Projecte Final de Carrera

Programació i simulació del robot mòbil Guardian mitjançant ROS i Gazebo

Marc Benetó Juan

20 de Setembre del 2011

Codirector a l'empresa: Roberto Guzmán Diana
Codirector a l'ETSINF: Ángel Rodas Jordà

Universitat Politècnica de València
Escola Tècnica Superior d'Enginyeria Informàtica

Índex

1 Introducció. Motivació. Objectius.

La robòtica és la branca de la tecnologia encarregada del disseny i construcció de màquines capaces de realitzar tasques repetitives, tasques d'una gran precisió, tasques perilloses per a un ésser humà o simplement, tasques irrealitzables sense l'ajuda d'estes. Aquesta àrea de la ciència combina disciplines tant variades com la informàtica, l'electrònica, la mecànica, la intel · ligència artificial i l'enginyeria de control. No és d'estranyar que, amb l'avanç que ha experimentat en els últims anys la informàtica, la robòtica haja deixat de ser una raresa per començar a convertir-se en quelcom habitual en les nostres vides. La cada volta major capacitat de càlcul i la miniaturització dels elements informàtics fa possible la construcció de robots per a tots els àmbits. Podem trobar robots des de fàbriques on es duen a terme els més complicats processos industrials fins a l'interior de les nostres cases fent la neteja.

Aprofitant l'oportunitat que ofereix la Universitat Politècnica de València de realitzar pràctiques en empreses amb les quals mantenen convenis, vaig poder realitzar este projecte a l'empresa **Robotnik Automation SLL** de la mateixa ciutat. Esta empresa, capdavantera en l'estat en l'àmbit de la robòtica, es dedica al desenvolupament de productes i prestació de serveis de enginyeria i I+D en el sector de la robòtica i l'automàtica. Com a becari de l'àrea de I+D les meues tasques foren l'aprenentatge d'un nou *framework* per a la robòtica (ROS) i el posterior ús per a la programació i simulació del robot mòbil **Guardian** en este llenguatge.

ROS, sigles de **Robot Operating System**, és un *framework* que ens proporciona serveis estàndard dels sistemes operatius com l'abstracció de hardware, el control de dispositius a baix nivell, la implementació prèvia de funcions comuns, la comunicació entre processos i la gestió de paquets. Està basat en una arquitectura de graf, on el processament té lloc als nodes. Estos poden rebre, publicar i multiplexar sensors, controls, estats, planificacions, actuacions i altres missatges. Es tracta d'un llenguatge molt útil i ràpid donada la seua estructura modular, que fa estalviar molt de codi donat que podem re-utilitzar el codi que el propi llenguatge ens ofereix a les seues "llibreries", des d'interfícies per als sensors fins a complexes algorismes de navegació autònoma. No obstant això, cal dir que la corba d'aprenentatge és bastant pronunciada i exigeix un nivell avançant en C++ o Python (no em d'oblidar que ROS és bàsicament una interfície que s'executa sobre un d'estos dos llenguatges). Per últim cal ressenyar que este *framework* és software lliure i està enfocat per a treballar sobre GNU/Linux, més concretament amb la distribució Ubuntu, conta amb una gran comunitat que el suporta darrere i les pàgines de col · laboració amb

ella segueixen molt la filosofia 2.0, el que facilita i ajuda molt a la difusió d'este llenguatge i pel que suposem que és un ferm candidats a convertir-se en un dels estàndards *de facto* per al futur.

Una vegada amb el que és el llenguatge ROS clar, podem definir els objectius del projecte que ens ocupa:

- Aprenentatge i reforç dels coneixements adquirits en la carrera sobre robòtica mòbil.
- Aprenentatge d'un nou *framework*: ROS.
- Programació i simulació del robot mòbil *Guardian* en este últim llenguatge.

Esta memòria està composada de set capítols. En ells s'expliquen els conceptes utilitzats per al desenvolupament final del projecte així com la implementació d'aquest. Anem a fer una vista general del que la memòria ens ofereix, oferint un xicotet resum de cada capítol. El primer capítol amb el que ens trobem, deixant de banda l'actual és l'anomenat *"El robot mòbil Guardian"*. En aquest capítol s'estudia el robot que més tard modelarem i simularem i que al cap i a la fi serà un dels principals eixos del nostre projecte. Veurem les seues característiques tècniques, parts principals i les aplicacions. Més tard s'oferiran uns conceptes bàsics al capítol *"Fonaments de robòtica mòbil"*. En este, s'explicaran principis bàsics de cinemàtica diferencial, control, odometria i navegació autònoma, tots ells sense entrar en cap detall d'implementacions. Una vegada estudiats els fonaments del robot, ens centrarem en el *framework* que anem a utilitzar per a la programació i simulació d'este: ROS. Estudiarem els objectius amb els que fou creat, l'arquitectura que segueix i els conceptes bàsics des dels diferents nivells que l'integren. A continuació, estudiarem la simulació en el món de la robòtica i quins avantatges ens ofereix, a més, ens centrarem en l'estudi del simulador a utilitzar: Gazebo. Una vegada estudiat l'estat de l'art referent al nostre projecte ens centrarem en la implementació del treball. Ací és on es troba reflexada la major part de la feina realitzada. En aquest capitol es detallen minuciosament els passos seguits per dur a terme el projecte, des de la instal · lació de ROS fins a la creació de mons virtuals. Més endavant, en el capitol *"Descripció del treball"*, explicarem l'arquitectura del sistema resultant, com ha sigut la nostra interacció amb la comunitat de software lliure i disposarem d'exemples i guies d'utilització. Per acabar, oferirem les nostres conclusions del projecte i els possibles treballs futurs que poden sorgir en base a aquest

2 El robot mòbil Guardian

A esta secció anem a estudiar el robot mòbil sobre el qual es centra el nostre treball, la plataforma **Guardian** de l'empresa **Robotnik Automation SLL**. Anem a centrar-nos en les característiques tècniques del robot per més tard estudiar les seues possibles aplicacions. No obstant, abans que res, anem a fer una vista general del que és aquest robot i que ens pot oferir.

2.1 Vista general

En els últims anys, han proliferat un bon nombre de plataformes mòbils, gran part d'elles dissenyades per a utilitzar-se en investigació i desenvolupament. Els diferents fabricant, com *Mobile Robots Inc*, *Robosoft*, *Kteam*, o la pròpia *Robotnik*, han introduït al mercat gran varietat de plataformes mòbils amb les que poden investigar en robòtica mòbil, navegació, localització, SLAM, etc, i que serveixen al mateix temps per al desenvolupament d'aplicacions en el cresquent sector de la robòtica de serveis.

El robot *Guardian* pertany a este tipus de plataformes, i s'ha desenvolupat amb l'objectiu de proporcionar una plataforma d'interior i exterior que superarà la mobilitat del seus competidors naturals, tant en entorns urbans com en entorns naturals i amb prestacions que permitisquen la seua utilització en el producte final i no només en el prototipus demostrador.

Guardian és una plataforma mòbil modular molt adient per a tasques d'investigació, seguretat i inspecció, gràcies a que disposa d'alta mobilitat, la qual cosa li permet actuar en llocs de difícil accés per a altres tipus de plataformes (escales, terrenys pedregosos, forts desnivells...). Permet la integració de múltiples sensors (laser indoor, outdoor, càmeres, capçals de visió estèreo, unitat de medició inercial, GPS...) i accionaments (braços robotitzats modulars, unitats pan-tilt...). *Guardian* ofereix un espai suficient en el seu interior per a incorporar dos ordinadors. Açò fa que siga possible un major processament a bord per a visió, telemetria làser, DGPS... La plataforma té un pes de 85Kg que permet la carrega de fins a 100Kg addicionals en forma d'equipament. També s'ha de destacar que no requereix d'un equipament extern voluminós, només d'un maletí de control.

A continuació disposem de dos fotos amb diferents versions d'esta plataforma mòbil. Com vegem l'estructura és sempre la mateixa, variant això si, la mida d'esta, mentre que els sensors i diferents elements que l'integren poden variar depenent de l'aplicació per a la que estiguen dissenyats.

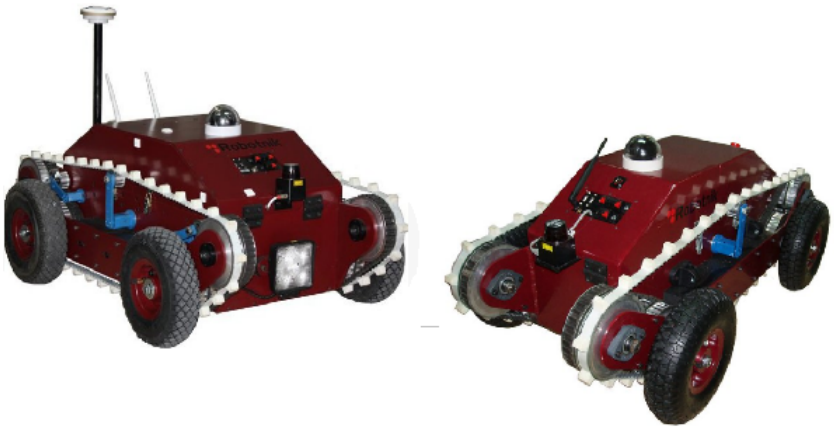

Figura 2.1: Diferents versions de la plataforma mòbil Guardian.

2.2 Característiques tècniques

A continuació presentem dos taules resum de les característiques tècniques més importants del Guardian, una dedicada a les característiques mecàniques i altra a les de control del robot. Hem de tindre en compte que cada robot fabricat és diferent a l'anterior ja que s'adapta al màxim a l'aplicació per a la que ha sigut dissenyat, per açò volem remarcar que estes taules només són orientatives.

Mides	1050 x 600 x 400 mm
Pes	85 Kg
Capacitat de càrrega	50-100 Kg
Velocitat	1,25 m/s
Tipus de protecció	IP54 / Opcions IP65 i IP66
Sistema de tracció	Orugues combinades amb rodes
Motorització	2 eixos, control diferencial (skid) 2 x 1000W
Bateries	2x12V, 50 Ah
Autonomia	3h de funcionament normal
Rang de temperatures	Dels -10º als +50ºC
Màxima pendent	45º

Taula 2.1: Característiques mecàniques del robot Guardian

Sistema modular	Permet muntar combinacions d'un o dos manipuladors.
	Permet la connexió de qualsevol tipus de sensorització.
	Ports USB i RS-232 a l'exterior.

Controlador	Versió bàsica: Radio controlada.
	Versió completa: PC empotrat amb el sistema operatiu **LinuxRT**.
	Drivers per a **Player/Stage** i **ROS**.

Taula 2.2: Característiques de control del robot Guardian

2.3 Parts principals

En la secció que ara ens ocupa anem a estudiar les diferents parts que conformen el robot. Si ens fixem en la figura de més avall, el robot disposa de quatre elements clarament diferenciats, estos són: el **bastidor principal** (*main chassis*), els dos **accionaments laterals** -dret i esquerre- (*lateral drive side*), la **coberta davantera** (*front cover*) i la **coberta posterior** (*back cover*).

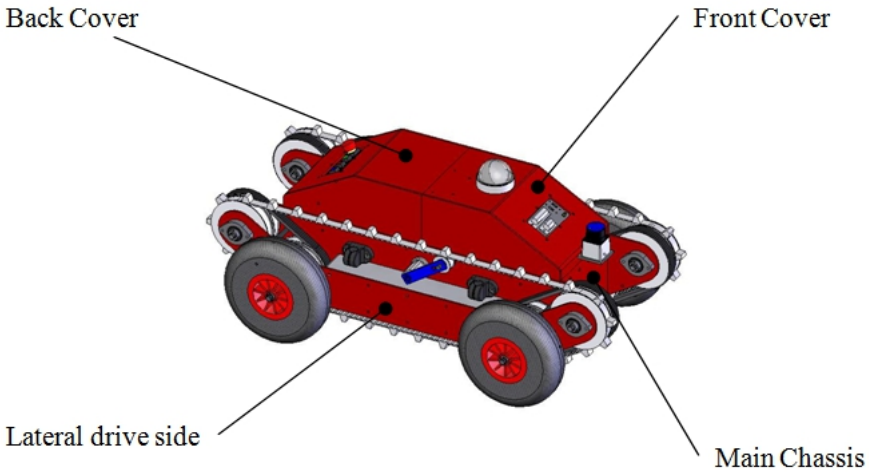

Figura 2.2: Parts principals del robot Guardian. Robotnik Automation SLL ©

Anem a vore amb un poc més de detall estes quatre parts principals:

- **Bastidor principal:** Ací dins estan allotjats tots els components elèctrics: Motors, controladores, ordinadors (depenent del model), receptors de ràdio...

- **Accionaments laterals:** Situats tant a l'esquerra com a la dreta del robot, és on estan col · locats els elements que mouen el robot, és a dir, les orugues i les rodes. Esta part del robot és la que més elements mecànics conté i per això necessitarà un cert manteniment.

- **Cobertes (davantera i posterior):** Permet accedir a l'interior del robot on es troben molts dels elements de control del robot com la controladora AX3500 o l'ordinador d'abord.

Com vegem els **accionaments laterals** són la part mecànica més complexa del robot, anem a veure'ls més detalladament:

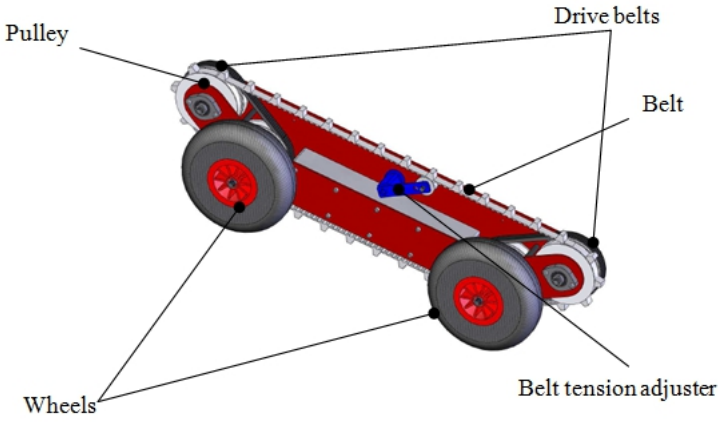

Figura 2.3: Accionaments laterals del robot. Robotnik Automation SLL ©.

- **Politja (*pulley*):** Transmet el moviment de les corretges de transmissió a l'eix on estan fixades. És necessari revisar l'estat de les politges una vegada a l'any.

- **Corretges de transmissió (*drive belt*):** Transmet el moviment des dels motors. Com les politges, és necessari comprovar l'estat de desgast una vegada a l'any.

- **Corretges/orugues (*belt*):** Estes corretges/orugues tenen dues funcions. Primerament, la de transmetre el moviment entre les politges anteriors i posteriors. En segon lloc, fent ús dels "tacos" externs que incorporen poden evitar obstacles així com pujar escales. És necessari comprovar el nivell de debilitació i consistència dels "tacos" una vegada a l'any com a mínim.

- **Ajustadors de la tensió de les corretges (*belt tension adjusters*):** Estos elements mantenen la tensió de les corretges. Per a un funcionament correcte és necessari comprovar el seu estat cada 6 mesos.

- **Rodes (*wheels*):** Les rodes fan possible la transmissió habitual del robot. És necessari comprovar la pressió de les rodes una vegada al mes.

Una vegada vistes les parts més importats, volem demostrar la gran modularitat del robot amb un senzill exemple. Este model en concret, disposa d'una estructura mòbil

on podem instal · lar sensors i demés elements de mesura. Esta estructura, com es d'esperar, la podem instal · lar i desinstal · lar fàcilment. A continuació vegem com queda el robot amb l'estructura.

Figura 2.4: Estructura mòbil per al Guardian. Robotnik Automation SLL ©.

Esta estructura mòbil està composada per 5 peces que poden ser connectades fins a aplegar a una altura aproximada de 1100 mm. Hi ha dos tipus de peces per fer-ho: les de 250 mm i les de 150 mm, que poden ser combinades a petició de l'usuari. El resultat, una vegada muntat, és el següent:

Figura 2.5: Robot Guardian amb l'estructura mòbil ja incorporada. Robotnik Automation SLL ©.

2.4 Aplicacions

Els camps d'aplicació d'una plataforma mòbil tant flexible com esta són molts, no obstant, podem englobar-los bàsicament en dos grans apartats: investigació i robòtica de servei. Com que es tracta d'un producte comercial no podem entrar en detalls, per això anem només a enumerar algunes de les aplicacions que ha desenvolupat el *Guardian* a estos dos camps.

Pel que fa a la investigació, este plataforma s'ha utilitzat per al desenvolupament d'algorismes de mapeig i localització simultània (SLAM[1]) en exteriors així com investigació d'algorisme de *Vision based* SLAM.

Per altra banda, entre les aplicacions ja desenvolupades per a este robot la més comú és la de monitorització i medició remota com per exemple medició de radiacions, medició 3D d'excavacions arqueològiques. També, amb els sensors adequats este robot pot utilitzar-se per a la desactivació d'explosius, ja siga buscant i desactivat explosius improvisats (***IEDD***) o desactivant explosius armamentístics (***EOD***).

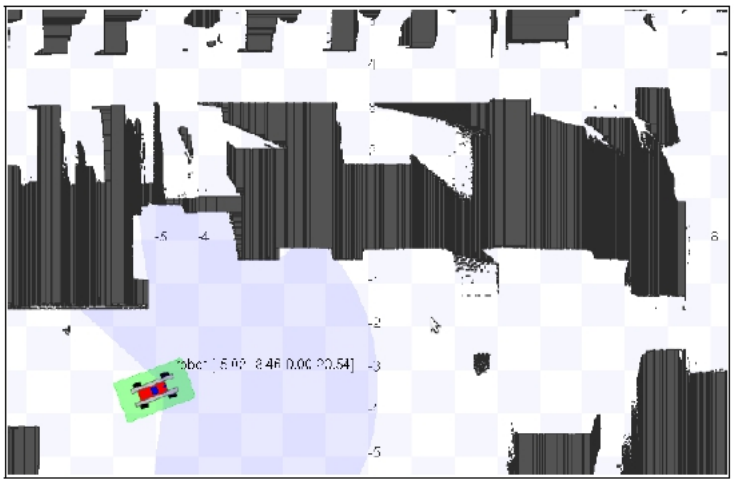

Figura 2.6: Mapa obtingut en *Player/Stage* mitjançant SLAM.

[1]Simulatenous Localization and Mapping

3 Fonaments de robòtica mòbil

Un robot mòbil és aquell que és capaç de moure's per un entorn en concret. Com podem suposar per açò últim, este tipus de robot haurà de disposar d'una intel · ligència interna que el permeta posicionar-se en tot moment i a més puga reaccionar davant qualsevol obstacle que puga aparèixer en un moment determinat. No de bades, este tipus de robot és un dels més complicats de dissenyar, ja que els entorns i situacions a les que és pot enfrontar la màquina són pràcticament infinites.

És usual vore este tipus de robots en la industria de transformació, en la indústria militar i en entorns de seguretat. Així i tot i com ressenyàvem al principi, cada volta és més habitual vore este tipus de robot a les nostres llars, ja siga passant l'aspiradora, arreglant el jardí o fent altres feines més comuns.

Podem classificar els robots mòbils per diversos criteris, estos són:

Segons l'entorn en que es mouen	Robots terrestres o amb rodes.
	Robots aeris (UAV)
	Robots submarins (AUV)
	Robots polars.
Segons el dispositiu utilitzat per moure's	Robot articulat.
	Robot amb rodes
	Robot amb cadenes/orugues

Taula 3.1: Classificació dels robots mòbils.

En esta secció estudiarem els robots mòbils amb rodes en profunditat des de les diferents parts que l'integren i que hem cregut que són les més característiques. Primerament parlarem de les lleis de la cinemàtica en les que este tipus de robots basen el seu moviment. Més endavant estudiarem el control automàtic, és a dir, com els propis motors del robot modificaran el comportament del sistema (l'eixida dels motors en este cas) mitjançant les magnituds d'entrada i la realimentació encoder-motor. En un tercer lloc vorem que és la odometria i quins mètodes s'utilitzen per posicionar en l'espai un robot mòbil en base a la rotació de les seues rodes. Per últim ens centrarem en la navegació del robot i els diferents algorismes de localització que s'utilitzen com són **MCL** (*Monte Carlo Localization*) i **SLAM** (*Simultaneous Localization And Mapping*).

3.1 Cinemàtica diferencial

La cinemàtica és la rama de la física encarregada d'estudiar les lleis del moviment dels cosos, sense arribar a tindre en compte els fets que provoquen este moviment i centrant-se en la trajectòria que seguiran. No és difícil trobar la relació entre la robòtica mòbil i està part de la mecànica ja que esta és la manera de descriure i formalitzar els moviments i trajectòries que efectua un robot.

La cinemàtica diferencial en la robòtica consisteix en dos rodes directrius muntades sobre un eix comú on cada roda pot anar independentment avant o arrere. Esta diferència de velocitats entre les dos rodes permet al robot efectuar moviments de gir sobre un punt que estarà al llarg de l'eix d'estes dos rodes. Este punt s'anomenarà **ICC**, sigles en anglés de *Instantaneous Center of Curvature* (Centre Instàntani de Curvatura/Rotació). A la imatge podem vore l'esquema d'una configuració amb dos rodes diferencials.

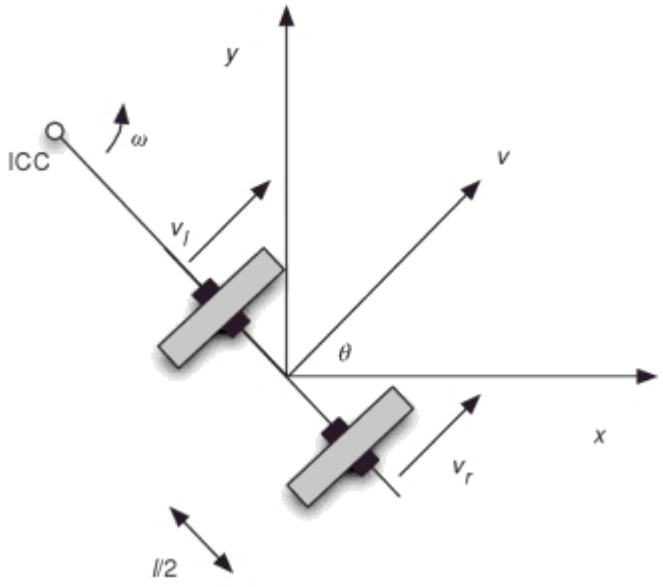

Figura 3.1: Esquema d'una configuració diferencial.

A l'esquema podem vore les dos rodes sobre el mateix eix (l'esquerra i la dreta) i la diferència que hi ha entre elles (**l**). A més, veiem el ICC i la distància que hi ha entre ell i el punt central del robot (**R**). Suposem a partir d'ací que cada roda té una velocitat, **Vl** per la roda esquerra i **Vr** per la dreta. Si variem les velocitats d'estes dos rodes, podem variar la trajectòria que el robot prendrà. Com que la ràtio **de rotació ω** sobre el ICC ha de ser el mateix per a les dues rodes, podem escriure les següents equacions:

$$\omega(R + l/2) = V_r$$

$$\omega(R - l/2) = V_l$$

A partir d'estes equacions podem obtenir les dues següents, amb les quals podrem saber per a cada instant de t la distància del centre del robot al **ICC (R)** i la **velocitat angular del gir (ω)**.

$$R = \frac{l}{2}\frac{V_l+V_r}{V_l-V_r}$$

$$\omega = \frac{Vr-V_l}{l}$$

D'ací obtenim el tres tipus de moviment que poden efectuar els robots amb esta cinemàtica:

- Si **Vl = Vr** , obtenim un moviment en línia recta. R és infinita i efectivament, no hi ha rotació. **ω** també serà zero.
- Si **Vl = -Vr** , aleshores R = 0 i tenim una rotació sobre el punt mig de l'eix de les rodes, és a dir, el robot rotarà sobre si mateixa.
- Si **Vl = 0** el robot rotarà sobra la roda esquerra. En este cas R = l/2. El mateix es complirà si Vr = 0.

De les anteriors equacions podem obtenir les diferents velocitats (**lineal** i **angular**) al centre del mòbil. Estes equacions més generals ens seran útils més endavant com a paràmetres d'entrada i per modelar el moviment del robot.

$$v = \frac{Vr+Vl}{2}$$

$$\omega = \frac{Vr-V_l}{2}$$

3.1.1 Cinemàtica directa per a robots diferencials

El problema cinemàtic directe consisteix en determinar quina és la posició i orientació de l'extrem final del robot, amb respecte a un sistema de coordenades que es pren com a referència, coneguts els valors de les diferents velocitats i els paràmetres geomètrics dels elements del robot. En la figura anterior, hem suposat que el robot està en una posició (x, y) i dirigit en una direcció formant un angle ϑ amb l'eix de les X. Assumim que el robot està centrat al punt mig al llarg de l'eix de les rodes. Modificant els paràmetres de control **Vl** i **Vr** podem fer que el robot es moga a diferents posicions i orientacions. Sabent estes velocitats i utilitzant l'equació tercerà que em presentat, podem localitzar el centre del radi de curvatura (ICC).

$$ICC = [x - R\sin(\vartheta), y + R\cos(\vartheta)]$$

A partir de l'equació anterior podem saber la posició del robot en qualsevol t + δt.

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} \cos(\omega\partial t) & -\sin(\omega\partial t) & 0 & x - ICC_x & ICC_x \\ \sin(\omega\partial t) & \cos(\omega\partial t) & 0 & y - ICC_y & + \; ICC_y \\ 0 & 0 & 1 & \theta & \omega\partial t \end{bmatrix}$$

Esta equació només descriu el moviment del robot rotant una distància R sobre el seu ICC amb una velocitat angular de ω. Podem vore el desplaçament del robot com a 1) translació del ICC al origen del centre de coordenades del sistema, 2) rotació sobre l'origen una quantitat angular ω per un temps δt i 3) translació, novament al ICC.

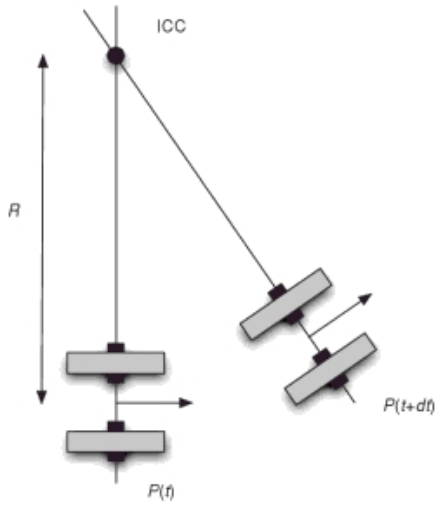

Figura 3.2: Esquema del model diferencial directe.

3.1.2 Cinemàtica inversa per a robots diferencials

El problema cinemàtic invers consisteix en trobar els valors que deuen adoptar les velocitats del robot (**Vr** i **Vl**) per a que el seu extrem es posicione i oriente segons una determina localització espacial. Al contrari que el problema cinemàtic directe, el càlcul de la cinemàtica inversa no és senzill ja que consisteix en la resolució de una sèrie d'equacions que depenen molt de la configuració del robot, a més d'existir diferents solucions que resolen el problema.

En general, podem descriure la posició d'un robot capaç de moure's en una direcció en concret $\mathbf{\Theta}$ **t** a una velocitat donada $\mathbf{V(t)}$ com:

$$x(t) = \int V(t)\cos[\theta(t)]dt$$

$$y(t) = \int V(t)\sin[\theta(t)]dt$$

$$\theta(t) = \int \omega(t)dt$$

Malauradament, un robot amb cinemàtica diferencial imposa el que són conegudes com a limitacions no-holonòmiques en l'establiment de la seua posició. Per exemple, el robot no podrà moure's mai al llarg del seu eix. Una limitació no-holonòmica similar és la d'un cotxe que només pot girar amb les seues rodes davanteres. Com sabem, serà impossible que es moga lateralment fent així que el seu aparcament requerisca un conjunt més complicat de maniobres. Açò ens serveix per exemplificar que en el problema cinemàtic invers no anem a poder simplement especificar la posició desitjada (**x, y, ϑ**) i obtindre directament les velocitats que ens farien que el robot es traslladés ahí, sinó que és una cosa més complicada i que queda fora de l'objectiu d'este text. No obstant això, si que podem indicar quines són les equacions per al casos especials.

- Per al cas en el que **Vl = Vr** (moviment en línia recta)

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x + v\cos(\theta)\delta t \\ y + v\sin(\theta)\delta t \\ \theta \end{bmatrix}$$

- Per al cas en que **Vl=-Vr** (moviment rotacional sobre ell mateix)

$$\begin{bmatrix} x' \\ y' \\ \theta' \end{bmatrix} = \begin{bmatrix} x \\ y \\ \theta + 2v\delta t/l \end{bmatrix}$$

El fet de que estos casos especials siguen senzill de modelar, motiva l'estratègia de fer moure el robot en línies rectes la major part del temps possible per a efectuar els girs sobre ell mateixa quan siga necessari.

3.2 Control

Un sistema de control és un conjunt de components interconnectats dedicats a obtenir la eixida desitjada per a un sistema o procés, de manera que es redueixen les probabilitats d'errors i s'obtinguen els resultats buscats. Al cas que ens ocupa, la robòtica mòbil, trobem sistemes de control implementats la major part de les voltes per controladors PID (Proporcional-Integral- Derivatiu) que s'encarreguen de controlar la velocitat o posició de les rodes depenent del model. Este control pot efectuar-se en diferents parts del sistema, des de a la programació del software del robot com implícitament a l'electrònica de la placa controladora dels motors. A més, pot efectuar-se de diferents formes, estes són:

- **Sistema de control en llaç obert:** És aquell sistema en que només actua el procés sobre la senyal d'entrada i dona com resultat una senyal d'eixida independent a la senyal d'entrada, però en base a la primera. Això significa que no hi ha retroalimentació cap al controlador per que aquest puga ajustar l'acció de control. Es a dir, la senyal d'eixida no es converteix en senyal d'entrada per el controlador.
- **Sistema de control en llaç tancat:** Son els sistemes en els que l'acció de control està en funció de la senyal d'eixida. Els sistemes de circuit tancat utilitzen la retroalimentació des d'un resultat final per ajustar l'acció de control en conseqüència.

El **llaç obert** ens proporciona un control més senzill però per contra no assegura l'estabilitat del sistema ja que no es compara l'eixida amb l'entrada, a més de poder quedar afectat per les pertorbacions. Açò fa que siga pràcticament descartat en la robòtica, ja que en esta és preferible un controlador estable a pertorbacions i possibles variacions internes, sent capaç de retroalimentar l'eixida amb l'entrada i efectuant segons això les accions de control adients.

3.2.1 El control proporcional (P)

A continuació presentarem el control de tipus proporcional. Anem a estudiar-lo detalladament ja que és el que utilitza el nostre robot per al control de velocitat de les rodes. En este tipus de control la relació entre l'eixida del controlador **u(t)** i la senyal d'error **e(t)** és la següent:

$$u(t) = Kp\, e(t)$$

El que ve a ser que l'acció de control serà proporcional a la senyal de error. La funció de transferència serà per tant, una constant, on **Kp** es denominarà guany proporcional.

$$\frac{U(s)}{E(s)} = Kp$$

Siga quin siga el mecanisme real i la forma d'aplicar la potència als accionaments, el controlador proporcional és en essència un amplificador de guany ajustable, ja que existeix una relació lineal continua entre el valor de la variable controlada i la posició final de l'element de control. Anem a vore com reacciona el sistema al ser excitat per un escaló d'amplitud **A**. Analitzarem el sistema tant en bucle obert, com en tancat.

En **bucle obert**, a una senyal de entrada en escaló d'amplada **A**, el procés respondrà amb un eixida:

$$Y(s) = R(s)G(s) = A\frac{G(s)}{s}$$

Si apliquem el teorema del valor final, l'eixida en règim permanent serà:

$$y_{ss} = \lim sY(S) = \lim sA\frac{G(s)}{s} = A \cdot G(0)$$

En bucle tancat, amb realimentació unitària i afegint el controlador proporcional, la resposta en estat estacionari y_{ss} a un escaló és:

$$y_{ss}(s) = \lim sY(s) = \lim s\frac{K_pG(s)}{1+K_pG(s)}\cdot R(s) = \lim s\frac{A}{s}\cdot\frac{K_pG(s)}{1+K_pG(s)} = \frac{K_pG(0)}{11+K_pG(0)}\cdot A$$

Podem observar que per a valors alts de **Kp**, l'eixida del sistema seguirà pràcticament a la senyal de referència. Per tant, un augment del guany del controlador permet reduir l'error en l'estat estacionari, no obstant, per a poder eliminar este error és necessari que la funció de transferència en llaç obert continga algun element integrador i el sistema siga estable, cosa que este tipus de controlador no proporciona. A més a més, la utilització de valors elevats del guany, pot provocar fenòmens com l'aparició de saturació en alguns elements, fent que el sistema entre en règim no lineal o inclús arribant a provocar la inestabilitat d'aquest.

3.2.2 El control PID. Cas d'estudi: Control en bucle tancat respecte a la velocitat.

Un dels controladors de llaç tancat més utilitzats és el **PID**. Segons les estadístiques, més d'un 80% dels controladors presents a l'àmbit industrial són d'este tipus (d'este

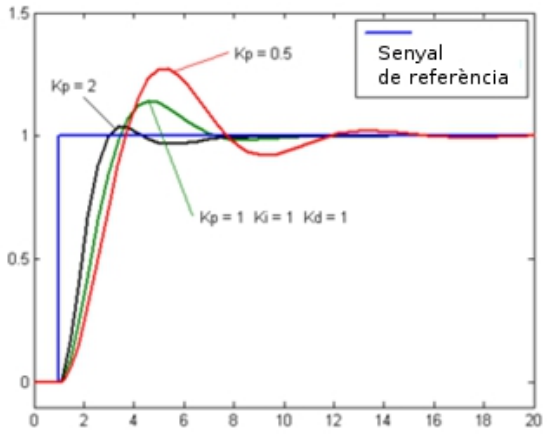

Figura 3.3: Gràfic del control P

o de qualsevol de les seues configuracions, com poden ser **P**, **PI** i en menor mesura **PD**). Aquest controlador es fa servir dins del camp del control automàtic per a realitzar les operacions bàsiques de l'àlgebra lineal: el producte per un escalar, la suma i la resta. Malgrat la seua senzillesa i facilitat d'utilització proporciona uns resultats molt satisfactoris. Tot i que ja hem apuntat que el nostre robot utilitzarà un controlador P a la simulació, no podem passar per alt el control **PID** per les raons abans mencionades i perquè el robot real utilitza un control d'este tipus integrat a la electrònica de la controladora. Veiem com funciona.

El **control PID en bucle tancat respecte a la velocitat** es basa en ajustar la potència d'eixida segon la diferència entre la velocitat desitjada (la que proporciona l'usuari) i la posició de la roda actual (la capturada pels encoders). Cada instant, depenent de la configuració, el controlador mesurarà la velocitat actual del motor i li restarà la posició desitjada per obtenir l'anomenat error de velocitat (e_v).

El resultat obtingut és aleshores multiplicat per un guany **proporcional** indicat per l'usuari. L'efecte d'esta part de l'algorisme és el d'aplicar potència al motor proporcionalment amb la diferencia entre la velocitat actual i la desitjada: quan més lluny s'estiga, més potència s'aplicarà i viceversa. Un major guany proporcional farà que l'algorisme aplique un major nivell d'energia a un error donat, de manera que el motor reaccionarà més ràpidament als canvis en les ordres.

$$u_1 = K_p\, e(t)$$

El component **diferencial** de l'algorisme calcula els canvis del error d'un instant donat al següent. Aquest canvi serà un nombre relativament gran cada vegada que

un canvi brusc es produïsca en el valor de la velocitat desitjada o el valor de la velocitat mesurada. El valor d'aquest canvi es multiplicarà per un guany diferencial que l'usuari pot seleccionar i afegir a l'eixida. L'efecte d'aquesta part de l'algorisme és el de donar un impuls d'energia extra en arrencar el motor a causa dels canvis en el valor de la velocitat desitjada. El component diferencial també és en gran mesura l'encarregat d'ajudar a reduir qualsevol excés i oscil · lació.

$$u_2(t) = \frac{K_p}{T_i} \int e(\tau)\delta\tau$$

Per últim, el component **integral** de l'algorisme fa una suma de l'error al llarg del temps. Este component ajuda al controlador a aplegar i mantindre la velocitat exacta dessitjada quan el valor de l'error està aplegant a zero. Açò es dona quan la velocitat mesurada és pròxima o directament és la desitjada.

$$u_3(t) = K_p T_d \frac{\delta e}{\delta t}$$

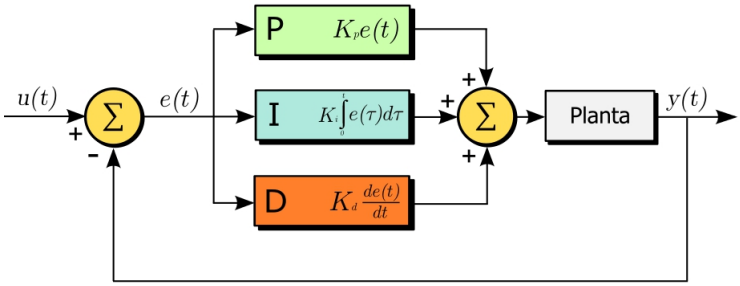

Figura 3.4: Estructura del controlador PID.

Nota: A les formules presentades u_i(t) és l'eixida de cadascun dels tres termes del controlador, e(t) és el senyal d'error, K_p és la constant proporcional, T_i és el temps integral i T_d és el temps derivatiu.

3.3 Odometria

La **odometria** és el mètode que utilitzen els robots mòbils amb rodes per posicionar-se d'una forma aproximada en respecte a la seua posició inicial a l'espai. La idea fonamental de la odometria és la integració d'informació incremental del moviment al llarg del temps, la qual cosa comporta una inevitable acumulació d'errades, que van argumentat proporcionalment amb la distància recorreguda pel mòbil.

La odometria es basa en equacions simples i en les dades obtingudes pels encoders situats a les rodes del robot. No obstant, la odometria suposa que les revolucions de

les rodes poder ser traduïdes amb un desplaçament lineal respecte al sòl, cosa que no és absolutament certa, ja que s'ha de tindre en compte fenòmens com la fricció de les rodes amb el sòl, la deformitat al llarg del temps d'estes o el fet de que no giren a les mateixes velocitats. Tot açò, unit a les més diverses raons i imprevistos, comporta unes imprecisions en la traducció de la informació obtinguda de l'encoder amb el que realment s'ha desplaçat el robot, per això, es recomanable que el robot es posicione addicionalment per sistemes de posicionament absoluts com les **IMU** (*Inertial Measurement Unit*) o el conegut **GPS** (*Global Positioning System*).

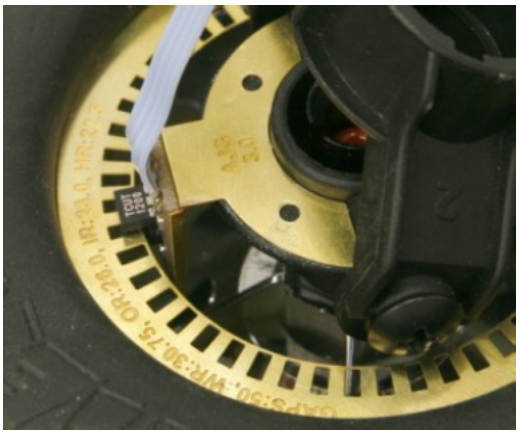

Figura 3.5: Detall d'un encoder muntat sobre una roda.

Els avanços més destacables en este camp giren entorn a l'**odometria visual**. Esta tecnologia naix motivada pel fet de que aquells robots mòbils sense rodes no poden fer servir l'odometria clàssica, com són per exemple els robots articulats. A més a més, intenta corregir les mancances sistemàtiques de l'odometria basada en rodes com són que els diàmetres de les rodes no siguen iguals, que estes estiguen mal alineades o que l'encoder siga un sistema discret i no continu amb els defectes que açò pot comportar en la mesura. L'odometria visual fa servir imatges seqüencials de les càmeres per estimar la distància viatjada, esta permet millorar la precisió en la navegació dels robots de qualsevol tipus de locomoció en qualsevol superfície.

3.4 Navegació autònoma

La navegació en la robòtica és la ciència de dirigir un mòbil al llarg del medi, ja siga terra, aigua o aire. L'objectiu inherent de qualsevol sistema de navegació és aplegar a la destinació sense que el robot es perda o tinga un accident. La capacitat de reacció davant de situacions inesperades ha de ser la principal qualitat per a desenvolupar-se, d'una forma eficaç, en entorns no estructurats. Les tasques involucrades en la navegació d'un robot mòbil són: la percepció de l'entorn a través dels sensors, de forma que li permeta crear una abstracció del món (**mapping**);

la planificació d'una trajectòria lliure d'obstacles per a aplegar al punt de de destí seleccionat (**planning**); i el guiat del vehicle a través de la referència construïda. De forma simultània, el vehicle pot interactuar amb certs elements de l'entorn. La navegació en la robòtica mòbil és tot un món en si mateixa, amb una gran quantitat d'algorismes de guiat, planificació i conducció que queden fora de l'objectiu d'esta memòria. No obstant, creguem necessari explicar el funcionament de dos algorismes de posicionament importants, el **MCL** i el **SLAM**, ja que són els que ROS implementa ara per ara i utilitzarem al nostre robot.

3.4.1 MCL

Monte Carlo Localization (MCL) es un mètode Monte Carlo (una classe d'algorisme basada en la repetició de resultats aleatoris per extraure els resultats desitjats) que serveix per determinar la posició del robot donat un mapa del seu entorn i basant-se en el mètode de localització Markov. Es tracta bàsicament d'una implementació del filtre de partícules aplicat a la localització del robot, i s'ha convertit en un mètode molt popular en la robòtica. En este mètode, un gran nombre d'hipotètiques configuracions inicials son dispersades aleatòriament per l'espai. Amb cada actualització dels sensors, la probabilitat de que cada configuració hipotètica siga correcta s'actualitza basant-se en un mètode estadístic dels sensors i en la teoria de Baye. De la mateixa manera, tots els moviments que el robot efectua són afegits a esta estadística de configuracions hipotètiques. En quan la probabilitat d'una configuració hipotètica és massa baixa, es reemplaçada per una nova configuració a l'atzar.

3.4.2 SLAM

Un mètode més interessant ja que integra la localització i el mapeig d'una forma concurrent és el **SLAM**, sigles angleses de ***Simultaneous Localization and Mapping***. Esta tècnica busca resoldre els problemes que planteja el col · locar un robot mòbil en un entorn i posició desconeguts, i que ell mateix siga capaç de construir incrementalment un mapa consistent de l'entorn a l'hora que utilitze este mapa per determinar la seua pròpia localització. La solució a este problema, no es trivial, i ha sigut objecte d'estudi per part de la comunitat científica durant les últimes dècades ja que el soroll inherent als sensors, els inevitables errors i aproximacions comeses en els models empleats fan que siga un problema difícil de resoldre, ja siga des d'un punt de vista conceptual (entorn confús, canviant i gens clar) com computacional (quantitat d'informació, forma i costos de tractar-la, així com emmagatzemar-la...).

Així doncs, en la base de qualsevol solució a la problemàtica del **SLAM** ens anem a trobar sempre amb la necessitat de treballar amb quantitats creixents d'informació

(contaminada en menor o major mesura), que manipularem mitjançant models que no són més que aproximacions a la realitat. Les solucions que millor resolen el problema del **SLAM** són aquelles que es basen en tècniques probabilístiques ja que permeten fer front a les incongruències inherents al procés que hem comentat anteriorment. Este tipus d'algorismes es basa en l'algorisme de Bayes, que relaciona entre si les probabilitats marginal i condicional de dues variables aleatòries. La manera en que treballa este tipus de formulació es considerar que tant la posició del robot com els elements que modelen el seu entorn son variables aleatòries. Així, els algorismes existents modelen de manera probabilística i utilitzant la inferència per a determinar aquella configuració que és més probable tenint en compte les mesures que van obtenint-se. Estos algorismes són:

- **Filtre Extés de Kalman (EKF):** és una de les solucions més utilitzades al problema SLAM i també una de les que millors resultats proporciona. Proporciona característiques tant interessants com el fet de descriure l'entorn a partir d'entitats geomètriques de forma compacta i la possibilitat de realimentar-se gràcies a l'ús d'una matriu de covariàncies del sistema completa que es capaç de tancar bucles amb èxit.

- **Mapa d'ocupació de cel · les (Occupancy Grid Mapping):** Este mètode es basa en la discretització de l'espai, dividint-lo en unitats d'una mida predefinida que es classifiquen com ocupades o buides amb una nivell de confiança o probabilitat. Estes solucions suposen que la posició del robot és coneguda. En la pràctica, es necessita algun mètode alternatiu de localització que estime la posició del robot en cada instant. Malgrat este desavantatge, és un algorisme senzill d'implementar i molt robust a més de distingir entre zones ocupades i buides, aconseguint una partició i descripció completa i detallada de l'espai explorat, fent molt senzilla la planificació i generació de trajectòries utilitzant tècniques convencionals.

4 ROS: Un framework de codi obert per a la robòtica

Escriure software per a robots és complicat, sobretot degut a que l'escala i els enfocaments de la robòtica no paren de créixer. Diferents tipus de robots poden tindre quantitats de hardware molt diferent, fent que la reutilització de codi siga un problema no trivial. A més a més, la mida del codi requerit pot ser enorme, ja que pot anar des de la programació del driver de les rodes fins a la percepció o el raonament abstracte. Per superar estos reptes, molts desenvolupadors han creat prèviament una gran quantitat de *frameworks* per facilitat el prototipat de software per a experiments, donant com a resultat molt del software que actualment s'utilitza tant en industria com en investigació. Cadascun d'estos *frameworks* foren dissenyats amb un objectiu en particular, pot ser en resposta a les febleses percebudes en altres *frameworks* disponibles o per donar-li prioritats a aspectes que el desenvolupador considera que són més importants en el procés.

ROS, el *framework* que anem a estudiar i hem utilitzat per a la realització del projecte, és també producte de les compensacions i prioritats fetes durant el seu cicle de disseny. Segons els dissenyadors, s'ha posat detall en la gran integració del producte, cosa que el farà útil en una gran varietat de situacions així com en un món on la robòtica és cada vegada una cosa més complexa. Com a visió general, podem vore ROS com un *framework* de codi obert per a robots, que proporciona els servicis que podrien esperar-se d'un sistema operatiu, com el control de dispositius a baix nivell, l'abstracció del hardware, el pas de missatges entre processos o inclús la gestió de paquets. Cal destacar, que tota la tasca de desenvolupament i implantació d'este nou *framework* l'esta realitzant l'empresa nordamericana **Willow Garage**. Esta empresa ha sigut fundada per gent provinent de **Google**, així com també és financiada per esta mateixa empresa.

A continuació descriurem d'una forma detallada els objectius general de ROS i com s'han implementat en el seu desenvolupament.

4.1 Objectius

Els objectius principals de ROS poden ser resumits en:

Peer-to-peer: Un sistema construït utilitzant ROS consisteix en un nombre de processos, potencialment en diferents hosts, connectats en temps d'execució en una topologia punt-a-punt (*peer-to-peer*). Tot i que alguns frameworks basats en un servidor central (per exemple, CARMEN[1]) poden tindre beneficis sobre el disseny multi-procés i multi-host, un servidor central és problemàtic si els ordinadors estan connectats en una xarxa heterogènia.

Per exemple, en els grans robots de servei per als que ROS ha sigut dissenyat solen haver diversos ordinadors abord connectats via Ethernet. Este segment de la xarxa està pontejant via wireless a ordinadors d'offboard de gran potència que estan executant tasques de comput molt intensives, com poden ser visió o reconeixement de la parla. Que s'execute el servidor central a bord o fora del robot, dona com a resultat un tràfic innecessari creuant el feble pont inalambric, ja que molts dels missatges o serveis necessitats es troben en xarxes i subxarxes, ja siga fora o dins de vehicle. Per altra banda, la connectivitat *peer-to-peer*, combinada adequadament amb el buffering per software quan és necessari, resol el problema d'una forma completa. La topologia *peer-to-peer* requereix algun tipus de mecanisme de cerca que permeta als processos trobar-se uns amb els altres en temps d'execució. Açò és el que s'anomenarà name service, o master, i vorem en breu.

Multillenguatge: Quan s'escriu codi sempre es tenen unes preferències d'un llenguatge respecte un altre. Este preferències venen donades per l'experiència personal, la facilitat de depuració, la sintaxis, l'eficiència en temps d'execució... Per este raons, ROS ha sigut creat per ser neutral en el llenguatge i per això suporta, a dia de hui diferents llenguatges com són: C++, Python, Octave, LISP, Java i molts llenguatges més que estan en procés de ser suportats.

L'especificació ROS es només a la capa dels missatges, no aplega més enllà. La negociació i configuració de la connexió *peer-to-peer* ocorre en XML-RPC[2], per al qual existeix la seua implementació en la major part dels llenguatges. En lloc de proporcionar una aplicació basada en C amb interfícies de codi auxiliar generat per als principals idiomes, ROS es implementat de forma nativa en cada idioma de destí, d'esta manera es segueixen millor les convencions de cada llenguatge. Per donar suport al desenvolupament creuat entre idiomes, ROS utilitza una interfície simple i neutral de cara al llenguatge, l'anomenada IDL per descriure els missatges escrits entre mòduls. IDL utilitza fitxers de text per descriure els camps de cada missatge i permet la composició d'estos. Per exemple:

```
Header header
Point32[] pts
ChannelFloat32[] chan
```

[1]Carnegie Mellon Robot Navigation Toolkit

[2]Protocol de crida a procediments remots (RPC) que fa servir XML per codificar les crides i HTTP com a mecanisme de transport de les dades.

Els compiladors de cada llenguatge suportat generaran implementacions natives que seran com objectes "natius" del llenguatge i que seran automàticament serialitzats i des-serialitzats per ROS quan un missatge siga enviat o rebut. Conste que estes 3 senzilles línies IDL[3] són traduïdes a C++ com a 137 línies en natiu, i 96 en Python, la qual cosa estalvia molt de temps i errades. Tot açò en conjunt ens proporciona un esquema de processament de llenguatge neutral, on els diferents llenguatges que implementen el sistema poden estar combinats com es desitge.

Basat en ferramentes: Amb l'objectiu de controlar la complexitat de ROS, els dissenyadors optaren per un disseny de microkernel, on un gran nombre de xicotetes ferramentes s'utilitzen per construir i executar els diversos components de ROS. Estes ferramentes realitzen diverses tasques, ja siga des de navegar per el sistema de fitxers, obtenir i modificar paràmetres de la configuració, visualitzar la topologia de la xarxa *peer-to-peer*, mesurar la utilització de l'ample de banda, representar dades gràficament, fins a auto-generar documentació. A pesar de que s'hagueren pogut integrar alguns aspectes globals dins del modul master, amb esta filosofia es creu que la possible pèrdua d'eficiència queda més que compensada pel guany que suposa en estabilitat, control, claredat i comprensió.

Lleuger: Com ja s'ha comentat anteriorment, molt del software existent en robòtica conté drivers i algorismes que haurien de poder-se reutilitzar en altres projectes. Desafortunadament, degut a diverses raons, la majoria d'estos codis son tant complexos que és molt difícil extraure la funcionalitat i reutilitzar-los fora del seu context original. El sistema de compilació (*build system*) de ROS realitza una compilació modular seguint una estructura d'arbre, açò, junt a l'ús de CMake fa que siga fàcil mantenir l'estructura lleugera que propugna ROS. La col · locació de pràcticament tota la complexitat en llibreries i el fet de només crear xicotets executables que exporten estes funcionalitats a ROS, permet escriure codi més senzill i reutilitzable.

A més, ROS reutilitza codi de nombrosos projectes *open-source*, com drivers, sistemes de navegació, simuladors del projecte Player, algorismes de visió d'OpenCV, algorismes de planificació d'OpenRAVE i molts més. En cada cas, ROS és utilitzat únicament per exposar diverses opcions de configuració i per a dirigir les dades de i fins a el software respectiu, quan un xicotet *wrapping* (envoltori) és possible. Per beneficiar-se del les continues millores de la comunitat, el compilador de ROS pot actualitzar automàticament el codi de repositoris externs i aplicar pedaços.

Gratuit i Open-Source: El codi font complet de ROS és disponible públicament. Amb açò s'intenta facilitar la depuració a tots els nivells del software que l'integren

[3]Llenguatge utilitzat per descriure la interfície de components de software d'una forma netural, el que permet la comunicació entre components de diferents llenguatges.

i més concretament quan molt del hardware i el software s'està depurant en paral · lel. ROS es distribuït baix dels termes de la llicència BSD, la qual permet el desenvolupament tant de projectes comercials, com no. ROS passa les dades entre els mòduls utilitzant comunicacions entre processos, i per açò no necessita que els mòduls estiguen enllaçats conjuntament en el mateix executable. Sent així i seguint amb la filosofia modular, els diferents components que integren un paquet poden disposar de diferents llicències, des de GPL fins a BSD propietària.

4.2 Arquitectura

Els conceptes fonamentals de ROS són els nodes, missatges, tòpics i serveis. Estos s'organitzen amb la següent arquitectura.

Els **nodes** són processos que efectuen còmputs, ROS està dissenyat per ser modular a una escala mínima: un sistema està comprés típicament per uns quants nodes. En este context, el terme "node" es intercanviable pel de "modul de software". L'ús que fem del terme node prové de la visualització dels sistemes basats en ROS en temps d'execució: quan molts nodes estan executantse, es convenient visualitzar la comunicació *peer-to-peer* com si es tractara d'un graf, amb els processos com a nodes d'este i els enllaços *peer-to-peer* com a arcs. Els nodes es comuniquen els uns amb els altres passant-se missatges.

Un **missatge** és una estructura de dades fortament tipada. Els missatges suporten els tipus estàndards primitius (*integer, floating point, boolean,* etc.) així com arrays d'estos mateixos tipus i constants. Els missatges poden ser composats d'altres i els arrays d'altres missatges. Un node envia un missatge publicant-lo en un **tòpic** en concret. Un node que estiga interessat en un cert tipus d'informació es subscriurà al tòpic apropiat. Poden haver múltiples ***publishers*** (publicadors) i ***subscribers*** (subscriptors) per a un únic tòpic, i un únic node pot publicar i/o subscriure's a múltiples tòpics.

Tot i que la comunicació basada en tòpics i el model de publicadors/subscriptors és un paradigma flexible de comunicacions, el seu esquema d'encaminament per difusió no és apropiat per a les transaccions síncrones. En ROS solucionarem açò fent ús dels **serveis** (*service)*, que queden definits per un nom en *string* i un parell de missatges fortament tipats: un per a la petició i un parell per a la resposta.

4.3 Conceptes i sintaxis

4.3.1 Sistema de fitxers

A este nivell trobarem els diferents components que trobarem al disc com son les piles (*stacks*), manifests, paquets, i les especificacions dels missatges i serveis. A continuació descriurem el concepte de cadascun d'ells i la seua sintaxis.

Paquets

Són la unitat principal d'organització de software en ROS. Un paquet contindrà processos ROS (nodes), llibreries que depenen de ROS, *datasets*, fitxers de configuració i tot el que puga ser útil i lògic a un mòdul de software. Un paquet té per objectiu proporcionar una funcionalitat de la forma mínima possible, és a dir, el suficient per a que siga útil però sense que resulte massa pesat.

És senzill crear paquets, podem fer-ho a mà o amb la ferramenta roscreate-pkg. Un paquet és simplement un directori que està inclòs ROS_PACKAGE_PATH o a ROS_ROOT (variables d'entorn de ROS) i a més conté un **manifest.xml** del que parlarem més avant. Els paquets solen anar continguts en stacks.

Podem fer servir les següents ferramentes que ens faciliten la feina amb els paquets:

- **rospack:** Obté informació sobre els paquets. El compilador de ROS fa servir esta ferramenta per localitzar els diferents paquets i compilar les seues dependències.
- **roscreate-pkg:** Crea un nou paquet.
- **rosmake:** Compilar un paquet i les seues dependències.
- **rosdep:** instal · la les dependències del sistema d'un paquet.
- **rxdeps:** visualitza les dependències d'un paquet com si es tractara d'un graf.

Manifest

El manifest (**manifest.xml**) és una especificació mínima sobre un *package* i suporta una gran varietat de ferramentes ROS, des de la compilació a la documentació i distribució. A més de proporcionar una especificació mínima de metadades sobre el paquet, una tasca important dels manifestos és la de declarar les dependències

d'una manera independent i neutral al llenguatge i sistema operatius utilitzats. La presència d'un arxiu **manifest.xml** en un directori es important ja que qualsevol directori ROS que incloga este arxiu es considera un paquet.

El fitxer de manifest mínim es com un fitxer *readme* típic, on s'indica qui ha escrit el paquet i baix de quina llicència ha sigut alliberat. La llicència és important ja que els paquets són mitjans pels qual es distribueix el codi ROS. Els arxius de manifest més comuns inclouen etiquetes **<depend>** i **<export>** que ajuden a gestionar la instal · lació i l'ús del paquet.

L'etiqueta **<depend>** apunta a un altre paquet ROS que ha de ser instal · lat. Pot tindre diversos significats depenent del contingut del paquet al que estiga assenyalant. Per altra banda, l'etiqueta **<export>** incorpora *flags* especifiques del llenguatge sobre el que estem utilitzant ROS a l'hora de compilar. Per exemple, per a un paquet que contiga roscpp (la implementació sobre C++ de ROS), l'etiqueta export haurà de declarar els fitxers de capçalera i les biblioteques que han de ser recollides per altres paquets que depenen d'ell.

Piles (Stacks)

En ROS els paquets estan organitzats en *stacks*. Considerant que l'objectiu dels paquets és el de crear col · leccions el més xicotetes possibles de codi fàcilment reutilitzable, l'objectiu de les piles és el de simplificar el procés de compartir este codi. Les piles són el mecanisme principal per a la redistribució del codi ROS. Cada pila du associada una versió i pot declarar dependències a altres piles. Estes dependències també declaren un nombre de versió, la qual cosa proporciona major estabilitat en el desenvolupament.

Les piles agrupen paquets que conjuntament ofereixen una funcionalitat, com per exemple la *navigation stack* o la *manipulation stack*. A diferència de les tradicionals biblioteques de software que estan enllaçades en temps de compilació, estes piles poden proveir funcionalitat en temps d'execució via tòpics i serveis.

Una vegada explicat açò, és senzill crear piles a mà. Una pila és simplement un directori que es troba al ROS_ROOT o ROS_PACKAGE_PATH i té un fitxer **stack.xml** al seu interior. Qualsevol paquet en este directori està considerat part de la pila. Per a propòsits d'alliberament, pot afegir-se un *CMakeList.txt* i un *Makefile* a l'arrel de la pila. La ferramenta **roscreate-stack** pot generar estos fitxers automàticament.

Stack Manifest

A l'igual que els manifestos dels paquets, els manifestos de piles (en anglés, *stack manifest* – **stack.xml**) són una especificació mínima que dona suport a la distribució

i instal · lació de piles ROS. A més de proporcionar metadades sobre la pila, un objectiu important d'estos manifestos és el de declarar les dependències sobre altres piles. La presència d'un fitxer **stack.xml** en un directori és significativa ja que qualsevol directori que es trobe al ROS_PACKAGE_PATH i que contiga este fitxer està considerat com una pila, i qualsevol paquet per baix d'esta serà considerarà part d'ella. Tot el que havíem dit sobre la forma dels manifestos normals és aplicable a estos *stack manifests*.

Descripció dels missatges (msg)

ROS utilitza un llenguatge simplificat de descripció de missatges per descriure el valor de les dades que un node publicarà. Esta descripció facilita el treball de generar automàticament el codi font per al tipus de missatge en els diferents llenguatges finals que fan ús de ROS. Estes descripcions de missatges es troben emmagatzemades en els fitxers **.msg** en el subdirectori **msg/** dels paquets ROS.

Hi ha dos parts d'un fitxer .msg: els camps i les constants. Els camps són les dades que s'envien dins del missatge. Les constants defineixen valors útils per interpretar estos camps, com per exemple el valor de pi o del radi de les rodes.

Tipus primitiu	Serialització	C++	Python
bool	unsigned 8-bit int	uint8_t	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int
int16	signed 16-bit int	int16_t	int
uint16	unsigned 8-bit int	uint16_t	int
int32	signet 32-bit int	int32_t	int
uint32	unsigned 32-bit int	uint32_t	int
int 64	signed 64-bit int	int64_t	long
uint 64	unsigned 64-bit int	uint64_t	long
float32	32-bit ieee float	float	float
float64	64-bit ieee float	double	float
string	ascii string	std::string	string
time	sec/nsecs signed 32-bits ints	ros::Time	rospy.Time
duration	sec/nsecs signed 32-bits ints	ros::Duration	rospy.Duration

Taula 4.1: Relació entre les diferents representacions de msg.

Descripció dels serveis (srv):

A l'igual que per als missatge, ROS utilitza un llenguatge simplificat de descripció de serveis (*srv*) per descriure els tipus de servei ROS. Açò és basa directament en

el format de missatges ROS per a permetre la comunicació petició/resposta entre nodes. Les descripcions dels serveis es troben en els fitxers **.srv** i en el subdirectori **srv/** de cada paquet i consistixen en un fitxer de text senzill amb una petició i una resposta dividides per una línia '—'. Este seria un exemple de servei que pren com a entrada una *string* i dona com a resposta una altra.

```
string str
---
string str
```

4.3.2 Còmput

A este nivell explicarem els components que trobarem presents al nivell de xarxa *peer-to-peer*. Tots estos elements participen en el còmput de dades i transmissió d'informació fent ús de l'estructura de graf que ens proporciona ROS.

Nodes

Un node és un procés que realitza un còmput. Els nodes es comuniquen pel graf mitjançant els tòpics, serveis RPC i el Servidor de Paràmetres del que parlarem més endavant. Estos nodes estan destinats a operar a un nivell modular, així, un sistema de control d'un robot comprendrà diversos nodes. Per exemple, un node controlarà el làser, mentre un altre controlarà el motor de les rodes, un altre la localització i un altre la planificació de la ruta, entre altres.

L'ús dels nodes que fa ROS proporciona diversos beneficis per al conjunt del sistema. Hi ha tolerància a errades, ja que els incidents que puguen afectar al sistema ho faran de forma aïllada als diferents nodes individuals. La complexitat del codi es troba molt reduïda en comparació als sistemes monolítics, a més, el detall de les implementacions es troba ben protegit ja que els nodes exposen una API mínima per a la resta dels nodes del graf o implementacions alternatives, així que fins i tot pot ser substituït el codi del propi node sense que afecte als qui es relacionen amb ell.

Com no podia ser d'altra manera, tots els nodes **s'escriuren** utilitzant les llibreries client de ROS, com poden ser **roscpp** per a C++, **rospy** per a Python o **rosjava** per al llenguatge de Sun. Recomanem estudiar les diferents API's[4] ja que són extenses però imprescindibles per entendre la forma de treballar en ROS i com es relaciona amb el llenguatge final. Per altra banda, podem **accedir** als nodes mitjançant la ferramenta de línia de comandes **rosnode** per mostrar informació de *debug*, incloent publicacions, subscripcions i diferents connexió. Adjuntem una taula on es mostren les diferents utilitats d'esta ferramenta:

[4]Vista general del roscpp: http://www.ros.org/wiki/roscpp/Overview

rosnode info	Proporciona informació sobre el node.
rosnode kill	Mata un node en execució.
rosnode list	Mostra una llista dels nodes en actiu.
rosnode machine	Mostra un llistat dels nodes en actiu d'una màquina.
rosnode ping	Prova la connectivitat a un node en concret.

Taula 4.2: Utilitats de rosnode.

ROS Master

El ROS *master* proporciona un servei de noms i de registre per a la resta dels nodes en el sistema ROS. Fa un seguiment dels *publisher* i *subscribers* dels tòpics, així com dels serveis. El paper del mestre és permetre que els nodes individuals ROS puguen localitzar-se entre ells. Una vegada que aquests nodes és troben un a l'altre, es comuniquen entre si *peer-to-peer*. El master també proporciona l'anomenat Servidor de Parametres. Per últim, cal indicar que la forma més comú de llançar el *master* és utilitzant l'ordre **roscore** , la qual carrega el mestre a més d'altres components essencials.

A continuació mostrarem un exemple per il · lustrar d'una manera més clara la tasca que realitza el *master*. Suposarem que tenim dos nodes: el node d'una camèra i un node "image_viewer". Una seqüència tipica d'esdeveniments comença amb el node de la camèra notificant al master que vol publicar imatges en el tòpic "images" **(Figura 4.1)**. A continuació, el node "camera" publicarà imatges al topic "images", però ningú està subscrit a este topic ja que ningun tipus de dada està siguent enviat. Arribats a este punt, el node "image_viewer" vol subscriure's al topic "images" per vore si hi ha alguna imatge per a obtenir. (**Figura 4.2**) Finalment, ara que el tòpic "images" te un publicador i un subscriptor, el node mestre notificarà al node "camera" i "image_viewer" sobre l'existència de l'altre i així poden iniciar la transferència d'imatges d'un a l'altre (**Figura 4.3**).

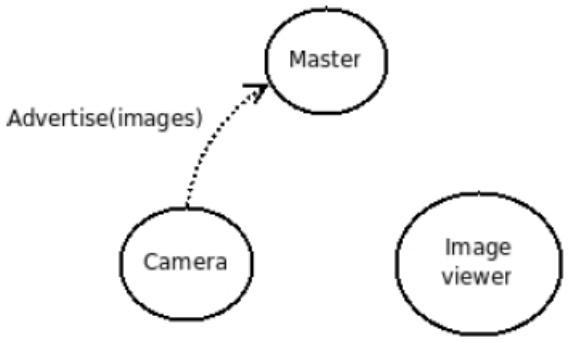

Figura 4.1: Exemple de connexió del master 1/3

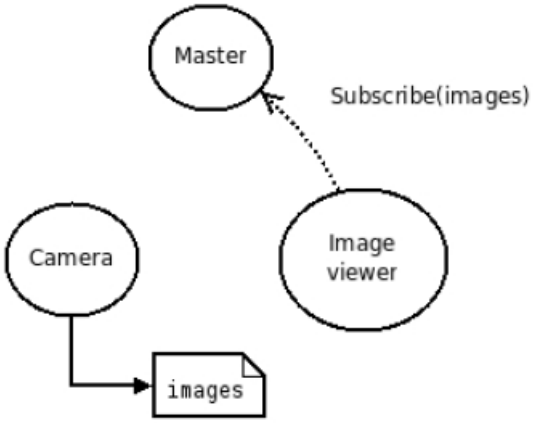

Figura 4.2: Exemple de connexió del master 2/3

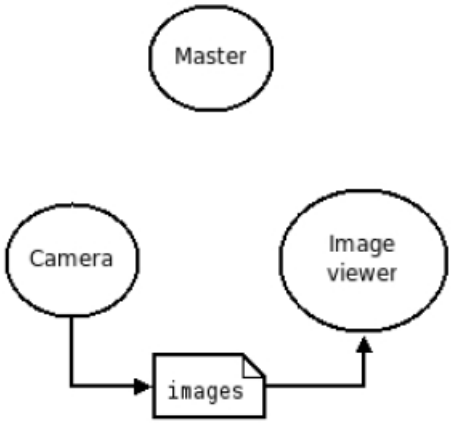

Figura 4.3: Exemple de connexió del master 3/3

Servidor de paràmetres

El servidor de paràmetres es un diccionari multi-variat i compartit que és accessible via una API de xarxa. Els nodes usen este servidor per emmagatzemar i rebre paràmetres en temps d'execució. Ja que no està dissenyat per a alt rendiment és millor utilitzar-lo en aplicacions estàtiques, no de dades binaries, com per exemple, paràmetres de configuració. Està pensat per ser visible globalment i que les eines puguen inspeccionar i modificar l'estat del sistema conforme siga convenient. El servidor de paràmetres està implementat usant **XML-RPC** i s'executa a l'interior del Master, el que significa que la seua API es accessible normalment via biblioteques **XML-RPC**.

Missatges

Com ja hem indicat, els nodes es comuniquen uns amb els altres publicant missatges en tòpics. Un missatge és simplement una estructura de dades que comprén camps fortament tipats. Els tipus estàndard primitius (*int, float, bool,* etc.) estan suportats i a més existeixen els arrays de tipus primitius. Els missatges poden incloure estructures niuades i arrays (com els *structs* de C). Tota està descripció s'inclourà en el directori **/msg** i en fitxers **.msg**. Per altra banda, és interessant presentar la ferramenta de línia de comandes **rosmsg**, que ens permet obtindre, com moltes altres, informació sobre els missatges.

rosmsg show <type>	Mostra els camps d'un tipus de missatge.
rosmsg users <type>	Busca paquets que utilitzen un tipus de msg concret.
rosmsg packages	Mostra un llistat de tots els paquets amb missatges.
rosmsg package <pkg>	Mostra un llistat de tots els msg utilitzats pel paquet.
rosmsg md5 <type>	Obté el md5 sum d'un missatge.

Taula 4.3: Utilitats de la ferramenta rosmsg.

Tòpics

Els tòpics són busos amb un nom sobre els qual els nodes intercanvien missatges. Els tòpics proporcionen una semàntica anònima de *publicació/subscripció*, la qual cosa separa la producció d'informació del seu consum. En general, els nodes no saben amb qui s'estan comunicant. En lloc d'això, els nodes que estan interessants en un tipus de dada es subscriuen al tòpic que els interessa i els nodes que generen informació la publiquen sobre un tòpic. Poden haver múltiples publicadors i subscriptors a un tòpic. Els tòpics estan destinats a la comunicació unidirecional en *streaming*. Els nodes que necessiten realitzar crides a procediments remots, com per exemple, respondre a una petició, deuen utilitzar els serveis en compte dels nodes.

Cada tòpic està fortament tipat pel tipus de missatges que hem definit anteriorment. El *Master* no comprovarà la consistència de tipus en els publicadors, ara bé, els subscriptors no establiran la connexió de no ser que el tipus de missatge esperat i el que es transportat pel tòpic coincidisquen.

Pel que fa al transport d'informació, ROS suporta el transport de missatges sobre TCP/IP i UDP. El transport basat en TCP/IP és conegut com TCPROS i proporciona missatges d'informació continua sobre connexions persistents TCP/IP. Per altra banda, el transports basat en UDP, que es conegut com UDPROS només està suportat a dia de hui per roscpp i funciona separant els missatges a enviar en paquets UDP. UDPROS es un tipus de transport per a baixes latències i transport no-crític, així que és més adequat per propòsits com la teleoperació de robots mòbils. Els node

ROS negociaran el transport desitjat en temps d'execució. Per exemple, si un node prefereix utilitzar UDPROS però l'altre no el suporta, hauran d'utilitzar TCPROS que si que està suportat pels dos.

Com és habitual, presentem una taula on expliquem de manera resumida les funcionalitats de la ferramenta de línia de comandes per gestionar els tòpics, **rostopic**.

rostopic hz <nom topic>	Mostra la ratio de publicació d'un tòpic en Hz.
rostopic echo <nom topic>	Mostra les dades que circulen per pantalla.
rostopic bw <nom topic>	Mostra l'amplada de banda utilitzada pel tòpic.
rostopic find <msg type>	Busca topics per tipus.
rostopic list	Mostra tots els topics presents als sistema.
rostopic pub <nom topic>	Publica informació a un tòpic en concret.
rostopic type <nom topic>	Mostra el tipus de topic.
rostopic info <nom topic>	Publica informació sobre un topic en concret.

Taula 4.4: Utilitats de la ferramenta rostopic.

Serveis

El model basat en publicadors/subscriptors és un paradigma de comunicació molt flexible, tot i que el seu transport molts-a-molts unidireccional no és el més adequat per a les peticions/respostes **RPC** requerides habitualment pels sistemes distribuïts. Les peticions/respostes són efectuades via serveis (*services*), els quals queden definits per un parell de missatges: un per la petició i un altre per la resposta. Un node ROS pot oferir un servei baix un nom (*string*). El client cridarà al servei enviant el missatge de petició i esperant a la resposta. Les llibreries client solen representar esta interacció al programador com si d'una crida de procediment remot es tractara. El serveis, com hem vist en la secció anterior, queden definits per els fitxers **.srv**, els quals són compilats en el codi font per les biblioteques de clients ROS (**roscpp**, **rospy**, ...).

Un client pot establir una connexió persistent a un servei, el qual establirà un alt rendiment a canvi de menys solidesa als canvis del proveïdor de serveis. Tot i que l'ús de serveis, almenys en la nostra experiència no ha sigut massa habitual, creguem convenient presentar les utilitats que ens proporciona l'eina de terminal **rosservice**.

rosservice args <name>	Mostra els arguments necessaris per cridar un *srv*.
rosservice call <name>	Crida al servei amb els arguments indicats.
rosservice node <name>	Mostra els nodes que fan ús d'un servei determinat.
rosservice find <type>	Busca serveis per tipus.
rosservice list	Mostra tots els serveis presents als sistema.
rosservice type <name>	Mostra el tipus de servei.
rosservice info <name>	Publica informació sobre un servei en concret.

Taula 4.5: Utilitats de la ferramenta rosservice.

4.3.3 Alt nivell

Com hem vist en seccions anteriors, ROS ens proporciona uns conceptes i una arquitectura clara de com ha d'organitzar-se el sistema. En el primer nivell hem vist com ROS descriu l'estructura i els diferents components, per a després, en un segon nivell vore com estos components interactuen entre si en temps d'execució. No obstant això, i encara que ROS intente estar el més allunyat com siga possible de conceptes complexos, existeixen utilitats i conceptes d'alt nivell que no poden ser englobats en cap de les seccions anteriors ja que agrupen diferents ferramentes i conceptes d'estes. Estos ens faciliten molt la feina, per això és necessària la seua implementació. A continuació explicarem els més importants:

Transformacions

Els sistemes robotitzats sovint necessiten fer un seguiment sobre les seues relacions espacials per diverses raons, ja siga entre un robot mòbil i algun marc (o *frame*, anem a utilitzar els dos termes de forma indiferent) de referència fixe per a la localització, com entre els diferents *frames* de sensors i actuadors fins a col · locar *frames* de referència com a objectius en finalitats de control.

Per unificar i simplificar el tractament dels marcs espacials ROS incorpora un sistema de transformacions anomenat **tf**. El sistema **tf** construeix un arbre de transformacions dinàmic que relaciona tots els *frames* de referència del sistema. A l'igual que els fluxes d'informació que provenen dels diferents subsistemes del robot (encoders, algorismes de localització, etc) el sistema **tf** produeix fluxos de transformacions entre els nodes de l'arbre, construint una ruta entre el node desitjat i efectuant els càlculs necessaris. En altre paraules, des d'un nivell abstracte podem vore com l'arbre de transformacions defineix les compensacions en termes de translació i rotació entre els diferents marcs de coordenades. Este tipus de càlculs sol ser pesat, amb tendència a l'error i dificil de depurar quan està programat a mà, no obstant, la implementació d'este sistema de **tf**, combina amb la estructura dinàmica de pas de missatges de ROS, permet una aproximació automatitzada i sistemàtica al problema.

Per a clarificar tot açò anem a posar un exemple. Considerem un robot mòbil amb un làser muntat sobre ell. Ens referirem als dos objectes mitjançant dos marcs coordenats, un que es correspon al punt central de la base del robot i que anomenarem "base_link" i altre que es correspon al punt central del làser i que anomenarem "base_laser". En aquest punt, suposem que tenim algunes dades del làser en la forma de distàncies des del centre del làser. En altres paraules, tenim algunes dades al sistema de coordenades de "base_laser". Suposem que volem utilitzar esta informació per a ajudar a la base del robot mòbil a evitar obstacles mentre navega. Per fer açò amb èxit, necessitem una forma de transformar la informació que ens arriba del frame "base_laser" al frame "base_link". En resum, necessitem definir una relació entre els dos marcs coordenats.

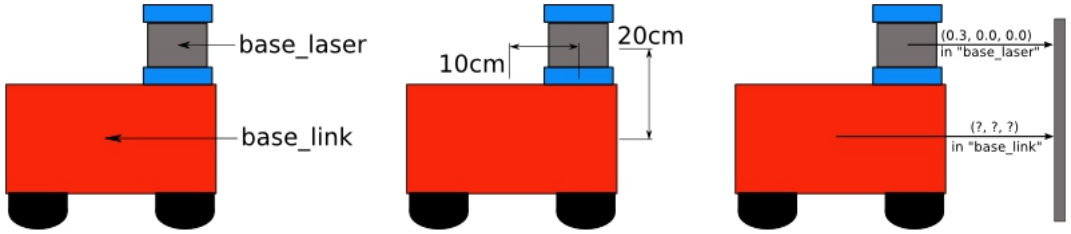

Figura 4.4: Situació dels diferents marcs coordenats.

Definint la relació, hem d'assumir que sabem que el làser està montat 10cm per davant i 20cm per damunt del punt central de la base del robot mobil. Açò ens proporciona una translació que ens relaciona els dos marcs de "base_link" a "base_laser": (x: 0.1m, y: 0.0m, z: 0.2m).

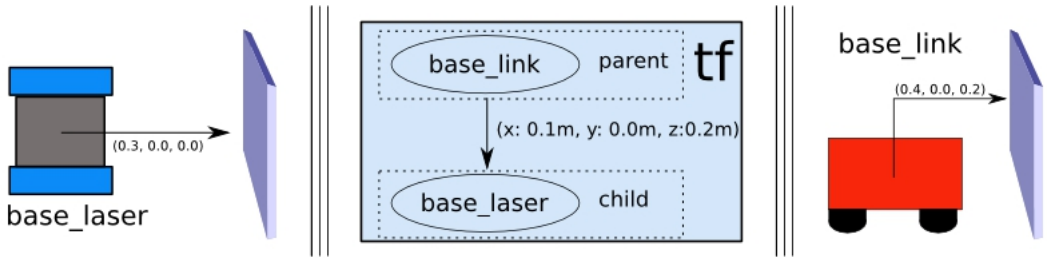

Figura 4.5: Efecte de les tf sobre un objecte situat a 0.3 cm.

Arribats a este punt, podem triar de gestionar aquesta transformació nosaltres mateixa o deixar que **tf** treballe per nosaltres. Una vegada definida esta relació entre "base_link" i "base_laser" hem d'afegir-la a l'arbre de transformacions. Conceptualment, cada node en este arbre correspon a un marc coordenat i cada aresta es correspon a la transformació necessària per moure's d'un node al fill. **Tf** utilitza una estructura d'arbre per garantir que només hi ha un únic recorregut que uneix els dos marcs coordenat i suposa que totes les arestes en l'arbre es dirigeixen dels pares als nodes fills.

Visualització i monitorització

Quan estem dissenyant i depurant software robòtic, sol ser necessari observar l'estat del sistema conforme s'executa. Tot i que **printf** és una tècnica comú per la depuració de programes en una maquina senzilla, pot ser difícil exportar este procediment a un sistema distribuït de gran escala i a més pot resultar molt poc eficient per a supervisar software d'una forma general i sistemàtica. En lloc d'açò, ROS explota la seua naturalesa dinàmica de connexió en graf per "aprofitar" qualsevol flux de missatges en el sistema. A més a més, la independència entre publishers i subscribers permet la creació de visualitzadors d'ús general. Es poden escriure programes molt senzills que es subscriguen a un tòpic en concret i que mostren per pantalla un tipus de dada, com puguen ser imatge o fluxos provinents del làser. Tanmateix, un concepte més evolucionat de tot açò ens el proporciona **rviz**, un programa de visualització que utilitza una arquitectura de plugins i que es distruit conjuntament amb ROS. Els panells de visualització poden ser instanciats dinàmicament per visualitzar una gran quantitat de dades, com per exemple imatges, núvols de punts, primitives geomètriques, posicions renderitzades del robot, etc. Estos plugins poden ser escrits fàcilment per visualitzar els nostres propis tipus de dada.

Per altra banda, gràcies a Python s'ha escrit una de les ferramentes més potents de ROS, **rostopic**, la qual s'encarrega de filtrar missatges utilitzant les expressions introduïdes en la línia de comandes, donant com a resultat un capturador de missatges configurable que pot convertir qualsevol fluxe de dades en flux de text. Estos fluxes de text poden ser entubats a altres ferramentes UNIX com *grep, sed, i awk* per crear complexes ferramentes de monitorització sense escriure cap línia de codi. D'una forma similar, **rxplot** proporciona la funcionalitat d'un oscil · loscopi virtual, mostrant l'estat de qualsevol variable per pantalla en temps real.

Per últim i no per això menys important, presentem la ferramenta **rxgraph**. Esta ferramenta ens permet visualitzar i inspeccionar gràficament l'estructura de node del sistema en execució. La seua eixida, generada dinàmicament, mostra els nodes com a ovals, els tòpics com a rectangles i les connexions com a arcs. És molt útil ja que clarifica l'estructura i ens permet vore l'evolució del sistema conforme aquest evoluciona. Més endavant serà utilitzada per explicar l'arquitectura del sistema resultant.

Model URDF

URDF[5] és una especificació **XML**[6] per a la descripció de robots. S'intenta mantindre el més general com siga possible, però açò té certes limitacions, com el no poder

[5]Unified Robot Description Format

[6]Extensible Markup Language. Metallenguatge extensible d'etiquetes, desenvolupat pel W3C.

representar tots els tipus de robots o components d'estos. La principal limitació que ens trobem és que només es poden representar robots que seguisquen una estructura d'arbre, deixant fora els robots paral · lels. A més a més, l'especificació assumeix que el robot està format per cosos rígids (*links*) units per articulacions (*joints*). Els elements flexibles no estan suportats. L'especificació dona suport a:

- La representació cinemàtica i dinàmica del robot.
- La representació visual del robot.
- El model de col · lisions del robot.

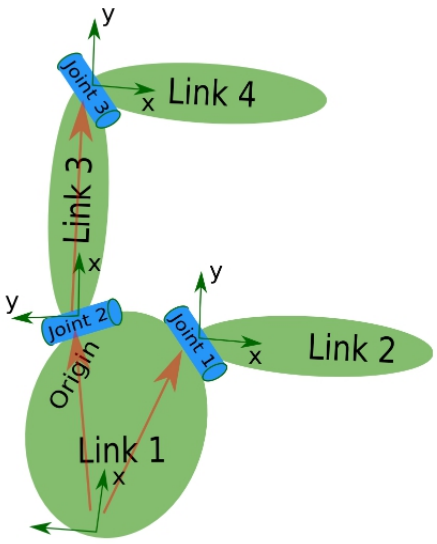

Figura 4.6: Esquema visual d'una configuració URDF.

La representació d'un robot consisteix en un conjunt de *links* i *joints* que es connecten. Un fitxer típic de representació **URDF** d'un robot tindrà esta estructura.

```
<robot name="pr2">
<link> ... </link>
<link> ... </link>
<link> ... </link>
<joint> .... </joint>
<joint> .... </joint>
<joint> .... </joint>
</robot>
```

Per a una visió completa del que és i com s'utilitza el URDF, recomanem visitar la pàgina de referència[7] XML del format. Més endavant a esta mateixa memòria serà explicat com crear i llançar un URDF del robot que hem simulat.

[7]http://www.ros.org/wiki/urdf/XML

5 Simulació

5.1 Introducció

La robòtica és un camp que genera gran interés, malgrat això, la inversió que hi ha que fer per accedir al hardware resulta, de forma general, prohibitiva. La simulació per ordinador és una alternativa vàlida i de baix cost per als qui consideren l'esforç econòmic un impediment. A més a més, no només per això ens pot interessar la simulació per ordinador, sinó que hi ha raons addicionals que motiven a interessar-se per esta:

- **Mobilitat:** Des d'on volem podem simular qualsevol entorn sense haver d'estar present. Per exemple, podem estar a una oficina sense cap robot present mentre estem dissenyant el sistema de control d'un grup de robots a un entorn industrial.

- **Adaptabilitat:** Les modificacions que apliquem al robot o a l'entorn no tenen cost econòmic.

- **Gestió de recursos:** Un simulador, utilitzat junt a una capa de abstracció de hardware, permet desenvolupar software per a un robot que existeix però al que no es té accés en un determinat moment. Açò és especialment útil quan hi ha més d'un desenvolupador i un únic robot.

- **Protopipat:** Mitjançant la simulació, és possible implementar el software d'un robot abans que este haja sigut construït. Açò és útil per a realitzar qualsevol tipus de prova abans de que el robot real es construïsca.

- **Reusabilitat:** Ja que l'ús de simuladors sol estar acompanyat d'algun tipus de capa d'abstracció, el codi generat pot ser compartit i utilitzat per altres persones.

- **Control del temps:** Alguns simuladors permeten parar o accelerar el temps, ja siga avant o arrere, de la simulació. Açò pot ser útil per a inspeccionar l'estat dels robots en certs moment o per a fer proves de durabilitat.

Tipus de robot	Simuladors
Robots amb rodes	Carmen, Player/Stage Project, Xraptor, Gazebo, etc.
Braços robotitzats	RobotStudio, SimRobot, Kuka.Sim Pro, RobotWorks
Robots amb cames	Virtual KHR-1
Robots submarins	DeepWorks, Neptune Simulator

Taula 5.1: Simuladors per a robòtica

Podem trobar una gran quantitat de simuladors, estos els hem organitzat en diverses seccions: robots amb rodes, braços robotitzats, robots amb cames i robots submarins.

Una vegada explicats els avantatges de la simulació per a robots i mostrar un llistat dels més importants, en centrarem en el que hem utilitzat per al nostre projecte: **Gazebo**.

5.2 Gazebo

Gazebo és un simulador multi-robot tant per a entorns interiors com exteriors. Com Stage, és capaç de simular poblacions de robots, sensors i objectes però a diferència de l'anterior, Gazebo ho simula en un món tridimensional. A més a més, este simulador genera *feedback* realista dels sensors, col · lisions i dinàmica d'objectes. Cal indicar primer que res, que comparant-lo amb altres simuladors, Gazebo encara es troba en un estat poc madur (actualment es troba en la versió 0.8), però així i tot, donat que és software lliure, és 3D i més realista i complet que altres, fa que siga una opció molt interessant.

Gazebo és normalment utilitzat junt al servidor de dispositius Player. Este proporciona l'abstracció d'un mecanisme central de xarxa (un servidor) sobre el que el els controlador del robot (clients) poden interactuar sobre el hardware del robot. Quan s'utilitza amb Gazebo, Player proporciona dades simulades en lloc de dades provinents dels sensors reals. En principi, els programes client no tenen perquè notar la diferència entre els dispositius reals i els simulats per Gazebo.

Este simulador també pot ser controlat mitjançant una **API C** de baix nivell (*libgazebo*). Esta llibreria està inclosa per a permetre integrar Gazebo a tercers en els seues propis servidors o arquitectures. És més, la forma en que utilitzarem nosaltres Gazebo dista molt de la comentada més amunt i és possible gràcies a esta **API C**. En el nostre cas, el simulador s'encarregarà de simular-ho tot, des de l'entorn fins al robot i els diferents sensors que l'integren. ROS es comunicarà amb ell mitjançant plugins que faran possible la integració, així com la comunicació entre controladors i altres elements en temps real. També es comunicaran mitjançant

tòpics com és habitual i on Gazebo serà un node més del graf. Este tema serà explicat més endavant en la secció 6.1.3. (***Programació del controlador P en temps real***) i en la 6.2. (***Arquitectura***).

A continuació, enumerarem les característiques més destacades d'este simulador:

- Simulació dels sensors estàndard d'un robot, incloent sonar, scanners, GPS, IMU, camares estereo i monoculars.
- Inclou models de robots ampliament utilitzats com el Piooneer2DX, Pioon-er2AT i el SegwarRMP.
- Simulació realista de la física dels cosos rígids: els robots poden espentar i agafar coses i generalment interactuar amb el món d'una manera convincent.
- Compatible amb Player: els robots i els sensors poden ser controlars desde interficies estàndards de Player.
- Autònom: els diferents programes poden interactuar directament amb el simulador (és a dir, sense passar per Player) utilitzat la llibreria ***libgazebo***.
- Interficie gràfica d'usuari potent: actualment, la majoria d'elements simulats poden ser controlats o inspeccionats directament mitjançant la GUI del simulador.
- Permet skins/malles: Els models simples que dissenyem poden millorar-se amb les "skins" realistes que inclou el simulador o les nostres pròpies que podem obtenir de programes de modelat 3D.
- Gazebo és software lliure alliberat sota la llicència GNU Public Licese.

Una vegada introduit el simulador que anem a utilitzar, anem a anar un pas més enllà i a estudiar en quins motors gràfics i llibreries està basat Gazebo. Les dos més importants són OGRE[1] i ODE[2]. Per clarificar el tema, a la **Figura 5.1** mostrem un gràfic amb totes les dependències del simulador i l'arquitectura resultat, per a una visió completa del tema recomanem visitar la següent pàgina[3].

5.2.1 OGRE

OGRE és un motor flexible de renderitzat 3D escrit en **C++** i dissenyat per fer senzill i intuïtiu per als desenvolupadors el poder produir aplicacions utilitzant gràfics

[1]Object-Oriented Graphics Rendering Engine. Motor flexible de renderitzat 3D
[2]Open Dynamics Engine. Llibreria per simular la dinàmica de cosos rígids.
[3]http://www.ros.org/wiki/gazebo/Version_1.0_Design_Specification

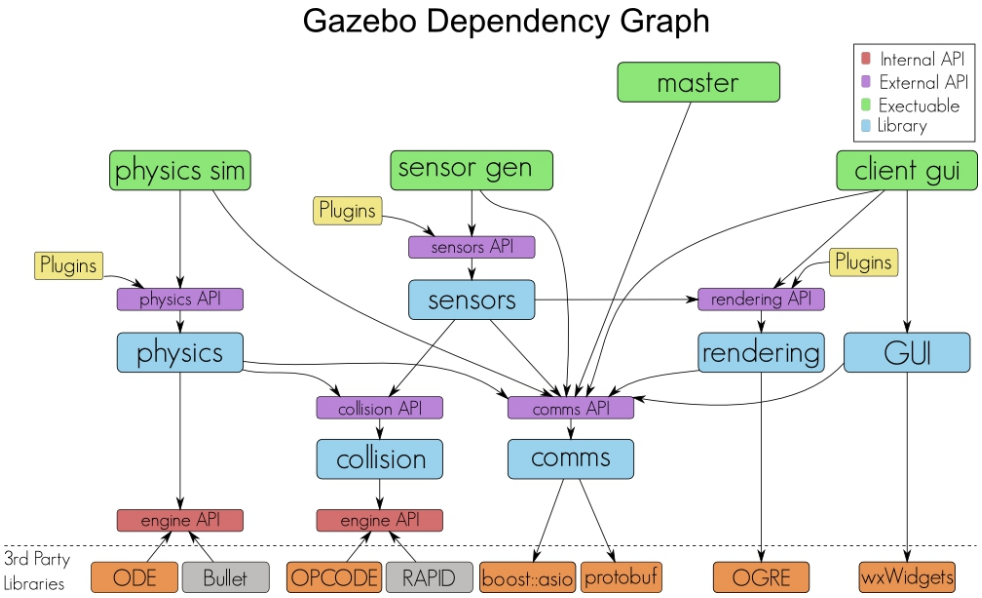

Figura 5.1: Gràfic de dependències de Gazebo

3D. La llibreria classe resumeix els detall per a utilitzar els sistemes de llibreries subjacents com **Direct3D** i **OpenGL** i proporciona una interfície basada en objectes del món així com altres objectes d'alt nivell.

Com el seu nom indica, OGRE és "només" un motor de renderitzat. Com a tal, el seu objectiu principal és el de proporcionar una solució general per a la representació de gràfics. Encara que també compta amb altres característiques (classes vectors i matrius, maneig de memòria, etc) estan considerades complementaries. No és una solució "tot en un" per a simulació ja que ni proporciona suport per a àudio ni el que és més important, per a la física.

Generalment, açò és considerat com el principal inconvenient de OGRE, però també pot ser vist com una característica del motor. El fet de triar OGRE com a motor gràfic permet als desenvolupadors la llibertat de triar el motor físic, d'introducció de dades o audio que ells vulguen i permet als desenvolupadors d'OGRE centrar esforços en els gràfics.

A continuació, enumerem les dos característiques més destacades d'este motor de *rendering*:

- OGRE disposa d'un disseny orientat a objectes amb una arquitectura basada en plugins que permet afegir fàcilment característiques, fent que siga altament modular.

- OGRE és multiplataforma i suporta **OpenGL** i **Direct3D**. Pot renderitzar el mateix contingut en diferents plataformes sense que el creador del continguts

haja de prendre en consideració les diferents capacitats de cada plataforma. Açò redueix la complexitat de lliberar un joc/simulació en diferents sistemes. Actualment existeixen els binaris precompilar per a Linux, Mac OS X i per a la majoria de versions de Windows.

- OGRE disposa d'una comunitat molt activa i existeixen grans quantitat d'informació sobre com utilitzar-lo.

5.2.2 ODE

Open Dynamics Engine (ODE) es tracta d'un motor físic escrit en C/C++. Els seus components principals són un motor de simulació de la dinamica de cosos rígids i un motor de detecció de colisions. ODE s'utilitza per simular les interaccions dinàmiques entre els elements de l'espai. No està lligat a cap paquet gràfic en particular tot i que inclou un molt bàsic anomenat *drawstuff*. Este suporta moltes formes geomètriques: caixes, esferes, capsules, *trimesh*, cilindres i *heigmaps*, ODE és una opció freqüent per a les aplicacions de simulació robòtica, permetent des de la simulació de la locomoció de robots mòbils fins al *grasping*.

A més de tots estos detalls tècnics cal indicar que ODE és software lliure i està alliberat baix les llicències **BSD** i **LGPL**. A més d'açò, conta amb una forta i activa comunitat encarregada d'ajudar als iniciats, proposar millores o inclús escriure les seues pròpies. També disposa d'una wiki[4] molt completa amb gran quantitat de manuals i guies.

Per últim, indicar que des de que es va llançar ODE en 2001, gran quantitat de jocs i aplicacions han utilitzat este motor físic, com per exemple, *BloodRayne 2, Call of Juarez, S.T.A.L.K.E.R, Titan Quest, World of Goo, X-Moto* i *OpenSimulator*. Com vegem, el software lliure no està per a res renyit amb la professionalitat.

[4]http://opende.sourceforge.net/wiki/index.php/Main_Page

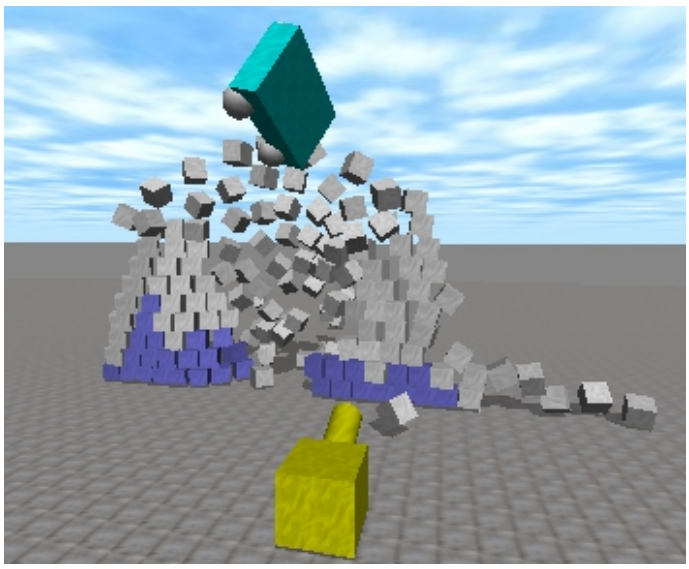

Figura 5.2: Col · lisió de diversos objectes. Esta demo està inclosa en els fitxers fonts de ODE.

6 Implementació

Una vegada estudiats els diferents components que integren el nostre sistema i compresos els fonaments en que es basen podem començar a descriure com ha sigut la implementació del projecte. Ací tractarem d'una manera guiada i detallada l'evolució del treball, des de la instal · lació de ROS fins a la integració de la *Navigation Stack*, així com la creació de diferents móns on fer les proves de Navegació. No obstant, abans d'entrar en detalls, senyalarem l'imprescindible per al desenvolupament del projecte:

- Ubuntu Maverick Meerkat 10.10
- PC amb targeta gràfica GeForce 430 GT d'1GByte i el driver corresponent
- ROS Diamondback
- Simulador Gazebo

6.1 Instal · lació

Podem instal · lar l'entorn ROS de diverses formes ja siga de forma independent (i des dels repositoris o des de les fonts) o de forma conjunta amb el software d'un dels cada volta més robots que suporten ROS.

6.1.1 Instal · lació independent des dels repositoris d'Ubuntu

Per a instal · lar ROS haurem de configurar primerament els repositoris d'Ubuntu per a que permeta els "restricted", "universe", i "multiverse". Una vegada fet açò, configurarem el nostre equip per a que permeta l'obtenció de sotfware del web ROS.org:

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

A continuació, configurarem les claus i actualitzarem el **apt** per confirmar que hem afegit correctament el nou repositori.

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
sudo apt-get update
```

Una vegada ací, podem triar entre diferents tipus d'instal · lació però recomanem la completa, que inclou ROS junt a un conjunt de ferramentes on poden trobar des de les rx, rviz, passant per stacks de navegació i percepció, fins a simuladors. Ens col · loquem en el directori **/opt** i creem un nou, **/ros**. A l'interior d'este en crearem un altre, **/diamondback** i executarem:

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

Per finalitzar, haurem d'afegir ROS a les variables d'entorn del sistema operatiu i evitar que en cada nova terminal hagem de fer-ho, podem escriure este conjunt d'ordres a la terminal:

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

6.1.2 Instal · lació des dels respositoris junt al software d'un robot

Instal · lar ROS junt a software per simulació i control d'un robot en concret no dista molt del que hem vist fins ara i a més ens proporciona, en el cas del PR2 unes llibreries que, tot i no ser-ho oficialment, acaben sent estàndards *de facto* en la programació i simulació de robots baix ROS. En el cas d'estar iniciant-se, és molt recomanable realitzar este tipus d'instal · lació.

Per a açò, l'únic que haurem de fer es dirigir-nos a *www.ros.org*, entrar a l'apartat **Install** i triar entre un dels robots que se'ns ofereixen. En la majoria dels casos l'únic que haurem de canviar serà el paquet que volem baixar, recomanem la instal · lació completa del PR2.

```
sudo apt-get install ros-diamondback-pr2-desktop
```

6.1.3 Instal · lació des del codi font

En este tipus d'instal · lació ens encarregarem primerament de descarregar el codi font per posteriorment anar compilant aquells paquets que ens siguen necessaris. Este tipus d'instal · lació només és recomanable en el cas d'haver treballat prèviament amb ROS o haja de treballar necessàriament sobre versions *trunk*[1] ja que pot resultar confús i pesat el satisfer correctament totes les dependències per a disposar d'un sistema estable.

Primerament instal · larem les ferramentes bàsiques que ens seran necessàries al llarg de la compilació.

```
sudo apt-get install build-essential python-yaml cmake subversion
wget python-setuptools mercurial
```

Tot seguit ens serà necessari **rosinstall**, una ferramenta que s'encarrega de comprovar canvis al codi font de ROS a partir de múltiples repositoris de control de versions i actualitzar-los segons convinga.

```
sudo easy_install -U rosinstall
```

Per finalitzar esta primera part, baixarem el codi que quedarà desat al fitxer que li indiquem,en el nostre cas: **/opt/ros/diamondback**. Com sempre recomanem baixar la versió completa per evitar descarregar després el màxim de paquets a mà.

```
rosinstall /opt/ros/diamondback "http://packages.ros.org/
cgi-bin/gen_rosinstall.py?rosdistro=diamondback&variant=
desktop-full&overlay=no"
```

Una vegada amb açò baixat ens serà necessari compilar executant l'orde següent. Esta, s'encarregarà de compilar el codi buscant les dependències necessàries ordenadament.

```
rosmake * --rosdep-install
```

[1] Al camp del desenvolupament de software, es refereix a la branca sense etiqueta (versió) d'un arbre de fitxers. Normalment és la base del projecte i és on els desenvolupadors incorporen les ultimes novetats.

6.2 Model URDF

El primer que férem una vegada instal · làrem el sistema i tinguérem clars els conceptes bàsics de ROS va ser crear el model URDF del robot mòbil Guardian. Per fer més senzilla la feina, començàrem per dissenyar un model simple del robot i el descomposàrem en el mínim possible de parts, quatre rodes i un bastidor. Primerament creàrem el bastidor:

```
<?xml version="1.0"?>
<robot name="guardian">

    <link name="chassis">
    <inertial>
        <mass value="20.0" />
        <origin xyz="0 0 0" />
        <inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1"/>
    </inertial>
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>
        <box size="0.5 0.75 0.2" />
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0 " />
        <geometry>
        <box size="0.5 0.75 0.2" />
        </geometry> </collision>

</link>
<gazebo reference="chassis">

    <material>Gazebo/Red</material>
    <turnGravityOff>false</turnGravityOff>

</gazebo>
</robot>
```

Com podem observar, es segueix l'estructura que hem comentat en l'apartat corresponent a l'URDF:

- **Inertial:** A esta secció definim la cinemàtica i dinàmica del cos. En este cas hem creat un bastidor de 20kg de pes amb una inèrcia estàndard definida en l'etiqueta inertia.

- **Visual:** A visual donarem forma al cos que estem creant. Ací com podem vore creem un objecte de tipus "box" (caixa) amb unes mides de 0'5 en l'eix x, 0'75 en l'y i 0'2 en el z. Açò ens crearà un rectangle 3D.

- **Colision:** En esta etiqueta definim quin volem que siga l'objecte que utilitze el simulador per al model de col · lisions. Generalment no canvia respecte al definit a l'estructura visual.

- **Gazebo:** Esta etiqueta especial per al simulador Gazebo ens permet configurar paràmetres que d'altra manera no podrien ser visualitzats amb este simulador, com per exemple el color, plugins, sensors, friccions dels diferents elements, material o inclús si està activa o no la física del simulador sobre este objecte en concret.

Una vegada creat el bastidor haguérem de crear les 4 rodes del vehicle, explicarem com creàrem una d'elles (la roda trasera esquerra, per exemple) i com la connectàrem amb el bastidor:

```
<link name="back_left_wheel">

    <inertial>
        <mass value="1.0" />
        <origin xyz="0 0 0" />
        <inertia ixx="1" ixy="0" ixz="0" iyy="1" iyz="0" izz="1" />
    </inertial>
    <visual>
        <origin xyz="0 0 0" rpy="0 0 0" />
        <geometry>

            <cylinder radius="0.125" length="0.2" />
        </geometry>
    </visual>
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0 " />
        <geometry>

            <cylinder radius="0.125" length="0.2" />
        </geometry>
    </collision>

</link>
<gazebo reference="back_left_wheel">
```

```
        <mu1 value="100.0" />
        <mu2 value="10.0" />
        <kp value="1000000.0" />
        <kd value="1.0" />
        <material>Gazebo/Grey</material>
        <turnGravityOff>false</turnGravityOff>
    </gazebo>
```

Com vegem, no hi ha diferències aparents respecte al model anterior, però si ens fixem, vegem que el pes i la forma varien ja que per a este model simplificat hem utilitzat cilindres d'un kilogram de pes per simular les rodes. També cal destacar que hem introduït per primera vegada les etiquetes **mu1**, **mu2**, **kp** i **kd**. Estes etiquetes, corresponents als coeficients de fricció, el valor de rigidesa del contacte i el de l'armortiguació, respectivament, s'utilitzen per al càlcul físic en el motor del simulador. Convé posar uns valors adequats i provar el major nombre de configuracions possibles per tractar d'ajustar-los el màxim al comportament del robot real.

Una vegada creats estos dos cossos (*links*), hem de juntar-los mitjançant una articulació (*joint*). Esta operació és una de les més dificultoses del modelat URDF i convé posar atenció per a que l'articulació resultant tinga el moviment desitjat. Presentem ací l'articulació creada:

```
    <joint name="joint_back_left_wheel" type="continuous">
        <parent link="chassis"/>
        <child link="back_left_wheel"/>
        <origin xyz="-0.35 -0.375 0" rpy="0 1.5708 0" />
        <axis xyz="0 0 1" />
    </joint>
```

A la capçalera definirem el nom i el tipus d'articulació. Podem triar entre sis tipus d'articulació (*revolute*, *continuous*, *planar*, *fixed*, *primastic* i *floating*). En el nostre cas, com volem simular una articulació sense límits superiors ni inferiors utilitzarem el tipus *continuous*. A continuació relacionarem els dos elements, en el nostre cas el pare serà l'element "chassis" creat al principi i el fill la roda trassera esquerra. Tot i que ja queden relacionats els dos cosos, en cas de deixar-ho així apareixerien els dos elements situats al centre de la simulació, per a això, definim el punt d'origen de l'articulació amb les coordenades **xyz**. També haurem de modificar a eixa mateixa etiqueta les orientacions respecte a cada angle (**rpy**), ja que de no canviar-ho tindriem la roda i el bastidor a un mateix planol. Per últim definim la direcció dels eixos en el marc de referència de l'articulació.

Una vegada explicat açò, només hem de repetir el procés en les tres rodes restants per conseguir un model aproximat del robot a simular. El resultat esquematitzat i al simulador serà el següent:

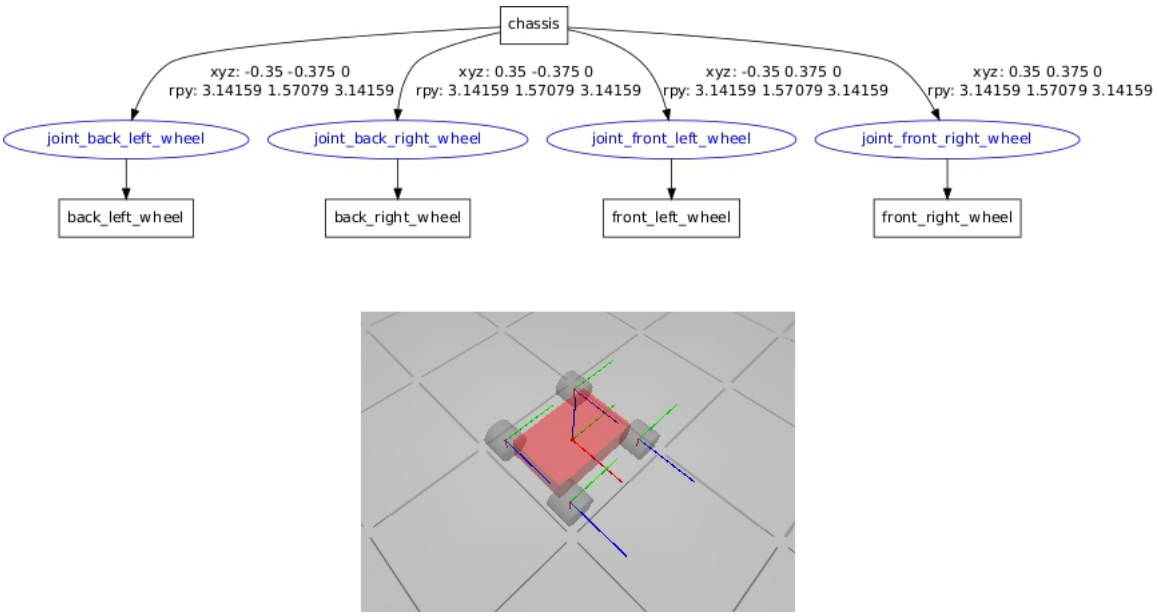

Figura 6.1: Primera versió del URDF de forma esquematitzada i al simulador Gazebo.

Arribats a este punt és convenient revisar l'aspecte visual i físic del robot. Gràcies a ferramentes de disseny assistit per computador, com per exemple SolidWorks o AutoCAD podem obtenir fitxers **.stl**. Este tipus de fitxers, descriuen únicament la geometria de la superfície del model, sense cap resprentació del color, la textura o altres artibuts CAD comuns, no obstant, ens donen un aspecte més aproximat del que és l'aspecte del robot real. Gràcies també a estos programes podem obtenir fàcilment les matrius d'inèrcia de cada objecte, cosa que ens serà imprescindile per a una simulació realista.

Per a incloure estos canvis només hem d'afegir i canviar unes quantes línies. En el cas de la inèrcia posarem a l'etiqueta "*inertia*" de l'apartat "*inertial*" els valors corresponents a cada sòlid que hagem obtés. Per altra banda, respecte als elements visuals haurem de canviar el que abans eren "*box*", "*cylinders*" o "*sphere*" per l'etiqueta "*mesh*", que ens permet introduir una malla com l'arxiu **.stl** que abans comentàvem a l'escala que desitgem. L'etiqueta "*geometry*" de l'apartat visual i de col · lisions quedarà d'una forma semblant a açò:

```
<geometry>
<mesh filename="package://guardian_description/meshes/chassis.stl"/>
</geometry>
```

Amb estos senzill canvis aconseguirem un model més realista tant física com visualment:

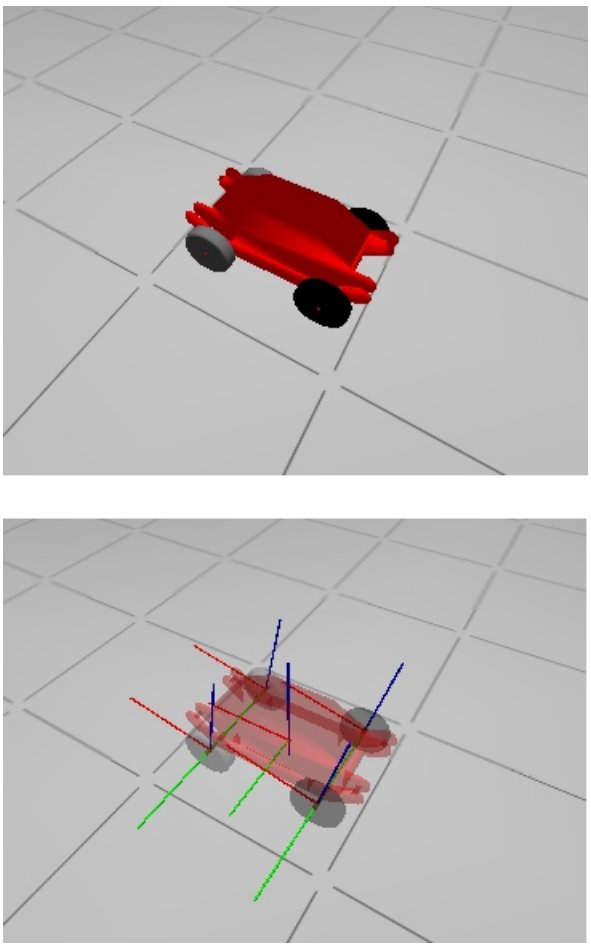

Figura 6.2: Model URDF millorat. Vista normal i vista de la física amb Gazebo.

6.3 Programació del controlador P en temps real

En esta secció ens ocuparem de la programació del controlador en temps real per al moviment del robot. Esta secció pot ser siga la part més complicada del procés de programació del nostre robot en ROS i més encara en quan estem fent-ho sobre una simulació ja que es combinaran conceptes avançats de ROS amb altres de Gazebo per poder relacionar els dos sistemes. Per fer més comprensible esta secció anem a dividir-la en dos parts.

A la primera descriurem de manera detallada la stack ***pr2_mechanism***. Com ja hem comentat abans, molts dels elements que incorpora este robot acaben sent l'estàndard *de facto* en la programació d'altres i ací tinguem el millor exemple. Esta classe s'encarrega de definir l'estructura d'un control en temps real sobre ROS. Al següent apartat explicarem de forma pràctica el nostre cas i com ho férem per seguir l'estàndard anterior i aplegar a disposar d'un controlador per a les rodes del robot i que este fora controlable mitjançant referències de velocitat.

6.3.1 La pila pr2_mechanism

Esta pila conté la infraestructura necessària per controlar el robot ***PR2*** (o qualsevol similar) en un bucle de control en temps real. Conté biblioteques molt útils per escriure un controlador en temps real amb el que es puga interactuar amb ell. Les biblioteques que ens seran més útils estan contingudes en el següents paquets:

- pr2_controller_interface: L'interfície en C++ per a un controlador en temps real.
- pr2_controller_manager: Infraestructura que permet executar i controlar diferents controlador en un bucle en temps real.
- pr2_mechanism_model: Implementa el model d'un robot controlat per esforç, per això, es basa en el model URDF d'aquest.
- pr2_hardware_interface: L'interfície en C++ per al hardware del PR2, des d'actuadors fins a accelerometres, sensors de pressió i més. Només útil en cas de no estar simulant.

Podem vore dues parts clarament diferenciades però imprescindibles a l'hora d'escriure un controlador d'este tipus en ROS, una en la que es tracta la gestió del temps real (paquets ***pr2_controller_interface*** i ***pr2_controller_manager***) i una altra on es controla la part física del robot (paquets ***pr2_hardware_interface*** i ***pr2_mechanism_model***).

Pel que fa a la **gestió del temps real** hem d'indicar que ROS no és, en principi, un framework que suporte al 100% el temps real. Ara bé, existeixen ferramentes com el *Controller Manager* (paquet ***pr2_controller_manager***) que ens permet executar una infraestructura capaç de executar controladors en un bucle de temps real. En cada cicle d'este bucle, tots els controladors carregats al *Controller Manager* seran llançat segons l'ordre establert per un planificador[2] intern que este proporciona. Esta ferramenta proporciona servicis ROS per poder carregar, llançar, parar i descarregar els controladors als CM. Vist des d'una altra perspectiva, tot el graf seguirà executant-se i passant-se informació de forma habitual en un entorn de temps no-real, mentre que per una altra disposarem d'un xicotet mòdul on tot el que s'execute dins satisfarà les regles del temps real. La comunicació entre el que s'executa a l'interior del CM i el que hi ha fora d'este entorn de temps real es farà mitjançant el tòpic **/joint_state**, on es publicarà un missatge del tipus **sensor_msgs/JointState** a una freqüència de 100Hz.

Arribats a este punt podem suposar que la classe que defineix la interfície per a qualsevol controlador en temps real que haja d'executar-se sobre el *Controller Manager*

[2]La funció del planificador és repartir el temps disponible d'un microprocessador entre tots els processos que estan a l'espera de ser executats.

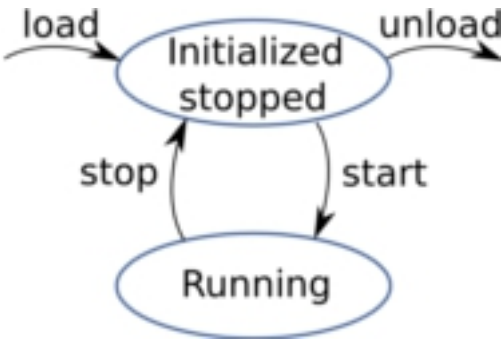

Figura 6.3: Estats d'un controlador

serà la **pr2_controller_interface**. Per implementar esta interfície el nostre controlador haurà d'heretar de la classe base **pr2_controller_interface::Controller** i implementar obligatòriament estos quatre mètodes:

- **init():** És el mètode serà l'encarregat d'inicialitzar el controlador quan el carreguem al CM, tot açò en temps no-real. És important destacar la diferència entre inicialitzar *(init)* i executar (*starting*) un controlador, ja que la inicialització pot realitzar-se prou abans de que el controlador vaja a ser executat. El mètode **init()** pren dos arguments, ***robot*** (que es tracta d'un **pr2_mechanism_model::RobotState**, classe encarregada de descriure el model mecànic del robot, més endavant parlarem d'ell) i ***n*** (un manejador del node del controlador en qüestió del tipus **ros::NodeHandle**)

- **starting():** Este mètode s'executa cada volta que el CM executa un controlador.

- **update():** A este mètode és on es fa la vertadera feina de control i serà cridat periòdicament pel CM a una freqüència de 1000Hz.

- **stop():** S'executarà cada volta que el CM ordene parar un controlador, esta parada es farà efectiva en el mateix cicle que l'ultima cridada a **update()**, immediatament després de que esta haja sigut cridada.

Figura 6.4: Flux d'execució a un controlador.

Com hem vist més amunt, la relació entre el temps real i la **mecànica del robot** es farà a l'hora d'inicialitzar el controlador gràcies a la classe **pr2_mechanism_model** però, que és exactament este paquet i que ens proporciona?

El paquet **pr2_mechanism_model** conté la classe C++ **pr2_mechanism_model ::RobotState** la qual ens proporciona una interfície a les articulacions del robot i una descripció del model d'aquest. El següent gràfic mostra com participa el model mecànic tant en el robot real com en simulació.

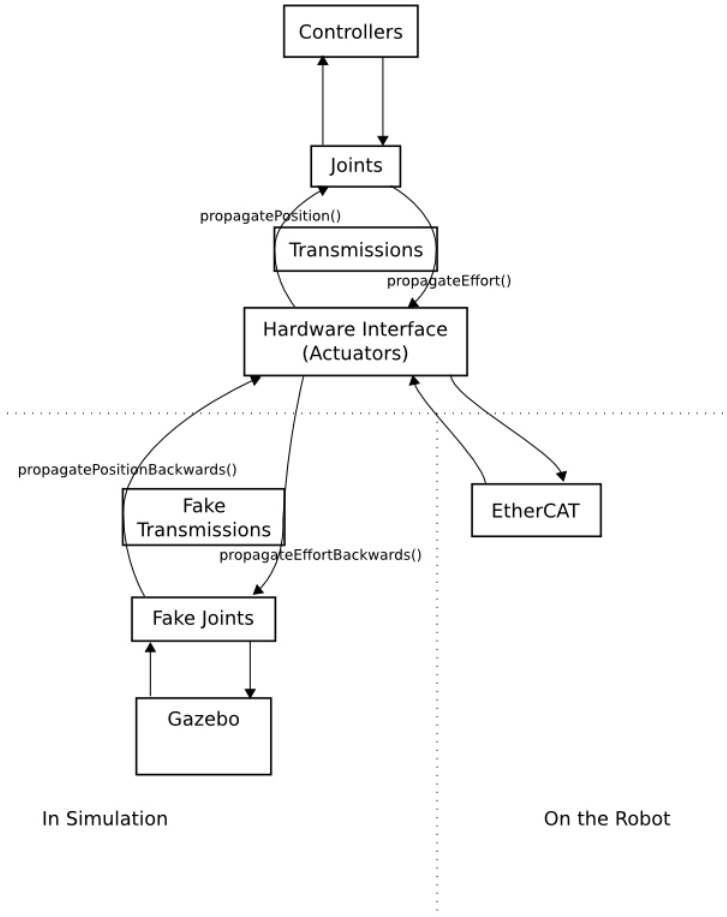

Figura 6.5: Model mecànic del robot i la seua relació tant en simulació com al robot real.

Com podem deduir esta classe és molt extensa i una vegada més anem a remetre a la documentació oficial per a més informació. De totes maneres anem a fer una vista general del que suposa esta classe centrant-nos en el que més anem a utilitzar a l'hora de simular.

- **pr2_mechanism_model::RobotState:** Com ja hem dit abans, quan un controlador s'inicialitza, el CM passa al controlador un punter al **Robot-State**. Este descriu el model cinemàtic i dinàmic del robot en l'estat actual. Aquest estat queda definit per la posició, velocitat i esforç de cadascuna de les articulacions del robot, que obtindrem de l'**URDF** dissenyat prèviament.
- **pr2_mechanism_model::JointState:** Donat que el model mecànic del

robot es basa en l'estat de les articulacions, estes seran imprescindibles a l'hora de controlar-lo. Les articulacions ens proporcionaran les dades necessàries (paràmetres *position* i *velocity*) per realitzar els càlculs de control, que una vegada efectuats, podrem fer que actuen sobre elles senzillament modificant el paràmetre *effort*. Suposarem este model de caixa negra que més endavant serà explicat.

6.3.2 Programant el nostre controlador

Com hem vist programar un controlador en temps real en ROS no és senzill, ja que s'ha de tenir en compte i seguir una estructura predefinida i algunes vegades un poc confusa. No obstant l'esforç paga la pena, ja que una volta programat, és molt fàcil comunicar-se amb ell i amb la garantia de que s'executarà en temps real.

Per dissenyar el nostre controlador ens basarem en el tutorial ***"Writing a realtime joint controller"***[3], disponible a la *wiki* de ROS i al que baix fem referència. En este tutorial se'ns mostra la manera estàndard de programar un controlador en temps real que afegirem al *Controller Manager* com si es tractara d'un *plugin*.

El diagrama que seguirem serà el següent:

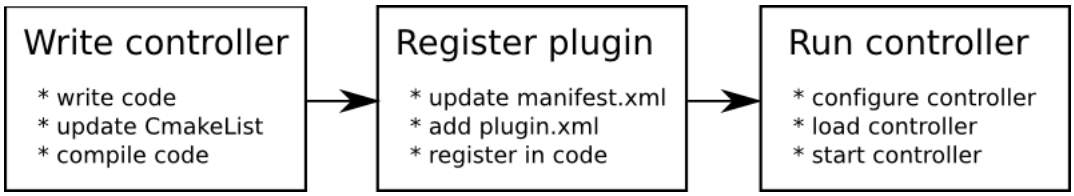

Figura 6.6: Diagrama a seguir en la programació del controlador.

En la primera fase del disseny començarem per crear un fitxer de capçaleres **.h** que continga la interfície del nostre programa: variables, mètodes, i declaracions públiques i privades d'estos, entre altres. Com ja vam dir a l'anterior secció, el controlador haurà de satisfer la **pr2_controller_interface** per a que puga ser llançat per un *Controller Manager*, per això esta classe haurà d'incloure a la seua capçalera esta referència a més d'altres que seran necessàries per la representació del model cinemàtic/dinàmic del robot, el seu control i navegació.

```
#include <pr2_controller_interface/controller.h>
#include <pr2_mechanism_model/joint.h>
#include <geometry_msgs/Twist.h>
#include <tf/transform_broadcaster.h>
#include <nav_msgs/Odometry.h>
```

[3]http://www.ros.org/wiki/pr2_mechanism/Tutorials/Writing%20a%20realtime%20joint%20controller

Per satisfer esta interfície, haurà d'implementar necessàriament els ja coneguts quatre mètodes bàsics d'un controlador: **init(), starting(), update(), stop()**. Declarem ací també tantes variables del tipus **pr2_mechanism_model::JointState** com articulacions anem a controlar, en el nostre cas volem controlar quatre articulacions/motors corresponents a les quatre rodes del vehicle:

```
pr2_mechanism_model::JointState* joint_state_blw_;
pr2_mechanism_model::JointState* joint_state_flw_;
pr2_mechanism_model::JointState* joint_state_brw_;
pr2_mechanism_model::JointState* joint_state_frw_;
```

Tampoc haurem d'oblidar la declaració de variables per emmagatzemar posicions i velocitats inicials, així com els diferents *callbacks* per poder actuar sobre el controlador. Més endavant especificarem com programar este mètode de comunicació.

Una vegada declarada la interfície de la nostra classe podem implementar-la al fitxer **.cpp** corresponent. Anem a descriure que fa cada mètode i comentar les parts més interessants del codi. El codi complet el podem trobar als annexes.

- **init():** En este mètode es relacionaran les diferents articulacions del robot amb les variables del tipus **pr2_mechanism_model::JointState** que havíem creat prèviament a la interfície. A més a més, s'inicialitzaran les variables de posicions linears i angulars, de posició i es crearà un subscriptor al tòpic **guardian_controller/command** que esperarà missatges del tipus **geometry_msgs::Twist** i que quan reba un d'estos executarà el *callback* que descriurem més endavant anomenat *commandCallback*. Com podem suposar pel nom, este *callback* esperarà referències de velocitat, ja siga per teclat o qualsevol altre perifèric, així com d'algorismes de la *Navigation Stack*.

- **starting():** En este mètode obtenim les posicions i velocitats inicials de cada roda/articulació. Gràcies als atributs de les variables **pr2_mechanism_model ::JointState** i al tractar-se d'una simulació, només hem d'accedir a estes variables que s'actualitzen mitjançant el simulador, la qual cosa simplifica de forma considerable l'acció. En cas de voler fer açò en un robot real, hauríem de dissenyar algun mecanisme per a que cada variable d'articulació anara actualitzant els seus atributs interns mitjançant encoders o IMU's.

- **update():** Com ja hem dit abans, ací es realitzarà la vertadera feina de control, és a dir, on s'aplicaran les fórmules de cinemàtica diferencial i de control automàtic proporcional explicades a la secció de fonaments. Primerament, s'obtindran les velocitats (**v_left_mps** i **v_right_mps**) en eixe instant gràcies, com sempre, a l'atribut *velocity* de la classe **pr2_mechanism_model ::JointState** (esta acció es correspondria amb la lectura dels encoders d'un robot real). Com ja veiérem a l'apartat de cinemàtica diferencial, estes dos velocitats juntament a les que es poden obtindre a partir d'elles com son la

velocitat lineal i angular del robot, caracteritzen el comportament del mòbil. Una vegada dit i fet açò, es calcula l'acció de control sobre les articulacions mitjançant un control P, tant per a velocitats lineal com angular. Ací tenim les línies on du a terme este senzill càlcul:

```
double uv= kpv * epv;
double uw= kpw * epw;
```

Una vegada calculada l'acció de control respecte a la velocitat, podem saber gràcies a la cinemàtica inversa quina ha de ser esta per al control del motor.

```
double dUl = uv - 0.5 * GUARDIAN_D_TRACKS_M * uw;
double dUr = uv + 0.5 * GUARDIAN_D_TRACKS_M * uw;
```

Amb totes les accions calculades, només falta enviar estes ordres al motor. A un robot real açò ho faríem mitjançant els actuadors. Com que estem a una simulació resulta molt més còmode gràcies, una vegada més, a l'atribut *commanded_effort_* de les variables **pr2_mechanism_model::JointState**. Ací baix mostrem com s'actua sobre una de les quatre rodes del mòbil.

```
joint_state_blw_->commanded_effort_ = saturation
(-10.0 * (joint_state_blw_->velocity_ - dUl), -limit, limit);
```

- **stop():** Crida al mètode del mateix nom de la classe pare *pr2_controller_interface*.
- **commandCallback(const geometry_msgs::TwistConstPtr& msg):** Este mètode s'executarà cada vegada que es reba un missatge del tipus geometry_msgs::TwistConstPtr pel tòpic guardian_controller/command, és a dir, cada volta que s'envie una ordre per teclat/joystick. L'objectiu d'este serà, agafar del missatge original (punter a missatge del tipus Twist) la informació de les velocitats i passar-li-la a un altre mètode ja en forma de geometry_msgs::Twist.
- **setCommand(const geometry_msgs::Twist &cmd_vel):** És el mètode encarregat d'interpretar el missatge de velocitats aplegat pel *callback* i fixar les velocitats lineals i angulars de referència en conseqüència. Estes velocitats són les que després s'han de tindre en compte en el control de velocitats del mètode **update()**.
- **double saturation(double u, double min, double max):** Mètode auxiliar per saturar els possibles valors d'esforç de cara a actuar sobre les rodes. Podem donar referències tant altes/baixes que els valors que obtindríem no podrien

ser interpretats correctament pel motor. Per a evitar això, saturem per baix i per dalt estes ordres, per a donat el cas que aplegue un valor estrany el convertisca en un que estiga dins del límit estipulat per *min* i *max*.

Una vegada escrit el codi del nostre controlador haurem de compilar-lo en forma de biblioteca (i no d'executable com és habitual). Una volta fet açò haurem de **registrar-lo** com si d'un plugin es tractara per fer que siga possible la seua execució en temps real. Açò ho podem fer de dos formes: automàticament amb un fitxer de configuració que busque el controlador cada volta que el procés en temps real s'inicie o enllaçant manualment el controlador quan el procés ja estiga executant-se. Siga com siga, primerament hem de declar el plugin i açò es fa amb l'addició d'una línia al final del codi font del controlador i posterior recompilació. En el nostre cas serà la següent:

```
PLUGINLIB_DECLARE_CLASS(guardian_controller, GuardianControllerPlugin,
guardian_controller_ns::GuardianControllerClass,
pr2_controller_interface::Controller)
```

A continuació, modificarem l'arxiu **manifest.xml** del paquet on ens trobem per afegir, en cas de no haver estat encara indicades, les dependències a **pluginlib** (l'encarregat de gestionar els plugins) i **pr2_controller_interface**, a més a més, haurem de declarar explícitament amb la sentència export que estem publicant un plugin.

```
<export>
<pr2_controller_interface plugin="${prefix}/controller_plugins.xml"/>
</export>
```

Per últim, crearem un fitxer de definició del plugin que anomenarem **controller_plugins.xml**. Tindrà la següent forma:

```
<library path="lib/libguardian_controller_lib">
<class name="guardian_controller/GuardianControllerPlugin"
type="guardian_controller_ns::GuardianControllerClass"
base_class_type="pr2_controller_interface::Controller" />
</library>
```

Fins ací hem escrit el codi del controlador i l'hem publicat per a que siga possible llançar-lo en l'entorn de temps real de ROS, però de res ens serveix el controlador si no el **llancem**, cosa que, com de costum, podem fer de diverses formes. Anem a explicar de forma detallada com fer-ho de forma automàtica, ja que al cap i a la fi és el que utilitzarem sempre. En cas de voler fer-ho de forma manual, recomanem mirar esta web[4].

[4]http://www.ros.org/wiki/pr2_mechanism/Tutorials/Running%20a%20realtime%20joint%20controller

Per **llançar un controlador** de forma fàcil i quasi automàtica l'únic que hem de fer és crear un arxiu de configuració **.yaml** on és relacionen articulacions de l'**URDF** amb variables del controlador. Este fitxer te la següent forma al nostre controlador, com vegem, apareix el nom d'este, el tipus i la relació entre els diferents elements:

```
guardian_controller:
type: guardian_controller/GuardianControllerPlugin
joint_blw: joint_back_left_wheel
joint_flw: joint_front_left_wheel
joint_brw: joint_back_right_wheel
joint_frw: joint_front_right_wheel
```

Ara només hem d'afegir-lo al fitxer **.launch** per a que es llance juntament amb l'**URDF** i el Controller Manager, amb la qual cosa, l'aspecte que tindria l'arxiu encarregat de llançar, ara per ara, la simulació del robot Guardian amb controlador i sobre un món buit a Gazebo seria la següent:

```
<launch>
  <include file="$(find gazebo_worlds)/launch/empty_world.launch"/>
  <include file="$(find pr2_controller_manager)/controller_manager.launch"/>
  <param name="robot_description" textfile="$(find guardian_description)/urdf/guardian.urdf" />
  <node name="spawn_object" pkg="gazebo" type="spawn_model" args="-urdf -param robot_description -x
1.0 -y 0.75 -z 0.5 -model guardian -unpause" respawn="false" output="screen" />
  <rosparam file="$(find guardian_controller)/guardian_controller.yaml" command="load"/>
  <param name="pr2_controller_manager/joint_state_publish_rate" value="100.0" />
  <node name="guardian_controller" pkg="pr2_controller_manager" type="spawner"
args="guardian_controller" respawn="false" output="screen"/>
</launch>
```

No obstant tot l'anterior, encara no hem acabat. Cal explicar com funciona la caixa negra que hem suposat abans per a que les variables tipus **pr2_mechanism_model::JointState** continguen en cada instant els atributs actuals de velocitat, esforç i posició de cada articulació. Com estem parlant en tot moment d'una simulació, anem a obviar el model d'actuadors/encoders com ja hem nomenat més amunt per accedir directament a les variables de les articulacions per llegir i modificar el seu estat. Però, com es relaciona l'**URDF** amb el controlador i este amb el *Controller Manager*? L'**URDF** amb el controlador ho acabem de vore mitjançant fitxers **.yaml** de configuració però que hi ha del *Controller Manager*?

Per donar resposta a esta pregunta haurem de fer servir un plugin de Gazebo, més concretament el **gazebo_ros_controller_manager**. Este plugin proporciona una interfície entre el robot simulat i el **pr2_controller_manager**. Gazebo, per la seua part, proporciona un control mitjançant força/par dels enllaços (*links*) i articulacions (*joints*) simulades. Este plugin exposa un conjunt d'estats de pseudoactuadors per al **pr2_controller_manager** per ROS mitjançant l'ús de les transmissions inverses, conforme està definit en **pr2_mechanism_controllers**.

6.4 Teleoperació

Podem definir la teleoperació com el conjunt de tecnologies que comprenen la operació o govern a distància d'un dispositiu per un ésser humà. Per tant, un sistema de teleoperació serà aquell que permeta controlar a distància un dispositiu, en el nostre cas, el robot mòbil Guardian que controlarem virtualment mitjançant el teclat o el joystick segons conviga. A esta secció explicarem com foren dissenyades estes dos formes de controlar el robot.

6.4.1 Teclat

Dissenyar la teleoperació via teclat no és una cosa complexa ja que amb un simple arxiu **.launch** podem tenir-la disponible. Com bé recordarem de la secció anterior, el controlador disposava d'un subscriptor que esperava missatges del tipus **geometry_msgs::Twist** pel tòpic **guardian_controller/command**. La pregunta que segurament ens hem fet quan hem explicat l'existència d'este subscriptor és, qui és l'encarregat de publicar estos missatges? Com podem suposar l'encarregat d'açò serà el teclat (o el joystick, encara que de forma diferent) però, com podem fer per a que les nostres pulsacions de teclat generen un missatge estandarditzat del tipus **Twist**?

El que farem per a açò és aprofitar-nos d'un dels avantatges que ens ofereix ROS, l'alta reutilització del codi, així que simplement podem utilitzar la teleoperació mitjançant teclat del robot **PR2** i adaptar els paràmetres per a un control per teclat del nostre robot. El fitxer **.launch** quedaria com segueix, i com podem vore, simplement creem un node del tipus **teleop_pr2_keyboard** i canviem certs paràmetres com el tòpic d'eixida i les velocitats:

```
<launch>
  <node pkg="pr2_teleop" type="teleop_pr2_keyboard" name="spawn_teleop_keyboard" output="screen">
      <remap from="cmd_vel" to="guardian_controller/command" />
      <param name="walk_vel" value="0.5" />
      <param name="run_vel" value="1.5" />
      <param name="yaw_rate" value="0.75" />
      <param name="yaw_run_rate" value="1.5" />
  </node>
</launch>
```

6.4.2 Joystick

Pel que fa al disseny de la teleoperació mitjançant joystick no és tant senzilla com la del teclat. En este cas haurem de disposar de dos nodes, un primer node encarregat de llegir del joystick i transmetre els missatges de tipus **sensor_msgs::Joy** a un

segon node que els interprete conforme el control que volem dissenyar i els transforme en referències de velocitat, és a dir en **geometry_msgs::Twist**.

El primer node, anomenat **joy**, el podem trobar al paquet **joystick_drivers** i per tant només haurem d'encarregar-nos de llançar-lo. Este node farà d'interfície entre qualsevol joystick genèric que estiga suportat per Linux i ROS. Més concretament, este node publicarà un missatge del tipus **sensor_msgs::Joy** que no és més que un missatge on es contindrà l'estat actual de cadascun dels botons i eixos del dispositiu. No obstant, amb açò no podem actuar sobre el controlador que hem programat anteriorment, ja que com bé sabem, este espera referències a velocitats.

Donat que amb el node anterior només no podrem realitzar la teleoperació amb èxit, hem de dissenyar un altre que s'encarregue de transformar l'estat dels eixos i els botons en referències de velocitat. Per a fer açò, crearem un nou paquet amb dependències a **sensor_msgs** (ja que **Joy** serà el tipus d'entrada del node), **geometry_msgs** (ja que **Twist** serà el tipus d'eixida del node) i **roscpp** (llibreria amb la que utilitzem ROS sobre C++). Una vegada fet açò ens centrarem en la programació del codi. El nostre nou node bàsicament serà un **main()** molt simple que inicie el node i cree un objecte del tipus *GuardianJoy*, creat expresament. Este objecte s'executarà de manera cíclica i fixarà els diferents paràmetres del joystick, com la correspondència dels eixos amb les diferents velocitats o el que és més important, el tòpic al qual haurà de publicar la informació i el tòpic sobre el que rebrà les ordres del joystick. Este últim, cada volta que reba informació del tipus esperat (**sensor_msgs::Joy**) s'encarregarà d'executar el *callback* corresponent, que serà qui realment faça la feina a este node. A este *callback* detectarem els diferents moviments possibles del robot (en el cas del Guardian avant/arrere, gir a la esquerra/dreta o rotació sobre si mateix a l'esquerra/dreta) i generarem les referències de velocitats adequades per a cadascun d'ells. Abans d'acabar l'execució d'este callback, s'enviarà pel tòpic d'eixida (en el nostre cas **guardian_controller/command**).

Per últim, escriurem un senzill **.launch** que s'encarregue de llançar els dos nodes i configurar els paràmetres d'estos per adaptar-lo el millor possible al joystick i al robot. Adjuntem una imatge del flux d'informació entre nodes per clarificar el que acabem de comentar.

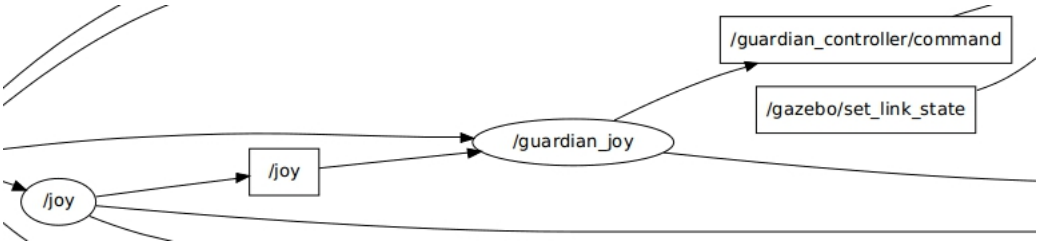

Figura 6.7: Gràfic de la teleoperació per joystick.

6.5 Afegint altres dispositius al nostre model URDF

Fins ara, el nostre model **URDF** només contenia quatre tipus diferents d'elements: els ***joints***, els ***links***, les etiquetes referides al simulador Gazebo i el ***plugin*** per integrar el *Controller Manager* en la simulació. Tot i que amb estos elements ens és suficient per a simular qualsevol robot mecànica i físicament, no podem incorporar sensors i càmeres si no contem amb els plugins corresponents. A esta secció explicarem com afegir detalladament un làser *Hokuyo URG-04LX*, una **IMU** i una càmera per visualitzar per on avança el robot.

6.5.1 Làser

Primerament, crearem un link/cos que serà el punt del robot des d'on es capturaran les dades. Idealment seria un cub el més xicotet possible i situat allà on volem que el làser siga visible. Per relacionar este nou **link** amb el bastidor del robot crearem un nou **joint** que ho faça, sempre tenient el compte el mantenir l'estructura d'arbre que propugna el model **URDF**. Una vegada fet açò el làser ja podria capturar les dades des del punt anterior però si volem veure'l en la simulació haurem d'afegir els elements visuals necessaris per a fer-ho. Podríem haver posat simplement un cub, però hem agafat una malla (en format **.stl** com el bastidor del robot) d'un làser **Hokuyo** que hi ha als repositoris de ROS, així es mostrarà un làser a la simulació i no un simple punt des d'on es capturen les dades.

Per últim, l'apartat més important, ací és on realment li donem al link/cos creat al principi les propietats d'un làser. Açò ho fem incorporant este fragment de codi a l'**URDF**:

```
<gazebo reference = "hokuyo_laser_link">
  <sensor:ray name="hokuyo_laser">
    <rayCount>640</rayCount>
    <!-- Consulteu el codi font per vore tots els paràmetres  -->
    <updateRate>10.0</updateRate>
    <controller:gazebo_ros_laser name="gazebo_ros_hokuyo_laser_controller"
plugin="libgazebo_ros_laser.so">
        <gaussianNoise>0.005</gaussianNoise>
        <alwaysOn>true</alwaysOn>
        <updateRate>10.0</updateRate>
        <topicName>hokuyo_laser_topic</topicName>
        <frameName>hokuyo_laser_link</frameName>
        <interface:laser name="gazebo_ros_hokuyo_laser_iface" />
    </controller:gazebo_ros_laser>
  </sensor:ray>
</gazebo>
```

Com vegem es tracta novament d'un plugin per a Gazebo que carrega un controlador encarregat de recopilar les dades que van des d'un làser simulat i publicar-les en forma de **sensor_msgs::LaserScan** al tòpic que indiquem a l'etiqueta corresponent. A més del tòpic al qual es publica la informació, hem d'adaptar els diferents valors (*minRange, maxRange, updateRange, minAngle i maxAngle*) als del nostre sensor real/desitjat.

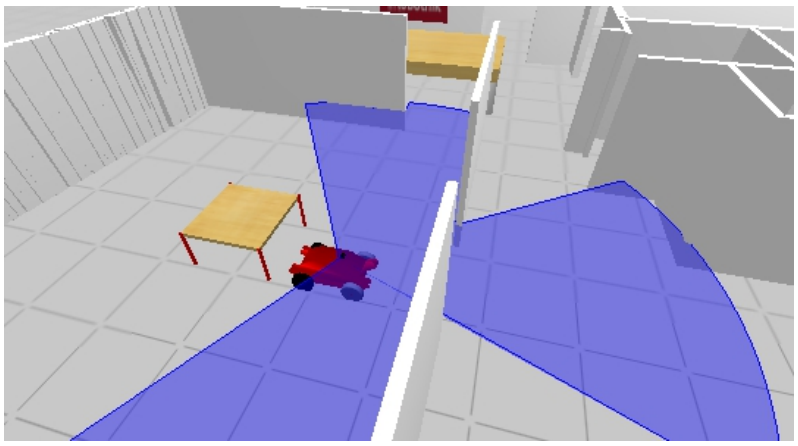

Figura 6.8: Laser ja instal · lat sobre el robot. Si ens fixem, és el punt negre des del qual ixen les línies blaves que representen gràficament el seu abast.

6.5.2 IMU

Una **IMU**, sigles angleses d'*Inertial Measurement Unit* (unitat inercial de mesura) és un dispositiu electrònic que mesura velocitats, orientacions i forces gravitacionals fent uns d'una combinació d'acceleròmetres i giroscopis. En un sistema de navegació, les dades obtingudes per la IMU es transmeten a l'ordinador, qui calcula la posició actual basant-se en els diferents valors obtinguts (velocitats, temps, orientacions...). És un sistema molt fiable i per això ens serà molt útil de cara a l'aplicació dels algorismes de navegació al nostre robot simulat.

Per a afegir una **IMU** al nostre robot procedirem igual que al cas anterior, amb la diferència de que ara no ens caldrà polir l'aspecte visual ja que estos dispositius solen situar-se normalment al centre i a l'interior dels robots. El plugin que utilitzarem en esta situació i que col · locarem a l'**URDF** és el **gazebo_ros_imu** i s'encarregarà de publicar un missatge del tipus **sensor_msgs::Imu** al tòpic que especifiquem.

```
<gazebo>
 <controller:gazebo_ros_imu name="imu_controller" plugin="libgazebo_ros_imu.so">
      <alwaysOn>true</alwaysOn>
      <updateRate>50.0</updateRate>
      <bodyName>imu_link</bodyName>
      <topicName>imu_data</topicName>
      <gaussianNoise>2.89e-08</gaussianNoise>
      <xyzOffsets>0 0 0</xyzOffsets>
      <rpyOffsets>0 0 0</rpyOffsets>
      <interface:position name="imu_position"/>
 </controller:gazebo_ros_imu>
</gazebo>
```

6.5.3 Càmera

També podem afegir càmeres al nostre robot simulat. Des d'estes podrem tindre una perspectiva subjectiva d'abord del robot o configurar-lo per a tasques de vigilància remota, entre altres.

Com ja suposarem a estes altures, només haurem d'afegir el *plugin* del controlador necessari per disposar de la funcionalitat de la càmera que en este cas serà **gazebo_ros_camera**. Posem a continuació una captura de la simulació on puguem observar el robot i la imatge obtinguda desde la càmera. Per a obtenir este punt de vista només hem tingut que subscriure-nos al tòpic on hem definit que es publique la informació (**/image_raw**) mitjançant esta ordre: **rosrun image_view image_view image:=/image_raw**.

Figura 6.9: Imatge obtinguda des de la càmera incorporada al robot.

6.5.4 Altres plugins

Creguem interessant afegir una taula amb tots els plugins dels que disposa Gazebo. Com hem vist, la instal · lació no dista molt d'un model a un altre i les utilitats que es poden obtenir són molt variades.

Nom	Descripció
`GazeboRosTime`	Publica el temps en simulació com a temps global pel tòpic `/clock`.
`GazeboRosTemplate`	Proporciona una interfície C++ per a fer els nostres propis plugins.
`GazeboRosFactory`	Proporciona una interfície de manipulació de models sobre serveis.
`GazeboRosCamera`	Explicat més amunt.
`GazeboRosP3D`	Obté el la posició inercial de qualsevol cos amb el que estiga relacionat.
`GazeboRosLaser`	Explicat més amunt.
`GazeboRosBumper`	Ofereix la utilitat d'un *bumper* via missatges del tipus `ContactsState`.
`GazeboRosIMU`	Explicat més amunt.
`GazeboRosSimIface`	Proporciona una interfície per poder posicionar qualsevol cos en simulació.
`GazeboRosForce`	Proropociona una interfície per poder aplicar una `Wrench` sobre qualsevol cos simulat.
`GazeboRosBlockLaser`	Proporciona un tipus d'escànner làser diferent al comentat.
`GazeboRosF3D`	Obté les forces externs sobre qualsevol cos en un missatge del tipus `WrenchStamped`.
`GazeboRosDiffdrive`	Proporciona un control diferencial genèric basat en el del robot Erratic.

Figura 6.10: Plugins dinàmics disponibles per al simulador Gazebo.

6.6 Integració amb la Navigation Stack

La ***Navigation Stack*** és molt simple a nivell conceptual. Es tracta d'una pila que proporciona utilitats de navegació autònoma mitjançant diferents algorismes només prenent informació de l'odometria i dels sensors i actuant sobre la base mòbil del robot enviant referències de velocitat. A més, tota la navegació es farà de forma quasi transparent a nosaltres. No obstant, no és trivial fer funcionar la *Navigation* en qualsevol robot, sinó que l'haurem d'adaptar a l'estàndard de ROS i per a això se'ns exigirà el següent:

- Que el robot funcione sobre ROS.
- Que es dispose d'un arbre de **tf** (transformacions) consistent.
- Que es publique la informació dels sensors utilitzant un format de missatges de ROS.

A hores d'ara només disposem d'un *base_controller* (***guardian_controller***) que rep referències de velocitat per via teclat/joystick, a més d'un làser incorporat a la simulació que publica missatges del tipus **sensor_msgs::LaserScan** al tòpic **hokuyo_laser_topic** i una IMU simulada que fa el que li és propi amb missatges del tipus **sensor_msgs::Imu** i el tòpic **imu_data**. No obstant, encara ens falta per desenvolupar la odometria i les transformacions estàtiques del robot (distàncies entre els diferents elements, com per exemple entre la base i el làser). A partir d'ací dissenyarem estos dos nodes per a continuació descriure detalladament les configuracions dels dos algorismes disponibles en esta pila.

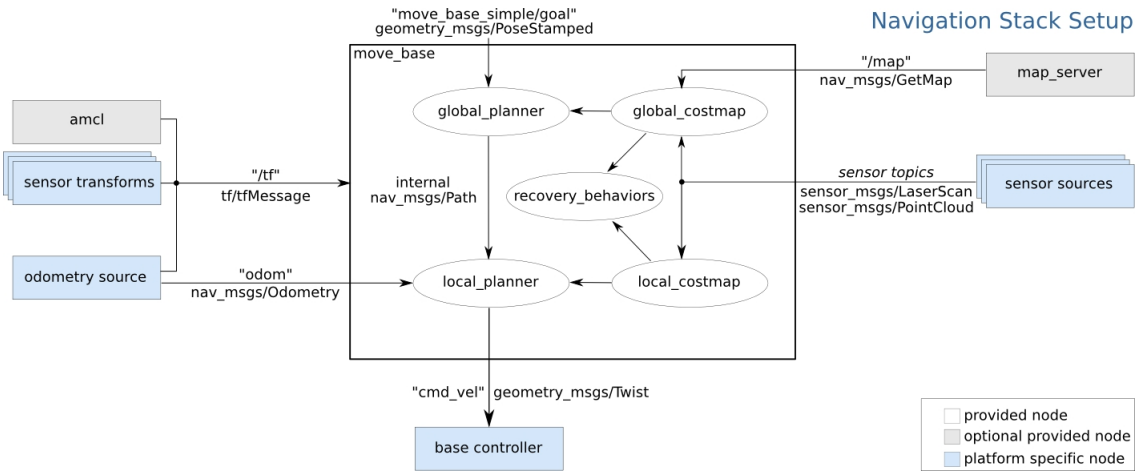

Figura 6.11: Esquema de la configuració de la Navigation Stack.

6.6.1 Odometria

Una bona odometria és fonamental en la navegació autònoma. Per fer-la possible dissenyarem un nou paquet anomenat *guardian_odometry*. Tot i que la funcionalitat del càlcul de l'odometria podia haver-se incorporat al mètode **update()** del controlador, creguem que d'esta forma es respecta més la filosofia modular de ROS així com es fomenta la claredat del codi.

Este node tindrà accés al que hem definit com "encoders" del simulador, les variables del tipus **pr2_mechanism_model::JointState**, així que haurem d'especificar-ho a les dependències, com també ho farem amb els missatges de l'**IMU**, entre altres. Pel que fa a l'eixida, haurà de publicar transformacions i missatges d'odometria, així que incorporarem les dependències corresponents a la capçalera.

Una vegada clara l'entrada/eixida del node, podem definir la seua funcionalitat. Este, esperarà missatges **JointState** i d'**Imu** mitjançant dos *callbacks* declarats només s'inicia el node. Arribats a este punt hem d'aclarir una cosa important. Si no ens hem perdut en l'explicació, sabrem que els missatges del tipus **JointState** ofereixen la velocitat, esforç i posició de cada articulació d'una forma precisa. Tenint estos valors i sabent que la posició del nostre mòbil queda definida pel centre de les quatre rodes podríem directament obtenir una posició quasi perfecta i publicar-la en forma de missatge d'odometria sense fer cap càlcul cinemàtic. No obstant, em volgut cenyir-nos el màxim possible a la realitat, i tot i que podríem fer el que hem comentat i tenir una odometria més precisa, preferim només obtindre la velocitat de les articulacions, com si de la informació obtinguda d'un encoder es tractara per, amb açò i les fórmules adequades obtenir la posició aproximada del robot. Una vegada aclarit açò continuem amb el disseny del node. Este tindrà per eixides uns publicadors de missatges d'odometria i un *broadcaster* per als missatges de les transformacions.

```
ros::Subscriber joint_subs = n.subscribe("/joint_states", 1, jsCb);
ros::Subscriber imu_subs = n.subscribe("/imu_data", 1, imuCb);
ros::Publisher odom_pub = n.advertise<nav_msgs::Odom>("/odom", 100);
tf::TransformBroadcaster odom_broadcaster;
```

La resta del codi d'inicialització s'encarrega de les variables de posició (*px, py, pa*) així com algunes variables auxiliars per a la gestió del temps o físiques com la separació entre rodes i el diàmetres d'estes. Pel que fa al bucle principal, és on es farà el càlcul de l'odometria i s'executarà cíclicament cada volta que es reba un missatge. A este node es calcularan les velocitats de les rodes esquerres i dretes del mòbil, tal i com descriu el model cinemàtic diferencial del robot i a partir d'estes s'obtindrà la **velocitat lineal**. Pel que fa a la **velocitat angular**, donat que es tracta d'una simulació i resulta quasi impossible ajustar els paràmetres de fricció en el terra com al món real, obtindrem esta velocitat directament gràcies a la **IMU**. Una vegada tinguem descrit el robot per estes dos velocitats, podem obtenir fàcilment la posició aplicant una senzilla operació del moviment diferencial.

```
// Compute Velocity
vx = linearSpeedMps_ * cos(robot_pose_pa_);
vy = linearSpeedMps_ * sin(robot_pose_pa_);
// Compute Position
robot_pose_pa_ += ang_vel_z_ * fSamplePeriod;
robot_pose_px_ += vx * fSamplePeriod;
robot_pose_py_ += vy * fSamplePeriod;
```

Per últim, amb estes dades de la posició en l'instant actual calculades, creem els missatges de tf i odometria (completant tots els camps, des dels referents a velocitats i posicions fins als referents a temps i informació sobre qui és el marc que ho publica) i els enviem pel respectiu publicador/*broadcaster*. Amb açò tindrem disponible un node que publique missatges del tipus **nav_msgs::Odometry** i **geometry_msgs::TransformStamped**.

6.6.2 Transformacions estàtiques

Com ja hem explicat en l'apartat corresponent als conceptes d'alt nivell de ROS, les transformacions permeten establir les relacions entre els diferents marcs coordinats mitjançant una estructura d'arbre i emmagatzemant-les en el temps, açò permet a l'usuari transformar qualsevol punt, vector o informació d'un marc coordinat a un altre, inclús en qualsevol moment passat desitjat.

Este, com l'anterior, és un node que haguérem pogut "evitar", ja que existeix un node a les llibreries de ROS que s'encarrega d'açò amb un simple pas de paràme-

tres, el **static_transform_publisher**, però que per clarificar conceptes i arribar a l'interior del llenguatge hem preferit programar nosaltres de zero.

L'únic que farà este node és, iniciar-se i limitar-se a publicar amb el rati que especifiquem les transformacions estàtiques del robot, és a dir, un bucle infinit mentre el node estiga en execució. En el nostre cas, i conforme s'estableix a les **REP**[5]'s, ens faran falta dos transformacions, la que va del *chassis_footprint* al *chassis* (anomenat bastidor, fins ara) i la que va d'este últim al *làser*. La primera transformació és un més dels requeriments de l'adaptació a la *Navigation Stack* i només serà la distància entre els dos elements (que en efecte només estaran separats una distància en l'eix z). Per altra banda, la segona transformació serveix per una vegada obtingudes les dades del làser aplicar una "correcció" per a que el controlador de la base puga fer-ne ús, cosa que em explicat de manera detallada a l'apartat teòric de les transformacions. A continuació el codi amb que implementem l'explicat:

```
broadcaster.sendTransform(tf::StampedTransform(tf::Transform(tf::Quaternion(0,0,0,1),
tf::Vector3(0.0,0.0,0.130)),ros::Time::now(),"base_footprint", "base_link"));

broadcaster.sendTransform(tf::StampedTransform(tf::Transform(tf::Quaternion(0,0,0,1),
tf::Vector3(0.12,0.0,0.285)),ros::Time::now(),"base_link", "hokuyo_laser_link"));
```

6.7 Configurant la Navigation Stack

Una vegada en este punt anem a fer un resum de tot els nodes i elements que tenim en relació a la navegació. Disposem d'un node controlador (***guardian_controller***) que rep missatges amb referències de velocitat. També tenim un làser simulat que publica la informació d'allò que escaneja en la simulació de forma estandarditzada al tòpic **hokuyo_laser_topic** així com una **IMU** també simulada i que envia informació al tòpic **imu_data**. A la secció anterior acabem de dissenyar dos nodes, un que publicarà la odometria (***guardian_odometry***) i la transformació relacionada amb esta i un altre que publicarà les transformacions estàtiques (***guardian_tf***). Si ens fixem en la **Figura 6.11** de més amunt, vorem com ja estem preparats per configurar la navegació autònoma ja que el paquet ***move_base*** té tots els requeriments acomplits. Però, que és exactament este paquet?

6.7.1 El paquet move_base

El node ***move_base*** proporciona la interfície ROS per configurar, executar i interactuar amb la *Navigation Stack* d'un robot, podríem dir que es tracta del component principal d'esta. D'una forma més tècnica, ***move_base*** proporciona la

[5]ROS Enhancement Proposals. Són uns documents on s'estableix d'una manera completa l'estil que hauria de tenir la programació en ROS

implementació d'una acció (més informació al paquet ***actionlib***) que donat un objectiu en el món, intentarà assolir-lo movent la base mòbil del robot. Este node, conté internament dos planificadors per a fer possible les tasques de navegació, el global i el local, que estudiarem més endavant.

A continuació anem a descriure els diferents elements que composen este paquet i de quina forma han estat implementats.

Recovery Behaviours (Comportaments de recuperacio)

Quan executem el node ***move_base*** el resultat serà que este intente assolir la posició objectiu amb la base conforme a la tolerància que haja especificat l'usuari prèviament. Este node pot de manera opcional realitzar un comportament de recuperació quan el robot es perceba encallat. En eixe cas, ***move_base*** efectuarà per defecte el següent algoritme. En primer lloc, els obstacles fora de la regió especificada per l'usuari s'esborraran del mapa intern del robot. Després, si es possible, el robot efectuara una rotació sobre si mateix per aclarir l'espai que l'envolta. Si açò també falla, el robot efectuara un aclariment més agressiu del mapa, eliminant tots els obstacles fora de la regió rectangular sobre la qual pot rotar sobre si mateix. Açò vindrà seguit per una altra pivotació sobre si mateix. Si tot açò falla, el robot considerarà el seu objectiu com inassolible i notificarà a l'usuari que ha avortat la navegació. Estos comportaments de recuperació poden ser configurats, així com desactivats mitjançant els paràmetres **recovery_behaviors** i **recovery_behavior_enabled**.

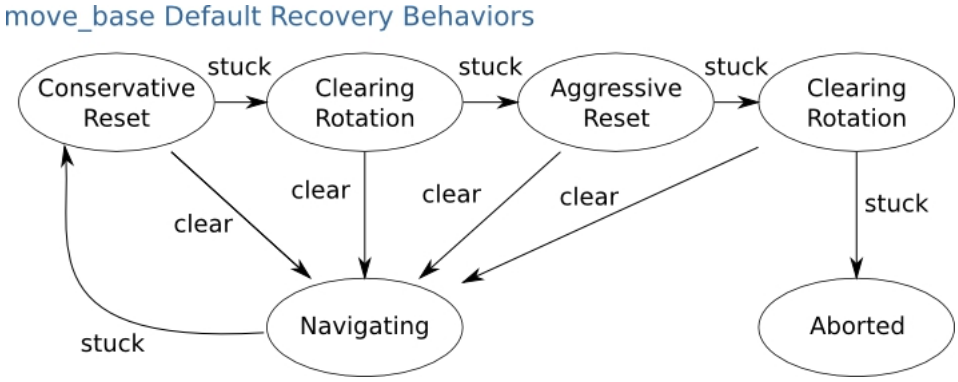

Figura 6.12: Comportaments de recuperació per defecte de ***move_base***.

Una vegada estudiat quin és el comportament que s'espera d'este important node anem a estudiar els diversos elements que el composen:

Costmaps (mapes de costos)

La *Navigation Stack* fa ús de dos mapes de costos (el **local** i el **global**) per emmagatzemar la informació sobre els obstacles. El *global_costmap* s'utilitza per a la

planificació global i crea planificacions a llarg terme sobre l'entorn complet mentre que el *local_costmap* serà utilitzat per a evitar obstacles. Hi ha algunes opcions de configuració que seria convenient que els dos *costmaps* compartiren i d'altres que només ens interessa que estiguen establertes a cada mapa individual i per tant, la configuració dels *costmaps* es farà en tres fitxers diferent on s'emmagatzemaran en cadascun les opcions de configuració **comuns**, les **globals** i les **locals**. A la **Figura 6.13** mostrem els diferents fitxers de configuració al nostre robot i remitim una vegada més a la *wiki*[6] corresponent per a una explicació completa dels diferents paràmetres.

Comuns:
```
obstacle_range: 2.5
raytrace_range: 3.0
footprint: [[0.5, -0.275], [0.5, 0.275], [-0.5,0.275], [-0.5, -0.275]]
inflation_radius: 0.55

observation_sources: hokuyo_laser

hokuyo_laser: {sensor_frame: hokuyo_laser_link, data_type: LaserScan, topic: hokuyo_laser_topic,
marking: true, clearing: true}
```

Globals:
```
global_costmap:
  global_frame: /map
  robot_base_frame: /base_footprint
  update_frequency: 5.0
  static_map: true
```

Locals:
```
local_costmap:
  global_frame: /odom
  robot_base_frame: /base_footprint
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 6.0
  height: 6.0
  resolution: 0.05
```

Figura 6.13: Els tres tipus de ***costmaps***.

Local Planner (planificador local)

El planificador local és el responsable de generar les comandes de velocitat que seran enviades a la base mòbil del robot i que seguiran una planificació local (*local_plan*). El *local_planner* té en compte tant la cinemàtica com la dinàmica del robot i utilitzarà la informació dels sensors per dur a terme una navegació segura.

[6] http://www.ros.org/wiki/costmap_2d

Global_Planner

El planificador global (*global_planner*) és el responsable de generar una planificació d'alt nivell per a la *Navigation Stack*. Donat un objectiu situat lluny del robot, este planificador crearà una sèrie de "*waypoints*" que el planificador local haurà de seguir. La implementació actual de la *Navigation Stack* utilitza un planificador global basat en una malla que assumeix que el robot té forma circular, tot i que açò ultim es pot modificar. Això significa que el planificador global produirà punts que encaixen i puguen ser seguits pel *footprint* (projecció del robot al plànol base, per exemple en el cas del robot **Guardian** serà un rectangle mentre que en el cas d'un robot **Roomba** un cercle) d'este definit prèviament, però també podrà generar-ne d'altres que siguen inassolibles per a este i ací és on entrarà en joc el planificador local. També cal indicar, que el planificador global no té en compte la dinàmica del robot, així que també es poden general plans inassolibles dinàmicament parlant.

6.7.2 Tipus de navegació

Una vegada arribats a este punt ja tenim la *Navigation Stack* quasi configurada. No obstant, encara falta seleccionar la forma que tindrà el robot de posicionar-se/localitzar-se en el mapa. Podem dir que hi ha dos formes de fer açò, el posicionament local (basat únicament en la odometria) i el posicionament mitjançant mapes (ja siguen preestablerts o generats simultàniament). A continuació descriurem com configurar la *Navigaton Stack* per a cada tipus de navegació.

Local

La navegació local està implementada "per defecte" per la *Navigaton Stack* ja que esta només utilitza la odometria del robot per localitzar-se. En esta navegació no hi ha cap tipus de mapa i per tant no es generarà cap ruta global. El robot farà servir els sensors per, conforme vaja aproximant-se al seu objectiu evitar els possibles obstacles.

AMCL

Si ens fixem en la **Figura 6.11** vegem que hi ha dos nodes que apareixen de color gris i que segons l'esquema son opcionals (***map_server*** i ***amcl***), açò és perquè com vorem, la navegació mitjançant localització AMCL proporciona una millor experiència que la basada únicament amb l'odometria, però no és imprescindible per a utilitzar la pila com ja hem vist en la secció anterior. L'explicació teòrica de com

funciona l'AMCL ja va ser comentada en el seu moment a la secció 3.3. A esta secció vorem com està implementat este sistema en ROS.

Primerament i com es tracta d'un algorisme de localització respecte a un mapa prefixat, necessitarem un servidor de mapes que ens oferisca esta funcionalitat i este serà ***map_server***. Ací podrem pujar els nostres mapes i configurar diferents paràmetres (resolució, origen, *tresh...*).

En segon lloc serà necessari un node que oferisca la funcionalitat de l'algorisme en questió, este serà amcl. Este node està fortament parametritzat en tres seccions (filtre general, model de làser, i model d'odometria) per poder adaptar-lo correctament al nostre robot i la seua odometria.

Pel que fa a la forma de cridar a aquest algorisme, hi haurà dos arxius, un anomenat **move_base_amcl** que s'encarregarà d'iniciar el ***map_server*** amb el mapa corresponent, llançar el filtre de particules EKF (si és adient, açò serà necessàri en el cas de que la tf entre el mapa i el frame de l'odometria no siga publicada pel node ***guardian_odometry***), executar el ***move_base*** i cridar a l'execució del node ***amcl*** amb els paràmetres que configurarem a l'arxiu **amcl_guardian.launch** del que tot seguit mostrem l'estructura.

```
<launch>
 <node pkg="amcl" type="amcl" name="amcl" output="screen">

  <remap from="scan" to="hokuyo_laser_topic" />
  <remap from="cmd_vel" to="guardian_controller/command" />

  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha5" value="0.1"/>
  <param name="transform_tolerance" value="0.2" />
  <param name="gui_publish_rate" value="10.0"/>
  <param name="laser_max_beams" value="30"/>
  <param name="min_particles" value="500"/>
  <param name="max_particles" value="5000"/>

  <param name="odom_frame_id" value="odom"/>
  <param name="base_frame_id" value="base_footprint"/>
  <param name="global_frame_id" value="map"/>
  <param name="initial_pose_x" value ="5.2"/>
  <param name="initial_pose_y" value ="3.129"/>
  <param name="initial_pose_a" value ="0.0"/>
 </node>
</launch>
```

SLAM

Pel que fa a l'algorisme **SLAM** no necessitarem el ***map_server***, ja que com sabem este algorisme no utilitza un mapa prefixat sinó que va creant-lo simultàniament amb la navegació per estar localitzat en respecte al mapa en tot moment. Així doncs, en este tipus de localització i mapeig necessitarem únicament del node que proporcione esta funcionalitat (***slam_gmapping***) i del ***move_base*** que s'encarregarà de la navegació pròpiament dita. Disposarem d'un fitxer **move_base_slam.launch** que llançarà els nodes comentats.

Este node a grans trets fa ús de les **tf** (**hokuyo_laser_link** → **base_link** i **base_link** → **odom**) i de les dades obtingudes del làser per a obtenir la transformació corresponent entre el mapa que es va generant i el robot, que serà en efecte la localització d'este. A l'igual que el node **amcl**, el **slam_gmapping** disposa de gran quantitats de paràmetres configurables en respecte al làser i la generació del mapa, novament recomanem visitar la wiki[7] corresponent per a una millor compressió.

6.8 Creació de móns personalitzats

Un dels avantatges de les simulacions és que podrem dissenyar tants mons com puguem imaginar, des de la nostra pròpia casa fins al més complex industrial on haja de treballar el robot i tot al nostre ordinador, sense causar cap molèstia, ni suposar cap perill ni despesa. Hi ha moltes formes de crear móns sobre Gazebo, nosaltres hem triat una amb la que s'obtenen uns molt bons resultats d'una forma prou senzilla i eficient. Bàsicament, es tracta de dissenyar totes les parets com un bloc, per després, anar afegint els objectes que necessitem (mobiliari, decoració, maquinaria auxiliar...). A continuació detallem com férem el plànol de les instal · lacions de l'empresa Robotnik.

Primerament, obtinguem un plànol de l'empresa que simplificarem i al qual només deixarem amb les parets, les quals repassarem amb color negre i deixarem de blanc les zones per on podrà circular el robot. Una vegada fet açò, obtindrem un fitxer amb l'aspecte de la **Figura 6.14**:

Una vegada disposem d'este fitxer **robotnik.png**, dissenyarem un altre, de nom **robotnik.model** en el que crearem a partir de la figura plana un model 3D.

```
<?xml version="1.0" ?>
<model:physical name="map_model">
 <xyz>0 0 0</xyz>
 <rpy>0 0 0</rpy>
 <static>true</static>
 <body:map name="map_body">

    <geom:map name="map_geom">
     <image>robotnik.png</image>
     <threshold>200</threshold>
     <granularity>1</granularity>
     <negative>false</negative>
     <scale>0.0275</scale>
     <offset>0 0 0</offset>
```

[7]http://www.ros.org/wiki/gmapping

```
      <material>Gazebo/White</material>
      <height>2</height>
     </geom:map>

  </body:map>
</model:physical>
```

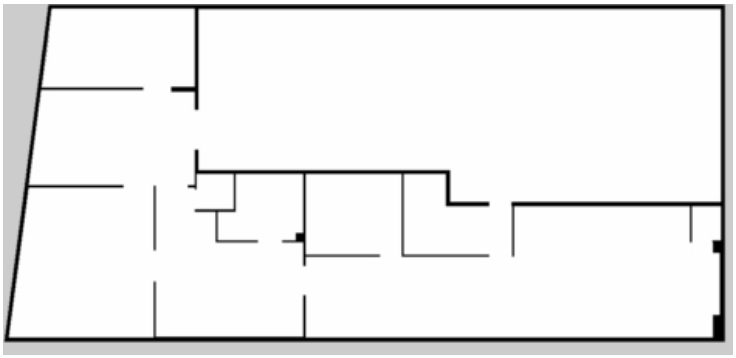

Figura 6.14: Planol de les instal · lacions de Robotnik.

Hem de tindrem molt en compte els atributs **image** i **height**, ja que el primer ens permet seleccionar la imatge que volem fer 3D i el segon l'altura que tindran les parts pintades de negre (en este cas les parets). El resultat d'açò i del seu posterior llançament en Gazebo és el següent:

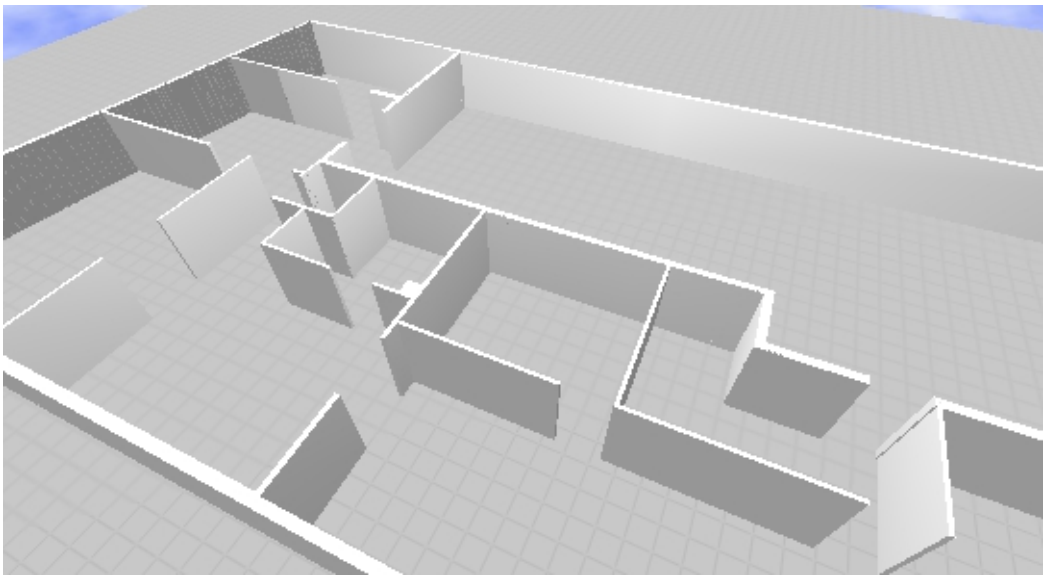

Figura 6.15: Les parets en 3D.

Com vegem, amb pocs passos podem tenir un entorn 3D bastant detallat de les parets de l'oficina. No obstant esta disposa, com és normal, de mobiliari, cartells, decoració... com que tots estos objectes són **URDF**'s podem crear un fitxer anomenat

robotnik.launch on ens encarregarem de carregar-los al servidor de paràmetres per posteriorment llançar-los d'una manera més còmoda. Quan llancem el fitxer **robotnik.launch** acabat de crear, este s'encarregarà de fer tota la feina. Recomanem revisar-lo als annexes. Per finalitzar, mostrem una imatge del món ja complet.

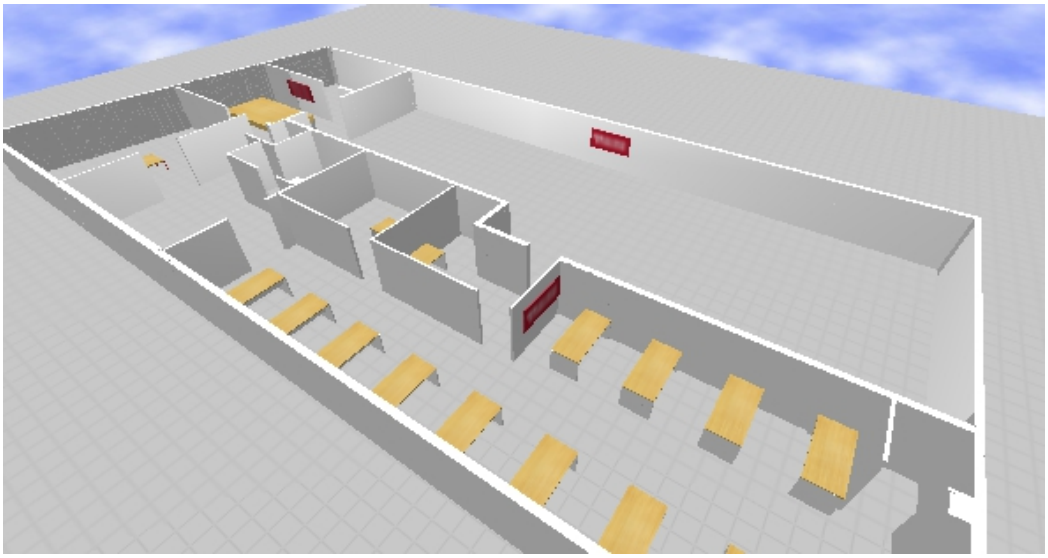

Figura 6.16: El món dissenyat ja acabat.

7 Descripció del treball realitzat

A esta secció explicarem, una vegada ja implementat, diferents aspectes destacables del treball fet durant estos mesos. Dividirem la secció en tres parts, **Arquitectura**, **Interacció amb la comunitat de software lliure** i **Exemples i guies d'utilització**.

7.1 Arquitectura

En esta secció pretenem donar una visió general de l'arquitectura resultant a dos nivells. Primerament des d'un punt de vista a nivell de graf (és a dir, processos i nodes en execució) per a després estudiar el nivell del sistema de fitxers resultant.

7.1.1 Nivell de graf

Per a estudiar este nivell, hem fet ús de la ferramenta **rxgraph**, ja comentada en seccions anteriors. Gràcies a esta ferramenta, podem obtenir un graf on se'ns mostren tots els nodes en execució i la seua comunicació per tòpics. L'aspecte d'este nivell variarà molt depenent del que estem executant, la teleoperació o qualsevol de les navegacions autònomes. Per fer-ho fàcilment comprensible ens centrarem en la teleoperació per després indicar les particularitats de la navegació autònoma.

Si iniciem una **teleoperació mitjançant teclat** (amb el joystick només que canviaren el nom de dos nodes) sobre la simulació i executem l'ordre **rxgraph**, obtenim la següent imatge.

Com puguem observar, tots els nodes estan connectats al tòpic **/clock**, este és publicat per el *Clock Server*, qui publicarà regularment missatges amb la informació relativa al temps. A les simulacions, el paper de *Clock Server* l'efectua el propi simulador que en el nostre cas serà **Gazebo**. Podem vore com del node Gazebo ix una fletxa cap al tòpic, cosa que vol indicar que eixe és el publicador.

Una vegada clar açò, vegem que la teleoperació es resol simplement amb la creació del node corresponent i la publicació de les diferents ordres de forma directa al node

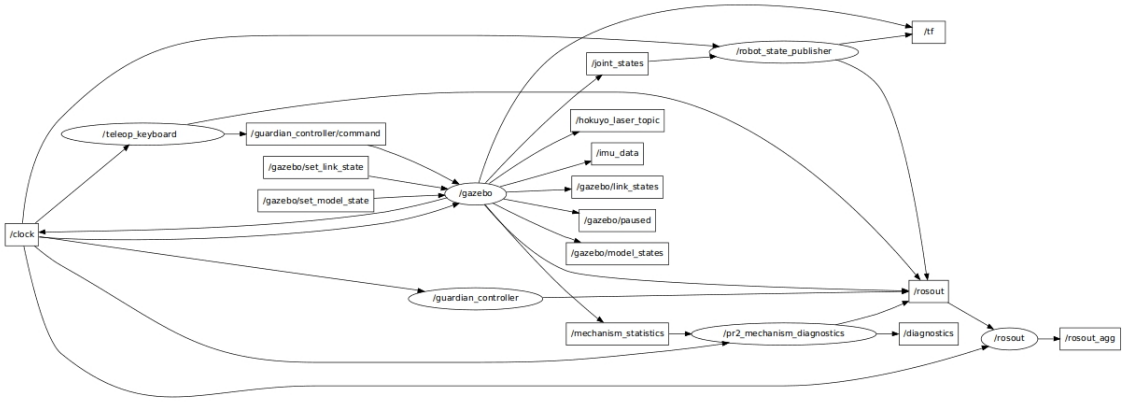

Figura 7.1: Arquitectura de la simulació a nivell de graf.

del simulador mitjançant el tòpic **/guardian_controller/command**. No obstant, si recordem, a la nostra implementació el node que rebia les ordres del teclat no era el simulador, sinó el node ***guardian_controller***, node que si observem ací, vegem que no te cap connexió tret de les dos obligatòries. A que és degut açò? Com sabem, el nostre controlador s'executa en temps real com a plugin del ***Controller Manager***, cosa que fa que s'execute en un entorn diferent on el temps real està garantit a diferència de a tot el sistema restant. Açò té com a conseqüència que la comunicació entre el plugin/controlador i els diferents elements del sistema no siga l'habitual (tòpics, serveis o accions) sinó que es realitza d'una forma transparent a nosaltres.

Per acabar amb este exemple, vegem que Gazebo publica una gran quantitat d'informació. Transformacions, *scans* del làser, informació obtinguda de la IMU, estat de les diferents articulacions... Açò és perquè tal i com ja hem comentat diverses vegades, Gazebo actua com si ell mateixa fora el món real on els *scans* del làser ens els proporcionaria el driver del propi làser, el mateix per a la IMU i on la informació de les articulacions/rodes la obtindríem mitjançant encoders.

Com vegem a primera vista pot semblar una arquitectura sense sentit, ja que el fet d'utilitzar un simulador fa que moltes parts del sistema siguen com una caixa negra, cosa que no ocorre als sistemes reals on cada node té perfectament delimitat el seu àmbit d'actuació i comunicacions.

Per acabar amb la secció ens agradaria comentar que als annexes està disponible un graf de l'**arquitectura del sistema efectuant una navegació autònoma**. Per motius d'espai, no podem col · locar-la ací mateix, però si ens fixem i no ens deixem espantar per la gran quantitat de tòpics publicats pel node ***move_base***, vorem com bàsicament la idea és la mateixa modificant certs aspectes:

- En compte d'haver una teleoperació on s'envien directament les ordres, hi ha un tòpic anomenat **/move_base_node/goal** que s'encarregarà de rebre la posició objectiu.

- Una vegada que el node ***move_base*** efectue el càlcul, passarà la informació amb les referències de velocitat per a aplegar a l'objectiu mitjançant el tòpic de habitual (**/guardian_controller/command**).

- En cas de que s'utilitze algun algorisme de localització (*amcl* o *slam*) el node corresponent s'encarregarà de publicar les transformacions respecte al mapa, cosa que al gràfic no apareix ja que per motius d'espai preferirem mostrar una navegació local.

La forma en que podem visualitzar i interactuar sobre qualsevol navegació autònoma és mitjançant rviz. Podem subscriure'ns a gran quantitat de tòpics que **move_base** ofereix, així com actuar sobre ells (molt important per facilitat les coses en els objectius/*goals* i la posició/*pose*).

7.1.2 Nivell de sistema de fitxers

A esta secció anem a descriure el sistema de fitxers resultant, és a dir com s'han organitzat els diferents paquets i quina utilitat s'espera de cadascun d'ells.

El nivell superior d'organització d'este sistema serà la stack. Esta s'anomena **guardian-ros-pkg** i la podrem trobar disponible al nostre repositori[1]. Este nivell conté els sis paquets que conformen el sistema així com el fitxer manifest de la pila, anomenat **stack.xml**. A continuació descriurem els paquets en qüestió:

- **guardian_2dnav:** Este paquet conté tots els necessari per llançar les diferents navegacions autònomes al robot: AMCL, SLAM i local. Disposa d'una carpeta anomenada maps on s'emmagatzemen els ***mapes*** que poden ser utilitzats en localització, així com d'una carpeta per cada tipus d'esta. A l'interior d'estes, es troben els fitxers de configuració específics per a cadascuna d'elles.

- **guardian_controller:** Este és un dels paquets més importants del robot. En ell s'aplica el bucle de control sobre les rodes, es relacionen estes amb el seus actuadors corresponent així com és s'habilita per rebre ordres pel teclat o el joystick. A l'interior del paquet trobem dos carpetes, una anomenada **/include** on trobem les capçaleres del controlador i altra /src, on trobarem el codi font que implementa estes capçaleres. També trobarem diversos fitxers de configuració per a habilitar-lo com a plugin, estos són **controller_plugins.xml** i **guardian_controller.yaml**. No hem d'oblidar que com tots els paquets, es disposarà dels típics fitxers **manifest.xml**, **CMakeList** i **Makefile**.

[1] http://code.google.com/p/guardian-ros-pkg

- **guardian_description:** Ací es contenen els models URDF del robot, les *meshes*[2] i els diferents elements gràfics que trobem a la simulació, com cartells, taules, cadires... El paquet està dividit en cinc carpetes. Destaquem la carpeta **/launch** perquè es on es contenen els fitxers necessaris per iniciar el robot tres maneres predisenyades diferents: amb l'entorn de Robotnik, amb un entorn buit o directament sense controlador.

- **guardian_joystick_teleop:** Node que permet la utilització d'un joystick per controlar el robot Guardian. Com ja s'ha comentat anteriorment està pensat per al joystick **Logitech Freedom 2.4** però es pot i es permet adaptar fàcilment a altres dispositius. A l'interior d'este paquet trobem dos carpetes, la **/src** on trobarem el codi i la **/launch** on es troba el fitxer necessari per iniciar còmodament este node.

- **guardian_odometry:** Calcula l'odometria del robot basant-se en el moviment de les rodes i certs valors de la IMU. Publica finalment l'odometria al tòpic **/odom**. Este paquet conté solament una carpeta anomenada **/src** i els respectius fitxers habituals que contenen tots els paquets: **manifest.xml**, **CMakeList** i **Makefile**.

- **guardian_tf**: Publica les transformacions estàtiques (relatives al bastidor del robot) al tòpic **/tf**. Segueix la mateixa estructura que el paquet que implementa la odometria amb una carpeta **/src** i els fitxers habituals.

7.2 Interacció amb la comunitat de software lliure

Ja hem comentat anteriorment que ROS compta amb una important comunitat. Esta es dedica a donar suport a iniciats amb el *framework*, resoldre dubtes, escriure guies, compartir software, resoldre i detectar *bugs*, entre altres. Per a introduir-nos en esta comunitat podem fer-ho de moltes formes, depenent del nostre nivell, experiència o voluntat de dedicació. A continuació ens referim a les més importants.

- **ROS Wiki:** La wiki de ROS. D'ací és d'on hem obtingut la major part de la informació. És una pàgina molt completa que va actualitzant-se cada dia per part de desenvolupadors i usuaris. Compta amb una gran quantitat de guies i tutorials, des dels més senzills i generals fins els més complexos i específics.

- **ROS Answers:** És el primer lloc on podem acudir quan una cosa no ens funciona tal i com esperàvem. Podem fer tot tipus de preguntes, des de les més tècniques (requeriments hardware, instal · lacions...) passant per les centrades en el llenguatge fins a les més teòriques. Cal remarcar que en març del 2011

[2]Malles en format **.stl** per donar un aspecte realista al robot.

comptava amb uns 300 missatges i a dia de hui (setembre del mateix any) ronda els 2000.

- **ROS Users:** És una llista de correu per als usuaris de ROS. Útil per a preguntes que no hagen obtingut resposta a **Answers**. També és un bon lloc per a comunicar-se directament amb els desenvolupadors, així com anunciar nous repositoris i actualitzacions.

- **ROS Code:** Lloc on reportar *bugs*, propostes de millores o inclús on podem proposar les nostres pròpies solucions software si tenim el nivell suficient.

Pel que fa a la **nostra experiència** començàrem utilitzant la Wiki, bàsicament per seguir manuals introductoris, tutorials i guies. No obstant, prompte començaren a aparèixer els primers problemes, la majoria d'ells resolts efectuant cerques sobre **Answers** o directament preguntant. Més tard, quan ja teníem el model URDF dissenyat i el controlador implementat detectarem un problema. Quan féiem pivotar el robot sobre si mateix este es frenava en el moment en que s'alineava en qualsevol dels eixos de coordenades del món simulat. A peu de pàgina es troba el link a la pregunta[3] que formulàrem en ROS Answers, on podem trobar també videos explicatius.

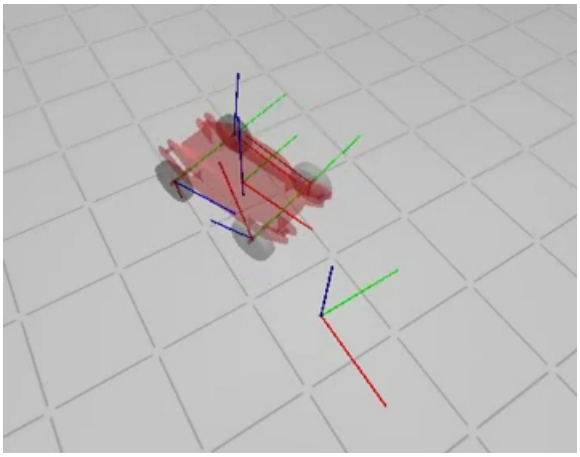

Figura 7.2: Problema de rotació detectat. Recomanem visualitzar els videos per a una millor comprensió.

Després de moltes proves i canvis en els paràmetres dels coeficients de fricció de les rodes en el terra no trobarem solució. Més endavant, i per confirmar que no es tractava d'un error en el controlador implementat, férem totes les proves anteriors efectuant el gir actuant sobre les rodes directament (amb l'ordre **apply_joint_effort**) i sense el controlador, amb la qual cosa el robot seguia comportant-se de la mateixa manera. Veient que no trobàvem cap resposta, decidirem posar-nos en contacte

[3]http://answers.ros.org/question/607/rotation-error-in-gazebo-simulation

directament amb el mantenidor del simulador Gazebo, John Hsu, enginyer sènior a **Willow Garage**. Li enviarem el codi font i li explicarem totes les proves que havíem fet. Al cap d'un dies rebem una resposta on ens explica que està tenint els mateixos problemes.

Ja passades unes setmanes, rebem un correu on ens explica que Gazebo no implementa correctament la física del motor ODE pel que fa a les friccions en el terra. El simulador només té en compte una direcció quan calcula la fricció, però no la seua perpendicular, cosa que si que té en compte el motor ODE, però no el simulador, amb la qual cosa està obviant la fricció perpendicular i per la qual està fallant sempre que efectua una pivotació sobre si mateix. Al cap d'uns dies rebem resposta amb els canvis que permeten que el simulador transmeta correctament estos paràmetres al motor físic. Per provar-los haguérem de reinstal · lar ROS per a fer una instal · lació neta i des dels fitxers fonts, cosa que ens permetria incorporar els canvis proposats abans de compilar. Una vegada fet açò, el resultat fou l'esperat i conjuntament amb els desenvolupadors de Willow Garage havíem aconseguit resoldre un *bug* de Gazebo, *bug* que a la pròxima versió d'este simulador ja no estarà.

Més endavant, detectarem una errada similar. Segons el motor ODE les articulacions "mímiques" estan permeses (una articulació imita el moviment d'una altra) però Gazebo no ho implementa. Ja que detectarem el problema i veient que encara no s'havia reportat, decidirem enviar-lo[4] a **ROS Code**, per si a algún desenvolupador s'animava a solucionar-lo de cara a la nova versió del simulador.

Per acabar i com no pot ser d'altra manera, una vegada tinguérem prou depurat i provat el nostre codi, decidirem posar-lo a disposició de tot el món pujant-lo a un repositori svn[5] a **Google Code**. Des d'ací qualsevol usuari de ROS es pot descarregar el nostre codi, proposar canvis, posar-se en contacte amb nosaltres o documentar-se sobre el robot. El fet de tindre un respositori tant complet dona dret a aparèixer al llistat i menú de robots que utilitzen ROS a la pàgina oficial, tal i com podem vore ací[6].

7.3 Exemples i guies d'utilització

A esta secció anem a exemplificar a mode de guia les cinc formes diferents que hi ha de controlar la simulació del robot **Guardian**. Podem vore dos grans maneres de fer-ho: controlar-lo via teleoperació (ja siga en teclat o joystick) o mitjançant la *Navigation Stack* (utilitzant el posicionament **local**, **AMCL** o **SLAM**). Anem

[4] https://code.ros.org/trac/ros-pkg/ticket/5005

[5] Subversion és un sistema de control de versions dissenyat específicament per reemplaçar el popular CVS.

[6] http://www.ros.org/wiki/Robots

a suposar durant tota la secció que farem les proves en l'escenari/món creat en la **secció 6.8**. que es correspon amb les instal · lacions de l'empresa Robotnik.

7.3.1 Teleoperació mitjançant teclat

Per a teleoperar el robot mitjançant el teclat només haurem d'executar dos senzilles ordres a diferents pestanyes d'una terminal, la primera per carregar el món i el robot, i la segona per executar el node del teclat.

```
roslaunch guardian_description guardian_robotnik.launch
roslaunch guardian_controller teleop_keyboard.launch
```

Una vegada fet açò tindrem disponibles les següents finestres, una amb el Gazebo obert mostrant la simulació i una altra amb la terminal executant el node ***teleop_keyboard***. Com podem vore, el robot utilitzarà les tecles **W-A-S-D** per efectuar un moviment recte, a l'esquerra, enrrere o a la dreta respectivament, mentre que farà servir les tecles **Q-E** per a pivotar sobre si mateix. Tot açò ho farà amb una velocitat normal, mentre que si mantinguem presionada la tecla **Shift**, ho farà a una velocitat superior.

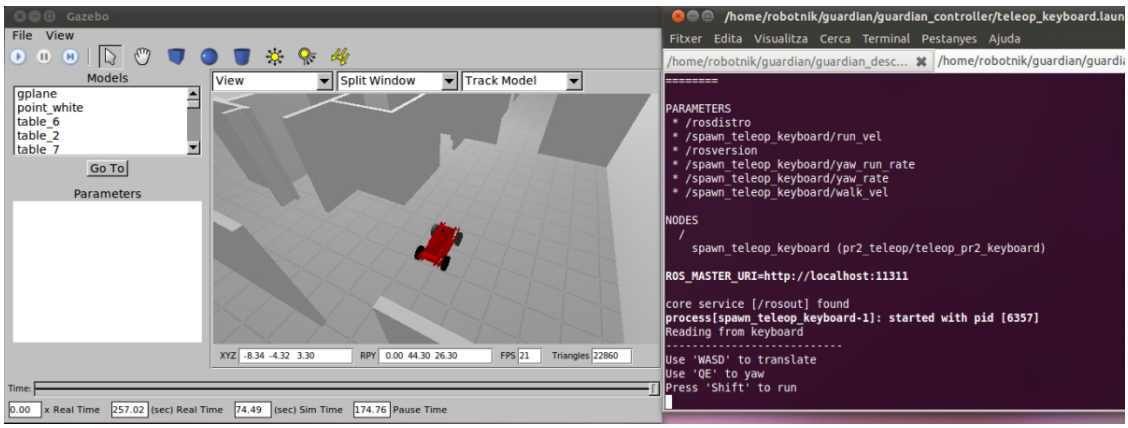

Figura 7.3: Aspecte després de llançar la simulació i el node ***teleop_keyboard***.

A continuació disposem d'unes captures com a exemple d'una teleoperació simple. Primerament tirarem cap avant a una velocitat normal pressionant **W**. Cal indicar que mentre no pressionem cap altra tecla el robot seguirà anant recte, este només pararà en cas de pressionar qualsevol tecla no esperada (qualsevol que no siga **W-A-S-D-Q-E**). En el cas de que es presione alguna de les tecles anteriors, el robot canviarà de sentit com veurem tot seguit.

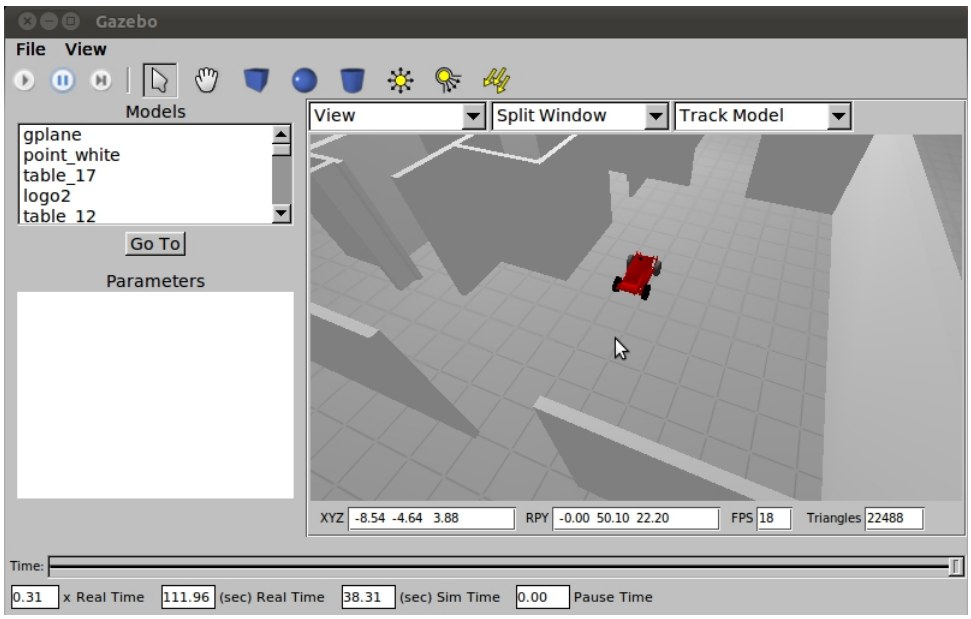

Figura 7.4: Moguem avant el robot presionant **W**.

7.3.2 Teleoperació mitjaçant joystick

La teleoperació mitjançant joystick conté algunes diferencies en respecte a la teleoperació per teclat que cal comentar. Primerament, hem de senyalar que este node va ser escrit per a un joystick **Logitech Freedom 2.4**, així que pot ser en altres models s'han d'adaptar alguns paràmetres per a que el control siga el òptim. Per llançar este tipus de teleoperació només hem de fer el següent:

```
roslaunch guardian_description guardian_robotnik.launch
roslaunch guardian_joystick_teleop guardian_joy.launch
```

Per a poder controlar millor el robot, afegirem un control de velocitat mitjançant els botons superiors **3** i **4**. Inicialment i per seguretat, la velocitat del robot està fixada a un 10% de la seua potència i amb successives pulsacions d'estos botons podem fixar-la a la desitjada. És molt útil en entorns perillosos o amb poca mobilitat, ja que no haurem de preocupar-nos de si les nostres accions sobre la palanca del joystick son més o menys violentes, ja que tindrem la velocitat en tot moment limitada.

Pel que fa al control és l'esperat, palanca avant/arrere (roig a la imatge) generarà un moviment igual, mentre que si tirem la palanca a l'esquerra només es mouran les rodes dretes i si fem el propi amb la palanca cap a la dreta només ho faran les esquerres (de color verd a la imatge). Així i tot, esta ultima forma no és la més eficient per girar. Per a fer-ho d'una manera eficient podem fer que el robot pivote sobre ell mateix, cosa que farem girant la palanca sobre si mateixa a l'esquerra o la dreta (moviment blau a l'esquema). Cal destacar que este moviment no el tenen tots

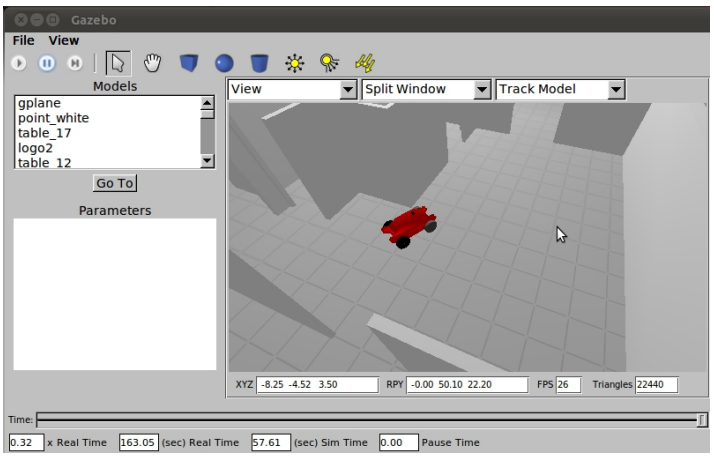

Figura 7.5: Després de pivotar sobre si mateix pressionant **E**, el robot anirà rapidament cap enrrere si presionem **Shif+S**.

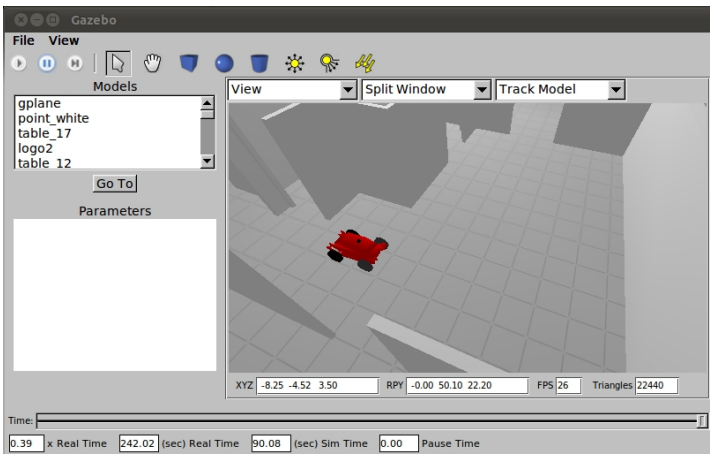

Figura 7.6: Una vegada en la posició dessitjada, presionarem la **barra espaiadora**.

els joysticks i en cas d'utilitzar algun que no tinga este moviment disponible haurem d'eliminar els dos moviments anteriors esquerra/dreta només movent les rodes d'un costat per el d'estes pivotacions, on cada costat du una direcció.

7.3.3 Navegació autònoma utilitzant posicionament local

Una vegada vistes les formes de teleoperar sobre el robot, anem a passar a explicar com llançar i fer ús de la navegació autònoma. Primerament hem de deixar clara una cosa que pot ser més endavant ens dificulte la comprensió d'alguns conceptes i és que **Gazebo** és un simulador i per tant simularà el món real mentre que **Rviz** és només un visualitzador 3D del que aquest està observant i capturant pels diferents

Figura 7.7: Moviments als eixos del Logitech Freedom 2.4

sensors que l'integren. Una vegada deixat açò clar, que tornem a repetir, pareix clar, anem a vore com executar este tipus de navegació.

Primerament iniciarem el simulador **Gazebo** com estem fent habitualment, seguidament cridarem als nodes encarregats de publicar les transformacions estàtiques i l'odometria (***guardian_tf*** i ***guardian_odometry***) per després llançar el node ***move_base*** amb la configuració corresponent. Per últim i per tal de visualitzar i actuar sobre el robot d'una forma més còmoda, executarem el visualitzador **Rviz**. A continuació la seqüència d'ordres seguides:

```
roslaunch guardian_description guardian_robotnik.launch
roslaunch guardian_2dnav guardian_configuration.launch
roslaunch guardian_2dnav move_base_local.launch
rosrun rviz rviz
```

Açò donarà com a resultat que se'ns obrin dos finestres, una corresponent al simulador i altra al visualitzador, a esta última vorem el robot, els obstacles i els *scans* del làser en cas d'haver-ho fet correctament (per a açò hi ha un tutorial molt complet i que recomanem seguir ací[7]). Este es l'aspecte que mostrarà:

Una vegada obertes estes dos finestres podrem punxar a l'**Rviz** a "2D Nav Goal" i a continuació seleccionar en qualsevol punt de la finestra. Com podem vore al vídeo que adjuntem[8], el robot es mourà fins a la posició seleccionada, fent ús només de la localització mitjançant la odometria.

[7]http://www.ros.org/wiki/navigation/Tutorials/Using%20rviz%20with%20the%20Navigation%20Stack
[8]http://www.youtube.com/watch?v=uWzecxRjBqE

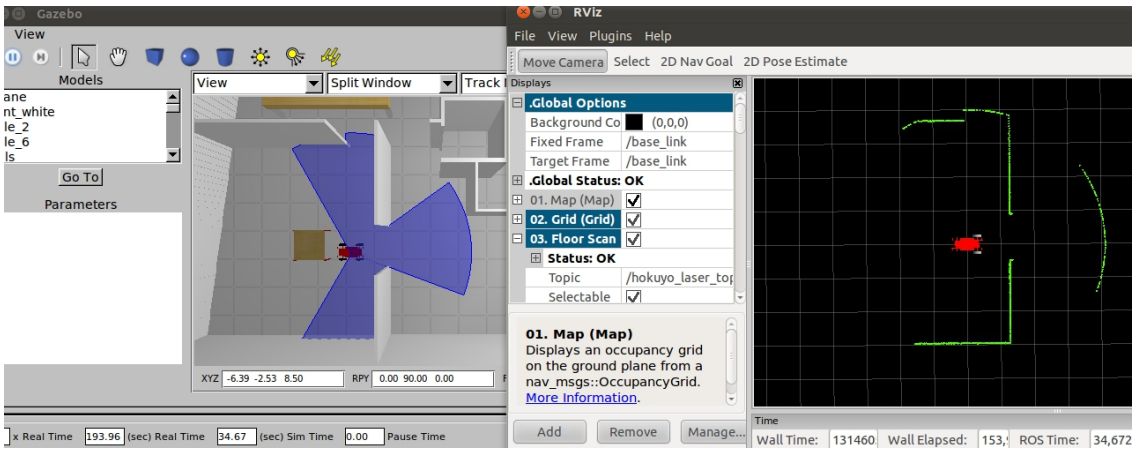

Figura 7.8: Navegació autònoma acabada d'iniciar.

7.3.4 Navegació autònoma utilitzant posicionament AMCL

Com hem vist en la secció corresponent, este tipus de navegació fa servir, a banda de l'odometria clàssica, un mapa i l'algorisme AMCL per localitzar-se. Este algorisme utilitza un filtre de partícules per dur un seguiment de la posició aproximada del robot sobre un mapa conegut. Açò, juntament amb l'odometria, fa que la navegació utilitzant este sistema de localització siga molt precisa. Per executar este tipus de navegació seguirem els següents passos. Com vegem només canviarà el penúltim pas, on cridarem al node ***move_base*** però ara amb una altra configuració.

```
roslaunch guardian_description guardian_robotnik.launch
roslaunch guardian_2dnav guardian_configuration.launch
roslaunch guardian_2dnav move_base_amcl.launch
rosrun rviz rviz
```

Posem un vídeo[9] on s'exemplifica com s'utilitza esta navegació. Com vegem, és més intuïtiva que l'anterior ja que a l'Rviz vegem el mapa intern carregat al ***map_server*** i el robot respecte a este en tot moment, així com els obstacles i els *scans* obtinguts del món real (en este cas Gazebo). A l'igual que en l'anterior cas, podem seleccionar un destí punxant sobre "2D Nav Goal" i seleccionant, ara sí, qualsevol punt sobre el mapa, sempre tenint en compte de no seleccionar una paret o un obstacle.

[9]http://www.youtube.com/watch?v=g7mQLRG56Vk

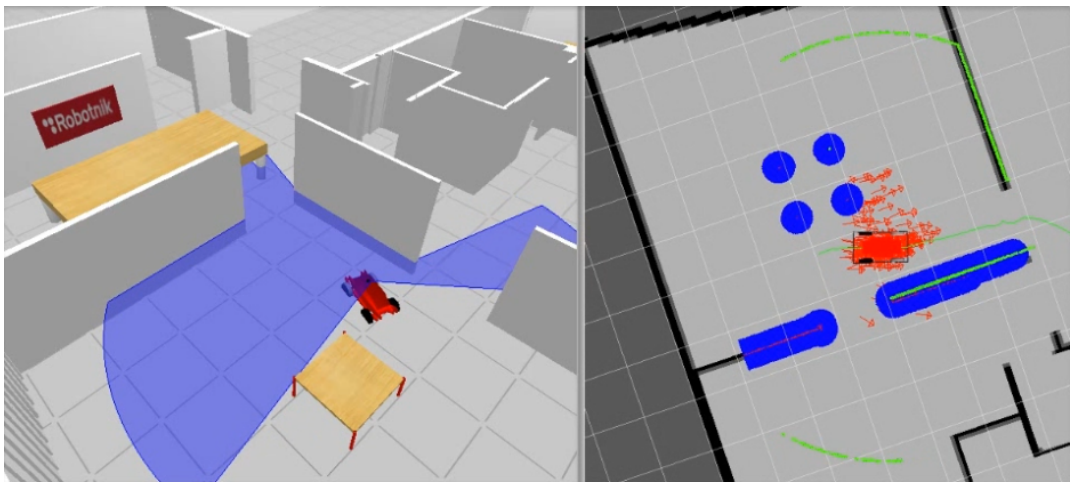

Figura 7.9: Navegació autònoma amb localització AMCL.

7.3.5 Navegació autònoma utilitzant posicionament SLAM

Ja per últim presentem la navegació SLAM. Com ja hem estudiat, este sistema s'encarrega de localitzar i mapejar simultàniament un entorn. Amb açò volem dir que al principi el robot només tindrà com a mapa global el seu entorn immediat, és a dir el que des del seu punt inicial puguem capturar els làsers. Segons avance el robot, este anirà construint el mapa, que al mateix temps servirà per a situar-se d'una forma més precisa que només amb l'odometria. És un sistema molt complet i útil ja que obtenim el millor del AMCL sense haver de conèixer prèviament el mapa. Per llançar-lo tant sols haurem d'executar les següents accions.

```
roslaunch guardian_description guardian_robotnik.launch
roslaunch guardian_2dnav guardian_configuration.launch
roslaunch guardian_2dnav move_base_amcl.launch
rosrun rviz rviz
```

Una vegada fet açò tindrem una imatge pareguda a esta, on si li donem un objectiu en el mapa vorem com conforme avança el robot es va creant el mapejat de l'entorn tal i com demostra el video[10].

[10]http://www.youtube.com/watch?v=KWHDUUDdtus

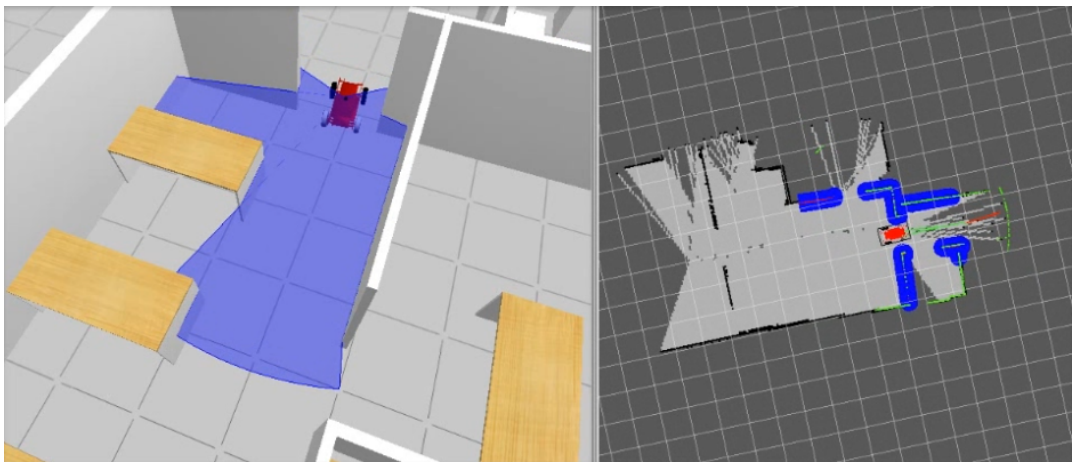

Figura 7.10: Navegació autònoma amb localització SLAM.

8 Conclusions i treballs futurs

Per acabar, anem a fer un xicotet resum de tot el que ha suposat este treball. Els objectius principals del projecte eren la programació i simulació d'un robot en ROS. Per a aconseguir-ho, primerament, ens haguérem d'introduir en un nou *framework* de treball, ROS, un ferm candidat a convertir-se en l'estàndard a l'àmbit de la robòtica d'ací uns anys. Una vegada aprés els elements bàsics d'este *framework*, començàrem a fer el disseny físic del nostre robot, complicant-lo cada vegada més a l'hora que afegint-li funcionalitats. Més tard dissenyarem i implementàrem el control i la teleoperació. Una vegada fetes les proves pertinents, començarem la part dedicada a la navegació autònoma, aprenent nous algorismes de guiat així com utilitzant les implementacions que incorpora este *framework*. Com vegem el projecte ha seguit un procés clarament incremental, poc a poc i partit de zero hem aconseguit programar i simular un robot en ROS per a més tard i fent ús dels avantatges del llenguatge, provar diferents algorismes de navegació autònoma. Cal indicar, que el projecte en si mateix ens va ocupar uns quatre mesos.

Podem concloure en que ha sigut un projecte molt interessant des de les diferents vessants en que el podem vore. Primerament, i com no pot ser d'altra manera, hem acomplit els objectius inicials, la programació i simulació del robot en ROS, i a més els hem ampliat afegit-li la possibilitat de navegació autònoma. Amb tot el desenvolupament del projecte hem aprés com utilitzar el nou *framework* per a robòtica ROS, i a més, amb açò, hem millorat els nostres coneixements de C++, ja que la implementació del *framework* que utilitzàrem era la basada en este llenguatge. També hem interactuat amb la comunitat de software lliure, hem conegut gent directament implicada en el desenvolupament de ROS, així com hem contribuït amb ella amb els nostres repositoris i documentació. Per acabar, el fet de realitzar el projecte en una empresa, m'ha permés dedicar-me més a ell, aprenent més i obtenint un millor resultat, a més, el estar com a becari a una empresa de les poques del sector de la robòtica de servici a l'estat com és **Robotnik** és molt motivant i enriquidor.

Pel que fa a treballs futurs, el projecte es troba en un punt on a partir d'ell poden sorgir gran quantitat d'altres projectes o investigacions. A continuació llistem les més importants:

- Sense anar més lluny, l'eixida natural d'este projecte una volta implementada la simulació és el dur estos nodes al robot real. Està adaptació no serà

immediata però molt d'estos nodes podran ser aprofitats directament o realitzant xicotets canvis, altres en canvi s'hauran de reescriure completament, tot i que arribats a este punt la cosa serà més senzilla, doncs ja tinguem certa experiència.

- Desenvolupament i investigació amb nous algorismes de navegació sobre la simulació. Amb açò ens estalviem proves "perilloses" amb el robot real, millor accés als resultats i possibilitat d'investigació en qualsevol moment, encara que no tinguem el robot a l'abast.

- Integració de nous dispositius com per exemple càmeres, braços robotitzats, nous sistemes de tracció...

- Migració del codi ROS escrit en versió ***Diamondback*** a la propera versió amb eixida imminent de nom ***Electric***.

En definitiva, considere que el projecte ha sigut una experiència molt enriquidora que m'ha permés aprofundir d'una manera única en el món de la robòtica, a més, el fet de plasmar-ho en una memòria ha fet que puga fiançar els coneixements adquirits al llarg d'aquest període a més dels obtinguts al llarg de la carrera.

Annexes

Donada la inconveniència de plasmar en aquest treball escrit la totalitat del codi desenvolupat i disposant d'un repositori públic on, a més d'estar la versió referida al text podem obtenir la última versió, optem per no transcriure'l i adreçar-vos a la següent direcció:

`http://code.google.com/p/guardian-ros-pkg/`

També, i per la mateixa raó, vos indiquem l'adreça del *Cheat Sheet* del framework.

`http://www.ros.org/wiki/Documentation?action=AttachFile&do=get&target=ROScheatsheet.pdf`

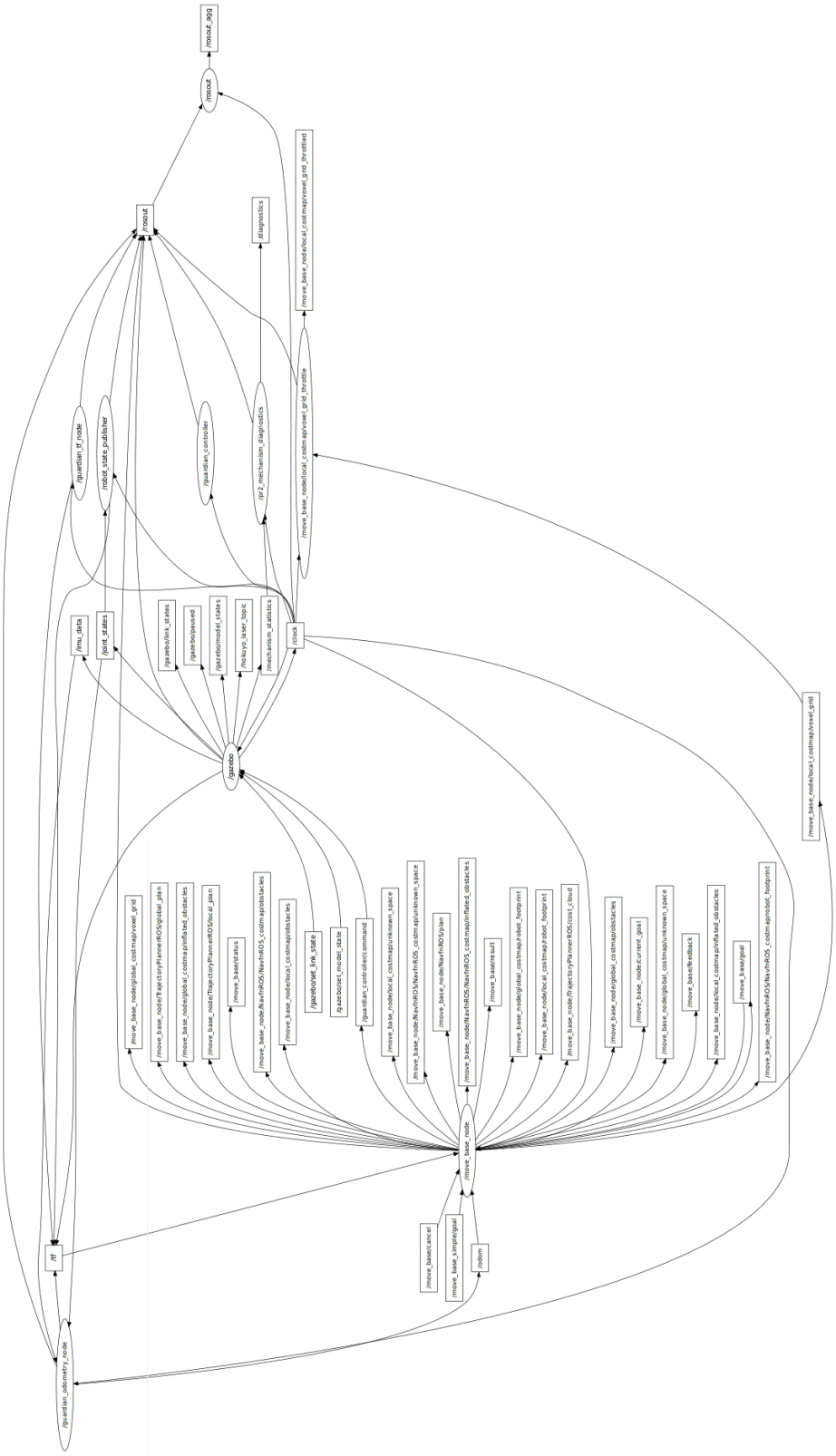

Figura 8.1: Arquitectura de la simulació amb navegació autònoma a nivell de graf.

Bibliografia

[**Addison-Wesley**] Phillip McKerrow. *Introduction To Robotics*, 1991.

[**Cambridge University Press**] Gregory Dudek i Michael Jenkin. *Computational Principles of Mobile Robotics*, 2010.

[**ICRA99**] Frank Dellaert, Dieter Fox, Wolfram Burgard i Sebastian Thrun. *Monte Carlo Localization for Mobile Robots*, 1999.

[**ICRA09**] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, Andrew Ng. *ROS: an open-source Robot Operating System*, 2009.

[**ICRA10**] Eitan Marder-Eppstein, Eric Berger, Tully Foote, Brian P. Gerkey i Kurt Konolige. The office marathon: Robust navigation in an indoor office environment, 2010.

[**ODE**] Open Dynamics Engine. http://www.ode.org/, 2011.

[**OGRE**] Ogre-open-source graphics rendering engines. http://www. ogre3d.org/, 2011.

[**ROS Wiki**] Ros. http://www.ros.org, 2011.

[**ROS Wiki**] Ros stacks. http://www.ros.org/wiki/Stacks, 2011.

[**ROS Wiki**] Ros master. http://www.ros.org/wiki/Master, 2011.

[**ROS Wiki**] Ros messages. http://www.ros.org/wiki/Messages, 2011.

[**ROS Wiki**] Ros nodes. http://www.ros.org/wiki/Nodes, 2011.

[**ROS Wiki**] Ros packages. http://www.ros.org/wiki/Packages, 2011.

[**ROS Wiki**] Ros parameter server. http://www.ros.org/wiki/Parameter%20Server, 2011.

[**ROS Wiki**] Ros services. http://www.ros.org/wiki/Services, 2011.

[**ROS Wiki**] Gazebo plugins. http://www.ros.org/wiki/gazebo_plugins, 2011.

[**ROS Wiki**] Ros topics. http://www.ros.org/wiki/Topics, 2011.

[**ROS Wiki**] Ros URDF. http://www.ros.org/wiki/urdf, 2011.

[**SLAM**] Søren Riisgaard i Morten Rufus Blas. *SLAM for Dummies*.

[**University of Michigan**] J. Borenstein, H. R. Everet i L. Feng. *Where amb I?*, 1996.

[**Wikipedia**] Cinemàtica. http://ca.wikipedia.org/wiki/Cinem%C3%A0tica, 2011.

[**Wikipedia**] Control PID. http://es.wikipedia.org/wiki/Proporcional_integral_derivativo, 2011.

[**Wikipedia**] Robotica. http://en.wikipedia.org/wiki/Robot, 2011.

[**Willow Garage**] Robot PR-2. http://www.willowgarage.com/pages/pr2/overview, 2011.

[**Willow Garage**] Simulador Gazebo. http://gazebo.willowgarage.com/, 2011.

[**Willow Garage**] Willow Garage. http://www.willowgarage.com, 2011.

Índex de figures

Índex de taules