Das Player/Stage System

Thimo Langbehn
<thimo@g4t3.de>

31. Januar 2011

Inhaltsverzeichnis

Wenn ein System aus verschiedene Software-Komponenten konstruiert werden soll, sind gemeinsame Schnittstellen erforderlich. Gerade in der Robotik, wo Systeme aus vielen komplexen Teilen zusammengesetzt werden, ist es sinnvoll, existierende Treiber und Software für diese Komponenten zu nutzen und mit der eigenen Software zu integrieren.
Die scheitert aber häufig an dem hohen Integrationsaufwand, verursacht durch inkompatible Schnittstellen und Konzepte. Um diese Problem zu beheben existieren einige Rahmensysteme (*frameworks*), welche Schnittstellen oder auch Konzepte für Softwareteile vorgeben, und eventuell auch eine Menge an unterstützenden Funktionen anbieten. Eines der am weitesten verbreiteten *frameworks* in der Robotik ist `Player`, aus dem `Player Project`.
Das `Player Project` [5] umfasst drei Teilprojekte, `Player`, `Stage` und `Gazebo`. Sie werden als `Player Project` zusammengefasst, weil `Stage` und `Gazebo` beide eine `Player`-Integration besitzen und zusammen entwickelt werden.
Der erste Teil, `Payer`, ist ein Robotik-*framework* und definiert Schnittstellen für übliche Bausteine von Robotern. Neben diesen Schnittstellen wird durch `Player` auch ein Kommunikationssystem gestellt, über das die einzelnen Komponenten miteinander verbunden werden. Damit spezifiziert `Player` ein Komponentenmodell und beinhaltet zugleich eine Implementierung dieses Modells. Schließlich liefert `Player` auch Realisierungen der Schnittstellen für diverse Sensoren, Aktoren oder Algorithmen.
Um die experimentelle Arbeit mit Software zu erleichtern macht es Sinn, Simulatoren zu verwenden. Sie erlauben das praktischen Testen von neuen Algorithmen ohne große Risiken für Mensch oder Material, beschleunigen den Vergleich von verschiedenen Roboter- oder Umgebungs-Variationen und ermöglichen es, nicht relevante Details wie beispielsweise Stromversorgung, Kommunikationsstörungen oder Hardwaredefekte auszublenden.
Die beiden anderen Teile aus dem `Player Project` sind solche Simulatoren.
`Stage` ist ein Roboter und Sensor-Simulator. Sein Ziel ist es, eine große Zahl von Robotern gleichzeitig zu simulieren. Dazu gibt [12] an, 200 mobile Roboter mit Sonar in Echtzeit auf einem 600MHz Pentium simulieren zu können, mit einer zeitlichen Auflösung von 100ms und einer räumlichen von 2cm. Für diese Leistungsfähigkeit wurde `Stage` auf die Simulation von 2 Dimensionen beschränkt. Für die Auswertung von Sensoren und die Darstellung wird ab der Version 3 noch eine Höhe verwendet. Insbesondere für den Konfigurationsraum der simulierten Roboter steht diese Dimension aber nicht zur Verfügung. `Stage` wird deshalb auch als 2.5D Simulator bezeichnet.
`Gazebo` ist, wie `Stage`, ein Roboter Simulator, er simuliert jedoch alle drei Raumdimensionen und sogar Festkörperdynamik. Das macht die Simulation natürlich deutlich Ressourcen-intensiver, obgleich auch `Gazebo` technisch in der Lage ist mehr als

einen Roboter gleichzeitig zu simulieren.
Der Hauptgegenstand dieses Textes jedoch ist das *framework*, `Player`. In Abschnitt 1 wird auf einige der in `Player` definierten Schnittstellen eingegangen. Anschließend führt Abschnitt 2 in die Struktur und Verwendung von `Player` ein und als Abschluss werden in Abschnitt 3 einige der existierenden Komponenten von `Player` vorgestellt. Die Diagramme in diesem Text basieren auf den Fundamental Modeling Concepts (FMC) und der Unified Modeling Language (UML).

1 Schnittstellen

Die von `Player` definierten Schnittstellen formen die Basis des Projekts. Sie ermöglichen die hohe Kapselung von Anwendungsspezifischen Modulen und erlauben es, Nutzer und Anbieter einer Funktion unabhängig voneinander zu entwickeln und auch zu testen.
`Player` definiert diese Schnittstellen in Form eines Nachrichten-Protokolls und zugehörigen Datentypen. Jede Schnittstelle definiert eine Menge von Nachrichten, und jede Nachricht kann, neben dem Protokoll-Teil, maximal eine Instanz der definierten Datentypen beinhalten.
Diese Nachrichten-basierten Schnittstellen werden in Bibliotheken für einzelne Programmiersprachen verfügbar gemacht und gekapselt. Im Folgenden werden die wichtigsten Schnittstellen beschrieben.

1.1 Position2d

Das `Position2d` Interface ist wohl das wichtigste für radgetriebene Roboter und zugehörige Simulationen. Es erlaubt das Steuern eines Roboters in bis zu vier verschiedenen Modi, sowie das Setzen und Abfragen von relativen oder absoluten Positionierungen, wie sie durch Odometrie oder Global Positioning System (GPS) geliefert werden. Weiterhin können die aktuelle Geschwindigkeit, der Motor-Status und die groben Abmessungen des Roboters ermittelt werden. Es sind auch Nachrichten zum Schalten der Motoren, zum Einstellen der Parameter von PID-Reglern für Geschwindigkeit und Position und zum Konfigurieren eines Geschwindigkeitsprofils vorhanden. Eine Implementierung muss allerdings nicht alle diese Operationen unterstützen.
In dem `Position2d`-Interface von `Player` werden vier Steuerungsarten für die Bewegung eines Roboters unterschieden:

1. Geschwindigkeit
2. Zielposition

```
/** position 2d velocity command */
typedef struct player_Position2d_cmd_vel
{
  /** translational velocities [m/s,m/s,rad/s] (x, y, yaw)*/
  player_pose2d_t vel;
  /** Motor state (FALSE is either off or locked, depending on the
      driver). */
  uint8_t state;
} player_Position2d_cmd_vel_t;
```

Listing 1: Player Position2d Interface - Geschwindigkeit Setzen

3. Autoparameter
4. Kurs

Eine Implementierung des `Position2d` Interface kann einen oder mehrere dieser Varianten berücksichtigen.
Die von den meisten Implementierungen angebotene Steuerungsmöglichkeit ist die Vorgabe von einer oder zwei Geschwindigkeiten für die Translation und einer Rotationsgeschwindigkeit. Die Geschwindigkeiten orientieren sich an dem Roboterkoordinatensystem. Die Kombination dieser Anteile führt zur Vorgabe einer Bahnkurve oder Geraden (wenn die Rotationsgeschwindigkeit 0 ist). Bei vielen Robotern wird aufgrund von kinematische Zwangsbedingungen nur die x-Komponente einer Geschwindigkeit unterstützt. Der Datentyp für das Kommando zum Setzen des Geschwindigkeitsvektors ist als Beispiel in Listing 1 aufgeführt. Das Verhalten dieses Kommandos kann optional über einen Konfigurationswert beeinflusst werden beispielsweise um zwischen direkter Motorsteuerung und Geschwindigkeitssteuerung zu wechseln, wenn die implementierende Komponente solche Alternativen anbietet.
Etwas aufwendiger zu realisieren ist die Steuerungsmöglichkeit über eine Zielposition und -orientierung (in Weltkoordinaten). Hier muss die Implementierung in der Regel einen Positionsregler vorsehen.
Die als Autoparameter aufgeführte Variante erlaubt die Vorgabe einer Geschwindigkeit und eines Lenkwinkels. Dies ist für Kraftfahrzeug-artige Antriebe vorgesehen und impliziert das Unvermögen eine reine Rotation durchzuführen.
Die letzte Steuerungsart, hier als Kurs bezeichnet, erlaubt das Einstellen einer Geschwindigkeit und einer Richtung (relativ zum Weltkoordinatensystem). Dabei bewegt der Roboter sich auf einer Geraden in die angegebene Richtung mit der vorgegebenen Geschwindigkeit. Diese Art der Ansteuerung kann generell nur von omnidirektionalen Antrieben wie dem Killough-Antrieb realisiert werden.
Das `Position2d` Interface erlaubt die Abfrage und das Setzen einer Position. Dies ist für *Position Tracking* gedacht, kann aber auch von Lokalisierungsalgorithmen zur

Ausgabe ihrer Resultate verwendet werden.
Von den verbleibenden Operationen werden die Abfrage der Geometrie und Startposition eines Roboters von vielen Werkzeugen genutzt um die ihre Darstellung des Roboters zu initialisieren.

1.2 Power

Über das `Power`-Interface kann der aktuelle Ladezustand und Verbrauch (oder Ladungszuwachs) abgerufen werden. Bei Ladestationen ist es auch möglich den Ladevorgang zu kontrollieren.

1.3 Localize

Das `Localize` Interface ist entwickelt worden, um die Resultate insbesondere von probabilistischen Lokalisierungsalgorithmen abrufen zu können. Hiermit können mehrere Positions-Hypothesen inklusive Varianzen und Kovarianzen[1] abgefragt werden. Damit ist die `Localize` Schnittstelle in der Lage, alle relevanten Informationen eines mit Gaußschen Verteilungen arbeitenden Verfahrens einzuholen.
Für Partikel-Filter existieren Operationen und Datenstrukturen um die aktuellen Partikel abzufragen, und die mitgelieferten Werzeuge sind auch in der Lage diese darzustellen, was eine schnelle und intuitive Einschätzung des Verfahrens möglich macht.
Schließlich erlaubt das Interface auch das Setzen der geschätzten Pose inklusive ihrer (Ko-)Varianzen, normalerweise zur Initialisierung oder Zurücksetzung eines Algorithmus.

1.4 Map

`Map` ist die erste der beiden Kartenschnittstellen. Sie erlaubt das Abrufen von Rasterkarten oder von Liniensegmenten einer Vektorkarte. Bei der zweiten Variante werden alle Liniensegmente auf einmal übertragen, bei den Rasterkarten kann sowohl der Anfragende, als auch der Kartenlieferant den gelieferten Ausschnitt begrenzen. Für Vektorkarten existiert eine neue Schnittstelle basierend auf dem OGC Standard. (GEOS). For more information about OGC see `http://opengeospatial.org` For more information about GEOS see `http://geos.refractions.net/ro/doxygen_docs/html`

[1] Die vollständige Kovarianzmatrix ist erst ab Rev. 8859 verfügbar.

1.5 Laser

`Laser` ist eine Schnittstelle zur Abfrage der Distanzwerte eines Laserentfernungsmessers.
Statt ihr sollte, so möglich, `Ranger` verwendet werden. Wenn ein Sensor noch `Laser` anbietet aber von einem Algorithmus schon `Ranger` erwartet wird, kann die Komponente `lasertoranger` zur Umsetzung verwendet werden.

1.6 Ranger

Das neuere `Ranger`-Interface soll die alten Schnittstellen `Laser`, `Ir` und `Sonar` ersetzen. Es ermöglicht die Ansteuerung von einem einzelnen Sensor oder einer Reihe von Sensoren (Sensorbank). Im Falle des einzelnen Sensors können mehrere Entfernungswerte, wie sie von einem twodimensional (2D) Laserentfernungsmesser geliefert werden, ausgelesen werden, Wenn eine Sensorbank repräsentiert wird, darf jeder einzelne Sensor nur einen Entfernungswert liefern. Die Schnittstelle ermöglicht das Abfragen der Sensorposition(-en) und Abmessung(-en). In einem Datensatz sind Entfernung- oder Intensitätswerte enthalten, und optional auch die Geometrie des Sensors zum Zeitpunkt der Messung (für bewegbare Sensoren).

1.7 Fiducial

Insbesondere zum Testen von Algorithmen ist es sinnvoll, von uninteressanten Teilen wie Mustererkennung oder Triangulation zu abstrahieren. In solchen Fällen bietet sich das `Fiducial` Interface an. Es liefert die relativen Koordinaten samt Orientierung und Identifikation eines Objekts (als *fiducials* bezeichnet) mit seinen optionalen (Ko-)Varianzen. Damit eignet es sich aber auch für Mustererkennungssysteme, welche beispielsweise aus einem Kamerabild solche Informationen extrahieren können.
Den Kern der Schnittstelle formen die Nachrichten mit wahrgenommen *fiducials* im Field Of View (FOV) des Sensors. Weiterhin gibt es noch die Möglichkeit, das FOV des Sensors einzustellen oder die Abmessungen und Position eines *fiducials* abzufragen.

1.8 Camera

Mit dem `Camera`-Interface kann eine Komponente Bilder, beispielsweise einer Kamera oder aus einer Datei, zur Verfügung stellen. Der Transport geschieht Bildweise, und die Bilder werden dabei unkomprimiert oder im JPEG File Interchange Format (JFIF), und nacheinander übertragen. In der Schnittstelle ist auch eine Möglichkeit zum Wechseln der Bildquelle vorgesehen, wenn mehrere zur Auswahl stehen.

1.9 Blobfinder

Bildanalyse-Systeme, welche "Farbklekse" (*blobs*) in einem Bild finden, können ihre Ergebnisse über das `Blobfinder` Interface zugänglich machen. So könnte eine Komponente, beispielsweise durch eine Kamera (oder das `Camera` Interface), Bilder eines Ballons erhalten, und dessen Position über diese Schnittstelle anderen Komponenten zugänglich machen. Dieses Interface ist insbesondere auch deswegen bedeutsam, weil es mit geringem Aufwand zu simulieren ist. Transportiert werden unter anderem eine Identifikation, die Farbe, Fläche, und Position des erkannten Objekts.

1.10 Graphics2d

Das `Graphics2d`-Interface ermöglicht es einer Komponente, durch das `Player` *framework* hindurch Zeichenoperationen auszuführen. Dies ist beispielsweise hilfreich bei Algorithmen, bei denen ein Teil des internen Zustand visuell gut dargestellt werden kann, aber keine Schnittstelle existiert welche für die zugehörigen Daten geeignet ist. In der Situation kann mittels `Graphics2d` eine anwendungsspezifisches Interface vermieden werden.

2 Architektur

Die in Abschnitt 1 beschriebenen Schnittstellen erlauben es, die Software eines Roboters in sachlich separate Software-Komponenten aufzuspalten. Das ist sinnvoll, da kleine Komponenten jeweils nur wenige Abhängigkeiten besitzen und somit leichter testbar und wiederverwendbar sind. Für den Aufbau eines Systems wird jedoch noch ein Mechanismus benötigt, diese Teile miteinander zu verbinden und zu einem Ganzen zusammenzusetzen. Diese Aufgabe übernimmt das Kommunikationssystem von `Player`. Dort ist ein netzwerktransparentes Protokoll zum Vermitteln von Nachrichten implementiert.

2.1 Player Server

Der zentrale Teil des Kommunikationssystems von `Player` ist `Player Server`. Neben dem Transportmechanismus für Nachrichten besitzt der `Player Server` einen *Plugin*-Mechanismus zum Einbinden der verschiedenen Komponenten. Diese Komponenten werden in `Player` aus historischen Gründen als *driver* bezeichnet. Dabei handelt es sich nicht immer um Module die Hardware ansteuern, sondern um Software-Teile, welche eine oder mehrere der durch `Player` definierten Schnittstellen implementieren. Da der Begriff *driver* häufig zu Verwirrung führt, wird an dieser Stelle weiterhin die alternative Bezeichnung Komponente verwendet.

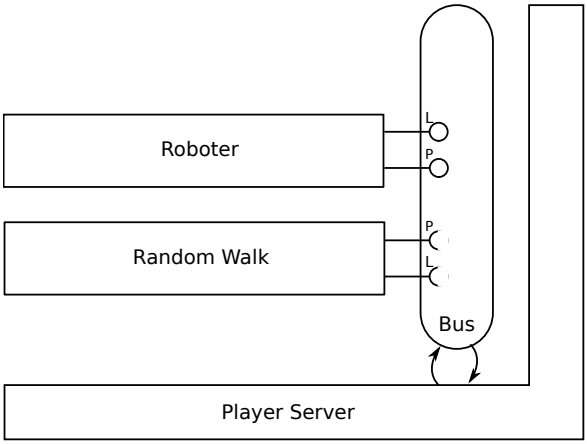

Abbildung 1: Der `Player Server` mit zwei Komponenten (FMC-Block)

Eine Komponente kann eine oder mehrere der in `Player` definierten Schnittstellen implementieren, aber auch von solchen abhängen. Dabei können Schnittstellen zwangsweise für die Funktion einer Komponente erforderlich sein oder optional genutzt werden. Es ist auch möglich, dass eine Komponente mehrere Instanzen einer Schnittstelle erfordert oder bereitstellt.
Der `Player Server` ermöglicht es, eine Menge von Komponenten zu instanziieren und miteinander zu verbinden.
In Abbildung 1 wird ein Beispiel eines solchen Komponenten-Netzwerks mit zwei Elementen dargestellt. Der `Player Server` erzeugt die Instanzen der Komponenten, eine vom Typ `Roboter`, und eine vom Typ `Random Walk`.
Die `Roboter`-Komponente implementiert zwei Schnittstellen, einmal `Position2d` (`P`) und einmal `Laser` (`L`). Von dieser Art von Komponenten stammt die Bezeichung *driver* in `Player`, sie abstrahiert von der spezifischen Ansteuerung des Antriebssystems und des Laserscanners und macht sie über die passenden `Player`-Schnittstellen verfügbar, wie ein Treiber.
Die Komponente `Random Walk` stellt keine Schnittstelle zur Verfügung, erfordert aber zwei zum Betrieb, eine `Position2d` (`P`) und eine `Laser` (`L`). Bei `Random Walk` handelt es sich um eine Komponente, welche keinerlei Interaktion mit der Hardware hat, sie stellt nur die Implementierung eines Algorithmus zur Verfügung.
Der `Player Server` implementiert gar keine Schnittstellen, er betreibt nur das Nachrichten Transportsystem (hier `Bus`).
Damit die beiden Komponenten miteinander interagieren können, müssen sie verknüpft werden, eine Aufgabe die ebenfalls von dem `Player Server` übernommen

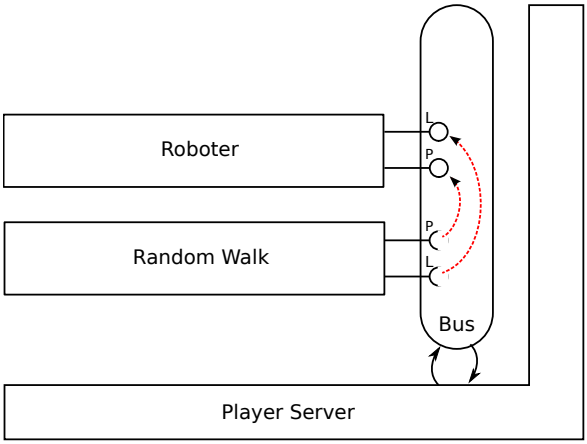

Abbildung 2: Zwei verbundene Komponenten (FMC-Block)

wird.
In Abbildung 2 werden die erstellten Verbindungen dargestellt. Verbindungen werden immer auf Basis von Schnittstellen erstellt, und es wird immer eine benötigte und eine zur Verfügung gestellte Schnittstelle des selben Typs miteinander verbunden. Dabei wählt der `Player Server` die zu erstellenden Verbindungen nicht selber aus, sondern sie müssen, zusammen mit den zu erstellenden Komponenten, als Konfiguration vorgegeben werden.
Die dazu passende Konfiguration für den `Player Server` ist in Listing 2 angegeben. Darin sind zwei `driver`-Blöcke definiert. Jeder Block besitzt zumindest das `name`-Attribut, in dem die Bezeichnung der Komponente angegeben ist, um die es in dem Block geht.
Jeder Block führt zu einer Instanz einer Komponente. Wenn beispielsweise zwei Instanzen der `RandomWalk`-Komponente benötigt würden, so müssten zwei Blöcke (jeweils mit `RandomWalk` als Wert für `name`) dazu erstellt werden. Ein Block kann eine Liste von Schnittstellen anbieten, die mit `provides` gekennzeichnet, oder auch Schnittstellen erfordern, die mit `requires` gekennzeichnet sind. Jede Instanz-Bezeichnung einer Schnittstelle ist mit doppelten Hochkommata eingefasst. Im einfachsten Fall besteht so eine Instanz aus dem Namen der Schnittstelle, einem Doppelpunkt und der Instanznummer. Damit der `Player Server` zwei Komponenten bzw. deren Schnittstellen miteinander verbindet müssen diese Schnittstellen-Bezeichnungen bei dem Anbieter und Nutzer der Instanz exakt übereinstimmen. Aber auch bei korrekter Angabe würden die Verbindungen erst hergestellt, wenn die Komponenten instanziiert werden, und die Komponenten werden normalerweise erst dann instanziiert, wenn

```
driver
(
name "Roboter"
provides [ "position2d:0" "laser:0" ]
)

driver
(
name "RandomWalk"
requires [ "position2d:0" "laser:0" ]
alwayson 1
)
```

Listing 2: Zwei verbundene Komponenten (player config)

sich jemand mit einer ihrer Schnittstellen verbindet. Bei einem reinen Komponenten-System würde also zunächst gar nicht passieren. Um dies zu umgehen, kann bei einer Komponente die Zeile `alwayson 1` angegeben werden, welche dazu führt, dass sie direkt beim Starten des `Player Server` geladen und gestartet wird.
Wenn die in Listing 2 angegebene Konfiguration in der Datei `beispiel.cfg` stünde, dann könnte das beschriebene System mit dem Aufruf `$> player beispiel.cfg` geladen werden.
Das in Abbildung 2 dargestellte System könnte in dieser Konfiguration beispielsweise auf einem eingebetteten System betrieben werden, an dem ein Laserscanner und ein Antrieb angeschlossen sind.
Angenommen dieses System soll um einen Monte Carlo Localization (MCL) Algorithmus erweitert werden. `Player` enthält mit `AMCL` bereits eine passende Komponente, welche eine adaptive MCL realisiert. Wie bei `Random Walk` auch handelt es sich dabei um eine Algorithmus Komponente ohne Hardwarebindung. Und wie die beiden anderen Teile würde die `AMCL`-Komponente durch den `Player Server` geladen und verbunden werden. Wenn das kleine, eingebettete System des Roboters damit überfordert ist, kann die Komponente aber auch auf einem anderen, über Local Area Network (LAN) oder einem anderen Netzwerk mit dem Roboter verbundenen Personal Computer (PC) betrieben werden. In dem Fall würde auf dem PC eine andere Instanz des `Player Server`s laufen und die Lokalisierungs-Komponente laden.
Diese Situation ist in Abbildung 3 skizziert. Die `AMCL`-Komponente benötigt ein `Laser`-Interface zur Erfassung der Umgebung und zudem noch die an den Roboter gesendeten Bewegungs-Kommandos. Da solche Kommandos von dem Nutzer einer Schnittstelle an den Anbieter gesendet werden (von `Random Walk` an `Roboter`), stellt `AMCL` ein `Position2d`-Interface zur Verfügung, obwohl es sich aus logischer Perspektive um einen Nutzer (Konsumenten) der Schnittstelle handelt. Mit `W` ist eine Instanz

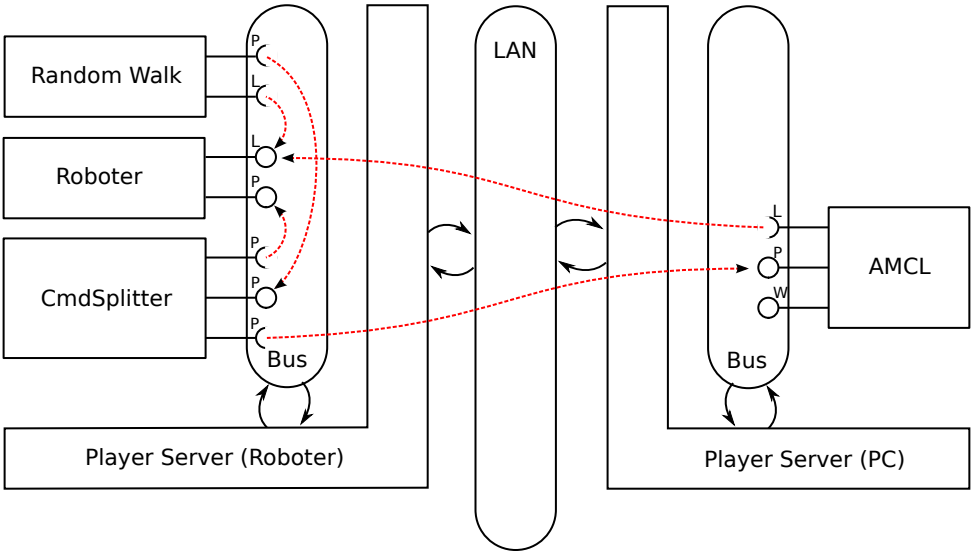

Abbildung 3: Zwei Instanzen des `Player Server`s (FMC-Block)

des `Localize`-Interface gekennzeichnet, welche die Komponente zur Abfrage seiner Ergebnisse anbietet.
Die Netzwerkabstraktion des Nachrichten-Transportsystems von `Player` erlaubt es, Komponenten über Rechnergrenzen hinweg transparent miteinander zu kombinieren. Daten-Nachrichten wie die Messungen des `Laser` Teils werden von `Roboter` automatisch an alle Interessenten (*subscriber*) versendet. Die Verbindung des `Laser`-Interface erfordert somit nicht mehr Aufwand als sie in der Konfiguration des `Player Server`s auf dem PC anzugeben (ein Anbieter an mehrere Nutzer).
Um die Kommandos von `Random Walk` an mehrere Empfänger zu senden (ein Nutzer an mehrere Anbieter) stellt `Player` die Komponente `CmdSplitter` zur Verfügung. Diese tut genau das, was der Name vermuten lässt, sie dupliziert die empfangenen Kommandos und sendet sie an alle angeschlossenen Interface-Anbieter.
Der `Player Server` ist also nicht, wie der Name vermuten lassen würde, nur ein Anbieter von Diensten, sondern eine Umgebung, in der Komponenten instanziiert, und durch sie Schnittstellen sowohl angeboten als auch genutzt werden können. Und dies ist auch über verschiedene Instanzen des `Player Server` hinweg möglich. Damit ähnelt der `Player Server` mehr einem *application server* als einem klassischen Anbieter-Programm.

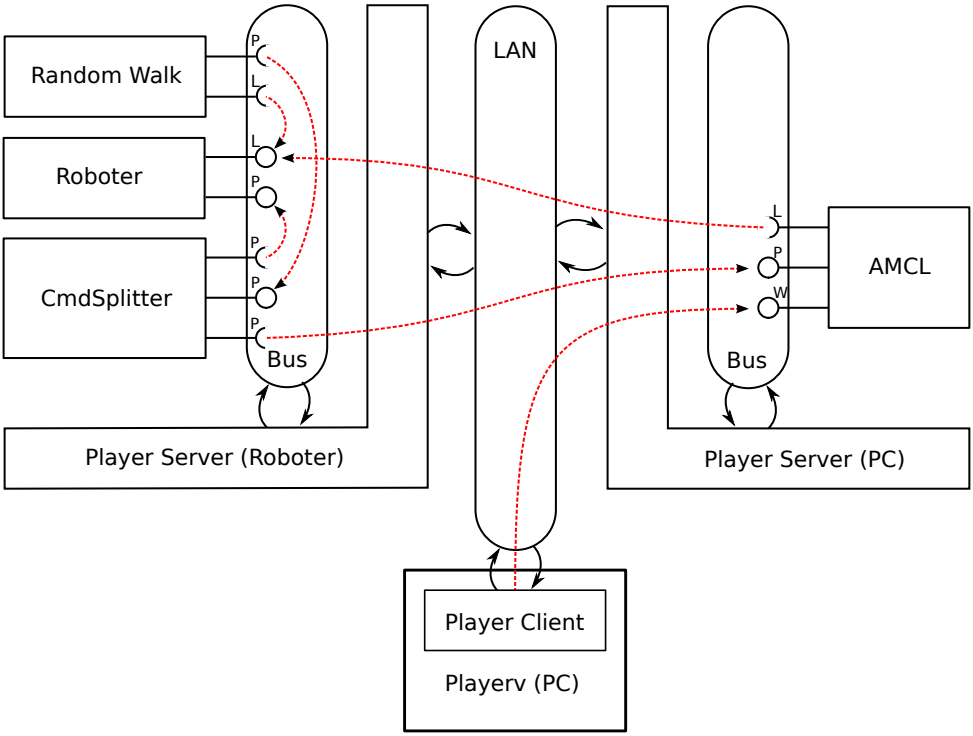

Abbildung 4: Zwei Instanzen des `Player Server`s mit `playerv` (FMC-Block)

2.2 Player Clients

Mit in `Player` enthalten sind Client-Bibliotheken für verschiedene Programmiersprachen, welche die in `Player` definierten Schnittstellen als Application Programming Interface (API) nutzbar machen. Dabei kümmert sich die Client-Bibliothek um den Nachrichtentransport und eventuell vorzunehmende Konvertierungen.
Die Client-Bibliotheken bieten, im Gegensatz zum `Player Server`, keinen Mechanismus zum Laden von Komponenten an und können somit Schnittstellen nur nutzen und nicht selbst anbieten.
Um das in Abbildung 3 von der `AMCL`-Komponente angebotene `Localize`-Interface zu nutzen, könnte ein auf einer Client-Bibliothek aufsetzendes Programm verwendet werden. Solch ein Programm würde sich über das LAN mit der PC-Instanz des `Player Server`s verbinden, die Daten zur Lokalisierung abfragen und möglicherweise in Textform oder graphisch darstellen.

```
from playercpp import *
# Verbinden mit dem Player Server auf diesem Rechner
pc = PlayerClient('127.0.0.1')

# Erstelle einen Proxy fuer die Instanz Position2d:0
p2p = Position2dProxy(pc, 0)

# Abfragen der Geometriedaten
p2p.RequestGeom()
size = p2p.GetSize()
print 'Die Abmessungen sind: %.3f x %.3f x %.3f' % (size.sw, size.sl,
    size.sh)

# Aufraeumen
del p2p
del pc
```

Listing 3: Beispielclient in Python

Bei `Player` werden schon einige solcher häufig benötigten Client-Programme mitgeliefert. Eins davon ist `playerv`, welches in der Lage ist, Daten von dem meisten Schnittstellen darzustellen und diese auch anzusteuern (Kommandos zu senden), einschließlich `Localize`-Informationen.
Dieses Werkzeug könnte also auf einem beliebigen, an das LAN angeschlossenen Rechner gestartet und mit dem `Localize`-Interface von `AMCL` verbunden werden. In Abbildung 4 wurde die vorherige Situation um solch eine Instanz von `playerv` ergänzt.
In `Player` sind Schnittstellen zu den Sprachen `C`, `C++`, `Python`, `Ruby`, `Ada`, `Java`, `Lisp` sowie zu `Matlab` und `Octave` enthalten. Insbesondere die Skriptsprachen beschleunigen das Schreiben von Clients ungemein. In Listing 3 ist ein Beispiel für `Python` angegeben.

2.3 Werkzeuge

Im folgenden werden einige der von `Player` zur Verfügung gestellten Werkzeuge beschrieben.

2.3.1 Playerprint

Eines der einfachsten und dennoch unverzichtbaren Werkzeuge ist `playerprint`. Es ermöglicht regelmäßige Ausgaben der durch Schnittstellen von Komponenten gelieferten Daten auf einem Terminal.
Dabei werden alle von `Player` definierten Schnittstellen unterstützt.

Mögliche Aufrufparameter werden beim parameterlosen Aufruf des Programms aufgelistet. Wird der `host` Parameter nicht angegeben nimmt `playerprint` die IPv4-Adresse `127.0.0.1` dafür an, also das *loopback-Interface*. Für den Port wird als Vorgabe der Standardport des `Player Servers` verwendet (`6665`). Der einzige Pflichtparameter für `playerprint` ist die Typenbezeichnung für die abzufragende Schnittstelle, beispielsweise `Position2d`, `Laser` oder `Localize`. Mit `$> playerprint localize` könnten beispielsweise auf dem PC die Ergebnisse der `AMCL`-Komponente in Abbildung 3 ausgelesen werden.

2.3.2 Playerjoy

Wenn ein Roboter das Setzen einer Geschwindigkeit über eine `Position2d` oder eine `Position3d` Schnittstelle erlaubt, kann `playerjoy` dazu verwendet werden den Roboter zu steuern. Dabei steht ein Eingabemodus über die Tastatur zur Verfügung, oder über einen der angeschlossenen Joysticks.

2.3.3 Playerv

Das Arbeitspferd unter den verfügbaren Werkzeugen ist `playerv`, es ermöglicht das Auslesen und Visualisieren der meisten üblichen Schnittstellen und auch das Ansteuern der wichtigsten Funktionen. Dieses Programm besitzt ein Graphical User Interface (GUI) mit einem Koordinatensystem, in dem verschiedene Daten wie die Roboterposition, Sensordaten oder eine Karte dargestellt werden können. `playerv` ist in der Lage, sich zu mehr als einen Roboter über zugehörige `Position2d` Schnittstellen zu verbinden, ihre Daten darzustellen und sie zu steuern. In der Regel macht es aber mehr Sinn, nur eine zusammengehörige Gruppe von Komponenten von einem Autonomous Mobile Robot (AMR) darzustellen und für mehrere Roboter `playernav` zu verwenden.

2.3.4 Playernav

Mit `playernav` existiert ein GUI-Programm mit dem sich eine Karte darstellen, und die Positionen einer Menge von Robotern komfortabel visualisieren lassen. Es wurde entwickelt um das abstraktere `planner-` und `Localize`-Interface anzusteuern. Im Gegensatz zu `playerv` können jedoch keine weiteren Schnittstellen genutzt werden. `pmap` ist ein Partikel-Filter-basiertes Simultanous Localization And Map building (SLAM)-Werkzeug für `Laser-` und Odometrie-Daten (`Position2d`). Erstellte Karten können mit `rmap` verbessert werden. Das Werkzeug enthält noch die separat nutzbare Teile `lodo` und `omap`. Die Bibliothek `lodo` dient dem Korrigieren einer Odometrie-Pose durch Laser-Scanner Daten. Mit `omap` können Gitterkarten aus (zusammengesetzten) Laser-Scanner Daten erstellt werden.

Im folgenden noch eine liste der weiteren, in `Player` verfügbaren Werkzeuge:

`barcodes` ist ein einfaches Skript zum Erstellen von Universal Product Code (UPC) Barcodes aus einer `Xfig` Vorlage.

`dgps_server` ist ein Programm zum Sammeln und weiterleiten von Differential Global Positioning System (DGPS) Korrekturdaten an GPS Komponenten.

`logsplitter` spaltet Logdateien anhand von Zeitdifferenzen der Einträge auf.

`playercam` stellt Bilder einer `camera-` oder `blobfinder`-Schnittstelle dar.

`playerprop` ermöglicht das Abfragen und Setzen von Komponenten-Eigenschaften.

`playervcr` erlaubt das Steuern der Aufnahme oder Wiedergabe von Nachrichten über die `readlog` und `writelog` Komponenten.

`playerwritemap` schreibt Karteninformationen eines `Map`-Interface in eine Portable Network Graphics (png)-Datei.

3 Komponenten

Player besitzt eine große Menge an bereits vorhandenen Komponenten. Diese Komponenten lassen sich grob in drei Kategorien einteilen:

Sensortreiber stellen eine Schnittstelle zu einem Hardwaresensor zur Verfügung.

Aktortreiber kapseln Antriebe, Greifer und ähnliche aktive Hardware eines Roboters.

Algorithmik-Komponenten stellen hardwareungebundene Prozesse und Funktionalität über `Player`-Schnittstellen bereit.

Eine der bekanntesten Komponenten für `Player` ist `Stage`, das so entstehende System, `Player/Stage`, wird in Unterabschnitt 3.7 beschrieben. Wenn eine realitätsnähere Simulation erforderlich ist kann auch `Gazebo` mit `Player` betrieben werden.
Im Folgenden werden einige der anderen in `Player` enthaltenen Komponenten vorgestellt. Dabei wird sich auf die für diese Arbeit untersuchten Module beschränkt.

3.1 CmdSplitter

`CmdSplitter` ist eine Komponente, die Kommandos, welche auf einem angebotenen Interface eingehen, an alle dafür konfigurierten (genutzten) Schnittstellen repliziert. Der Typ der Schnittstelle ist dabei beliebig, muss jedoch für die eine angebotene und alle genutzten Instanzen gleich sein. Die Anzahl der genutzten Instanzen ist ebenfalls beliebig, Antworten auf Kommandos werden aber nur von der ersten entgegengenommen und an den ursprünglichen Sender des Kommandos weitergeleitet.

3.2 VFH

Die `vfh`-Komponente implementiert ein kollisionsvermeidendes Verhalten auf Basis des Vector Field Histogram (VFH) Algorithmus, genau gesagt die VFH+ Variante [9]. Die verbesserte Version des ursprünglichen VFH Algorithmus versucht beispielsweise Sackgassen zu vermeiden. Diese Komponente benötigt ein `Laser` oder `Sonar` Interface zur Hinderniserkennung und ein `Position2d` Interface, um den Roboter anzusteuern. Die Zielposition kann über ein angebotenes `Position2d` oder `Planner` Interface vorgegeben werden.

3.3 Sicklms200

Der in der Robotik sehr verbreitete Laserentfernungsmesser `LMS200` von `SICK` kann über die `SickLms200`-Komponente angesteuert und ausgelesen werden. Als Anschlussvarianten sind Seriell oder Seriell über Universal Serial Bus (USB) möglich. Die Daten werden über ein `Laser`-Interface zur Verfügung gestellt. Es existieren auch Variante für die Modelle `LMS400`, `LD-MRS` und den bei outdoor Robotern häufigen `S3000`.

3.4 Obot

Für den von der Firma `Botrics` hergestellten `Obot` existiert eine Komponente, die den Differentialantrieb und den Batteristatus über die `Player`-Schnittstellen `Position2d` bzw. `Power` nutzbar macht. Der auf dem `Obot` montierte Laserentfernungsmesser kann durch die `SickLms200`-Komponente separat angesteuert werden.

3.5 Wavefront

`Wavefront` ist eine von `Player` mitgelieferte Implementierung des *Wavefront* Algorithmus welche über das `Planner` Interface gesteuert werden kann. Bei dem Algorithmus wird mittels einer vorhandenen Karte inkrementell ein optimaler Pfad unter Vermeidung statischer Hindernisse gefunden.

Der Algorithmus startet bei dem Zielfeld und berechnet die Weg-Kosten für direkt angrenzende Felder basierend auf dem Minimum der bereits bearbeiteten Zellen in ihrer unmittelbaren Umgebung. Dies wird solange fortgesetzt bis der Startpunkt erreicht ist, von welchem dann der Weg entlang des fallenden Kostengradienten gewählt wird.

3.6 AMCL

Als Lokalisierungsalgorithmus existiert mit `amcl` eine Implementierung der adaptiven MCL nach [3]. Adaptiv ist die Implementierung in Hinsicht auf die Partikelanzahl, welche zur Repräsentation der zweidimensionalen Probability Density Function (PDF) verwendet werden. Wenn die Position des Roboters (noch) nicht sehr genau bekannt ist, wird eine große Anzahl an Partikeln eingesetzt, um dem Problem des dauerhaften Hypothesenverlusts zu begegnen. Konvergiert das Filter auf einige wenige oder nur eine wahrscheinliche Pose, wird die Anzahl der Partikel verringert, um überflüssige Ressourcennutzung zu vermeiden.

3.7 Player/Stage

`Stage` ist ein effizienter Roboter-Simulator für zwei Dimensionen, für den auch eine Kapselung als `Player`-Komponente existiert [11]. Die zentrale `Stage`-Komponente stellt einige optionale, von der konkreten Simulation unabhängige Schnittstellen zur Verfügung, zum Beispiel das `Simulation`- oder ein `Graphics2d`-Interface. Die Schnittstellen für simulierte Roboter hängen jedoch von der virtuellen Welt ab. Über eine Konfigurationsdatei (`worldfile`) werden ein oder mehrere Roboter samt Aktorik und Sensorik spezifiziert. Für jeden simulierten Roboter kann dann über das `Stage`-*plugin* auf die simulierten Geräte zugegriffen werden.
In Abbildung 2 wurde eine einfache Konstellation von Komponenten dargestellt. Die gleiche funktionale Simulation ist in Abbildung 5 dargestellt, nur wurde die Roboter-Komponente durch die `Stage`-Komponente mit einem simulierten Roboter ausgewechselt. Für die `Random Walk`-Komponente ändert sich nichts, nur in der Konfiguration des `Player Server` muss die Komponentenauswahl und -Verknüpfung angepasst werden. Dabei ist eine Schnittstelle hinzugekommen, die mit `S` beschriftete Instanz des `Simulation`-Interface von `Stage`. Dabei handelt es sich um eine in `Player` angelegte, aber dort noch nicht abschließend spezifizierte Schnittstelle. Sie kann also vorerst ignoriert werden.
Die in `Stage` simulierten Roboter können wie die anderen Komponenten in `Player` beschrieben und vernetzt werden, vorausgesetzt dass sie in der virtuellen Welt von `Stage` vorgesehen sind. Als kleine Besonderheit wird bei allen der gleiche `driver` (die `Player`-Bezeichung für Komponente), das `Stage`-*plugin*, angegeben.

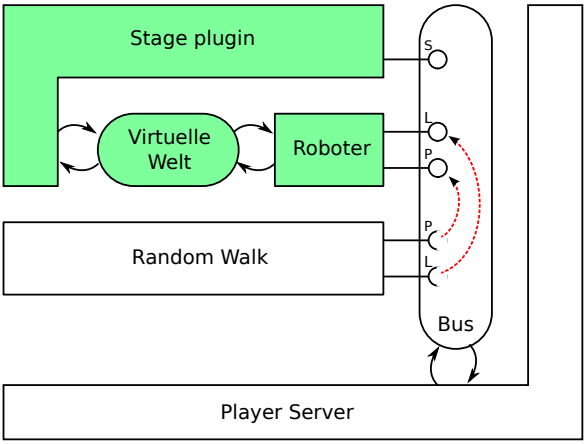

Abbildung 5: Der `Player Server` mit einem Stage-Roboter (FMC-Block)

Mit dieser einfachen Änderung der Konfiguration des `Player Server`s kann ohne Änderungen an den Algorithmen von einem simulierten Roboter auf ein (passendes) physisches System und zurück gewechselt werden, und das für alle vom Simulator unterstützten Schnittstellen. Einfacher geht es kaum!

4 Referenz

4.1 Allgemeine Typen in Player

4.1.1 Datentypen

Der am häufigsten verwendete Abstract Data Type (ADT) in `Player` ist `player_pose2d_t` [2] für die Repräsentation einer 2D-Pose bzw. `player_pose3d_t` für die 3D-Variante. Das in Listing 4 enthaltene Attribut `pa` für den Winkel in der X-Y-Ebene entspricht dem `pyaw` in Listing 5.

4.1.2 Nachrichtentypen

In den Schnittstellen von `Player` werden hauptsächlich drei Nachrichtentypen verwendet: *data*, *command* und *request*.

[2] Diese und einige andere Datentypen finden sich in `libplayerinterface/player.h` des Quellenverzeichnisses von `Player`.

```
/** @brief A pose in the plane */
typedef struct player_pose2d
{
  double px;  /** X [m] */
  double py;  /** Y [m] */
  double pa;  /** yaw [rad] */
} player_pose2d_t;
```

Listing 4: Player Datentyp `player_pose2d_t`

```
/** @brief A pose in space */
typedef struct player_pose3d
{
  double px;     /** X [m] */
  double py;     /** Y [m] */
  double pz;     /** Z [m] */
  double proll;  /** roll [rad] */
  double ppitch; /** pitch [rad] */
  double pyaw;   /** yaw [rad] */
} player_pose3d_t;
```

Listing 5: Player Datentyp `player_pose3d_t`

data Nachrichten, kodiert mit `PLAYER_MSGTYPE_DATA`, werden asynchron verschickt und in der Regel verwendet um neue Zustände bekannt zu machen.

command Nachrichten, kodiert mit `PLAYER_MSGTYPE_CMD`, werden asynchron verschickt und in der Regel verwendet um einen Zustand zu setzen.

request Nachrichten, kodiert mit `PLAYER_MSGTYPE_REQ` werden synchron verschickt und erfordern immer eine Antwort. Die Antwort kann, wie alle anderen Nachrichten eine Instanz eines Datentypen beinhalten.

```
/** position 2d velocity command */
typedef struct player_position2d_cmd_vel
{
  /** translational velocities [m/s,m/s,rad/s] (x, y, yaw)*/
  player_pose2d_t vel;
  /** Motor state (FALSE is either off or locked, depending on the
      driver). */
  uint8_t state;
} player_position2d_cmd_vel_t;
```

Listing 6: Player Position2d Interface - Geschwindigkeit Setzen label

PLAYER_POSITION2D_CMD_VEL
Der Datentyp für das PLAYER_POSITION2D_CMD_POS ist in Listing 7 aufgeführt.

```
/** position2d position command */
typedef struct player_position2d_cmd_pos
{
  /** position [m,m,rad] (x, y, yaw)*/
  player_pose2d_t pos;
  /** velocity at which to move to the position [m/s] or [rad/s] */
  player_pose2d_t vel;
  /** Motor state (FALSE is either off or locked, depending on the
      driver). */
  uint8_t state;
} player_position2d_cmd_pos_t;
```

Listing 7: Player Position2d Interface - Zielposition Setzen

```
/** position2d command setting velocity and steering angle */
typedef struct player_position2d_cmd_car
{
  /** forward velocity (m/s) */
  double velocity;
  /** relative turning angle (rad) */
  double angle;
} player_position2d_cmd_car_t;
```

Listing 8: Player Position2d Interface - Autoparameter Setzen

```
/** position2d command setting velocity and heading */
typedef struct player_position2d_cmd_vel_head
{
  /** forward velocity (m/s) */
  double velocity;
  /** absolute turning angle (rad) */
  double angle;
} player_position2d_cmd_vel_head_t;
```

Listing 9: Player Position2d Interface - Kurs Setzen

```
/** position2d data */
typedef struct player_position2d_data
{
  /** position [m,m,rad] (x, y, yaw)*/
  player_pose2d_t pos;
  /** translational velocities [m/s,m/s,rad/s] (x, y, yaw)*/
  player_pose2d_t vel;
  /** Are the motors stalled? */
  uint8_t stall;
} player_position2d_data_t;
```

Listing 10: Player Position2d Interface - Positionsdaten

Glossar

A*

A* ist ein Wegfinde Algorithmus, der den günstigsten Pfad von einem Startpunkt zu einem Zielpunkt auf einer Rasterkarte findet, indem er die nach einer Heuristik scheinbar günstigste Variante zuerst untersucht. Der Algorithmus ist aufgrund seiner Einfachheit und Geschwindigkeit nicht nur in der Robotik weit verbreitet.

application server

Ein *application server* ist ein Programm, welches Anwendungen dynamisch aus verschiedenen Komponenten zusammensetzen und über definierte Schnittstellen verfügbar machen kann. Dabei werden den Komponenten bestimmte Funktionen wie Persistenz oder Kommunikation als Teil eines *frameworks* angeboten.

Doxygen

Doxygen ist ein Werkzeug, welches Strukturen und Zusammenhänge aus Quellcode extrahiert und daraus mit Kommentaren eine Dokumentation in verschiedenen Ausgabeformaten generieren kann.

FMC

Fundamental Modeling Concepts (FMC) sind eine Beschreibungssprache und Methodik für Strukturierung und Veranschaulichung von komplexen Systemen, insbesondere von Softwaresystemen. Anders als bei der UML steht die Verständlichkeit und die Veranschaulichung von Konzepten und Architektur mittels Abstraktion im Vordergrund. Einen schnellen und informativen Überblick liefert [1], eine ausführliche Beschreibung mit Beispielen und Konzepten findet sich in [6]. In [8] werden unter anderem die Spezifikationen und Softwarewerkzeuge verfügbar gemacht.

`Gazebo`

`Gazebo` ist ein Roboter-Simulator, ausgelegt auf eine realistische Simulation von Robotern inklusive ihrer physikalischen Interaktionen mittels Festkörperdynamik. `Gazebo` beinhaltet eine GUI zur Darstellung und Manipulation

der virtuellen Welt und ist als Bibliothek oder als `Player`-Komponente verfügbar.

IPv4
Das *Internet Protocol* in der Version 4 ist ein in `RFC 791` definiertes Protokoll zur netzwerkübergreifenden Adressierung und Vermittlung von Datenpaketen ohne weitere Leistungen. Es formt (noch) die Grundlage des Internets.

JFIF
JPEG File Interchange Format (JFIF) ist ein Grafikformat für JPEG-komprimierte Rastergrafiken. Es legt als Komprimierungsmethode die Huffman-Kodierung, und als Farbraum YCbCr fest.

Joystick
Ein Joystick (Steuerknüppel) ist ein an einen Computer angeschlossenes Eingabegerät, das häufig zur Steuerung von (simulierten) Fahrzeugen verwendet wird.

JPEG
JPEG ist die nach ihrem Entwickler, der *Joint Photographic Experts Group*, benannte Norm `ISO/IEC 10918-1`. Sie beschreibt verschiedene Komprimierungs- und Kodierungsmethoden für zweidimensionale Rastergrafiken. Umgangssprachlich ist, wenn von einer `JPEG`-Datei gesprochen wird, eine Datei im JFIF gemeint.

Killough-Antrieb
Der Killough-Antrieb besteht aus drei starr montierten, separat ansteuerbaren Rädern, welche durch eine Rollenkonstruktion keine kinematische Zwangsbedingung in Achsenrichtung erzeugen. Durch diesen Antrieb entsteht ein System ohne anholonome Zwangsbedingungen, abgesehen von Massenträgheit, Schlupf und Motorleistung.

kinematische Zwangsbedingung
Eine kinematische Zwangsbedingung ist eine, meist durch den Antrieb eines Roboters vorgegebene Einschränkung der Bewegungsfreiheit innerhalb des Konfigurationsraums. Ein Differentialantrieb beispielsweise erlaubt nur Translationen in x-Richtung und Rotation um das kinematische Zentrum.

Komponentenmodell
Komponentenmodell ist ein Begriff aus der Softwaretechnik und bezeichnet eine Spezifikation von Struktur, Schnittstellen und Eigenschaften von Software-Komponenten, sowie deren Interaktions- und Kombinationsoptionen.

Konfigurationsraum
Der Konfigurationsraum eines bewegbaren Objekts oder Akteurs beschreibt die möglichen Orte und Orientierungen, die erreichbar sind. Dynamische Einschränkungen oder Singularitäten haben keinen Einfluss.

Lageregelung
Eine Lageregelung (oder Positionsregelung) ist ein System mit Gegenkopplung zur Kompensation von Abweichungen der Ist-Position eines bewegbaren Objekts gegenüber einer Soll-Position. Es wird häufig über eine Kaskadenregelung realisiert.

Laserentfernungsmesser
Ein Laserentfernungsmesser ist ein Gerät zur Distanzmessung zu einem Punkt, basierend auf Messungen der Laufzeit, Phasenverschiebung oder mittels Triangulation.

loopback-Interface
Das *loopback-interface* (auch *loopback-adapter*) ist eine bei vielen Computersystemen vorhandene virtuelle Netzwerkkarte, welche netzwerkfähigen Programmen auf dem gleichen System die Kommunikation untereinander ermöglicht, ohne große Performanceeinbußen oder Angriffsmöglichkeiten in Kauf zu nehmen. Bei IPv4 hat diese Schnittstelle die IP-Adresse `127.0.0.1`, bei IPv6 ist es `::1`.

Odometrie
Odometrie (auch Hodometrie, von griechisch hodoós-Weg und métron-Maß) bezeichnet die Verfahren, von der Beschleunigung, der Geschwindigkeit, dem Winkel oder der Winkeldifferenz der einzelnen Räder eines Antriebs über die Kinematik auf den zurückgelegten Weg zu schließen.

Partikel-Filter
Das Partikel-Filter repräsentiert eine Wahrscheinlichkeitsdichtefunktion, nicht über Funktionsparameter, sondern über Stichproben, sogenannte Partikel. Diese sind mit einer Gewichtung versehen und können regelmäßig neu gezogen werden, um die Auflösung zu verbessern oder die Partikelanzahl (und damit den Ressourcenaufwand) an die Komplexität der repräsentierten Verteilung anzupassen.

PID-Regler
Ein Standardregler, welcher den Proportional-, Integral- und Differentialanteil der Regelabweichung als Wirkelemente berücksichtigen kann. Eine genauere

Beschreibung des Aufbaus, der Funktion und der Optimierung solcher Regler findet sich in [7].

`Player`
: `Player` ist ein Robotik-*framework* und der namensgebende Teil des `Player Project`. Es spezifiziert diverse Kommunikations-Schnittstellen für Roboterkomponenten, stellt eine Kommunikationsinfrastruktur bereit und beinhaltet auch eine Reihe von Schnittstellenimplementierungen für verschiedene Sensoren, Aktoren und Algorithmen.

`Player Project`
: Das `Player Project` ist der organisatorische Rahmen zum Robotik-*framework* `Player` und den Roboter-Simulatoren `Stage` und `Gazebo`. Alle diese Teile sind als Open Source verfügbar [4].

plugin
: Als *plugin* wird ein Softwaremodul bezeichnet, welches ein bestehendes Programm ohne Neuübersetzung, oftmals auch zur Laufzeit, um Funktionalität erweitert oder bestehendes Verhalten ändert. Häufig werden *plugins* durch dynamisch nachladbare Bibliotheken (*shared libraries*) realisiert.

Pose
: Pose bezeichnet die Kombination von Position und Orientierung eines Objekts in seinem Konfigurationsraum.

Position Tracking
: Mit *position tracking* wird der Vorgang bezeichnet, bei dem eine globale Position inkrementell fortgeschrieben wird.

Rastergrafik
: Rastergrafiken sind Bilder, welche aus einer zweidimensionalen Sequenz von Farbwerten bestehen, welche jeweils in einem gleichmäßigen Rechteck oder Quadrat (Pixel) zu einem Bild zusammengesetzt werden.

Rasterkarte
: Eine Rasterkarte (oder Gitterkarte) bezeichnet eine meist zweidimensionale, örtlich diskrete Form einer Umgebungsrepräsentation. Die diskreten Bereiche der Karte werden häufig Zellen genannt und bilden die kleinste unterscheidbare Einheit. Solche Karten werden häufig als Rastergrafik gespeichert, wobei jedes Pixel eine Zelle repräsentiert, wobei in der Robotik häufig nur zwei Werte unterschieden werden: Einer für den Zustand "frei" und einer für ein durch ein Hindernis teilweise oder ganz belegte Zelle.

Software-Komponente
Eine Software-Komponente ist ein vollständig durch Schnittstellen gekapseltes Software-Modul mit definierten Abhängigkeiten, welches einem Komponentenmodell genügt und als Element mit anderen Komponenten zu einem System verknüpft werden kann.

`Stage`
`Stage` ist ein Roboter-Simulator ausgelegt auf die effektive Simulation von größeren Mengen von Robotern. `Stage` beinhaltet eine GUI zur Darstellung und Manipulation der virtuellen Welt und ist als Bibliothek oder als `Player`-Komponente verfügbar.

UML
Die Unified Modeling Language (UML) ist eine Spezifikationssprache für Software. Mit der UML lassen sich Abläufe und Strukturen als Diagramme in verschiedenen Varianten darstellen, und mit geeigneten Werkzeugen in Quellcode abbilden. Auch der umgekehrte Prozess ist möglich. So werden beispielsweise bei Doxygen extrahierte Strukturen als *Class-Diagram* und *Collaboration-Diagram* dargestellt.

Vektorkarte
Eine Vektorkarte repräsentiert eine Umgebung nicht mittels räumlicher Diskretisierung, sondern mit mathematischen Beschreibungen der Konturen und Positionen von Objekten.

VFH
Der VFH Algorithmus implementiert eine lokales, kollisionsvermeidendes Wegfindungsverfahren auf Basis eines zweidimensionalen Histogramm-Filters [2]. Der Algorithmus wurde durch VFH+ und VFH* verbessert.

VFH*
Die derzeitig letzte Version des VFH Algorithmus integriert A*, um zuvor bestimmte Richtungen zu bestätigen und ein global optimales Verhalten zu produzieren [10].

VFH+
Die verbesserte Version des VFH Algorithmus erweitert das Original um ein etwas weiter vorausschauendes Verhalten (Sackgassenvermeidung) und eine Kostenfunktion [9]. Mit VFH* wird es über einen lokalen Wegfindungsalgorithmus hinaus erweitert.

***Wavefront* Algorithmus**
: Der *Wavefront* Algorithmus ist ein Wegfinde Algorithmus, der auf einer Rasterkarte arbeitet. Er bestimmt iterativ, ausgehend von einem Zielpunkt, die geringste Entfernung zu allen an bereits bekannte Zellen angrenzenden Zellen, indem er für jede zu bestimmende Zelle das Entfernungs-Minimum der angrenzenden, bereits untersuchten Zellen ermittelt und um eins inkrementiert. Hindernis-Zellen werden dabei ignoriert. Der Algorithmus terminiert, wenn er die vorgegebene Startposition erreicht hat. Der Pfad entsteht, indem von einer Zelle immer in eine angrenzende Zelle gewechselt wird, deren ermittelter Entfernungswert geringer ist.

`Xfig`
: `Xfig` ist ein Programm zum Erstellen von Vektorgrafiken unter dem `X Window System`. Es unterstützt diverse Ausgabeformate und besitzt ein sehr einfaches, Textzeilen-basiertes Speicherformat. Ein besonderes Merkmal der Software ist die Möglichkeit, LaTeX-Beschriftungen in den Zeichnungen zu verwenden.

Acronyms

2D	twodimensional.
ADT	Abstract Data Type.
AMR	Autonomous Mobile Robot.
API	Application Programming Interface.
DGPS	Differential Global Positioning System.
FMC	Fundamental Modeling Concepts.
FOV	Field Of View.
GPS	Global Positioning System.
GUI	Graphical User Interface.
JFIF	JPEG File Interchange Format.
LAN	Local Area Network.

MCL	Monte Carlo Localization.
PC	Personal Computer.
PDF	Probability Density Function.
png	Portable Network Graphics.
SLAM	Simultanous Localization And Map building.
UML	Unified Modeling Language.
UPC	Universal Product Code.
USB	Universal Serial Bus.
VFH	Vector Field Histogram.

Literatur

[1] Fundamental modeling concepts. http://de.wikipedia.org/wiki/Fundamental_Modeling_Concepts, September 2010.

[2] Johann Borenstein and Y. Koren. The vector field histogram - fast obstacle avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 7(3):278–288, June 1991.

[3] Dieter Fox, Wolfram Burgard, Frank Dellaert, and Sebastian Thrun. Monte carlo localization: Efficient position estimation for mobile robots. In *AAAI-99 Proceedings*, 1999.

[4] Brian Gerkey, Richard T. Vaughan, and Andrew Howard. The player project. https://playerstage.org, 2010.

[5] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the International Conference on Advanced Robotics (ICAR)*, pages 317–323, July 2003.

[6] Andreas Knöpfel, Bernhard Groene, and Peter Tabeling. *Fundamental Modeling Concepts: Effective Communication of IT Systems*. John Wiley & Sons, 2006.

[7] Manfred Ottens. Einführung in die regelungstechnik. Skript zur Vorlesung, 2008.

[8] Peter Tabeling, Bernhard Gröne, and Andreas Knöpfel. Fundamental modeling concepts. http://www.fmc-modeling.org.

[9] Iwan Ulrich and Johann Borenstein. Vfh+: Reliable obstacle avoidance for fast mobile robots. In *Proceedings of the 1998 IEEE International Conference on Robotics and Automation*, pages 1572–1577, May 1998.

[10] Iwan Ulrich and Johann Borenstein. Vfh*: Local obstacle avoidance with look-ahead verification. In *IEEE International Conference on Robotics and Automation*, pages 1505–2511, April 2000.

[11] Richard T. Vaughan and Brian P. Gerkey. Really reusable robot code and the player/stage project. In *Software Engineering for Experimental Robotics*, pages 267–289. Springer-Verlag, 2006.

[12] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2121–2427, October 2003.