Der erste Schritt besteht in der Implementierung einer fähigen Infrastruktur, um sowohl die effiziente Implementierung als auch Erprobung von Lösungsansätzen zu ermöglichen. Nach dem Vorbild von [1] wird eine Kombination aus ROS und Gazebo verwendet, wodurch ein Algorithmus zunächst in der Simulationsumgebung Gazebo getestet und im unmittelbaren Anschluss auf den realen Roboter übertragen werden kann. Bei Gazebo handelt es sich um eine Simulationsumgebung, die einerseits komplexe Dynamik in einer dreidimensionalen Welt zu simulieren vermag, andererseits über ein Plugin mit ROS gekoppelt werden kann, wodurch eine einzelne Implementierung sowohl für die Simulation als auch den realen Roboter verwendet werden kann.

Um den Turtlebot in ROS zu modellieren, wird das sogenannte Universal Robotic Descripction Format (URDF) verwendet [1], welches einer XML-ähnlichen Struktur folgt. In [1]wird die URDF-Datei um zusätzliche Tags erweitert, welche die Kompatibilität zu dem in Gazebo verwendeten Simulation Description Format (SDF) schaffen. Um die Simulationsergebnisse zu visualsieren verwenden Takaya et al. Das RViz Packet.

Die ROS Architektur wird in drei Ebenen: Filesystem, Computation Graph und Community, unterteilt.

Die untereste Ebene, das Dateisystem, setzt sich aus den elementaren Bausteinen von ROS zusammen, wobei als elementarer Baustein so gennante Packages verwendet werden. Dabei handelt es sich Ordner, welche die minimalen Anforderungen an ein ausführbares Programm unter ROS enthalten. Die Dokumentation eines Package erfolgt in Form eines Manifest, welches relevante Information wie Abhängikeiten und Compiler-Einstllungen enthält. Um umfangreiche Funktionalitäten zu erreichen, werden mehrere Packages zu seinem so genannten Stack zusammengefasst. Als Beispiel dient ein Navigationsstack, der sich aus Funktionen für die Auswertung von Sensorik, SLAM-Verfahren, Pfadplannung und Ansteuerung der Aktorik zusammensetzt, welche jeweils als eigenes Package implementiert werden. Die Dokumentation des Stacks erfolgt ebenfalls in Form eines Manifest, welches in diesem Fall als Stack-Manifest bezeichnet wird.

Am Beispiel von Stacks wird bereits deutlich, dass ein Grundprinzip von ROS darin besteht, dass mehrere Prozesse interagieren. Um Daten zwischen Prozessen auszutauschen werdne Nachrichten verwendet, welche im Rahmen von ROS als Messages bezeichnet werden.

Zusätzlich gibt es noch Services, die ähnlich wie Nachrichten allerdings nur kurz beschrieben werden und deshalb auch an dieser Stelle nicht weiter ausgearbeitet werden.

In der zweiten Ebene wird ROS als Netzwerk beschrieben, in welchem die verschiedenen Prozesse verknüpft sind. Die einzelnen Elemente des Netzwerks werden als Nodes bezeichnet, wobei es sich um Prozesse handelt, die über das ROS Netzwerk miteinander verbunden sind. Ein besonderer Bestandteil des Netzwerks stellt der so genannte Master dar, der eine Namen Registrierung und Lookup für die restlichen Nodes bereitstellt, wodurch dieser notwendig für die Kommunikation zwischen den Nodes wird. Eine weitere zentrale Einheit bildet der Paramter Server, welcher die Möglichkeit bietet, Daten in einer zentralen Position mithilfe von Keys zu speichern. Die Kommunikation zwischen den Nodes erfolgt über Nachrichten, welche in Form von Message Types auf der Ebene des Dateisystems festgelegt wurden. Bei der Nachrichtenkommunikation wird ein Publish/Subscribe Schema eingehalten, welches darauf basiert, dass alle Nachrichten mit einem Namen, der als Topic bezeichnet wird, versendet werden. Nodes können Topics subscriben, woraufhin alle Nachrichten dieser Topic an sie weitergeleitet werden. Da es sich bei der Nachrichtenkommunikatoin mithilfe von Topics um eine many-to-many Verbindung handelt, können Topics nicht genutzt werden, um eine direkte Kommunikation zwischen zwei Nodes umzusetzen, wofür die so genannten Services verwendet werden. Wenn eine Node einen Service anbietet, muss dieser einen einzigartigen Namen besitzen, woraufhin andere Nodes den Service über das ROS-Netzwerk ansprechen und nutzen können. Als letzte Einheit der Computation-Graph-Ebene sind die so genannten Bags zu nennen, welche als Datenspeicher fungieren. Beispielsweise können Nachrichten in Bags gespeichert werden, um den gesamten Werteverlauf eines Sensors im Anschluss an ein Experiment zu rekonstruieren.

Erstellen von eigenen Packages erfolgt mithilfe des Befehls

Roscrate-pkg [package\_name] [depend1] [depend2] [depende3]

Beispiel: roscreate-pkg chapter2\_tutorials std\_msgs rospy roscpp

Erstellt das Package chapter2\_tutorials, das von den Packeten std\_msgs, rospy und roscpp abhängt. Wichtig dabei ist, dass das neue Packet bzw. der Ordner, der das Packet darstellt, auf dem ROS\_PACKAGE\_PATH liegt.

Anleitung nach ROS-Wiki für Turtlebot Simulation in Gazebo/Visualisierung mit Rviz [3]:

Intsallation:

sudo apt-get install ros-kinetic-turtlebot ros-kinetic-turtlebot-apps ros-kinetic-turtlebot-interactions ros-kinetic-turtlebot-simulator ros-kinetic-kobuki-ftdi ros-kinetic-rocon-remocon ros-kinetic-rocon-qt-library ros-kinetic-ar-track-alvar-msgs

Fertiges Beispiel, um den Turtlebotsimulation in Gazebo zu starten:

roslaunch turtlebot\_gazebo turtlebot\_world.launch

Durch das festlegen der Umgebungsvariable TURTLEBOT\_GAZEBO\_WORLD\_FILE kann eine .world-Datei bestimmt warden, die als Umgebung für den Trutlebot in Gazebo genutzt wird.

Um die Geschwindigkeit de sRoboters zu setzen, muss die Topic /mobile\_base/commands/velocity gepublished werden, wofür die Nachricht geometry\_msgs/Twist mit dem Format:

rostopic pub /mobile\_base/commands/velocity geometry\_msgs/Twist "linear:

x: 0.1

y: 0.0

z: 0.0

angular:

x: 0.0

y: 0.0

z: 0.0"

ausgegeben werden.

Alternative Installation [https://bitbucket.org/damienjadeduff/456\_kinetic\_turtlebot/src/b8eacebf174bdf2e11cb288dc61cac4bd163fb01/install\_456\_students.sh?at=master&fileviewer=file-view-default]

sudo apt-get -y install ros-kinetic-turtlebot-gazebo ros-kinetic-desktop-full ros-kinetic-turtlebot-rviz-launchers ros-kinetic-warehouse-ros || { echo "Could not install main turtlebot packages."; exit 1;}

sudo apt-get -y install ros-kinetic-gazebo-ros ros-kinetic-gazebo-plugins ros-kinetic-gazebo-ros-control python-rosdep python-wstool ros-kinetic-ros ros-kinetic-warehouse-ros libspnav-dev || { echo "Could not install some extra ROS packages."; exit 1;}

Welt setzen:

TURTLEBOT\_GAZEBO\_WORLD\_FILE=/opt/ros/kinetic/share/turtlebot\_gazebo/worlds/corridor.world

export TURTLEBOT\_GAZEBO\_WORLD\_FILE

roslaunch turtlebot\_gazebo turtlebot\_world.launch

Über Rviz kann man zusätzlich visualisieren:

roslaunch turtlebot\_rviz\_launchers view\_robot.launch

Als großen Vorteil bringt Rviz mit, dass man nachschauen kann wie die topics für Tiefen- und Farbbilder heißen.

Teleoperation:

roslaunch turtlebot\_teleop keyboard\_teleop.launch

Position des Turtlebot in Gazebo anzeigen lassen:

rostopic echo /gazebo/model\_states/pose[9]/position

9 ist in diesem Fall der Index der ‘mobile\_base’ findet man über

rostopic echo /gazebo/model\_states/name

**27.11.2017:**

Morgens habe ich mich in die Partikel-Filter einglesen, die zugrunde liegende Idee ist recht trivial, weshalb ich mich recht schnell davon ab- und der Inbetriebnahme des Navigation-Stacks hingewandt habe. Allerdings bin ich dann auch bei dem Navigations-Stack auf ein paar unschöne Probleme gestoßen.

Als erstes habe ich meine Kenntnisse bezüglich launch-files auf Stand gebracht, was bereits zu ersten brauchbaren Ergebnissen geführt hat. Bei der Problembekämpfung habe ich auch noch beiläufig über Gmapping und hector-SLAM dazugelernt. Nämlich arbeitet gmapping mit einem Partikel-Filter und mit Ray-Casting, was dazu führt, dass gmapping große Schwierigkeiten hat sich in ähnlichen Szenaieren zu lokalisieren. Das wurde bei dem Korridor-Besipiel sehr deutlich, wo der Roboter kontinuierlich in der Gegen umherspringt. Allgemein hat gmapping in den Simulationsszenarien sehr schlecht abgeshcnitten. Der Kommentar des Tutors war, man muss sich Mühe geben eine gute Karte zu zeichnen.

Aber auch hector\_slam hat seine Schwächen und zwar braucht es einen Weitwinkel Laserscanner, um zu funktionieren. In seiner Verteidigung muss man sagen, das es eben dafür konzepiert wurde. Endergebnis ist im Korridor wieder das vollständige Versagen des Ansatzes. Was zu dem Problem geführt hat, dass keiner der beiden Ansätze eine vernünftige Karte des Testszenarios hinbekommt. Außerdem wurde mir klar, dass ich den Sensor eventuell in gazebo implementieren muss, was sowohl zeit- als auch nervenintensiv ist. Besonders beschissen ist, dass das Kartenmalen überhaupt nicht Bestandteil meiner Arbeit ist. Aus diesem Grund habe ich mir überlegt, dass ich die Karte auch händisch malen kann und mit komischen Laserscanner die Navigation simuliere.

Der Vorteil ist, dass ich in der Simulationsumgebung die Kofiguration des Stacks durcharbeiten und dokumenteiren kann. Dann ist der Übergang in die Realität auf Seite der Infrastruktur kein Sprung ins kalte Wasser. Allerdings wird es schwieriger die Ergebnisse des Algorithmus mit der Realität zu vergleichen. An der Stelle muss man aber auch erwähnen, dass die Definition der Ziele noch bevorsteht. Insbesondere die Forderung, dass die Roboter Hindernissen ausweichen sollen, ist gleichermaßen wage wie nicht simulierbar. Insofern ist sowieso in Frage zu stellen, wie aussagekräftig die Simulation sein kann. Vielleicht muss man sich auf Seite der Simulation damit begnügen, dass sie lediglich zur Konfiguration und Illustration der Algorithmen genuttz werden kann. Zusätzlich besthe die Möglichkeit, dass man in der Realität auftretende Probleme in der Simulationsumgebung isoliert rekonstruieren kann, wodurch die Untersuchung vereinfacht wird.Ein weiteres Argument ist, dass die Konfiguration und Implementierung von Sensoren in gazebo pisslangweilig ist, keinen der Beteiligten interessiert udn ich es auch für keienn besonders wertvolle Erfahrung halte.

So betrachtet, denke ich, dass der richtige Ansatz darin besthet, die Karte händisch zu zeichnen und mit Standard-Konfiguration den Navigations-Stack möglichst schnell zum Laufen bekommen. Dabei gilt es die Konfiguration der Navigation möglichst detailiert zu erfassen, um Gefühl und Verständnis für die Stellschrauben zu erlangen. Anhand der Ergebnisse wird dann entschieden, wie es mit der Realität weitergeht, wobei die Anforderungen vom Chief auch nochmal wichtig sind.

Daraus würde ich für die nächsten zwei Wochen das Ziel formuileren, den vereinfachten Simulationsfall zum Laufen zu bekommen, um den Navigations-Stakc zu konfigurieren. Am Ende der zwei Wochen soll eine Dokumentation der Algorithmen plus deren Umsetzung in Ros vorliegen, die ich wiederum mit dem Chief durchgehen kann, um die letztendlichen Ziele für die Realität zu definieren. Das heißt für morgen: Erstmal die Karte zusammenpfuschen, Termin bei Martens über die Aufgabenstellung, mit der Karte Navigationsstack anschmeisen, und dann die Euinzelteile angehen. Navigationsstack soll sowohl mit als auch ohne AMCL laufen.

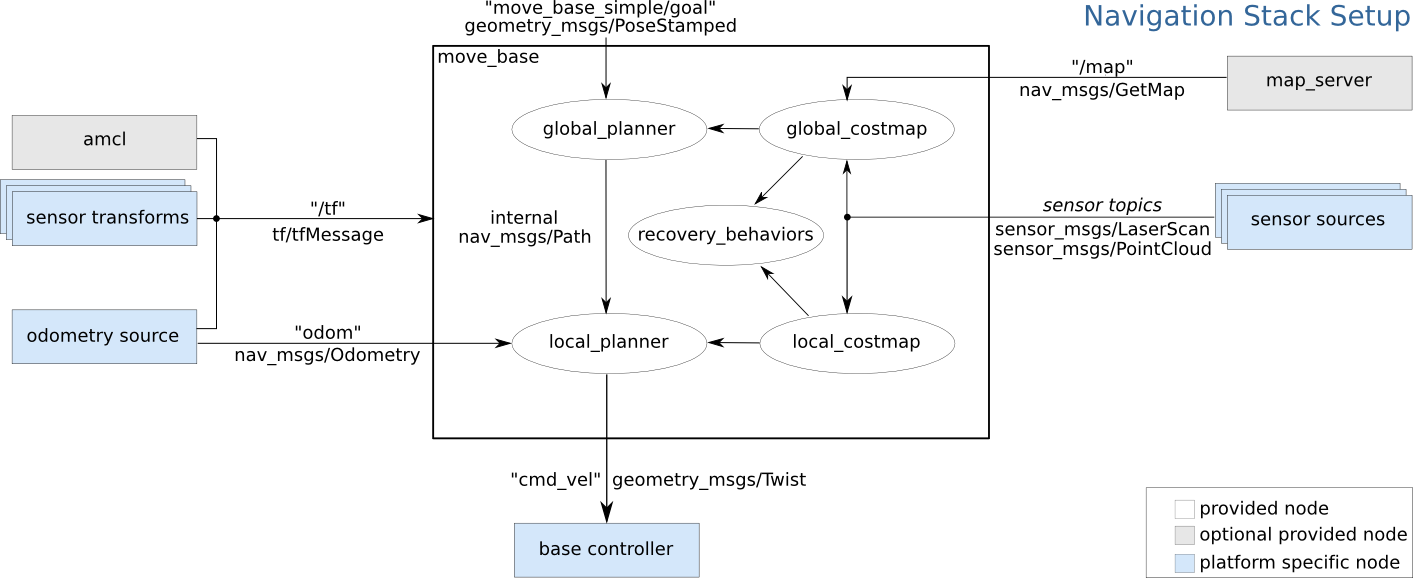
**28.11.2017:**

Die ersten Aufgaben heute bestand darin den Navigation-Stack in der Standard-Konfiguration zum Laufen zu bringen. Als erster Punkt stand die Erstellung der Korridor-Karte auf dem Plan. Die Lösung von dem Problem bringt MATLAB und der map\_server. Die bei der Kartographierung erstelle Karte wird über eine MATLAB-ROS-Node publiziert, von dem map\_server empfangen, der die Karte wiederum abspeichert.

Als nächstes war dann der Navigation-Stack dran, wobei das Ganze sowohl mit als auch ohne AMC-Lokalisierung funktionieren sollte. Außerdem sollen playground und Korridor-Karte verwendet werden können. Nach Tutorial hat der Standard-Fall mit Playground und AMCL recht schnell funktioniert, nach paar Problemchen auch ohne AMCL (hier mus sman manuell die Transformation von Odometrie zu Karten-Frame vorgeben, was ohne Lokalisierung die Identitätsabbildung ist). Um den Korridor zum Laufenzu bringen war der map\_server schwierig, weil in dem gespeicherten yaml-file noch eine NaN-Wert war, der den Server zum Absturz bringt. Den Faller kann man allerdings händisch korrigieren, woraufhin das Ganz funktoiniet.

**29.11.2017:**

Die letzten Tage stand die Inbetriebnahme des ROS-Navigation-Stack auf dem Programm, was bisher auch zu zufriedenstellenden Ergebnissen geführt. Jetzt geht es damit weiter das Ganze mit einem höherem Detailierungsgrad zu verstehen. Es soll die Brücke zwischen der ROS-Implementation und dem bisher dokumentierten Algorithmen geschlagen werden. Dafür starten wir wieder bei dem berühmt berüchtigten Bild:



Anhand der Darstellung werden die Hauptbestandteile ersichtlich, denen man auch jeweils Algorithmen und Funktionsprinzipien zuordenen kann. Diese zu anschauliche Darstellung verleitet einen dazu, zu glauben, das System verstanden zu haben. Allerdings steckt der Teufel im Detail: Sobald man sich näher damit beschäftigt, wie die – zum Teil nur oberflächlich- dokumentierten Algorithmen im ROS-Stack parametrisiert werden, kommen die ersten Probleme auf. An diesen Stellen tretten die bisher vernachlässigten Kopplungen zwischen mathematischer Algorithmik und informatiklastiger ROS-Implementation auf. Um von Verstädnis sprechen zu können, müssen wir diese Verzahnung in ihre kleinsten Bestandteile zerlegen und nachvollziehen. Wir beginne wie so viele; aber mit dem Anspruch tiefer vorzudringen:

Die Steuerung des Roboters wird im Navigation-Stack von dedm Packet move\_base übernommen, indem der globale und lokale Planer realisiert werden. Für die Algorithmen werden eine entsprechend globale und lokale Kostenkarte bereitsgestellt. Als letzte Komponente des Packets ist das recovery\_behavior zu nennen, dessen Beschreibung vorest noch warten muss. Die algorithmischen Prinzipien der beiden Planer wurden bereits ausführlich erläutert. Jetzt wenden wir uns den pragmatischeren Aspekten zu. Zu Beginn gilt die Aufmerksamkeit einmal mehr dem globalen Planner.

**Global-Planer**

Für die Umsetzung der globalen Plannung wird das ROS-Packet global\_planner [4] verwendet. Dieser verwendet eine Kostenkarte, wobei es sich um ein Occupancy-Grid, also eine ortsdiskrete Gitterdarstellung, handelt. Die Kostenkarte erfüllt die Funktion der Aktionsgewichtung. Für die Ermittlung des Pfades kann wahlweise entweder Dijkstra‘s Algorithmus oder A\* verwendet werden. Bei Betrachten der Wiki-Dokumentation fällt einem recht schnell auf, dass man die wissenschaftlichen Erklärungen der Lehrbücher hinter sich gelassen hat und es nun mit Internetdokumentation zu tun hat. Bei der Dokumentation wird das potential von Zellen und Pfaden berechnet, was synonym mit den hier genannten Kosten zu verstehen ist.

Ingesamt können bei dem Planer sieben Paramter konfiguriert werdne [4], wobei lediglich die folgende für den Anwendungsfall interessant sind:

- /allow\_unkown (bool, default:true):

Mit dem Pramter wird bestimmt, ob auch Pfade durch unbekannte Zellen geplant werden soll. Bei einer unbekannten Zelle ist nicht gewisst, ob diese frei oder von einem Hindernis belegt ist.

- /default\_tolerance (double, defualt:0.0):

Toleranzabstand mit dem der geplante Pfad die Zielposition erreichen soll, da es sich um eine diskrete Darstellung handelt stellt die Default-Toleranz von 0 kein Problem dar.

- /use\_dijkstra (book, default: true):

Flag ob dijkstra oder A\* verwendet werden soll. Der Grund für Dijkstra ist, dass die in A\* verwendeten Heuristiken manchmal gegen die Unterschätzungsforderung verstoßen, weshalb suboptimale Pfade erzeugt werden können.

- /use\_quadratic (bool, default: true)

Führt nach [4] dazu, dass eine quadratische Approximation für die Berechnung des Potentials/der Kosten genutzt wird. Im Folgendensatz shcreibt der Autor dennoch „Of what, the maintainer of this package has no idea“ [4]. Etwas tiefer im Internet stößt man auf die seriös wirkende Antwort: Und zwar berechnet das navfn-Packet (global\_planner ist eine neuerer Ersatz der jedoch die gleichen Funktionen bietet) das Potential aus der mit einer Zelle verbunden Kosten und zusätzlich der Distanz zwischen der vorherigen und folgenden Zelle. Die quadratische Approximation wird an dieser Stelle verwendet, um auch Quer-Verbindunge zu ermöglichen, da wie in unseren Beispieln nur oben/unten/links/rechts also vierweg-Verbindungen betrachtet werden.

-/use\_grid\_path(bool, default:false)

Wen auf true gestellt, folgt der Pfad wie in unserem Beispiel den Zellen, was dazu führt, dass der Roboter sich nur in 8 Richtungen bewegen kann (durch use\_qudratic wird die schräge bewegung über zellen möglich?!?!). Stellt man den Paramter auf false wird der Pfad geglättet, woraus rundere Bahnen resultieren.

**Globale Kostenkarte**

Die Basis für die Berechnung des globalen Pfades stellt die globale Kostenkarte dar, die mithilfe des ROS-Packetes „costmap\_2d“ [6] erstellt wurde.

Morgen steht die Kostenkarte an, als ertes wie das costmap packet mits eienn drei layern funktoiniert, wie konfiguriert werden kann, bzw wie diese Konfiguration für globale und lokale kostenkarte aussieht.

Als steht der lokale Planner auf dem Program, was auch morgen noch drin sein sollte.

[1] Kenta Takaya, Toshinori Asaiy, Valeri Kroumov, Florentin Smarandach: „Simulation Environment for Mobile Robots Testing Using ROS and Gazebo“

[2] Aaron Martinez, Enrique Fernandez: “Learning ROS for Robotics Programming”

[3] TurtleBot – ROS-Wiki: [http://wiki.ros.org/Robots/TurtleBot?distro=indigo#turtlebot.2BAC8-Tutorials.2BAC8-indigo.Simulation](http://wiki.ros.org/Robots/TurtleBot?distro=indigo" \l "turtlebot.2BAC8-Tutorials.2BAC8-indigo.Simulation)

[4] Global-Planner <http://wiki.ros.org/global_planner>

[5] ros-blog über navfn/global\_planer quadratic approx [https://answers.ros.org/question/11388/navfn-algorism/?answer=16891#answer-container-16891](https://answers.ros.org/question/11388/navfn-algorism/?answer=16891" \l "answer-container-16891)

[6] costmap\_2d http://wiki.ros.org/costmap\_2d