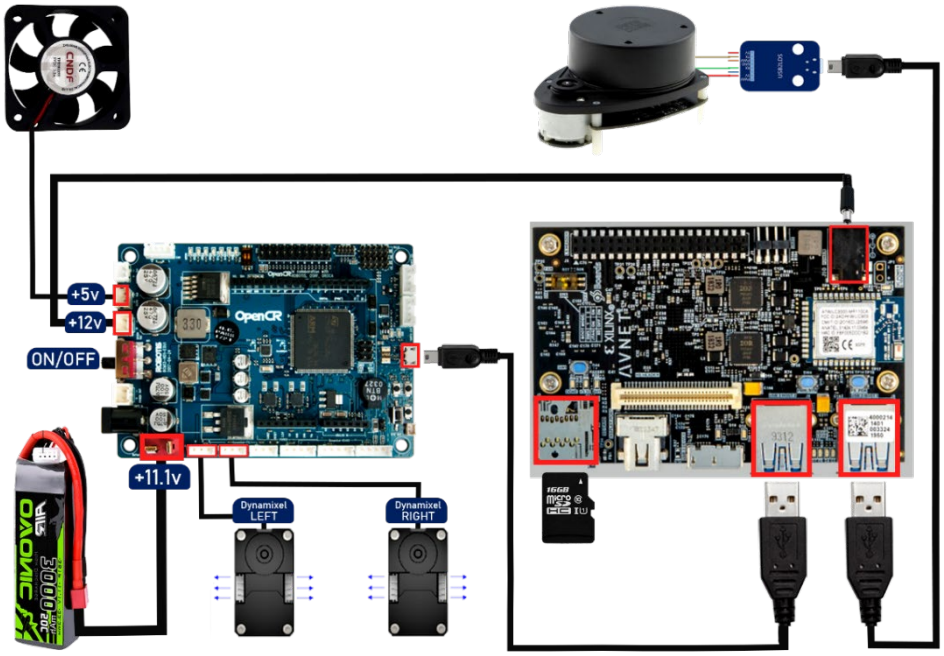
a) b)

Fuente: El autor.

Para la conexión del ventilador se analizan dos posibilidades, conectarlo a la Ultra96v2 a través de los pines dedicados con los que cuenta la tarjeta o suministrar su alimentación desde la OpenCR. La primera opción ofrece mejor control de temperatura y potencia del ventilador al estar conectado a un pin con PWM, sin embargo, también requiere mayor intervención eléctrica de la tarjeta, ya que ésta no tiene los pines en forma de conector sino para soldarla directamente. En contraste, suministrando la alimentación desde la OpenCR se usa el mismo conector que originalmente provee de voltaje a la Raspberry, por lo cual no es necesario intervenir físicamente a la tarjeta.

La Figura 3-3 presenta la interconexión eléctrica y de comunicación de los elementos que componen el robot. Como se puede apreciar, la comunicación USB entre la OpenCR y la Ultra96V2 permanece invariable con respecto a la conexión original, asimismo la conexión de ésta última con el LiDAR se realiza a través del controlador USB2LDS.

Figura 3-3: Diagrama de conexión de los elementos que componen el robot



Fuente: El autor

3.2. Software: sistema operativo del SoC

Una vez realizadas las adecuaciones mecánicas a la plataforma robótica para integrar la tarjeta de desarrollo, se procede a la construcción del sistema operativo para la Ultra96V2. Inicialmente se implementa a través de PetaLinux, herramienta que exige un equipo de cómputo que cumpla con especificaciones como: procesador a 2GHz con mínimo 8 núcleos, 100GB de espacio libre y 8GB de RAM.

Para la construcción del sistema operativo se parte del análisis de compatibilidad de las versiones existentes de BSP[1]. La Tabla 3-1 muestra la compatibilidad de las versiones de XILINX con las distribuciones de UBUNTU y a la vez, la compatibilidad de dicho sistema operativo con las distribuciones del middleware ROS.

Tabla 3-1: Compatibilidad ROS, UBUNTU y XILINX.

DISTRIBUCIONES		**VERSIONES DE XILINX**			
ROS	**UBUNTU**	**2018.3**	**2019.2**	**2020.2**	**2021.1**
Kinetic Kame	Xenial Xerux	16.04.3 – 16.04.4	16.04.5 – 16.0.6	16.04.5 – 16.04.6	16.04.5 – 16.04.6
Melodic Morenia	Bionic Beaver	---	18.04.1 – 18.04.2	18.04.1 – 18.04.4	18.04.1 – 18.04.5
Noetic Ninjemys	Focal Fossa	---	---	---	20.04 – 20.04.1

Las distribuciones de ROS son compatibles con una única versión de UBUNTU, igualmente existe una estrecha relación de dependencia entre dichas versiones del sistema operativo y las herramientas de desarrollo de XILINX, por tanto, al analizar estas características se encuentra compatibilidad entre el BSP 2018.3, la versión Xenial Xerux de UBUNTU y la distribución Kinetic Kame de ROS. Sin embargo, la fecha de lanzamiento de estas últimas data de mediados de 2016, lo que permite inferir que existen características y actualizaciones más recientes tanto del sistema operativo como del middleware necesarios para la construcción del sistema operativo de la Ultra96V2, por lo cual se descarta la implementación de dicho sistema a partir de un BSP existente.

Debido a lo anterior, se plantea construir el sistema operativo desde cero, para ello es necesario encontrar una capa de ROS compatible para ser incluida en la receta de Yocto

[1] Para el inicio de este trabajo se encontraba disponible la versión de BSP 2018.3.

en la herramienta PetaLinux. Esta opción fue descartada ya que en los inicios del desarrollo del sistema operativo no se encontraban disponibles capas compatibles. Fue hasta junio de 2021 que en el repositorio oficial de ROS.ORG en GitHub estuvo disponible la rama META-ROS para ROS2 y ROS1, en las distribuciones Melodic y Noetic de UBUNTU.

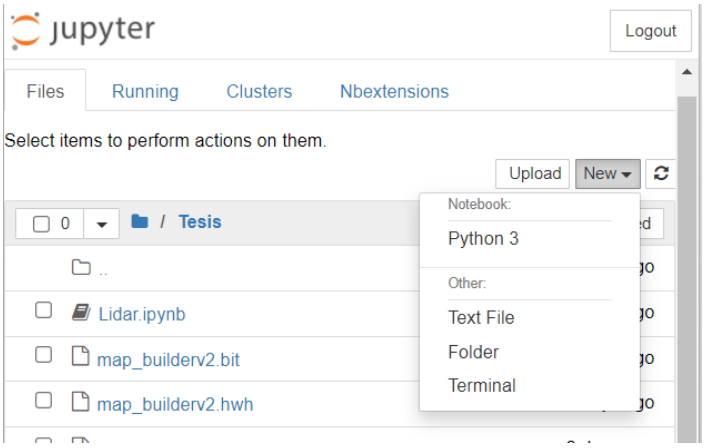
Finalmente, se construye el sistema operativo de la tarjeta Ultra96V2 a partir del Framework PYNQ según la compatibilidad de las versiones con las herramientas de Xilinx que se presentan en la Tabla 3-2. Con PetaLinux también es posible personalizar PYNQ para que incluya ROS, sin embargo, como se explica anteriormente, para la fecha de realización del proyecto no había una capa de ROS compatible para la receta de Yocto.

Tabla 3-2: Compatibilidad de PYNQ con las versiones de Xilinx.

VERSIÓN PYNQ	VERSIONES XILINX
2.4	2018.3
2.5	2019.1 – 2.019.2
2.6	2020.1 – 2020.2

De esta manera se realiza la instalación de la versión PYNQ 2.6 sin modificar, basada en la distribución Bionic Beaver de UBUNTU, cuyo entorno se presenta en la Figura 3-4. Desde el entorno de PYNQ se pueden crear Notebooks, realizar programas en Python, administrar el almacenamiento, gestión de carpetas y archivos. También cuenta con una terminal completamente funcional que permite la instalación de paquetes vía *apt-get*.

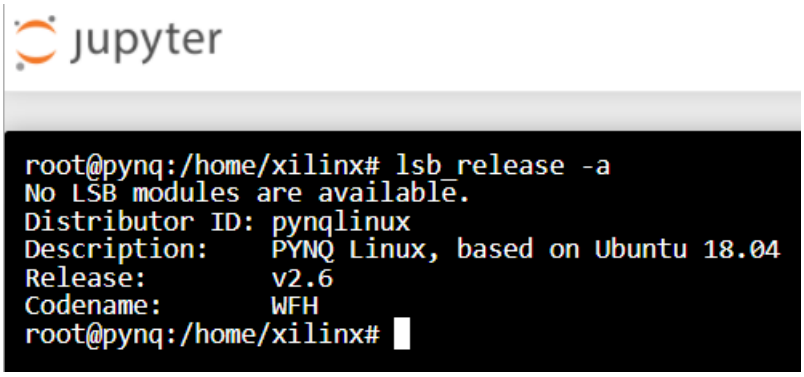
Figura 3-4: Entorno de Jupyter desde el Framework PYNQ



Fuente: El autor

Sin embargo, la instalación de ROS en el Framework PYNQ no es posible de manera directa, puesto que como se puede verificar en la Figura 3-5 que muestra la información del sistema operativo desde la terminal, la distribución de UBUNTU, aunque está basada en Bionic, tiene su *codename* como WHF así que el programa de instalación no encuentra los paquetes para esa distribución en el repositorio oficial de ROS.

Figura 3-5: Terminal de Jupyter mostrando la versión del SO



Fuente: El autor

La solución consiste en anular la detección automática del sistema operativo con la variable de entorno *ROS_OS_OVERRIDE* para forzar la instalación desde la lista de paquetes para UBUNTU Bionic. Una vez instalado ROS Melodic Morenia se realiza la instalación de los paquetes de TurtleBot3 que distribuye el fabricante del robot.

3.3. Configuración de la red

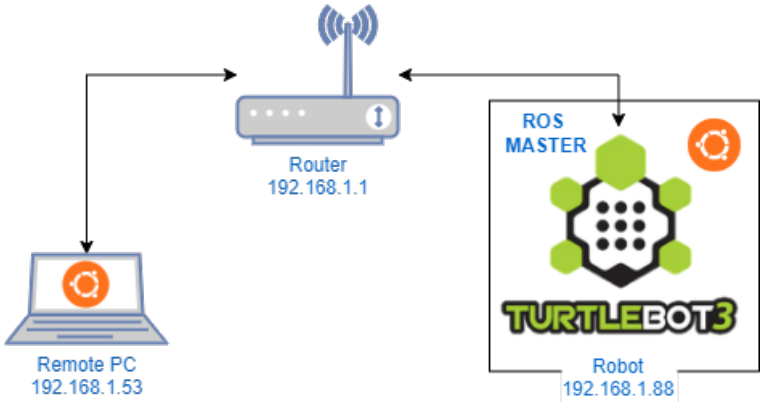
La siguiente etapa en la integración consiste en articular el entorno de trabajo, esto es el sistema operativo del equipo desarrollador, el sistema operativo de la tarjeta y establecer las direcciones del host y el ROS Master.

Normalmente cuando se trabaja con el TurtleBot3 se configura la red para que el ROS Master corra desde el computador remoto, esto se hace por dos razones, la primera porque el sistema operativo en el robot no cuenta con interfaz gráfica, y la segunda, porque esto demanda mayores recursos de la tarjeta del robot. Sin embargo, esta configuración también tiene sus desventajas como la imposibilidad de operar el robot sin un computador remoto que no tenga ROS instalado para que los nodos se registren en el Master.

Dado que el presente trabajo utiliza el Framework de PYNQ, no se requiere acceder al robot a través de SSH como se hace habitualmente con el TurtleBot3 original; por el contrario, se puede acceder a Jupyter notebook desde cualquier navegador web, incluso el de un teléfono celular. Por ello se decide dejar el Ros Master en el robot.

En cuanto al computador que actúa como host y como equipo de desarrollo HW/SW, se decide contar dos opciones de sistema operativo para afrontar los diversos problemas de compatibilidad que se puedan presentar en el proyecto y optimizar el tiempo de desarrollo. Así, el equipo se configura en Dual Boot con Ubuntu 18.04 y Windows 11 con Ubuntu Bionic en el subsistema de Windows para Linux (WSL2), y se establecen las direcciones de red en los archivos *bashrc* para ambos, aprovechando que el router asigna la misma IP al equipo sin importar en que sistema operativo está funcionando. En la Figura 3-6 se presenta el diagrama de red con Linux en el Remote PC, esta configuración es la habitual que se hace con el TurtleBot3 original, con la diferencia en que el Ros Master se ubica es en el robot.

Figura 3-6: Diagrama de la red con Linux

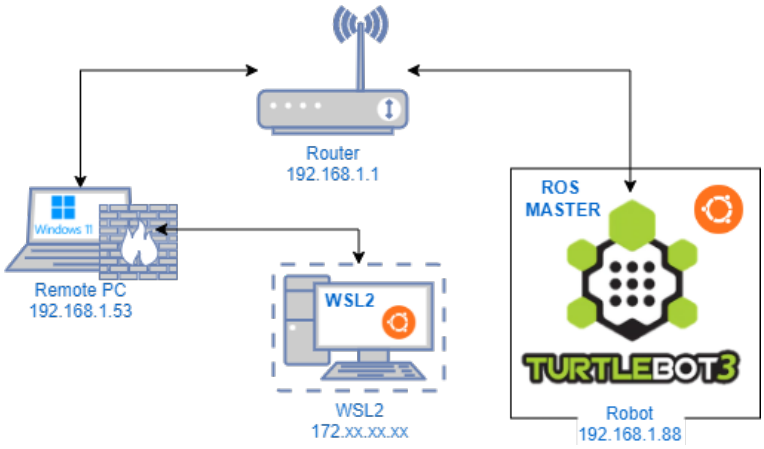


Fuente: El autor

Con respecto a la configuración de la red en Windows 11 presentada en la Figura 3-7, dado que no se puede instalar ROS1 se utilizó WSL2 con Ubuntu Bionic. Sin embargo, WSL2 se encontraba en su versión Beta en la que el adaptador predeterminado en puente se cambió por un adaptador de red virtual Hyper-V lo que derivó en varios problemas que dificultan el acceso a los recursos de red con WSL2 y que han sido documentados y reportados por la comunidad ante Microsoft. Por ejemplo, la IP del conmutador Hyper-V

cambia cada vez que se reinicia Windows, por lo que la puerta de enlace en WSL2 también cambia. A su vez, al hacer el redireccionamiento (port forwarding) del puerto 22 para el servidor SSH y del 11311 para la comunicación con el Ros Master, el firewall de Windows los bloquea.

Figura 3-7: Diagrama de la red con Windows 11 y WSL2



Fuente: El autor

Precisamente, de la comunidad de GitHub[2] se encuentra la solución temporal a los problemas de comunicación entre el Host PC y WSL2 que consiste en realizar un script para redireccionar los puertos TCP de los servicios WSL2 al sistema operativo host y crear de manera automática las reglas en el firewall de Windows, este script se agrega en el programador de tareas para que se ejecute al iniciar Windows. Por otro lado, para solventar la ausencia de interfaz gráfica en WSL2, se utiliza X410 para que las aplicaciones GUI como RViz y Rqt_Graph se puedan visualizar en Windows 11.

[2] https://github.com/microsoft/WSL/issues/4150

4. Desarrollo de la aplicación

El codiseño hardware/software presenta como desafío principal identificar las rutinas y/o funciones que se deben ejecutar en hardware o software. De esta manera, el desarrollo de la aplicación inicia con la implementación en software para establecer un punto de partida en cuanto al desempeño del algoritmo y comparar los tiempos de ejecución, de tal forma que se puedan definir los criterios para el particionamiento de las tareas. Esto es, poder encontrar secciones del algoritmo que se ejecutan de manera repetitiva o que hacen uso de operaciones trigonométricas, matriciales, entre otras, que suponen un alto costo computacional y que podrían ser susceptibles de implementarse en hardware para reducir dichos costos. En la sección 1.4 se presentó el resumen de las tareas consideradas críticas en el proceso de navegación del robot, para el presente proyecto se selecciona la tarea de construir el mapa de ocupación a partir de los datos del LiDAR del robot móvil.

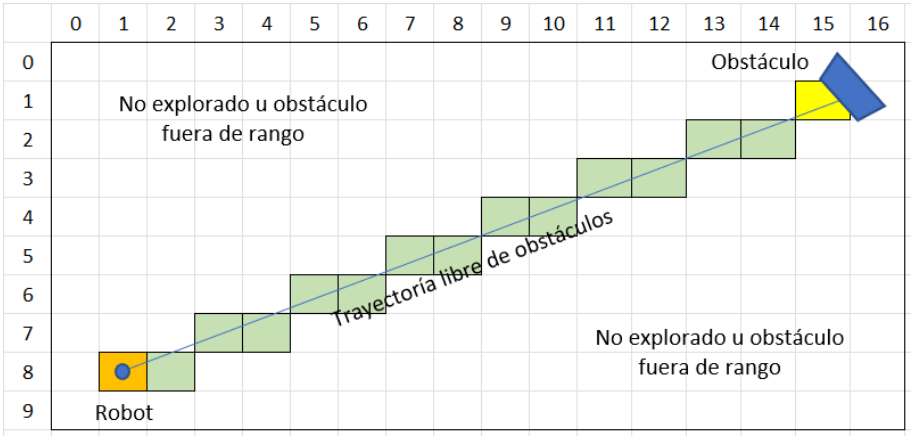
4.1. Implementación en software

Los requerimientos de la rutina son obtenidos del análisis de la cantidad de datos que se van a procesar y de cómo se deben presentar. El TurtleBot3 tiene el sensor LiDAR LDS01, situado en el plato superior del robot, que da cinco revoluciones en un segundo y captura 360 valores por cada revolución, uno por cada grado de desplazamiento angular. Estos valores corresponden a la distancia del LiDAR hasta los objetos que se encuentran en el entorno del robot y se encuentran en coordenadas polares, con estos datos la rutina calcula la posición del robot en coordenadas cartesianas y lo ubica en una cuadrícula de ancho predeterminado por el usuario, a su vez cada punto (x,y) que representa un obstáculo en el entorno del robot se representa por una cuadro en el mapa de ocupación.

Para cada cuadro en la posición (m,n) que representa un obstáculo se evalúa el rayo que lo une con la posición del robot mediante el algoritmo de línea de Bresenham para determinar que cuadros conforman la mejor aproximación a una línea recta. La Figura 4-1

muestra la aplicación del algoritmo para los puntos Robot(8,1) y Obstáculo(1,15), los cuadros en verde corresponden al conjunto de coordenadas que devuelve el algoritmo.

Figura 4-1: Mapa de ocupación definido con el algoritmo de Bresenham



Fuente: El autor.

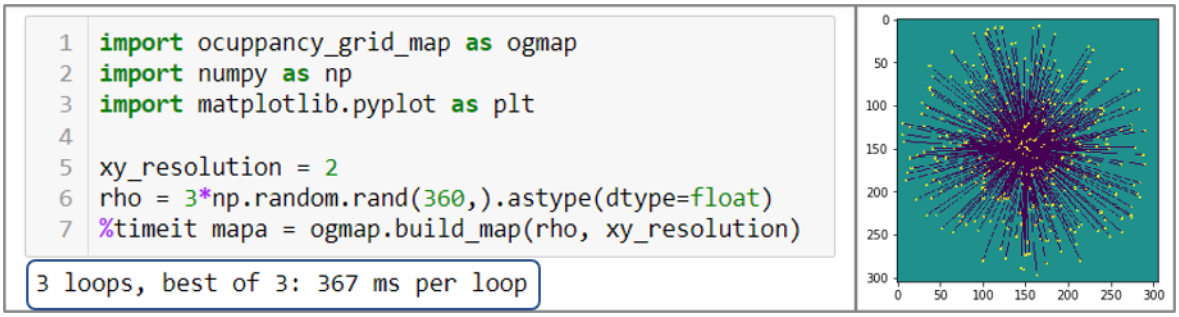
Con estas coordenadas el mapa se construye asignándole a cada cuadro un valor 0 o 1 para representar la ausencia o presencia de un obstáculo o un valor de 0.5 que denota incertidumbre sobre el estado actual del cuadro. A continuación, se presenta el pseudocódigo que realiza el proceso de construcción del mapa, el algoritmo de línea de bresenham se presenta como otra función, pero también fue implementado.

```
kernel_lidar(Limits, Points, Resolution){
        Find Origin from Limits & Resolution
        Initialize map = Unknown state
        for each (x,y) in Points do
                FreePath = bresenham(Origin, Points)
                for each (ix,iy) in FreePath do
                        map(ix,iy) = Free
                end for
                map(x,y) = Busy
        end for
        Return map
        }

void lidar(Input, Ouput, xy_res) {
        Load Rho & Theta from Input
        Convert (Rho,Theta) to rectangular coordinates (x,y)
        Compute Limits min(x,y) & max(x,y)
        Fit each Point (x,y) to grid Resolution
        map = kernel_lidar(Limits, Points, Resolution)
        Write map to Output
        }
```

En la Figura 4-2 se observa la respuesta en un notebook del algoritmo escrito en lenguaje Python ante una entrada aleatoria que simula los datos del LiDAR.

Figura 4-2: Resultado de la ejecución del algoritmo en software.



Fuente: El autor

El tiempo de ejecución de la rutina fue de 367ms, la cual fue calculada con la instrucción *%timeit*. Este tiempo es bastante alto para desarrollar tareas de procesamiento en tiempo real, ya que corresponde tan solo a un único barrido del LiDAR.

4.2. Implementación en hardware

A diferencia de la rutina en software, para la implementación en hardware se deben definir varios aspectos relacionados con los recursos de la tarjeta como son las interfaces de la parte lógica (PL) y el procesador (PS), la forma de comunicación y transferencia de datos entre ellas, la forma de describir la IP y la integración del Overlay en Vivado. Esto obedece a que las rutinas que se implementan en hardware en efecto son elementos físicos que se interconectan entre sí, con limitaciones en los tamaños de los buses, en la cantidad de memoria instalada, entre otras.

4.2.1. Interfaces PS-PL

Las interfaces PS-PL permiten la conexión punto a punto entre el bloque PS y el bloque PL para la transferencia de datos, señales y direcciones entre clientes maestros y esclavos. Dichas interfaces son esenciales para la comunicación con los aceleradores de Hardware ubicados en la FPGA. La Tabla 4-1 muestra los doce puertos para la comunicación entre PS y PL con los que cuenta la tarjeta ULTRA96V2, los cuales se encuentran agrupados en siete interfaces tipo AXI.

Tabla 4-1: Interfaces PS-PL.

Interfaz	Tamaño del Bus	Tamaño de la dirección	Maestro	Esclavo
S_AXI_ACP_FPD	128	40	PL	PS FPD
S_AXI_ACE_FPD	128	40	PL	PS FPD
S_AXI_HPC{0,1}_FPD	128	49	PL	PS FPD
S_AXI_HP{0:3}_FPD	32/64/128	49	PL	PS FPD
S_AXI_LPD	32/64/128	49	PL	PS LPD
M_AXI_HPM{0,1}_FPD	32/64/128	40	PS FPD	PL
M_AXI_HPM0_LPD	32/64/128	32	PS LPD	PL

En las primeras cinco interfaces de la tabla, el PL actúa como maestro mientras que PS lo hace como esclavo. El ACP provee una ruta unidireccional entre el PL y la memoria caché L2 de la APU, en contraste el puerto ACE ofrece una ruta bidireccional entre la memoria PL y la unidad de Interconexión Coherente de Caché (CCI), permitiendo el intercambio de información actualizada entre los bloques de Hardware y el PS.

Los dos puertos Coherentes de Alto Rendimiento (HPC) están conectados directamente a la CCI y a la Unidad de Administración de Memoria del Sistema (SMMU); así mismo, la interfaz de Alto Rendimiento (HP) que comprende cuatro puertos, pasa a través de la SMMU pero no se conecta a la CCI sino al conmutador central de interconexión, llegando hasta el controlador de la memoria DDR a través de tres puertos dedicados (Ver Anexo A), esta última característica los ubica como los mejores puertos para el proyecto debido al requerimiento de intercambio de datos entre PS y PL. Finalmente, La interfaz S_AXI_LPD, donde el PL actúa como maestro, ofrece una ruta de alto rendimiento hasta el dominio de baja potencia (LPD) del PS y también permite acceder al RPU cuándo está apagado el domino de potencia completa (FPD) del procesador.

Las dos últimas interfaces mostradas en la Tabla 4-1, donde el PL actúa como esclavo (HPM) comprenden tres puertos, dos de ellos para el FPD y el restante para el LPD. Estas interfaces son adecuadas para proporcionar a los maestros FPD en el PS acceso a las memorias en el PL, para que puedan transferir grandes cantidades de datos, por tanto, al igual que la interfaz HP, son elegidos para el control del flujo de datos entre el diseño de hardware y la memoria del sistema.

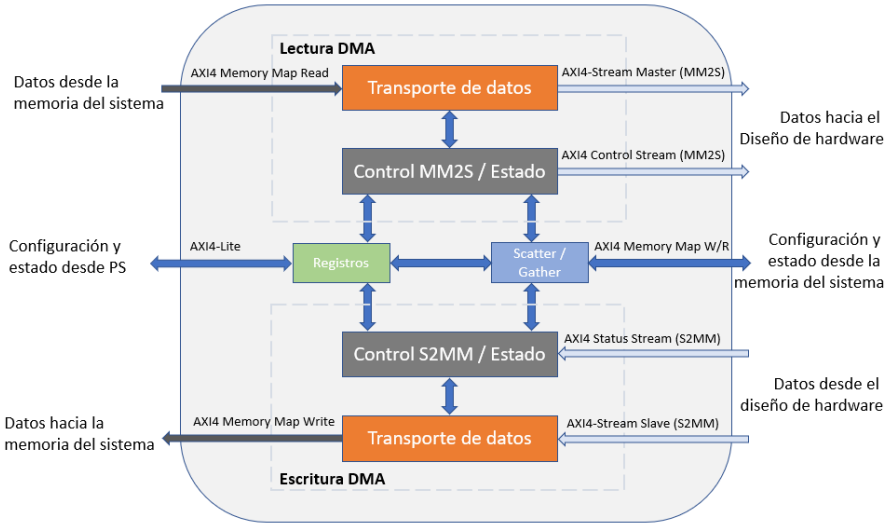
4.2.2. Intercambio de datos entre PS-PL

Para el desarrollo del sistema de Hardware en el dispositivo Zynq MPSoC se debe evaluar la cantidad de datos que se transfieren entre el PS y PL, intercambio que depende de las restricciones de la aplicación, el tipo de datos y los recursos de Hardware del sistema. A partir de la rutina en Software se define que desde el PS se deben enviar los datos de Rho y Theta, es decir 360x2 valores en punto flotante. A su vez, el diseño de Hardware debe retornar el mapa de ocupación en forma de una matriz de datos en punto flotante. La distancia máxima de detección de obstáculos por parte del LiDAR es de 3.5m con una resolución mínima de cuadricula de 2cm, que resulta en una malla de 350x350. Por lo cual, se define el tamaño del mapa de ocupación en 400x400, teniendo en cuenta un pequeño margen de tolerancia.

Normalmente, la transferencia de datos en un sistema embebido se lleva a cabo a partir de instrucciones que realiza el procesador para acceder a la memoria, sin embargo, este método es ineficiente cuando se trata de grandes cantidades de datos como los que implica el mapa de ocupación, puesto que el procesador estaría ocupado accediendo a la memoria sin la posibilidad de realizar otras acciones. La solución a este inconveniente se encuentra en el subsistema de hardware DMA que se encarga de acceder a la memoria del sistema, con una intervención mínima por parte del procesador. La familia del SoC UltraScale a la que pertenece la tarjeta seleccionada cuenta con unidades DMA en el PS y de bloques IP CORE AXI DMA en el PL, la cantidad de éstos últimos está limitada solamente por la disponibilidad de recursos de la FPGA.

La Figura 4-3 muestra las interfaces del controlador AXI DMA, el cual posee dos bloques independientes, uno para lectura y otro para escritura, su configuración se realiza desde el procesador cuando está en modo simple, a su vez, también permite hacer la configuración desde un conjunto de descriptores precargados en la memoria del sistema con el modo Scatter/Gather. En el modo simple la configuración de la DMA se realiza desde el procesador mediante la interfaz AXI4-Lite, este es un puerto esclavo, por tanto, desde el procesador utiliza la interfaz HPM.

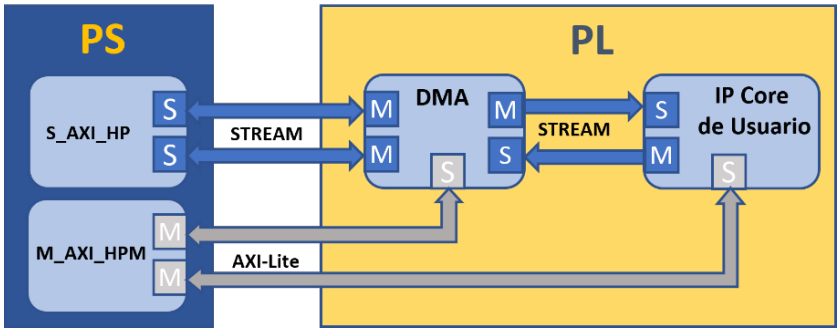
Figura 4-3: Interfaces en el controlador AXI DMA.



Fuente: El autor

El transporte de datos se da mediante los puertos S2MM y MM2S usando el protocolo AXI4-Stream, el cual actúa como una interfaz estándar para conectar componentes que desean intercambiar datos, en este caso la memoria del sistema y el diseño de hardware. Los puertos maestros MM2S de la DMA llevan los datos desde la memoria del sistema hacia el diseño hardware, una vez procesados en el PL, se transportan de nuevo a la memoria del sistema a través del puerto S2MM, que, por ser maestro, se debe conectar a alguno de los puertos de la interfaz S_AXI_HP, los cuales tienen conexión dedicada con la memoria del sistema. La Figura 4-4 muestra un esquema simplificado de la conexión de la DMA con el IP Core del usuario y el Procesador.

Figura 4-4: Esquema de conexión de la DMA en modo simple



Fuente: El autor

4.2.3. Diseño del IP Core en HLS

El diseño del IP Core del usuario se realiza en Vitis 2020.2 con lenguaje de alto nivel HLS, que permite describir de manera comportamental el diseño de hardware para luego convertirlo en una estructura RTL que se exporta como IP hacía Vivado para hacer la integración. Con base en las decisiones que se tomaron sobre las interfaces PS-PL y la manera como se transportan los datos se inicia el proceso de descripción del hardware.

Con respecto a las interfaces, la directiva *#pragma HLS INTERFACE* define la forma en la que se crean los puertos RTL a partir de los argumentos de la función principal, éstos se sintetizan en interfaces y puertos para definir el protocolo de comunicación entre el diseño HLS y los componentes externos al diseño que se integran en vivado (Xilinx, 2021). Por tanto, se establecen dos puertos axis con el protocolo *Stream* para la entrada y la salida de la función, y un puerto axi_lite que agrupa las señales de control de la DMA y los registros que se requieren para intercambiar datos simples entre PS-PL como la resolución de la cuadrícula que representa el mapa.

A continuación, se desarrolla la rutina en HLS siguiendo el pseudocódigo de la sección 4.1 y agregando algunas de las directivas de optimización que provee Vitis HLS para mejorar la latencia, la administración de los recursos de hardware y el rendimiento del diseño a través de paralelización. En el diseño se usaron las directivas ARRAY_PARTITION, para particionar un arreglo de gran tamaño en arreglos más pequeños que contribuyen a la paralelización, PIPELINE para habilitar la ejecución concurrente de operaciones en un bucle al reducir el intervalo de inicio[3], UNROLL para permitir que las iteraciones en un bucle se produzcan en paralelo a través de la creación de múltiples copias del bucle en el diseño RTL.

El programa en HLS se sintetiza y en la Figura 4-5 se observan los detalles del rendimiento del diseño por cada bucle, como la latencia y la ejecución concurrente en los bucles. Los bucles se han etiquetado de acuerdo con la tarea que cumplen, de tal manera que se observa el cumplimiento del pseudocódigo planteado en la sección 4.1.

[3] Tiempo necesario para ejecutar una iteración

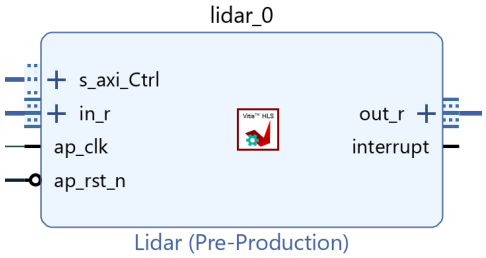
Figura 4-5: Resumen de la síntesis del código en HLS

Modules && Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined
lidar		-0.05	-	-	-	0	-	no
kernel_bham		-0.05	7944	7,944E4	-	7944	-	no
sin_or_cos_float_s		-	10	100.000	-	1	-	yes
p_hls_fptosi_float_i32		-	0	0.0	-	0	-	no
generic_round_float_s		-	1	10.000	-	1	-	no
Load_Rho		-	45	450.000	1	1	45	yes
Initialize_Map		-	129600	1,296E6	1	1	129600	yes
Load_Theta		-	408	4,080E3	50	1	360	yes
Build_map		-	?	?	?	-	360	no
Write_Map		-	16201	1,620E5	3	1	16200	yes

Fuente: El autor

Una vez se ha sintetizado el IP Core, éste se agrega al catálogo de Vivado para poder integrarlo en el diseño de hardware, la Figura 4-6 se visualiza en forma de bloque con entradas y salidas.

Figura 4-6: IP Core sintetizado con Vitis HLS una vez importado en Vivado



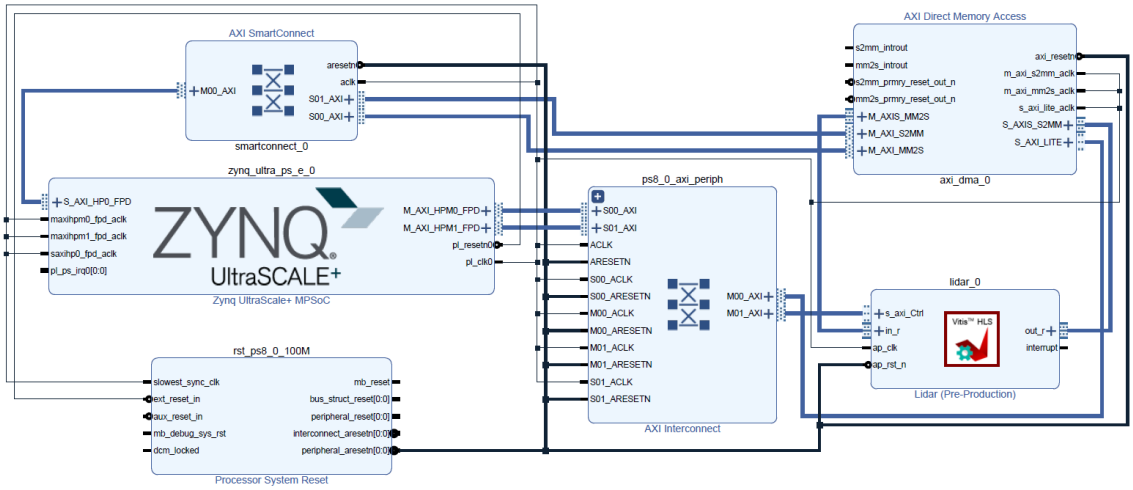
Fuente: El autor

El bloque cuenta con un pin de entrada in_r y un pin de salida out_r con interfaz AXI4 en modo Stream que recibe y envía los datos entre las partes PS y PL del SoC. Mediante el pin axilite s_axi_Ctrl se envían los demás parámetros de la función en hardware y también se controla el flujo de los datos entre las IP del diseño. También cuenta con un pin para el reloj y un pin para el reset. Aunque la IP cuenta con un pin de interrupción, éste no se utilizó en el diseño.

A continuación, se crea un bloque de diseño y se agrega el Procesador, el IP Core importado de Vitis, la DMA y los módulos de interconexión que permiten la comunicación entre los diferentes elementos y recursos del MPSoC, luego se configura la DMA en modo simple, se define el tamaño del Stream de datos y se habilita para lectura y escritura,

igualmente, en el PS se seleccionan las interfaces S_AXI_HP y M_AXI_HPM como se definió en la sección anterior. En la Figura 4-7 se resume la cantidad de los elementos de hardware que intervienen en el diseño, vivado realiza la conexión de los bloques de manera semiautomática.

Figura 4-7: Diseño completo de hardware

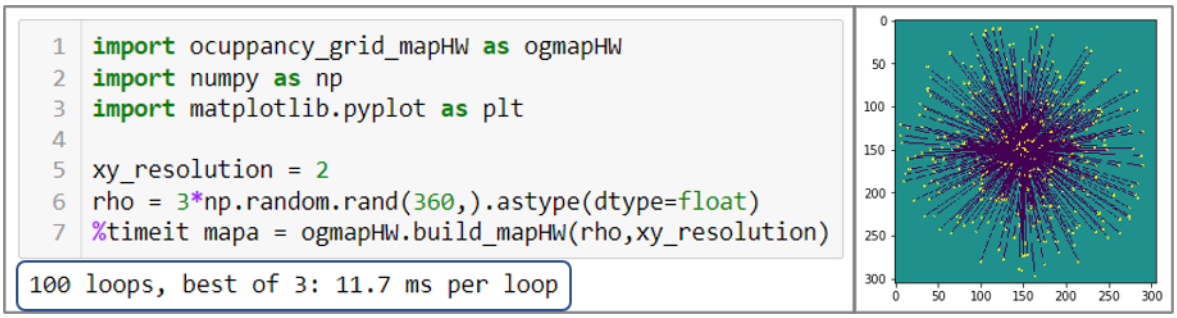


Fuente: El autor

El bloque de diseño se 'envuelve' en un contenedor (Wrapper) que mapea los puertos de entrada/salida del diseño a los pines físicos de la tarjeta los cuales están en el archivo de restricciones. Con el Wrapper listo se realiza la validación del diseño y se transforma en una representación a nivel de compuertas con la sintetización. Una vez finaliza la sintetización, se implementa el diseño, esto consiste en la optimización de la lógica, su colocación en los recursos del dispositivo físico, la conexión entre los componentes y la optimización física y de la potencia del diseño.

Para exportar el diseño implementado se genera el bitstream, que es el archivo de programación que requiere PYNQ para poder acceder a los Overlays en la capa de hardware a través de la capa de aplicación. En la Figura 4-8 se observa el resultado de la ejecución en hardware de la función Build_mapHW. Al igual que en software, se tiene la misma estructura del código y se utilizan datos aleatorios para simular el LiDAR. El diseño de hardware toma menos de 12 ms para procesar la misma cantidad de datos comparado con los 367ms que requiere la función en el software lo que corresponde a una acelaración de hasta 30 veces.

Figura 4-8: Resultado de la ejecución del algoritmo en implementado en hardware.



Fuente: El autor

4.3. Codiseño y Resultados

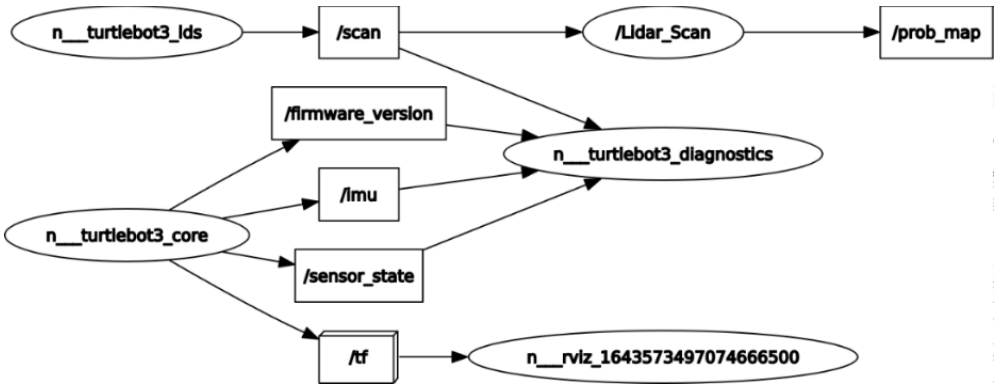
Ahora que se ha programado la rutina en software y se ha realizado la descripción del hardware se hace el codiseño de la aplicación robótica con los datos reales adquiridos por el LiDAR, utilizando el Framework PYNQ. En un notebook se crea la rutina principal que se conecta con ROS para adquirir los datos del LiDAR y para visualizar el mapa de ocupación en Rviz, a su vez, esta rutina hace uso de las funciones, la desarrollada en software y la descrita en hardware. El pseudocódigo que se muestra a continuación corresponde a la Clase principal de la aplicación.

```
class Lidar(object):
        def __init__(self):
                Initialize rho & theta
                Set a Subscriber Node to LaserScan with a Callback
                Set a Publisher Node for Topic OccupancyGrid
        def callback_lidar(self, msg):
                self.rho = msg.ranges
        def void_main(self):
                xy_resolution = 2 #cm
        while True:
                try:
                        mapa = build_map(self.rho, xy_resolution,SW_or_HW)
                        Create a 'OccupancyGrid' message
                        Publish message in Node
                except KeyboardInterrupt:
                        break
if __name__ == '__main__':
        rospy.init_node('Lidar_Scan')
        my_node = Lidar()
        my_node.void_main()
```

Los topics y nodos activos de ROS se visualizan con la herramienta de *rqt_graph* y se presentan en la Figura 4-9, allí se observa que el nodo de la clase principal */Lidar_Scan* está suscrito al topic */scan* de donde obtiene los datos del LiDAR, el nodo en sí mismo se

encarga de procesar los datos en la clase principal seleccionando la ejecución de la función build_*map* en hardware o en software.

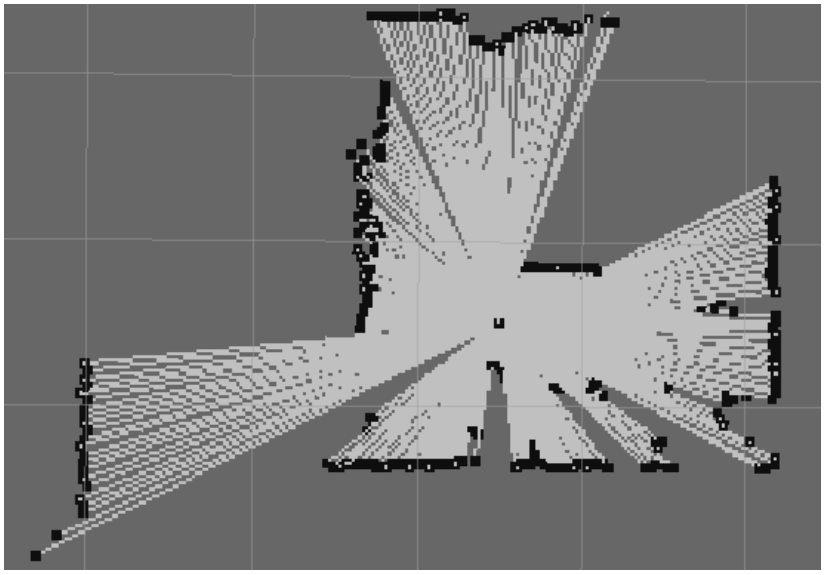
Figura 4-9: Gráfica de la aplicación RQT_GRAPH (obtenida de ROS).



Fuente: El autor

Así mismo, el nodo principal publica en el topic */prop_map* el mapa de ocupación, este mensaje es de tipo *OccupancyGrid* que pertenece a la clase de ROS que define la estructura de los mensajes para los algoritmos de navegación. A través de Rviz se visualiza el mapa de ocupación resultante (Figura 4-10), ya sea que se utilice la función en software o la de hardware para procesar los datos, para ROS esta decisión es transparente.

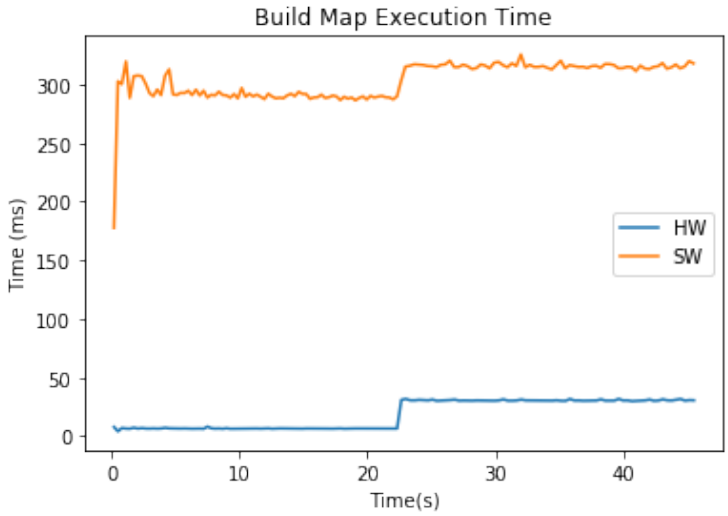
Figura 4-10: Mapa de ocupación generado por la aplicación RViz.



Fuente: El autor

La Figura 4-11 presenta el tiempo de ejecución de las rutinas de construcción del mapa de ocupación tanto en software como en hardware. Cuando RViz no está consumiendo el nodo que publica el mapa, el tiempo de ejecución fue de aproximadamente 295ms y 10ms para el software y el hardware respectivamente.

Figura 4-11: Comparación del tiempo de ejecución de las funciones en HW y SW

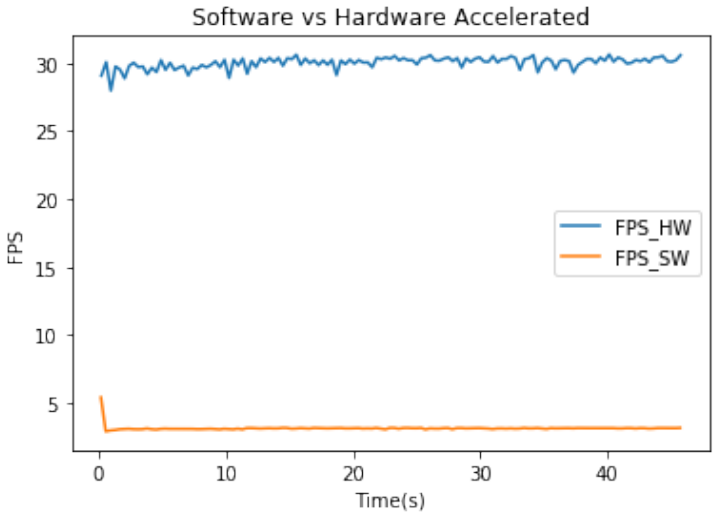


Fuente: El autor

Una vez se inicia a visualizar el mapa en la herramienta de RViz estos tiempos aumentan debido a que el mensaje que se envía desde el PS se debe construir y luego transmitir y esto supone un tiempo considerable. Así el tiempo de ejecución de la función en el PS sube hasta aproximadamente 315ms, por su parte, el diseño de hardware en el PL tan solo tarda alrededor de 30ms. Es decir, la mejora del tiempo de ejecución es del orden de 10x. La diferencia entre los tiempos de ejecución antes y después del inicio de RViz se mantienen constante en 20ms, esto se explica porque la construcción del mensaje del mapa se realiza en software así que no hay forma de que el diseño de hardware lo modifique.

De manera similar, en la Figura 4-12 que muestra la cantidad de cuadros por segundo (FPS) procesados, se observa una mejora de hasta 10 veces con la función acelerada por hardware, esto es ~3FPS frente a aproximadamente ~30FPS.

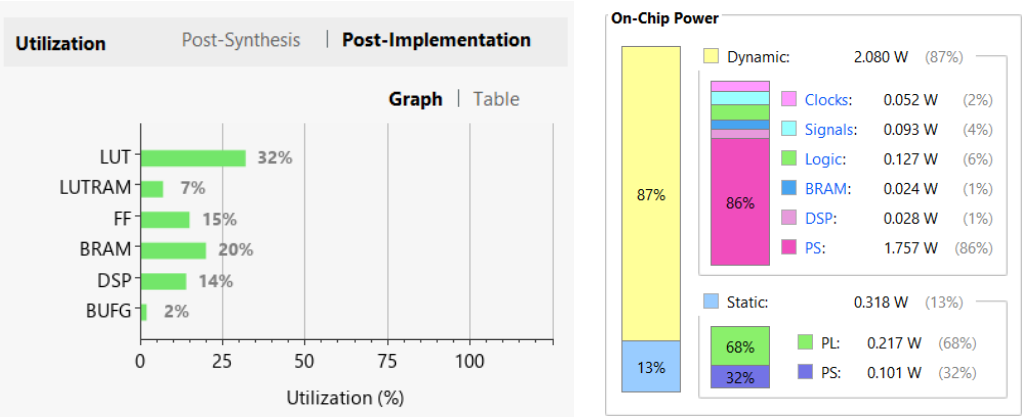
Figura 4-12: Comparación entre FPS de la rutina en software y el diseño de hardware



Fuente: El autor

Con relación a la utilización de los recursos de la Ultra96v2, vivado entrega el reporte de la Figura 4-13 que contiene el porcentaje de recursos lógicos consumidos por la aplicación, de manera similar, el reporte proporciona la estimación de la distribución de la potencia a nivel del chip MPSoC que para el presente diseño Xilinx lo estima en 2,08W.

Figura 4-13: Reporte de Vivado sobre los recursos utilizados y la potencia estimada



Fuente: El autor

5. Conclusiones y recomendaciones

5.1. Conclusiones

La integración de ROS en una tarjeta de desarrollo con recursos de hardware reconfigurable como una FPGA, amplía las posibilidades de solución de problemas complejos en la robótica móvil, ya que le provee al desarrollador la posibilidad de particionar las tareas para que las ejecute la unidad de procesamiento o el bloque de lógica programable. Así, la aceleración por hardware, esto es la asignación al bloque PL de las tareas críticas o los algoritmos computacionalmente complejos, reduce significativamente los tiempos de ejecución de las rutinas.

El uso de software de alto nivel como HLS facilita la migración de algoritmos desarrollados para ROS en C++, sin embargo, para aprovechar las ventajas del diseño de hardware se debe conocer la arquitectura del dispositivo debido a que la descripción del hardware asigna recursos físicos del FPSoC y por tanto son limitados. Asimismo, la implementación de una solución no es única ya que en el PL existen diferentes conexiones entre los bloques de hardware con distintas ventajas y desventajas que se deben evaluar según la aplicación para poder optimizar el diseño a través de las directivas que dispone Vitis HLS para habilitar la ejecución concurrente, administrar el acceso a la memoria del sistema, intercambiar datos entre el bloque de lógica programable y el procesado, entre otras.

Si bien Petalinux es la principal herramienta para la creación y personalización de imágenes de arranque para las tarjetas de Xilinx, es importante tener en cuenta que en el evento de requerir modificar el diseño se tiene que reconstruir todo el sistema operativo y volver a empaquetar el Bitstream del nuevo diseño de hardware. Lo anterior resulta tedioso en un ambiente de desarrollo donde se deben probar varios prototipos antes de obtener un diseño definitivo. En contraste, PYNQ proporciona una forma más simplificada de

acceder a su PL, ya que incluye diferentes operaciones de alto nivel en la API de Python, de tal manera que no se requiere construir de nuevo el sistema operativo con PetaLinux, sino que desde la capa de aplicación se carga el archivo Bitstream generado por vivado, el cual contiene la información para programar la FPGA.

El espíritu de la comunidad de ROS consiste en compartir algoritmos y librerías para facilitar el desarrollo de aplicaciones para robots; sin embargo, se ha limitado solo a herramientas de software. Por ello, a partir de este trabajo se propone la creación de un repositorio de IPs compatibles con los algoritmos de ROS donde con herramientas de hardware reconfigurable se aborde la solución de los nuevos retos de la robótica móvil, como la fusión de sensores, el procesamiento de grandes cantidades de datos en tiempo real, la implementación de técnicas de inteligencia artificial, entre otras nuevas técnicas que actualmente son un desafío para implementar en las tarjetas basadas únicamente en CPUs o GPUs.

Dentro de las tareas que se pueden abordar con una arquitectura como la presentada en este proyecto están algunas más complejas como la navegación autónoma, la evasión de obstáculos, el reconocimiento de patrones, la clasificación de objetos, la interacción humano robot, entre otras. Muchas de estas tareas son de alto costo computacional y por ello demandan que el robot móvil esté equipado con tarjetas de desarrollo con excelentes prestaciones computacionales. Asimismo, como el aumento en las capacidades de cómputo del robot normalmente va ligado a un alto consumo de energía, contar con dispositivos potentes, pero de bajo consumo como las FPGA es imprescindible para la autonomía del robot dado que deben operar con baterías.

La implementación en hardware de la función que construye el mapa de ocupación mejora el rendimiento de la aplicación al reducir el tiempo de ejecución hasta en diez veces, comparado con la ejecución en el procesador, sin embargo, existe un cuello de botella que no se resuelve con el uso de la FPGA, ya que este es atribuible a la comunicación entre los nodos de ROS cuyo protocolo está desarrollado por completo en software. En trabajos futuros se podría explorar la posibilidad de diseñar en hardware todo el protocolo de comunicación de ROS.

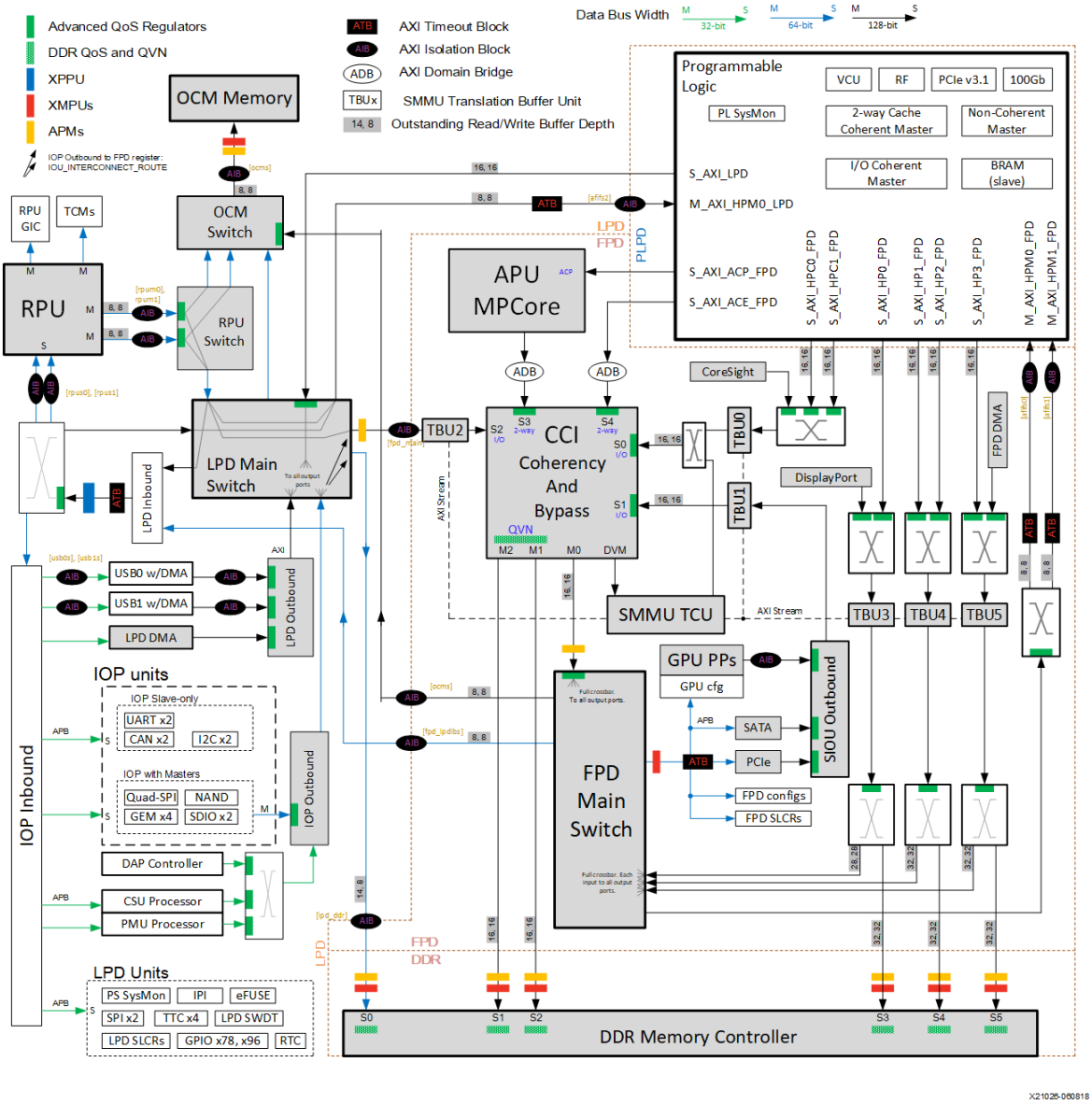
5.2. Recomendaciones

En las pruebas realizadas con Petalinux se trabajó con un equipo de cómputo que cumplía con los requerimientos mínimos que establece Xilinx; sin embargo, las 8GB de capacidad de la memoria RAM fue insuficiente para poder utilizar las herramientas de Xilinx de manera fluida, por lo tanto, se recomienda mínimo 12GB y que el espacio de almacenamiento sea en estado sólido con mínimo 100GB libres.

Aunque se utilizaron los dos sistemas operativos para el diseño y desarrollo de hardware, se recomienda utilizar Linux ya que permite ocupar hasta 8 procesadores lógicos del computador para el proceso de síntesis, implementación y creación del Bitstream, en contraste Windows solamente utiliza dos, esto hace que el proceso de diseño sea considerablemente más demorado comparado con Linux.

Para desarrolladores que quieran incursionar en el diseño de hardware se recomienda iniciar con el Framework de PYNQ. Para soluciones finales o prototipos más complejos se recomienda el uso de Petalinux ya que permite mayores opciones de personalización del sistema operativo y una mejor administración de los recursos de hardware, ya que todavía no todos están accesibles como Overlays en PYNQ.

Anexo A: Arquitectura MPSoC +UltraScale



Bibliografía

Acosta, I. (2018). *Mobile robot for research on path planning algotithsms: planning system*. 79.

Albaladejo, J., De Andrés, D., Lemus, L., & Salvi, J. (2004). Codesign methodology for computer vision applications. *Microprocessors and Microsystems*, *28*(5-6 SPEC. ISS.), 303–316. https://doi.org/10.1016/j.micpro.2004.03.010

Asqui, L. (2017). *Planificación Y Seguimiento De Caminos De Manera Autónoma Para Robots Móviles Tipo Uniciclo*.

Azzedine, A., Diguet, J. P., & Pillippe, J. L. (2002). Large exploration for HW/SW partitioning of multirate and aperiodic real-time systems. *Hardware/Software Codesign - Proceedings of the International Workshop*, 85–90. https://doi.org/10.1145/774801.774807

Barber, R., Crespo, J., Gómez, C., Hernámdez, A., & Galli, M. (2018). Mobile Robot Navigation in Indoor Environments Geometric, Topological, and Semantic Navigation. *IntechOpen*, *i*(tourism), 25.

Bingo, H. (2018). Development of a Control Target Recognition for Autonomous Vehicle Using FPGA with Python. *Proceedings - 2018 International Conference on Field-Programmable Technology, FPT 2018*, 422–423. https://doi.org/10.1109/FPT.2018.00089

Biookaghazadeh, S., Ren, F., & Zhao, M. (2018). Are FPGAs Suitable for Edge Computing? *The USENIX Workshop on Hot Topics in Edge Computing*. http://arxiv.org/abs/1804.06404

Castellet, D. (2018). *Estudio para el diseño y programación de un algoritmo de planificación de trayectorias para un coche autónomo*. Universitat Politecnica de Catalunya.

Cebollada, S., Payá, L., Flores, M., Peidró, A., & Reinoso, O. (2021). A state-of-the-art review on mobile robotics tasks using artificial intelligence and visual data. *Expert*

Systems with Applications, *167*, 114195. https://doi.org/10.1016/J.ESWA.2020.114195

Choset, H. (2005). Principles of Robot Motion, Theory and Applications. In *The MIT Press* (Vol. 53). http://books.google.de/books/about/Principles_of_Robot_Motion.html?id=S3biKR21i-QC&redir_esc=y

Colomer, J. (2018). Estudio de los Sensores para la Detección de Obstáculos Aplicables a Robots Móviles [Universitat Oberta de Catalunya]. In *Universitat Oberta de Catalunya*. http://openaccess.uoc.edu/webapps/o2/bitstream/10609/80846/6/jacobarTFM0618memoria.pdf

Corrochano, J. A. (2020). *Diseño de los módulos de Percepción y Navegación para un robot móvil autónomo*. Universidad Politécnica de Madrid.

Crockett, L. H., Northcote, D., Ramsay, C., Robinson, F. D., & Stewart, R. W. (2019). *Zynq ® MPSoC With PYNQ and Machine Learning Applications*. Strathclyde Academic Media.

Distributions - ROS Wiki. (n.d.). Retrieved October 2, 2021, from http://wiki.ros.org/Distributions

Espinoza, I., & Zuñiga, C. (2021). *Implementación de un sistema de navegación reactiva-social y telepresencia en el prototipo de un robot móvil diferencial*. Universidad de las Fuerzas Armadas.

Estefo, P., Simmonds, J., Robbes, R., & Fabry, J. (2019). The Robot Operating System: Package reuse and community dynamics. *Journal of Systems and Software*, *151*, 226–242. https://doi.org/10.1016/j.jss.2019.02.024

Florez, J. (2009). *Arquitectura genérica para sistemas de robótica móvil*. http://repositorio.unicauca.edu.co:8080/handle/123456789/479

Gómez, D. (2015). *Desarrollo de una técnica SLAM para ambientes dinámicos tridimensionales* [Universidad Nacional de Colombia]. http://www.bdigital.unal.edu.co/52599/

Gu, M., Guo, K., Wang, W., Wang, Y., & Yang, H. (2016). An FPGA-based real-Time simultaneous localization and mapping system. *2015 International Conference on Field Programmable Technology, FPT 2015*, *61373026*, 200–203. https://doi.org/10.1109/FPT.2015.7393150

Gul, F., Rahiman, W., & Nazli Alhady, S. S. (2019). A comprehensive study for robot navigation techniques. *Cogent Engineering*, *6*(1), 1–25. https://doi.org/10.1080/23311916.2019.1632046

Husky UGV - Outdoor Field Research Robot by Clearpath. (n.d.). Retrieved October 1, 2021, from https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/

Jaramillo, M. (2020). *An alternative trajectory planning for a differential wheeled robot, aimed to unknown environment mapping with minimum energy consumption, based on a simplified dynamic model*. Universidad Nacional de Colombia.

Joseph, T., & Whitaker, L. (2019). *Towards a Prototype Platform for ROS Integrations on a Ground Robot* [University of Arkansas]. https://scholarworks.uark.edu/etd/3178

Kelly, A. (2013). *Mobile Robotics*. Cambridge University Press. https://doi.org/10.1017/CBO9781139381284

Kendall, A., Grimes, M., & Cipolla, R. (2015). *PoseNet: A Convolutional Network for Real-Time 6-DOF Camera Relocalization*. 2938–2946. https://doi.org/10.1109/ICCV.2015.336

Klančar, G., Zdešar, A., Blažič, S., & Škrjanc, I. (2017). Chapter 5 - Sensors Used in Mobile Systems. In G. Klančar, A. Zdešar, S. Blažič, & I. Škrjanc (Eds.), *Wheeled Mobile Robotics* (pp. 207–288). Butterworth-Heinemann. https://doi.org/https://doi.org/10.1016/B978-0-12-804204-5.00005-6

Ko, N. Y., & Kuc, T. Y. (2015). Fusing range measurements from ultrasonic beacons and a laser range finder for localization of a mobile robot. *Sensors (Switzerland)*, *15*(5), 11050–11075. https://doi.org/10.3390/s150511050

Kostavelis, I., Nalpantidis, L., Boukas, E., Aviles, M., Stamoulias, I., Lentaris, G., Diamantopoulos, D., Siozios, K., Soudris, D., & Gasteratos, A. (2014). SPARTAN: Developing a Vision System for Future Autonomous Space Exploration Robots. *Journal of Field Robotics*, *24*(4), 273–274. https://doi.org/10.1002/rob

Kundu, A. S., Mazumder, O., Dhar, A., & Bhaumik, S. (2016). Occupancy grid map generation using 360° scanning xtion pro live for indoor mobile robot navigation. *2016 IEEE 1st International Conference on Control, Measurement and Instrumentation, CMI 2016*, *Cmi*, 464–468. https://doi.org/10.1109/CMI.2016.7413791

Lentaris, G., Stamoulias, I., Soudris, D., & Lourakis, M. (2016). HW/SW codesign and FPGA acceleration of visual odometry algorithms for rover navigation on mars. *IEEE Transactions on Circuits and Systems for Video Technology*, *26*(8), 1563–1577.

https://doi.org/10.1109/TCSVT.2015.2452781

Li, Y., & Shi, C. (2019). Localization and Navigation for Indoor Mobile Robot Based on ROS. *Proceedings 2018 Chinese Automation Congress, CAC 2018*, *1*(1), 1135–1139. https://doi.org/10.1109/CAC.2018.8623225

Magyar, G., Sinčák, P., & Krizsán, Z. (2015). Comparison study of robotic middleware for robotic applications. *Advances in Intelligent Systems and Computing*, *316*(January). https://doi.org/10.1007/978-3-319-10783-7_13

Marín Paniagua, L. J. (2014). *Localización de robots móviles de recursos limitados basada en fusión sensorial por eventos*. Universitat Politecnica de Valencia.

Marosy, G., Kovács, Z., & Horváth, G. (2009). Effective Mars Rover Platform Design with Hardware / Software Co-design. *12th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 7–10.

Matthies, L., Maimone, M., Johnson, A., Cheng, Y., Willson, R., Villalpando, C., Goldberg, S., Huertas, A., Stein, A., & Angelova, A. (2007). Computer vision on Mars. *International Journal of Computer Vision*, *75*(1), 67–92. https://doi.org/10.1007/s11263-007-0046-z

Neto, A. M. (2015). Short-term visual mapping and robot localization based on learning classifier systems and self-organizing maps. *2015 IEEE Intelligent Vehicles Symposium (IV)*, 235–240. https://doi.org/10.1109/IVS.2015.7225692

Nitta, Y., Tamura, S., & Takase, H. (2018). A Study on Introducing FPGA to ROS Based Autonomous Driving System. *Proceedings - 2018 International Conference on Field-Programmable Technology, FPT 2018*, 424–427. https://doi.org/10.1109/FPT.2018.00090

Nitta, Y., Tamura, S., & Takase, H. (2019). ZytleBot : FPGA integrated development platform for ROS based autonomous mobile robot. *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 422–423. https://doi.org/10.1109/FPL.2019.00077

Ohkawa, T., Yamashina, K., Matsumoto, T., Ootsu, K., & Yokota, T. (2016). Architecture exploration of intelligent robot system using ROS-compliant FPGA component. *Proceedings of the 2016 27th International Symposium on Rapid System Prototyping: Shortening the Path from Specification to Prototype, RSP 2016*, 72–78. https://doi.org/10.1145/2990299.2990312

Panigrahi, P. K., & Bisoy, S. K. (2021). Localization strategies for autonomous mobile

robots: A review. *Journal of King Saud University - Computer and Information Sciences*, *xxxx*. https://doi.org/10.1016/j.jksuci.2021.02.015

Podlubne, A., & Gohringer, D. (2019). FPGA-ROS: Methodology to Augment the Robot Operating System with FPGA Designs. *2019 International Conference on Reconfigurable Computing and FPGAs, ReConFig 2019*. https://doi.org/10.1109/ReConFig48160.2019.8994719

Purwanto, Ardilla, F., & Wibowo, I. K. (2017). Design and implementation of fpga soft processor for holonomic robot low level control. *Proceedings - 2016 International Electronics Symposium, IES 2016*, 197–202. https://doi.org/10.1109/ELECSYM.2016.7861001

Qin, C., Ye, H., Pranata, C. E., Han, J., Zhang, S., & Liu, M. (2020). LINS: A Lidar-inertial state estimator for robust and efficient navigation. *IEEE International Conference on Robotics and Automation (ICRA)*, 8899–8905.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., & Ng, A. (2009). ROS: an open-source Robot Operating System. *Workshops at the IEEE International Conference on Robotics and Automation*.

Romanov, A. M., Romanov, M. P., Morozov, A. A., & Slepynina, E. A. (2019). A navigation system for intelligent mobile robots. *Proceedings of the 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering, ElConRus 2019*, 652–656. https://doi.org/10.1109/EIConRus.2019.8657234

Ruiz-Sarmiento, J. R., Galindo, C., Gonzalez-Jimenez, J., Blanco, J. L., & Teatinos, C. De. (2011). Navegación Reactiva de un Robot Móvil usando Kinect. *Actas ROBOT 2011*.

Schenck, W., Horst, M., Tiedemann, T., Gaulik, S., & Möller, R. (2017). Comparing parallel hardware architectures for visually guided robot navigation. *Concurrency Computation*, *29*(4), 1–28. https://doi.org/10.1002/cpe.3833

Smith, C. U., Frank, G. A., & Cuadrado, J. L. (1985). Architecture Design and Assessment System for Software/Hardware Codesign. *Proceedings - Design Automation Conference*, 417–424. https://doi.org/10.1145/317825.317921

Sosa Valverde, M. A. (2020). *Aplicación de sistemas embebidos en la construcción de robots móviles. El caso del club de robótica de la Universidad Técnica de Cotopaxi*. Universidad Técnica de Cotopaxi.

Sugata, Y., Ootsu, K., Ohkawa, T., & Yokota, T. (2017). Acceleration of Publish/Subscribe Messaging in ROS-compliant FPGA Component. *ACM International Conference*

Proceeding Series, 1–6. https://doi.org/10.1145/3120895.3120904

SUMMIT-XL Mobile Robot - Indoor & Outdoor | Robotnik®. (n.d.). Retrieved October 1, 2021, from https://robotnik.eu/products/mobile-robots/summit-xl-en-2/

Teich, J. (2012). Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, *100*(SPL CONTENT), 1411–1430. https://doi.org/10.1109/JPROC.2011.2182009

Thrun, S. (2003). *Robotic mapping: a survey*.

Tsardoulias, E., & Mitkas, A. P. (2017). Robotic frameworks, architectures and middleware comparison. *ArXiv*, *September*.

TurtleBot2. (n.d.). Retrieved October 1, 2021, from https://www.turtlebot.com/turtlebot2/

TurtleBot3. (n.d.). Retrieved October 1, 2021, from https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

Vallvé, J., & Andrade-Cetto, J. (2015). Mobile Robot Exploration with Potential Information Fields. *Robotics and Autonomous Systems*, *69*(1), 68–79. https://doi.org/10.1016/j.robot.2014.08.009

Velásquez, C. (2017). *Desarrollo de Algoritmo de Mezclado de Mapas por Ocupación de Celdas Aplicado a la Navegación y Exploración Colaborativa de Entornos Internos Desconocidos*. Universidad Nacional de Colombia.

Wahab, M. N. A., Nefti-Meziani, S., & Atyabi, A. (2020). A comparative review on mobile robot path planning: Classical or meta-heuristic methods? *Annual Reviews in Control*, *50*(March), 233–252. https://doi.org/10.1016/j.arcontrol.2020.10.001

Xilinx. (2021). *Vitis High-Level Synthesis User Guide 2020.2*. *2*(2020), 1–606. www.xilinx.com

Yamashina, K., Kimura, H., Ohkawa, T., Ootsu, K., & Yokota, T. (2016). CReComp: Automated Design Tool for ROS-Compliant FPGA Component. *Proceedings - IEEE 10th International Symposium on Embedded Multicore/Many-Core Systems-on-Chip, MCSoC 2016*, 138–145. https://doi.org/10.1109/MCSoC.2016.47

Yamashina, K., Ohkawa, T., Ootsu, K., & Yokota, T. (2015). Proposal of ROS-compliant FPGA component for low-power robotic systems. *2nd International Workshop on FPGAs for Software Programmers*, *9808*(Fsp), 98082N.