ROS - Move Base

Nicolas Limpert und Christian Schnieder

[1] Fachhochschule Aachen - University of Applied Sciences
[2] Robotik WS 2014 / 2015

Zusammenfassung. Dieser Bericht stellt eine Zusammenfassung der verschiedenen Komponenten des in ROS implementierten Move_Base-Pakets dar.
Das ROS-Package move_base stellt die Möglichkeit dar mit gegebener Karte und Anfangsposition ein örtliches Ziel mit einem Roboter zu erreichen, auf dem der Navigation Stack [1] in ROS vollständig implementiert ist, da die Move_Base lediglich eine Schnittstelle für den Benutzer darstellt, damit der Nutzer einen Navigationsbefehl absetzen (beispielsweise über das Tool *RViz*) kann und dieser Befehl von der Base unter Berücksichtigung von Kollisionsvermeidung ausgeführt wird.

1 Einleitung

Das Move_Base-Paket ist ein ROS-Paket, das dazu verwendet wird einem Roboter ein örtliches Ziel zu vermitteln und dieses Ziel in Kombination von mehreren ROS-Nodes (globalen- und lokalen Planer, globale und lokale costmap, etc.) zu versuchen, zu erreichen.
Das Ziel wird der Move Base in Form einer Action vermittelt, mit der Idee aus einer ROS-Message nach Möglichkeit eine Reihe von Fahrbefehlen unter Berücksichtigung von Kollisionsvermeidung, optimaler Pfadplanung (in Abhängigkeit von lokalem und globalem Planner) auszuführen.
Das Move_Base-Paket stellt dadurch einen essentiellen Teil des nav_cores in ROS dar, da es einem Benutzer bzw. einer Anwendung einfach ermöglicht ein gewünschtes örtliches Ziel zu erreichen, indem das Zusammenspiel zwischen den verschiedenen Komponenten des nav_cores koordiniert wird.

2 Motivation

Aufgrund der steigenden Komplexität in Robotersystemen möchte man die verschiedenen Teilaufgaben gemäß moderner Softwareentwicklungsverfahren gestalten bzw. wie in der objektorientierten Softwareentwicklung einer Instanz des Systems eine Aufgabe erteilen, die sich dann um die Erfüllung (oder Scheiterung) dieser Aufgabe kümmert.
Durch eine geschickte Gestaltung der verschiedenen Softwarekomponenten erhält man eine einfache Wart- und Erweiterbarkeit des Systems, die sich in einem Robotersystem so äußern könnte dass man einem Teil der Software mitteilt dass

man ein bestimmtes Ziel erreichen möchte und dieses Ziel dann von diesem Teil erfüllt wird.
Dies ermöglicht die Move_Base, in dem ihr mittels ROS-Message ein Ziel in Form einer geometry_msgs/PoseStamped Nachricht vermittelt wird, welches Ziel zu erreichen ist. Nachfolgend wird im Detail erklärt, welche Komponenten dabei wichtige Rollen spielen.

3 Komponenten

Move_base besteht aus folgenden Komponenten: [2]

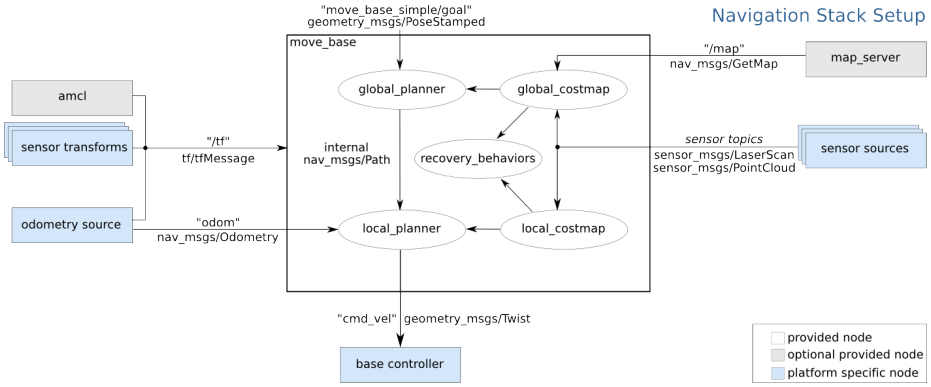

Abb. 1. Komponentenübersicht
[3]

Globaler Planer Dies kann ein beliebiger globaler Planer sein, solange er das Interface nav_core::BaseGlobalPlanner erfüllt. Er ist für die Pfadplanung innerhalb der Karte zuständig, führt also beispielsweise eine A* - Suche durch, um von einem gegebenen Anfangspunkt den gewünschten Endpunkt zu erreichen.

Lokaler Planer Dies kann ein beliebiger Planer sein, solange er das Interface nav_core::BaseLocalPlanner erfüllt. Aufgabe des lokalen Planers ist das Erreichen des nächsten Punkts den der globale Planer gegeben hat, bzw. das ausgeben von Fahrbefehlen zum erreichen dieses nächsten Punkts.

Globale Costmap Dies kann ein beliebiger Planer sein, solange er das Interface nav_core::BaseLocalPlanner erfüllt

Lokale Costmap Dies kann ein beliebiger Planer sein, solange er das Interface nav_core::BaseLocalPlanner erfüllt

Recovery Behaviour Dies kann ein beliebiger Planer sein, solange er das Interface nav_core::BaseLocalPlanner erfüllt.

Nachfolgend werden die einzelnen Komponenten beschrieben.

3.1 Globaler Planer

Der globale Planer erhält unter dem ROS-Topic *move_ base_ simple/goal* eine Message vom Typ *geometry_ msgs/PoseStamped*, die unter Berücksichtigung der globalen Costmap beispielsweise mithilfe von A* berechnet, welcher Pfad durch die einzelnen Knotenpunkte in der Karte zum Ziel führen.
Hierbei wird seitens der ROS-Wiki als grundliegende Implementationen *navfn* und der *carrot_ planner* aufgeführt:

navfn Über die jeweilige Karte wird ein Grid gelegt, durch das von der Anfangsposition aus bis zum Endpunkt eine Suche mittels Dijkstra durchgeführt wird, die den optimalen Pfad liefert, sofern er verfügbar ist.
Laut Diskussion wurde hier auf Dijkstra gesetzt, da er im Vergleich zu einem A* - Algorithmus eher optimale Pfade liefert. [6]

carrot_planner Wenn der Zielpunkt innerhalb eines Objekts liegt, wird entlang des Vektors vom Roboter zur Zielposition so lange zurück gegangen, bis ein Punkt ausserhalb des Objekts gefunden wurde, den der Roboter erreichen kann. Dadurch wird dem Roboter ermöglicht, das Ziel so nah wie möglich zu erreichen. Es wird hierbei also nicht viel geplant, sondern lediglich einem lokalen Planer ein einfacher Plan zur Verfügung gestellt. [7]

3.2 Lokaler Planer

Aufgabe des lokalen Planers ist das Erreichen des nächsten Punkts, der im vom globalen Planer übermittelten Plan steht. Grundliegend ist in ROS der *base_ local_ planner* implementiert:

base_local_planner Dieses Paket beinhaltet zwei verschiedene Implementierungen zur Ausführung eines vom globalen Planer gegebenen Pfads. Dazu zählt das *Trajectory Rollout* und der *Dynamic Window Approach (DWA)*, die folgendes Verfahren gemeinsam haben:

1. Ermittle verschiedene Bewegungsmuster die dem Roboter zur Verfügung stehen (dx, dy, dtheta)
2. Simuliere eine Bewegung für jedes dieser Bewegungsmuster um abzuschätzen, was beim Ausführen dieser Bewegung passieren würde wenn die Bewegung für einen bestimmten (kurzen) Zeitraum ausgeführt wird.
3. Bewerte jedes Bewegungsmuster, das in der Simulation ausgeführt wurde anhand von Werten wie Abstand zu anderen Objekten, Nähe zum Zielpunkt, Nähe an den globalen Pfad und Geschwindigkeit. Verwerfe ungültige Bewegungsmuster, also solche, die mit anderen Objekten kollidieren.
4. Nimm das Bewegungsmuster mit der höchsten Bewertung und gebe dieses als ausführbare Steuerbefehle (standardmäßig *geometry_ msgs::Twist* an Topic *cmd_ vel*) aus.

5. Wiederhole ab 1. Punkt.

Der konkrete Unterschied zwischen dem *Trajectory Rollout* und dem *Dynamic Window Approach (DWA)* besteht in der Erstellung der Bewegungsmuster. *Trajectory Rollout* probiert verschiedene mögliche Geschwindigkeiten über die gesamte gegebene Simulationszeit in Abhängigkeit von den für den Roboter möglichen Beschleunigungen.
DWA probiert die verschiedenen Geschwindigkeiten für nur einen Simulationsschritt (auch in Abhängigkeit von den Beschleunigungen).
DWA wird laut ROS-Wiki aufgrund seiner höheren Effizienz (es probiert innerhalb von kleineren möglichen Bewegungsmuster als *Trajectory Rollout*) [8] Zum *DWA* gibt es zusätzlich eine separate Implementierung in ROS:
http://wiki.ros.org/dwa_local_planner.

3.3 Globale und lokale Costmap

Die Costmap berechnet anhand von Sensorinformationen ein 2D oder 3D - Raster mit Belegungen und erweitert die Karte um Belegungen bzw. TODO [9]

3.4 Recovery Behaviours

Die Recovery Behaviours sind ein optionales Feature der Move_Base und eignen sich für einen Umstand, in dem der Roboter zwar durch den globalen Planer einen gültigen Plan erhalten hat, aber diesen, beispielsweise aufgrund der Dynamik in der Umwelt, nicht länger verfolgen kann und deswegen einen neuen Pfad berechnen muss.

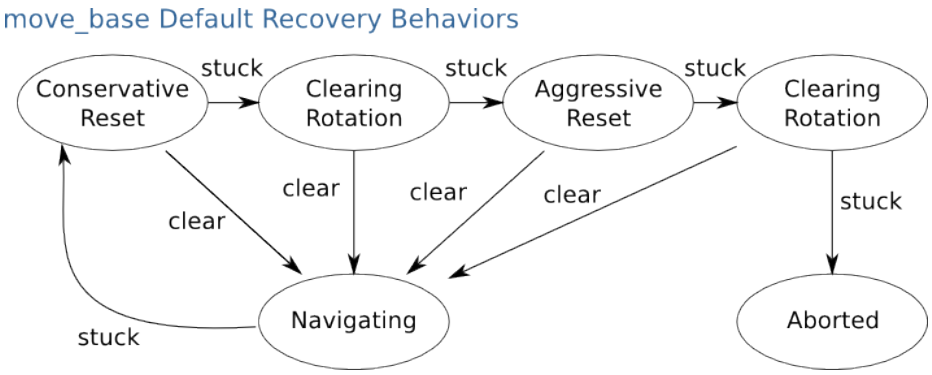

Abb. 2. Recovery Behaviours
[4]

Dabei wird folgendes Schema verwendet: [5]

Navigating	Solange die Navigation ohne Probleme weiter läuft (globaler Pfad ist verfolgbar), fahre mit der Navigation fort.
Conservative Reset	Wenn der Roboter feststeckt werden zu Beginn alle Objekte ausserhalb eines vom Benutzer festgelegten Bereichs entfernt. Danach wird nach Möglichkeit eine Rotation auf der Stelle durchgeführt, die den Raum innerhalb der costmaps zu klären, damit ein neuer Pfad gefunden werden kann.
Aggresive Reset	Wenn das Conservative Reset nicht geholfen hat, wird beim Aggressive Reset die costmap innerhalb eines festgelegten Bereichs vollständig zurückgesetzt und anschließend eine erneute Clearing Rotation durchgeführt.
Aborted	Wenn auch der vorige Schritt fehlschlägt wird davon ausgegangen dass aus der aktuellen Situation kein Ausweg besteht und die Navigationsaufgabe wird abgebrochen.

4 Beispielhafte Implementierung an einem Rover

Nachfolgend wird ein Beispiel mithilfe der vom Maskor-Institut der Fachhochschule Aachen gestellten Rover aufgeführt, mit dem die Verwendung der *move_base* möglich ist, in dem man für den globalen und lokalen Planer andere Pakete nimmt, die sich eher für Roboter eignen, die ein Fahrgestell nach Ackermann besitzen.

4.1 Problem: Ackermann im Vergleich zu (nicht-) holonomen Robotern

Ein Roboter, der ein Ackermann-Fahrgestell besitzt unterscheidet sich von anderen mobilen Robotern grundliegend darin dass er sich einerseits nicht auf der Stelle drehen kann und sich auch nicht seitlich bewegen kann.
Die oben genannten Implementationen der globalen und lokalen Planer gehen aber davon aus dass der Roboter sich zumindest auf der Stelle drehen kann.
Dies erschwert die Möglichkeit, den *ROS Navigation Stack* mit Robotern zu verwenden, die Ackermann-getrieben sind. [10]

Lösung Für die Pfadplanung (bzw. Aktionsplanung) existiert eine von der Carnegie Mellon University (Dr. Maxim Likhachev [14]) entwickelte Bibliothek, die sich (*SBPL - Search Based Planning Library*) [11] nennt und es, neben anderen Features, erlaubt unter Verwendung des *SBPL Lattice Planners* eine Folge von Aktionen durchzuführen, die einem Ackermann-getriebenen Roboter ermöglichen ein Ziel zu erreichen.
SBPL kommt hier unter der Verwendung des Lattice Planners als globaler Planer zum Einsatz, da er die Möglichkeit darstellt, einen Pfad zu berechnen der den Aktionenraum des gegebenen Roboters berücksichtigt und nicht darüber hinaus geht.

4.2 Globaler Planer

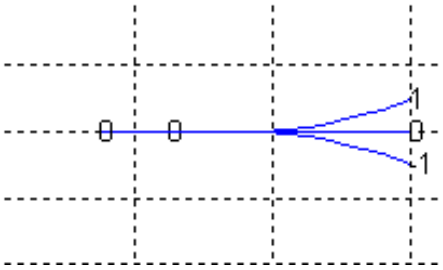

Abb. 3. Motion Primitive
[13]

Wie bereits beschrieben kommt hier der SBPL Lattice Planner zum Einsatz.
Damit der Lattice Planner weiß, welche Möglichkeiten er für die Umsetzung seines Plans hat, benötigt er neben der Karte (die nach wie vor eine costmap ist) eine Reihe von Motion Primitives, diese werden nachfolgend beschrieben:

Motion Primitives sind im SBPL-Kontext Beschreibungen für Aktionen, die ein Roboter in einem Zustand durchführen kann.
Am Beispiel des Rovers wäre das also entweder eine Geradeausfahrt oder eine Fahrt stark oder leicht zur einen oder anderen Seite eingeschlagen.

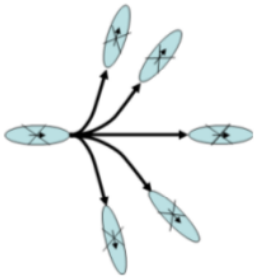

Abb. 4. Motion Primitives im Graphen
[13]

Pfaderstellung SBPL legt mit den Motion Primitives einen Graph über die Umweltsind im SBPL-Kontext Beschreibungen für Aktionen, die ein Roboter in einem Zustand durchführen kann. Am Beispiel des Rovers wäre das also entweder eine Geradeausfahrt oder eine Fahrt stark oder leicht zur einen oder anderen Seite eingeschlagen.

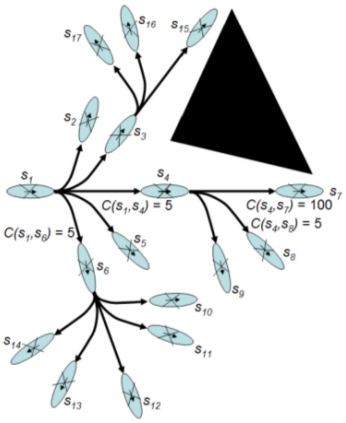

Abb. 5. Motion Primitive
[13]

4.3 Lokaler Planer

4.4 Globale und lokale Costmap

4.5 Globale Costmap

4.6 Recovery Behaviours

5 Action API

Die move_base Knoten stellt eine Implementierung des SimpleActionServer (siehe actionlib Dokumentation) zur Verfüpgung der die Ziele der geometry_msgs/PoseStamped Nachrichten enthält. Die Kommunikation mit den move_base Knoten ist direkt über ROS möglich. Es wird aber empfohlen die Ziele über den SimpleActionClient zu schicken, sofern eine Nachverfolgung des Status gewünscht wird.

5.1 Action Subscribed Topics

move_base/goal (move_base_msgs/MoveBaseActionGoal)
Ein Ziel das move_base in der Welt verfolgen soll.

move_base/cancel (actionlib_msgs/GoalID)
Anfrage um ein bestimmtes Ziel nicht mehr zu verfolgen.

5.2 Action Published Topics

move_base/feedback (move_base_msgs/MoveBaseActionFeedback)
Enthält die aktuelle Position der Basis in der Welt

move_base/status (actionlib_msgs/GoalStatusArray)
Enthät Statusinformationen der Ziele die an move_base geschickt werden.

move_base/result (move_base_msgs/MoveBaseActionResult)
Rückgabewert ist leer für move_base action.

6 Topics

6.1 Subscribed Topics

move_base_simple/goal (geometry_msgs/PoseStamped)
Bietet ein non-action interface für move_base für Benutrzer bei dennen die Nachverfolgung der Aktionen nicht relevant ist.

6.2 Published Topics

cmd_vel (geometry_msgs/Twist)
Forlaufende Bewegungsbefejle die durch eine mobile base ausgeführt werden sollen.

7 Services

make_plan (nav_msgs/GetPlan)
Ermöglicht einem externen Benutzer einen Plan zu einem Ziel zu bekommen ohne das move_base diesen verfolgt.

clear_unknown_space (std_srvs/Empty)
Ermöglicht einen externen Benutzer den Bereich um den Roboter auf leer zu setzen. Dieses kann verwendet werden wenn der Roboter in einen neue Umgebung gesetzt wird.

clear_costmaps (std_srvs/Empty)
Erlaubt einen externen Benutzer Objecte aus der costmap zu entfernen. Da dieses zu Kollisionen führen kann, sollte diese Funktion mit Vorsicht benutzt werden. Sie kann dennoch hilfreich sein, wenn es Fehlern in der costmap gibt.

8 Parameter

Nachfolgend werden Parameter an die Move_Base aufgelistet, die entweder beim Starten der Move_Base beispeilsweise in einem Launch-File oder über den ROS-Parameterserver gesetzt werden.

base_global_planner (string, default: navfn/NavfnROS")
Name des Plugins das als Globaler Planer mit move_base benutzt wird. VGL. Abschnitt "Komponenten / Globaler Planer".

base_local_planner (string, default: "base_local_planner/TrajectoryPlannerROSFF-or 1.1+ series)
VGL. Abschnitt "Komponenten / Lokaler Planer".

recovery_behaviors (list, default: [name: conservative_reset, type: clear_costmap_recovery/ClearCostmapRecovery, name: rotate_recovery, type: rotate_recovery/RotateRecovery, name: aggressive_reset, type: clear_costmap_recovery/ClearCostmapRecovery])
VGL. Abschnitt "Komponenten / Recovery Behaviours".

controller_frequency (double, default: 20.0)
Die Rate in Hz in der die Fahrbefehle an den Basiskonten gesendet werden.

planner_patience (double, default: 5.0)
Wert wie lange in Sekunden der Planer versucht einen Plan zu finden bis space-clearing Operationen eingeleitet werden (siehe Recovery Behaviours).

controller_patience (double, default: 15.0)
Wert wie lange in Sekunden der Kontroller auf richtige Komandos wartet bis spaceclearing Operationen eingeleitet werden.

conservative_reset_dist (double, default: 3.0)
Abstand von Hindernissen in Meteren zum Roboter die aus der costmap entfernt werden wenn versucht wird Platz in der costmap zu schaffen.

recovery_behavior_enabled (bool, default: true)
Ein und ausschalten der move_base recovery behaviors.

clearing_rotation_allowed (bool, default: true)
Erlaubt oder Verbietet Rotationen um die eigene Achse beim Versuch von clear out space Hinweis: Diese Parameter wird nur beim Standard recovery behaviors benutzt

shutdown_costmaps (bool, default: false)
Eingabe Paramter ist ein Boolean. Gibt an ob die costmap ausgeschaltet wird wenn die move_base auf inaktiv gesetzt ist

oscillation_timeout (double, default: 0.0)
Wert in Sekunden bis das recovery behaviour ausgelöst wird.

oscillation_distance (double, default: 0.5)
Strecke in Metern die der Roboter zurücklegen muss um sicher zu sein nicht zu schwingen. Setzt den Timer oscillation_timeout zurück.

planner_frequency (double, default: 0.0)
Frequenz in Hertz in der die Globale Planber Schleife ausgeführt wird. 0.0 führt den Globalen Planer nur bei einem neuen Ziel aus oder der lokale Planer eine unmögliche Route meldet.

Literatur

[1] Navigation, “http://wiki.ros.org/navigation”
[2] Move_Base, “http://wiki.ros.org/move_base”
[3] Move_Base-Components Image
“http://wiki.ros.org/move_base?action=AttachFile&do=view&target
[4] Recovery Behaviours Image
“http://wiki.ros.org/move_base?action=AttachFile&do=get&target=
[5] Rotate Recovery “http://wiki.ros.org/rotate_recovery”
[6] navfn, “http://wiki.ros.org/navfn”
[7] CarrotPlanner, “http://wiki.ros.org/carrot_planner”
[8] base_local_planner, “http://wiki.ros.org/base_local_planner”
[9] costmap2d, “http://wiki.ros.org/costmap_2d”
[10] Ackermann Group, “http://wiki.ros.org/Ackermann
[11] SBPL, “http://sbpl.net”
[12] SBPL Lattice Planner, “http://wiki.ros.org/sbpl_lattice_planner”
[13] SBPL ROS School,
“http://www.ros.org/wiki/Events/CoTeSys-ROS-School?
action=AttachFile&do=get&target=robschooltutorial_oct10.pdf”
[14] Dr. Maxim Likhachev, “http://www.cs.cmu.edu/ maxim/”