Robotics
Laboratory

Christian Rossi

Academic Year 2023-2024

Abstract

The course is composed by a set of lectures on autonomous robotics, ranging from the main architectural patterns in mobile robots and autonomous vehicles to the description of sensing and planning algorithms for autonomous navigation. The course outline is:

- Mobile robots' kinematics.
- Sensors and perception.
- Robot localization and map building.
- Simultaneous Localization and Mapping (SLAM).
- Path planning and collision avoidance.
- Robot development via ROS.

Contents

CHAPTER 1

Introduction

1.1 Middleware origins and usage

Middleware was introduced by d'Agapeyeff in 1968, the concept of a wrapper emerged in the 1980s as a bridge between legacy systems and new applications. Today, it is prevalent across various domains, including robotics. Examples outside of robotics include Android, SOAP, and Web Services.

The concept of Middleware is widely recognized in software engineering. It serves as a computational layer, acting as a bridge between applications and low-level details. It's important to note that Middleware is more than just a collection of APIs and libraries.

Issues The challenge lies in fostering cooperation between hardware and software components. Robotics systems face architectural disparities that affect their integration. Ensuring software reusability and modularity is also a critical concern in this context.

Main features The key attributes of middlewares include:

- *Portability*: offering a unified programming model irrespective of programming language or system architecture.
- *Reliability*: middlewares undergo independent testing, enabling the development of robot controllers without the need to delve into low-level details, while leveraging robust libraries.
- *Manage the complexity*: handling low-level aspects through internal libraries and drivers within the middleware, thereby reducing programming errors and speeding up development time.

1.2 Middlewares

Several middleware have been developed in recent years:

- *OROCOS*: originating in December 2000 as an initiative of the EURON mailing list, OROCOS evolved into a European project with three partners: K.U. Leuven (Belgium),

LAAS Toulouse (France), and KTH Stockholm (Sweden). The requirements for OROCOS encompass open source licensing, modularity, flexibility, independence from specific robot industries, compatibility with various devices, software components covering kinematics, dynamics, planning, sensors, and controllers, and a lack of dependency on a singular programming language. The structure includes:

  - Real-Time Toolkit (RTT): offering infrastructure and functionalities tailored for real-time robot systems and component-based applications.
  - Component Library (OCL): supplying ready-to-use components such as drivers, debugging tools, path planners, and task planners.
  - Bayesian Filtering Library (BFL): featuring an application-independent framework encompassing (Extended) Kalman Filter and Particle Filter functionalities.
  - Kinematics and Dynamics Library (KDL): facilitating real-time computations for kinematics and dynamics.

- *ORCA*: the project's objective is to emphasize software reuse in both scientific and industrial applications. Its key properties include enabling software reuse through the definition of commonly-used interfaces, simplifying software reuse via high-level libraries, and encouraging software reuse through regularly updated software repositories. ORCA defines itself as an "unconstrained component-based system".

  The primary distinction between OROCOS and ORCA lies in their communication toolkits. OROCOS utilizes CORBA, whereas ORCA employs ICE. ICE, a contemporary framework created by ZeroC, functions as an open-source commercial communication system. ICE offers two fundamental services: the IceGrid registry (Naming service), which facilitates logical mapping between various components, and the IceStorm service (event service), which forms the foundation for publisher-subscriber architecture.

- *OpenRTM*: RT-Middleware (RTM) serves as a widely adopted platform standard for assembling robot systems by integrating software modules known as robot functional elements (RTCs). These modules include components such as camera, stereo vision, face recognition, microphone, speech recognition, conversational, head and arm, and speech synthesis. OpenRTM-AIST (Advanced Industrial Science and Technology) is built upon CORBA technology to realize the extended specifications of RTC implementation.

- *BRICS*: the objective is to uncover the most effective strategies for developing robotic systems by examining several critical areas: Initially, by thoroughly examining the weaknesses found in current robotic projects. Subsequently, by delving into the integration between hardware and software components within these systems. Thirdly, by advocating for the incorporation of model-driven engineering principles in the development process. Moreover, by creating a tailored Integrated Development Environment (IDE) specifically for robotic projects, known as BRIDE Finally, by defining benchmarks aimed at evaluating the strength and effectiveness of projects based on BRICS principles.

- *ROS*: introduced by Willow Garage in 2009, the Robot Operating System (ROS) serves as a meta-operating system tailored for robotics, boasting a diverse ecosystem replete with tools and programs. ROS has expanded to encompass a vast global community of users. The developer community stands out as one of ROS's most significant features.

CHAPTER 2

Robot Operating System

2.1 Introduction

The key features of ROS:

1. *Decentralized framework*: ROS operates on a decentralized architecture, facilitating robust communication and coordination among various components.
2. *Code reusability*: ROS promotes the reuse of code, allowing developers to efficiently leverage existing modules and algorithms for faster development.
3. *Language neutrality*: ROS supports multiple programming languages, enabling developers to work in their preferred language while seamlessly integrating with the ROS ecosystem.
4. *Seamless real robot and simulation testing*: ROS provides an environment for easy testing on both real robots and simulations, ensuring smooth transition from development to deployment.
5. *Scalability*: ROS is designed to scale efficiently, accommodating projects of various sizes and complexities without compromising performance or functionality.

ROS is composed by the following elements:

- *File system utilities*: ROS provides tools for managing files and directories, simplifying data organization and access within the ROS environment.
- *Construction tools*: ROS offers construction tools to streamline the development process, facilitating the creation and configuration of robotic systems and components.
- *Package management*: ROS includes robust package management capabilities, allowing users to easily install, update, and manage software packages and dependencies.
- *Monitoring and graphical user interfaces* (GUIs): ROS features monitoring tools and graphical user interfaces to visualize and analyze system behavior, aiding in debugging, optimization, and user interaction.
- *Data logging*: ROS supports data logging mechanisms for recording and analyzing sensor data, system state, and other relevant information, facilitating research, analysis, and debugging tasks.

2.2 Computational graph

The computation graph is the peer-to-peer network of ROS processes that are processing data together.

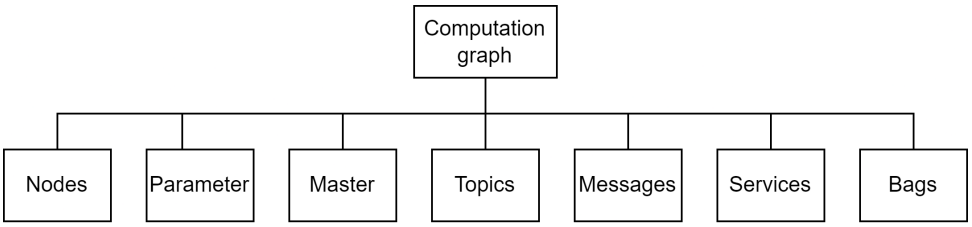

Figure 2.1: ROS computational graph

2.2.1 Nodes

ROS executable units include Python scripts and C++ compiled source code. They are processes responsible for computation within the ROS framework. These units, referred to as nodes, exchange information through a graph structure. Nodes are designed to operate at a fine-grained scale and are integral components of robot systems, which are composed of multiple interconnected nodes. To initiate a node, the following command is utilized:

```
rosrun package_name node_name
```

2.2.2 Master

The ROS Master serves as the central hub, facilitating naming and registration services crucial for node interactions. In every system, including distributed architectures, there exists a single master. This centralized entity enables ROS nodes to effectively locate each other.

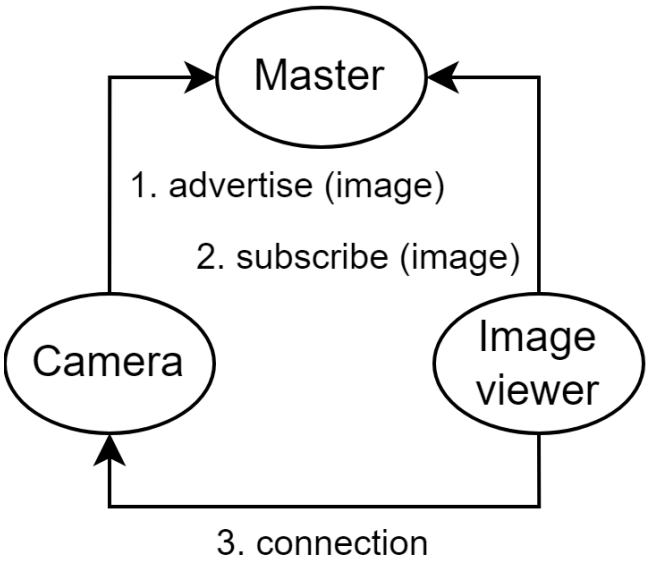

Figure 2.2: ROS master

In this scenario, the camera node advertises the master with an image, making it available on a specific topic. Subsequently, the image viewer node subscribes to the same topic through

the master. Once the requested topic is found, the ROS master facilitates communication by providing the node name, allowing the subscriber to establish communication with the node that generated the image. This interaction demonstrates the pivotal role of the ROS master in mediating communication between nodes in the ROS ecosystem.

2.2.3 Topics

Topics serve as named channels for communication within the ROS framework, implementing the publish/subscribe paradigm. They facilitate communication between nodes by allowing multiple nodes to publish messages on a topic and multiple nodes to read those messages from the same topic. Topics are associated with specific message types, but they do not guarantee message delivery.

It's advisable to avoid scenarios where multiple talkers are connected to the same topic in ROS. This is because ROS lacks a mechanism to discern the origin of a message in such cases.

2.2.4 Messages

Communication on topics involves the exchange of messages, which define the type of information being transmitted. A wide range of predefined message types is available. Additionally, it is possible to create custom messages using a straightforward language. These custom messages can incorporate existing message types along with base types as needed. The available message types include:

```
std_msgs/Header.mgs
// uint32 seq
// time stamp
// string frame_id

std_msgs/String.msg
// string data

sensor_msgs/Joy.msg
// std_msgs/Header header
// float32[] axes
// int32[] buttons
```

2.2.5 Services

Services function akin to remote function calls within ROS, adhering to the client/server paradigm. When a service call is made, the code waits for its completion, ensuring a guarantee of execution. Services utilize message structures for their operation.

2.2.6 Parameter server

The parameter server is a shared, network-accessible multivariable dictionary utilized by nodes for storing and accessing parameters during runtime. It's not optimized for performance or data exchange. This server is linked to the master, constituting one of the functionalities provided by ROScore. The procedure for accessing and modifying data in the parameter server is accomplished using the following command:

```
rosparam [set|get] name value
```

The parameter server can store a range of data types, including 32-bit integer, Boolean, string, double, ISO8601 date, and base64-encoded binary data.

2.2.7 Bags

Bags serve as a file format (*.bag) designed for storing and replaying messages within ROS. They constitute the primary mechanism for data logging, capable of recording all exchanges occurring on the ROS graph, including messages, services, parameters, and actions. Bags are essential tools for data analysis, storage, visualization, and algorithm testing within the ROS ecosystem. The commands used to modify the bag is:

```
rosbag record -a
```

2.2.8 ROScore

ROScore is a foundational component of ROS-based systems, comprising nodes and programs essential for operation. It must be active to facilitate communication among ROS nodes. Launching ROScore via the command initiates its elements, which include a ROS master, a ROS parameter server, and a ROSout logging node.

2.3 ROS file system

The foundation of the ROS file system revolves around packages, depicted in the diagram below.

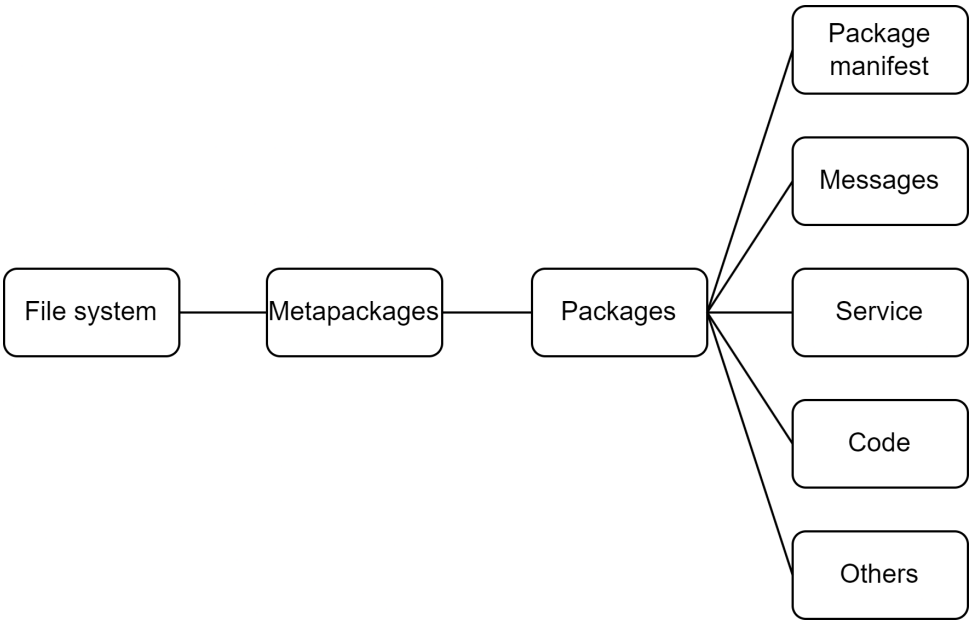

Figure 2.3: ROS file system

Packages serve as the fundamental units within the ROS file system. They are essential references for most ROS commands and encompass nodes, messages, and services. Each package is described by a `package.xml` file and acts as a mandatory container for its contents.

Additionally, there are metapackages, which aggregate logically related elements. Unlike packages, metapackages are not directly utilized when navigating the ROS file system. They contain other packages and are described by a package.xml file as well, but they are not obligatory components.

2.3.1 Packages

Packages serve as fundamental units within the ROS file system, acting as the cornerstone for most ROS commands. They encapsulate nodes, messages, and services, providing a structured organization for ROS functionalities. Each package is described by a package.xml file, serving as a mandatory container that delineates the package's properties and dependencies. The general structure of a package includes the following components:

- Folder Structure:
  - /src, /include, /scripts (for coding).
  - /launch (for launch files).
  - /config (for configuration files).
- Required Files:
  - CMakeLists.txt: contains build rules for catkin.
  - package.xml: provides metadata for ROS.

2.3.2 Metapackages

Metapackages are collections of logically related elements within ROS, serving as aggregations rather than individual components. Unlike regular packages, they are not typically used when navigating the ROS file system. Metapackages encompass other packages and are described by a package.xml file, yet they are not obligatory structures within ROS.

CHAPTER 3

ROS commands and files

3.1 Basic commands

To initiate ROScore within a tmux terminal, execute the following command:

```
roscore
```

3.1.1 Topics

To view active topics, use the command:

```
rostopic list
```

To publish a new permanent topic, utilize:

```
rostopic pub /topic_name message_type message_name data
```

For publishing a new temporary topic, use:

```
rostopic pub --once /topic_name message_type message_name data
```

To publish a new topic at a specific frequency, employ:

```
rostopic pub -r frequency /topic_name message_type message_name data
```

To check the running frequency of a particular topic, execute:

```
rostopic hz /topic_name
```

To display the content of a specific topic, input:

```
rostopic echo /topic_name
```

To view information about a topic with a graphical user interface (GUI), employ:

```
rqt_topic
```

3.1.2 Nodes

To view active nodes, utilize the command:

```
rosnode list
```

To access information about a specific node, use:

```
rosnode info /name
```

For viewing information about a specific node with a graphical user interface (GUI), use:

```
rqt_graph
```

To initiate a node, the command is:

```
rosrun package_name node_name
```

For redirecting the topic associated with a particular node, utilize:

```
rosrun package_name node_name /old_topic:=/new_topic
```

To generate a node resembling an existing one but with an altered name, execute:

```
rosrun \package_name \node_name --name:=\new_node_name
```

3.1.3 Services

To invoke a service, employ:

```
rosservice call /service_name parameters
```

3.2 Bags and saves

To record a series of commands given to a specific topic we write:

```
rosbag record /topic_name
```

To check the data inside a bag we write

```
rosbag info bag_name.bag
```

To play the data inside a bag we write

```
rosbag play bag_name.bag
```

To play in loop the data inside a bag we write

```
rosbag play -l bag_name.bag
```

To visualize data from a playing a bag we write:

```
rviz
```

Alternatively, to visualize simple numeric data we write:

```
rosrun plotjuggler plotjuggler
```

And select ROS topic subscriber as the source streaming. From the time series list we can finally select the specific topic.

3.2.1 Tf

Tf files are included into a bag file. They can be also static, if the element does not change during time, and it is published only at the beginning of the execution.

3.3 Publisher

The structure of a publisher is as follows:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

int main(int argc, char **argv) {

// initialize the node with the name "talker"
// ros::init_options::AnonymousName initialize the node without a name
ros::init(argc, argv, "talker");

// handle used to call all the ros releated functions
ros::NodeHandle n;

// create the publisher called "chatter_pub" with the type of message
// the message is a simple "String" and it is publish on the topic "chatter"
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1);

// we want a loop with a running frequency of ten
ros::Rate loop_rate(10);

int count = 0;

// automatic check on the condition
while (ros::ok()) {

        // message "msg" creation
        std_msgs::String msg;

        // setting the field in the message, "String" has only this field
        msg.data = "hello world!";

        // print on the log console of the message
        ROS_INFO("%s", msg.data.c_str());

        // message publishing
        chatter_pub.publish(msg);

        // used for the loop to let the system know that one loop is done
        ros::spinOnce();
```

```
        // sleep for the requested time
        loop_rate.sleep();
        ++count;
}
return 0;
}
```

3.4 Subscriber

The structure of a subscriber is as follows:

```
#include "ros/ros.h"
#include "std_msgs/String.h"

// code executed when we received a message
void chatterCallback(const std_msgs::String::ConstPtr& msg) {
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {

    // initialize the node with the name "listener"
    ros::init(argc, argv, "listener");

    // handle used to call all the ros releated functions
    ros::NodeHandle n;

    // create the subsriber called "sub" with the topic "chatter"
    // the function "chatterCallback" is called when something is published
    ros::Subscriber sub = n.subscribe("chatter", 1, chatterCallback);

    // standard used outside a loop when the code is executed
    ros::spin();

    return 0;
}
```

3.5 Other files

The CMakeLists file has the following structure:

```
cmake_minimum_required(VERSION 2.8.3)
// name of the package
project(pub_sub)

## Find catkin and any catkin packages
find_package(catkin REQUIRED COMPONENTS roscpp std_msgs)
```

```
## Declare a catkin package
catkin_package()

## Build talker and listener
include_directories(include ${catkin_INCLUDE_DIRS})

// publisher executables
add_executable(pub src/pub.cpp)
target_link_libraries(pub ${catkin_LIBRARIES})

// subscriber executables
add_executable(sub src/sub.cpp)
target_link_libraries(sub ${catkin_LIBRARIES})
```

The package.xml file has the following structure:

```
<?xml version="1.0"?>
<package format="2">
  <!-- package name  -->
  <name>pub_sub</name>
  <version>0.0.0</version>
  <description>The pub_sub package</description>

  <!-- One maintainer tag required, multiple allowed, one person per tag -->
  <!-- Example:  -->
  <!-- <maintainer email="jane.doe@example.com">Jane Doe</maintainer> -->
  <maintainer email="simone@todo.todo">simone</maintainer>

  <!-- One license tag required, multiple allowed, one license per tag -->
  <!-- Commonly used license strings: -->
  <!--   BSD, MIT, Boost Software License, GPLv2, GPLv3, LGPLv2.1, LGPLv3 -->
  <license>TODO</license>

  <!-- Url tags are optional, but multiple are allowed, one per tag -->
  <!-- Optional attribute type can be: website, bugtracker, or repository -->
  <!-- Example: -->
  <!-- <url type="website">http://wiki.ros.org/pub_sub</url> -->

  <!-- Author tags are optional, multiple are allowed, one per tag -->
  <!-- Authors do not have to be maintainers, but could be -->
  <!-- Example: -->
  <!-- <author email="jane.doe@example.com">Jane Doe</author> -->


  <!-- The *depend tags are used to specify dependencies -->
  <!-- Dependencies can be catkin packages or system dependencies -->
  <!-- Examples: -->
  <!-- Shortcut for packages that are both build and exec dependencies -->
  <!--   <depend>roscpp</depend> -->
```

```
  <!--   Note that this is equivalent to the following: -->
  <!--   <build_depend>roscpp</build_depend> -->
  <!--   <exec_depend>roscpp</exec_depend> -->
  <!-- Use build_depend for packages you need at compile time: -->
  <!--   <build_depend>message_generation</build_depend> -->
  <!-- Use build_export_depend for packages to build against this package: -->
  <!--   <build_export_depend>message_generation</build_export_depend> -->
  <!-- Use buildtool_depend for build tool packages: -->
  <!--   <buildtool_depend>catkin</buildtool_depend> -->
  <!-- Use exec_depend for packages you need at runtime: -->
  <!--   <exec_depend>message_runtime</exec_depend> -->
  <!-- Use test_depend for packages you need only for testing: -->
  <!--   <test_depend>gtest</test_depend> -->
  <!-- Use doc_depend for packages nedded for building documentation: -->
  <!--   <doc_depend>doxygen</doc_depend> -->
  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_export_depend>roscpp</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>
  <exec_depend>roscpp</exec_depend>
  <exec_depend>std_msgs</exec_depend>

  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here -->

  </export>
</package>
```

3.6 Project compilation

To compile the file, ensure you're located in the directory `catkin_ws` and execute `catkin_make`. Upon successful completion, verify the presence of a folder named `build`. Afterwards, initiate the nodes with the following command:

```
rosurun \package_name \node_name
```

To initiate a package, the `launch` file within the launch directory is utilized. The format of this file follows:

```
<launch>
 <group ns="group_name">
    <node pkg="package_name" name="initial_name" type="node_name"/>
 </group>
</launch>
```

To execute this file, the command employed is:

```
roslaunch file_name.launch
```

3.7 Custom messages

To define custom messages in ROS, you typically create them inside a folder named `msg` within a package, often referred to as `custom_messages`. Here's a step-by-step guide:

1. Create a folder named `custom_messages` inside your ROS package.
2. Inside `custom_messages`, create a folder named `msg`.
3. In the `msg` folder, create a file with a `.msg` extension and a name of your choice. This file will contain the definition of your custom message.
4. Open the `package.xml` file of your package and uncomment the following lines to declare dependencies on message generation and runtime:

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

5. Inside the `CMakeLists.txt` file located in the `custom_messages` folder, add the following lines:

```
find_package(
catkin REQUIRED COMPONENTS
roscpp
rospy
std_msgs
message_generation)

catkin_package(
CATKIN_DEPENDS message_runtime)

add_message_files(
FILES
Num.msg)

generate_messages(
DEPENDENCIES
std_msgs)
```

6. After modifying the `CMakeLists.txt`, make sure to build your ROS workspace by running `catkin_make` or `catkin_build`.

With these steps completed, you can include your custom messages in the header files of publishers or subscribers that require them. To do this, include the header file generated for your custom message in your source code. For example, if your custom message is named `MyCustomMessage`, you would include it in your source file like this:

```
#include "package_name/MyCustomMessage.h"
```

Now you can use `MyCustomMessage` in your code as needed.

3.8 Services

To create a new service in ROS, you typically follow these steps:

1. Create a folder named `service` within your ROS package.
2. Inside the `service` folder, create a folder named `srv`.
3. Within the `srv` folder, create a file with a `.srv` extension. This file will define your service request and response format, with inputs and outputs separated by a line with three dashes (`---`).
4. In the `src` folder of your package, define a `.cpp` file that implements the service.
5. In your `.cpp` file, initialize the node and create a function that executes the service. This function should return `true` to send the response.
6. Optionally, define a client that calls the service with the proper values.

To call the service, use the command:

```
rosrun service client_name parameters_list
```

3.9 Timer

Inside the `timer` directory, there exists a `src` folder housing files with the extension `.cpp`. The layout resembles that of a subscriber, but it includes a timer initialized as follows:

```
ros::Timer timer = a.createTimer(ros::Duration(1), timerCallback);
```

This line sets up a ROS timer to execute a callback function at intervals of 1 second.

3.10 Service parameters

To prevent the need for recompilation of the entire node when altering parameters, global parameters can be defined. These parameters are typically specified in a `cpp` file within the primary `src` folder.

After defining the parameter file, it should be included in the launch file like so:

```
<launch>
 <param name="name" value="second" />
 <node pkg="parameter_test" name="param" type="param" output="screen"/>
</launch>
```

CHAPTER 4

Advanced topics

4.1 Message filters

Message filters are useful for synchronizing multiple topics, requiring the topics to have a header and timestamp. They can synchronize with either exact or approximate time. For camera topics, a custom version of synchronization is used.

Message filters without policy The message filters without policy are initialized as follows:

```
message_filters::Subscriber<geometry_msgs::Vector3Stamped>
                                                sub1(n, "topic1",1);
message_filters::Subscriber<geometry_msgs::Vector3Stamped>
                                                sub2(n, "topic2",1);
message_filters::TimeSynchronizer<geometry_msgs::Vector3Stamped,
geometry_msgs::Vector3Stamped> sync(sub1, sub2, 10);
sync.registerCallback(boost::bind(&callback, _1, _2));
```

Message filters with policy The message filters with policy are initialized as follows:

```
typedef
message_filters::sync_policies::ExactTime<geometry_msgs::Vector3Stamped,
geometry_msgs::Vector3Stamped> MySyncPolicy;

message_filters::Synchronizer<MySyncPolicy>
                                  sync(MySyncPolicy(10), sub1, sub2);
sync.registerCallback(boost::bind(&callback, _1, _2));
```

4.2 ActionLib

Node A sends a request to Node B to perform a task. There are two primary methods for handling these requests:

- *Services*:

  - Designed for tasks with small execution times.
  - The requesting node can wait for the task to complete.
  - No status updates or cancellation options are available.

- *Actions*:
  - Suitable for tasks with long execution times.
  - The requesting node cannot wait for the task to complete.
  - Provides status monitoring and cancellation options.

The `actionlib` package in ROS serves as an implementation similar to threads, based on a client/server paradigm. It provides tools to:

- Create servers that execute long-running tasks, which can be preempted.
- Create clients that interact with these servers.
- The ActionClient and ActionServer communicate using the ROS Action Protocol, which is built on top of ROS messages.

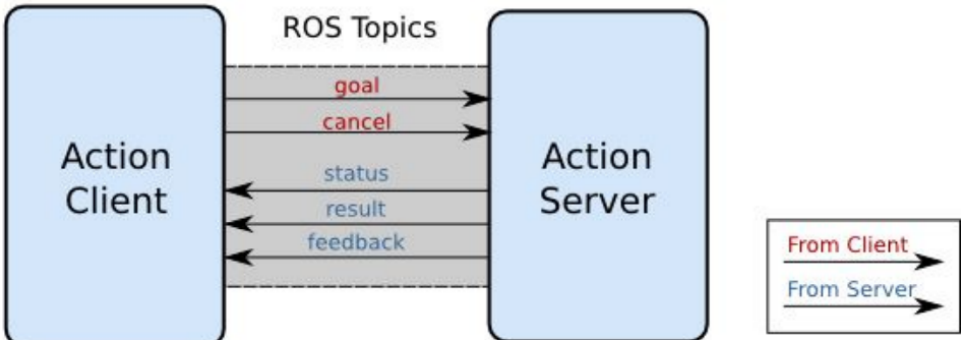

Figure 4.1: Client-server interaction

The main functions of the actionlib package include:

- *Goal*: Sends new goals to the server.
- *Cancel*: Sends cancellation requests to the server.
- *Status*: Notifies clients of the current state of every goal in the system.
- *Feedback*: Sends clients periodic auxiliary information for a goal.
- *Result*: Sends clients one-time auxiliary information upon completion of a goal.

Action templates are defined by a name and additional properties through a .action structure in ROS. Each instance of an action has a unique Goal ID, which provides a robust way for the action server and action client to monitor the execution of a particular instance of an action.

4.3 Rosbag tools

Rosbag tools allow for various operations directly on bag files, including converting, extracting data, and editing. These tools are accessible via both Python and C++ APIs.

Bag to CSV To transform a bag file into a CSV file, use the following code:

```
import rosbag
import csv
from turtlesim.msg import Pose

bag = rosbag.Bag('/home/airlab/robotics/bags/pose.bag')
f = open('/home/airlab/robotics/bags/pose.csv', 'w')
writer = csv.writer(f)
header = ['timestamp', 'x','y','theta' ]
writer.writerow(header)
```

CSV to bag To transform a CSV file into a bag file, use the following code:

```
for topic, msg, t in bag.read_messages(topics=['/turtle1/pose']):
    writer.writerow ([t,msg.x, msg.y,msg.theta])
bag.close()
f.close()
```

Edit a bag To edit a bag file, use the following code:

```
import rosbag
import csv
from turtlesim.msg import Pose

bag_in = rosbag.Bag('/home/airlab/robotics/bags/pose.bag')
bag_out = rosbag.Bag('/home/airlab/robotics/bags/pose_shifted.bag', 'w')
for topic, msg, t in bag_in.read_messages(topics=['/turtle1/pose']):
    msg.x +=2
    t.secs += 30
    bag_out.write ('/turtleS/pose',msg, t)
bag_in.close()
bag_out.close()
```

4.4 ROS distributed

ROS can function as a distributed system across multiple devices interconnected on the same network. Large projects typically leverage distributed systems for scalability and efficiency. Remote monitoring of robots becomes straightforward with a single ROS network.

To utilize ROS across multiple devices, it's essential to designate one device to run the ROS master, achieved by executing the command `roscore` on that device exclusively. For all other nodes, specifying the IP address of the master is necessary. To find your IP, use the `ifconfig` command and look for `inet addr`.

To ensure proper configuration, export all required variables, and append them to your `/.bashrc` file to apply them automatically for every new terminal session:

```
$ gedit ~/.bashrc
```

Master configuration Begin by setting the master's IP:

```
export ROS_MASTER_URI=http://master_ip:11311
```

Next, inform the ROS master of your IP:

```
export ROS_IP=master_ip
```

Client configuration Similarly, configure the client by setting the master's IP:

```
export ROS_MASTER_URI=http://master_ip:11311
```

And specifying the client's IP to the ROS master:

```
export ROS_IP=master_ip
```

On the master PC, initiate the ROS core with the command `roscore`. To verify proper functionality on the clients, open a new terminal and execute `rostopic list` without initiating `roscore` previously. You should observe topics on the ROS network. Now, all clients are interconnected and capable of communication and node initiation on the distributed ROS network.

Time synchronization Recording high-throughput bags often necessitates splitting recordings across different ROS devices. To utilize these bags collectively, timestamp coherence is crucial. Hence, synchronizing the clocks of all devices on the ROS network is necessary.

The standard method involves employing an NTP server on the master device and configuring chrony clients on all other devices.

1. Install the NTP server on the master and chrony on other nodes.
2. Modify the chrony configuration file located at `/etc/chrony/chrony.conf`.
3. After making changes, stop and restart chrony for them to take effect:

   ```
   $ sudo service chrony stop
   $ sudo service chrony start
   ```

4. Monitor synchronization progress:

   ```
   $ chronyc tracking
   ```

4.5 Latched messages

In ROS, latched messages provide a mechanism where subscribers receive the last published message upon subscribing, particularly useful for low-frequency publishers such as maps. To implement this in code, when creating a publisher, set the third argument to true:

```
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1, true);
```

With this configuration, subscribers don't need to wait for new published messages; they immediately receive the last published message upon subscription.

4.6 Asynchronous spinner

In standard ROS operation, the conventional approach involves dedicating one thread to each node. While this works well for many scenarios, there are situations where a more dynamic and flexible approach is required.

An asynchronous spinner offers a solution without necessitating explicit thread management. It allows for real thread implementation behind the scenes, efficiently managing multiple subscribers, especially in cases where computations are time-consuming.

This approach becomes invaluable in scenarios where actions are unsuitable, and it's preferred to handle everything within the same node. Asynchronous spinners streamline the processing flow, enhancing performance and resource utilization.

CHAPTER 5

Navigation

5.1 Introduction

The Robot Operating System (ROS) offers a robust platform for implementing navigation capabilities in robotic systems, leveraging a wide array of pre-existing, high-quality components. The ROS navigation stack, in particular, stands out for its comprehensive suite of tools designed to facilitate path planning, obstacle avoidance, and localization.

The primary elements of the ROS navigation stack include:

- `move_base`: this is the central node within the ROS navigation stack, responsible for integrating all planning and control functionalities. It relies on dynamically configurable plugins via ROS's `pluginlib`, allowing for easy extension and customization. The core logic is built upon the `nav_core` class.

- `nav_core`: this class serves as a central interface for navigation goals and velocity commands. It supports plugins that can be swapped out at runtime, enabling flexible and dynamic updates to the navigation logic. It relies on sensor data and map information provided by the map server to function effectively.

- `costmap`: a crucial component that processes sensor data to generate a 2D or 3D occupancy grid, where each cell represents a different cost value. The costmap is divided into two types:

  - *Global costmap*: used for long-term path planning across the entire environment.
  - *Local costmap*: focuses on short-term planning and obstacle avoidance in the robot's immediate vicinity.

  Both costmap have distinct configurations but share some common settings.

- `map_server`: this tool is essential for publishing and saving maps. It provides map data through both topics and services and can handle dynamically generated maps. When combined with `costmap_2d`, it manages multi-layered 2D maps, inflating obstacles based on sensor inputs. The map consists of a YAML file for map metadata, and an image file encoding occupancy data.

- `amcl`: the Adaptive Monte Carlo Localization (AMCL) system is a probabilistic localization method that uses a 2D map. It requires laser scan data and benefits from odometry information to estimate the robot's position within the global frame. AMCL transforms laser scan data to the odometry frame and publishes the transformation between the global and odometry frames.

Other elements Certain platform-specific elements need to be implemented manually, including:

- Low-level robot interaction.
- Sensor drivers.
- Sensor measurement processing.
- Odometry estimation.
- High-level task planning.

Many of these components are available as existing ROS packages, such as drivers and `robot_pose_ekf`.

5.1.1 Requirements

For effective operation, the ROS Navigation stack requires:

- Sensors for localization and obstacle avoidance, such as `sensor_msgs/LaserScan` or `sensor_msgs/Poi`
- A source of odometry, such as `nav_msgs/Odometry`.
- Mechanisms to convert `geometry_msgs/Twist` messages into motor commands.
- A well-formed tf tree to accurately represent the positions of sensors, the robot, and the map.

While the ROS Navigation stack is versatile and adaptable, optimal performance is achieved under certain hardware conditions:

- It works best with differential drive or holonomic robots.
- It requires a planar laser scanner for effective scanning and localization.
- It performs best with robots that have square or circular shapes.

5.2 Mapping

Mapping in ROS is accomplished using `gmapping`, a ROS wrapper for the OpenSLAM GMapping algorithm. This Simultaneous Localization and Mapping algorithm facilitates real-time map creation and localization using laser scans and odometry data.

Requirements To utilize `gmapping`, the following are required:

- Odometry data.
- A horizontally-mounted, fixed laser range-finder
- A complete tf tree that includes base to laser transformation, and base to odometry transformation.

Usage To generate a map using `gmapping`, follow these steps:

1. Drive your robot around the environment.
    - Explore the entire area you want to map.
    - Collect as much data as possible.
    - Create loops to provide the algorithm with reference points.
2. Save the collected data in a ROS bag file.
3. Play back the bag file.
4. Start `gmapping` to process the data.
5. Save the generated map.

Bag and real time Using bag files:

- Processing is faster since you can use pre-collected data.
- Allows for multiple trials and parameter tuning.
- Useful for experimenting with different settings without needing to rerun the physical robot.

Processing in real-time:

- Immediate feedback allows for early stopping if issues arise.
- Ability to restart quickly in case of problems.
- Direct visualization of results ensures complete coverage of the mapped area.

APPENDIX A

Other useful commands

A.1 Tmux

To create a new terminal session, use:

```
tmux new -s session1
```

To exit the current terminal session, either:

```
tmux detach
```

or press: ctrl + b + d.
To join an existing session, execute:

```
tmux a
```

To split the terminal into multiple sections:

- Horizontal split: ctrl + b + ".
- Vertical split: ctrl + b + %

To exit a split section, input:

```
exit
```

To navigate between terminals, press ctrl + b followed by the arrow keys.
To check active tmux sessions, use:

```
tmux ls
```

To join a specific session, utilize:

```
tmux a -t session1
```

For scrolling the page, press: ctrl + b + [.