

UAS ROBOTIC

• Technical Report Chapter 1 : Introduction to ROS

Persyaratan teknis

Untuk mengikuti bab ini, satu-satunya yang Anda perlukan adalah komputer standar yang menjalankan Ubuntu 20.04 LTS atau distribusi Debian 10 GNU/Linux.

Mengapa kita harus menggunakan ROS?

Robot Operating System (ROS) adalah kerangka kerja fleksibel yang menyediakan berbagai alat dan perpustakaan untuk menulis perangkat lunak robot. Ia menawarkan beberapa fitur canggih untuk membantu pengembang dalam tugas-tugas seperti penyampaian pesan, komputasi terdistribusi, penggunaan kembali kode, dan implementasi algoritma canggih untuk aplikasi robotik. Proyek ROS dimulai pada tahun 2007 oleh Morgan Quigley dan pengembangannya dilanjutkan di Willow Garage, sebuah penelitian robotika laboratorium untuk mengembangkan perangkat keras dan perangkat lunak sumber terbuka untuk robot. Tujuan dari ROS adalah untuk menetapkan cara standar untuk memprogram robot sambil menawarkan perangkat lunak siap pakai komponen yang dapat dengan mudah diintegrasikan dengan aplikasi robotik khusus. Ada banyak alasan memilih ROS sebagai framework pemrograman, dan beberapa di antaranya adalah sebagai berikut :

- Kemampuan kelas atas: ROS hadir dengan fungsionalitas yang siap digunakan. Misalnya, paket Simultaneous Localization and Mapping (SLAM) dan Adaptive Monte Carlo Localization (AMCL) di ROS dapat digunakan untuk navigasi otonom pada robot bergerak, sedangkan paket MoveIt dapat digunakan untuk perencanaan gerak bagi manipulator robot. Kemampuan ini bisa langsung digunakan di kita

perangkat lunak robot tanpa kerumitan. Dalam beberapa kasus, paket ini cukup untuk menjalankan tugas robotika inti pada platform berbeda. Selain itu, kemampuan ini sangat dapat dikonfigurasi; kita dapat menyempurnakan masing-masing menggunakan berbagai parameter.

- Banyak alat: Ekosistem ROS dilengkapi dengan banyak alat untuk debugging, visualisasi, dan simulasi. Alat-alat tersebut, seperti `rqt_gui`, `RViz`, dan `Gazebo`, adalah beberapa alat sumber terbuka terkuat untuk debugging, visualisasi, dan simulasi. Kerangka perangkat lunak yang memiliki alat sebanyak ini sangatlah jarang.

- Dukungan untuk sensor dan aktuator kelas atas: ROS memungkinkan kita menggunakan driver perangkat yang berbeda dan paket antarmuka berbagai sensor dan aktuator dalam robotika. Sensor kelas atas tersebut mencakup LIDAR 3D, pemindai laser, sensor kedalaman, aktuator, dan banyak lagi. Kami dapat menghubungkan komponen-komponen ini dengan ROS tanpa kesulitan.

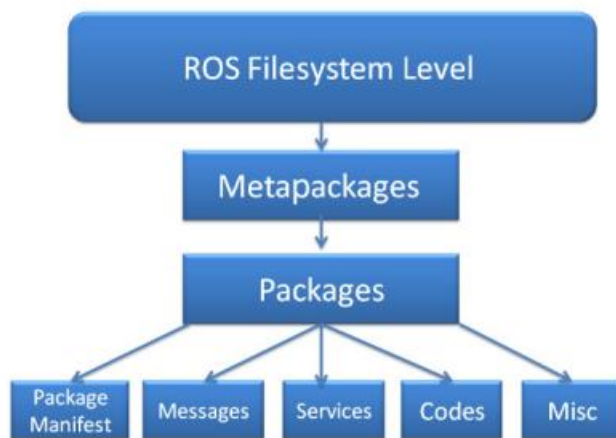
- Pengoperasian antar-platform: Middleware penyampaian pesan ROS memungkinkan komunikasi antar program yang berbeda. Di ROS, middleware ini dikenal sebagai node. Node ini dapat diprogram

dalam bahasa apa pun yang memiliki perpustakaan klien ROS. Kita dapat menulis node dengan high-level di C++ atau C dan node lainnya dengan Python atau Java.

- **Modularitas:** Salah satu masalah yang dapat terjadi pada sebagian besar aplikasi robot yang berdiri sendiri adalah jika salah satu thread kode utama mengalami error, seluruh aplikasi robot dapat berhenti. Di ROS, situasinya berbeda; kami menulis node yang berbeda untuk setiap proses, dan jika satu node mogok, sistem masih dapat bekerja. Memahami sistem file ROS level 5
- **Penanganan sumber daya secara bersamaan:** Menangani sumber daya perangkat keras melalui lebih dari dua proses selalu memusingkan. Bayangkan kita ingin memproses gambar dari kamera untuk deteksi wajah dan deteksi gerakan; kita dapat menulis kode sebagai satu kesatuan yang dapat melakukan keduanya, atau kita dapat menulis sepotong kode berulir tunggal untuk konkurensi. Jika kita ingin menambahkan lebih dari dua fitur ke thread, perilaku aplikasi akan menjadi rumit dan sulit untuk di-debug. Namun di ROS, kita dapat mengakses perangkat menggunakan topik ROS dari driver ROS. Node ROS dalam jumlah berapa pun dapat berlangganan pesan gambar dari driver kamera ROS, dan setiap node dapat memiliki fungsi yang berbeda. Hal ini dapat mengurangi kompleksitas dalam komputasi dan juga meningkatkan kemampuan debugging seluruh sistem.

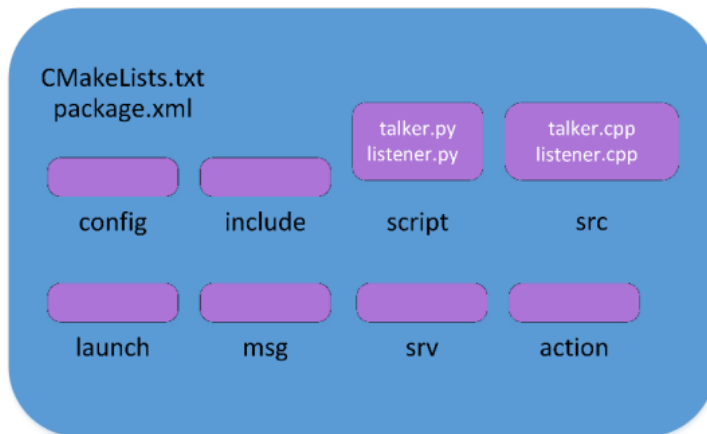
Memahami tingkat sistem file ROS

ROS lebih dari sekedar kerangka pengembangan. Kita dapat menyebut ROS sebagai meta-OS, karena ROS tidak hanya menawarkan alat dan perpustakaan tetapi bahkan fungsi mirip OS, seperti abstraksi perangkat keras, manajemen paket, dan rantai alat pengembang. Seperti sistem operasi sebenarnya, file ROS diatur pada hard disk dengan cara tertentu, seperti yang digambarkan dalam diagram berikut:



Paket ROS

Struktur khas paket ROS ditunjukkan di sini:



Definisi package.xml dalam paket tipikal ditunjukkan pada tangkapan layar berikut:

```
<?xml version="1.0"?>
<package>
  <name>hello_world</name>
  <version>0.0.1</version>
  <description>The hello_world package</description>
  <maintainer email="jonathan.cacace@gmail.com">Jonathan Cacace</maintainer>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>

  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
  </export>
</package>
```

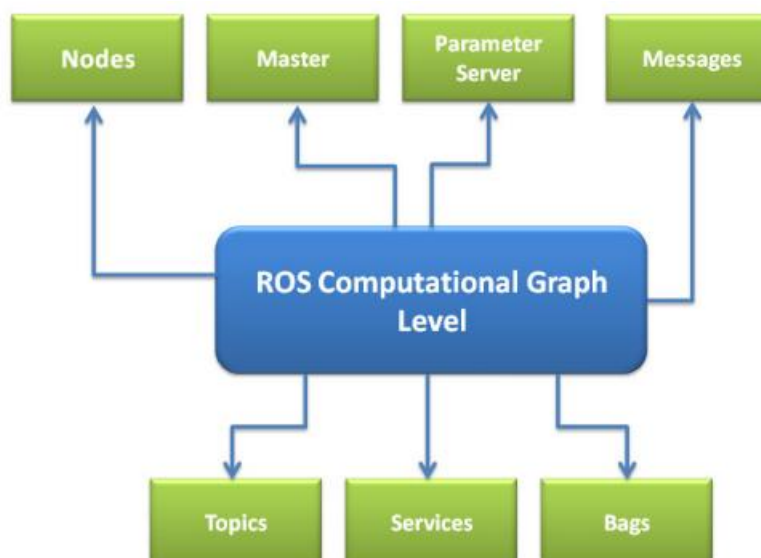
Pesan ROS

Node ROS dapat menulis atau membaca data dari berbagai jenis. Berbagai jenis data ini dijelaskan menggunakan bahasa deskripsi pesan yang disederhanakan, juga disebut pesan ROS. Deskripsi tipe data ini dapat digunakan untuk menghasilkan kode sumber untuk tipe pesan yang sesuai dalam bahasa target yang berbeda.

Primitive type	Serialization	C++	Python
bool (1)	Unsigned 8-bit int	uint8_t (2)	bool
int8	Signed 8-bit int	int8_t	int
uint8	Unsigned 8-bit int	uint8_t	int (3)
int16	Signed 16-bit int	int16_t	int
uint16	Unsigned 16-bit int	uint16_t	int
int32	Signed 32-bit int	int32_t	int
uint32	Unsigned 32-bit int	uint32_t	int
int64	Signed 64-bit int	int64_t	long
uint64	Unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string(4)	std::string	string
time	secs/nsecs unsigned 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Memahami level grafik komputasi ROS








Perhitungan dalam ROS dilakukan dengan menggunakan jaringan node ROS. Jaringan komputasi ini disebut grafik komputasi. Konsep utama dalam grafik komputasi adalah node ROS, master, server parameter, pesan, topik, layanan, dan tas. Setiap konsep dalam grafik dikonstruksikan ke grafik ini dengan cara yang berbeda. Paket terkait komunikasi ROS, termasuk pustaka klien inti, seperti roscpp dan rospy, dan implementasi konsep, seperti topik, node, parameter, dan layanan, adalah termasuk dalam tumpukan yang disebut `ros_comm` (http://wiki.ros.org/ros_comm). Tumpukan ini juga terdiri dari alat-alat seperti rostopic, rosparam, rosservice, dan rosnodet untuk melakukan introspeksi konsep-konsep sebelumnya. Tumpukan `ros_comm` berisi paket middleware komunikasi ROS, dan paket-paket ini secara kolektif disebut lapisan grafik ROS:



Distribusi ROS

Pembaruan ROS dirilis dengan distribusi ROS baru. Distribusi ROS yang baru terdiri dari versi terbaru dari perangkat lunak intinya dan serangkaian paket ROS baru/yang diperbarui. ROS mengikuti siklus rilis yang sama dengan distribusi Linux Ubuntu: versi baru ROS dirilis setiap 6 bulan. Biasanya, untuk

setiap versi Ubuntu LTS, versi ROS LTS dirilis. Dukungan Jangka Panjang (LTS) dan berarti perangkat lunak yang dirilis akan dipertahankan untuk waktu yang lama (5 tahun untuk ROS dan Ubuntu):

Distro	Release date	Poster	Turtle, turtle in tutorial	EOL date
ROS Noetic Ninjemys (Recommended)	May 23rd, 2020			May, 2025 (Focal EOL)
ROS Melodic Morenia	May 23rd, 2018			May, 2023 (Bionic EOL)
ROS Lunar Loggerhead	May 23rd, 2017			May, 2019
ROS Kinetic Kame	May 23rd, 2016			April, 2021 (Xenial EOL)

Ringkasan

ROS sekarang menjadi kerangka perangkat lunak yang sedang tren di kalangan robotika. Memperoleh pengetahuan tentang ROS akan sangat penting di tahun-tahun mendatang jika Anda berencana membangun karir Anda sebagai insinyur robotika. Dalam bab ini, kita telah mempelajari dasar-dasar ROS, terutama untuk menyegarkan Anda tentang konsep-konsep jika Anda sudah mempelajari tentang ROS. Kami membahas perlunya mempelajari ROS dan keunggulannya di antara platform perangkat lunak robotika saat ini. Kami membahas konsep dasar, seperti master ROS dan server parameter, serta memberikan penjelasan tentang cara kerja roscore. Pada bab selanjutnya, kami akan memperkenalkan manajemen paket ROS dan membahas beberapa contoh praktis sistem komunikasi ROS

- Technical Report Chapter 2 : Getting Started with ROS Programming

Persyaratan teknis

Untuk mengikuti bab ini, Anda memerlukan laptop standar yang menjalankan Ubuntu 20.04 dengan ROS Noetic terinstal. Kode referensi untuk bab ini dapat diunduh dari repositori GitHub berikut: <https://github.com/PacktPublishing/Mastering-ROS-for-Robotics-Programming-Third-edition.git>. Kode yang diperlukan terdapat dalam folder Bab2/mastering_ros_demo_pkg. Anda dapat melihat kode bab ini beraksi di sini: <https://bit.ly/3iXO5IW>.

Membuat paket ROS

Paket ROS adalah unit dasar program ROS. Kita dapat membuat paket ROS, membangunnya, dan merilisnya ke publik. Distribusi ROS yang kami gunakan saat ini adalah Noetic Ninjemys. Kami menggunakan sistem build catkin untuk membangun paket ROS. Sistem build bertanggung jawab untuk menghasilkan target (yang dapat dieksekusi/pustaka) dari kode sumber tekstual yang dapat digunakan oleh pengguna akhir. Dalam distribusi lama, seperti Electric dan Fuerte, rosbuilt adalah sistem pembangunannya. Karena berbagai kekurangan rosbuilt, catkin muncul. Hal ini juga memungkinkan kami untuk memindahkan sistem kompilasi ROS lebih dekat ke Cross Platform Make (CMake). Ini memiliki banyak keuntungan, seperti mem-porting paket ke OS lain, seperti Windows. Jika OS mendukung CMake dan Python, paket berbasis catkin dapat di-porting ke OS tersebut.

Persyaratan pertama untuk bekerja dengan paket ROS adalah membuat catkinworkspace ROS. Setelah menginstal ROS, kita dapat membuat dan membangun catkinworkspace bernama catkin_ws:

```
mkdir -p ~/catkin_ws/src
```

Untuk mengkompilasi ruang kerja ini, kita harus mencari lingkungan ROS untuk mendapatkan akses ke fungsi ROS:

```
sumber /opt/ros/noetic/setup.bash
```

Beralih ke folder source src yang kita buat sebelumnya:

```
cd ~/catkin_ws/src
```

Inisialisasi ruang kerja catkin baru:

```
catkin_init_workspace
```

Kita dapat membangun ruang kerja meskipun tidak ada paket. Kita dapat menggunakan perintah berikut untuk beralih ke folder ruang kerja:

```
cd ~/catkin_ws
```

Perintah catkin_make akan membangun ruang kerja berikut:

```
catkin_make
```

Perintah ini akan membuat direktori devel dan build di ruang kerja catkin Anda. File setup yang berbeda terletak di dalam folder devel. Untuk menambahkan ruang kerja ROS yang dibuat ke lingkungan ROS, kita harus mengambil sumber salah satu file berikut. Selain itu, kita dapat mengambil sumber file setup ruang kerja ini setiap kali sesi bash baru dimulai dengan

perintah berikut:

```
echo "sumber ~/catkin_ws/devel/setup.bash" >> ~/.bashrc sumber ~/.bashrc
```

Setelah mengatur ruang kerja catkin, kita dapat membuat paket kita sendiri yang memiliki node sampel untuk mendemonstrasikan cara kerja topik, pesan, layanan, dan actionlib ROS. Perhatikan bahwa jika Anda belum menyiapkan ruang kerja dengan benar, maka Anda tidak akan dapat mengaturnya. dapat menggunakan perintah ROS apa pun. Perintah `catkin_create_pkg` adalah cara paling mudah untuk membuat paket ROS. Perintah ini digunakan untuk membuat paket kita, di mana kita akan membuat demo berbagai konsep ROS.

Beralih ke folder `src` ruang kerja catkin dan buat paket dengan menggunakan perintah berikut:

```
catkin_create_pkg nama_paket [ketergantungan1] [ketergantungan2]
```

Folder kode sumber: Semua paket ROS, baik yang dibuat dari awal atau diunduh dari repositori kode lain, harus ditempatkan di folder `src` ruang kerja ROS; jika tidak, data tersebut tidak akan dikenali oleh sistem ROS dan dikompilasi.

Berikut adalah perintah untuk membuat contoh paket ROS:

```
catkin_create_pkg mastering_ros_demo_pkg roscpp std_msgs actionlib actionlib_msgs
```

Dependensi dalam paket ini adalah sebagai berikut:

- `roscpp`: Ini adalah implementasi C++ dari ROS. Ini adalah pustaka klien ROS yang menyediakan API kepada pengembang C++ untuk membuat node ROS dengan topik, layanan, parameter ROS, dan sebagainya. Kami menyertakan ketergantungan ini karena kami akan menulis node ROS C++. Paket ROS apa pun yang menggunakan node C++ harus menambahkan ketergantungan ini.
- `std_msgs`: Paket ini berisi tipe data primitif ROS dasar, seperti integer, float, string, array, dan seterusnya. Kita bisa langsung menggunakan tipe data ini di node kita tanpa mendefinisikan pesan ROS baru.
- `actionlib`: Metapackage `actionlib` menyediakan antarmuka untuk dibuat

tugas yang dapat diakhiri di node ROS. Kami membuat node berbasis `actionlib` dalam paket ini. Jadi, kita harus menyertakan paket ini untuk membangun node ROS.

- `actionlib_msgs`: Paket ini berisi definisi pesan standar yang diperlukan untuk berinteraksi dengan server tindakan dan klien tindakan.

Setelah pembuatan paket, dependensi tambahan dapat ditambahkan secara manual dengan mengedit file `CMakeLists.txt` dan `package.xml`. Kita akan mendapatkan pesan berikut jika paket telah berhasil dibuat:

```
Created file mastering_ros_v2_pkg/package.xml
Created file mastering_ros_v2_pkg/CMakeLists.txt
Created folder mastering_ros_v2_pkg/include/mastering_ros_v2_pkg
Created folder mastering_ros_v2_pkg/src
Successfully created files in /home/jcacace/mastering_ros_v2_pkg. Please
adjust the values in package.xml.
```

Setelah membuat paket ini, buat paket tanpa menambahkan node apa pun dengan menggunakan perintah `catkin_make`. Perintah ini harus dijalankan dari jalur ruang kerja catkin. Perintah berikut menunjukkan cara membuat paket ROS kosong kami:

```
cd ~/catkin_ws && catkin_make
```

Setelah build berhasil, kita dapat mulai menambahkan node ke folder src paket ini.

Folder build dalam file build CMake terutama berisi executable dari node yang ditempatkan di dalam folder src ruang kerja catkin. Folder devel berisi skrip Bash, file header, dan file executable di berbagai folder yang dihasilkan selama proses build. Kita telah melihat cara membuat dan mengkompilasi node ROS menggunakan catkin_make. Sekarang mari kita bahas cara bekerja dengan topik ROS.

Membuat node ROS

Node pertama yang akan kita bahas adalah demo_topic_publisher.cpp. Node ini akan menerbitkan nilai integer pada topik yang disebut /numbers. Salin kode saat ini ke dalam paket baru atau gunakan file yang sudah ada dari repositori kode buku ini. Berikut kode lengkapnya:

```
#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

int main(int argc, char **argv) {
    ros::init(argc, argv, "demo_topic_publisher");
    ros::NodeHandle node_obj;
    ros::Publisher number_publisher = node_obj.advertise<std_
msgs::Int32>("/numbers", 10);
    ros::Rate loop_rate(10);
    int number_count = 0;
    while ( ros::ok() ) {
        std_msgs::Int32 msg;
        msg.data = number_count;
        ROS_INFO("%d",msg.data);
        number_publisher.publish(msg);
        loop_rate.sleep();
        ++number_count;
    }
    return 0;
}
```

Kode dimulai dengan definisi file header. Secara khusus, ros/ros.h adalah header utama ROS. Jika kita ingin menggunakan API klien roscpp dalam kode kita, kita harus menyertakan header ini. std_msgs/Int32.h adalah definisi pesan standar dari tipe data integer.

Di sini, kami mengirimkan nilai integer melalui suatu topik. Jadi, kita memerlukan tipe pesan untuk menangani data integer. std_msgs berisi definisi pesan standar tipe data primitif, sedangkan std_msgs/Int32.h berisi definisi pesan integer. Sekarang, kita dapat menginisialisasi node ROS dengan sebuah nama. Perlu dicatat bahwa node ROS harus unik:

```
ros::init(argc, argv, "demo_topic_publisher");
```


Selanjutnya kita perlu membuat objek Nodehandle yang digunakan untuk berkomunikasi dengan sistem ROS. Baris ini wajib untuk semua node ROS C++:

```
ros::NodeHandle node_obj;
```

Baris berikut membuat penerbit topik dan memberi nama topik `"/numbers"` dengan tipe pesan `std_msgs::Int32`. Argumen kedua adalah ukuran buffer. Ini menunjukkan berapa banyak pesan yang disimpan dalam buffer jika penerbit tidak dapat mempublikasikan data dengan cukup cepat. Jumlah ini harus ditetapkan dengan mempertimbangkan tingkat penerbitan pesan. Jika program Anda dipublikasikan lebih cepat dari ukuran antrean, beberapa pesan akan dihapus. Angka terendah yang diterima untuk ukuran antrean adalah 1, sedangkan 0 berarti antrean tak terhingga:

```
ros::Publisher number_publisher = node_obj.advertise
```

```
msgs::Int32>("/numbers", 10);
```

Kode berikut digunakan untuk mengatur frekuensi loop utama program dan, akibatnya, kecepatan penerbitan dalam kasus kita:

```
ros::Rate loop_rate(10);
```

Ini adalah perulangan while tak terbatas, dan berhenti saat kita menekan Ctrl + C. Fungsi `ros::ok()` mengembalikan nol saat ada interupsi. Ini dapat menghentikan perulangan while ini:

```
while ( ros::ok() ) {
```

Baris berikut membuat pesan ROS bilangan bulat, memberinya nilai bilangan bulat. Di sini, data adalah nama field dari objek pesan:

```
std_msgs::Int32 msg;
```

```
msg.data = number_count;
```

Ini akan mencetak data pesan. Baris berikut digunakan untuk mencatat informasi ROS dan mempublikasikan pesan sebelumnya ke jaringan ROS:

```
ROS_INFO("%d",msg.data);
```

```
number_publisher.publish(msg);
```

Terakhir, baris ini akan memberikan penundaan yang diperlukan untuk mencapai frekuensi 10 Hz:

```
loop_rate.sleep();
```

Sekarang kita sudah membahas node penerbit, kita bisa membahas node pelanggan, yaitu `demo_topic_subscriber.cpp`.

Berikut adalah definisi dari node pelanggan:

```

#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include <iostream>

void number_callback(const std_msgs::Int32::ConstPtr& msg) {
    ROS_INFO("Received [%d]", msg->data);
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "demo_topic_subscriber");
    ros::NodeHandle node_obj;
    ros::Subscriber number_subscriber = node_obj.subscribe("/
numbers", 10, number_callback);
    ros::spin();
    return 0;
}

```

Seperti sebelumnya, kode dimulai dengan definisi file header. Kemudian, kami mengembangkan fungsi panggilan balik, yang akan dijalankan setiap kali pesan ROS datang ke topik /numbers. Setiap kali data mencapai topik ini, fungsi akan memanggil dan mengekstrak nilainya dan mencetaknya ke konsol:

```

void number_callback(const std_msgs::Int32::ConstPtr& msg) {
    ROS_INFO("Received [%d]", msg->data);
}

```

Ini adalah definisi pelanggan, dan di sini, kami memberikan nama topik yang diperlukan untuk berlangganan, ukuran buffer, dan fungsi panggilan balik. Kami juga berlangganan topik /numbers. Kita melihat fungsi panggilan balik di bagian sebelumnya:

```

ros::Subscriber number_subscriber = node_obj.subscribe("/
numbers", 10, number_callback);

```

Ini adalah loop tak terbatas di mana node akan menunggu pada langkah ini. Kode ini akan mempercepat callback setiap kali data mencapai topik dan akan berhenti hanya ketika kita menekan Ctrl + C:

```
ros::spin();
```

Sekarang kodenya sudah selesai. Sebelum kita menjalankannya, kita perlu mengkompilasinya seperti yang dibahas di bagian selanjutnya.

Ringkasan

Dalam bab ini, kami memberikan berbagai contoh node ROS yang menerapkan fitur ROS seperti topik, layanan, dan tindakan ROS. Alat tersebut digunakan di setiap paket ROS, baik yang sudah tersedia di repositori ROS maupun yang Anda buat. Kami juga membahas cara membuat dan mengkompilasi paket ROS menggunakan pesan khusus dan standar. Biasanya, paket yang berbeda menggunakan pesan khusus untuk menangani data yang dihasilkan oleh nodenya, jadi penting untuk dapat mengelola pesan khusus yang disediakan oleh sebuah paket.

• Technical Report Chapter 3 : Working with ROS for 3D Modeling

Paket ROS untuk pemodelan robot

ROS menyediakan beberapa paket bagus yang dapat digunakan untuk membangun model robot 3D. Pada bagian ini, kita akan membahas beberapa paket ROS penting yang biasa digunakan untuk membuat dan memodelkan robot:

- **urdf**: Paket ROS yang paling penting untuk memodelkan robot adalah paket urdf. Paket ini berisi parser C++ untuk URDF, yang merupakan file XML yang mewakili model robot. Komponen berbeda lainnya membentuk urdf, seperti berikut:

A. **urdf_parser_plugin**: Paket ini mengimplementasikan metode untuk mengisi struktur data URDF.

B. **urdfdom_headers**: Komponen ini menyediakan header struktur data inti untuk menggunakan parser urdf.

C. **collada_parser**: Paket ini mengisi struktur data dengan mengurai file Collada.

D. **urdfdom**: Komponen ini mengisi struktur data dengan mengurai file URDF.

Kita dapat menentukan model robot, sensor, dan lingkungan kerja menggunakan URDF. Kami juga dapat menguraikannya menggunakan parser URDF. Kami hanya dapat mendeskripsikan robot dalam URDF yang memiliki struktur seperti pohon pada tautannya, yaitu robot tersebut akan memiliki tautan yang kaku dan akan dihubungkan menggunakan sambungan. Tautan fleksibel tidak dapat direpresentasikan menggunakan URDF. URDF dibuat menggunakan tag XML khusus, dan kita dapat mengurai tag XML ini menggunakan program parser untuk diproses lebih lanjut. Sebelum mengerjakan pemodelan URDF, mari kita definisikan beberapa paket ROS yang menggunakan file model robot:

- **joint_state_publisher**: Alat ini sangat berguna ketika merancang model robot menggunakan URDF. Paket ini berisi node bernama `joint_state_publisher`,

yang membaca deskripsi model robot, menemukan semua sambungan, dan menerbitkan nilai sambungan ke semua sambungan tidak tetap. Sumber berbeda untuk nilai masing-masing sendi juga tersedia. Kami akan membahas paket ini dan penggunaannya secara lebih rinci di bagian selanjutnya.

- **joint_state_publisher_gui**: Alat ini sangat mirip dengan `joint_`

`state_publisher`. Ia menawarkan fungsionalitas yang sama dengan `joint_`

`state_publisher` dan, selain itu, mengimplementasikan serangkaian penggeser yang dapat digunakan oleh pengguna untuk berinteraksi dengan setiap sambungan robot yang memvisualisasikan keluaran menggunakan RViz. Dalam hal ini, sumber nilai gabungan adalah GUI penggeser. Saat mendesain URDF, pengguna dapat memverifikasi rotasi dan translasi setiap sambungan menggunakan alat ini.

- **kdl_parser**: Paket ini berisi alat parser untuk membangun pohon Kinematic and Dynamic Library (KDL) dari model robot URDF. KDL adalah perpustakaan yang digunakan untuk menyelesaikan masalah kinematik dan dinamis.

- **robot_state_publisher**: Paket ini membaca status gabungan robot saat ini dan menerbitkan pose 3D setiap tautan robot menggunakan pohon kinematika yang dibuat dari URDF. Pose 3D robot dipublikasikan sebagai `tf` (transform) ROS. `tf` ROS mempublikasikan hubungan antara kerangka koordinat robot.

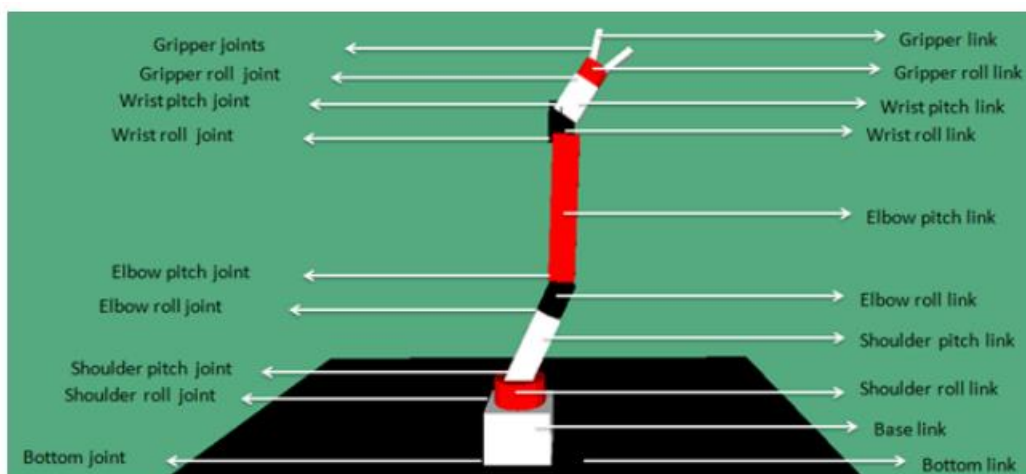
- xacro: Xacro adalah singkatan dari XML Macros, dan kita dapat menganggap file xacro sebagai file URDF dengan beberapa add-on. Ini berisi beberapa add-on untuk membuat URDF lebih pendek dan lebih mudah dibaca serta dapat digunakan untuk membuat deskripsi robot yang kompleks. Kami dapat mengubah xacro menjadi URDF kapan saja menggunakan alat ROS. Kita akan mempelajari lebih lanjut tentang xacro dan penggunaannya di bagian selanjutnya.

Sekarang kita telah menentukan daftar paket yang terlibat dalam pemodelan 3D robot, kita siap menganalisis model pertama kita menggunakan format file URDF.

Membuat deskripsi robot untuk manipulator robot tujuh DOF

Sekarang, kita dapat membuat beberapa robot kompleks menggunakan URDF dan xacro. Robot pertama yang akan kita bahas adalah lengan robot tujuh DOF, yang merupakan manipulator tautan serial dengan banyak tautan serial. Lengan tujuh DOF secara kinematis berlebihan, yang berarti ia memiliki lebih banyak sambungan dan DOF daripada yang dibutuhkan untuk mencapai posisi dan orientasi sasarannya. Keuntungan dari manipulator redundan adalah kita dapat memiliki lebih banyak konfigurasi gabungan untuk posisi dan orientasi tujuan yang diinginkan. Hal ini akan meningkatkan fleksibilitas dan keserbagunaan gerakan robot serta dapat menerapkan gerakan bebas tabrakan yang efektif di ruang kerja robot.

Mari kita mulai dengan membuat lengan tujuh DOF; model keluaran akhir dari lengan robot ditampilkan di sini (berbagai sambungan dan tautan pada robot juga ditandai pada diagram):



Robot sebelumnya dijelaskan menggunakan xacro. Kita dapat mengambil file deskripsi sebenarnya dari repositori hasil kloning. Kita dapat menavigasi ke folder urdf di dalam paket kloning dan membuka file `seven_dof_arm.xacro`. Kami akan menyalin dan menempelkan deskripsi ke paket saat ini dan mendiskusikan aspek utama dari deskripsi robot ini. Sebelum kita membuat file model robot, mari laporkan beberapa spesifikasi lengan robot

Spesifikasi lengan

Dalam daftar berikut, karakteristik lengan tujuh DOF dilaporkan:

- Derajat kebebasan: 7
- Panjang lengan : 50 cm
- Jangkauan lengan: 35 cm
- Jumlah tautan: 12

- Jumlah sambungan: 11

Seperti yang Anda lihat, kita dapat mendefinisikan berbagai jenis sambungan. Sekarang mari kita bahas jenis sambungan lengan tujuh DOF.

Jenis sendi

Berikut daftar joint yang memuat nama joint dan tipe robotnya:

Joint number	Joint name	Joint type	Joint limits
1	bottom_joint	Fixed	--
2	shoulder_pan_joint	Revolute	-150° to 114°
3	shoulder_pitch_joint	Revolute	-67° to 109°
4	elbow_roll_joint	Revolute	-150° to 41°
5	elbow_pitch_joint	Revolute	-92° to 110°
6	wrist_roll_joint	Revolute	-150° to 150°
7	wrist_pitch_joint	Revolute	92° to 113°
8	gripper_roll_joint	Revolute	-150° to 150°
9	finger_joint1	Prismatic	0 cm to 3 cm
10	finger_joint2	Prismatic	0 cm to 3 cm

Ringkasan

Dalam bab ini, kita terutama melihat pentingnya pemodelan robot dan bagaimana kita dapat memodelkan robot di ROS. Kami membahas paket yang digunakan di ROS untuk memodelkan struktur robot, seperti urdf, xacro, dan joint_state_publisher serta GUI-nya. Kita membahas URDF, xacro, dan tag URDF utama yang dapat kita gunakan. Kami juga membuat model sampel dalam URDF dan xacro dan mendiskusikan perbedaan keduanya. Setelah ini, kami membuat manipulator robot kompleks dengan tujuh DOF dan melihat penggunaan paket joint_state_publisher dan robot_state_publisher. Di akhir bab, kami meninjau prosedur perancangan robot bergerak penggerak diferensial menggunakan xacro. Pada bab selanjutnya kita akan melihat simulasi robot-robot tersebut menggunakan Gazebo.

- **Technical Report Chapter 4 : Simulating Robots Using ROS and Gazebo**

Simulasi lengan robot menggunakan Gazebo dan ROS

Pada bab sebelumnya, kami merancang lengan tujuh DOF. Pada bagian ini, kita akan melakukan simulasi

robot di Gazebo menggunakan ROS.

Sebelum memulai dengan Gazebo dan ROS, kita harus menginstal paket berikut agar berfungsi dengan Gazebo dan ROS:

```
sudo apt-get install ros-noetic-gazebo-ros-pkgs ros-noetic-gazebo-msgs ros-noetic-gazebo-plugins ros-noetic-gazebo-ros-control
```

Versi default yang diinstal dari paket Noetic ROS adalah Gazebo 11.x. Kegunaannya masing-masing paketnya adalah sebagai berikut:

- gazebo_ros_pkgs: Berisi wrapper dan alat untuk menghubungkan ROS dengan Gazebo.
- gazebo-msgs: Berisi pesan dan struktur data layanan untuk antarmuka dengan Gazebo dari ROS.

Membuat model simulasi lengan robot untuk Gazebo 99

- gazebo-plugins: Ini berisi plugin Gazebo untuk sensor, aktuator, dan sebagainya.
- gazebo-ros-control: Ini berisi pengontrol standar untuk berkomunikasi antara ROS dan Gazebo.

Setelah instalasi, periksa apakah Gazebo sudah terpasang dengan benar menggunakan perintah berikut:

```
roscore & roslaunch gazebo_ros gazebo
```

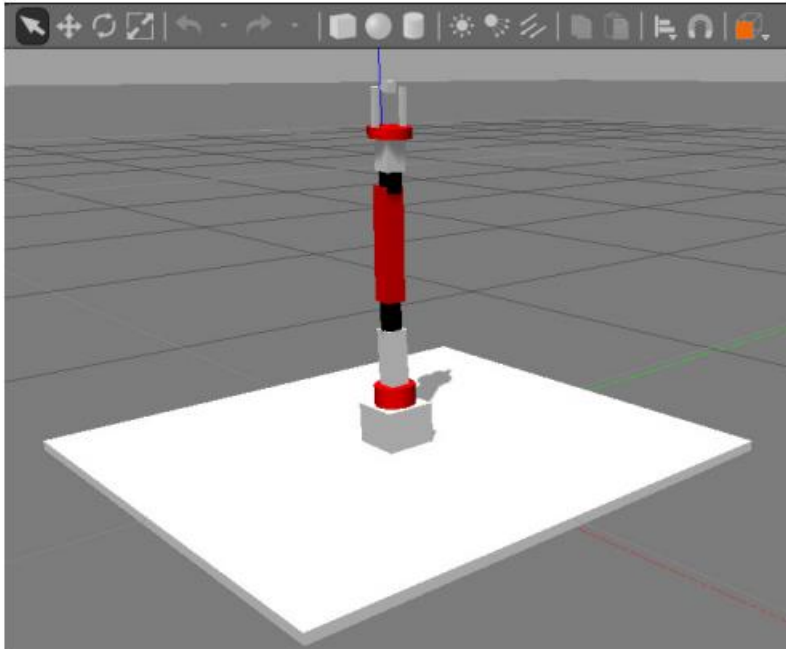
Perintah ini akan membuka Gazebo GUI. Jika kita memiliki simulator Gazebo, kita dapat melanjutkan untuk mengembangkan model simulasi tujuh lengan DOF untuk Gazebo.

Membuat model simulasi lengan robot untuk Gazebo

Kita dapat membuat model simulasi lengan robot dengan memperbarui deskripsi robot yang ada dengan menambahkan parameter simulasi.

Kita dapat membuat paket yang diperlukan untuk mensimulasikan lengan robot menggunakan perintah berikut:

```
catkin_create_pkg seven_dof_arm_gazebo gazebo_msgs gazebo_
plugins gazebo_ros gazebo_ros_control mastering_ros_robot_
description_pkg
```



Menambahkan warna dan tekstur pada model robot Gazebo

Pada simulasi robot dapat kita lihat bahwa setiap link mempunyai warna dan tekstur yang berbeda. Tag berikut di dalam file .xacro memberikan tekstur dan warna pada tautan robot:

```
<gazebo reference="bottom_link">
  <material>Gazebo/White</material>
</gazebo>
<gazebo reference="base_link">
  <material>Gazebo/White</material>
</gazebo>
<gazebo reference="shoulder_pan_link">
  <material>Gazebo/Red</material>
</gazebo>
```

Menambahkan tag transmisi untuk menggerakkan model

Untuk menggerakkan robot menggunakan pengontrol ROS, kita harus mendefinisikan elemen `<transmission>` untuk menghubungkan aktuator ke sambungan. Berikut adalah makro yang ditentukan untuk transmisi:

```

<xacro:macro name="transmission_block" params="joint_name">
  <transmission name="tran1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="${joint_name}">
      <hardwareInterface>hardware_interface/
PositionJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
</xacro:macro>

```

Ringkasan

Dalam bab ini, kami mencoba mensimulasikan dua robot: satu adalah lengan robot dengan tujuh DOF, dan yang lainnya adalah robot bergerak beroda diferensial. Kami memulai dengan lengan robot dan mendiskusikan tag Gazebo tambahan yang diperlukan untuk meluncurkan robot di Gazebo. Kami membahas cara menambahkan sensor penglihatan 3D ke simulasi. Kemudian, kami membuat file peluncuran untuk memulai Gazebo dengan lengan robot dan mendiskusikan cara menambahkan pengontrol ke setiap sambungan. Kami menambahkan pengontrol dan mengerjakan setiap sambungan.

Seperti lengan robot, kami membuat URDF untuk simulasi Gazebo dan menambahkan plugin Gazebo-ROS yang diperlukan untuk pemindai laser dan mekanisme penggerak diferensial. Setelah menyelesaikan model simulasi, kami meluncurkan simulasi menggunakan file peluncuran khusus. Terakhir, kita melihat cara menggerakkan robot menggunakan teleop node.

Kita dapat mempelajari lebih lanjut tentang lengan robot dan robot bergerak yang didukung oleh ROS di <http://wiki.ros.org/Robots>.

- **Technical Report Chapter 5 : Simulating Robots Using ROS, CoppeliaSim, and Webots**

Menyiapkan CoppeliaSim dengan ROS

Sebelum mulai bekerja dengan CoppeliaSim, kita perlu menginstalnya di sistem kita dan mengkonfigurasi lingkungan kita untuk memulai jembatan komunikasi antara ROS dan adegan simulasi. CoppeliaSim adalah perangkat lunak lintas platform, tersedia untuk berbagai sistem operasi seperti Windows, macOS, dan Linux. Ini dikembangkan oleh Coppelia Robotics GmbH dan didistribusikan dengan lisensi pendidikan dan komersial gratis. Unduh versi terbaru simulator CoppeliaSim dari halaman unduh Coppelia Robotics di <http://www.coppeliarobotics.com/downloads.html>, pilih eduversion untuk Linux. Pada bab ini, kita akan mengacu pada versi CoppeliaSim 4.2.0.

Setelah selesai mengunduh, ekstrak arsipnya. Pindah ke folder unduhan Anda dan gunakan perintah berikut:

```
tar vxf CoppeliaSim_Edu_V4_2_0_Ubuntu20_04.tar.xz
```

Versi ini didukung oleh Ubuntu versi 20.04. Akan lebih mudah untuk mengganti nama folder ini dengan sesuatu yang lebih intuitif, seperti ini:

```
mv CoppeliaSim_Edu_V4_2_0_Ubuntu20_04 CoppeliaSim
```

Untuk mengakses sumber daya CoppeliaSim dengan mudah, akan lebih mudah untuk mengatur variabel lingkungan COPPELIASIM_ROOT yang menunjuk ke folder utama CoppeliaSim, seperti ini:

```
gema "ekspor COPPELIASIM_ROOT=/path/ke/CoppeliaSim/folder >>
```

```
~/bashrc"
```

Di sini, /path/to/CoppeliaSim/folder adalah jalur absolut ke folder yang diekstraksi. CoppeliaSim menawarkan mode berikut untuk mengontrol robot simulasi dari aplikasi eksternal:

- Antarmuka pemrograman aplikasi jarak jauh (API): API jarak jauh CoppeliaSim terdiri dari beberapa fungsi yang dapat dipanggil dari aplikasi eksternal yang dikembangkan dalam C/C++, Python, Lua, atau MATLAB. API jarak jauh berinteraksi dengan CoppeliaSim melalui jaringan, menggunakan komunikasi soket. Anda dapat mengintegrasikan API jarak jauh di node C++ atau Python untuk menghubungkan ROS dengan adegan simulasi. Daftar semua API jarak jauh yang tersedia di CoppeliaSim dapat ditemukan di situs web Coppelia Robotics, di <https://www.coppeliarobotics.com/>

[helpFiles/en/remoteApiFunctionsMatlab.htm](#). Untuk menggunakan API jarak jauh, Anda harus mengimplementasikan sisi klien dan server, sebagai berikut:

A. Klien CoppeliaSim: Sisi klien berada di aplikasi eksternal. Hal ini dapat diimplementasikan dalam node ROS atau dalam program standar yang ditulis dalam salah satu bahasa pemrograman yang didukung.

B. Server CoppeliaSim: Sisi ini diimplementasikan dalam skrip CoppeliaSim dan memungkinkan simulator menerima data eksternal untuk berinteraksi dengan adegan simulasi.

- RosInterface: Ini adalah antarmuka saat ini untuk memungkinkan komunikasi antara ROS dan CoppeliaSim. Di masa lalu, plugin ROS digunakan, tetapi sekarang sudah tidak digunakan lagi.

Dalam bab ini, kita akan membahas cara berinteraksi dengan CoppeliaSim menggunakan plugin RosInterface yang mereplikasi fungsi API jarak jauh secara transparan. Dengan menggunakan antarmuka ini, CoppeliaSim akan bertindak sebagai node ROS yang dapat berkomunikasi dengan node lain melalui layanan ROS, penerbit ROS, dan pelanggan ROS. Antarmuka diimplementasikan oleh perpustakaan eksternal yang sudah tersedia di folder CoppeliaSim. Sebelum menyiapkan plugin RosInterface, kita perlu mengkonfigurasi lingkungan untuk menjalankan CoppeliaSim. Pertama, kita perlu memaksa sistem operasi kita untuk memuat pustaka bersama Lua dan Qt5 dari folder akar CoppeliaSim. Lua adalah bahasa pemrograman yang digunakan untuk berbagai aplikasi tingkat tinggi, dan digunakan dari CoppeliaSim untuk memprogram robot simulasi langsung dari antarmukanya.

Sekarang, kami siap memulai simulator. Untuk mengaktifkan antarmuka komunikasi ROS, perintah roscore harus dijalankan di mesin Anda sebelum membuka simulator, sedangkan untuk membuka CoppeliaSim, kita dapat menggunakan perintah berikut:

```
cd $COPPELIASIM_ROOT
```

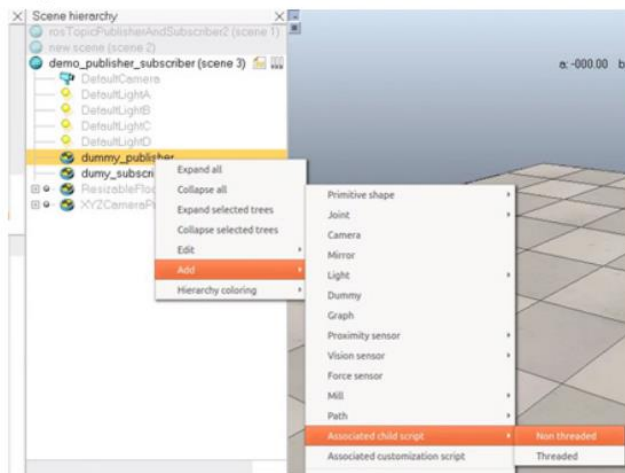
```
./coppeliaSim.sh
```

Memahami plugin RosInterface

Plugin RosInterface adalah bagian dari kerangka CoppeliaSim API. Meskipun plugin terpasang dengan benar di sistem Anda, operasi pemuatan akan gagal jika roscore tidak berjalan pada saat itu. Dalam hal ini, fungsi ROS tidak dapat bekerja dengan baik. Untuk mencegah perilaku tak terduga seperti itu, nanti kita akan melihat cara memeriksa apakah plugin RosInterface berfungsi dengan baik. Mari kita bahas cara berinteraksi dengan CoppeliaSim menggunakan topik ROS.

Berinteraksi dengan CoppeliaSim menggunakan topik ROS

Kami sekarang akan membahas cara menggunakan topik ROS untuk berkomunikasi dengan CoppeliaSim. Hal ini berguna ketika kita ingin mengirimkan informasi ke objek simulasi, atau mengambil data yang dihasilkan oleh sensor atau aktuator robot. Cara paling umum untuk memprogram adegan simulasi simulator ini adalah dengan menggunakan Luascripts. Setiap objek adegan dapat dikaitkan ke skrip yang secara otomatis dipanggil saat simulasi dimulai dan dieksekusi secara siklis selama waktu simulasi. Pada contoh berikutnya, kita akan membuat adegan dengan dua objek. Satu akan diprogram untuk menerbitkan data bilangan bulat dari topik tertentu sementara yang lain berlangganan topik ini, menampilkan data float di konsol CoppeliaSim. Gunakan menu drop-down pada panel Scene Hierarchy, pilih Add | Entri tiruan. Kita dapat membuat dua objek, objek dummy_publisher dan objek dummy_subscriber, dan mengaitkan skrip dengan masing-masing objek. Gunakan tombol kanan mouse pada objek yang dibuat, dan pilih Tambah | Skrip anak terkait | Entri non-utas, seperti yang ditunjukkan pada tangkapan layar berikut:



Ringkasan

Dalam bab ini, kami terutama mereplikasi apa yang telah kami lakukan di bab sebelumnya dengan Gazebo, menggunakan simulator robot lain: CoppeliaSim dan Webots. Ini adalah program perangkat lunak simulasi multiplatform yang mengintegrasikan berbagai teknologi dan sangat serbaguna. Berkat UI intuitifnya, robot ini mungkin lebih mudah digunakan oleh pengguna baru. Kami pada dasarnya menyimulasikan dua robot, satu diimpor menggunakan file URDF dari lengan tujuh DOF yang dirancang pada bab sebelumnya, dan yang lainnya adalah robot beroda diferensial populer yang disediakan oleh model simulasi Webots. Kami mempelajari cara menghubungkan dan mengontrol sambungan robot model kami dengan ROS dan cara menggerakkan robot seluler penggerak diferensial menggunakan topik. Di bab berikutnya, kita akan melihat cara menghubungkan lengan robot dengan paket ROS MoveIt dan ponsel robot dengan tumpukan Navigasi.

- **Technical Report Chapter 6 : Using the ROS MoveIt! and Navigation Stack**

Dalam bab ini, kita akan membahas masalah perencanaan gerak. Menggerakkan robot dengan mengendalikan sendi-sendinya secara manual mungkin merupakan tugas yang sulit, terutama jika kita ingin menambahkan batasan posisi atau kecepatan pada pergerakan robot. Demikian pula, mengendarai robot bergerak dan menghindari rintangan memerlukan perencanaan jalur. Oleh karena itu, kami akan menyelesaikan masalah ini menggunakan ROS MoveIt! dan tumpukan Navigasi.

Pindahkan! mewakili sekumpulan paket dan alat untuk melakukan manipulasi seluler di ROS. Halaman web resmi (<http://moveit.ros.org/>) berisi dokumentasi, daftar robot yang menggunakan MoveIt!, dan berbagai contoh untuk mendemonstrasikan pick and place, menggenggam, perencanaan gerak sederhana menggunakan inverse kinematika (IK), dan sebagainya pada. 156 Menggunakan ROS MoveIt! dan Tumpukan Navigasi.

Pindahkan! berisi perangkat lunak canggih untuk perencanaan gerak, manipulasi, persepsi tiga dimensi (3D), kinematika, pemeriksaan tabrakan, kontrol, dan navigasi. Selain antarmuka baris perintah (CLI), ia memiliki beberapa antarmuka pengguna grafis (GUI) yang bagus untuk mengonfigurasi robot baru di MoveIt!. Ada juga plugin ROS Visualization (RViz) yang memungkinkan perencanaan gerakan dari UI yang nyaman. Kita juga akan melihat cara merencanakan gerak robot kita menggunakan MoveIt! Antarmuka pemrograman aplikasi (API) C++.

Berikutnya adalah tumpukan Navigasi, seperangkat alat dan pustaka canggih lainnya yang berfungsi terutama dengan navigasi robot seluler. Tumpukan Navigasi berisi algoritma navigasi siap pakai yang dapat digunakan pada robot bergerak, terutama untuk robot beroda diferensial. Dengan menggunakan tumpukan ini, kita dapat membuat robot menjadi otonom, dan itulah konsep terakhir yang akan kita lihat di tumpukan Navigasi.

Bagian pertama bab ini terutama akan berkonsentrasi pada MoveIt! paket, instalasi, dan arsitektur. Setelah membahas konsep utama MoveIt!, kita akan melihat cara membuat MoveIt! paket untuk lengan robot kami, yang dapat memberikan perencanaan jalur sadar tabrakan pada robot kami. Dengan menggunakan paket ini, kita dapat melakukan perencanaan gerak (kinematika terbalik) di RViz dan dapat berinteraksi dengan Gazebo atau robot sungguhan untuk menjalankan jalur.

Setelah membahas antarmuka, kita akan membahas lebih lanjut tentang tumpukan Navigasi dan melihat cara melakukan navigasi otonom menggunakan Simultaneous Localization And Mapping (SLAM) dan Adaptive Monte Carlo Localization (Amcl).

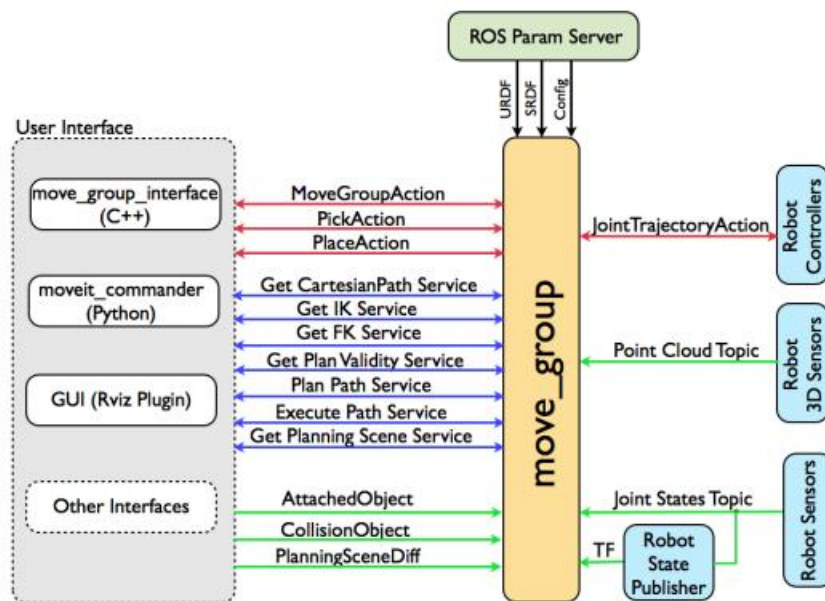
Gerakan itu! Arsitektur

Sebelum menggunakan MoveIt! di sistem ROS kami, Anda harus menginstalnya. Prosedur instalasinya sangat sederhana dan hanya dengan satu perintah. Dengan menggunakan perintah berikut, kami menginstal MoveIt! inti, satu set plugin dan perencana untuk ROS Noetic:

```
sudo apt-get install plugin ros-noetic-moveit ros-noetic-moveit ros-noetic-moveit-planners
```

Mari kita mulai dengan MoveIt! dengan mendiskusikan arsitekturnya. Memahami arsitektur MoveIt! membantu memprogram dan menghubungkan robot dengan MoveIt!. Kami akan segera mempelajari arsitektur dan konsep penting MoveIt!, dan mulai berinteraksi dan memprogram robot kami.

Berikut ini ikhtisar MoveIt! Arsitektur:



Node move_group

Kita dapat mengatakan bahwa `move_group` adalah jantung dari MoveIt!, karena node ini bertindak sebagai integrator berbagai komponen robot dan memberikan tindakan/layanan sesuai dengan kebutuhan pengguna.

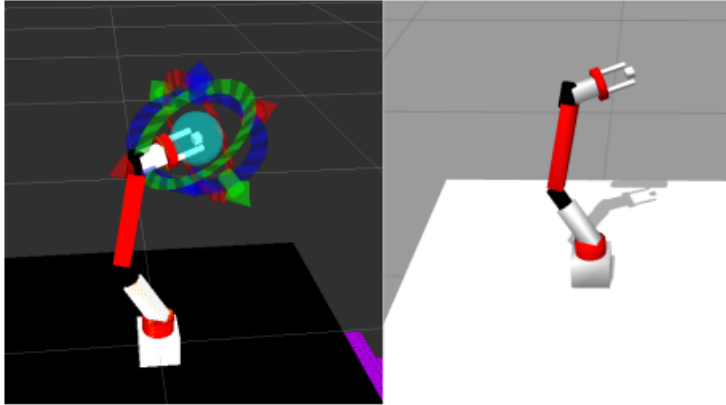
Melihat arsitekturnya, terlihat jelas bahwa node `move_group` mengumpulkan informasi robot seperti point cloud, status gabungan robot, dan transformasi (TF) robot dalam bentuk topik dan layanan.

Dari server parameter, ia mengumpulkan data kinematika robot, seperti Unified Robot Description Format (URDF), Semantic Robot Description Format (SRDF), dan file konfigurasi. File SRDF dan file konfigurasi dibuat saat kita membuat MoveIt! paket untuk robot kami. File konfigurasi berisi file parameter untuk menetapkan batas gabungan, persepsi, kinematika, efektor akhir, dan sebagainya. Kita akan melihat file-file tersebut ketika kita mendiskusikan pembuatan MoveIt! paket untuk robot kami.

Setelah MoveIt! mendapatkan semua informasi yang diperlukan tentang robot dan konfigurasinya, kita dapat mulai memerintahkan robot dari UI. Kita bisa menggunakan C++ atau Python MoveIt! API untuk memerintahkan node `move_group` untuk melakukan tindakan seperti pick/place, IK, dan forward kinematics (FK), antara lain. Dengan menggunakan plugin perencanaan gerak RViz, kita dapat memerintahkan robot dari GUI RViz

Karena node `move_group` adalah integrator sederhana, ia tidak menjalankan algoritma perencanaan gerakan apa pun secara langsung melainkan menghubungkan semua fungsi sebagai plugin. Ada plugin untuk pemecah kinematika, perencanaan gerak, dan sebagainya. Kita dapat memperluas kemampuannya melalui plugin ini. Setelah perencanaan gerak, lintasan yang dihasilkan berkomunikasi dengan pengontrol di robot menggunakan antarmuka `FollowJointTrajectoryAction`. Ini adalah antarmuka tindakan di mana server tindakan dijalankan pada robot, dan `move_node` memulai klien tindakan yang berkomunikasi dengan server ini dan mengeksekusi lintasan pada robot nyata atau pada simulator robot.

Di akhir diskusi kita tentang MoveIt!, kita akan melihat cara menghubungkan MoveIt! dengan GUI RViz ke Gazebo. Tangkapan layar berikut menunjukkan lengan robot yang dikendalikan dari RViz dan lintasan dijalankan di dalam Gazebo:



Memahami tumpukan Navigasi ROS

Tujuan utama dari paket Navigasi ROS adalah untuk menggerakkan robot dari posisi awal ke posisi tujuan, tanpa menimbulkan benturan dengan lingkungan. Paket Navigasi ROS hadir dengan implementasi beberapa algoritma terkait navigasi yang dapat dengan mudah membantu mengimplementasikan navigasi otonom pada robot bergerak. Pengguna hanya perlu memasukkan posisi tujuan robot dan data odometri robot dari sensor seperti wheel encoder, Inertial Unit Pengukuran (IMU), dan Sistem Pemosisian Global (GPS), serta aliran data sensor lainnya, seperti data pemindai laser atau awan titik 3D dari sensor seperti sensor Kedalaman Merah-Hijau-Biru (RGB-D). Output dari paket Navigasi akan berupa perintah kecepatan yang akan mengarahkan robot ke posisi tujuan tertentu. Tumpukan Navigasi berisi implementasi algoritma standar, seperti SLAM, A*(star), Dijkstra, amcl, dan sebagainya, yang bisa langsung digunakan di aplikasi kita.

Ringkasan

Bab ini menawarkan gambaran singkat tentang MoveIt! dan tumpukan Navigasi ROS dan mendemonstrasikan kemampuannya menggunakan simulasi Gazebo dari pangkalan bergerak lengan robot. Bab ini dimulai dengan MoveIt! ikhtisar dan mendiskusikan konsep detail tentang MoveIt!. Setelah mendiskusikan MoveIt!, kami menghubungkan MoveIt! dengan Gazebo. Setelah antarmuka, kami mengeksekusi lintasan dari MoveIt! di Gazebo. Bagian selanjutnya adalah tentang tumpukan Navigasi ROS. Kami juga mendiskusikan konsep dan cara kerjanya. Setelah mendiskusikan konsepnya, kami mencoba menghubungkan robot kami di Gazebo ke tumpukan Navigasi dan membuat peta menggunakan SLAM. Setelah ini, kami melakukan navigasi otonom menggunakan amcl dan peta statis. Pada bab berikutnya, kita akan membahas pluginlib, nodelet, dan pengontrol. Berikut adalah beberapa pertanyaan berdasarkan apa yang kita bahas dalam bab ini.

- Technical Report Chapter 7 : Exploring the Advanced Capabilities of ROS MoveIt!

Perencanaan gerak menggunakan antarmuka move_group C++

Di Bab 6, Menggunakan ROS MoveIt! dan Navigation Stack, kita membahas cara berinteraksi dengan lengan robot dan cara merencanakan jalurnya menggunakan MoveIt! Plugin perencanaan gerak ROS Visualization (RViz). Di bagian ini, kita akan melihat cara memprogram gerakan robot menggunakan move_group C++ API. Perencanaan gerak menggunakan RViz juga dapat dilakukan secara terprogram melalui API C++ move_group. Langkah pertama untuk mulai bekerja dengan API C++ adalah membuat paket ROS lain yang memiliki paket MoveIt! paket sebagai dependensi. Kita dapat membuat paket yang sama menggunakan perintah berikut:

```
catkin_create_pkg seven_dof_arm_test catkin cmake_modules
interactive_markers moveit_core moveit_ros_perception moveit_
ros_planning_interface pluginlib roscpp std_msgs
```

Perencanaan gerak jalur acak menggunakan MoveIt! C++ API

Contoh pertama yang akan kita lihat adalah rencana gerak acak menggunakan MoveIt!. C++ API. Anda akan mendapatkan file kode bernama test_random.cpp dari folder src. Kode dan deskripsi setiap baris berikut. Saat kita mengeksekusi node ini, ia akan merencanakan jalur acak dan mengeksekusinya, seperti yang diilustrasikan dalam cuplikan kode berikut:

```
#include <moveit/move_group_interface/move_group_interface.h>
int main(int argc, char **argv)
{
    ros::init(argc, argv, "move_group_interface_demo");
    // start a ROS spinning thread
    ros::AsyncSpinner spinner(1);
    spinner.start();
    // this connects to a running instance of the move_group node
    //move_group_interface::MoveGroup group("arm");
    moveit::planning_interface::MoveGroupInterface group("arm");
    // specify that our target will be a random one
    group.setRandomTarget();
    // plan the motion and then move the group to the sampled
    target
    group.move();
    ros::waitForShutdown();
}
```

Untuk membuat kode sumber, kita harus menambahkan baris kode berikut ke CMakeLists.txt. Anda akan mendapatkan file CMakeLists.txt lengkap dari paket yang ada sendiri:

```
add_executable(test_random_node src/test_random.cpp)
add_dependencies(test_random_node seven_dof_arm_test_generate_
messages_cpp)
target_link_libraries(test_random_node
${catkin_LIBRARIES})
```

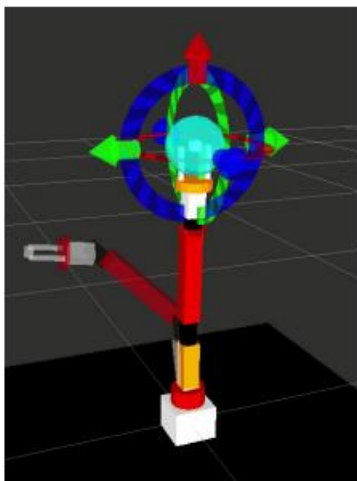
Kita dapat membuat paket menggunakan perintah `catkin_make`. Pertama, periksa apakah `test_random.cpp` dibuat dengan benar atau tidak. Jika kode dibuat dengan benar, kita dapat mulai menguji kode tersebut. Perintah berikut akan memulai RViz dengan lengan 7-DOF dengan plugin perencanaan gerak:

```
roslaunch seven_dof_arm_config demo.launch
```

Pindahkan efektor akhir untuk memeriksa apakah semuanya berfungsi dengan baik di RViz. Jalankan node C++ untuk perencanaan ke posisi acak menggunakan perintah berikut:

```
roslaunch seven_dof_arm_test test_random_node
```

Output dari RViz ditampilkan berikutnya. Lengan akan memilih posisi acak yang memiliki kinematika terbalik (IK) yang valid dan rencana gerak dari posisi saat ini:



Dalam contoh ini, kita hanya mencoba menggerakkan robot dengan pose target acak dan layak untuk efektor akhirnya. Di bagian selanjutnya, kita akan menetapkan pose yang diinginkan padanya.

Pemeriksaan tabrakan dengan lengan robot menggunakan MoveIt!

Seiring dengan perencanaan gerak dan algoritma penyelesaian IK, salah satu tugas terpenting yang dilakukan secara paralel di MoveIt! adalah pemeriksaan tabrakan dan penghindarannya. Pindahkan! dapat menangani benturan diri dan benturan lingkungan, memanfaatkan Perpustakaan Tabrakan Fleksibel (FCL) bawaan (http://gamma.cs.unc.edu/FCL/fcl_docs/webpage/generated/index.html), sebuah sumber terbuka proyek yang mengimplementasikan berbagai algoritma deteksi dan penghindaran tabrakan. Pindahkan! mengambil kekuatan FCL dan menangani tabrakan di dalam adegan perencanaan menggunakan kelas `collision_detection::CollisionWorld`. Gerakan itu! pemeriksaan tabrakan mencakup objek—seperti jerat, bentuk primitif—seperti kotak dan silinder, dan peta okto. Pustaka OctoMap (<http://octomap.github.io/>) mengimplementasikan kisi okupansi 3D, yang disebut octree, yang terdiri dari informasi probabilistik mengenai hambatan di lingkungan. Gerakan itu! Paket

dapat membuat octomap menggunakan informasi point-cloud 3D dan dapat langsung memasukkan perpustakaan OctoMap ke perpustakaan FCL untuk pemeriksaan tabrakan.

Seperti halnya perencanaan gerak, pemeriksaan tabrakan sangat intensif secara komputasi. Kita dapat menyempurnakan pemeriksaan tabrakan antara dua benda—misalnya, tautan robot, atau dengan lingkungan—menggunakan parameter yang disebut Matriks Tabrakan yang Diizinkan (ACM). Jika nilai tabrakan antara dua link diatur ke 1 di ACM, tidak akan ada pemeriksaan tabrakan. Kami mungkin mengatur ini untuk tautan yang berjauhan satu sama lain. Kita dapat mengoptimalkan proses pemeriksaan tabrakan dengan mengoptimalkan matriks ini.

Melakukan manipulasi objek dengan MoveIt!

Memanipulasi objek adalah salah satu penggunaan utama lengan robot. Kemampuan untuk mengambil objek dan menempatkannya di lokasi berbeda di ruang kerja robot sangat berguna, baik dalam aplikasi industri maupun penelitian. Proses pengambilan juga dikenal sebagai menggenggam dan merupakan tugas yang kompleks karena banyak kendala yang diperlukan untuk mengambil suatu objek dengan cara yang benar. Manusia menangani operasi penangkapan menggunakan kecerdasannya, namun robot memerlukan aturan untuk melakukannya. Salah satu kendala dalam menggenggam adalah kekuatan yang mendekat; efektor akhir harus menyesuaikan gaya genggam untuk mengambil objek tetapi tidak membuat deformasi apa pun pada objek saat menggenggam. Selain itu, pose menggenggam juga diperlukan untuk mengambil suatu objek dengan sebaik-baiknya dan harus diperhitungkan dengan mempertimbangkan bentuk dan posenya. Di bagian ini, kita akan berinteraksi dengan objek adegan MoveIt! untuk mensimulasikan operasi pengambilan dan tempat.

Ringkasan

Dalam bab ini, kita menjelajahi beberapa fitur lanjutan MoveIt!, yang menunjukkan cara menulis kode C++ untuk mengontrol manipulator robot yang disimulasikan dan nyata. Bab ini dimulai dengan diskusi tentang pemeriksaan tabrakan menggunakan MoveIt!. Kita melihat bagaimana menambahkan objek tumbukan menggunakan MoveIt! API, dan melihat impor langsung mesh ke lokasi perencanaan. Kita membahas node ROS untuk memeriksa tabrakan menggunakan MoveIt! Lebah. Setelah mempelajari tentang tumbukan, kami beralih ke persepsi menggunakan MoveIt!. Kami menghubungkan data titik cloud simulasi ke MoveIt! dan membuat octomap di MoveIt!. Setelah mendiskusikan aspek-aspek ini, kami beralih ke antarmuka perangkat keras MoveIt! menggunakan servo DYNAMIXEL dan pengontrol ROS-nya. Pada akhirnya, kami melihat lengan robot sungguhan yang disebut lengan COOL dan antarmukanya ke MoveIt!, yang seluruhnya dibuat menggunakan pengontrol DYNAMIXEL. Pada bab selanjutnya, kita akan membahas jenis platform robot lainnya, robot udara, dan cara mengintegrasikan serta memprogramnya menggunakan ROS.

• Technical Report Chapter 8 : ROS for Aerial Robots

Menggunakan robot udara

Saat ini, kendaraan terbang sangat populer. Bahkan dalam konfigurasi utamanya yang dikendalikan oleh pengontrol radio, beberapa kendaraan terbang dapat dianggap sebagai robot yang merespons lingkungannya agar tetap berada di udara. Kendaraan semacam itu dapat menggunakan sensor eksternal untuk memperkirakan keadaan dan posenya, sehingga memungkinkannya terbang secara mandiri. Tentu saja, memberikan otonomi pada robot terbang lebih rumit daripada melakukan hal yang sama pada robot darat karena beberapa alasan, yang tercantum di sini:

- **Stabilisasi:** Robot terbang harus mampu menyesuaikan posisinya untuk mempertahankan posisi dan orientasinya relatif terhadap lingkungan. Sensor inersia saja tidak cukup untuk menyelesaikan tugas ini, karena mereka tidak mampu memperkirakan perbedaan posisi yang disebabkan oleh gangguan eksternal (seperti angin atau aliran udara di tanah), atau kemungkinan kesalahan yang dihasilkan karena sensor unit pengukuran inersia.
- **Sumber daya komputasi yang rendah:** Dibandingkan dengan robot darat, platform penerbangan memiliki masalah muatan. Oleh karena itu, hanya perangkat keras kecil dan ringan yang harus digunakan. Oleh karena itu, komputer pendamping kecil harus digunakan.
- **Masalah debugging:** Selama pengembangan strategi fusi dan kontrol sensor, debugging bukanlah tugas yang mudah. Masalah yang berkaitan dengan kerangka acuan atau penguatan kendali yang salah dapat menyebabkan platform udara jatuh. Hal ini dapat menyebabkan kerusakan pada robot dan orang di sekitarnya.
- **Komunikasi dengan stasiun bumi:** Komunikasi antara PC pendamping UAV dan stasiun bumi biasanya bergantung pada protokol komunikasi berdaya rendah dan lambat untuk mengatasi jarak antara robot dan PC darat.

Masalah lain dengan robot ini adalah pengontrol robot diimplementasikan pada papan tertanam yang terintegrasi. Ini disebut autopilot dan, dalam beberapa kasus, performa gerakan robot sangat bergantung pada autopilot. Pada beberapa bagian berikutnya, kita akan membahas sensor perangkat keras dasar UAV dan fungsi autopilotnya masing-masing. Kemudian kita akan mempelajari cara mensimulasikan robot terbang sungguhan dengan menghubungkannya dengan ROS.

perangkat keras UAV

Inti dari UAV adalah autopilot. Ini bertanggung jawab atas inisialisasi dan antarmuka sensor onboard. Selain itu, autopilotlah yang menerima masukan untuk mengontrol aktuator UAV (baling-balingnya) dengan benar. Konfigurasi platform yang berbeda tersedia untuk UAV. Yang paling umum adalah quadrotor. Ini memiliki empat motor dan dapat digerakkan dengan konfigurasi silang (X) atau plus (+). Selain itu, dalam versi koaksialnya, quadrotor memiliki dua jalur motor. Setiap sumbu quadrotor memiliki dua motor dan baling-baling yang dipasang secara koaksial, sehingga totalnya ada delapan. Hal yang sama berlaku untuk hexacopters dan octocopters. Namun, strategi kendali tidak secara langsung bergantung pada konfigurasi badan pesawat, karena autopilot secara langsung menerjemahkan data kendali menjadi input motor.

Sensor utama autopilot adalah Inertial Measurement Unit (IMU). Modul ini digunakan untuk menghitung sikap, ketinggian, dan arah penerbangan. Biasanya mencakup

mengikuti:

- Girooskop yang menentukan sikap pesawat, termasuk pitch and roll-nya. Hal ini menunjukkan gerak rotasi pesawat tersebut.
- Akselerometer yang menentukan laju perubahan kecepatan pesawat terhadap ketiga sumbu.
- Altimeter atau barometer yang menentukan ketinggian pesawat di atas permukaan tanah. Pada ketinggian rendah, sensor sonar yang menghadap ke bawah dapat digunakan untuk menentukan ketinggian hingga beberapa meter.
- Magnetometer berfungsi sebagai kompas untuk menunjukkan arah pesawat dengan menggunakan medan magnet bumi sebagai acuan.

Sensor inersia menggabungkan sensor-sensor ini untuk mengukur dan menampilkan informasi lengkap berkaitan dengan karakteristik penerbangan quadrotor. Biasanya, unit ini akan mengukur percepatan dan orientasi pesawat terbang dalam tiga dimensi. Sensor ini memungkinkan penerbangan di dalam dan luar ruangan. Namun, mereka mengalami kesalahan kecil yang mungkin terakumulasi selama penerbangan. Sensor penting lainnya untuk UAV adalah Global Positioning System (GPS). Sensor ini memungkinkan robot memperkirakan posisi global dirinya dalam kaitannya dengan garis lintang dan garis bujur, sehingga memungkinkan robot untuk menstabilkan posisinya. Namun sensor ini hanya bisa digunakan di luar ruangan. Oleh karena itu, teknik lain berdasarkan penglihatan atau sensor LiDAR harus digunakan di lingkungan dalam ruangan. Sekarang kita telah memeriksa elemen dasar autopilot, mari kita bahas salah satu autopilot open source yang paling umum digunakan pada robot udara – autopilot Pixhawk.

Komunikasi PC/autopilot

Untuk mengirim dan menerima informasi dari platform udara (simulasi atau nyata), kita dapat menggunakan dua mode berikut:

- Stasiun bumi: Perangkat lunak tingkat tinggi yang dapat dihubungkan ke autopilot untuk mengirimkan perintah seperti lepas landas dan mendarat atau menyampaikan informasi navigasi titik arah.
- API: Memprogram API memungkinkan pengembang mengelola perilaku robot.

Dalam kedua kasus tersebut, komunikasi dikelola oleh protokol MAVLink. Micro Air Vehicle Link (MAVLink) dan merupakan protokol untuk berkomunikasi dengan kendaraan kecil tak berawak. Ini dirancang sebagai perpustakaan penyusunan pesan khusus header. Hal ini sebagian besar digunakan untuk komunikasi antara Ground Control Station (GCS) dan kendaraan tak berawak, dan dalam interkomunikasi subsistem kendaraan. Contoh paket datagram ditunjukkan pada gambar berikut:

Pesan tidak lebih dari 263 byte. Pengirim selalu mengisi kolom System ID dan Component ID agar penerima mengetahui dari mana paket berasal. ID sistem adalah ID unik untuk setiap kendaraan atau stasiun bumi. Stasiun bumi biasanya menggunakan ID sistem tinggi seperti 255, sedangkan kendaraan default menggunakan 1. ID komponen untuk stasiun bumi atau pengontrol penerbangan biasanya 1. Bidang ID Pesan dapat dilihat di common.xml dan ardupilot.xml, selanjutnya ke nama pesan. Misalnya, ID pesan HEARTBEAT adalah 0. Terakhir, bagian data pesan menampung nilai bidang individual yang sedang dikirim. Saat ini, versi terbaru MAVLink adalah 2.0 dan kompatibel dengan protokol versi pertama. Artinya jika suatu perangkat memahami pesan MAVLink2, maka perangkat tersebut pasti memahami pesan MAVLink1. Sedangkan untuk protokol transport, MAVLink didasarkan pada komunikasi serial. Oleh karena itu, pesan dari papan dapat dibaca dengan menerapkan komunikasi serial klasik berdasarkan User Datagram Protocol (UDP).

Singkatnya, MAVLink menyediakan protokol komunikasi standar untuk mendapatkan data dari UAV dan mengirimkan perintah kepada mereka. Seperti banyak tumpukan kontrol lainnya, PX4 menggunakan kerangka komunikasi MAVLink untuk berinteraksi dengan Ground Control Station (GCS) atau PC onboard. Contoh pesan MAVLink yang dihasilkan oleh UAV antara lain

pengikut:

- Posisi global: Output dari GPS tetap UAV
- Posisi lokal: Posisi Cartesian UAV, dihasilkan melalui posisi global dan sensor lokal lainnya
- Attitude : Informasi mengenai sikap UAVA terhadap perintah yang diterima oleh UAV, antara lain sebagai berikut :
- Lepas landas: Untuk lepas landas pada posisi global tertentu dan pada ketinggian tertentu.
- Setpoint: Posisi yang ingin dicapai. Posisi tersebut dapat ditentukan dengan berbagai cara: setpoint lokal, global, posisi, dan kecepatan dapat diterima.
- Mode penerbangan: Mode penerbangan yang diinginkan. Mode penerbangan menentukan cara robot merespons masukan pengguna dan mengontrol pergerakan kendaraan.

Mode yang berbeda dapat mencakup kontrol posisi, kontrol sikap, dan mode OFFBOARD. Saat menggunakan mode OFFBOARD, kendaraan mematuhi titik setel posisi, kecepatan, atau sikap yang disediakan melalui MAVLink. Dalam konteks ini, setpoint mungkin disediakan oleh komputer pendamping (biasanya dihubungkan melalui kabel serial atau Wi-Fi). Seperti biasa, kita tidak perlu mengimplementasikan protokol MAVLink dari awal. Kita bisa menggunakan pembungkus perpustakaan yang dibuat di ROS ini, yang disebut mavros.

Ringkasan

Bab ini memperkenalkan konsep robot udara dan membahas elemen utamanya. Kami juga menjelaskan salah satu papan autopilot paling terkenal yang digunakan untuk mengembangkan aplikasi khusus dengan UAV – papan kontrol Pixhawk yang menjalankan autopilot PX4. Setelah kami mempelajari cara menggunakan platform multirotor nyata dan mengintegrasikannya dengan ROS, kami kemudian membahas dua modalitas simulasi. Sangat penting untuk mensimulasikan pengaruh algoritma kontrol sebelum menjalankannya pada UAV nyata. Hal ini untuk mencegah kerusakan pada robot dan orang di sekitarnya. Pada bab berikutnya, kita akan membahas cara menghubungkan papan mikrokontroler dan aktuator dengan ROS.