Telematics
Computer Systems

Freie Universität Berlin

Computer Systems and Telematics — Distributed, Embedded Systems

Bachelorarbeit

Design und Implementierung einer Analysesoftware im Kontext eines Referenzsystems zur Indoorlokalisierung

Benjamin Aschenbrenner

Matr. 4292264

Betreuer: Prof. Dr.-Ing. habil. Jochen Schiller
Betreuender Assistent: Dipl.-Inform. Heiko Will

Institut für Informatik, Freie Universität Berlin, Deutschland

12. Dezember 2011

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, sind als solche gekennzeichnet. Die Zeichnungen oder Abbildungen sind von mir selbst erstellt worden oder mit entsprechenden Quellennachweisen versehen. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner Prüfungsbehörde eingereicht worden.

Berlin, den 12. Dezember 2011

(Benjamin Aschenbrenner)

Zusammenfassung

Zusammenfassung

Diese Arbeit steht in Kontext zum Aufbau eines Referenzsystems zur indoor Lokalisierung. Das Referenzsystem soll es ermöglichen die Genauigkeit von indoor Lokalisierungen, durchgeführt mit mobilen Sensorknoten eines Wireless Sensor Network (WSN), zu ermitteln. Die Grundlage des Referenzsystems ist dabei ein mobiler Roboter, welcher autonom, vorgegebene Wegpunkte in einer Karte abfährt. Bei einer solchen Fahrt zeichnet der Roboter sowie an ihm befestigte Sensorknoten einen Pfad durch regelmäßige Lokalisierung auf. Aufbauend darauf, geht es im Rahmen dieser Arbeit um die Implementierung eines Analysewerkzeugs namens *Pathcompare*, welches ermöglicht, die dabei entstandenen Pfaddaten zusammenzuführen, für den Tester aufzuwerten und zu visualisieren. Neben mittleren Abstand zum gewählten Referenzpfad werden Parameter wie Pfadlänge, Anzahl der Pfadpunkte, empirische Varianz und eine Liste der größten Abweichungen angezeigt. Alle Daten können als Comma Separated Values (CSV) exportiert werden. *Pathcompare* ist in das ROS integriert und so entwickelt, dass es über Plug-ins erweitert und angepasst werden kann.

Abstract

This thesis is associated with the development of a indoor localization reference system. The aim of this reference system is to evaluate the precision of mobile sensor node localization data. The sensor nodes are organized in a wireless sensor network. The basis of the reference system is a mobile robot, that is able to autonomously navigate to given waypoints in a map. While moving, the robot and sensor nodes mounted on it, generate path data by continuously localizing. This work is about the creation of an analysing tool named *Pathcompare* that is used to merge the path data of different sources and visualize them for a tester. The software shows the median distance of a path to the given reference path, the overall pathlength, total number of points per path, variance and also a list of the greatest distances to the reference path. All results can be exported in a CSV file. *Pathcompare* is integrated into the ROS and can be extended via a Plug-in mechanism.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Glossar

Robot Operating System Ein Framework welches zahlreiche Pakete bzgl. Nachrichtenaustausch zwischen verteilten Programmen, Hardwareabstraktion und Robotik bietet. v, 4, 7

Akronyme

API Application Programming Interface. 8, 15

CST Computer Systems and Telematics. 1

CSV Comma Separated Values. v, 16

IMU Inertial Measurement Unit. 3

MEMS Microelectromechanical systems. 3

MOC Meta Object Compiler. 11

UIC User Interface Compiler. 11

WSN Wireless Sensor Network. v, 1, 5

XML-RPC XML - Remote Procedure Call. 8

KAPITEL 1

Einleitung

1.1 Motivation

Systeme zur Lokalisierung werden für zahlreiche Zwecke genutzt und deren Bedeutung wächst parallel zur Verbreitung immer neuer sogenannter Location Based Services. Dies sind Dienste, die dem Nutzer auf Grundlage von Positionsdaten Informationen generieren können. Für Anwendungen im Freien haben sich satellitengestützte Lokalisierungssysteme, welche theoretisch hohe Genauigkeit bieten können, etabliert. Als bekanntes Beispiel sei hier das *GPS* genannt, welches zivil und militärisch genutzt wird. Allerdings ergeben sich auch Anwendungsumgebungen in denen derartige Systeme gar nicht bzw. nur ungenau funktionieren oder z.B. aus Kostengründen bewusst gemieden werden. Dies sind typischerweise Umgebungen, in denen die Satellitensignale zu stark gedämpft werden oder vor allem durch Reflexionen bedingte Laufzeitverschiebungen sich negativ auf die Genauigkeit auswirken, wie z.B.:

- innerhalb von Gebäuden ("indoor")
- im Untergrund (Tunnel, Höhlen u.ä.)
- im Bereich dicht bebauter urbaner Gebiete

Um in solchen Umgebungen dennoch Lokalisierung zu ermöglichen, wurden und werden viele theoretische Konzepte und konkrete Systeme entwickelt. In der Arbeitsgruppe Computer Systems and Telematics (CST) an der *FU-Berlin*, wurde dem Problem der indoor Lokalisierung mit der Entwicklung eines WSN basierten Systems begegnet. Die dabei verwendete Infrastruktur und Lokalisierungsmethoden zielen darauf ab, die Umgebung in welcher sich lokalisiert wird, nicht vorher mit statischer Infrastruktur bestücken zu müssen. Das macht das System für spontane oder kostenkritische Lokalisierungsszenarien attraktriv wie beispielsweise für:

- Rettungseinsätze
- Forschungsmissionen
- militärische Einsätze

So entstand vorausgehende Forschung der Arbeitsgruppe in diesem Bereich u.a. in Kooperation mit der Berliner Feuerwehr. Die Sensorknoten des WSN sind in mobile Knoten und Ankerknoten aufgeteilt. Das Ziel ist es, mit einem mobilen Knoten eine Lokalisierung durchzuführen. Die Grundlage dafür bieten die Ankerknoten, deren Positionen bekannt sind. Die mobilen Knoten ermitteln dann direkt mittels der Ankerknoten oder indirekt über andere mobile Knoten ihre Position. Die Lokalisierung kann dabei mithilfe verschiedener Algorithmen erfolgen. Da diese Algorithmen untereinander und in Abhängigkeit verschiedener Umgebungseigenschaften mit unterschiedlicher Präzision lokalisieren, ist es notwendig Vergleiche zwischen ihnen anzustellen. Ein Weg dies zu tun sind Simulationen der verschiedenen Algorithmen anzustellen. Simulationen können kostengünstig und schnelle Resultate liefern. Der Nachteil ist allerdings, dass sie stets von realen Bedingungen abstrahieren. So kann es sein, dass eine durch Abstraktion ausgeblendete Eigenschaft einer realen Umgebung, direkt oder indirekt, die durch Simulation erwarteten Resultate verzerrt. Daher entstand der Bedarf auch praktische Tests in einer indoor Umgebung durchzuführen und es wurde die Aufgabe gestellt, ein Referenzsystem aufzubauen. Ein solches Referenzsystem soll in einer indoor Umgebung präzise Lokalisierungen durchführen, welche als Referenzwerte für Lokalisierungen der mobilen Sensorknoten dienen. Das Referenzsystem ist mithilfe eines mobilen Roboters realisiert worden, welcher durch Abfahren einer Strecke und stets wiederholende Lokalisierung den gefahrenen Pfad aufzeichnet. An ihm angebrachte mobile Sensorknoten führen ebenfalls wiederholt Lokalisierungen durch. Im Rahmen dieser Arbeit und im Kontext des Referenzsystem wurde eine Software, genannt Pathcompare, entwickelt. Diese ermöglicht es, Pfade des Referenzsystems und die der Sensorknoten, während eines Tests, zusammenzuführen, um einem Tester eine Übersicht über die erreichte Präzision der Lokalisierungen von Sensorknoten gegenüber dem Referenzpfad aufzuzeigen. Pathcompare bietet dazu eine übersichtliche GUI und kann außerdem durch Plug-ins in seiner Funktionalität erweitert werden.

1.2 Struktur der Arbeit

Die Arbeit ist in mehrere Abschnitte gegliedert, die unterschiedliche Aspekte von Pathcompare erläutern. Zunächst wird im unmittelbar folgenden Abschnitt “Aufgabenstellung” die Zielsetzung von Pathcompare und der gewählte Lösungsansatz aufgezeigt. Dazu wird auf die Gesamtstruktur des Referenzsystems eingegangen und der mobile Roboter genauer beleuchtet. Danach folgt der Abschnitt “Implementierung”. Hier wird im Detail auf Pathcompares Bestandteile eingegangen. Dabei werden deren Funktionsweisen und Designentscheidungen betrachtet. Ein wesentliches Element dieses Abschnitts ist außerdem die Betrachtung von Pathcompares Plug-in Konzept und den Voraussetzungen zum Schreiben eigener Pathcompare Plug-ins. Im Abschnitt “Anwendung” werden durchgeführte Tests von einzelnen Komponenten Pathcompares betrachtet und nötige Voraussetzungen zum erfolgreichen Ausführen der Anwendung aufgezeigt. Das “Fazit” fasst die wesentlichen Eigenschaften von Pathcompare zusammen und verweist auf mögliche Verbesserungen und noch offene Fragestellungen.

1.3 Aufgabenstellung

Wie in der Einleitung erwähnt, behandelt diese Arbeit die Entwicklung einer Software, welche bei einem praktischen Test der Sensorknoten in Verbindung mit dem Referenzsystem entstandene Pfaddaten zusammenführt und für den Nutzer aufwertet. Die Software trägt den Namen Pathcompare. Im folgenden wird die Abgrenzung von Pathcompare zum Referenzsystem beschrieben. Dazu wird zunächst der Aufbau und die grundlegende Funktionsweise des Referenzsystems erläutert.

1.3.1 Referenzsystem

Im Referenzsystem steht auf Ebene der Hardware ein mobiler Roboter. Dieser hat das Ziel, sich genau zu lokalisieren, um Referenzwerte für die montierten, mobilen Sensorknoten zu liefern. Da er sich während der Testfahrten indoor bewegt, kann er für seine Lokalisierung keine satellitengestützten Lokalisierungssysteme wie GPS verwenden, aus den in der Einleitung genannten Gründen.

Zwei weitere Ansätze die in Betracht gezogen wurden, um die Bewegung des Roboters nachzuvollziehen und somit Positionsdaten zu gewinnen, sind:

- Inertial Navigation
- Odometrie

Bei der inertial Navigation wird mithilfe von Beschleunigungs- und Gyromessungen auf die ausgeführte Bewegung geschlossen. Diese Messungen lassen sich durch Microelectromechanical systems (MEMS), die in einer Inertial Measurement Unit (IMU) zusammengefasst werden, durchführen. MEMS werden in großen Stückzahlen produziert und sind kostengünstig. Typischerweise sind aber die Messungen, auch bei Stillstand, durch Jitter belastet. Dieser kann zwar durch geeignete Filter geglättet werden, lässt sich allerdings nicht ganz ausschließen. Auf längere Strecken entsteht durch die Aufsummierung der Fehler eine zunehmende Differenz zwischen geschätzer und tatsächlicher Position. Dieser Effekt wird als Drift bezeichnet.

Bei der Odometrie werden die Antriebsdaten ausgewertet, um auf die Bewegung des Roboters zu schließen. Geht man davon aus, dass der Untergrund auf dem der Roboter fährt für dessen Räder geeignet ist und die Räder nicht wegen beispielsweise mangelnder Bodenhaftung stark durchdrehen, so kann sie auf kurzen Strecken sehr genaue Abschätzungen liefern. Allerdings ist auch eine Odometriemessung stets mit einem Fehler behaftet. Dieser Fehler summiert sich über die Zeit auf und die geschätzte Position weicht immer weiter von der tatsächlichen ab. Man hat also auf längeren Strecken ebenfalls mit einem Drift zu rechnen.

Beide Methoden haben gemeinsam, dass sie unabhängig von Informationen aus der Umgebung des Roboters arbeiten. Somit können sie allerdings den beschriebenen Drift in der Lokalisierung niemals korrigieren, da sie nicht die Positions auf Plausibilität mit der Umgebung abgleichen. Für das Referenzsystem ist Drift aber nicht akzeptabel. Aus diesem Grund erfasst der Roboter Abstände zu Hindernissen seiner Umgebung mithilfe einer *Microsoft Kinect*. Die Kinect erstellt mithilfe eines, im infrarot Bereich gestrahltem, optischen Musters ein Tiefenbild. Die Reichweite liegt dabei bei maximal 10 Meter Entfernung bei einem Blickwinkel von ca. 59°. Tests haben gezeigt, dass die Genauigkeit der Tiefenmessung mit

zunehmender Entfernung abnimmt. Im Nahbereich von zwei Metern ermöglicht die Kinect eine überraschend präzise Distanzmessung. Die Genauigkeit liegt dabei im Zentimeterbereich. Außerdem verfügt der Roboter über eine Karte der Testumgebung. Diese Karte in Kombination mit der *Microsoft Kinect* ermöglicht während einer Testfahrt eine Lokalisierung durch folgende grundlegende Schritte durchzuführen:

1. Abschätzung der derzeitigen Position und Ausrichtung durch Odometrie
2. Abgleich mit Karte und Korrektur der Pose

Zum Abtasten der Umgebung hätte alternativ auch ein Laserscanner gewählt werden können, welcher hohe Reichweite mit hoher Genauigkeit kombiniert. Dies wäre aber mit hohen Anschaffungskosten verbunden gewesen. Mit dem Ziel ein günstiges Referenzsystem zu schaffen, wurde schlussendlich ein sogenannter *TurtleBot* gebaut. Dies ist ein von *WillowGarage* spezifizierter low-cost Roboter. Im Kern besteht dieser aus einem *Roomba* Staubsaugerroboter von *iRobot*, einer *Microsoft Kinect* und einem Tragegerüst. Das Tragegerüst dient als Abstellfläche für einen Laptop und bietet im Anwendungsfall des Referenzsystems Platz zum Montieren der Sensorknoten.

Details zum Roboter und damit verbundener Software werden innerhalb einer separaten Bachelorarbeit, ebenfalls im Rahmen der Entwicklung des Referenzsystems, ausführlich erarbeitet.

Zusammenfassend kann man bezüglich des Referenzsystems feststellen, dass zum autonomen Fahren Programme des ROS genutzt werden. Außerdem wurde eine Software implementiert, die den Roboter vorgegebene Wegpunkte abfahren lässt und dabei gesammelte Lokalisierungsdaten innerhalb von ROS bereitstellt. Genauer wird im Kapitel Implementierung auf die Funktionsweise von ROS eingegangen, da Pathcompare mit ROS interagiert.

1.3.2 Ziele Pathcompares

Bei der Entwicklung von Pathcompare wurden folgende Ziele verfolgt:

1. Zusammenführen der Pfaddaten an einem Ort
2. Abweichung zum Referenzpfad bestimmen
3. Anpassungsfähig und erweiterbar

Der erste Punkt bezieht sich auf die durch wiederholte Lokalisierung des Roboters und der Sensorknoten anfallenden Pfaddaten. Ziel ist es, diese bei Durchführung eines Tests zu vereinen und einem Tester Informationen bezüglich der Daten zu visualisieren. Unter dem zweiten Punkt wird die wichtigste dabei ermittelte Information angeführt, nämlich die Bestimmung des Abstands eines Pfads zum Referenzpfad des Roboters. Zusätzlich, soll wie unter Punkt drei aufgeführt, Pathcompare anpassbar und erweiterbar für geänderte Testanforderungen sein. Wie die Ziele realisiert wurden, wird im Abschnitt Implementierung genauer beleuchtet.

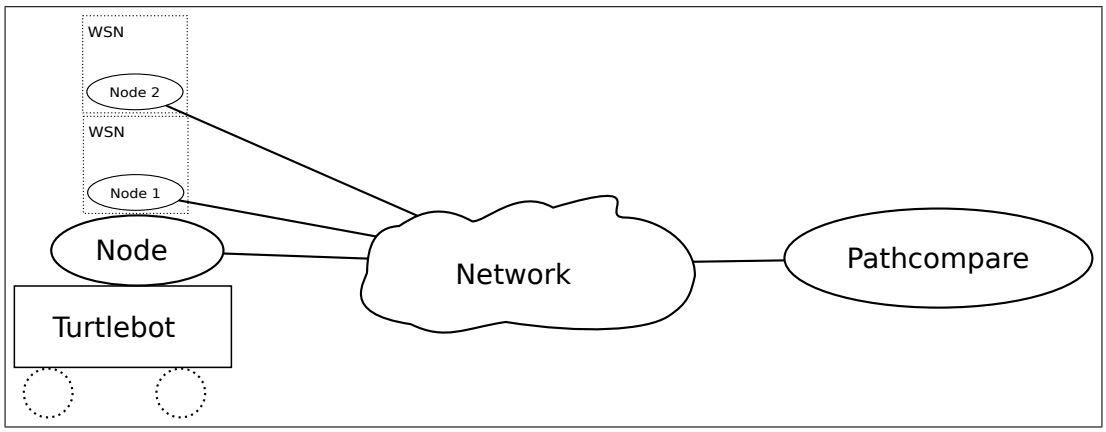

Abbildung 1.1: Struktur des Referenzsystems während der Testausführung

1.3.3 Überblick Testdurchführung

In Abbildung 1.1 wird ein Überblick auf die Komponenten gegeben, die an einer Testdurchführung beteiligt sind und wie diese miteinander in Beziehung stehen.

Erkennbar sind dabei zum einen die mobilen Sensorknoten, welche in das WSN integriert sind. Diese sind am *TurtleBot* Roboter befestigt. Die Sensorknoten und der Roboter sind dabei mit einer als "node" bezeichneten Komponente versehen. Diese dient zur Anbindung an das ROS. Während der Roboter direkt druch ROS betrieben wird, sind für die mobilen Sensorknoten zusätzliche Softwareadapter nötig, welche die Anbindung zu ROS vornehmen. Die Verbindung der Komponenten erfolgt dabei über ein Netzwerk, wobei der Roboter nach außen über WLAN kommuniziert, um ihn nicht in seiner Mobilität einzuschränken.

KAPITEL 2

Implementierung - Pathcompare

2.1 Technischer Rahmen

2.1.1 Robot Operating System - ROS

Innerhalb des Referenzsystems wird ROS im Zusammenhang mit der Steuerung des Roboters genutzt und um die während einer Testfahrt gewonnenen Pfaddaten zu übermitteln. Im Folgenden wird genauer auf die Fähigkeiten und Ziele von ROS eingegangen.

Obwohl der Name zunächst anderes vermuten lässt, ist ROS kein Betriebssystem im klassischem Sinne. Es ist vielmehr eine Sammlung von Anwendungen, welche auf ein Betriebssystem angewiesen sind, um ausgeführt werden zu können. ROS bietet aber Funktionalitäten, die abstrahiert betrachtet Betriebssystemfunktionen ähneln. Charakteristisch ist hierbei ROS's Fähigkeit lokal oder nichtlokal ausgeführte Programme miteinander zu verbinden und eine strukturierte Kommunikation zwischen diesen zu ermöglichen. Die grundlegenden Ansätze von ROS sind:

- multi-tool Ansatz
- verteiltes Rechnen
- peer-to-peer Kommunikation
- keine feste Bindung an eine Programmiersprache
- frei und Open-Source

Multi-tool Ansatz bedeutet, dass ROS die Fähigkeiten verschiedener Programme und Libraries zur Verfügung stellt. Diese sind jedoch nicht fest in den Kern von ROS eingebaut, sondern modular integriert. Als analoges Beispiel in Hinblick auf Betriebssysteme kann man ROS diesbezüglich mit einen Mikrokernel vergleichen. Die Modularität bietet den Vorteil, dass der ROS Kern vergleichsweise klein ist. Außerdem müssen während der Ausführung nur wirklich gebrauchte Tools geladen werden.

Die peer-to-peer Kommunikation bezieht sich auf die Kommunikation zwischen den ROS Modulen, welche durch den ROS Kern gesteuert wird. Der Kern von ROS ist ursprünglich in C++ implementiert. Es existieren jedoch bereits Portierungen in andere Sprachen wie

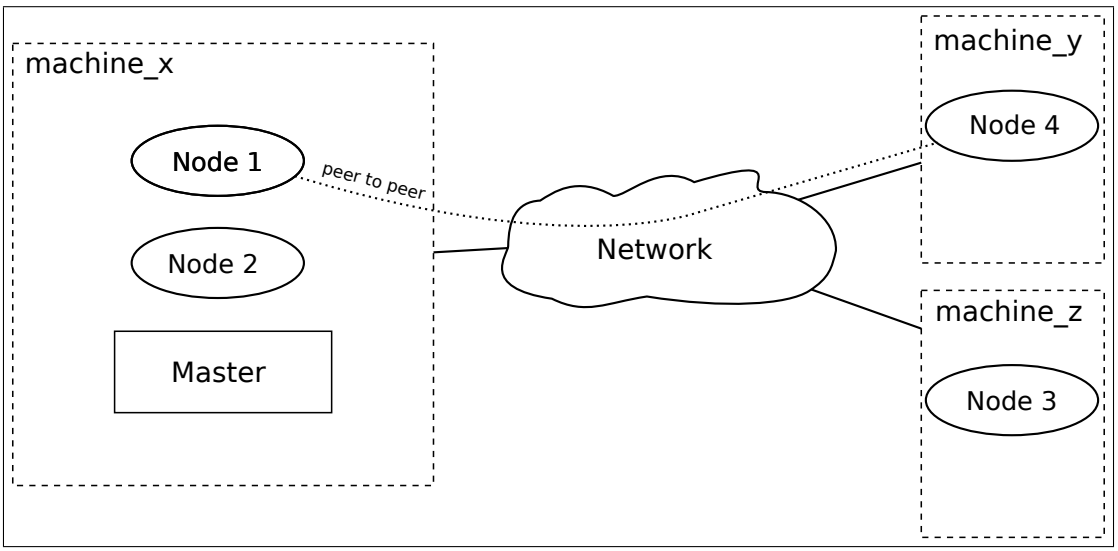

Abbildung 2.1: Beispielhafte Ausführung von ROS auf unterschiedlichen Rechnern

Python, Octave und Lisp, um die ROS-Application Programming Interface (API) einer größeren Zahl von Entwicklern und Projekten zur Verfügung zu stellen.

ROS ist darüber hinaus frei verfügbar und Open-Source. [1]

Man kann beliebige Programme als Module zur Erweiterung von ROS hinzufügen, wie es auch im Rahmen des Referenzsystems geschehen ist. Bei allgemeinem Nutzen und gegebener Pflege der Software besteht die Möglichkeit, dass diese offiziell zu ROS hinzugefügt werden.

Um die konkreten Abläufe und Komponenten innerhalb von ROS veranschaulichen zu können und damit auch den Bezug zu Pathcompare herstellen zu können, ist es zunächst erforderlich die Begrifflichkeiten innerhalb von ROS zu klären. Im Folgendem werden diese aufgezeigt, siehe dazu auch Abbildung 2.1.

Im Zentrum von ROS steht der sogenannte *master*. Dieser wird als einzelne Instanz gestartet und wartet dann darauf, dass sich Tools, die im Kontext von ROS gestartet werden, bei ihm anmelden. Ein als Prozess gestartetes tool wird dabei innerhalb von ROS als *node* bezeichnet. Ist der *master* nicht gestartet, können auch keine *nodes* ausgeführt werden. Die *nodes* sind also alle zunächst auf Kommunikation mit dem *master* angewiesen. Diese Kommunikation kann lokal oder nichtlokal ausgeführt werden, d.h. der *master* kann sich auch auf einem anderen Rechner als der *node* befinden, solange eine http Verbindung zwischen beiden hergestellt werden kann. Das Anmelden des *nodes* beim master erfolgt über einen XML - Remote Procedure Call (XML-RPC) über http. Für den Softwareentwickler auf Anwendungsebene ist diese Kommunikation zur Anmeldung allerdings vollständig durch die ROS API gekapselt und er muss sich in dieser Hinsicht nicht explizit um Verbindungsaufbau oder Nachrichtenaustausch kümmern. Wie in der Abbildung 2.1 dargestellt, können auch die einzelnen nodes auf unterschiedlichen Rechnern ausgeführt werden. Diese zentrale Fähigkeit von ROS lässt sich beispielsweise vorteilhaft ausnutzen durch:

[1] `www.ros.org/wiki/ROS/Installation`

- Verteilung oder Auslagerung rechenintensiver *nodes* auf potente Hardware
- Zusammenführenung von an unterschiedlichen Stellen gewonnenen Daten.

So muss beispielsweise ein mobiler Roboter Bilderkennungsaufgaben nicht selbst ausführen, sondern kann diese an einen *node* weiterleiten der auf einem Rechencluster ausgeführt wird. Der zweite Punkt, also das Zusammenführen von Daten ist besonders im Bezug auf diese Arbeit wichtig, da Pfaddaten des Roboters und der zu testenden Sensorknoten für *Pathcompare* verfügbar gemacht werden müssen. Die Kommunikation zwischen *nodes*erfolgt über sogenannte *messages*. Diese enthalten die serialisierte Form der zu übertragenden Daten. ROS bietet in seinen Kernpaketen bereits zahlreiche Definitionen für unterschiedliche *message* Typen, aber es ist auch möglich eigene zu generieren. Dies wird von zahlreichen Paketen getan, um Daten maßgeschneidert übertragen zu können. Einmal definierte *message* Typen können wiederum rekursiv in anderen *message* Typen eingebettet werden. Ein Beispiel für eine message ist in Quellcode 2.1 dargestellt. Dieser message Typ ist standardmäßig in ROS definiert. Sie besteht wie erkennbar aus den zwei Typen *Vector3* und *Quaternion*. Letzterer beschreibt die Rotation und Vector3 die Translation. Zusammengefügt ergibt dies eine Transformationsnachricht.

```
1 geometry_msgs/Vector3 translation
2   float64 x
3   float64 y
4   float64 z
5 geometry_msgs/Quaternion rotation
6   float64 x
7   float64 y
8   float64 z
9   float64 w
```

Quellcode 2.1: ROS transformation message

Soll ein *node* seine *messages* anderen *nodes* senden können, so muss er dies zunächst durch festlegen einer sogenannten *topic* beim *master* anmelden. Über den *master* wird dadurch diese *topic* für andere *nodes* im ROS sichtbar. Eine *topic* ist definiert durch eine sogenannte *topic-id*, einem eindeutigen String. Dieser ist vergleichbar mit einer URL und dient zur Identifikation innerhalb des ROS. Eine weitere festgelegte Eigenschaft einer *topic* ist der Typ der über sie versendeten *messages*. *Nodes* welche *messages* einer *topic* empfangen sollen, müssen diese *topic* dann beim *master* abonnieren. Der *master* vermittelt dann eine peer-to-peer Verbindung zwischen dem Anbieter node der topic und dem Abonnenten. Generell gilt dabei, dass topics einen unidirektionalen Kommunikationsweg darstellen. Nur der Anbieter versendet messages. Der oder die Abonnenten der topic sind reine Empfänger. In programmatischer Hinsicht wird beim Empfang neuer Nachrichten innerhalb des *nodes* eine festgelegte *callback* Methode aufgerufen, um die *messages* zu bearbeiten. Treffen dabei messages mit einer höheren Frequenz ein als abgearbeitet werden können, so kommt es irgendwann zu Verlusten, wenn die Größe der *message* Queue beim Empfänger überschritten wird. Die Größe der Queue kann jedoch durch die ROS API gesteuert werden.

Zwei weitere wichtige Begriffe in ROS betreffen die Organisierung der Dateien, die zu den einzelnen Tools gehören. Dies sind im einzelnen:

- package
- stack

Ein *package* beinhaltet den Code, Libraries sowie die ausführbare Datei eines Tools bzw. *nodes*. In ROS sind für packages bestimmte Ordnerstrukturen und Dateien festgelegt, sodass mithilfe der von ROS mitgebrachten Tools packages leicht gebaut, gesucht und gestartet werden können. Beispielsweise basiert das ROS build System auf cmake und so ist eine vorkonfigurierte cmake Buildatei, genannt *CMakeLists.txt*, in jedem package grundsätzlich enthalten. Eine Zusammenfassung mehrerer packages wird als *stack* bezeichnet.

Überträgt man die vorgestellten ROS Begriffe auf Pathcompare so ist dieses tool, während der Ausführung, ein einzelner *node*, welcher *topics* abonnieren kann, um *messages* zu empfangen. Es wird im Teil Implementierung darauf eingegangen, welche *messages* das genau sind. Alle zum Kompilieren und Ausführen nötige Dateien sind dabei in zwei *packages* namens *pathcompare* und *pathcompareplugins* aufgeteilt.

2.1.2 Qt

Pathcompare ist darauf ausgelegt alle Informationen für den Nutzer in einer Benutzeroberfläche ,fortan als GUI bezeichnet, zu visualisieren. Da die Anbindung an ROS über C++ erfolgt, lag es nahe, auch die GUI in C++ umzusetzen. Dazu wurde das Qt Framework gewählt. Qt ist in C++ implementiert, wobei allerdings auch Anbindungen für zahlreiche andere Sprachen wie z.B. Java, C#, Ruby oder Python existieren. Die Entwicklung von Qt begann 1991 und ist zum Zeitpunkt des Schreibens in der Version 4.7.4 verfügbar. Das Framework besteht dabei mittlerweile nicht mehr nur aus reinen GUI Bibliotheken, sondern stellt auch Netzwerk-, SQL- und andere Anwendungs-Bibliothken zur Verfügung.

Ein entscheidender Vorteil des Qt Frameworks ist die gebotene große Plattformunabhängigkeit. Gleichzeitig wird für alle unterstützten Betriebssysteme, deren nativer Look&Feel verwendet. Mit nativem Look&Feel ist gemeint, dass das Erscheinungsbild von GUI Elementen der Qt Anwendung dem Erscheinungsbild von GUI Elementen des Betriebssystems gleicht. Für die Entwicklung von *Pathcompare* waren neben den GUI Bibliotheken folgende Konzepte und Funktionen beim Entwickeln von Nutzen:

- Signal-Slot Konzept
- Plattformunabhängigkeit
- graphischer GUI Designer

Auf einige dieser Punkte und deren Bezug zu Pathcompare wird nun kurz eingegangen.

Das *Signal-Slot* Konzept dient dazu, bestimmte Objektveränderungen beobachtenden Objekten mitzuteilen. Es realisiert also das Entwurfsmuster des *Observer patterns*. *Signal-Slot* erspart es dem Programmierer einen Verweis auf das beobachtende Objekt, durch einen registrierenden Methodenaufruf beim aktualisierenden Objekt, zu hinterlegen. Das erleichtert den Entwicklungsprozess, da nicht mehr explizite Methoden zum Registrieren von Beobachtern definiert werden müssen. Stattdessen emittiert, im Falle einer mitzuteilenden Aktualisierung, das aktualisierende Objekt ein sogenanntes Signal, welches durch einen Qt Makro als *signal* ausgezeichnet ist. Bei Beobachter Objekten, die auf dieses Signal reagieren sollen, wird dann eine als *slot* deklarierte Methode selbständig aufgerufen. Vorausgesetzt die

Objekte wurden durch einen vorhergegangenen Aufruf, einer von Qt bereitgestellten, statischen connect Methode, verbunden. Wie bereits angesprochen erhöht dieses Konzept, durch Wegfall der manuellen Implementierung von Beobachter Registriermethoden die Flexibilität für den Entwickler deutlich. Allerdings ist der durch Qt gekapselte Code, welcher durch Auflösung der Makros und Kompilierung entsteht, aufblähend und dadurch langsamer bei der Ausführung als eine direkte Implementierung mit Beobachter Registriermethoden. Im Anwendungsbereich von Pathcompare ist diese Verlangsamung aber unmerklich und wiegt nicht die Vorteile für den Entwickler auf. Deshalb wurde das Konzept auch bei der Entwicklung von *Pathcompare* eingesetzt.

Im Zusammenhang mit dem Signal-Slot Konzepts wurde bereits erwähnt, dass Qt C++ um verschiedenen Makros erweitert. Diese Makros werden dabei nicht immer direkt in gültigen C++ Code übersetzt, sondern dienen als Annotationen. Dies hat Folgen für den Build Vorgang, denn die mit Annotationen versehenen Klassen müssen zuerst mit dem von Qt bereitgestellten Meta Object Compiler (MOC) übersetzt werden. Durch den MOC wird aus den mit Annotationen versehenem C++ Code, erneut C++ Code erzeugt, der anschließend durch weitere Compiler übersetzt werden kann. Standardmäßig wird in *Qt* das Build Programm qmake verwendet, welches die nötigen Aufrufe des MOC automatisch veranlasst. In Hinblick auf *Pathcompare* war aber eine Integration in das von ROS genutzte Build verfahren cmake notwendig. Hierbei muss beachtet werden, dass cmake die Dateien, welche einen MOC Aufruf notwendig machen, derzeit nicht selbständig erkennt. Diese müssen manuell in der Buildkonfigurationsdatei, genannt *CMakeLists.txt*, deklariert werden. Eine solche CMakeLists.txt Datei ist in jedem ROS package vorhanden und muss für dieses angepasst werden. Die sonstige Einbindung *Qt*-spezifischer Libraries und deren Verlinkung mit der Applikation ist in cmake in wenigen Zeilen deklariert. Somit war insgesamt die Verwendung von Qt in *Pathcompare* kein Hindernis für die Verwendung von cmake. Die Nutzung von cmake wird darüber hinaus, für große Projekte, von der Qt Dokumentation selbst empfohlen. [2]

Neben dem MOC existiert noch ein sogenannter User Interface Compiler (UIC). Dieser tritt im Zusammenhang mit dem graphischen GUI Designer in Erscheinung. Beim Erstellen einer GUI mit dem sogenannten QtDesigner wird eine XML Datei erstellt, welche den Aufbau der GUI abbildet. Der UIC ist dann dafür verantwortlich, diese Datei in C++ Klassen zu übersetzen. Auch hier muss cmake veranlasst werden zunächst für einen UIC Aufruf entsprechender Dateien zu sorgen.

2.2 Design der Software

Wie bereits im Abschnitt "Aufgabenstellung" erwähnt, lag der Fokus beim Design der Pathcompare GUI darauf, einfache Benutzung und übersichtliche Darstellung der Informationen für den Nutzer zu gewährleisten. Außerdem sollte die Software leicht erweiterbar sein, um flexibel auf neue Testbedingungen oder geänderte Anforderungen reagieren zu können. Im folgenden Abschnitt wird beschrieben wie Pathcompare diese Ziele realisiert.

[2] `http://developer.qt.nokia.com/quarterly/view/using_cmake_to_build_qt_projects`

2.2.1 Überblick Gesamtsystem

Im Folgenden wird die Software in einer Gesamtansicht dargestellt und anschließend auf die Funktionsweise ihrer Einzelkomponenten genauer eingegangen.

Wie erwähnt ist *Pathcompare* durch Plug-ins erweiterbar. Dadurch kann man zunächst das Gesamtsystem in zwei Teile untergliedern:

1. Rahmen
2. Plug-ins

Rahmen und Plug-ins sind dabei in zwei unterschiedlichen ROS packages organisiert. Das package *pathcompare* beinhaltet alle den Rahmen betreffenden Dateien und das package *pathcompareplugins* beinhaltet die Plug-ins. Die Aufteilung spiegelt sich auch in der GUI wider. Hierbei bildet die GUI des Rahmens die Grundlage für die Visualisierung von Plug-ins. Dabei wird den Plug-ins, nachdem sie geladen wurden, ein Platz innerhalb des Rahmens zugewiesen. Innerhalb dieses zugewiesenen Platzes können sie beliebige eigene GUI Elemente laden und haben die volle Kontrolle über diese. Insgesamt gehören die drei in Abbildung 2.2 abgebildeten GUI Elemente zum Rahmen, dies sind:

- Hauptfenster
- Topic Übersicht
- Plug-in Tab-Fenster

Das Hauptfenster dient dabei als Grundlage für die Topic Übersicht und das Plug-in Tab-Fenster. Die Topic Übersicht und das Plug-in Tab-Fenster sind vertikal getrennt. Die Topic Übersicht ist dabei ganz links angeordnet.

Rechts neben der Topic Übersicht schließt unmittelbar das Plug-in Tab-Fenster an. Wird ein Plug-in geladen, wird in diesem Tab-Fenster ein neuer Tab angelegt. Die Fläche dieses Tabs wird diesem Plug-in dann zur Verfügung gestellt um seine GUI darauf aufzubauen. Auf Details bezüglich des Ladens der Plug-ins und deren Funktionsweise wird im Abschnitt “Main Compare Plug-in” eingegangen.

Die Topic Übersicht zeigt die momentan im ROS verfügbaren Topics an. Um die Topic Übersicht zu strukturieren sind Topics desselben message Typs in Gruppen zusammengefasst dargestellt. Diese Gruppen werden dann kompakt im Topic Übersicht Fenster angezeigt und können dort durch den Nutzer aufgeklappt werden, um alle *topics* eines message Typs anzuzeigen. Dies ermöglicht es, nur solche Topics zu beobachten, die tatsächlich für den Nutzer relevant sind. Abbildung 2.3 zeigt eine typische Ansicht der Topic Übersicht mit aus- und eingeklappten Topic Gruppen.

Die Platzaufteilung zwischen der Topic Übersicht und des Plug-in Tab-Fensters, innerhalb des Hauptfensters, wurde zugunsten des Plug-in Tab-Fensters gewählt, sodass dieses mehr Platz einnimmt. Dies hat zur Folge, dass beim Vergrößern des Hauptfensters die Fläche des Plus-in Tab-Fensters vertikal und horizontal vergrößert wird. Die eingenommene Fläche der Topic Übersicht wächst allerdings nur vertikal wesentlich an. Diese Designentscheidung begründet sich darin, dass die Plug-ins den wesentlichen Inhalt für den Nutzer präsentieren. Die Topic Übersicht wird hingegen vermutlich nur kurzzeitig betrachtet werden und liegt

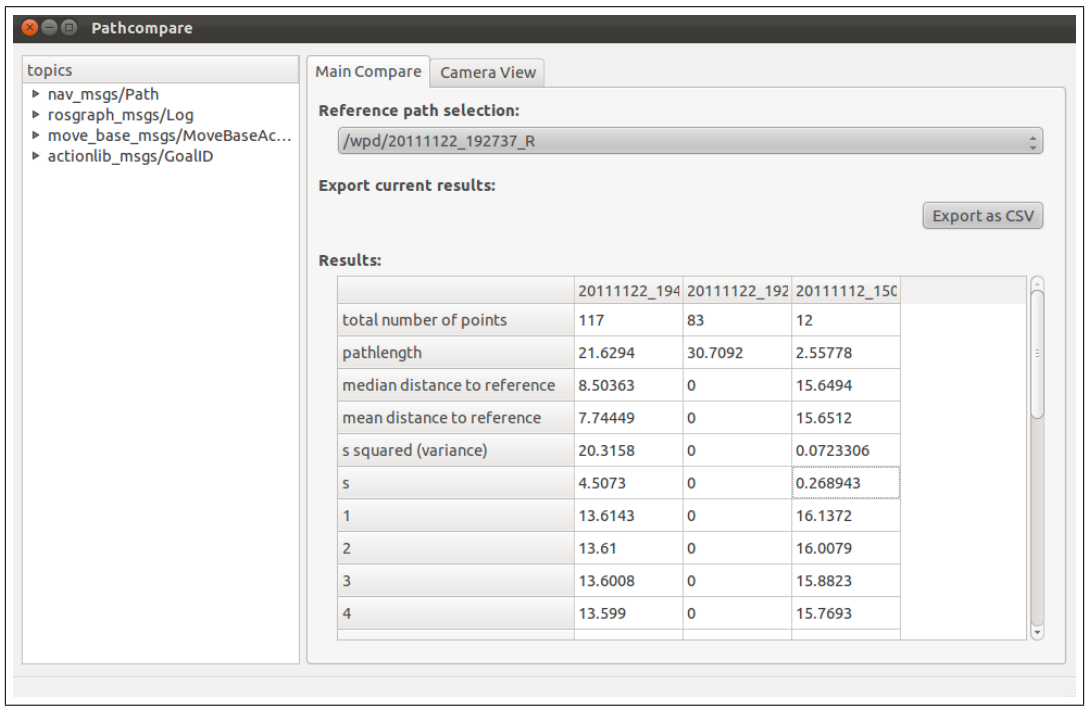

Abbildung 2.2: Hauptansicht von Pathcompare mit geladenen Plug-ins

nicht im Zentrum des Interesses. Darüber hinaus ist die Topic Übersicht sehr kompakt gestaltet und der horizontale Platzbedarf fällt gering aus.

2.2.2 Anbindung an ROS

Die Hauptaufgabe des Rahmens ist es eine Anbindung an ROS zu gewährleisten. Diese Anbindung muss es den Plug-ins ermöglichen, benötigte *topics* zu abonnieren. Für diese Anbindung ist in *Pathcompare* die Klasse *ROSManager* verantwortlich. Im Konstruktor dieser Klasse wird durch entsprechende ROS API Aufrufe der *node pathcompare* erstellt. Dies geschieht durch starten eines separaten Threads, da der ROS Code in einer eigenen Endlosschleife ausgeführt werden muss, die über die gesamte Lebenszeit des *nodes* bearbeitet wird. Innerhalb dieser Schleife wird beispielsweise das Eintreffen neuer *messages* von abonnierten *topics* abgearbeitet. Zugriff der Plug-ins auf ROS spezifische Funktionalität wird komplett über den *ROSManager* gekapselt. Diese Zentralisierung ist nötig, um den Zugriff der verschiedenen Plug-ins auf ROS zu koordinieren. Die zentrale Methode *subscribeToTopic()*, die durch den ROSManager allen Plug-ins zur Verfügung steht, ermöglicht es eine *topic* zu abonnieren. Die zu abonnierende topic wird dabei anhand ihrer topic-id identifiziert.

Beim Abonnieren einer *topic* wird in ROS typischerweise direkt eine Methode als Callback angegeben, welche eingehende *messages* bearbeitet. Beim einfachen Abonnieren ist es dabei zunächst nicht möglich, mehrere Callbacks zu registrieren. Beim einfachen Abonnieren muss außerdem der Message In *Pathcompare* bestehen in dieser Hinsicht allerdings zwei Schwierigkeiten, denen bei der Entwicklung begegnet werden musste:

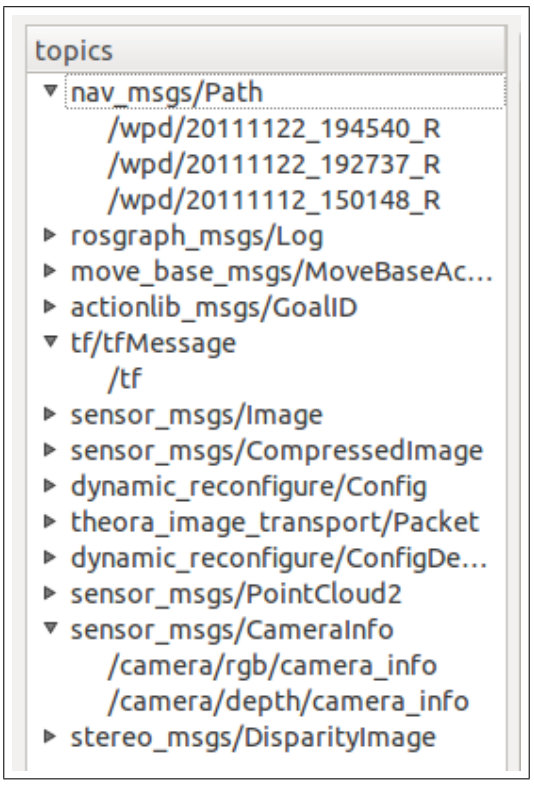

Abbildung 2.3: Topic Übersicht

1. eventuell abonnieren mehrere Plug-ins eine topic
2. topics können verschiedene *message* Typen haben

Das erste Problem wurde durch die Verwendung von *ros::message_filter::cache* gelöst. Objekten dieser Klasse wird bei der Initialisierung eine *topic* zugewiesen. Empfangene *messages* dieser *topic* werden in einem Ringpuffer mit einstellbarer Größe zwischengespeichert. Außerdem erlaubt diese Klasse die Registrierung beliebig vieler Callbacks. Der *ROSManager* sorgt dafür, dass für jede von den Plug-ins benötigte *topic* genau ein *ros::message_filter::cache* Objekt existiert. Plug-ins erhalten dann einen Verweis auf dieses Objekt und können ihren Callback registrieren. Abbildung 2.4 ist ein Beispiel für eine Ausführung von Pathcompare in Verbindung mit drei Plug-ins. Über das ROS werden die drei topics topic_x, topic_y und topic_z angeboten. Durch einen Aufruf von der *subscribeToTopic()* Methode des ROS-Manager soll Plug-in 1 die topic_x abonnieren. Der ROSManager legt aus diesem Grund einen ros::message_filter::cache an und gibt einen Verweis darauf an das Plug-in 1 zurück. Plug-in 2 möchte dieselbe topic abonnieren und erhält ebenfalls einen Verweis auf denselben Cache. Plug-in 3 verlangt jedoch die topic_y zu abonnieren und es erfolgt das anlegen eines neuen ros::message_filter::caches.

Weitere Methoden des ROSManagers erlauben es den Plug-ins, im ROS verfügbare topics zu erfragen. Für eine komplette Methodenübersich sei an dieser Stelle auf die Klassendefinition *rosmanager.h* im package pathcompare verwiesen.

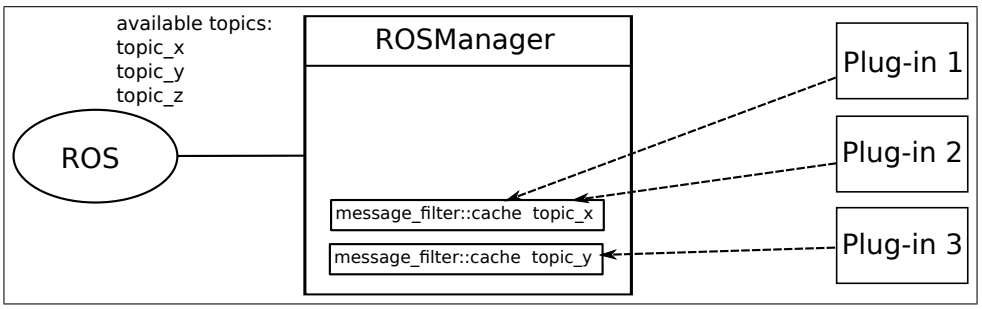

Abbildung 2.4: Anbindung an ROS durch ROSManager

Die Klasse ROSManager sorgt außerdem dafür, dass die Topic Übersicht regelmäßig aufgefrischt wird. Dadurch wird dem Nutzer ersichtlich, wann neue *topics* erscheinen oder nicht mehr verfügbar sind. Da die ROS API keinen Event Service anbietet, um Beobachter zu benachrichtigen falls sich der Status von *topics* ändert, stellt der *ROSManager* sekündlich eine Anfrage an den *master* nach dem aktuellen Stand der topics. Dies ist gesteuert über einen QTimer in Verbindung mit dem Signal-Slot Konzept.

2.2.3 Laden von Plug-ins

Das Einbinden von Plug-ins in den Rahmen steuert die Klasse *PluginLoader*. Sie sucht dabei in einem Verzeichnis nach möglichen Plug-in Dateien. Bei dieser Suche werden generell nur Shared Library Dateien betrachtet. Handelt es sich um eine für *Pathcompare* gültige Plug-in Datei, wird das Plug-in geladen under der *PluginLoader* erfragt den Namen des Plug-ins, über eine im Plug-in Interface spezifizierte Methode *getName()*. Anschließend wird im Plug-in Tab Fenster ein neuer Tab angelegt. Dabei wird im Tab Reiter der erfragte Name des Plug-ins eingetragen. Dadurch sind in der GUI die Plug-ins für den Nutzer leicht zu unterscheiden und können schnell angewählt werden. Der Plug-in Ordner steht während der Laufzeit von *Pathcompare* unter der Beobachtung des *PluginLoaders*. Obwohl *Pathcompare* bereits gestartet wurde, können also Plug-ins in das Plug-in Verzeichnis eingefügt werden und diese werden geladen. Dies bietet mehr Flexibilität für den Nutzer und Entwickler, da kein Neustart der Rahmen Anwendung sowie der ausgeführten Plug-ins erforderlich ist.

2.2.4 Main Compare Plug-in

Main Compare implementiert die eigentliche Hauptfunktionalität, um Pfaddaten des Referenzsystems und der Sensorknoten zu vergleichen und auszuwerten. Es ist dabei der Philosophie von *Pathcompare* folgend als Plug-in implementiert worden, welches beim Starten des Rahmens durch den *PluginLoader* geladen und ausgeführt wird.

In der GUI von Main Compare werden dem Nutzer verschiedene Informationen sowie Einstellungsoptionen bezüglich der empfangenen Pfaddaten angezeigt. Die Funktionen der einzelnen GUI Elemente werden im Folgenden aufgezeigt. Insgesamt gibt es drei Bereiche in der GUI. Die GUI ist in Abbildung 2.2 gezeigt. Die drei Bereiche lassen sich anhand drei verschiedener Labels abgrenzen. Von oben nach untern gesehen sind dies die Bereiche:

1. Reference path selection
2. Export results
3. Results

Der Bereich der "Reference path selection" beinhaltet eine Combo-Box, welche alle über ROS verfügbaren topics des message Typs *nav_ msgs/Path* beinhaltet. Der Nutzer kann dann aus dieser Liste einen Pfad auswählen, der als Referenzpfad genutzt werden soll. Das bedeutet, dass der ausgewählte Pfad als Referenz für die übrigen Pfade gilt. So werden dann die Abstandsberechnungen bezüglich dieses Pfades durchgeführt. Die Auswahl des Referenzpfades kann jederzeit geändert werden und die Abstände werden stets neu berechnet.

Unterhalb des "Reference path selection" befindet sich der "Export results" Bereich. Er dient dem Nutzer dazu, die berechneten und im "Results" Bereich visualisierten Ergebnisse, permanent zu speichern. Die Speicherung erfolgt dabei als CSV Datei. Der Export wird durchgeführt, wenn der Nutzer den in diesem Bereich vorhandenen Button drückt. Das genaue Format der gespeicherten Daten wird später erläutert.

Der "Results" Bereich dient dazu, alle ermittelten Informationen bezüglich der Pfade für den Nutzer anzuzeigen. Die Visualisierung erfolgt hierbei durch eine Tabellenstruktur. In dieser Tabelle wird für jede Path topic ein Spalte angelegt. In jeder Zeile einer Spalte sind dabei die unterschiedlichen Informationen eingetragen. Main Compare verhält sich so, dass topics, welche einmal in der Tabelle erfasst wurden, dort verbleiben. Sollte also eine topic nicht mehr im ROS zur Verfügung stehen, bleiben die angezeigten Informationen auf Basis der zuletzt empfangenen Werte dennoch erhalten.

In der Tabelle werden für den Nutzer folgende Informationen angezeigt:

- Anzahl der Lokalisierungen
- Berechnung der Pfadlänge
- Anzeige des Medians der Abstände
- Anzeige des arithmetischen Mittels der Abstände
- Berechnung der empirische Varianz s^2 der Abstände
- Berechnung der empirische Standardabweichung s der Abstände
- 20 größten Abstände zum Referenzpfad

Die angezeigten Informationen werden beim Empfang neuer Pfad *messages* stets aktualisiert. Generell gilt dabei, dass jede Path message den bisher zurückgelegten Pfad vollständig enthalten muss. Pfade können also nicht stückweise übertragen werden. Dieses vorgehen bietet die Möglichkeit, dass Pfaddaten nachträglich geändert werden können. Also kann beispielsweise die erste Lokalisierung innerhalb eines Pfades zu einem späteren Zeitpunkt abgeändert werden.

Um nachzuvollziehen wie die in Main Compare angezeigten Informationen ermittelt werden, bietet es sich an, zunächst den Typ der *messages*, welche über die Pfad topics empfangen werden, zu analysieren. Wie bereits beschrieben sind die messages vom Typ *nav_ msgs/Path*, der wie folgt definiert ist:

```
1 Header header
2    uint32 seq
```

```
3    time stamp
4    string frame_id
5  geometry_msgs/PoseStamped[] poses
6    Header header
7      uint32 seq
8      time stamp
9      string frame_id
10   geometry_msgs/Pose pose
11     geometry_msgs/Point position
12       float64 x
13       float64 y
14       float64 z
15     geometry_msgs/Quaternion orientation
16       float64 x
17       float64 y
18       float64 z
19       float64 w
```

Quellcode 2.2: ROS Path message

Zunächst ist in Quellcode 2.2 zu erkennen, dass sich dieser *message* Typ aus anderen Typen zusammensetzt. Auf der obersten Ebene ist dies ein *Header* und ein Array von *geometry_ msgs/PoseStamped*. Der Header dient dazu, die Path message von anderen empfangenen Path messages zu unterscheiden und dazu die messages ordnen zu können. Die Ordnung kann dabei über eine Sequenznummer (seq) oder einen Zeitstempel (stamp) hergestellt werden. Das Feld *frame_ id* definiert einen Bezugsrahmen innerhalb von ROS. Es wird aber im Kontext von Main Compare nicht ausgewertet und wird hier deswegen nicht genauer erläutert. Die eigentlichen Pfaddaten verbergen sich in dem Array von PoseStamped. Auch PoseStamped ist ein zusammengesetzter Typ, bestehend aus einem Header und einer Pose. Die Pose setzt sich aus einem Point und einer Quaternion zusammen. Der gefahrene Pfad wird nun dadurch in der Path message abgebildet, indem bei jeder durchgeführten Lokalisierung eine PoseStamped angelegt wird. Diese PoseStamped hält im Header die Zeit der Lokalisierung und eine Sequenznummer fest. In der Pose wird dann der bei der Lokalisierung ermittelte Ort als Point eingetragen. Die Quaternion hält fest, welche Ausrichtung dabei im Raum vorliegt und wird auch in die Pose eingefügt. Die Ausrichtung ist allerdings optional und wird in der derzeitigen Version von Main Compare nicht beachtet.

Außerdem lässt sich in Quellcode 2.2 erkennen, dass die Lokalisierungen dreidimensional erfolgen können. Die von Main Compare ausgeführten Berechnungen sind aber auch mit Daten niederer Dimension möglich, indem die nicht genutzte Dimension auf 0 gesetzt wird. Beispielsweise werden, in der derzeitigen Form des Referenzsystems, vom Roboter zweidimensionale Lokalisierungsdaten in der x-y Ebene übermittelt und die z Komponente erhält immer den Wert 0. Wie schon in Bezug auf die orientation erwähnt, werden nicht alle Komponenten der Pfad message in Main Compare verwendet. Für die derzeitige Version sind folgende Felder relevant:

- poses.stamp (Typ ros::time)
- pose.position.x (Typ float64)

- pose.position.y (Typ float64)
- pose.position.z (Typ float64)

Obwohl nicht alle Felder der Path message genutzt werden, ist dieser message Typ für Main Compare die geeignete Wahl, da er in ROS standardmäßig definiert ist. Dies erspart die Definition und Verteilung eines eigenen message Typs. Außerdem sind die ungenutzten Felder nicht übermäßig groß und werden die Kapazitäten des Übertragungskanals kaum belasten. Vernachlässigt man die ungenutzten Felder des Header, der eine in ROS definierte Standardmessage ist, fallen nur zusätzliche 24 Byte für die ungenutzte orientation pro übertragenem Point an. darüber hinaus ist es denkbar, dass die orientation in späteren Versionen von Main Compare doch noch betrachtet wird und somit die durchzuführenden Änderungen klein ausfallen.

Anhand der oben aufgezählten, genutzten Felder lässt sich ein Pfad abstrakt durch eine Menge M von Tupeln modellieren. Dabei setzen sich die Tupel aus einem Zeitstempel und einem dreidimensionalem Punkt zusammen. In Anlehnung an die verwendeten Bezeichner in Quellcode 2.2, sei M demnach wie folgt definiert:

$$M := \{(p.stamp, p.position) \mid p \in poses\}$$

Die konkrete Repräsentation dieses Modells erfolgt in Main Compare durch die Klassen *TopicPath* und *Position*. TopicPath repräsentiert dabei die Menge M. Position repräsentiert ein einzelnes Tupel und enthält somit den Zeitstempel und zugehörigen Punkt. Dem Modell entsprechend enthält ein TopicPath Objekt eine Liste von Position Objekten. Neben den Klassen TopicPath und Position gibt es noch eine Klasse *TopicPathManager*. Es wird genau ein Objekt dieser Klasse für jede im ROS verfügbare topic mit dem message Typ nav_msgs/Path angelegt. Diese Klasse bietet eine callback Methode, welche beim zum topic gehörenden message_filter::cache registriert wird. Kommt eine neue message an, wird die callback Methode aufgerufen und aus den Werten der Path message ein neues TopicPath Objekt erstellt. Anschließend werden anhand des neuen TopicPaths alle nötigen Berechnungen erneut durchgeführt und die Results Tabelle aktualisiert. Die Beziehung der Klassen Position, TopicPath und TopicPathManager werden in Abbildung 2.5 verdeutlicht.

Anhand der abstrakten Modellierung M eines Pfades, werden im Folgenden, die von Main Compare ausgeführten Berechnungen erläutert. In der konkreten Repräsentation erfolgen alle Berechnungen in der Klasse *TopicPathManager*.

Anzahl der Lokalisierungen

Um die Anzahl der Lokalisierungen zu ermitteln, wird in Main Compare lediglich die Mächtigkeit der Menge M bestimmt. Es gilt also:

$$numberofpoints = |M|$$

Im *TopicPathManager* ist dies entsprechend einfach realisiert.

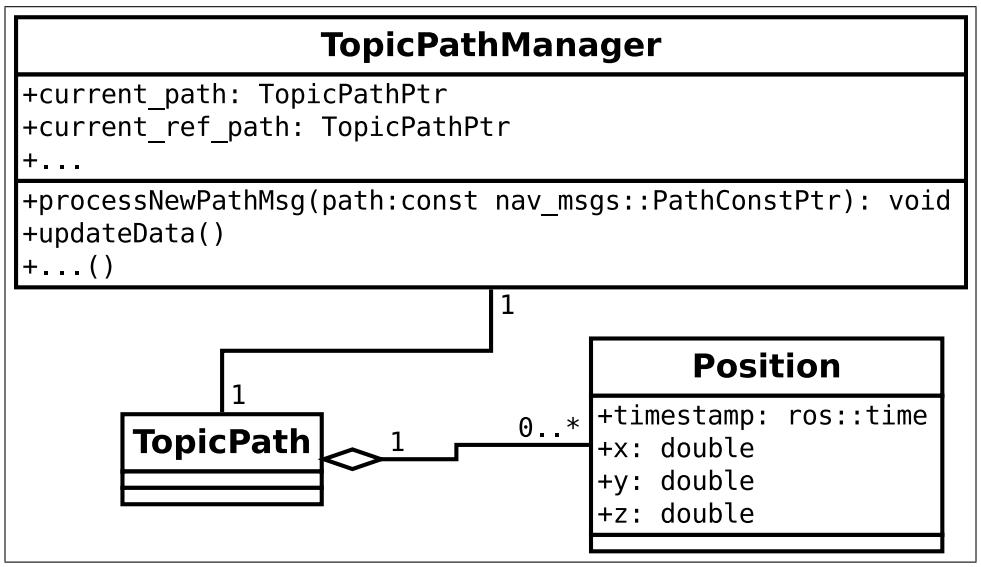

Abbildung 2.5: In Main Compare verwendete Klassen zum verarbeiten von Pfaden

Berechnung der Pfadlänge

Bei der Bestimmung der Pfadlänge werden zunächst alle Abstände zwischen je zwei direkt aufeinanderfolgenden Punkten im Pfad bestimmt. Die Summe all dieser Abstände entspricht dann der Gesamtlänge des Pfades. Dazu muss jedoch noch geklärt werden, wann ein Punkt p_1 in einer Path message auf einen anderen Punkt p_2 folgt. Das lässt sich über den Zeitstempel feststellen und ist anschaulich ausgedrückt dann der Fall, wenn keine weitere Lokalisierung in der Zeit zwischen der Lokalisierung p_1 und p_2 stattgefunden hat. Dies kann man abstrakt und bezogen auf M ausdrücken. Ein Punkt p_2 ist direkt folgend auf einen Punkt p_1, genau dann wenn für die zugehörigen Tupel

$t_1 := (z_1, p_1)$ und $t_2 := (z_2, p_2)$ mit $t_1, t_2 \in M$ gilt, dass:

$$z_2 > z_1 \wedge \nexists (z', p') \in M : z_2 > z' > z_1$$

Der Abstand zweier aufeinanderfolgender Punkte lässt sich einfach durch Vektorsubtraktion und anschließende Betragsbildung des resultierenden Vektors bestimmen.

Pfadvergleichsverfahren und Abstandsberechnung

Das Hauptziel von Main Compare ist es, dem Nutzer zu erleichtern, die Genauigkeit eines durch Lokalisierung gewonnenen Pfades, in Bezug auf einen Referenzpfad, abzuschätzen. Um dies zu tun, ist es notwendig, den Abstand von Lokalisierungen des zu untersuchenden Pfads, zu denen des Referenzpfads, zu ermitteln. In einem naiven Ansatz kann man die Lokalisierungen direkt miteinander vergleichen. Voraussetzung hierfür ist, dass für jeden Punkt des Referenzpfades auch ein Punkt im zu vergleichenden Pfad mit demselben Zeitstempel existiert. Der Vergleich zwischen Referenz- und zu untersuchendem Pfad fällt dann denkbar einfach aus, denn man muss nur den Abstand je zweier Punkte, mit demselben Zeitstempel, bestimmen. Das Problem bei dieser Methode ist jedoch, dass das Referenzsystem

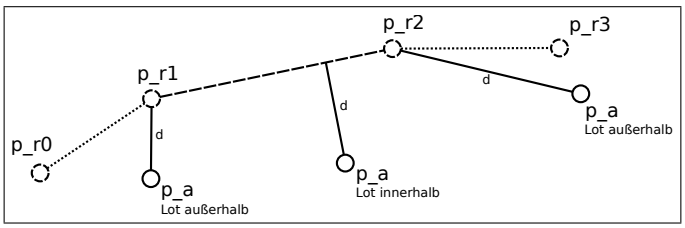

Abbildung 2.6: Bestimmung der Distanz eines Punktes p_a zum Referenzpfad

und die einzelnen Sensorknoten unabhängig voneinander sind und dadurch Lokalisierungen nicht immer zum selben Zeitpunkt oder mit einer anderen Frequenz durchführen können. Um dies zu verhindern, müsste man einen zentralen Taktgeber in das System integrieren, der durch ausgesendete Impulse, zum Beispiel in der Form einer ROS message, den Komponenten vorschreibt, wann eine Lokalisierung durchgeführt werden soll. Die Einführung eines solchen Taktgebers verkompliziert jedoch die Testausführung und setzt bei allen am Test beteiligten Komponenten voraus, dass deren Software auf diesen Mechanismus reagiert. Zudem ist man während des Tests zwingend auf eine ungestörte Netzwerkkommunikation angewiesen, da sonst der Impuls nicht zeitgleich oder gar nicht bei den Komponenten ankommt.

Aufgrund dieser schwerwiegenden Nachteile wurde bei der Entwicklung von Main Compare ein Ansatz gewählt, der es den Komponenten erlaubt Lokalisierungen zu beliebigen Zeitpunkten und mit beliebiger Frequenz durchzuführen. Die einzige Voraussetzung ist dabei, dass die Uhren der Komponenten, mit einem gewissen tolerierbaren Fehler, synchronisiert werden. Dies sichert, dass die Zeitstempel der Lokalisierungen zuverlässig vergleichbar sind. Der Abstand wird wie folgt ermittelt. Der Prozess der Abstandsbestimmung ist auch Abbildung 2.6 dargestellt.

Für jeden Punkt im zu vergleichenden Pfad A wird der Abstand zum Referenzpfad R bestimmt. Dazu wird der jeweilige Zeitstempel z_a eines Punktes p_a aus A einem passenden Zeitstempelintervall zweier aufeinanderfolgender Punkte p_{r2} , p_{r1} aus R zugeordnet, sodass gilt:

$$z_{r1} <= z_a < z_{r2}$$

Ist dieses Intervall gefunden, wird der Abstand von p_a zum Geradensegment $\overline{p_{r2}p_{r1}}$ bestimmt. Um diesen Abstand zu bestimmen, wird das Lot auf die von p_{r1} und p_{r2} bestimmte Gerade ermittelt. Fällt das Lot dabei zwischen p_{r1} und p_{r2} auf die Gerade, wird der Abstand vom Lotpunkt zu p_a errechnet. Fällt er jedoch außerhalb dieses Intervalls, muss der Abstand, je nach Lage des Lotpunkts, entweder zu p_{r1} oder p_{r2} bestimmt werden.

Charakteristisch bei dieser Art der Abstandsbestimmung ist, dass beim Referenzpfad zwischen zwei Lokalisierungen die ausgeführte Bewegung durch eine Gerade interpoliert wird. Es wird also angenommen, dass die tatsächlich ausgeführte Bewegung zwischen den Punkten einer Gerade entspricht. Daraus folgt aber, dass man die Lokalisierungsfrequenz des Referenzpfades an die Bewegungsgeschwindigkeit des Roboters anpassen muss. Dabei soll zwischen zwei Lokalisierungen des Referenzpfades die interpolierte Strecke klein bleiben, denn dadurch nähert sie sich stärker der tatsächlich gefahrenen Strecke an. Betrachtet man das

derzeitige Referenzsystem mit dem TurtleBot, so ist die Bewegungsgeschwindigkeit allerdings gering und eine sekündliche Lokalisierung erscheint ausreichend. Dies muss jedoch noch durch weitergehende Tests bestätigt werden.

Bestimmung empirischer Varianz und Standardabweichung

Auf Basis der zuvor beschriebenen Abstandsberechnung wird in Main Compare die empirische Varianz der Abstände bestimmt. Um diese zu berechnen, wird zunächst das arithmetische Mittel gebildet, welches ebenso in der Results Tabelle angezeigt wird. Anschließend wird die empirische Varianz s^2 gebildet. Die empirische Standardabweichung ist die Wurzel aus s^2 also s.

Bestimmung des Median der Abstände

Der Median der Abstände ist robuster gegenüber Ausreißern als das arithmetische Mittel und wird deshalb zusätzlich der Results Tabelle aufgeführt.

2.2.5 Das Plug-in Konzept

Im folgenden Abschnitt wird das Konzept des Pathcompare Plug-in Mechanismus erläutert und aufgezeigt, welche Schritte erforderlich sind, um eigene Plug-ins für Pathcompare zu erstellen. Realisiert sind die Plug-ins in Pathcompare mithilfe der Qt Plug-in API. Diese gibt Strukturen vor, um Plug-ins zu definieren. Außerdem beinhaltet sie Klassen, um Plug-ins während der Laufzeit einer Applikation zu laden. Bei Verwendung dieser API müssen, um Plug-ins zu erstellen, im wesentlichen zwei Schritte ausgeführt werde, dies sind:

1. Definition eines Interface
2. Implementierung des Interfaces durch Plug-in

Um eine Anwendung durch Plug-ins erweiterbar zu machen, ist also zunächst erforderlich ein Interface zu definieren. Dieses Interface legt fest, welche Methoden durch ein Plug-in zu implementieren sind. Das Interface hat dabei die Form einer Klassendefinition mit nur rein virtuellen Funktionen. Zusätzlich wird diese Klassendefinition durch Qt Makros ergänzt, um es als Interface zu markieren. In Pathcompare ist dieses Interface vorgegeben und trägt den langen Namen *ComparatorPluginFactoryInterface*. Wie die Implementierung dieses Interfaces erfolgt, wird nun anhand eines vorhandenen Plug-ins für Pathcompare erläutert. Dieses Plug-in trägt den Namen Camera View. In der derzeitigen Version von Pathcompare wird es, ebenso wie Main Compare, beim Starten der Anwendung geladen. Es dient dem Nutzer dazu ROS topics des Typs sensor_msgs/Image zu visualisieren. Dieser Typ von Topic wird beispielsweise innerhalb von ROS genutzt um Kamerabilder der Kinect zur Verfügung zu stellen. Camera View könnte also bei einer Fahrt des TurtleBots eingesetzt werden um dem Tester ein Bild der aktuellen Umgebung des Roboters zu vermitteln. Die Gestaltung der GUI dieses Plug-ins fällt einfach aus. Siehe diesbezüglich auch Abbildung 2.7. Der Nutzer kann mit einer Combobox eine topic des Typs sensor_msgs/Image auswählen. Nach der Auswahl, werden die empfangenden Bilder, der gewählten Image topic, eingeblendet.

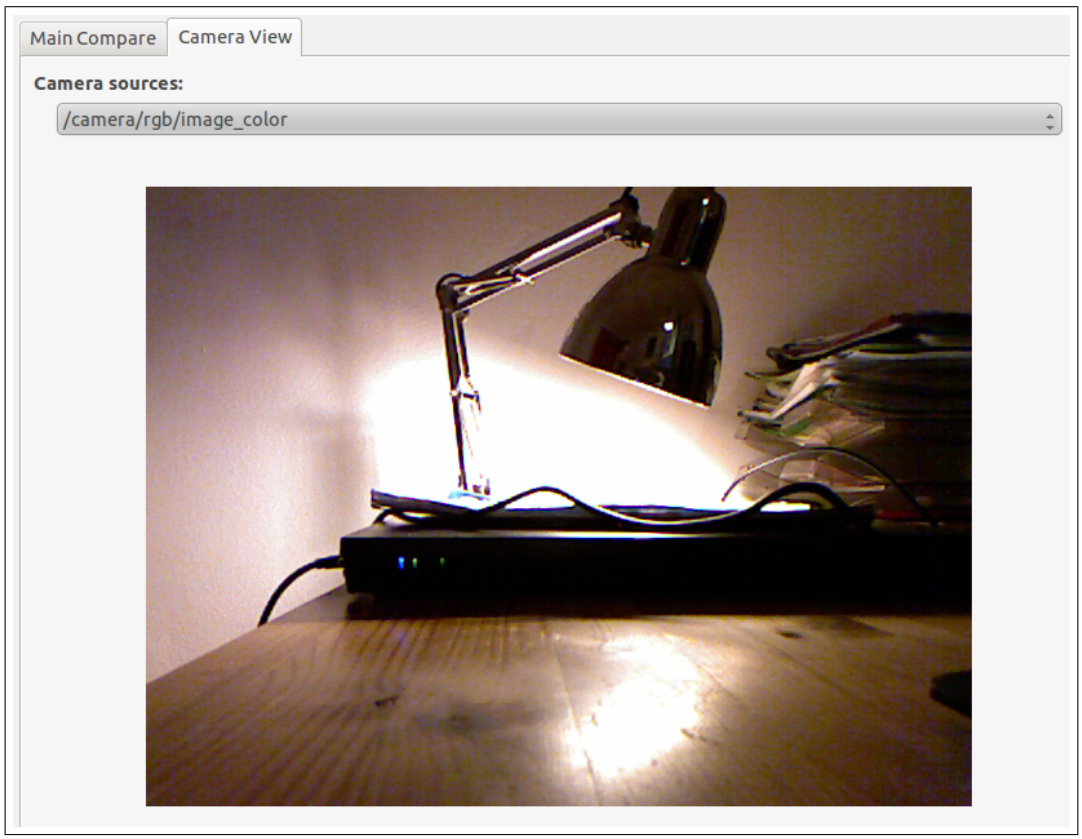

Abbildung 2.7: GUI des Camera View Plug-ins

Wie zuvor beschrieben, muss jedes Pathcompare Plug-in das Interface *ComparatorPlugin-FactoryInterface* implementieren, so auch Camera View.

Dieses Interface beinhaltet zwei Methoden, siehe dazu auch Quellcode 2.3.

```
1
2           virtual ComparatorPluginPtr createComperatorPlugin(ROSManager *
                ros_manager, QWidget *tab_widget) const = 0;
3           virtual QString getPluginName() const = 0;
```

Quellcode 2.3: Interfacemethoden - Plug-ins müssen diese implementieren

Die Methode createComparatorPlugin() ist eine Factorymethode und erzeugt das eigentliche Objekt, welches die Funktionalität des Plug-ins beinhaltet. Dieser Umweg ist notwendig, da Plug-ins keine typischen Objekte sind. Denn bei Qt Plug-ins handelt es sich um shared libraries. In einer shared library können zwar Methoden ausführbar hinterlegt werden, aber kein eigenständiges Objekte mit Attributen existieren. Der Rückgabetyp von createComparatorPlugin() ist ComparatorPluginPtr, definiert durch Quellcode 2.4.

```
1 //typedef for shared_ptr reference to ComparatorPlugin
2 typedef boost::shared_ptr<ComparatorPlugin> ComparatorPluginPtr;
```

Quellcode 2.4: Typdefinition von CompoaratorPluginPtr

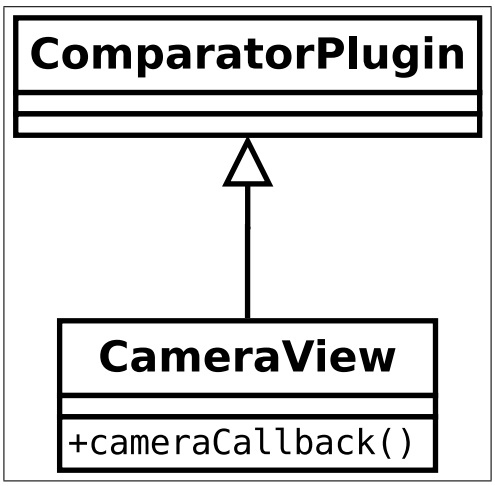

Abbildung 2.8: ComparatorPlugin als Basisklasse zu CameraView

Es handelt sich also um einen Verweis auf ein Objekt der Klasse ComparatorPlugin. Diese Klasse ist der Basistyp, für das in der Factory Methode dynamisch erzeugte Objekt. In Camera View ist die createComparatorPlugin() Factory Methode wie folgt implementiert worden, siehe Quellcode 2.5:

```
1 ComparatorPluginPtr PathcompareMainFactoryPlugin::createComparatorPlugin(
      ROSManager * ros_manager, QWidget *tab_widget) const
2 {
3         ComparatorPluginPtr comp_plugin(static_cast<ComparatorPlugin *>(
              new CameraView(ros_manager, tab_widget)));
4         return comp_plugin;
5 }
```

Quellcode 2.5: Implementierung der createComparatorPlugin in Camera View

Es wird hier ein Objekt des Typs CameraView angelegt, welches den Basistyp ComparatorPlugin besitzt, siehe dazu auch **??**. Im Konstruktor erhält es Verweise auf den ROSManager, welcher benötigt wird um ROS spezifische Aktionen durchzuführen. Als zweites Konstruktorargument wird die Zeichenfläche für die Camera View GUI übergeben. Am Ende der Methode wird das fertig angelegte Objekt zurückgegeben und steht damit Pathcompare zur Verfügung.

Die zweite Methode im Plug-in Interface, getName() liefert lediglich den Namen des Plug-ins zurück und wird in Pathcompare durch die Klasse PluginLoader genutzt.

Zusammenfassend kann man also sagen, dass die eigentlichen Pathcompare Plug-in Interface Methoden ohne großen Aufwand zu implementieren sind. Ein vollständiges Beispiel zur Implementierung neuer Plug-ins ist durch Camera View gegeben. Zu finden sind dabei alle relevanten Code- und Builddateien im package pathcompareplugins/cameraview.

KAPITEL 3

Anwendung

In diesem Abschnitt soll ein typischer Anwendugsfall für Pathcompare und dem Main Compare Plug-in beschrieben werden.

3.1 Ausführung Pathcompare

Pathcompare ist in die ROS packages pathcompare und pathcompareplugins aufgeteilt. Die ausführbare Datei der Pathcompare Hauptanwendung befindet sich im package pathcompare. Im package pathcompareplugins befinden sich die Pathcompare Plug-ins. Diese werden, wie im vorigen Abschnitten detailiert beschrieben, zur Laufzeit dynamisch durch die Hauptanwendung eingebunden.

Ist ein ROS master gestartet, kann Pathcompare durch folgenden den ROS Befehl ausgeführt werden.

```
1 rosrun pathcompare pathcompare
```

Quellcode 3.1: starting pathcompare

Dieser Befehl sucht dabei automatisch die ausführbare Dabei in der pathcompare Ordnerstruktur. Alternativ kann diese aber auch manuell ausgeführt werden. Dies setzt natürlich voraus, dass Pathcompare vorher erfolgreich kompiliert wurde. Für eine genaue Installationsanleitung siehe dazu den Anhang Installation oder die in den packages enthaltenen README.txt Dateien.

3.2 Ausführung des Masters

Test haben gezeigt, dass als Ausführungsort für den ROS master sich dieselbe Maschine anbietet, auf der auch Pathcompare läuft. Der Grund dafür ist, dass Pathcompare häufige Anfragen an dem master, zum auffrischen der Topic Übersicht, stellt. Läuft der master nicht lokal, sollte zumindest für eine Netzwerkanbindung mit möglichst niedriger Latenz gesorgt werden. Es kann ansonsten dazu kommen, dass Teile der GUI blockiert sind. Es

wurde versucht diesem Problem durch caching entgegenzusteuern, indem stets eine Liste der aktuellen topics lokal im Programm vorgehalten wird. Die Liste soll dabei in einem separaten Thread regelmäßig aktualisiert werden werden. Allerdings reichte die Entwicklungszeit nicht aus um diese Änderung in die derzeitige Version fehlerfrei einzuarbeiten und ist deshalb noch nicht integriert. Siehe dazu auch Kapitel Fazit.

3.3 Konfiguration der Sensorknoten und Roboters

Wie im Kapitel Implementierung beschrieben wertet das Plug-in Main Compare, nav_msgs::path messages aus. Diese werden von den Sensorknoten und dem Roboter bereitgestellt. Wann diese messages nach dem Start von Pathcompare gesendet werden kann beliebig bestimmt werden. Optimal ist es jedoch, wenn der Roboter und die Sensorknoten bereits während der Fahrt in regelmäßigen Abständen aktualisierte Path messages versenden. Dadurch erhält der Tester schon während der Testdurchführung von Main Compare ein Feedback welche Tendenzen sich durch die gemessenen Werte abzeichnen.

3.4 Tests

3.4.1 Topic Übersicht

Die Topic Übersicht innerhalb des Rahmens ließ sich unkompliziert Testen, nämlich durch starten verschiedener ROS nodes. Diese nodes stellen verschiedene topics zur Verfügung, welche in der Topic Übersicht eingeblendet werden sollen. Dies funktioniert zuverlässig und es konnten keine Probleme mit der Visualisierung festgestellt werden.

3.4.2 Abonnieren von topics durch Plug-ins

Das Abonnieren wird, wie im Abschnitt Implementierung genauer beschrieben, durch die Klasse ROSManager vorgenommen. Für die bisher entwickelten beiden Plug-ins Main Compare und Camera View traten während durchgeführter Defekttests keine Laufzeitfehler auf.

3.4.3 Main Compare

Eine Schwierigkeit beim Testen von Main Compare bestand zunächst darin, geeignete Eingabedaten in Form von path messages zu erhalten. Es konnten jedoch Pfaddaten, die durch Testfahrten des Roboters entstanden sind, genutzt werden. Dadurch konnten während durchgeführter Defekttests einige Fehler identifiziert und beseitigt werden. Allerdings konnten noch keine Tests mit sich kontinuierlich ändernden Pfaddaten durchgeführt werden. Theoretisch sollte das Verhalten des Programms sich aber nicht gegenüber den statisch Pfaddaten des Roboters ändern.

KAPITEL 4

Fazit

Pathcompare und Main Compare stellen in der derzeitigen Form ein Werkzeug dar, welches die Durchführung der mit dem Referenzsystem geplanten Tests erleichtern kann. Dies wird vorallem dadurch erreicht, dass es dem Nutzer ermöglicht wird, während der Testdurchführung, relevante Informationen an einem Ort zu beobachten. Zusätzlich wird durch die Plug-in Erweiterbarkeit stets die Möglichkeit offengehalten auf neue Testanforderungen zu reagieren oder Informationen andersartig zu visualisieren. Gleichzeitig werden beim Plug-in Entwickler und dem Nutzer keinerlei tiefergehende ROS Kenntnisse vorausgesetzt, da die Anbindung an ROS vollständig durch Pathcompare gekapselt wurde. Auch wenn Pathcompare seine geplante Grundfunktionalität in der derzeitigen Version realisiert, gibt es dennoch denkbare Erweiterungen und Anpassungen welche die Benutzbarkeit verbessern können. Außerdem sollten wie bereits im Abschnitt Anwendung angesprochen verstärkt Defekt- und Benutzertests durchgeführt werden um die Benutzbarkeit abzusichern und zu verbessern. Insgesamt wird aber eine solide Grundlage geliefert, welche die Test, mit den Sensorknoten und dem Referenzsystem, vereinfachen kann.