Proyecto de Grado

Presentado ante la ilustre Universidad de Los Andes como requisito parcial para obtener el Título de Ingeniero de Sistemas

Estudio e Implementación de una Plataforma de Software que permita generar Mapas para la Navegación de un Robot Móvil

Por

Br. Carlos Eduardo Paparoni Bruzual

Tutor: Prof. Dr. Rafael Rivas Estrada

Julio 2015

Estudio e Implementación de una Plataforma de Software que permita generar Mapas para la Navegación de un Robot Móvil

Br. Carlos Eduardo Paparoni Bruzual

Proyecto de Grado — Sistemas Computacionales, 60 páginas

Resumen: Un reto substancial en el desarrollo y masificación de los robots móviles autónomos es el proceso de reconocimiento o modelado de su entorno.
Existen múltiples soluciones de software que permiten el control de bases robóticas, planificación de rutas, localización y creación de mapas de entorno para navegación y ubicación de uno o varios robots en éste.
En este trabajo se presenta la investigación, documentación e implementación de una plataforma de software adecuada para el manejo del hardware robótico disponible en el Laboratorio de Sistemas Discretos, Automatización e Integración (LaSDAI), con el fin de soportar y utilizar el dispositivo Kinect®, desarrollado por la compañía Microsoft®, específicamente mediante el uso del emisor láser y de la cámara infrarroja del mismo, para realizar medidas y generar mapas del entorno que puedan ser utilizados para la localización y navegación de un robot móvil.
Se desarrollará utilizando UML 2.0 (Unified Modeling Language) para el modelado y como guía en su desarrollo, el método PXP que forma parte de los métodos Ágiles de desarrollo.

Palabras clave: Robots Móviles, Construcción de Mapas, Visión por Computador, Kinect, ROS

Este trabajo fue procesado en LaTeX.

Índice general

Índice de cuadros

Capítulo 1

Introduccion

En este capítulo se presenta una descripcion del Laboratorio de Sistemas Discretos, Automatizacion e Integracion (LaSDAI), en donde se llevó a cabo la elaboracion del proyecto. Se definen los antecedentes que son la base para la presentacion del problema así como también el planteamiento de este último, la justificacion del proyecto de grado, los objetivos y la metodología que encaminaron el desarrollo de la solucion del mismo.

1.1. Descripcion de LaSDAI

El Laboratorio de Sistemas Discretos, Automatizacion e Integracion (LaSDAI), se funda en el año 1993 bajo la direccion del Dr. Edgar Chacon con el apoyo del Programa de Nuevas Tecnologías del CONICIT (Consejo Nacional de Investigaciones Científicas y Tecnológicas), bajo el proyecto N^{o}I-22. LaSDAI está adscrito a la Escuela de Ingeniería de Sistemas de la Facultad de Ingeniería de la Universidad de Los Andes. A partir del año 2007 actualiza sus líneas de investigacion y es re-estructurado. Actualmente está conformado por personal docente, estudiantes y colaboradores de la Universidad de Los Andes.[**?**]

1.1.1. Personal

Miembros

- Dr. Eladio Dapena G. (Coordinador)

- Dr. Rafael Rivas Estrada
- Ing. Jose G. Gonzalez
- Dr. Edgar Chacón

Colaboradores

- PhD. Addison Rios
- Dr. José Aguilar
- Dr. José María Armingol (UC3M)
- Dr. Arturo de La Escalera (UC3M)
- Dra. Ana Corrales (UC3M)
- Ing. David Godoy
- Lic. Nadia González
- MSc. Asdrúbal Fernández

1.1.2. Mision

El Laboratorio de Sistemas Discretos, Automatizacion e Integracion, LaSDAI, es un espacio para la docencia, la investigacion y el desarrollo de productos, en las áreas de robótica, automatizacion industrial y vision por computador, con el objeto de coadyuvar en el desarrollo tecnológico del país. LaSDAI, tiene como meta difundir sus resultados y vincularse con el sector productivo nacional, con la consiga de I+D+I Investigacion, Desarrollo e Innovacion. Sus actividades de soporte a la docencia tanto en pregrado como postgrado, junto con el desarrollo de proyectos de investigacion y las labores de extension, constituyen un complemento que integra diferentes aristas para el desarrollo. Las actividades de extension, mediante asesorías y cursos, colaboran a lograr el autofinanciamiento como estrategia de consolidacion de nuestras actividades.

1.1.3. Vision

LaSDAI será un referente nacional, como un Instituto de Docencia, Investigacion y Desarrollo, en las áreas de robótica y automatizacion, de carácter autonomo y autofinanciado.

1.1.4. Objetivos

Objetivo General

El Laboratorio de Sistemas Discretos, Automatizacion e Integracion fue creado con el fin de desarrollar conceptos y herramientas que soporten la construccion de sistemas automáticos en ambientes de produccion continua, tal como son los de la industria de procesos, mediante una generalizacion de las técnicas de los ambientes de manufactura.

La automatizacion en la industria de Procesos Continuos es un proceso complejo, por la cantidad de elementos involucrados estén un proceso de mejoramiento, donde la automatizacion juega un papel esencial. Los paradigmas para la implantacion de una automatizacion integral en una industria compleja no han sido totalmente definidos, pues el mayor énfasis en la comunidad científica a nivel internacional ha sido dado a la industria de manufactura; de aquí la importancia que tiene para el grupo el desarrollo de trabajos de investigacion en el área, así como la formacion de personal.

Objetivos Específicos

- Realizar actividades de Investigacion, Desarrollo e Innovacion (I+D+I), en las áreas de Robótica, Automatizacion Industrial y Vision por Computador.
- Realizar asesorías a la Industria nacional e internacional.
- Divulgacion del conocimiento mediante la docencia, organizacion de eventos, publicacion de resultados, etc.
- Formacion de personal en las áreas a fines a nivel de pregrado, postgrado, doctorado en diversas universidades.
- Desarrollar proyectos en cooperacion con otros centros y laboratorios de investigacion.
- Formacion de personal mediante cursos de extension en empresas.
- Establecer relaciones de cooperacion con el sector productivo nacional mediante el desarrollo de proyectos.
- Participar en eventos científicos nacionales e internacionales.

1.1.5. Antecedentes

El filósofo Immanuel Kant propuso a través de su teoría de la percepcion, que nuestro conocimiento del mundo exterior depende de nuestras formas de percepcion. Así como el cuerpo humano posee, en general, cinco sentidos universalmente conocidos que le ayudan a percibir el entorno que lo rodea, el estudio de los mismos lo ha llevado a investigar y desarrollar maneras de emular estos sentidos de forma artificial, con múltiples propósitos; entre ellos, el de proveer a entidades hechas por el hombre de la habilidad de reconocer el mundo a su alrededor, y en consecuencia, la capacidad de actuar en él.

Nuestra condicion humana nos permite percibir la estructura en tres dimensiones del mundo a nuestro alrededor con aparente facilidad. Por ejemplo, con sólo ver alrededor en una habitacion llena de cosas, usted podría contar e inclusive nombrar a cada uno de los objetos que le rodea; inclusive, podría adivinar la textura de los mismos sin necesidad de hacer uso del sentido del tacto. Así mismo, la percepcion en tres dimensiones le permite juzgar con gran precision la distancia desde su ubicacion actual hasta cada objeto de interés, permitiéndole tocarlo, tomarlo o manipularlo si así lo desea. Esta percepcion, que nosotros como seres humanos llamamos sentido de la vista, se efectúa a través de células especializadas que tienen receptores que reaccionan a estímulos específicos (en este caso, ondas de radiacion electromagnética de longitudes específicas, que se registran como la sensacion de la luz), ubicadas en nuestros ojos.

Si bien la descripcion del sentido de la vista es -o parece ser- sencilla, se trata de un sentido sumamente complejo y de hecho, podría decirse que es uno de los sentidos más importantes para el ser humano, así como el más perfecto y evolucionado.

¿Por qué se habla de complejidad? Szeliski nos explica que, “en parte, es porque la vision es un problema inverso, donde buscamos encontrar variables desconocidas dada informacion insuficiente para especificar totalmente la solucion. Por tanto, debemos recurrir a modelos físicos y probabilísticos para discerner entre soluciones potenciales. Sin embargo, modelar el mundo visual en toda su complejidad es mucho más difícil que, por ejemplo, modelar el tracto vocal que produce sonidos hablados.” [**?**]

Esta complejidad lo hace convertirse en un campo de estudio de gran importancia, cuya denominacion, a los fines de la emulacion mencionada anteriormente, es de la

Vision Artificial, también conocida como Vision por Computador.

El inicio de la vision artificial, desde el punto de vista práctico, fue marcado por Larry Roberts, el cual, en 1961 creó un programa que podía "ver" una estructura de bloques, analizar su contenido y reproducirla desde otra perspectiva, demostrando así a los espectadores que esa informacion visual que había sido mandada al ordenador por una cámara, había sido procesada adecuadamente por él. [**?**]

1.2. Definicion del Problema

LaSDAI, en sus áreas de Vision por Computador y Robótica, desea estudiar alternativas de plataformas de software para poder utilizar robots autonomos, que provean soporte para sistemas de medicion láser, infrarrojo o una combinacion de ambos, mediante los cuales se pueda realizar medidas de distancias y así, poder generar mapas del entorno a través de dichas medidas.

Si bien se cuenta ya con dos plataformas robóticas (denominados "LR1" y "LR2") se carece de una plataforma programática común, con amplio soporte de la comunidad de investigacion en robótica y de conocimiento en LaSDAI. Esta condicion limita sustancialmente la investigacion, el uso y la difusion de tecnologías afines a la robótica y la vision por computadora, dejando de lado este campo de investigacion.

1.3. Justificacion

LaSDAI posee y utiliza dos plataformas robóticas, los cuales cuentan cada uno con interfaces programáticas desarrolladas por separado. Esto, si bien es adecuado para el uso específico de cada plataforma, supone problemas de intercomunicacion e interoperacion, sin mencionar el costo en mantenimiento de dichas interfaces a nivel de código. Por ende, establecer una plataforma de software común para ambos, reduce a lo mínimo necesario la codificacion personalizada para cada plataforma robótica, provee soporte al involucrar a un mayor número de personas y facilita el desarrollo de otras plataformas robóticas derivadas de esta.

1.4. Objetivos

1.4.1. Objetivo General

Investigar y desarrollar documentacion adecuada que permita establecer una plataforma común de software para el manejo y navegacion de robots móviles, que provea soporte a sensores tales como Microsoft Kinect, para obtener datos y realizar mediciones de entorno.

1.4.2. Objetivos Específicos

1. Analizar las alternativas en plataformas de software disponibles para control robótico.
2. Analizar el software disponible para elaboracion de mapas de entorno.
3. Analizar los requerimientos del módulo de creacion de mapas.
4. Generar un mapa de entorno mediante el software.
5. Realizar documentacion de la estructura de los mapas generados mediante el software.
6. Realizar documentacion adecuada y actualizada para la difusion y posterior uso del software.

1.5. Metodología Utilizada

Con la finalidad de llevar a cabo el desarrollo del proyecto de grado de forma eficiente y a la vez incorporar metodologías actuales enfocadas al desarrollo por parte de individuos (como es normalmente el caso en cuanto a proyectos de grado), se estudió el uso del método PSP [**?**] (Personal Software Process) mejorado con prácticas tomadas de los métodos de programacion Ágiles, en particular, el método Extreme Programming, orientado a una sola persona, es decir, PXP [**?**] (Personal eXtreme Programming) integrado con el método Kanban.

Esto se llevó a cabo mediante las siguientes actividades realizadas:

- Recoleccion de requerimientos.

- Planificacion.
- Inicializacion de iteracion
  - Diseño.
  - Implementacion.
  - Pruebas de sistema.
  - Retrospectiva, análisis de resultados.
- Finalizacion de iteracion

Capítulo 2

Marco Teórico

En este capítulo, se describen los fundamentos teóricos necesarios para el entendimiento y comprensión del proyecto. Se da una definición de los conceptos de robótica, robots, visión por computadora y la definición de las tareas realizadas por un robot autónomo para la navegación y ubicación de si mismo en un entorno. Se especifican las características técnicas del sistema Kinect y por último, se define la metodología Ágil para desarrollo de proyectos y los métodos PXP y Kanban para el desarrollo de aplicaciones, creando con esto una base teórica con la finalidad de tener una introducción del hardware y software utilizado, las herramientas de diseño y permitir al lector tener una idea de la naturaleza del contenido del resto del documento.

2.1. Robótica

La Robótica es aquella rama dentro de la Ingeniería que se ocupa de la aplicación de la informática al diseño y al uso de máquinas con el objetivo que de lo que de esto resulte pueda de alguna manera sustituir a las personas en la realización de determinadas funciones o tareas. En palabras más simples, la robótica es la ciencia y la tecnología de los robots, porque básicamente se ocupa del diseño, manufactura y aplicaciones de los robots que crea. En la Robótica se combinan varias disciplinas al mismo tiempo, como la mecánica, la electrónica, la inteligencia artificial, la informática y la ingeniería de control, en tanto, también, por los quehaceres que desempeña, resulta fundamental el

aporte que recibe y extrae de campos tales como el álgebra, los autómatas programables y las máquinas de estados.

2.2. Robots

El término robot alcanza su primera repercusión en la tercera década del siglo pasado, a instancias de R.U.R (robots Universal Rossum), una obra teatral de ciencia ficción escrita por el autor checo Karel Čapek, en la cual por primera vez se hace alusión al concepto de robot, extraído del término checo "robota", que significaba "trabajos forzados".

A su vez, el término "robótica" es acuñado por Isaac Asimov, definiendo a la ciencia que estudia a los robots. Asimov creó también las Tres Leyes de la Robótica, definidas de esta manera:

1. Un robot no puede actuar contra un ser humano o, mediante la inacción, permitir que un ser humano sufra daños.
2. Un robot debe obedecer las órdenes dadas por los seres humanos, salvo que estén en conflictos con la primera ley.
3. Un robot debe proteger su propia existencia, a no ser que esté en conflicto con las dos primeras leyes.

Desde sus comienzos como disciplina y como parte fundamental de la Ingeniería, la Robótica ha estado incansablemente buscando construir artefactos que materialicen el deseo humano de crear seres a su semejanza a quienes poder delegarles tareas, trabajos o actividades por demás pesadas y desagradables de llevar a cabo. Pero y aunque muchos ni se lo esperen, desde tiempos inmemoriales, muy, muy lejos de las computadoras, hubo unas cuantas expresiones de la robótica. Porque por ejemplo, los antiguos egipcios unieron brazos mecánicos a las estatuas de sus dioses y esgrimían que el movimiento de los miembros se llevaba a cabo por obra y gracias de estos, inclusive los griegos construyeron estatuas que operaban con sistemas hidráulicos, los cuales eran utilizados para fascinar a los adoradores de los templos.

Y también, aproximadamente entre los siglos XVII y XVIII, en Europa, se construyeron muñecos mecánicos muy ingeniosos que ostentaban algunas características como las que presentan los robots de la actualidad. En un constante e incansable ensayo a través de los siglos y cuando ya era un hecho la entrada en el nuevo milenio (2000), la empresa Honda Motor Co. Ltda. concretó a Asimo, el primer robot humanoide capaz de desplazarse de forma bípeda e interactuar con las personas.

La historia de la robótica va unida a la construcción de "artefactos", que trataban de materializar el deseo humano de crear seres a su semejanza y que lo descargasen del trabajo. El ingeniero español Leonardo Torres Quevedo (que construyó el primer mando a distancia para su automóvil mediante telegrafía sin hilo, el ajedrecista automático, el primer transbordador aéreo y otros muchos ingenios) acuñó el término "automática" en relación con la teoría de la automatización de tareas tradicionalmente asociadas.

2.2.1. Clasificación de los robots

En términos generales, un robot se clasifica por sus capacidades, así como también su área de operación, sus grados de autonomía o el fin con el que han sido construidos. Sin embargo, también pueden clasificarse en términos de la era en la que fue implementado, según su forma de construcción o la manera como son controlados. A continuación se exponen algunas formas comunes de clasificación:

Según su cronología

La que a continuación se presenta es la clasificación más común:

1ª Generación. Manipuladores. Son sistemas mecánicos multifuncionales con un sencillo sistema de control, bien manual, de secuencia fija o de secuencia variable.

2ª Generación. Robots de aprendizaje. Repiten una secuencia de movimientos que ha sido ejecutada previamente por un operador humano. El modo de hacerlo es a través de un dispositivo mecánico. El operador realiza los movimientos requeridos mientras el robot le sigue y los memoriza.

3ª Generación. Robots con control sensorizado. El controlador es una computadora que ejecuta las órdenes de un programa y las envía al manipulador para que realice los movimientos necesarios.

4ª Generación. Robots inteligentes. Son similares a los anteriores, pero además poseen sensores que envían información a la computadora de control sobre el estado del proceso. Esto permite una toma inteligente de decisiones y el control del proceso en tiempo real.

Según su arquitectura

La arquitectura, es definida por el tipo de configuración general del robot, puede ser metamórfica. El concepto de metamorfismo, de reciente aparición, se ha introducido para incrementar la flexibilidad funcional de un robot a través del cambio de su configuración por el propio robot. El metamorfismo admite diversos niveles, desde los más elementales (cambio de herramienta o de efecto terminal), hasta los más complejos como el cambio o alteración de algunos de sus elementos o subsistemas estructurales.

Los dispositivos y mecanismos que pueden agruparse bajo la denominación genérica del robot, tal como se ha indicado, son muy diversos y es por tanto difícil establecer una clasificación coherente de los mismos que resista un análisis crítico y riguroso. La subdivisión de los robots, con base en su arquitectura, se hace en los siguientes grupos: poliarticulados, móviles, androides, zoomórficos e híbridos.

1. Poliarticulados En este grupo se encuentran los robots de muy diversa forma y configuración, cuya característica común es la de ser básicamente sedentarios (aunque excepcionalmente pueden ser guiados para efectuar desplazamientos limitados) y estar estructurados para mover sus elementos terminales en un determinado espacio de trabajo según uno o más sistemas de coordenadas, y con un número limitado de grados de libertad. En este grupo, se encuentran los manipuladores, los robots industriales, los robots cartesianos y se emplean cuando es preciso abarcar una zona de trabajo relativamente amplia o alargada, actuar sobre objetos con un plano de simetría vertical o reducir el espacio ocupado en el suelo.

2. Móviles Son robots basados en carros o plataformas, dotados de un sistema locomotor de tipo rodante y con gran capacidad de desplazamiento. Siguen su camino por telemando o guiándose por la información recibida de su entorno a través de sus sensores. Estos robots aseguran el transporte de piezas de un punto a otro de una cadena de fabricación. Guiados mediante pistas materializadas a través de la radiación electromagnética de circuitos empotrados en el suelo, o a través de bandas detectadas fotoeléctricamente, pueden incluso llegar a sortear obstáculos y están dotados de un nivel relativamente elevado de inteligencia.
3. Androides Son robots que intentan reproducir total o parcialmente la forma y el comportamiento cinemática del ser humano. Actualmente, los androides son todavía dispositivos muy poco evolucionados y sin utilidad práctica, y destinados, fundamentalmente, al estudio y experimentación. Uno de los aspectos más complejos de estos robots, y sobre el que se centra la mayoría de los trabajos, es el de la locomoción bípeda. En este caso, el principal problema es controlar dinámica y coordinadamente en el tiempo real el proceso y mantener simultáneamente el equilibrio del robot.
4. Zoomórficos Los robots zoomórficos, que considerados en sentido no restrictivo podrían incluir también a los androides, constituyen una clase caracterizada principalmente por sus sistemas de locomoción que imitan a los diversos seres vivos. A pesar de la disparidad morfológica de sus posibles sistemas de locomoción es conveniente agrupar a los robots zoomórficos en dos categorías principales: caminadores y no caminadores. El grupo de los robots zoomórficos no caminadores está muy poco evolucionado. Los experimentos efectuados en Japón basados en segmentos cilíndricos biselados acoplados axialmente entre sí y dotados de un movimiento relativo de rotación. Los robots zoomórficos caminadores multípedos son muy numerosos y están siendo objeto de experimentos en diversos laboratorios con vistas al desarrollo posterior de verdaderos vehículos terrenos, piloteados o autónomos, capaces de evolucionar en superficies muy accidentadas. Las aplicaciones de estos robots serán interesantes en el campo de la exploración espacial y en el estudio de los volcanes.
5. Híbridos Corresponden a aquellos de difícil clasificación, cuya estructura se sitúa

en combinación con alguna de las anteriores ya expuestas, bien sea por conjunción o por yuxtaposición. Por ejemplo, un dispositivo segmentado articulado y con ruedas, es al mismo tiempo, uno de los atributos de los robots móviles y de los robots zoomórficos.

Según su modalidad de control

La modalidad de control se refiere a la dependencia –o no– de un operador humano que instruya órdenes al robot; por tanto, se subdivide en dos grandes grupos:

1. Teledirigidos Se define como robots teledirigidos a aquellos que necesitan la intervención de un operador humano, ya sea en forma parcial o total, por ejemplo los utilizados en la desactivación de explosivos.
2. Autónomos Se les llama autónomos a aquellos robots que son capaces de tomar sus propias decisiones basados en la comprensión del entorno en que se encuentren. Existen numerosos tipos de robots de variadas configuraciones que se encuadran en esta categoría, como es el caso de un brazo robot con visión artificial sin asistencia humana realizando tareas de clasificación de objetos, o de los robots móviles del tipo vehículo.

2.2.2. Robots Autónomos

Un robot autónomo es un robot que realiza comportamientos o tareas con un alto grado de autonomía, que es particularmente deseable en campos tales como la exploración del espacio, la limpieza de suelos, cortar el césped, y el tratamiento de aguas residuales.

Algunos robots de fábricas modernas son "autónomos" dentro de los límites estrictos de su entorno directo. Puede que no sea la existencia de todos los grados de libertad en su entorno, pero el lugar de trabajo del robot de la fábrica es un reto y, a menudo puede contener, variables caóticas e impredecibles. La orientación exacta y la posición del siguiente objeto de trabajo e incluso (en las fábricas más avanzadas) el tipo de objeto y la tarea requerida debe ser determinado. Esto puede variar de manera impredecible (por lo menos desde el punto de vista del robot).

Un área importante de la investigación robótica es permitir que el robot pueda hacer frente a su entorno ya sea en tierra, bajo el agua, en el aire, bajo tierra o en el espacio.

Un robot completamente autónomo puede:

- Obtener información sobre el medio ambiente (Regla #1)
- Trabajar por un período prolongado sin intervención humana (Regla #2)
- Mover todo o parte de sí mismo a través de su entorno operativo sin ayuda humana (Regla #3)
- Evitar situaciones que son perjudiciales para las personas, los bienes, o sí mismo, si esos son parte de sus especificaciones de diseño (Regla #4)

Un robot autónomo también puede aprender o adquirir nuevos conocimientos como ajustarse a nuevos métodos para llevar a cabo sus tareas o adaptarse a un entorno cambiante.

Al igual que otras máquinas, los robots autónomos todavía requieren de un mantenimiento regular.

Ejemplos:

Automantenimiento

El primer requisito para la autonomía física completa es la capacidad de un robot para cuidar de sí mismo. Muchos de los robots que funcionan con baterías en el mercado hoy en día pueden encontrar y conectarse a una estación de carga, y algunos juguetes como Aibo de Sony son capaces de realizar auto-acoplamiento para cargar sus baterías.

El mantenimiento realizado se basa en la "propiocepción", o la capacidad de sentir el propio estado interno. En el ejemplo de carga de la batería, el robot puede decir propioceptivamente que sus baterías están bajas y en consecuencia, buscar el cargador. Otro sensor propioceptivo es común para la supervisión de calor. El aumento de la propiocepción se requerirá para los robots para trabajar de forma autónoma, cerca de la gente y en ambientes hostiles. Las propiocepciones comunes incluyen sensores de detección térmica, óptica y háptica, así como el efecto Hall (eléctrica).

Sintiendo el medio ambiente

La exterocepción es la detección de información del medio ambiente. Los robots autónomos deben tener una gama de sensores ambientales para llevar a cabo su tarea y no meterse en problemas.

Los sensores exteroceptivos comunes incluyen el espectro electromagnético, el sonido, el tacto, química (olor), la temperatura, la distancia hacia múltiples objetos, y la altitud. Algunas cortadoras de césped robóticas adaptarán su programación mediante la detección de la velocidad en la que la hierba crece a medida que sea necesario para mantener un césped perfectamente cortado, y algunos robots de limpieza por aspiración toemem detectores de tierra que detectan la cantidad de suciedad que se está recogido y utilizan esta información para decirles que deben permanecer en un área durante más tiempo.

Desempeño de tareas

El siguiente paso en el comportamiento autónomo es para llevar a cabo realmente una tarea física. Una nueva área que muestra promesa comercial es la de los robots domésticos, con una avalancha de pequeños robots aspiradora que comienzan con iRobot y Electrolux en 2002 Mientras que el nivel de inteligencia no es muy alta en estos sistemas, que navegan en zonas extensas y conducen en situaciones estrechas alrededor de los hogares utilizando sensores de contacto y sin contacto. Ambos robots utilizan algoritmos propietarios para aumentar la cobertura por encima del simple rebote al azar.

El siguiente nivel de ejecución de la tarea autónoma requiere que un robot pueda realizar tareas condicionales. Por ejemplo, los robots de seguridad se pueden programar para detectar intrusos y responder de una manera particular, dependiendo de donde esté ubicado el intruso.

Navegación interior

Para que un robot pueda asociar comportamientos con un lugar (localización) requiere saber dónde está y ser capaz de navegar de punto a punto. Tal navegación

comenzó mediante guía cableada en la década de 1970 y progresó en la década de 2000 a la triangulación mediante balizas. Los robots autónomos comerciales actuales navegan basándose en la detección de caraterísticas naturales.

Los primeros robots comerciales en lograrlo fueron el robot de hospital HelpMate de Pyxus y el robot guardia CyberMotion, ambos diseñados por pioneros en robótica en la década de 1980. Estos robots utilizaban originalmente planos de piso CAD creados manualmente, detección mediante sonar y variaciones de seguimiento de paredes para navegar a través de edificios. La próxima generación, tales como PatrolBot y la silla de ruadas autónoma de MobileRobots, ambos introducidos en 2004, tienen la capacidad de crear sus propios mapas basados en láser de un edificio y navegar por zonas abiertas, así como corredores. Su sistema de control cambia su ruta sobre la marcha si algo bloquea el camino.

Inicialmente, la navegación autónoma se basaba en sensores planares, como los telémetros láser, que sólo pueden realizar detecciones en un plano o nivel. Los sistemas más avanzados ahora fusionan la información de diversos sensores, tanto para la localización (posición) y la navegación. Los sistemas tales como Motivity pueden contar con diferentes sensores en diferentes áreas, dependiendo de lo que proporcione los datos más fiables en el momento, y pueden volver a generar mapas de entorno de forma autónoma.

En lugar de subir escaleras, lo que requiere hardware altamente especializado, la mayoría de los robots de interior navegan en áreas accesibles para minusválidos, controlando ascensores y puertas electrónicas. Con este tipo de interfaces de control de acceso, los robots ya pueden navegar libremente en el interior. Subir o bajar escaleras de forma autónoma y abrir puertas de forma manual, son temas actuales de investigación.

A medida que estas técnicas en interiores se siguen desarrollando, los robots aspiradora adquirirán la capacidad de limpiar una habitación específica designada por el usuario o una planta entera. Los robots de seguridad podrán rodear intrusos de forma cooperativa e incluso cortarles las salidas. Estos avances también traen protecciones asociadas: los mapas internos de los robots permiten típicamente la definición de “zonas prohibidas” para evitar que los mismos entren de forma autónoma en ciertas regiones.

Navegación en exteriores La autonomía al aire libre se logra más fácilmente en el aire, ya que los obstáculos son raros. Los misiles de crucero son robots altamente autónomas y bastante peligrosos. Los aviones no tripulados se utilizan cada vez más para el reconocimiento. Algunos de estos vehículos aéreos no tripulados (UAV) son capaces de volar toda su misión sin ninguna interacción humana en absoluto, excepto posiblemente para el aterrizaje cuando una persona interviene mediante control remoto por radio. Sin embargo, algunos aviones son capaces de realizar aterrizajes automáticos y seguros.

La autonomía al aire libre es más difícil para los vehículos de tierra, debido a:

- La tridimensionalidad del terreno,
- Grandes disparidades en la densidad de superficie
- Exigencias climáticas
- Inestabilidad del medio ambiente detectado

En Estados Unidos, el proyecto MDARS (Mobile Detection Assessment and Response System), que definió y construyó un robot prototipo de vigilancia exterior en la década de 1990, se está llevando a producción y será implementado en 2006. El robot MDARS de General Dynamics puede navegar de forma semi-autónoma y detectar intrusos, utilizando la arquitectura de software MRHA (Multiple Robot Host Architecture) planeada para todos los vehículos militares no tripulados. El robot Seekur fue el primer robot comercial para demostrar las capacidades similares a MDARS para uso general por los aeropuertos, plantas de servicios públicos, instalaciones correccionales y Seguridad Nacional.

Los rovers MER-A y MER-B (conocidos actualmente como los rovers Spirit y Opportunity) pueden encontrar la posición del sol y navegar sus propias rutas a destinos sobre la marcha a través de:

- Cartografía de la superficie mediante visión en 3D
- Cálculo de zonas seguras e inseguras en la superficie dentro de ese campo de visión
- Cálculo de rutas óptimas en toda la zona segura hacia el destino deseado

- Conducción a lo largo de la ruta calculada
- La repetición de este ciclo hasta que el destino se alcanza, o no haya ninguna ruta conocida hacia el destino

El Rover de ESA en planificación, ExoMars Rover, es capaz de localización relativa basada en visión y localización absoluta para navegar de forma autónoma a trayectorias seguras y eficaces a objetivos a través de:

- La reconstrucción de modelos 3D del terreno que rodea al Rover con un par de cámaras estéreo
- La determinación de las zonas seguras e inseguras del terreno y la "dificultad" general para el Rover para navegar por el terreno
- Cálculo de caminos eficientes a través de la zona de seguridad hacia el destino deseado
- Conducir el Rover a lo largo del camino planeado
- La creación de un mapa de navegación de todos los datos de navegación anterior

El DARPA Grand Challenge y DARPA Urban Challenge han alentado el desarrollo de capacidades aún más autónomas para vehículos de tierra, mientras que este ha sido el objetivo demostrado por robots aéreos desde 1990 como parte de la AUVSI Internacional Aerial Robotics Competition.

2.3. SLAM

Para un robot autónomo, capaz de navegar por si mismo en un entorno, una de las tareas más desafiantes que puede realizar es la de estimar su posición y orientación en el ambiente en el cual navega, especialmente si no se cuenta con información previa sobre ese ambiente. De esto se trata SLAM, acrónimo en inglés para Simultaneous Localization And Mapping, o Localización y Mapeo Simultáneos.

2.3.1. Introducción

El problema de construcción de mapas y localización simultáneos pregunta si es posible que un vehículo autónomo comience en una ubicación desconocida dentro de un entorno desconocido y que construya de forma incremental un mapa de este entorno mientras que usa este mapa de forma simultánea para calcular de forma absoluta la localización o ubicación del vehículo.

La solución para este problema es, en múltiples aspectos, el "Santo Grial" de la comunidad de investigación de vehículos autónomos. Por ello, la principal ventaja de SLAM es que elimina la necesidad de infraestructuras artificiales o de poseer conocimiento topográfico a priori del entorno.

2.3.2. Historia

El problema general de SLAM ha sido objeto de considerable investigación desde el inicio de una comunidad de investigación de la robótica y de hecho antes de este en áreas como sistemas de navegación de vehículos tripulados y estudios geofísicos. Se han propuesto varios enfoques para abordar tanto el problema de SLAM y también problemas de navegación más simplificados donde se hacen disponible informaciones adicionales de mapa o de ubicación del vehículo.

En términos generales, estos enfoques adoptan una de tres filosofías principales. La más popular de ellas es la aproximación estimación-teoría o basada en el filtro de Kalman. La popularidad de este enfoque se debe a dos factores principales. En primer lugar, proporciona directamente tanto una solución recursiva para el problema de navegación y de una manera de computar estimaciones consistentes para la incertidumbre en ubicaciones de vehículos y referencias de mapas sobre la base de modelos estadísticos para el movimiento del vehículo y observaciones relativas de referencias. En segundo lugar, un corpus sustancial del método y la experiencia ha sido desarrollado en el sector aeroespacial, marítimo y otras aplicaciones de navegación, de las que la comunidad autónoma de vehículos puede extraer.

Una segunda filosofía es la de evitar la necesidad de estimaciones de posición absoluta y de medidas precisas de la incertidumbre y en lugar de ello, emplear

conocimiento más cualitativo de la ubicación relativa de las referencias y el vehículo para construir mapas y guiar el movimiento. El enfoque cualitativo para la navegación y el problema general de SLAM tiene muchas ventajas potenciales sobre la metodología de estimación-teoría en términos de limitar la necesidad de modelos precisos y los requisitos computacionales resultantes, y en su significativo "atractivo antropomórfico".

La tercera y muy amplia filosofía, elimina el filtro de Kalman o el riguroso formalismo estadístico al tiempo que conserva un enfoque esencialmente numérico o computacional para el problema de navegación y SLAM. Tales enfoques incluyen el uso de pareos de lugares de interés icónicos, el registro de un mapa global, regiones delimitadas y otras medidas para describir la incertidumbre. Un trabajo notable en esta filosofía se ha realizado mediante el uso de un enfoque bayesiano para mapeo de edificios que no asume las distribuciones de probabilidad de Gauss como es requerido por el filtro de Kalman. Esta técnica, aunque muy eficaz para la localización con respecto a los mapas, no se presta para proporcionar una solución gradual a SLAM, donde un mapa se construye gradualmente a medida que se recibe información de los sensores. [**?**]

2.4. Visión por Computadora

Es el campo de la Inteligencia Artificial enfocado a que las computadoras puedan extraer información a partir de imágenes, ofreciendo soluciones a problemas del mundo real. La visión para los humanos no es ningún problema, pero para las máquinas es un campo muy complicado. Influyen texturas, luminosidad, sombras, objetos complejos, etc.

El propósito de la visión artificial o por computadora, es programar un computador para que "entienda" una escena o las características de una imagen.

Los objetivos típicos de la visión artificial incluyen:

- La detección, segmentación, localización y reconocimiento de ciertos objetos en imágenes (por ejemplo, caras humanas).
- La evaluación de los resultados (por ejemplo, segmentación, registro).

- Registro de diferentes imágenes de una misma escena u objeto, es decir, hacer concordar un mismo objeto en diversas imágenes.
- Seguimiento de un objeto en una secuencia de imágenes.
- Mapeo de una escena para generar un modelo tridimensional de la escena; este modelo podría ser usado por un robot para navegar por la escena.
- Estimación de las posturas tridimensionales de humanos.
- Búsqueda de imágenes digitales por su contenido.

Estos objetivos se consiguen por medio de reconocimiento de patrones, aprendizaje estadístico, geometría de proyección, procesamiento de imágenes, teoría de grafos y otros campos. La visión artificial cognitiva está muy relacionada con la psicología cognitiva y la computación biológica.

2.5. Microsoft Kinect

Kinect para Xbox 360, o simplemente Kinect (originalmente conocido por el nombre en clave "Project Natal"), es un controlador de juego libre y entretenimiento creado por Alex Kipman, desarrollado por Microsoft para las videoconsolas Xbox 360 y Xbox One, y desde junio del 2011 para PC a través de Windows 7 y Windows 8.

Kinect permite a los usuarios controlar e interactuar con la consola sin necesidad de tener contacto físico con un controlador de videojuegos tradicional, mediante una interfaz natural de usuario que reconoce gestos, comandos de voz, y objetos e imágenes. El dispositivo tiene como objetivo primordial aumentar el uso de la Xbox 360, más allá de la base de jugadores que posee en la actualidad.

En sí, Kinect compite con los sistemas Wiimote con Wii MotionPlus y PlayStation Move, que también controlan el movimiento para las consolas Wii y PlayStation 3, respectivamente.

El nombre en clave "Proyecto Natal" responde a la tradición de Microsoft de utilizar ciudades como nombres en clave. Alex Kipman, director de Microsoft, quien incubó el proyecto, decidió ponerle el nombre de la ciudad brasileña Natal como un homenaje a su país de origen y porque la palabra natal significa "de o en relación al nacimiento",

lo que refleja la opinión de Microsoft en el proyecto como “el nacimiento de la próxima generación de entretenimiento en el hogar”. Poco antes de la E3 2010 varios weblogs tropezaron con un anuncio que supuestamente se filtró en el sitio italiano de Microsoft de que sugirió el título “Kinect”, que confirmó más tarde.

2.5.1. Características

El sensor de Kinect es una barra horizontal de aproximadamente 23 cm (9 pulgadas) conectada a una pequeña base circular con un eje de articulación de rótula, y está diseñado para ser colocado longitudinalmente por encima o por debajo de la pantalla de vídeo.

El dispositivo cuenta con una cámara RGB, un sensor de profundidad, un micrófono de múltiples matrices y un procesador personalizado que ejecuta el software patentado, que proporciona captura de movimiento de todo el cuerpo en 3D, reconocimiento facial y capacidades de reconocimiento de voz.

El sensor contiene un mecanismo de inclinación motorizado y en caso de usar una PC o un modelo original de Xbox 360, tiene que ser conectado a una toma de corriente, ya que la corriente que puede proveerle el cable USB es insuficiente; para el caso del modelo de Xbox 360 S esto no es necesario ya que esta consola cuenta con una toma especialmente diseñada para conectar el Kinect y esto permite proporcionar la corriente necesaria que requiere el dispositivo para funcionar correctamente.

El sensor de profundidad es un proyector de infrarrojos combinado con un sensor CMOS monocromo que permite a Kinect ver la habitación en 3D en cualquier condición de luz ambiental. El rango de detección de la profundidad del sensor es ajustable gracias al software de Kinect capaz de calibrar automáticamente el sensor.

2.5.2. Especificaciones Técnicas

Sus especificaciones más relevantes son:

- Cámara / sensor infrarrojo: Microsoft / X853750001 / VCA379C7130 / MT9M001

- Rango efectivo (aproximado): 1,2m – 3,5m (puede ser menor o mayor, dependiendo de las condiciones ambientales)
- Resolución: 640x480 píxeles @ 30 Hz (profundidad de 11 bits: 2048 niveles de sensibilidad)
- Cámara RGB: VNA38209015 / MT9M112 / MT9v112
- Resolución: 640x480 píxeles @ 30 Hz (VGA de 8 bits)
- Emisor / proyector infrarrojo: OG12 / 0956 / D306 / JG05A
- Proyector láser de 830 nm.
- Potencia de salida: 60m̃W

2.5.3. Requerimientos para uso en Robótica

Llenar.

2.6. Desarrollo Ágil de Proyectos

En todo proyecto, el objetivo deseado es (o debería ser) el de maximizar la eficiencia y disminuir el costo, ya sea monetario o de tiempo. Para lograr este objetivo se han propuesto diferentes formas de administrar o gerenciar los proyectos, con infinidad de niveles de control, desde prácticamente ningún control en lo absoluto, hasta la microgerencia de las tareas más mínimas posibles.

El desarrollo de un trabajo especial de grado, tesis, o proyecto de grado puede verse, tal como se denominó por último, como un proyecto en si mismo, donde el objetivo es cumplir con los objetivos generales y específicos dentro de un marco de tiempo determinado, por lo que puede aplicarse lo mencionado al inicio para cumplir con los objetivos de la mejor manera posible.

Así pues, en el área de la ingeniería de sistemas y el desarrollo de software, se han empleado igualmente innumerables métodos para llevar a cabo la gerencia de un proyecto; sin que esto pretenda convertirse en un estudio a profundidad de dichos métodos, se nombra a continuación uno de los más populares en la actualidad, junto algunos métodos que emplean su filosofía y cuyo fin no es otro que el de organizar el

desarrollo del proyecto y así poder rendir cuentas del mismo.

Las metodologías Ágiles de desarrollo, fueron propuestas en el año 2001 por diversos representantes de múltiples metodologías de desarrollo, tales como SCRUM, Programación Extrema, DSOM, Desarrollo de Software Adaptativo, Crystal y otros, con el fin de encontrar un terreno común que involucrara una alternativa al desarrollo de software pesado y orientado a la documentación.

Su idea consistió en enfocar el proceso de desarrollo en las personas en vez de al proceso en sí, reduciendo la burocracia y los procesos de planificación, documentación y modelado al mínimo posible, a través de principios documentados en un "manifiesto".

2.6.1. Principios del Desarrollo Ágil

El manifiesto para el Desarrollo Ágil de Software, indica el enfoque a seguir por esta metodología. Sus valores principales son los siguientes:

- Individuos e interacciones sobre procesos y herramientas
- Software funcionando sobre documentación extensiva
- Colaboración con el cliente sobre negociación contractual
- Respuesta ante el cambio sobre seguir un plan

En las propias palabras del manifiesto: "Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda".[**?**]

Los principios producto de estos valores son:[**?**]

- Nuestra mayor prioridad es satisfacer al cliente mediante la entrega temprana y continua de software con valor.
- Aceptamos que los requisitos cambien, incluso en etapas tardías del desarrollo. Los procesos Ágiles aprovechan el cambio para proporcionar ventaja competitiva al cliente.
- Entregamos software funcional frecuentemente, entre dos semanas y dos meses, con preferencia al periodo de tiempo más corto posible.
- Los responsables de negocio y los desarrolladores trabajamos juntos de forma cotidiana durante todo el proyecto.

- Los proyectos se desarrollan en torno a individuos motivados. Hay que darles el entorno y el apoyo que necesitan, y confiarles la ejecución del trabajo.
- El método más eficiente y efectivo de comunicar información al equipo de desarrollo y entre sus miembros es la conversación cara a cara.
- El software funcionando es la medida principal de progreso.
- Los procesos Ágiles promueven el desarrollo sostenible. Los promotores, desarrolladores y usuarios debemos ser capaces de mantener un ritmo constante de forma indefinida.
- La atención continua a la excelencia técnica y al buen diseño mejora la Agilidad.
- La simplicidad, o el arte de maximizar la cantidad de trabajo no realizado, es esencial.
- Las mejores arquitecturas, requisitos y diseños emergen de equipos auto-organizados.
- A intervalos regulares el equipo reflexiona sobre cómo ser más efectivo para a continuación ajustar y perfeccionar su comportamiento en consecuencia.

2.7. Personal Extreme Programming

La programación extrema o *eXtreme Programming* (XP) es una metodología de desarrollo de la ingeniería de software formulada por Kent Beck, autor del primer libro sobre la materia, *Extreme Programming Explained: Embrace Change* (1999). Es el más destacado de los procesos ágiles de desarrollo de software. Al igual que éstos, la programación extrema se diferencia de las metodologías tradicionales principalmente en que pone más énfasis en la adaptabilidad que en la previsibilidad.

Los defensores de XP consideran que los cambios de requisitos sobre la marcha son un aspecto natural, inevitable e incluso deseable del desarrollo de proyectos. Creen que ser capaz de adaptarse a los cambios de requisitos en cualquier punto de la vida del proyecto es una aproximación mejor y más realista que intentar definir todos los requisitos al comienzo del proyecto e invertir esfuerzos después en controlar los cambios en los requisitos.

Se puede considerar la programación extrema como la adopción de las mejores metodologías de desarrollo de acuerdo a lo que se pretende llevar a cabo con el proyecto, y aplicarlo de manera dinámica durante el ciclo de vida del software.

2.7.1. Características

Las características fundamentales del método son:

- Desarrollo iterativo e incremental: pequeñas mejoras, unas tras otras.
- Pruebas unitarias continuas, frecuentemente repetidas y automatizadas, incluyendo pruebas de regresión. Se aconseja escribir el código de la prueba antes de la codificación. Véase, por ejemplo, las herramientas de prueba JUnit orientada a Java, DUnit orientada a Delphi, NUnit para la plataforma.NET o PHPUnit para PHP. Estas tres últimas inspiradas en JUnit, la cual, a su vez, se inspiró en SUnit, el primer framework orientado a realizar tests, realizado para el lenguaje de programación Smalltalk.
- Programación en parejas: se recomienda que las tareas de desarrollo se lleven a cabo por dos personas en un mismo puesto. La mayor calidad del código escrito de esta manera -el código es revisado y discutido mientras se escribe- es más importante que la posible pérdida de productividad inmediata. En este sentido, la modalidad personal difiere por razones obvias; sin embargo, las demás características son perfectamente aplicables.
- Frecuente integración del equipo de programación con el cliente o usuario. Se recomienda que un representante del cliente trabaje junto al equipo de desarrollo.
- Corrección de todos los errores antes de añadir nueva funcionalidad. Hacer entregas frecuentes.
- Refactorización del código, es decir, reescribir ciertas partes del código para aumentar su legibilidad y mantenibilidad pero sin modificar su comportamiento. Las pruebas han de garantizar que en la refactorización no se ha introducido ningún fallo.
- Propiedad del código compartida: en vez de dividir la responsabilidad en el desarrollo de cada módulo en grupos de trabajo distintos, este método promueve

el que todo el personal pueda corregir y extender cualquier parte del proyecto. Las frecuentes pruebas de regresión garantizan que los posibles errores serán detectados.

- Simplicidad en el código: es la mejor manera de que las cosas funcionen. Cuando todo funcione se podrá añadir funcionalidad si es necesario. La programación extrema apuesta que es más sencillo hacer algo simple y tener un poco de trabajo extra para cambiarlo si se requiere, que realizar algo complicado y quizás nunca utilizarlo.

La simplicidad y la comunicación son extraordinariamente complementarias. Con más comunicación resulta más fácil identificar qué se debe y qué no se debe hacer. Cuanto más simple es el sistema, menos tendrá que comunicar sobre éste, lo que lleva a una comunicación más completa, especialmente si se puede reducir el equipo de programadores.

2.8. Kanban

Kanban, del japonés かんばん(看板), que es literalmente "letrero" o "cartelera", es un sistema de programación para la producción *Lean* y justo-a-tiempo.

Kanban se basa en una idea muy simple: el trabajo en curso (Work In Progress, WIP) debería limitarse, y sólo deberíamos empezar con algo nuevo cuando un bloque de trabajo anterior haya sido entregado o ha pasado a otra función posterior de la cadena. El Kanban (o tarjeta señalizadora) implica que se genera una señal visual para indicar que hay nuevos bloques de trabajo que pueden ser comenzados porque el trabajo en curso actual no alcanza el máximo acordado.

Kanban usa un mecanismo de control visual para hacer seguimiento del trabajo conforme este viaja a través del flujo de valor. Típicamente, se usa un panel o pizarra con notas adhesivas o un panel electrónico de tarjetas. Las mejores prácticas apuntan probablemente al uso de ambos.

Las metodologías Ágiles han obtenido buenos resultados proporcionando transparencia respecto al trabajo en curso y completado, así como en el reporte de métricas como la velocidad (cantidad de trabajo realizada en una iteración). Kanban

sin embargo va un paso más allá y proporciona transparencia al proceso y su flujo. Kanban expone los cuellos de botella, colas, variabilidad y desperdicios. Todas las cosas que impactan al rendimiento de la organización en términos de la cantidad de trabajo entregado y el ciclo de tiempo requerido para entregarlo. Kanban proporciona a los miembros del equipo y a las partes interesadas visibilidad sobre los efectos de sus acciones (o falta de acción). De esta forma, los casos de estudios preliminares están demostrando que Kanban cambia el comportamiento y motiva a una mayor colaboración en el trabajo. La visibilidad de los cuellos de botella, desperdicios y variabilidades y su impacto también promueve la discusión sobre la posibles mejoras, y los equipos comienzan rápidamente a implementar mejoras en su proceso.[**?**]

2.8.1. Trello

Trello es una plataforma de software que utiliza el paradigma Kanban para el manejo de proyectos. Iniciado en el año 2011 por la compañía Fog Creek Software y caracterizado como software de productividad, este utiliza un sistema de tableros, listas y tarjetas para llevar el control de múltiples tipos de proyecto, sin estar limitado a proyectos de desarrollo de software, por lo cual es altamente versatil. Las tarjetas aceptan comentarios, archivos adjuntos, votos, fechas de entrega y listas de verificación.

Para este proyecto, se abrió un tablero en Trello accesible desde la dirección `https://trello.com/b/yIMdcTCR/tesis-ros-kinect`, cuyas listas representan un híbrido entre las disponibles en PXP y Kanban, de la siguiente manera:

- Pendientes: Tareas que están por ser ejecutadas.
- Actuales: Tareas que están siendo ejecutadas actualmente.
- Entregadas: Tareas que están marcadas como realizadas, pero que no han sido verificadas aún.
- Rechazadas: Tareas que, habiendo sido entregadas, presentan alguna falla o requieren algún cambio, por lo cual se colocan en esta lista para ser revisadas.
- Listas: Tareas que han sido verificadas y aceptadas.
- Ideas: Tarjetas que representan ideas aplicar, o tareas que no pueden pasar a la lista “Pendientes” por no haber recursos disponibles para ellas.

—-Insertar imagen del tablero de Trello del proyecto de grado—-

Uso

Cada tarea a realizar se coloca como una tarjeta en la lista correspondiente a “Pendientes” o “Ideas” según sea el caso, y una vez estén disponibles los recursos para tomarla (siendo estos recursos tiempo, o disponibilidad de la o las personas involucradas) se pasa dicha tarjeta, mediante arrastrar y colocar, a la lista de “Actuales”. Esto permite ver el flujo de trabajo, tal como se mencionaba anteriormente, y permite saber el estado actual del proyecto si la tabla se mantiene actualizada.

Capítulo 3

Software de Control Robótico

Este capítulo pretende ahondar en las características de las plataformas o software de control robótico, comenzando por su definición, la exposición de las características más comúnes, los criterios bajo los cuales se selecciona una lista de posibles plataformas para el desarrollo del proyecto y finalmente, la justificación de la plataforma de software escogida.

3.1. Definición

Por lo general, cuando se piensa en robótica el enfoque se realiza sobre los componentes de hardware: actuadores, motores, sensores, etc., y se piensa muy poco sobre el software de control que se utilizará.

Podemos comenzar por definir dicho software como el conjunto de módulos o programas que se encarga indicarle al hardware del robot qué tareas o funciones debe realizar. Dichas funciones son altamente variadas y van desde el control de los actuadores o sensores hasta el manejo de algoritmos de navegación autónoma o reconocimiento del terreno.

Definir una plataforma común de software para múltiples robots es un reto porque, a menos que sean robots construidos en masa y con un propósito igualmente común, normalmente no hay dos robots iguales, o con la misma estructura y/o componentes. No obstante, la forma de crear una aplicación para un robot no es distinta de la empleada

para crear cualquier otro tipo de aplicación: un desarrollador escribe la aplicación en determinado lenguaje, lo compila enlazado con las librerías adecuadas para ser ejecutado en determinado sistema operativo y por último, ejecutarlo en el computador que controla al robot.

3.1.1. Software de Robots

Si bien el proceso de desarrollo de software para robots posee muchas características comúnes con el desarrollo de una aplicación común, éstos cuentan con requerimientos específicos que deben ser tomados en cuenta. Desarrollar software para robots móviles es una tarea compleja, ya que un robot es un sistema complejo en si mismo y además este proceso suele ser más exigente que la creación de una aplicación "estándar" como manejadores de bases de datos, *suites* ofimáticas, etc. Los siguientes son algunos condicionantes propios, que diferencian el desarrollo de software robótico de otros ámbitos:

1. A diferencia de una aplicación estándar, una aplicación robótica está directamente enlazada a la realidad física, donde dicho enlace es llevado a cabo a través de sensores y actuadores, donde los primeros obtienen información del entorno y los segundos la modifican. Por tanto, el programa debe poder responder a cualquier cambio en el entorno de forma ágil, para así poder enviar ajustes a los actuadores rápidamente. Esto obliga a que la actuación del software entre en la categoría de tiempo real, como mínimo *soft* o blando, si no estricto.
2. Por lo general, un robot debe realizar múltiples tareas a la vez: monitorear sensores, actualizar la interfaz gráfica con el usuario, enviar órdenes a los actuadores, comúnicar datos a otros procesos, procesar datos, etc., por lo cual se requiere concurrencia y esto añade complejidad, en mayor o menor grado.
3. Si bien esto no es imprescindible (en especial en robots autónomos), el software que es ejecutado en el robot debe actualizar constantemente la interfaz gráfica con el usuario, ya que es una herramienta muy útil para realizar depuración de datos y comportamiento y para visualizar en tiempo de ejecución, las estructuras y variables internas, tales como representaciones del mundo, mapas, estados, etc.

4. Siguiendo los actuales esquemas de desarrollo de software, el software de robots es cada vez más distribuido. Es usual que las aplicaciones de robots tengan que establecer alguna comúnicación con otros procesos ejecutándose en la misma máquina o en una diferente. La distribución ofrece posibilidades ventajosas como ubicar la carga computacional en nodos con mayor capacidad o la visualización remota; y en sistemas multirrobot, hace posible la integración sensorial, la centralización y la coordinación.
5. La diversidad que existe, tanto en hardware como software, hace más compleja la tarea de programación. En cuanto al hardware, cada día se hace mayor la gran diversidad de dispositivos sensoriales y de actuación, y por lo tanto de interfaces; esto obliga al programador a dominarlos para acceder a ellos desde las aplicaciones. Por otro lado, mientras que en muchos campos de la informática sí hay bibliotecas que un programador puede emplear para construir su propio programa, en el software de robots no hay un marco homogéneo ni hay estándares que propicien la reutilización de código y la integración. En robótica cada aplicación prácticamente ha de construirse desde cero para cada robot concreto.
6. Ya que el campo que estudia el comportamiento de robots, para dividirlo en unidades básicas sigue siendo materia de investigación, no existe una guía universalmente admitida sobre la forma de organizar el código de las aplicaciones de robots para que sean escalables y reutilizables. Cada desarrollador escribe su aplicación combinando ad hoc los bloques de código que puedan existir en su entorno.

3.1.2. Plataformas de Software

De la misma forma como los dispositivos electromecánicos -mejor conocidos como robots- que pretenden controlar, el software escrito para ellos presenta, con el pasar del tiempo, cada vez más complejidad y ofrecen mayor funcionalidad. Para ayudar con ambos aspectos, se ha ido implantando *middleware* que simplifica el desarrollo de nuevas aplicaciones en esas áreas, proporcionando contextos nítidos, estructuras de datos predefinidas, bloques muy depurados de código de uso frecuente, protocolos

estándar de comúnicaciones, mecanismos de sincronización, etc. De forma paralela, a medida que el desarrollo de software para robots móviles ha ido madurando también han ido apareciendo diferentes plataformas *middleware*.

Actualmente, los fabricantes más avanzados incluyen plataformas de desarrollo para simplificar a los usuarios la programación de sus robots. Por ejemplo, ActivMedia ofrece la plataforma ARIA para sus robots Pioneer, PeopleBot, etc.; iRobot ofrecía Mobility para sus B14 y B21; Evolution Robotics vende su plataforma ERSP; y Sony ofrece OPEN-R para sus Aibo.

De igual forma, muchos grupos de investigación han creado sus propias plataformas de desarrollo. Varios ejemplos son la suite de navegación CARMEN de Carnegie Mellon University, OROCOS, Player/Stage/Gazebo (PSG), Miro, JDE, MARIE, etc..

El objetivo fundamental de estas plataformas es hacer más sencilla la creación de aplicaciones para robots, a través de las siguientes características comúnes: uniformar y simplificar el acceso al hardware, ofrecer una arquitectura de software concreta y proporcionar un conjunto de bibliotecas o módulos con funciones de uso común en robótica que el cliente puede reutilizar para programar sus propias aplicaciones.

Este objetivo se intenta lograr mediante las siguientes acciones:

- Abstracción del Hardware El sistema operativo provee acceso a sensores y actuadores, aunque lo hace de forma más definida y compleja, totalmente opuesto a como lo hacen las plataformas. Un ejemplo de esto es el siguiente: si se dispone de un robot Pioneer equipado con un sensor láser SICK, la aplicación puede acceder a sus medidas a través de las funciones de la plataforma ARIA o pedirlas y recogerlas directamente a través del puerto serie. Utilizando ARIA basta invocar un método sobre cierto objeto de una clase y la plataforma se encargará de mantener actualizadas las variables con las lecturas. Utilizando sólo el sistema operativo, la aplicación debe solicitar y recoger periódicamente las lecturas al sensor láser a través del puerto serie, y debe conocer el protocolo del dispositivo para componer y analizar correctamente esos mensajes de bajo nivel.

  De la misma forma se ofrece el acceso abstracto para los actuadores. Por ejemplo, en vez de ofrecer comandos de velocidad para cada una de las dos ruedas motrices de un robot Pioneer, la plataforma Miro ofrece una sencilla interfaz de V-W

(velocidad de tracción y de giro) para la actuación motriz, la cual se encarga de hacer las transformaciones oportunas, de enviar a cada rueda las consignas necesarias para que el robot consiga esas velocidades comandadas de tracción y de giro.

En virtud que hay gran heterogeneidad en cuanto al hardware dentro de la robótica, un primer paso bastante importante para llevar a cabo la reutilización de software es la de homogeneizar el acceso al hardware; esta característica está presente en varias plataformas, aunque cada una lo hace a su manera. En la plataforma ERSP de Evolution Robotics el acceso al bajo nivel recibe el nombre de Hardware Abstraction Layer (HAL), y en Miro, Service Layer. En OPEN-R, en Mobility y en ARIA la API de acceso a los sensores y actuadores viene dada por los métodos de un conjunto objetos. En JDE y en PSG el acceso abstracto al hardware lo marca un protocolo entre las aplicaciones y los servidores.

—parafrasear desde—

- Arquitectura de software

  La arquitectura software de la plataforma de desarrollo fija la manera concreta en la que el código de la aplicación debe acceder a las medidas de los sensores, ordenar a los motores, o utilizar una funcionalidad ya desarrollada.

  Existen muchas alternativas de software para ello: llamar a funciones de biblioteca, leer variables, invocar métodos de objetos, enviar mensajes por la red a servidores, etc. Por ejemplo, la plataforma CARMEN presenta interfaces funcionales, Miro usa la invocación de métodos de objetos distribuidos, TCA requiere del paso de mensajes entre distintos módulos, JDE requiere la activación de procesos y la lectura o escritura de variables. Esta apariencia de las interfaces depende de cómo se encapsule en cada plataforma la funcionalidad ya desarrollada.

  Al escribirse dentro de la arquitectura software de la plataforma, el programa de la aplicación adopta una organización concreta y usualmente un lenguaje determinado. Así, se puede plantear como una colección de objetos (p.e. en OPEN-R), como un conjunto de módulos dialogando a través de la red (p.e. en TCA), como un proceso iterativo llamando a funciones, etc.. Hay plataformas

muy cerradas, que obligan estrictamente a cierto modelo. Por ejemplo, en la plataforma RAI (RWI, 1999) los clientes han de escribirse forzosamente como un conjunto de módulos RAI (hebras sin desalojo), con ejecución basada en iteraciones. Otras arquitecturas software son deliberadamente abiertas, restringen lo mínimo, como PSG o CARMEN.

Las arquitecturas software de las plataformas más avanzadas establecen mecanismos concretos para que la aplicación se pueda distribuir en varias unidades concurrentes. El mecanismo multitarea que ofrece la plataforma envuelve y simplifica la interfaz del kernel subyacente para la multiprogramación, en la cual se apoyan siempre. Al igual que ocurre con la interfaz abstracta de acceso al hardware, la interfaz abstracta de multitarea facilita la portabilidad.

- Funcionalidades de uso común Además de las librerías sencillas de apoyo, como filtros de color y demás, estas funcionalidades engloban técnicas relativamente maduras, ya sea de percepción o de algoritmos de control: localización, navegación local segura, navegación global, seguimiento de personas, habilidades sociales, construcción de mapas, etc.

  La ventaja de integrarlas en la plataforma es que el usuario puede reutilizarlas, enteras o por partes, lo cual permite acortar los tiempos de desarrollo y reducir el esfuerzo de programación necesario para tener una aplicación. Al incluir funcionalidad común, el desarrollador no tiene que repetir ese trabajo y puede construir su programa reutilizándolas, concentrándose en específicos de su aplicación. Adem muy probada, lo cual disminuye el numero de errores en el programa final. La forma concreta en que se reutilizan las funcionalidades depende nuevamente de la arquitectura de software y de cómo se encapsulen en ella: módulos, objetos distribuidos, objetos locales, funciones, etc. Los fabricantes suelen vender esas funcionalidades por separado o incluirlas como valor añadido de su propia plataforma. Por ejemplo, la plataforma ERSP incluye tres paquetes en su arquitectura básica: uno de interacción, navegación y otro de visión. En el módulo de interacción se incluye el reconocimiento del habla y la síntesis de voz, para interactuar de modo verbal con su robot. En el módulo de navegación se incluye la construcción automática de mapas, la localización en ellos y su

utilización para planificar trayectorias.

- Arquitectura cognitiva Se denomina arquitectura cognitiva de un robot a la organización de sus capacidades sensoriales y de actuación para generar un repertorio de comportamientos. Para comportamientos simples casi cualquier organización resulta válida, pero para comportamientos complejos se hace patente la necesidad de una buena organización cognitiva. De hecho, puede llegar a ser un factor crítico: con una buena organización sí se pueden generar ciertos comportamientos y con una mala no, porque la complejidad se vuelve inmanejable. A lo largo de la historia de la robótica han surgido escuelas cognitivas o paradigmas para orientar la organización del

  La división del comportamiento artificial en unidades reutilizables es una cuestión muy complicada. Sin embargo, es considerado que aún es demasiado pronto para buscar estándares ahí y recomiendan cenir por ahora la estandarización exclusivamente al acceso a los sensores y actuadores.

  La relación entre las arquitecturas cognitivas y las de software es multiple. Las arquitecturas cognitivas se materializan en alguna arquitectura software, de manera que los comportamientos generados siguiendo cierto paradigma acaban implementándose con algun programa concreto.

  Las propuestas cognitivas más fiables cuentan con implementaciones prácticas relevantes en arquitecturas software concretas: deliberativas (e.g. SOAR), híbridas (e.g. TCA (Simmons and Apfelbaum, 1998), Saphira, etc.), basadas en comportamientos (subsunción, JDE, etc.), etc. Una buena arquitectura cognitiva favorece la escalabilidad de la plataforma.

  Hay plataformas software debajo de las cuales subyace un modelo cognitivo, pero también hay otras en las que no. No obstante, unas arquitecturas software cuadran mejor con ciertas escuelas cognitivas que con otras. Los sistemas deliberativos clásicos cuadran con la programación lógica, con una descomposición funcional en bibliotecas (módulos especialistas) y con las aplicaciones monohilo con un sólo flujo iterativo de control (sensar-modelar-planificar-actuar). Por el contrario, los sistemas basados en comportamientos cuadran mejor con la programación concurrente, donde se tienen varios procesos funcionando en paralelo y que

colaboran al funcionamiento global. Resulta natural asimilar cada unidad de comportamiento al concepto software de proceso, o incluso al de objeto activo.

- Plataformas de software libre Un gran numero de las plataformas de desarrollo, principalmente las creadas por grupos de investigación, se publican con licencia de software libre, con la idea de contribuir al libre intercambio de conocimiento en el area y con ello al avance de la disciplina robótica.

  Player/Stage/Gazebo (PSG) La plataforma PSG fue creada inicialmente en la Universidad de South California. Está formada por los simuladores Stage y Gazebo, y por el servidor Player al que se conecta el programa de la aplicación para recoger los datos sensoriales o comandar las ordenes a los actuadores. El soporte a robots variados y los simuladores que incorpora la convierten en una plataforma muy completa. Además, cuenta con una creciente comúnidad de desarrolladores, muy activa, que continuamente anade nuevas capacidades y amplía el hardware soportado.

  En PSG los sensores y actuadores se contemplan como si fueran ficheros, dispositivos de modo carácter al estilo de Unix, que se manipulan con cinco operaciones básicas: abrir, cerrar, leer, escribir y configurar. Para usar un sensor, las aplicaciones tienen que abrir el dispositivo, configurarlo y leer de él, cerrándolo al final. De manera análoga se utilizan los actuadores. Cada tipo de dispositivo se define en PSG con una interfaz, de manera que los sensores sónar de algún robot en particular son instancias de la misma interfaz. El conjunto de posibles interfaces presenta una máquina virtual, con toda suerte de sensores y actuadores. El robot concreto instancia las interfaces relativas a los dispositivos realmente existentes.

  PSG tiene un diseño cliente/servidor: las aplicaciones establecen un diálogo por TCP/IP con el servidor Player, que es el responsable de proporcionar las lecturas sensoriales y materializar los comandos de actuación. Además de permitir acceso remoto, este diseno proporciona a las aplicaciones construidas sobre PSG gran independencia de lenguaje y mínimas imposiciones de arquitectura. La aplicación puede escribirse en cualquier lenguaje, y con cualquier estilo, simplemente respetando el protocolo de comúnicaciones con el servidor.

PSG se orienta principalmente a ofrecer una interfaz abstracta del hardware de robots y no a la identificación de bloques comúnes de funcionalidad. No obstante, se puede incorporar cierta funcionalidad adicional con nuevos mensajes del protocolo, y servicios anadidos a Player. Por ejemplo, la localización probabilística se ha añadido una interfaz más, localization, que proporciona multiples hipótesis de localización. Esta nueva interfaz supera a la tradicional, position, que acarrea la posición estimada desde la odometría.

ARIA: Otra plataforma de software libre muy utilizada es ARIA (ActivMedia Robotics Interface for Applications). ARIA está impulsada y mantenida por la empresa Active Media Robotics como interfaz de acceso al hardware de sus robots, pero se distribuye con licencia GPL y por tanto su código fuente está accessible. ARIA ofrece un entorno de programación orientado a objetos, que incluye soporte para la programación multitarea y para las comúnicaciones a través de la red. Las aplicaciones han de escribirse forzosamente en C++. ARIA está soportada e Linux y en Win32 OS, por lo que una misma aplicación escrita sobre su API puede funcionar en robots con uno u otro sistema operativo. Es un claro ejemplo de portabilidad.

En el bajo nivel, ARIA tiene un diseno de cliente/servidor: el robot está gobernado por un controlador que hace las veces de servidor. Ese servidor establece un diálogo a través del puerto serie con la aplicación, escrita utilizando ARIA, que actua como cliente. En ese diálogo al cliente las medidas de ultrasonido, odometría, etc. y se reciben las ordenes de actuación a los motores.

En cuanto al acceso al hardware, ARIA ofrece una colección de clases, que configuran una API articulada. La clase principal ArRobot contiene varios métodos y objetos relevantes asociados. Packet Receiver y Packet Sender están relacionados con el envío y recepción de paquetes por el puerto serie con el servidor. Dentro de la clase ArRangeDevices, se tienen clases más concretas como ArSonarDevice o ArSick cuyos objetos contienen métodos que permiten a la aplicación acceder a las lecturas de los sensores de proximidad (sónares o escáner láser, respectivamente).

A diferencia de otras plataformas orientadas a objetos, los objetos de ARIA

no son distribuidos, están ubicados en la máquina que se conecta físicamente al robot. No obstante, ARIA permite programar aplicaciones distribuidas utilizando ArNetworking para manejar comúnicaciones remotas, que es un recubrimiento de los sockets del sistema operativo subyacente.

En cuanto a la multitarea, las aplicaciones sobre ARIA pueden programarse en monohilo o multihilo. En este ultimo caso ARIA ofrece infraestructura tanto para hebras de usuario (ArPeriodicTask) como para hebras kernel (ArThreads). Las ArThreads son un recubrimiento de las Linux-pthreads o Win32-threads. Para resolver los problemas de sincronización y concurrencia, ofrece mecanismos como los ArMutex, y las ArCondition.

ARIA contiene comportamientos básicos como la navegación segura, sin chocar contra los obstáculos. Pero no incluye funcionalidad común como la construcción de mapas o la localización, que se venden por separado. Por ejemplo, ActivMedia vende el paquete MAPPER para la construcción de mapas, ARNL (Robot Navigation and Localization) para la navegación y localización, y ACTS (ColorTracking Software) para la identificación de objetos por color y su seguimiento.

Miro: Miro es una plataforma basada en objetos distribuidos para la creación de aplicaciones para robots, desarrollada en la Universidad de Ulm y publicada bajo licencia GPL. En cuanto a la distribución de objetos, Miro sigue el estándard CORBA, empleando la implementación TAO de CORBA, así como la librería ACE.

Miro consta de tres niveles: una capa de dispositivos, una capa de servicios y el entorno de clases. La capa de dispositivos (Miro Device Layer) proporciona interfaces en forma de objetos para todos los dispositivos del robot. Es decir, sus sensores y actuadores se acceden a través de los métodos de ciertos objetos, que dependen de la plataforma física. Por ejemplo, la clase RangeSensor define una interfaz para sensores como los sónares, infrarrojos o láser. La clase contiene métodos para desplazar un robot con tracción diferencial. La capa de servicios (Miro Services Layer) proporciona descripciones de los sensores y actuadores como servicios especificados en forma de CORBA IDL (Interface

> Definition Language). De esta manera su funcionalidad se hace accesible remotamente desde objetos que residen en otras máquinas interconectadas a la red, independientemente de su sistema operativo subyacente. El entorno de clases (Miro Class Framework) contiene herramientas para la visualización y la generación de históricos, así como módulos con funcionalidad de uso común en aplicaciones robóticas, como la construcción de mapas, la planificación de caminos o la generación de comportamientos.
>
> Dentro de Miro la aplicación robótica tiene la forma de una colección de objetos, remotos o locales. Cada uno ejecuta en su máquina y se comúnican entre sí a través de la infraestructura de la plataforma. Los objetos se pueden escribir en cualquier lenguaje que soporte el estándar CORBA, ya que Miro no impone ninguno en concreto, aunque ha sido enteramente escrita en C++. —parafrasear hasta—

Por lo visto anteriormente, y dadas sus ventajas, podemos considerar que la mejor opción para unificar el desarrollo de plataformas robóticas en el aspecto de software es hacerlo a través de (*frameworks*) o plataformas de software.

3.2. Criterios de Selección

Antes de mencionar las plataformas consideradas para el desarrollo de este proyecto, es muy importante destacar la forma éstas como serán discriminadas con la finalidad de seleccionar la más adecuada para el uso futuro en LaSDAI.

En primer lugar, y atendiendo a las diversas ventajas que esto implica, se enfoca como uno de los principales requisitos indispensables que la plataforma de software esté desarrollada o posea licencias de Software Libre, ya que esto facilita cualquier desarrollo futuro que desee realizarse en él. La existencia o ausencia de licencias de Software Libre basta como criterio en si mismo.

Después, se espera que dicha plataforma cuente con soporte actual de la comúnidad: es decir, que existan grupos de usuarios en foros y/o listas de correo que estén activas y que respondan a dudas de los usuarios. También se toma en cuenta que exista soporte por parte del equipo de desarrolladores a cargo de la plataforma y que ésta sea

actualizada regularmente. Esto es sumamente importante ya que hace más factible que nuevos desarrolladores puedan obtener respuesta a sus consultas o inconvenientes y que en caso de existir estos últimos, se pueda llegar a corregirlos. Para evaluar este criterio, se toman en cuenta la cantidad de grupos disponibles para consulta, de preferencia a través de foros de ayuda y la última fecha de actualización de la plataforma de software como tal; mientras más reciente, mejor.

Luego, se tiene como requerimiento altamente deseable (mas no excluyente), que dicha plataforma soporte los lenguajes de programación en uso actualmente por LaSDAI, o cuyo uso pueda ser deseable en un futuro no muy lejano. Este criterio se da por existente si la plataforma soporta al menos uno de los lenguajes mencionados.

Además, en virtud que este proyecto está basado en el uso del Kinect como sensor para generación de mapas de entorno, no es de extrañar que este sea un requerimiento con alto peso, ya que al contar con soporte ya existente, hace más sencilla la tarea de llevar a cabo la generación de mapas en un robot móvil. Si la plataforma cuenta con al menos un módulo soportado actualmente por algún desarrollador o grupo de desarrolladores, se coloca este criterio como afirmativo.

Por último, aunque vinculado al segundo requisito, se desea que la plataforma de software a escoger, posea documentación amplia y frecuentemente actualizada, porque sólo un programador sabe cuán difícil es desarrollar en un nuevo entorno sin poder contar con documentación o referencias adecuadas. Este criterio basa su existencia o ausencia en una evaluación propia y por tanto, subjetiva de la documentación, ya que establecer una serie de reglas para la evaluación de este criterio, es una materia de estudio en si mismo.

3.2.1. Evaluación de Criterios

Para llevar a cabo la evaluación de cada plataforma, se procedió de la siguiente forma:

Por cada uno de los elementos en la lista de plataformas (Tabla 3.1), se realizó una búsqueda no-exhaustiva en los motores de búsqueda correspondientes a Google Web (`https://google.com/`) y Google Académico (`https://scholar.google.co.ve/`), buscando a su vez características en la lista de criterios para cada uno, evaluándose en

cuanto a existencia o ausencia de cada parámetro, o en el caso de criterios que podían ser parciales, colocando la simbología correspondiente, tal como se detalló en la sección pasada.

3.3. Plataformas de Software Consideradas

Plataforma de Software	Software Libre	Soportado por la Comunidad	Soporte C / C++ / Python	Soporte Kinect	Documentación Actualizada
CARMEN	✓	✗	✓	✗	O
Microsoft Robotics Developer Studio	✗	✗	✗	✓	✓
MOOS	✓	O	✓	✗	O
OpenRDK	✓	O	✓	✗	✓
OpenRTM-aist	✓	✓	✓	✓	✓
ORCA	✓	✗	✓	✗	✓
Player	✓	O	✓	✓	✓
Robot Construction Kit (RoCK)	✓	O	O	✓	✓
Robot Operating System (ROS)	✓	✓	✓	✓	✓
Yet Another Robot Platform (YARP)	✓	O	✓	O	✓

Leyenda: ✓= sí, O = parcial, ✗= no

Cuadro 3.1: Plataformas de Software consideradas

3.4. Plataforma de Software Seleccionada

Dada la popularidad, la facilidad de uso e independencia de lenguajes del sistema de comúnicación interprocesos, la fácil integración de una amplia gama de herramientas como las de visualización de los datos de movimiento y sensores del robot, implementación de los algoritmos de planificación de ruta y de percepción y controladores de bajo nivel para sensores comúnmente utilizados y las herramientas de administración que permiten el monitoreo y la inspección de mensajes, hacen que ROS sea el candidato perfecto para el uso en proyectos en el área de robótica móvil.

Por último pero no por ello menos importante, ROS cuenta con amplia documentación [**?**] para cualquier nivel de usuario, desde principiante hasta experto, por lo que cualquier nuevo desarrollo puede efectuarse con una curva de aprendizaje leve, desde el punto de vista coloquial de esta frase. De igual forma, también cuenta

con múltiples foros activos donde los mismos usuarios interactúan para consultar y resolver dudas.

Capítulo 4

ROS

En este capítulo se desea profundizar en la definición de ROS (Robot Operating System) como plataforma de software seleccionada para el desarrollo de este proyecto de grado. Asimismo, se describen sus características principales, la arquitectura de software que utiliza, la instalación del mismo en el entorno de desarrollo a utilizar, y los módulos en los cuales se apoya este proyecto para llevar a cabo la generación de mapas de entorno.

4.1. Definición

Robot Operating System (Sistema Operativo de/para Robot) o sencillamente ROS, es, tal como su nombre implica, un sistema operativo para robots, de forma similar a los sistemas operativos para computadores de escritorio o servidores. Desarrollado y mantenido por la empresa Willow Garage desde 2008 hasta 2013, siendo tomada su dirección en ese año por la Fundación de Robótica de Código Abierto, es una colección de herramientas, librerías y convenciones que buscan simplificar la tarea de crear comportamientos de robot robustos y complejos a lo largo de una amplia variedad de plataformas robóticas.

La justificación de porqué hacer esto es porque decididamente, crear software robótico de propósito general y verdaderamente robusto es difícil, ya que si bien para un ser humano algunos problemas son triviales, no lo son en lo absoluto al momento de

tomar en cuenta las grandes variaciones entre instancias de tareas y entornos. Lidiar con estas variaciones es tan complicado que ningún individuo, laboratorio o institución pudiera esperar llevarlo a cabo por su propia cuenta.

Por ello, ROS fue construido desde cero con el fin de alentar el desarrollo de software robótico de forma colaborativa. Un ejemplo de esto es que, un laboratorio podría tener expertos en cartografía o mapeado de interiores y podría contribuir un sistema de excelente calidad para la producción de mapas. Otro grupo podría tener expertos en el uso de mapas para navegar, y otro grupo podría haber descubierto un enfoque de visión por computador que funciona bien para el reconocimiento de objetos pequeños entre el desorden. ROS fue diseñado específicamente para grupos como éstos para colaborar y construir sobre el trabajo del otro. [**?**]

Además, ROS es Software Libre y está distribuido bajo la licencia BSD, permitiendo el desarrollo de proyectos comerciales y no-comerciales. Una característica importante en cuanto a la arquitectura (que se detallará más adelante) es que ROS funciona a través de comunicación entre procesos, sin requerir que los módulos sean enlazados dentro del mismo ejecutable, por lo que cualquier sistema construido usando ROS como base puede tener control detallado sobre las licencias de software que utilicen sus módulos, ya sean GPL, BSD o cualquier otra hasta propietaria. [**?**]

4.2. Características Principales

Se pueden comentar las siguientes:

Comunicación entre pares: los sistemas robóticos complejos con múltiples enlaces podrían tener varios computadores de a bordo (para realizar tareas paralelas) conectados a través de una red. La ejecución de un maestro central podría dar lugar a la congestión severa en un enlace determinado. Usando una comunicación peer-to-peer o entre pares evitaría este problema. En ROS, una arquitectura peer-to-peer acoplado a un sistema de memoria intermedia o buffer y un sistema de búsqueda (un servicio de nombres llamado “maestro” en ROS), le permite a cada componente dialogar directamente con cualquier otro, de forma sincrónica o asincrónica como sea necesario.

Gratuito y de código abierto: Ser una plataforma de código abierto ofrece la reutilización de funciones ya existentes proporcionadas por muchos otros usuarios de ROS. Su código se suministra en repositorios como stacks, o "pilas". Otras personas han desarrollado capacidades sorprendentes para los robots que han sido "de código abierto" y son relativamente fáciles de añadir de forma incremental utilizando el entorno de desarrollo de ROS.

Delgado: Para combatir el desarrollo de algoritmos que se "enredan" o vinculan en un grado mayor o menor con el sistema operativo del robot y, por tanto, son difíciles de reutilizar posteriormente, los desarrolladores de ROS han planificado que los controladores y otros algoritmos sean contenidos en ejecutables independientes. Esto garantiza la máxima reutilización y, sobre todo, mantiene reducido su tamaño. Este método hace que ROS sea fácil de usar y ubica la complejidad en las bibliotecas. Esta disposición también facilita las pruebas unitarias y los sistemas desarrollados puede ser completamente independientes de otro sistema.

Multi-lenguaje: ROS es independiente del lenguaje, y se puede programar en varios lenguajes. La especificación ROS trabaja en la capa de mensajería. Las conexiones *peer-to-peer* se negocian en XML-RPC, que existe en un gran número de lenguajes. Para soportar un nuevo lenguaje, se pueden reenvolver clases C ++ son re-envueltos (lo cual se hizo para el cliente Octave, por ejemplo) o se escriben clases permitiendo que se generen mensajes. Estos mensajes se describen en IDL (*Interface Definition Language*). [**?**]

4.3. Arquitectura

ROS está implementado bajo los siguientes conceptos fundamentales:

- Nodos: Son procesos que realizan cálculos; en el contexto de ROS, este término es intercambiable con "módulo de software" ya que está diseñado para ser altamente modular: un sistema está compuesto típicamente de muchos nodos.
- Mensajes: Los nodos se comunican entre si al pasar mensajes, que no es más que una estructura de datos de tipo estricto. Los tipos de datos soportados pueden

ser estándar (entero, flotante, booleano, etc.), así como también arreglos de estos o constantes. Un mensaje puede estar compuesto por varios mensajes y el nivel de anidamiento al que pueden llegar es arbitrario.

- Tópicos: Un nodo publica un mensaje a través de un tópico, que es sencillamente una cadena de caracteres tal como "odometría" o "mapa". Un nodo que esté interesado en un tipo de dato específico se suscribirá al tópico apropiado. En cualquier momento dado, pueden existir múltiples publicadores o suscriptores de forma concurrente para un tópico particular y un nodo puede publicar o suscribirse a múltiples tópicos. Por lo general, los publicadores y suscriptores no están al tanto de la existencia del otro.
- Servicios: Si bien el modelo publicar-suscribir basado en tópicos es un paradigma de comunicaciones flexible, el esquema de enrutamiento de "emisión" no es apropiado para las transacciones síncronas, lo cual puede simplificar el diseño de algunos nodos. A esto se le llama "servicio" en ROS, definidos por un nombre y un par de mensajes tipados, uno para la petición y otro para la respuesta. Es de notar que, a diferencia de los tópicos, solo un nodo puede anunciar un servicio con un nombre particular; por ejemplo, solamente puede haber un servicio llamado "clasificar_imagen". [**?**]

4.4. Requisitos de Instalación

ROS está organizado en distribuciones, que cuentan cada una con sus respectivos requisitos de instalación. Por consenso, cada mes de mayo se lanza una nueva distribución de ROS, y las distribuciones de años pares son de soporte extendido, con 5 años de soporte. Las distribuciones de años impares son distribuciones regulares, con soporte por 2 años. Igualmente, la plataforma soportada por defecto es Ubuntu Linux, mientras que otros sistemas operativos, tales como OS X, Android, Arch Linux, Debian y Windows están bajo soporte experimental, por parte de la comunidad.

Al momento de la elaboración de este proyecto, la versión instalada es "*Indigo Igloo*" (con soporte LTS o *Long Term Support* – Soporte de Larga Duración), con los siguientes requisitos:

- Ubuntu Saucy (13.10) o Ubuntu Trusty (14.04 LTS)
- C++03
- Boost 1.53
- Lisp SBCL 1.0.x
- Python 2.7
- CMake 2.8.11 [**?**]

Durante el proceso de instalación, se lleva a cabo la satisfacción de dependencias.

4.5. Procedimiento de Instalación

4.5.1. Descripción de Entornos de Desarrollo

Los entornos y dispositivos utilizados para llevar a cabo el proyecto, para pruebas de instalación y/o funcionamiento, son los siguientes:

- Máquina Virtual: Oracle ® VirtualBox VM™, 2 GB RAM, 30 GB disco duro, con Ubuntu Trusty Tahr 14.04.2 de 64 bits.
- Computador Portatil: Lenovo ® Z50-70, procesador Intel ® Core™ i7-4510U, 6 GB RAM, 500 GB disco duro, con Ubuntu Trusty Tahr 14.04.2 de 64 bits.
- Microsoft ® XBOX 360 ® Kinect™

En la máquina virtual se llevó a cabo la comprobación del procedimiento de instalación, ya que es un entorno que permite restaurar a un punto anterior con facilidad, en caso de inconvenientes tales como paquetes mal instalados, etc.

4.5.2. Instalación

El proceso de instalación de ROS está excelentemente documentado en su *wiki* oficial accesible desde `http://wiki.ros.org/ROS/Installation`, por lo cual seguimos los pasos tomando nota de cualquier dependencia faltante o error.

Para comenzar, en el enlace previamente mencionado, hacemos clic en "Indigo installation instructions", puesto que es la versión de soporte extendido y es la compatible con la versión instalada de Ubuntu.

Una vez allí, bajo el apartado "Supported", hacemos clic en "Ubuntu".

En adelante, solamente seguimos los siguientes pasos haciendo énfasis en la versión instalada de Ubuntu, tomados directamente del sitio:

1. Configure sus repositorios de Ubuntu

   Configure sus repositorios de Ubuntu para activar los repositorios "restricted", "universe" y "multiverse". La guía de configuración de repositorios de Ubuntu, disponible en el siguiente enlace `https://help.ubuntu.com/community/Repositories/Ubuntu` (en inglés) detalla adecuadamente los pasos necesarios.

2. Configure el archivo sources.list

   Configure su computador para aceptar software desde packages.ros.org. Esta distribución de ROS **sólo** soporta Saucy (Ubuntu 13.10) y Trusty (Ubuntu 14.04) para paquetes Debian.

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release
       -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
   ```

3. Configure sus llaves

   ```
   sudo apt-key adv --keyserver hkp://pool.sks-keyservers.net --recv-key\\
        0xB01FA116
   ```

4. Instalación Primero debe asegurarse que su índice de paquetes Debian esté actualizado:

   ```
   sudo apt-get update
   ```

   Si está utilizando Ubuntu Trusty 14.04.2 y experimenta problemas con las dependencias durante la instalación de ROS, quizás deba instalar dependencias adicionales del sistema.

No instale estos paquetes si está utilizando 14.04, ya que destruirá su servidor X (gráfico):

```
sudo apt-get install xserver-xorg-dev-lts-utopic mesa-common-dev-lts-
    utopic libxatracker-dev-lts-utopic libopenvg1-mesa-dev-lts-utopic
    libgles2-mesa-dev-lts-utopic libgles1-mesa-dev-lts-utopic libgl1-
    mesa-dev-lts-utopic libgbm-dev-lts-utopic libegl1-mesa-dev-lts-
    utopic
```

No instale los paquetes anteriores si está utilizando 14.04, ya que destruirá su servidor X (gráfico). Alternativamente, intente instalar sólo lo siguiente para corregir problemas de dependencias:

```
sudo apt-get install libgl1-mesa-dev-lts-utopic
```

Existen muchas librerías y herramientas en ROS. Se han provisto cuatro configuraciones por defecto para iniciar. También se pueden instalar paquetes de ROS de forma individual.

- Instalación de Escritorio Completa: (Recomendada): ROS, rqt, rviz, librerías genéricas para robots, simuladores 2D/3D, navegación y percepción 2D/3D Indigo usa Gazebo 2, la cual es la versión por defecto en Trusty y es la recomendada. Si desea actualizar a Gazebo 3 vea las instrucciones en `http://wiki.gazebosim.org/wiki/Install/Gazebo_and_ROS#Gazebo_3.x_series` acerca de cómo actualizar el simulador.

  ```
  sudo apt-get install ros-indigo-desktop-full
  ```

- Instalación de Escritorio: ROS, rqt, rviz, y librerías genéricas para robots.

  ```
  sudo apt-get install ros-indigo-desktop
  ```

- ROS-Base: (Esencial) Las librerías de paquete, generación y comunicación. No incluye herramientas de entorno gráfico.

  ```
  sudo apt-get install ros-indigo-ros-base
  ```

- Paquete Individual: También puede instalar un paquete específico de ROS package (reemplace subguiones con guiones del nombre del paquete):

  ```
  sudo apt-get install ros-indigo-PAQUETE
  ```

  por ejemplo:

  ```
  sudo apt-get install ros-indigo-slam-gmapping
  ```

  Para listar paquetes disponibles, use:

  ```
  apt-cache search ros-indigo
  ```

5. Inicialice rosdep

   Antes de poder utilizar ROS, se debe inicializar rosdep. rosdep permite instalar dependencias del sistema con facilidad para código fuente que desee compilar y es requerido para poder ejecutar algunos componentes centrales en ROS.

   ```
   sudo rosdep init
   rosdep update
   ```

6. Configuración de entorno

   Es conveniente si las variables de entorno de ROS son agregadas automáticamente a su sesión Bash cada vez que se invoca un nuevo terminal:

   ```
   echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
   source ~/.bashrc
   ```

   Si tiene más de una distribución de ROS instalada, /.bashrc sólo debe tomar como fuente el setup.bash para la versión que esté en uso actualmente.

   Si simplemente desea cambiar el entorno del terminal actual, puede escribir:

   ```
   source /opt/ros/indigo/setup.bash
   ```

7. Obtener rosinstall

   rosinstall es una herramienta de línea de comandos frecuentemente utilizada en ROS que es distribuida por separado. Le permite descargar con facilidad muchos árboles fuentes para paquetes ROS con un solo comando.

Para instalar esta herramienta en Ubuntu, ejecute:

```
sudo apt-get install python-rosinstall
```

4.6. Módulos Disponibles para SLAM en ROS

Debido a la naturaleza propia, es improbable, para no decir imposible, poder realizar un listado exhaustivo de todos los módulos disponibles.

Capítulo 5

Generación de Mapas a través de RTAB-Map

En este capítulo se describe el software RTAB-Map, sus características, su proceso de funcionamiento, su instalación y uso en el entorno de desarrollo en sus presentaciones como software independiente o como módulo integrado a ROS y por último, la generación de un mapa de entorno a través del mismo.

5.1. Definición

RTAB-Map (*Real-Time Appearance-Based Mapping* o Cartografía en Tiempo Real Basada en Apariencias) es una aproximación a SLAM mediante Grafo RGB-D basado en un detector de cierre de lazo Bayesiano global. El detector de cierre de lazo usa un modelo de bolsa de palabras para determinar la probabilidad de que una nueva imagen provenga de una ubicación anterior o nueva. Cuando una hipótesis de cierre de lazo es aceptada, una nueva restricción es agregada al grafo del mapa, y un optimizador de grafo minimiza los errores en el mapa.

Un enfoque de manejo de memoria es utilizado para limitar el número de ubicaciones utilizadas para la detección de cierre de lazos y la optimización del grafo, para que las restricciones de tiempo real en entornos de gran escala sean siempre respetadas. RTAB-Map puede ser utilizado por si solo con un Kinect o cámara estéreo operado a mano

para obtener cartografía RGB-D de 6 grados de libertad, o en un robot equipado con un medidor de distancias láser para cartografía de 3 grados de libertad. [**?**]

5.2. Instalación

RTAB-Map soporta instalaciones en Linux, OS X y Windows y puede funcionar de dos maneras: como software independiente (no necesita otros paquetes de software aparte de él mismo y de los controladores de las cámaras a usar) o como un módulo de ROS, en cuyo caso lógicamente requiere que ROS esté instalado de antemano (y su compatibilidad se reduce a la misma de ROS).

5.2.1. Como software independiente

Para instalar el software de forma independiente, tomaremos las instrucciones correspondientes a la instalación en Ubuntu debido a que es el entorno de desarrollo utilizado. Estas instrucciones son tomadas del repositorio del proyecto en Github, a través de la dirección `https://github.com/introlab/rtabmap/wiki/Installation`:

- Con ROS ya instalado en el sistema:

  Si ya está instalado ROS en el sistema (como es el caso en el desarrollo del proyecto), ya algunas dependencias estarán instaladas:

  Dependencias según versión de ROS: Indigo:

  ```
  sudo apt-get install libsqlite3-dev libpcl-1.7-all libfreenect-
      dev libopencv-dev
  ```

  Hydro:

  ```
  sudo apt-get install libsqlite3-dev libpcl-1.7-all ros-hydro-
      libfreenect ros-hydro-opencv2
  ```

Descargue el código fuente de RTAB-Map desde Github:

```
git clone https://github.com/introlab/rtabmap.git rtabmap
cd rtabmap/build
cmake ..
make -j4
make install
```

Ya puede ejecutar la aplicación (llamada "rtabmap").

- Si ROS no está instalado:

  Dependencias del sistema:

  ```
  sudo apt-get install libsqlite3-dev libpcl-1.7-all libopencv-dev
  ```

  Para instalar libpcl-1.7-all, es posible que deba agregar los repositorios de ROS (en este caso particular, de la distribución de ROS compatible con la distribución de Ubuntu) realizando los siguientes pasos:

  ```
  sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(
     lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.
     list'
  wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
  sudo apt-get update
  ```

  Si desea habilitar las características SURF/SIFT (SURF: *Speeded-Up Robust Features* – Características Robustas Aceleradas; SIFT: *Scale-Invariant Feature Transform* – Transformación de Características Invariantes en Escala) en RTAB-Map, deberá descargar y generar OpenCV desde el código fuente para tener acceso al módulo no-libre/privativo:

```
cd opencv
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j4
sudo make install
```

Descargue el código fuente de RTAB-Map desde Github: obtenga la última versión o el código fuente actual

```
git clone https://github.com/introlab/rtabmap.git rtabmap
cd rtabmap/build
cmake ..
make -j4
sudo make install
```

Ya puede ejecutar la aplicación (llamada "rtabmap").

- Actualizar el código fuente de RTAB-Map

  Si desea incorporar los últimos cambios después de realizar el "git clone" puede actualizarlo de la siguiente forma:

```
cd rtabmap
git pull origin master
cd build
cmake ..
make -j4
sudo make install
```

5.2.2. Como módulo de ROS

Ubuntu cuenta con binarios para las versiones Hydro e Indigo; basta con ejecutar el siguiente comando, según sea la distribución de ROS:
ROS Hydro:

```
sudo apt-get install ros-hydro-rtabmap-ros
```

ROS Indigo:

```
sudo apt-get install ros-indigo-rtabmap-ros
```

Si se desea instalar desde fuente, el proceso (detallado en la página `https://github.com/introlab/rtabmap_ros#rtabmap_ros`) conlleva tener conocimiento del espacio de trabajo (*workspace*) de ROS, y la instalación desde fuente de la librería OpenCV. También se asume que se ha configurado el espacio de trabajo en el directorio `~/catkin_ws` y que el archivo `~/.bashrc` contiene lo siguiente:
ROS Hydro:

```
source /opt/ros/hydro/setup.bash
source ~/catkin_ws/devel/setup.bash
```

ROS Indigo:

```
source /opt/ros/indigo/setup.bash
source ~/catkin_ws/devel/setup.bash
```

Luego, se procede a descargar el código fuente de RTAB-Map desde Github (**NOTA:** No descargar dentro del espacio de trabajo) e instalarlo dentro del directorio `devel` en el espacio de trabajo, ejecutando lo siguiente:

```
cd ~
git clone https://github.com/introlab/rtabmap.git rtabmap
```

```
cd rtabmap/build
cmake -DCMAKE_INSTALL_PREFIX=~/catkin_ws/devel ..
make -j4
sudo make install
```

Ahora puede instalar el ros-pkg de RTAB-Map dentro del directorio `src` del espacio de trabajo Catkin:

```
cd ~/catkin_ws
git clone https://github.com/introlab/rtabmap_ros.git src/rtabmap_ros
catkin_make
```

5.3. Generación de Mapas de Entorno

Llenar.

5.3.1. Requerimientos

Llenar.

5.3.2. Procedimiento

Llenar.

5.3.3. Pruebas

Llenar.

Generación de mapas 3D

Llenar.

Generación de mapas 2D

Llenar.

Selección de altura mínima y máxima para generación del mapa 2D

Capítulo 6

Conclusión y Recomendaciones

6.1. Conclusión

Llenar.

6.2. Recomendaciones

Llenar.