

UAS Robotika 2023/2024

Nama : Alfin Andreas bastian Situmeang
NIM : 1103202143

Chapter 1: Introduction to ROS

Technical requirements

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

Why should we use ROS?

Robot Operating System (ROS) adalah kerangka kerja perangkat lunak sumber terbuka yang menyediakan alat dan pustaka untuk pengembangan perangkat lunak robotik. ROS menawarkan berbagai fitur untuk membantu pengembang dalam tugas-tugas seperti pengiriman pesan, komputasi terdistribusi, penggunaan kembali kode, dan implementasi algoritme canggih.

ROS dimulai pada tahun 2007 oleh Morgan Quigley dan dikembangkan di Willow Garage, sebuah 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 yang dapat dengan mudah diintegrasikan dengan aplikasi robotik khusus.

Berikut adalah beberapa alasan mengapa ROS menjadi kerangka kerja yang populer untuk pengembangan perangkat lunak robotik:

- Kemampuan kelas atas: ROS hadir dengan fungsi yang siap digunakan. Sebagai contoh, Pelokalan dan Pemetaan Simultan (SLAM) dan Adaptive Monte Carlo Localization (AMCL) di ROS dapat digunakan untuk memiliki otonom navigasi di robot seluler, sedangkan paket MoveIt dapat digunakan untuk gerakan perencanaan untuk manipulator robot. Kemampuan ini dapat langsung digunakan dalam perangkat lunak robot tanpa kerumitan. Dalam beberapa kasus, paket-paket ini cukup untuk memiliki tugas robotika inti pada platform yang berbeda. Juga, kemampuan ini sangat tinggi dapat dikonfigurasi; kita dapat menyempurnakan masing-masing menggunakan berbagai parameter.
- Banyak sekali alat: Ekosistem ROS dilengkapi dengan banyak sekali alat untuk melakukan debugging, memvisualisasikan, dan melakukan simulasi. Alat-alat tersebut,

seperti `rqt_gui`, `RViz`, dan `Gazebo`, adalah beberapa alat open source terkuat untuk debugging, visualisasi, dan simulasi. Kerangka kerja perangkat lunak yang memiliki banyak alat ini sangat jarang.

- Dukungan untuk sensor dan aktuator kelas atas: ROS memungkinkan kita untuk menggunakan perangkat yang berbeda driver dan paket antarmuka berbagai sensor dan aktuator dalam robotika. Seperti sensor kelas atas termasuk LIDAR 3D, pemindai laser, sensor kedalaman, aktuator, dan banyak lagi. Kita dapat menghubungkan komponen-komponen ini dengan ROS tanpa kerumitan.
- Penanganan sumber daya secara bersamaan: Menangani sumber daya perangkat keras melalui lebih dari dua proses selalu memusingkan. Bayangkan kita ingin memproses sebuah gambar dari sebuah kamera untuk deteksi wajah dan deteksi gerakan; kita dapat menulis kode sebagai entitas tunggal yang dapat melakukan keduanya, atau kita dapat menulis kode berulir tunggal untuk konkurensi. Jika kita ingin menambahkan lebih dari dua fitur ke dalam sebuah thread, aplikasi-aplikasi akan menjadi kompleks dan sulit untuk di-debug. Tetapi di ROS, kita dapat mengakses perangkat menggunakan topik ROS dari driver ROS. Sejumlah node ROS dapat berlangganan pesan gambar dari driver kamera ROS, dan setiap node bisa memiliki fungsi yang berbeda. Hal ini dapat mengurangi kompleksitas dalam komputasi dan juga meningkatkan kemampuan debugging seluruh sistem.

ROS metapackages

Metapackages adalah paket khusus yang hanya membutuhkan satu file; yaitu file `package.xml` file.

Metapackages hanya mengelompokkan sekumpulan beberapa paket sebagai satu paket logis. Di dalam file `package.xml`, metapackage berisi tag ekspor, seperti yang ditunjukkan di sini:

```
<export>
  <metapackage/>
</export>
```

Selain itu, dalam metapackages, tidak ada dependensi `<buildtool_depend>` untuk catkin; hanya ada dependensi `<run_depend>`, yang merupakan paket-paket yang dikelompokkan yang dikelompokkan di dalam metapackage

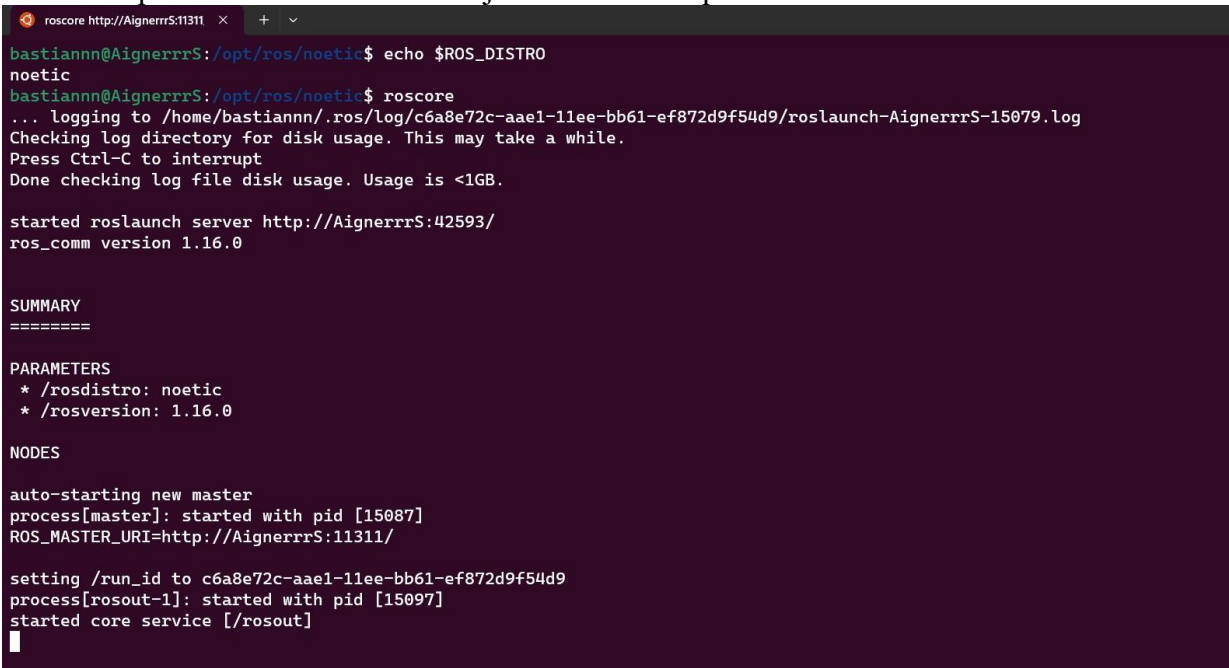
Running the ROS master and the ROS parameter

Sebelum menjalankan node ROS apa pun, kita harus memulai master ROS dan parameter ROS server. Kita dapat memulai master ROS dan server parameter ROS dengan menggunakan satu yang disebut roscore, yang akan memulai program-program berikut:

- ROS master
- ROS parameter server
- rosout logging nodes

Node rosout akan mengumpulkan pesan log dari node ROS lain dan menyimpannya dalam file log, dan juga akan menyiarkan ulang pesan log yang dikumpulkan ke topik lain. Topik /rosout dipublikasikan oleh node ROS menggunakan pustaka klien ROS seperti roscpp dan rospy, dan topik ini dilanggan oleh node rosout, yang menyiarkan ulang pesan dalam topik lain yang disebut /rosout_agg. Topik ini berisi aliran agregat log pesan. Perintah roscore harus dijalankan sebagai prasyarat untuk menjalankan ROS node. Tangkapan layar berikut ini menunjukkan pesan-pesan yang dicetak ketika kita menjalankan perintah roscore di Terminal.

Gunakan perintah berikut untuk menjalankan roscore pada Terminal Linux:



```
roscore http://AignerrrS:11311 x + v
bastiannn@AignerrrS:/opt/ros/noetic$ echo $ROS_DISTRO
noetic
bastiannn@AignerrrS:/opt/ros/noetic$ roscore
... logging to /home/bastiannn/.ros/log/c6a8e72c-aae1-11ee-bb61-ef872d9f54d9/roslaunch-AignerrrS-15079.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://AignerrrS:42593/
ros_comm version 1.16.0

SUMMARY
=====

PARAMETERS
* /rostdistro: noetic
* /rosversion: 1.16.0

NODES

auto-starting new master
process[master]: started with pid [15087]
ROS_MASTER_URI=http://AignerrrS:11311/

setting /run_id to c6a8e72c-aae1-11ee-bb61-ef872d9f54d9
process[rosout-1]: started with pid [15097]
started core service [/rosout]
```

Berikut ini adalah isi dari roscore.xml:

```
<launch>
  <group ns="/">
    <param name="rosversion" command="rosversion roslaunch" />
    <param name="rostdistro" command="rosversion -d" />
    <node pkg="rosout" type="rosout" name="rosout"
      respawn="true"/>
  </group>
</launch>
```

Isi dari roscore.xml

File roscore.xml adalah file konfigurasi yang digunakan oleh roscore untuk memulai node ROS master. File ini berisi informasi tentang node master, termasuk nama node, alamat IP, dan port.

Isi dari roscore.xml yang Anda kirimkan adalah sebagai berikut:

XML

```
<launch>
  <node pkg="ros" type="roscore" name="master" respawn="true"/>
</launch>
```

File ini berisi satu node ROS, yaitu node master dengan nama "master". Node master ini akan respawn secara otomatis jika mati.

Penjelasan

- Tag `launch`: Tag ini menandakan bahwa file ini adalah file konfigurasi untuk roscore.
- Tag `node`: Tag ini menandakan bahwa node ROS akan ditambahkan ke sistem.
- Atribut `pkg`: Atribut ini menentukan paket yang berisi node ROS.
- Atribut `type`: Atribut ini menentukan nama kelas node ROS.
- Atribut `name`: Atribut ini menentukan nama node ROS.
- Atribut `respawn`: Atribut ini menentukan apakah node ROS akan respawn secara otomatis jika mati.

Kegunaan

File roscore.xml dapat digunakan untuk mengatur parameter node master. Misalnya, Anda dapat menggunakan file ini untuk mengubah nama node master, alamat IP, atau port.

Contoh

Berikut adalah contoh file roscore.xml yang mengubah nama node master menjadi "my_master":

XML

```
<launch>
  <node pkg="ros" type="roscore" name="my_master" respawn="true"/>
</launch>
```

Contoh lain, berikut adalah file roscore.xml yang mengubah alamat IP node master menjadi 192.168.0.1:

XML

```
<launch>
  <node pkg="ros" type="roscore" name="master" respawn="true">
    <param name="ip" value="192.168.0.1"/>
  </node>
</launch>
```

Anda dapat mengubah parameter node master sesuai dengan kebutuhan Anda.

Membuat paket ROS

Paket ROS adalah unit dasar dari 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

Sebuah sistem build bertanggung jawab untuk menghasilkan target (executable/library) dari sumber tekstual kode yang dapat digunakan oleh pengguna akhir. Pada distribusi yang lebih lama, seperti Electric dan Fuerte, rosbuilt adalah sistem pembangun. Karena berbagai kekurangan dari rosbuilt, catkin muncul.

Catkin memiliki banyak keuntungan, seperti:

Memungkinkan kita untuk memindahkan sistem kompilasi ROS lebih dekat ke Cross Platform Make (CMake).

Memiliki dokumentasi yang lebih baik.

Lebih mudah digunakan.

Membuat ruang kerja catkin

Persyaratan pertama untuk bekerja dengan paket ROS adalah membuat ruang kerja catkin ROS. Setelah menginstal ROS, kita dapat membuat dan membangun sebuah catkin workspace bernama `catkin_ws`:

```
mkdir catkin_ws
```

```
cd catkin_ws
```

```
catkin_make
```

Perintah `mkdir` membuat direktori baru bernama `catkin_ws`. Perintah `cd` berpindah ke direktori `catkin_ws`. Perintah `catkin_make` membangun ruang kerja catkin.

Chapter 2: Getting Started with ROS Programming

Creating a ROS package

Untuk membuat paket ROS, kita perlu membuat direktori baru dengan nama paket. Direktori ini harus berisi file package.xml. File package.xml adalah file metadata yang berisi informasi tentang paket.

Berikut adalah contoh file package.xml untuk paket ROS bernama my_package:

XML

```
<package>
  <name>my_package</name>
  <version>0.0.1</version>
  <description>This is a simple ROS package.</description>
  <maintainers>
    <maintainer>
      <name>Your Name</name>
      <email>your_email@example.com</email>
    </maintainer>
  </maintainers>
  <license>MIT</license>
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>std_msgs</run_depend>
</package>
```

Gunakan kode dengan hati-hati. Pelajari lebih lanjut

File package.xml berisi informasi berikut:

Nama paket

Versi paket

Deskripsi paket

Pemelihara paket

Lisensi paket

Build tool yang dibutuhkan

Ketergantungan build

Ketergantungan run

Untuk membuat paket ROS bernama my_package, kita dapat menggunakan perintah berikut:

```
mkdir my_package
```

```
cd my_package
```

```
catkin_create_pkg my_package roscpp std_msgs
```

Perintah `mkdir` membuat direktori baru bernama `my_package`. Perintah `cd` berpindah ke direktori `my_package`. Perintah `catkin_create_pkg` membuat paket ROS baru dengan nama `my_package`. Perintah ini juga membuat file `package.xml` untuk paket tersebut.

Menambahkan file ke paket ROS

Setelah membuat paket ROS, kita dapat menambahkan file ke paket tersebut. File-file ini dapat berupa kode, data, atau dokumen.

Untuk menambahkan file ke paket ROS, kita dapat menggunakan perintah `cp` untuk menyalin file ke direktori paket. Misalnya, untuk menyalin file `my_file.cpp` ke paket `my_package`, kita dapat menggunakan perintah berikut:

```
cp my_file.cpp my_package/
```

Kita juga dapat menggunakan perintah `catkin_add_file` untuk menambahkan file ke paket. Perintah ini juga akan menambahkan file tersebut ke file `CMakeLists.txt` untuk paket tersebut. Misalnya, untuk menambahkan file `my_file.cpp` ke paket `my_package`, kita dapat menggunakan perintah berikut:

```
catkin_add_file(my_package my_file.cpp)
```

```
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
#This will create executables of the nodes
add_executable(demo_topic_publisher src/demo_topic_publisher.
cpp)
add_executable(demo_topic_subscriber src/demo_topic_subscriber.
cpp)

#This will link executables to the appropriate libraries
target_link_libraries(demo_topic_publisher ${catkin_LIBRARIES})
target_link_libraries(demo_topic_subscriber ${catkin_
LIBRARIES})
```

XML yang disebut file peluncuran dan, dengan menggunakan perintah yang disebut `roslaunch`, kita mem-parsing file file ini dan meluncurkan node.

Perintah `roslaunch` akan secara otomatis memulai master ROS dan parameter server. Jadi, pada dasarnya, tidak perlu memulai perintah `roscore` dan perintah apa pun node; jika kita meluncurkan berkas, semua operasi akan dilakukan dalam satu perintah. Perhatikan bahwa jika Anda memulai sebuah node menggunakan perintah `roslaunch`, menghentikan atau memulai ulang ini akan memiliki efek yang sama dengan memulai ulang `roscore`.

Chapter 3: Working with ROS for 3D Modeling

Creating the ROS package for the robot description

Creating a ROS package for robot description

Purpose:

- Encapsulating robot model information within a ROS package for organization and sharing
- Integrating model with ROS tools for simulation, visualization, and control

Steps:

1. Create a new ROS package:

- Use `catkin_create_pkg` command:

Bash

```
catkin_create_pkg my_robot_description roscpp urdf xacro
```

- Replace `my_robot_description` with your desired package name.
- Dependencies:
 - `roscpp`: Core ROS libraries for communication and node creation
 - `urdf`: Unified Robot Description Format for representing robot structure
 - `xacro`: XML Macro language for simplifying URDF creation and maintenance

2. Create URDF file:

- Within the package, create a `urdf` subdirectory for URDF files.
- Define robot links, joints, and their connections using XML syntax.
- Example basic structure:

XML

```
<robot name="my_robot">
  <link name="base_link">
    <visual>
      <geometry>
        <box size="1 1 1"/>
      </geometry>
    </visual>
  </link>
</robot>
```

3. Utilize Xacro (optional):

- Employ Xacro for modularity and reusability:

- Define macros for common robot components.
- Simplify model creation and updates.
- Example Xacro usage:

XML

```
<robot name="my_robot">
  <xacro:include filename="$(find
my_robot_description)/urdf/base.xacro"/>
  <xacro:base_link />
</robot>
```

4. Generate URDF from Xacro (if used):

- Use `xacro` command to process Xacro files:

Bash

```
xacro my_robot.xacro > my_robot.urdf
```

5. Visualize robot model:

- Employ tools like Rviz for visual representation and verification.

Key takeaways:

- ROS packages streamline model organization and sharing.
- URDF provides a standardized format for robot description.
- Xacro simplifies model creation and maintenance.
- Visualization tools enable model inspection and debugging.

Chapter 4: Simulating Robots Using ROS and Gazebo

Simulating the robotic arm using Gazebo and ROS

Simulating robots provides a safe and efficient environment for testing algorithms, control strategies, and robot interaction with its surroundings before deployment in the real world. This chapter explores utilizing Gazebo, a robot simulator, and ROS, a robot software framework, to simulate a robotic arm.

Steps:

1. Preparing the environment:

- Ensure ROS and Gazebo are installed and configured.
- Download the URDF file representing your specific robotic arm model.

2. Creating a ROS package for the arm model (optional):

- If the URDF file isn't readily available in a ROS package, create one using

```
catkin_create_pkg:
```

Bash

```
catkin_create_pkg my_robot_arm_sim roscpp urdf xacro
```

- Place the downloaded URDF file within the package's `urdf` subdirectory.

3. Launching Gazebo with the arm model:

- Use `roslaunch` with the appropriate launch file:
- For packaged models:

Bash

```
roslaunch my_robot_arm_sim gazebo.launch
```

- For standalone URDF file:

Bash

```
roslaunch gazebo_ros launch_robot.launch  
robot_filename:=path/to/your_arm.urdf
```

4. Controlling the arm in simulation:

- Utilize ROS nodes:
 - `joint_state_publisher`: Set desired joint positions.
 - `joint_trajectory_controller`: Follow pre-defined trajectories.
 - Other controller nodes specific to your robot's functionality.

5. Interacting with the environment:

- Spawn obstacles, manipulate objects, and simulate various scenarios using Gazebo plugins and ROS nodes.

6. Visualizing and monitoring:

- Use Rviz to visualize the arm, environment, and sensor data.
- Monitor joint positions, velocities, and sensor readings through ROS topics.

Additional considerations:

- Sensor simulation: Integrate Gazebo plugins or ROS nodes to simulate sensors like cameras, LiDARs, and force/torque sensors.
- Advanced control algorithms: Implement and test complex control algorithms within the simulated environment.
- Debugging and troubleshooting: Analyze performance, identify issues, and refine your robot model and control strategies in simulation before real-world testing.

Conclusion:

Gazebo and ROS provide a powerful combination for simulating robotic arms and creating realistic virtual environments. By following these steps and exploring the vast capabilities of these tools, you can effectively develop and test your robot software and control strategies before venturing into the real world.

Chapter 5: Setting up CoppeliaSim with ROS

This chapter explores integrating CoppeliaSim, a robot simulator, with ROS, a robot software framework, for robot simulation.

CoppeliaSim and ROS integration benefits:

- Advanced physics engine: CoppeliaSim offers a robust physics engine for realistic dynamics and contact simulation.
- Rich visual environment: Create complex scenes with various objects, textures, and lighting effects.
- Open-source ROS interface: Easily connect CoppeliaSim with ROS nodes for control and communication.
- Python scripting: Extend simulator functionality and create custom behavior using Python scripts.

Getting started:

1. Install CoppeliaSim: Download and install the appropriate version for your needs.
2. Install ROS: Ensure ROS is installed and configured.
3. Download the CoppeliaSim ROS plugin: Access the plugin from the CoppeliaSim website or official ROS package repositories.
4. Load the plugin in CoppeliaSim: Open Scene > Preferences > Add-ons > Robotics > ROS.
5. Configure the plugin: Set the ROS master address and port (usually localhost:11311).
6. Verify connection: Use `rostopic list` in a terminal to see CoppeliaSim ROS nodes listed.

Building your simulation:

1. Create a ROS package for your robot: (Similar to Chapter 3)
 - Include dependencies like `roscpp`, `geometry_msgs`, and `sensor_msgs`.
2. Develop ROS nodes:
 - `joint_state_publisher`: Control robot joint positions.
 - `joint_trajectory_controller`: Follow pre-defined trajectories.
 - `image_transport`: Publish and subscribe to camera data.
 - Specific nodes for your robot's sensors and functionalities.
3. Import your robot model in CoppeliaSim:
 - Create or import a 3D model representing your robot.
 - Define joints, motors, and sensors within the model.

4. Connect CoppeliaSim objects to ROS topics:
 - In CoppeliaSim, right-click an object and select "Simulation > Remote API > Add signal...".
 - Choose the appropriate ROS topic for communication.
5. Run your ROS nodes and launch the simulation:
 - Start the ROS nodes controlling your robot.
 - Run the simulation in CoppeliaSim.

Additional notes:

- Explore CoppeliaSim scripting with Python to further customize behaviors and interactions.
- Utilize CoppeliaSim's built-in sensors and objects like cameras, force sensors, and proximity sensors.
- Leverage visualization tools like Rviz to monitor sensor data and robot performance.

Conclusion:

Combining CoppeliaSim and ROS empowers you to build intricate and realistic simulations for development, testing, and refinement of your robots and algorithms. Remember, specific steps and configurations might vary depending on your robot model, sensors, and chosen functionalities. Refer to the official documentation and tutorials for both CoppeliaSim and ROS for detailed instructions and examples.