Chapter 12

Service Robot

12.1. Delivery Service Robot

SLAM and navigation are applied in various environments such as robotics for automobile, factories and production lines and service robots to carry and deliver objects. Its technology is rapidly improving. Chapter 10 and 11 covered about SLAM and navigation packages. In chapter 12, let's build an actual operational service robot based on the knowledge we have acquired in the previous chapters.

12.2. Configuration of a Delivery Service Robot

12.2.1. System Configuration

The system design of the delivery service robot can be divided into 'Service Core', 'Service Master', and 'Service Slave' as shown in Figure 12-1.

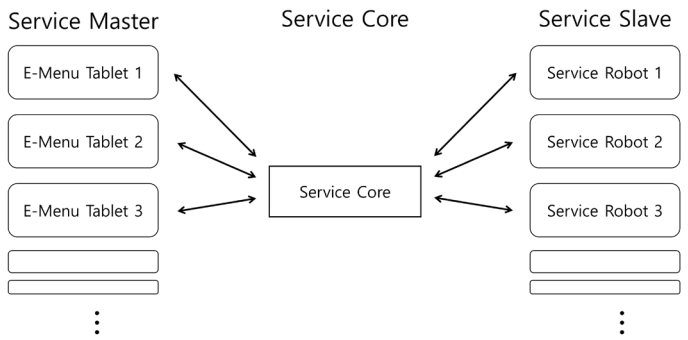

FIGURE 12-1 System design example of the delivery service robot

Service Core

Service core is a kind of database that manages the order status of customers and the service performance status of a robot. Upon receiving an order from a customer, it plays the key role in handling the order and scheduling the robot's service process. Each particular order and processing system is unique and affects the order of other customers, thus, every service core must exist as one core.

Service Master

A service master receives the customer's orders and transfers the order details to the service core. It also displays a list of items that can be ordered and notifies the service status to the customer. For the service master to do this, the service core's database must be synchronized.

Service Slave

As a robot platform and physical object that processes an order, the service slave updates the service status of the order to the service core in real time.

12.2.2. System Design

The service master, service slave, and service core can be as little as a single computer or as many as distributed to one per computer. However, a single computer may not be able to process all the work when running all the process on one computer and on the otherhand, heavily distributing to each computer may lead to overload on the wireless communication. Thus, careful allocation is critical for proper processing. Also since ROS 1.x is a system created to control one robot, in order to use ROS on multiple computers the following command should be entered to reduce the time difference between each computer.

```
$ sudo ntpdate ntp.ubuntu.com
```

For example, as in Figure 12-2, I used the following devices to implement a delivery service robot system.

- **Service Core**: NUC i5 (3 computers to execute roscore and service core)
- **Service Master**: 3 SAMSUNG NOTE 10.4 Android OS
- **Service Slave**: 3 Intel Joule 570x of TurtleBot3 Carrier (TurtleBot3 Waffle customized for a delivery robot)

FIGURE 12-2 Actual image of the delivery service robots and its operating systems.

Once the overall concept is completed, you will have to decide what messages should be exchanged between the nodes in each area as in Figure 12-3.

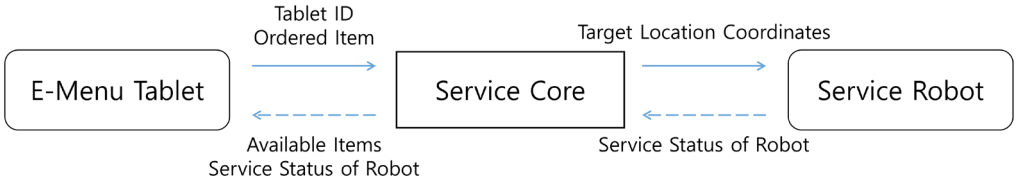

FIGURE 12-3 Example of the messages in each area being transmitted by the delivery service robot

Although one pad can supervise mutiple robots, to simplify the system structure, one pad per robot was paired to carry out the service. The pad used to place an order must send the information of the pad ID and selected items to the service core. Based on the pad ID, the service core determines which service robot to designate the work and transmits the corresponding target site for the ordered item to perform the service.

Based on path planning algorithm, the service robot moves to the received target site. While carrying out this mission, the robot transmits the service status of the robot including the collision with the obstacle, the route failure, the arrival of the target, and whether or not the ordered item is acquired. The service core processes information such as status on duplicate orders related to the ordered items, orders received from the robot while the robot is in service to present to the customer through the pad. Every information sent and received during this process uses customized new msg files or srv files.

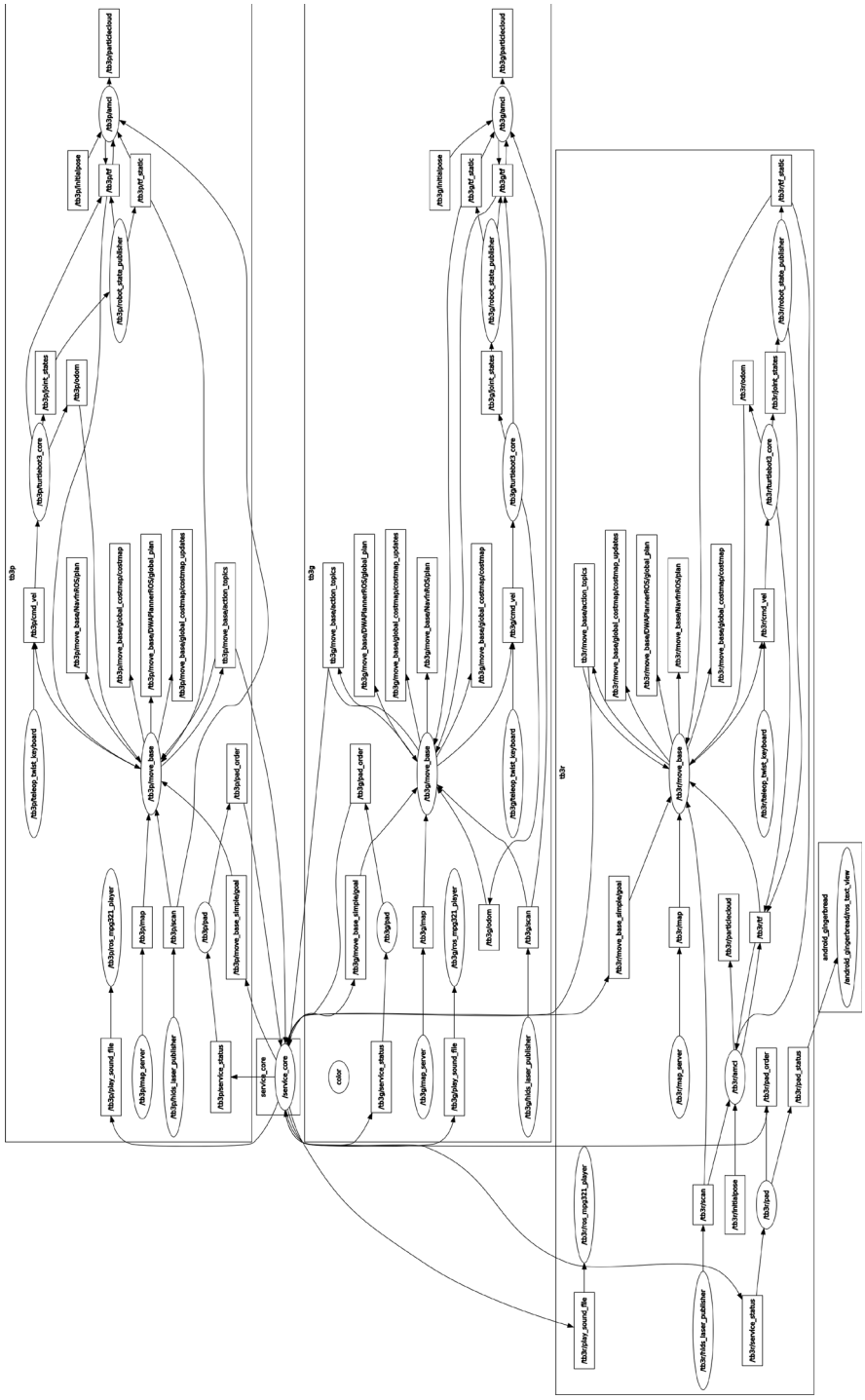

FIGURE 12-4 Example of a delivery service robot's node structure

Before getting into details about each node of the service robot, let's understand the general concept first. Figure 12-4 shows the correlation of all the nodes and topics of the delivery service robot to build in Chapter 12. The graph consists of 3 groups and the 'service_core'. The service robot related nodes are paired up with 'tb3p', 'tb3g', 'tb3r' namespaces in each group. The 'service_core' must supervise three groups and therefore should not be dependent on any group. If you want to configure a system that allows one pad to control mutiple robots instead of pairing each robot to a single pad, you need to create separate groups for each pad and each service robot under a separate namespace.

The reason why group namespace[1] is used when developing a service robot's system is because when a majority of robot or computers want to use one type of ROS package at the same time, it is impossible to execute duplicated names of nodes and topics registered in ROS master.

Among the topics grouped into three namespaces, Figure 12-5 shows a node graph that focuses on the topics that 'service_core' directly sends and receives. The 'service_core' receives an order through a topic called '/pad_order' and receives the robot's path finding and destination success through the '/move_base/action_topics' topic. It also delivers the service status via the '/service_status' topic, passes the file location of the recorded voice file via '/play_sound_file', and coordinates the robot's target point via '/move_base_simple/goal'.

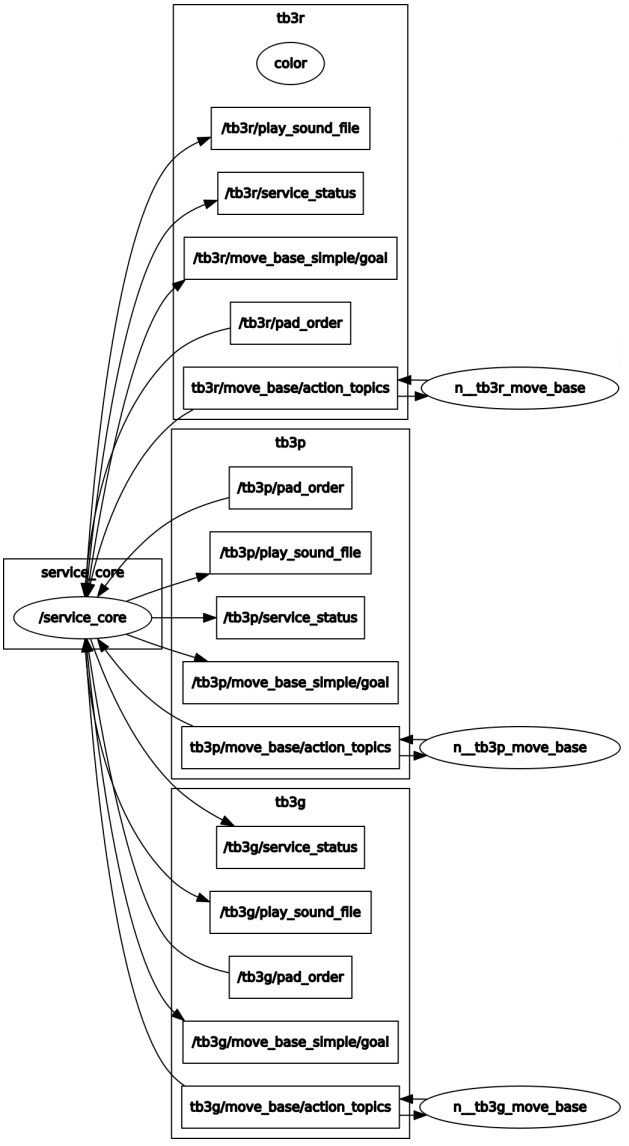

FIGURE 12-5 List of topics that service_core node sends and receives

1 http://wiki.ros.org/roslaunch/XML/group

Figure 12-6 is a flow chart for preparing the delivery service. Navigation uses the map made with SLAM. When serving, think about which location to use as a service location. The position should be identified and recorded as the coordinate values on the map created, and the coordinate values will be used in ROS parameter acquisition and configuration in 'service_core' to be described later. For my example, I need coordinate values of the location for the customer to place an order and location for the robot to get an ordered item. Refer to chapter 10 and 11 or details on SLAM and navigation.

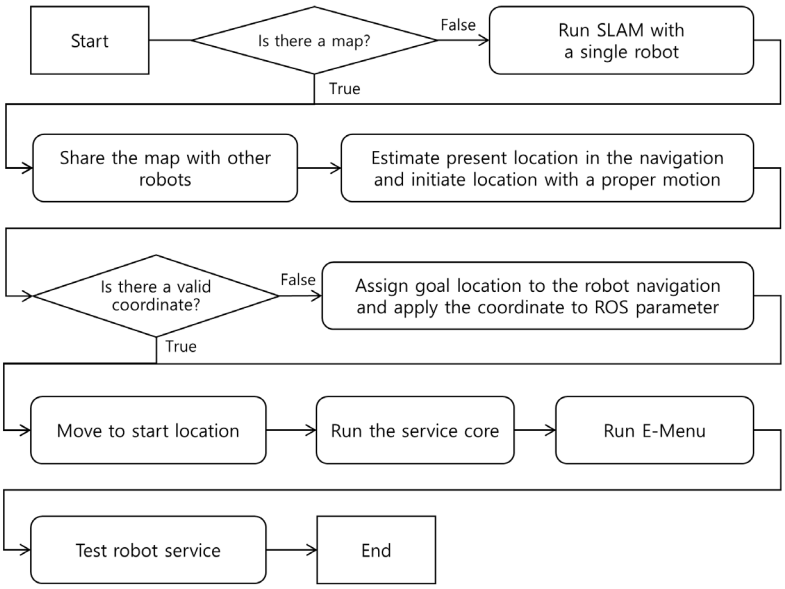

Figure 12-6 Flowchart for Preparing Delivery Services

The source code necessary to build a delivery service robot described here, are available in the address below. Let's move on to the configuration and sources of the nodes.

- https://github.com/ROBOTIS-GIT/turtlebot3_deliver

12.2.3. Service Core Node

Figure 12-7 shows the basic structure of the service core's source. This node starts with the main() function and first sets the ROS parameter via fnInitParam(). Afterwards, orders received from customers through the pad and cbReceivePadOrder() and cbCheckArrivalStatus() functions that receive data about the robot's target position arrival status are declared to execute as topic subscribe and through functions fnPubServiceStatus(), fnPubPose(), the service state and the target position of the robot coordinate values are published. This process is executed in an endless loop until the [Ctrl + c] key is pressed

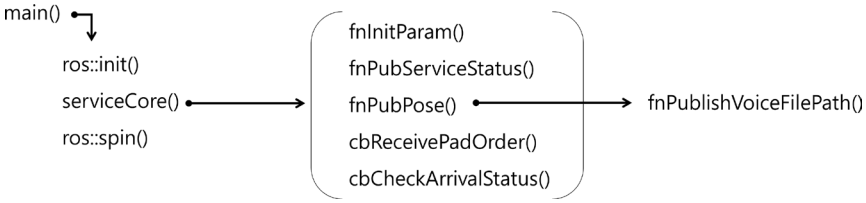

FIGURE 12-7 Basic structure of service core

When function serviceCore() is executed, function fnInitParam() is called and a publisher and subscriber are defined for various data transmission and reception. The 'service_core' must handle topics for the three robots, three pads. Therefore, three types of publisher and subscriber are defined. Each of their work is described as below.

Item	Description
pubServiceStatusPad	Publisher that publishes delivery service status to pad
pubPlaySound	Publisher that publishes the location of recorded voice files
subPadOrder	Subscribers that subscribe to customer orders from the pads
subArrivalStatus	Subscriber that subscribes to the arrival of a robot

TABLE 12-1 ServiceCore() function in service_core.cpp

/turtlebot3_carrier/src/service_core.cpp (parts of excerpt)

```
ServiceCore()
{
  fnInitParam();

  pubServiceStatusPadTb3p = nh_.advertise<turtlebot3_carrier::ServiceStatus>("/tb3p/
service_status", 1);
  pubServiceStatusPadTb3g = nh_.advertise<turtlebot3_carrier::ServiceStatus>("/tb3g/
service_status", 1);
  pubServiceStatusPadTb3r = nh_.advertise<turtlebot3_carrier::ServiceStatus>("/tb3r/
service_status", 1);

  pubPlaySoundTb3p = nh_.advertise<std_msgs::String>("/tb3p/play_sound_file", 1);
  pubPlaySoundTb3g = nh_.advertise<std_msgs::String>("/tb3g/play_sound_file", 1);
  pubPlaySoundTb3r = nh_.advertise<std_msgs::String>("/tb3r/play_sound_file", 1);

  pubPoseStampedTb3p = nh_.advertise<geometry_msgs::PoseStamped>("/tb3p/move_base_simple/goal", 1);
  pubPoseStampedTb3g = nh_.advertise<geometry_msgs::PoseStamped>("/tb3g/move_base_simple/goal", 1);
```

```
  pubPoseStampedTb3r = nh_.advertise<geometry_msgs::PoseStamped>("/tb3r/move_base_simple/goal", 1);

  subPadOrderTb3p = nh_.subscribe("/tb3p/pad_order", 1, &ServiceCore::cbReceivePadOrder, this);
  subPadOrderTb3g = nh_.subscribe("/tb3g/pad_order", 1, &ServiceCore::cbReceivePadOrder, this);
  subPadOrderTb3r = nh_.subscribe("/tb3r/pad_order", 1, &ServiceCore::cbReceivePadOrder, this);

  subArrivalStatusTb3p = nh_.subscribe("/tb3p/move_base/result", 1,
&ServiceCore::cbCheckArrivalStatusTB3P, this);
  subArrivalStatusTb3g = nh_.subscribe("/tb3g/move_base/result", 1,
&ServiceCore::cbCheckArrivalStatusTB3G, this);
  subArrivalStatusTb3r = nh_.subscribe("/tb3r/move_base/result", 1,
&ServiceCore::cbCheckArrivalStatusTB3R, this);

  ros::Rate loop_rate(5);

  while (ros::ok())
  {
    fnPubServiceStatus();

    fnPubPose();
    ros::spinOnce();
    loop_rate.sleep();
  }
}
```

The fnInitParam() function obtains the target pose (position + orientation) data of the robot on the map from the given parameter file. In this example, the robot should be able to move to three positions where customers can place an order and also be able to move to the three locations where ordered items can be acquired, therefore, it is necessary to store the information of the six coordinates of the locations on the map. The structure of the fnInitParam() function is as follows.

Item	Description
poseStampedTable	Coordinates where customer places and receives orders
poseStampedCounter	Coordinates of where the item is loaded on the robot

TABLE 12-2 fnInitParam() Function in service_core.cpp

/turtlebot3_carrier/src/service_core.cpp (parts of excerpts)

```
void fnInitParam()
{
  nh_.getParam("table_pose_tb3p/position", target_pose_position);
  nh_.getParam("table_pose_tb3p/orientation", target_pose_orientation);

  poseStampedTable[0].header.frame_id = "map";
  poseStampedTable[0].header.stamp = ros::Time::now();

  poseStampedTable[0].pose.position.x = target_pose_position[0];
  poseStampedTable[0].pose.position.y = target_pose_position[1];
  poseStampedTable[0].pose.position.z = target_pose_position[2];

  poseStampedTable[0].pose.orientation.x = target_pose_orientation[0];
  poseStampedTable[0].pose.orientation.y = target_pose_orientation[1];
  poseStampedTable[0].pose.orientation.z = target_pose_orientation[2];
  poseStampedTable[0].pose.orientation.w = target_pose_orientation[3];


  nh_.getParam("table_pose_tb3g/position", target_pose_position);
  nh_.getParam("table_pose_tb3g/orientation", target_pose_orientation);

  poseStampedTable[1].header.frame_id = "map";
  poseStampedTable[1].header.stamp = ros::Time::now();

  poseStampedTable[1].pose.position.x = target_pose_position[0];
  poseStampedTable[1].pose.position.y = target_pose_position[1];
  poseStampedTable[1].pose.position.z = target_pose_position[2];

  poseStampedTable[1].pose.orientation.x = target_pose_orientation[0];
  poseStampedTable[1].pose.orientation.y = target_pose_orientation[1];
  poseStampedTable[1].pose.orientation.z = target_pose_orientation[2];
  poseStampedTable[1].pose.orientation.w = target_pose_orientation[3];


  nh_.getParam("table_pose_tb3r/position", target_pose_position);
  nh_.getParam("table_pose_tb3r/orientation", target_pose_orientation);

  poseStampedTable[2].header.frame_id = "map";
```

```
poseStampedTable[2].header.stamp = ros::Time::now();

poseStampedTable[2].pose.position.x = target_pose_position[0];
poseStampedTable[2].pose.position.y = target_pose_position[1];
poseStampedTable[2].pose.position.z = target_pose_position[2];

poseStampedTable[2].pose.orientation.x = target_pose_orientation[0];
poseStampedTable[2].pose.orientation.y = target_pose_orientation[1];
poseStampedTable[2].pose.orientation.z = target_pose_orientation[2];
poseStampedTable[2].pose.orientation.w = target_pose_orientation[3];


nh_.getParam("counter_pose_bread/position", target_pose_position);
nh_.getParam("counter_pose_bread/orientation", target_pose_orientation);

poseStampedCounter[0].header.frame_id = "map";
poseStampedCounter[0].header.stamp = ros::Time::now();

poseStampedCounter[0].pose.position.x = target_pose_position[0];
poseStampedCounter[0].pose.position.y = target_pose_position[1];
poseStampedCounter[0].pose.position.z = target_pose_position[2];

poseStampedCounter[0].pose.orientation.x = target_pose_orientation[0];
poseStampedCounter[0].pose.orientation.y = target_pose_orientation[1];
poseStampedCounter[0].pose.orientation.z = target_pose_orientation[2];
poseStampedCounter[0].pose.orientation.w = target_pose_orientation[3];


nh_.getParam("counter_pose_drink/position", target_pose_position);
nh_.getParam("counter_pose_drink/orientation", target_pose_orientation);

poseStampedCounter[1].header.frame_id = "map";
poseStampedCounter[1].header.stamp = ros::Time::now();

poseStampedCounter[1].pose.position.x = target_pose_position[0];
poseStampedCounter[1].pose.position.y = target_pose_position[1];
poseStampedCounter[1].pose.position.z = target_pose_position[2];

poseStampedCounter[1].pose.orientation.x = target_pose_orientation[0];
```

```
  poseStampedCounter[1].pose.orientation.y = target_pose_orientation[1];
  poseStampedCounter[1].pose.orientation.z = target_pose_orientation[2];
  poseStampedCounter[1].pose.orientation.w = target_pose_orientation[3];



  nh_.getParam("counter_pose_snack/position", target_pose_position);
  nh_.getParam("counter_pose_snack/orientation", target_pose_orientation);

  poseStampedCounter[2].header.frame_id = "map";
  poseStampedCounter[2].header.stamp = ros::Time::now();

  poseStampedCounter[2].pose.position.x = target_pose_position[0];
  poseStampedCounter[2].pose.position.y = target_pose_position[1];
  poseStampedCounter[2].pose.position.z = target_pose_position[2];

  poseStampedCounter[2].pose.orientation.x = target_pose_orientation[0];
  poseStampedCounter[2].pose.orientation.y = target_pose_orientation[1];
  poseStampedCounter[2].pose.orientation.z = target_pose_orientation[2];
  poseStampedCounter[2].pose.orientation.w = target_pose_orientation[3];
}
```

The values of the parameters indicated in the 'target_pose.yaml' file are coordinate values on the map that the robot needs to perform its services. There are various ways to obtain the coordinate values, and the simplest method is to use the 'rostopic echo' command to get the pose value during navigation. However, these coordinates change with each mapping through SLAM so avoid rewritting the map while performing the actual navigation to reduce the positional changes of objects and obstacles so that you can continue with the same map.

/turtlebot3_carrier/param/target_pose.yaml

```
table_pose_tb3p:
  position: [-0.338746577501, -0.85418510437, 0.0]
  orientation: [0.0, 0.0, -0.0663151963596, 0.997798724559]

table_pose_tb3g:
  position: [-0.168751597404, -0.19147400558, 0.0]
  orientation: [0.0, 0.0, -0.0466624033917, 0.998910716786]

table_pose_tb3r:
  position: [-0.251043587923, 0.421476781368, 0.0]
```

```
  orientation: [0.0, 0.0, -0.0600887022438, 0.998193041382]


counter_pose_bread:
  position: [-3.60783815384, -0.750428497791, 0.0]
  orientation: [0.0, 0.0, 0.999335763287, -0.0364421763375]

counter_pose_drink:
  position: [-3.48697376251, -0.173366710544, 0.0]
  orientation: [0.0, 0.0, 0.998398746904, -0.0565680314445]

counter_pose_snack:
  position: [-3.62247490883, 0.39046728611, 0.0]
  orientation: [0.0, 0.0, 0.998908838216, -0.0467026009308]
```

Next, based on which service procedure the current robot is in, the fnPubPose() function is used to set the next target position when the robot reaches the target position. When the robot completes the service, all parameters are reset.

Item	Description
is_robot_reached_target	Whether the robot has reached its navigation destination
is_item_available	Availability of items
item_num_chosen_by_pad	Item number of the order
robot_service_sequence	Process that the robot is performing 0 - Waiting for customer's order 1 - Immediately after receiving customer's order 2 - Going to get item ordered 3 - Loading ordered items 4 - Moving to customer's location 5 - Delivering items to customer
fnPublishVoiceFilePath()	Function to publish the path of the recorded voice file
ROBOT_NUMBER	Robot's number(User defined)

TABLE 12-3 fnPubPose() Functions in service_core.cpp

/turtlebot3_carrier/src/service_core.cpp (parts of excerpt)

```
void fnPubPose()
{
  if (is_robot_reached_target[ROBOT_NUMBER])
  {
    if (robot_service_sequence[ROBOT_NUMBER] == 1)
    {
      fnPublishVoiceFilePath(ROBOT_NUMBER, "~/voice/voice1-2.mp3");

      robot_service_sequence[ROBOT_NUMBER] = 2;
    }
    else if (robot_service_sequence[ROBOT_NUMBER] == 2)
    {
      pubPoseStampedTb3p.publish(poseStampedCounter[item_num_chosen_by_pad[ROBOT_NUMBER]]);

      is_robot_reached_target[ROBOT_NUMBER] = false;

      robot_service_sequence[ROBOT_NUMBER] = 3;
    }
    else if (robot_service_sequence[ROBOT_NUMBER] == 3)
    {
      fnPublishVoiceFilePath(ROBOT_NUMBER, "~/voice/voice1-3.mp3");

      robot_service_sequence[ROBOT_NUMBER] = 4;
    }
    else if (robot_service_sequence[ROBOT_NUMBER] == 4)
    {
      pubPoseStampedTb3p.publish(poseStampedTable[ROBOT_NUMBER]);

      is_robot_reached_target[ROBOT_NUMBER] = false;

      robot_service_sequence[ROBOT_NUMBER] = 5;
    }
    else if (robot_service_sequence[ROBOT_NUMBER] == 5)
    {
      fnPublishVoiceFilePath(ROBOT_NUMBER, "~/voice/voice1-4.mp3");

      robot_service_sequence[ROBOT_NUMBER] = 0;
```

```
      is_item_available[item_num_chosen_by_pad[ROBOT_NUMBER]] = 1;

      item_num_chosen_by_pad[ROBOT_NUMBER] = -1;
    }
  }
}
··· omitted ···
```

The cbReceivePadOrder() function receives the number of the pad used at the time of ordering and the item number of the ordered item to determine whether the service is available. If the service is available, the 'robot_service_sequence' is set to '1' to start the service. The source code of this function is as follows.

Item	Description
pad_number	Number of the pad ordered from (Number of robot to service)
item_number	Item number of the order

TABLE 12-4 cbReceivePadOrder() Function in service_core.cpp

/turtlebot3_carrier/src/service_core.cpp (parts of excerpt)

```
void cbReceivePadOrder(const turtlebot3_carrier::PadOrder padOrder)
{
  int pad_number = padOrder.pad_number;
  int item_number = padOrder.item_number;

  if (is_item_available[item_number] != 1)
  {
    ROS_INFO("Chosen item is currently unavailable");
    return;
  }

  if (robot_service_sequence[pad_number] != 0)
  {
    ROS_INFO("Your TurtleBot is currently on servicing");
    return;
  }

  if (item_num_chosen_by_pad[pad_number] != -1)
  {
```

```
    ROS_INFO("Your TurtleBot is currently on servicing");
    return;
  }

  item_num_chosen_by_pad[pad_number] = item_number;

  robot_service_sequence[pad_number] = 1; // just left from the table

  is_item_available[item_number] = 0;
}
```

The cbCheckArrivalStatus() function shown below checks the moving state of the subscribing robot. The 'else' block of the code handles if the robot comes across an obstacle during its navigation or is unable to find its path in the algorithm.

/turtlebot3_carrier/src/service_core.cpp (parts of excerpt)

```
void cbCheckArrivalStatusTB3P(const move_base_msgs::MoveBaseActionResult rcvMoveBaseActionResult)
{
  if (rcvMoveBaseActionResult.status.status == 3)
  {
    is_robot_reached_target[ROBOT_NUMBER_TB3P] = true;
  }
  else
  {
    …omitted…
  }
}

void cbCheckArrivalStatusTB3G(const move_base_msgs::MoveBaseActionResult rcvMoveBaseActionResult)
{
…omitted…
}

void cbCheckArrivalStatusTB3R(const move_base_msgs::MoveBaseActionResult rcvMoveBaseActionResult)
{
…omitted…
}
```

The fnPublishVoicePath() function shown in the following source code publishes the location of the pre-recorded voice file as a String. In order to actually play the voice on ROS, you will need additional ROS Package.

/turtlebot3_carrier/src/service_core.cpp (parts of excerpt)

```
void fnPublishVoiceFilePath(int robot_num, const char* file_path)
  {
    std_msgs::String str;

    str.data = file_path;

    if (robot_num == ROBOT_NUMBER_TB3P)
    {
      pubPlaySoundTb3p.publish(str);
    }
    else if (robot_num == ROBOT_NUMBER_TB3G)
    {
      pubPlaySoundTb3g.publish(str);
    }
    else if (robot_num == ROBOT_NUMBER_TB3R)
    {
      pubPlaySoundTb3r.publish(str);
    }
  }
```

12.2.4. Service Master Node

In this example, the service master node runs on a tablet PC with the Android OS. However, nodes can also be programmed to accepted orders through the terminal. Details on ROS Java[2] programming using Android OS platform will be covered in the next chapter. The source of the service master node described below is created by applying 'android_tutorial_pubsub', a simple topic publishing and subscribing example source that is provided by ROS Java.

turtlebot3_carrier_pad/ServicePad.java

```
package org.ros.android.android_tutorial_pubsub;

import org.ros.concurrent.CancellableLoop;
```

2 http://wiki.ros.org/rosjava

```
import org.ros.message.MessageListener;
import org.ros.namespace.GraphName;
import org.ros.node.AbstractNodeMain;
import org.ros.node.ConnectedNode;
import org.ros.node.topic.Publisher;
import org.ros.node.topic.Subscriber;

import javax.security.auth.SubjectDomainCombiner;

public class ServicePad extends AbstractNodeMain {
  private String pub_pad_order_topic_name;
  private String sub_service_status_topic_name;
  private String pub_pad_status_topic_name;

  private int robot_num = 0;
  private int selected_item_num = -1;

  private boolean jump = false;
  private int[] item_num_chosen_by_pad = {-1, -1, -1};
  private int[] is_item_available = {1, 1, 1};
  private int[] robot_service_sequence = {0, 0, 0};

  public boolean[] button_pressed = {false, false, false};

  public ServicePad() {
    this.pub_pad_order_topic_name = "/tb3g/pad_order";
    this.sub_service_status_topic_name = "/tb3g/service_status";
    this.pub_pad_status_topic_name = "/tb3g/pad_status";
  }

  public GraphName getDefaultNodeName() {
    return GraphName.of("tb3g/pad");
  }

  public void onStart(ConnectedNode connectedNode) {
    final Publisher pub_pad_order = connectedNode.newPublisher(this.pub_pad_order_topic_name,
"turtlebot3_carrier/PadOrder");
    final Publisher pub_pad_status = connectedNode.newPublisher(this. pub_pad_status_topic_name,
"std_msgs/String");
```

```
    final Subscriber<turtlebot3_carrier.ServiceStatus> subscriber =
connectedNode.newSubscriber(this.sub_service_status_topic_name, " turtlebot3_carrier/
ServiceStatus");

    subscriber.addMessageListener(new MessageListener<turtlebot3_carrier.SerivceStatus>() {
      @Override
      public void onNewMessage(turtlebot3_carrier.SerivceStatus serviceStatus)
      {
        item_num_chosen_by_pad = serviceStatus.item_num_chosen_by_pad;
        is_item_available = serviceStatus.is_item_available;
        robot_service_sequence = serviceStatus.robot_service_sequence
      }
    });

    connectedNode.executeCancellableLoop(new CancellableLoop() {
      protected void setup() {}

      protected void loop() throws InterruptedException
      {
        str_msgs.String padStatus = (str_msgs.String)pub_pad_status.newMessage();
        turtlebot3_carrier.PadOrder padOrder =
(turtlebot3_carrier.PadOrder)pub_pad_order.newMessage();

        String str = "";

        if (button_pressed[0] || button_pressed[1] || button_pressed[2])
        {
          jump = false;

          if (button_pressed[0])
          {
            selected_item_num = 0;
            str += "Burger was selected";

            button_pressed[0] = false;
          }
          else if (button_pressed[1])
          {
            selected_item_num = 1;
```

```
    str += "Coffee was selected";

    button_pressed[1] = false;
  }
  else if (button_pressed[2])
  {
    selected_item_num = 2;
    str += "Waffle was selected";

    button_pressed[2] = false;
  }
  else
  {
    selected_item_num = -1;
    str += "Sorry, selected item is now unavailable. Please choose another item.";
  }

  if (is_item_available[selected_item_num] != 1)
  {
    str += ", but chosen item is currently unavailable.";
    jump = true;
  }
  else if (robot_service_sequence[robot_num] != 0)
  {
    str += ", but your TurtleBot is currently on servicing";
    jump = true;
  }
  else if (item_num_chosen_by_pad[robot_num] != -1)
  {
    str += ", but your TurtleBot is currently on servicing";
    jump = true;
  }

  padStatus.setData(str);
  pub_pad_status.publish(padStatus);

  if(!jump)
  {
    padOrder.pad_number = robot_num;
```

```
            padOrder.item_number = selected_item_num;
            pub_pad_order.publish(padOrder);
          }
        }

        Thread.sleep(1000L);
      }
    });
  }
}
```

In the code, MainActivity class receives and uses an instance of ServicePad class. Other than that, it is pretty much the same as general MainActivity class so we will omit detailed explanation and only look into the part corresponding to delivery service.

The number of the robot to be controlled by the tablet PC to which this example is uploaded is designated as robot_num as the following source code, and the number of the selected product on the tablet PC is initialized to '-1'.

```
private int robot_num = 0;
private int selected_item_num = -1;
```

In order to avoid duplicate orders, the tablet PC must be synchronized with the order status stored in the service core before receiving the customer's order. The following source code is declared in the service core in the same format as the array that records the order status.

```
private int[] item_num_chosen_by_pad = {-1, -1, -1};
private int[] is_item_available = {1, 1, 1};
private int[] robot_service_sequence = {0, 0, 0};
```

When you create an instance of the ServicePad class in the MainActivity class, you will specify the topic name as follows. In this example, each pad and robot pair is specified as a group namespace, so it is necessary to write the name used in the namespace in front of each topic name.

```
public ServicePad() {
  this.pub_pad_order_topic_name = "/tb3g/pad_order";
  this.sub_service_status_topic_name = "/tb3g/service_status";
  this.pub_pad_status_topic_name = "/tb3g/pad_status";
}
```

Specifies the node name that appears on the ROS. The node name should also be written with the group namespace.

```
public GraphName getDefaultNodeName() {
  return GraphName.of("tb3g/pad");
}
```

From the service core, the user receives service conditions such as the item number of ordered item, the availability of item selection, and the service status of the robot.

```
subscriber.addMessageListener(new MessageListener<turtlebot3_carrier.ServiceStatus>() {
  @Override
  public void onNewMessage(turtlebot3_carrier.ServiceStatus serviceStatus)
  {
    item_num_chosen_by_pad = serviceStatus.item_num_chosen_by_pad;
    is_item_available = serviceStatus.is_item_available;
    robot_service_sequence = serviceStatus.robot_service_sequence;
  }
});
```

This portion processes the customer's orders based on the overall service situation synchronized with the service core. Likewise, checks whether or not the order is duplicated, and if the service is available, publishes the order. Figures 12-8 and 12-9 show examples of successful and unsuccessful orders, respectively, when a customer selects an item drawn on the pad.

```
protected void loop() throws InterruptedException
{
  std_msgs.String padStatus = (std_msgs.String) pub_pad_status.newMessage();
  turtlebot3_carrier.PadOrder padOrder = (turtlebot3_carrier.PadOrder)
  pub_pad_order.newMessage();

  String str = "";

  if (button_pressed[0] || button_pressed[1] || button_pressed[2])
  {
    jump = false;

    if (button_pressed[0])
    {
```

```
    selected_item_num = 0;
    str += "Burger was selected";

    button_pressed[0] = false;
  }
  else if (button_pressed[1])
  {
    selected_item_num = 1;
    str += "Coffee was selected";

    button_pressed[1] = false;
  }
  else if (button_pressed[2])
  {
    selected_item_num = 2;
    str += "Waffle was selected";

    button_pressed[2] = false;
  }
  else
  {
    selected_item_num = -1;
    str += "Sorry, selected item is now unavailable. Please choose another item.";
  }

  if (is_item_available[selected_item_num] != 1)
  {
    str += ", but chosen item is currently unavailable.";
    jump = true;
  }
  else if (robot_service_sequence[robot_num] != 0)
  {
    str += ", but your TurtleBot is currently on servicing";
    jump = true;
  }
  else if (item_num_chosen_by_pad[robot_num] != -1)
  {
    str += ", but your TurtleBot is currently on servicing";
    jump = true;
  }
```

```
    padStatus.setData(str);
    pub_pad_status.publish(padStatus);

    if(!jump)
    {
      padOrder.pad_number = robot_num;
      padOrder.item_number = selected_item_num;
      pub_pad_order.publish(padOrder);
    }
  }

  Thread.sleep(1000L);
}
```

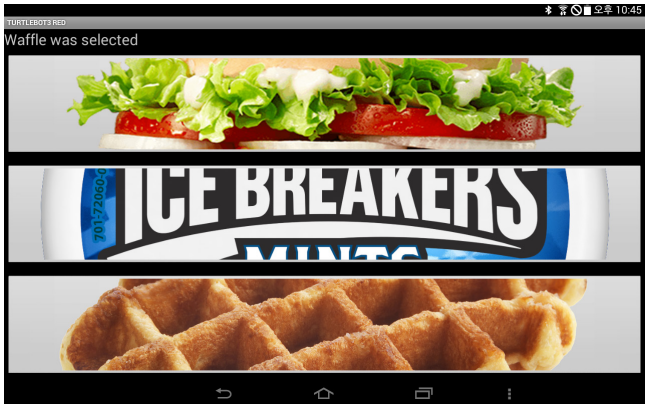

FIGURE 12-8 Example 1 of menu displayed on a pad (When the order is successfully received)

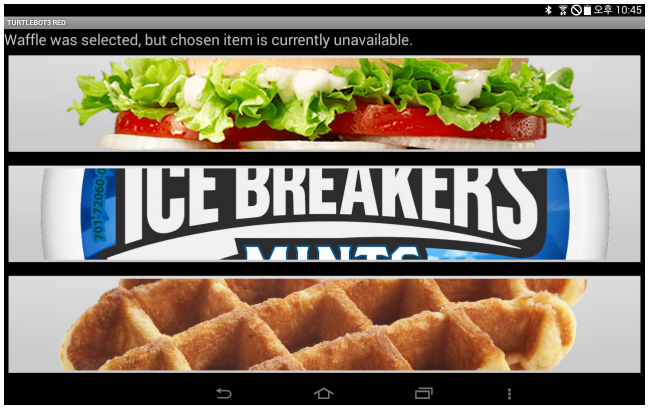

FIGURE 12-9 Example 2 of menu displayed on a pad (When unavailable item is selected)

12.2.5. Service Slave Node

The nodes supervised in the service slave node are directly related to the control of the robot. This example uses the TurtleBot3 Carrier[3], and the nodes that are executed when performing SLAM and navigation as described in Chapters 10 and 11 are the main nodes. Here we describe TurtleBot3 source code that have been modified to develop the TurtleBot3 Carrier. In this example, the packages shown in Figure 12-10 are mainly used. Each arrow represents a subpackage. Some of these packages need to be modified to make a delivery service robot.

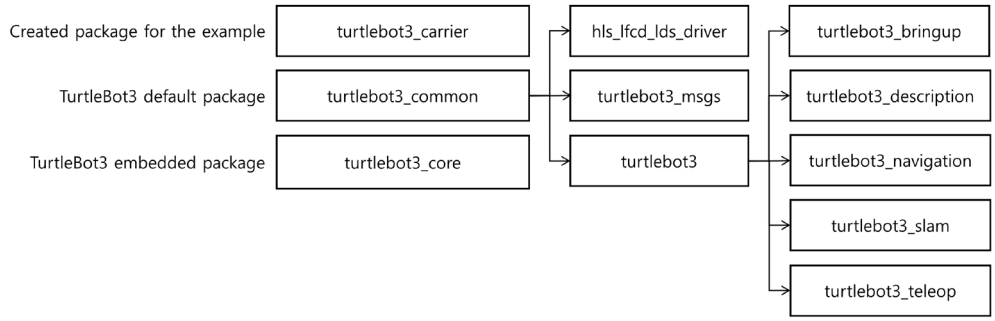

FIGURE 12-10 List of packages used

The following describes the modified source for creating a delivery service robot. The poll() function processes the distance measurement values of nearby objects using the LDS(HLS-LFCD2) laser sensor. The TurtleBot3 Carrier has pillars around the LDS and it is stacked-up for delivery services. Since it recognizes these pillars when the LDS is operated, it will affect the results of the SLAM or navigation. Therefore, when an object is detected at a distance less than a certain value, TurtleBot3 Carrier will set the detected value as '0'. Later, in the navigation algorithm, the value of distance '0' is recognized as 'no object', and the existence of the pillar does not affect SLAM or navigation.

hld_lfcd_lds_driver/src/hlds_laser_publisher.cpp

```
void LFCDLaser::poll(sensor_msgs::LaserScan::Ptr scan)
{
…omitted…

  while (!shutting_down_ && !got_scan)
  {
… omitted …
```

3 http://emanual.robotis.com/docs/en/platform/turtlebot3/friends/#turtlebot3-friends-carrier

```
    if(start_count == 0)
    {
      if(raw_bytes[start_count] == 0xFA)
      {
        start_count = 1;
      }
    }
    else if(start_count == 1)
    {
      if(raw_bytes[start_count] == 0xA0)
      {
… omitted …

        //read data in sets of 6
        for(uint16_t i = 0; i < raw_bytes.size(); i=i+42)
        {
          if(raw_bytes[i] == 0xFA && raw_bytes[i+1] == (0xA0 + i / 42))
          {
… omitted …

            for(uint16_t j = i+4; j < i+40; j=j+6)
            {
              index = (6*i)/42 + (j-6-i)/6;

              // Four bytes per reading
              uint8_t byte0 = raw_bytes[j];
              uint8_t byte1 = raw_bytes[j+1];
              uint8_t byte2 = raw_bytes[j+2];
              uint8_t byte3 = raw_bytes[j+3];

              // Remaining bits are the range in mm
              uint16_t intensity = (byte1 << 8) + byte0;
              uint16_t range = (byte3 << 8) + byte2;

              scan->ranges[359-index] = range / 1000.0;
              scan->intensities[359-index] = intensity;
            }
          }
        }
```

```
      /// Add pillars ///
      for(uint16_t deg = 0; deg < 360; deg++)
      {
        if(scan->ranges[deg] < 0.15)
        {
          scan->ranges[deg] = 0.0;
          scan->intensities[deg] = 0.0;
        }
      }
      /// end of addition ///

        scan->time_increment = motor_speed/good_sets/1e8;
      }
      else
      {
        start_count = 0;
      }
    }
  }
}
```

The 'turtlebot3_core' is a firmware installed on the control board OpenCR used by TurtleBot3, and 'turtlebot3_motor_driver.cpp' is a source that directly controls the actuator used in TurtleBot3. The actual service robot moves with objects loaded, so proper control is required for safe movement. Therefore, we added the profile control of Dynamixels which is not included in the original source of 'turtlebot3_motor_driver.cpp'. Here, the value of ADDR_X_PROFILE_ACCELERATION is 108. For more information about the actuator, please refer to the Dynamixel manual (http://emanual.robotis.com/).

turtlebot3_core(for TurtleBot3 Waffle)/turtlebot3_motor_driver.cpp

```
bool Turtlebot3MotorDriver::init(void)
{
… omitted …
  // Enable Dynamixel Torque
  setTorque(left_wheel_id_, true);
  setTorque(right_wheel_id_, true);

  /// begin addition ///
```

```
  // Set Dynamixel Profile Acceleration
  setProfileAcceleration(left_wheel_id_, 15);
  setProfileAcceleration(right_wheel_id_, 15);

  /// end of addition ///

  … omitted …

  return true;
}

bool Turtlebot3MotorDriver::setTorque(uint8_t id, bool onoff)
{
… omitted …

}

bool Turtlebot3MotorDriver::setProfileAcceleration(uint8_t id, uint32_t value)
{
  uint8_t dxl_error = 0;
  int dxl_comm_result = COMM_TX_FAIL;

  dxl_comm_result = packetHandler_->write4ByteTxRx(portHandler_, id, ADDR_X_PROFILE_ACCELERATION,
value, &dxl_error);
  if(dxl_comm_result != COMM_SUCCESS)
  {
    packetHandler_->printTxRxResult(dxl_comm_result);
  }
  else if(dxl_error != 0)
  {
    packetHandler_->printRxPacketError(dxl_error);
  }
}
```

The file 'turtlebot3_navigation.launch' launches nodes that are executed when using navigation in TurtleBot3. As described previously, nodes and ROS topics must be grouped into a group namespace in order to use the identical ROS package in many robots at the same time. In the launch source code above, we grouped the nodes with 'tb3g' namespace and used the remap function to receive messages from other nodes that are not grouped in the same namespace.

This method is also used when the topic name can not be modified in the source code. Please note that this modification should be done not only in the launch file, but also in all areas where grouping such as the RViz file is required.

turtlebot3/turtlebot3_navigation/turtlebot3_navigation.launch

```
<launch>

<!—- addition starts -->


  <group ns="tb3g">
    <remap from="/tf" to="/tb3g/tf"/>
    <remap from="/tf_static" to="/tb3g/tf_static"/>

<!-- addition ends -->

    <arg name="model" default="waffle" doc="model type [burger, waffle]"/>

    … omitted …

    </node>

<!—- addition starts -->


  </group>

<!-- addition ends -->

</launch>
```

If everything has gone smoothly up to this point, building the system shown in Figure 12-11 will be successful.

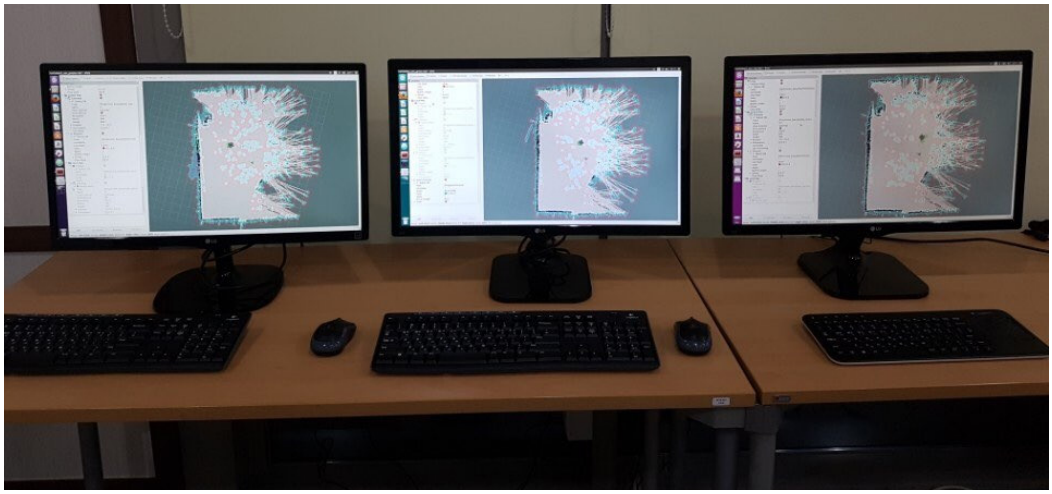

FIGURE 12-11 Navigation view of each robot displayed on RViz

12.3. Android Tablet PC Programming with ROS Java

In the previous section, we have described about placing orders using a pad, for example, the menu that operates on the pad in Figure 12-8. In this section, we will install Android Studio IDE[4] on Linux and build a ROS Java development environment. Then, explain a simple ROS Java example.

The following describes how to install the necessary packages for installing Android Studio IDE and applying ROS Java environment. ROS Java refers to the ROS client library running in the Java language. Let's first set the necessary environment for using Java. The necessary packages are the Java SE Development Kit (JDK), and you need to specify the location where you want to run it. This book describes how to download JDK 8, but you should modify it when the JDK version is updated.

```
$ sudo apt-get install openjdk-8-jdk
$ echo export PATH=${PATH}:/opt/android-sdk/tools:/opt/android-sdk/platform-tools:/opt/android-
studio/bin >> ~/.bashrc
$ echo export ANDROIS_HOME=/opt/android-sdk >> ~/.bashrc
$ source ~/.bashrc
```

4 https://developer.android.com/studio/index.html

The following command downloads the necessary tools for building in ROS Java. After that, install and build a package that contains the ROS Java system and examples. The 'android_core' folder is the workspace folder as 'catkin_ws' on ROS.

```
$ sudo apt-get install ros-kinetic-rosjava-build-tools
```

```
$ mkdir -p ~/android_core
$ wstool init -j4 ~/android_core/src https://raw.github.com/rosjava/rosjava/kinetic/
android_core.rosinstall
$ source /opt/ros/kinetic/setup.bash
$ cd ~/android_core
$ catkin_make
```

Here's how to install Android Studio IDE. To avoid confusion, I will use identical names and locations of the folders and files described in the ROS Android. These packages are add-on packages for installing mksdcard that allows you to use the SD card in your Android Studio IDE to implement the Virtual Device. If you do not install these packages, the mksdcard function fails to install.

```
$ sudo apt-get install lib32z1 lib32ncurses5 lib32stdc++6
```

This book installs the Android Studio IDE and SDK in '/opt' folder, which is the installation folder recommended by ROS Android[5]. To install to the '/opt' folder, you need to change the user permissions of '/opt' to be writable.

```
$ sudo chown -R $USER:$USER /opt
```

The Android Studio IDE installation file can be found here. https://developer.android.com/studio/index.html#download

When the installation file is downloaded, extract it to '/opt/android-studio'. After decompression, the folder is configured as shown in Figure 12-12.

5 http://wiki.ros.org/android

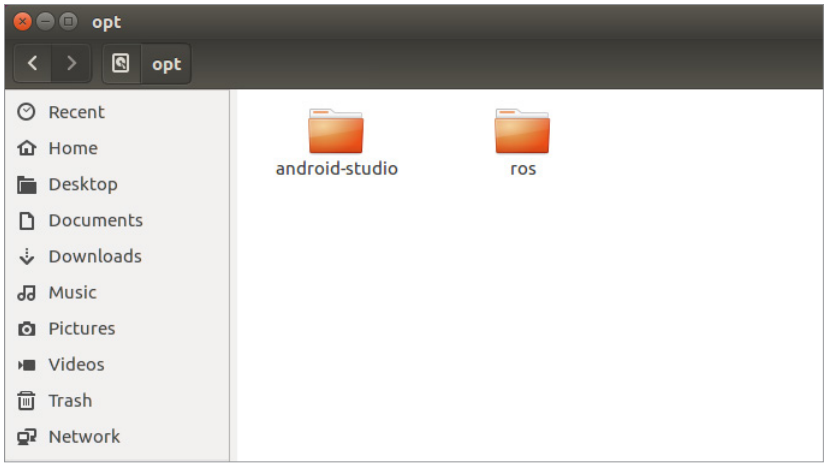

FIGURE 12-12 Decompressed Android Studio IDE

When the extraction is completed, enter the following command to start the IDE installation.

```
$ /opt/android-studio/bin/studio.sh
```

When the window shown in Figure 12-13 appears, click 'Run without import' to proceed to 'custom install'.

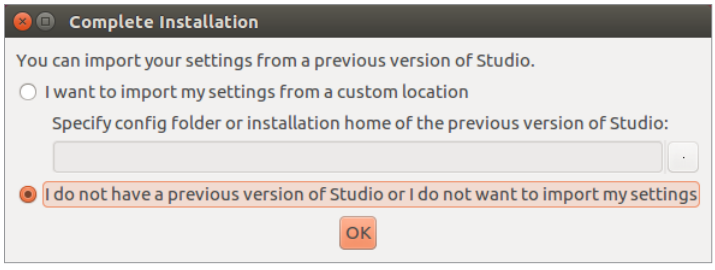

FIGURE 12-13 First window that appears during installation

After that, you should see the window for installing Android SDK like Figure 12-14. Please note that you need to set the installation location to '/opt/android-sdk'. If you do not have the 'android-sdk' folder, create it in the location and proceed.

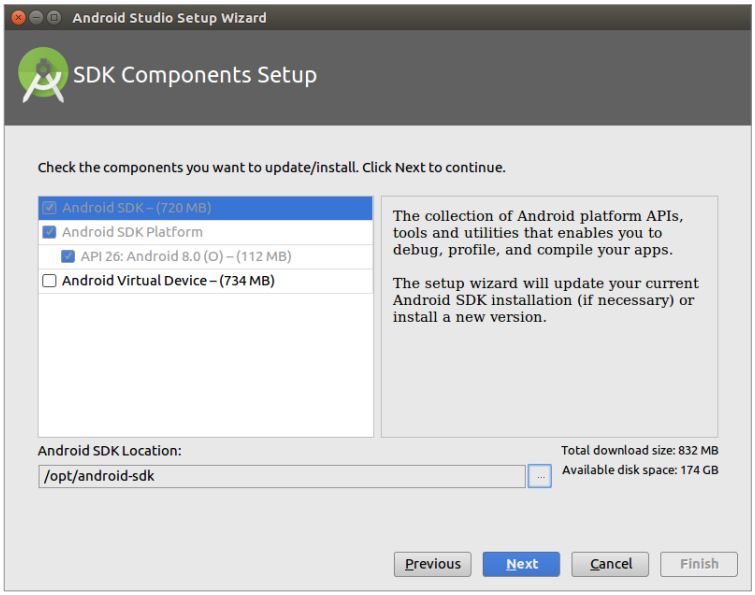

FIGURE 12-14 Android SDK installation screen

If the installation is completed, it ends as shown in Figure 12-15.

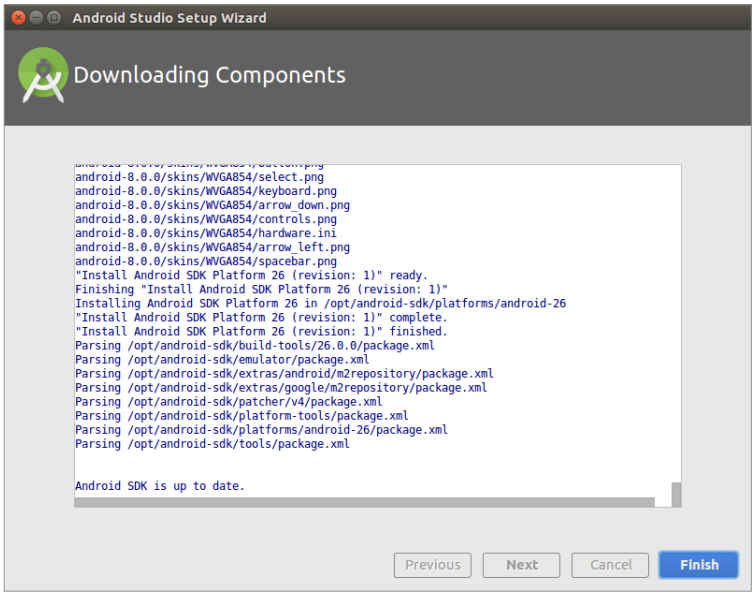

FIGURE 12-15 Installation completed screen

When the Android Studio IDE is successfully installed, a 'Welcome to Android Studio' window will appear as shown in Figure 12-16.

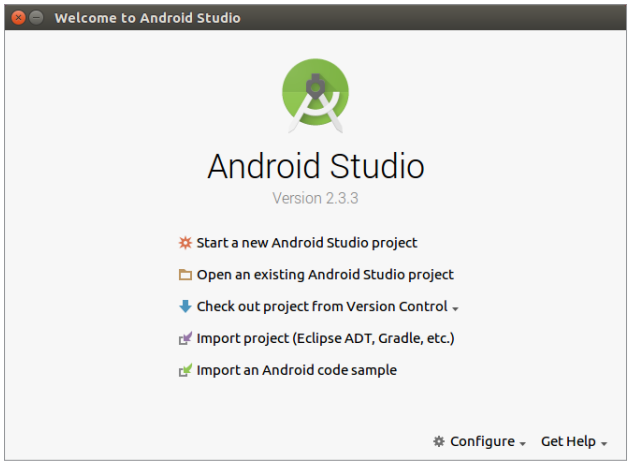

FIGURE 12-16 Welcome to Android Studio

Now, proceed to update the Android SDK through Configure → SDK Manager.

In Figure 12-17, the available SDKs to be updated are 10 (Gingerbread), 13 (Honeycomb), 15 (Ice cream) and 18 (Jellybean) and each number represents the API level of SDK.

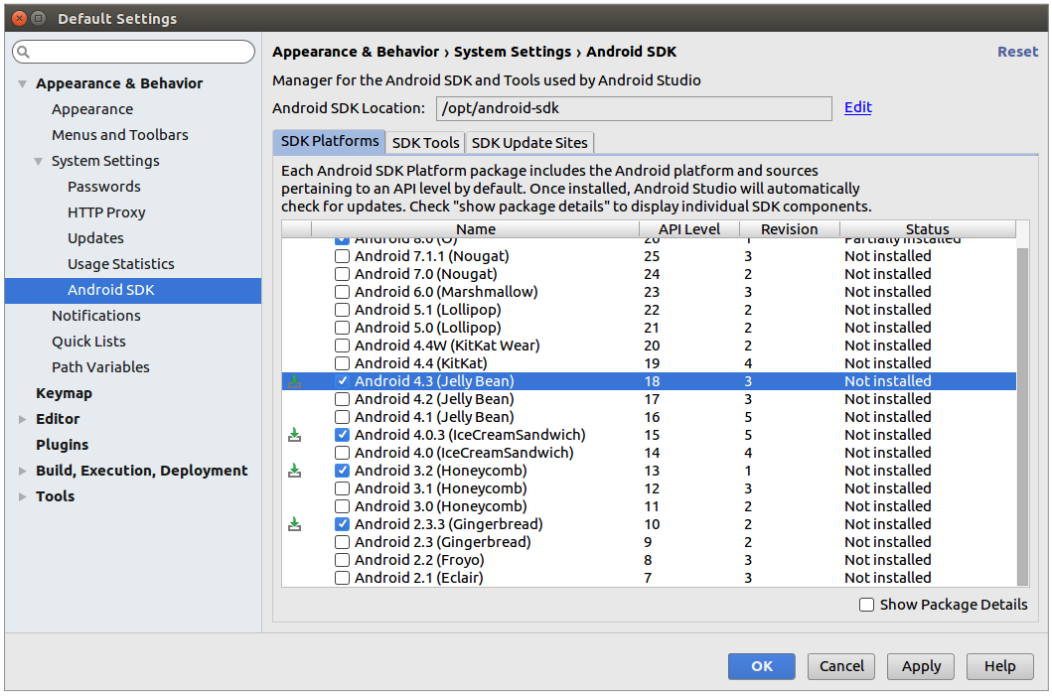

FIGURE 12-17 Setting up the Android SDK

After setting is completed, click on an existing Android Studio project in the window and import the previously installed 'android_core' as shown in Figure 12-18. When you click OK, the IDE window shown in Figure 12-19 appears and begins to build the imported source.

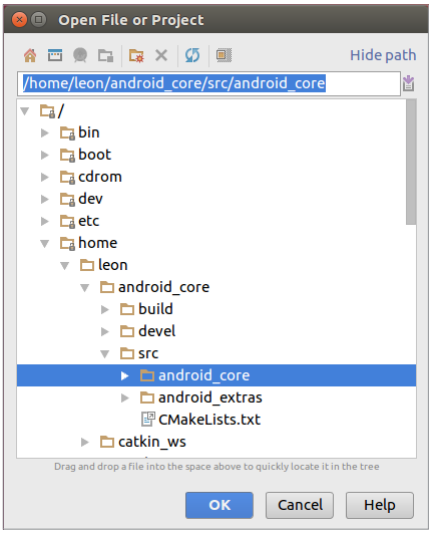

FIGURE 12-18 Import Project

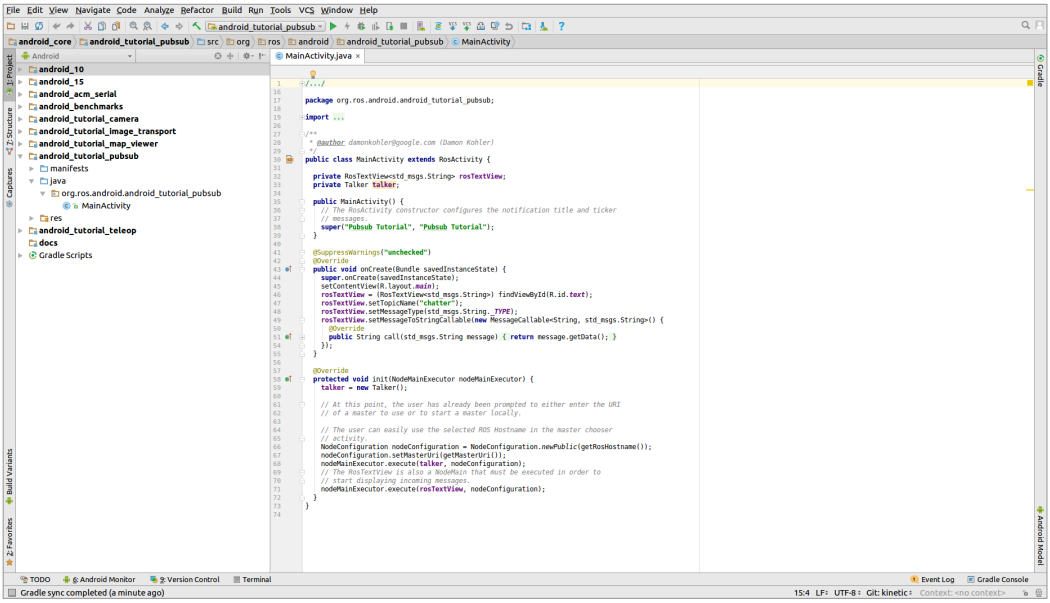

FIGURE 12-19 Import Screen of Android Studio IDE

Next, let's run the 'android_tutorial_pubsub' example. Select 'android_tutorial_pubsub' in the project selection window at the top of the window and click Playback next to it to find the device to execute the program. Select the appropriate device as shown in Figure 12-20 and click OK button.

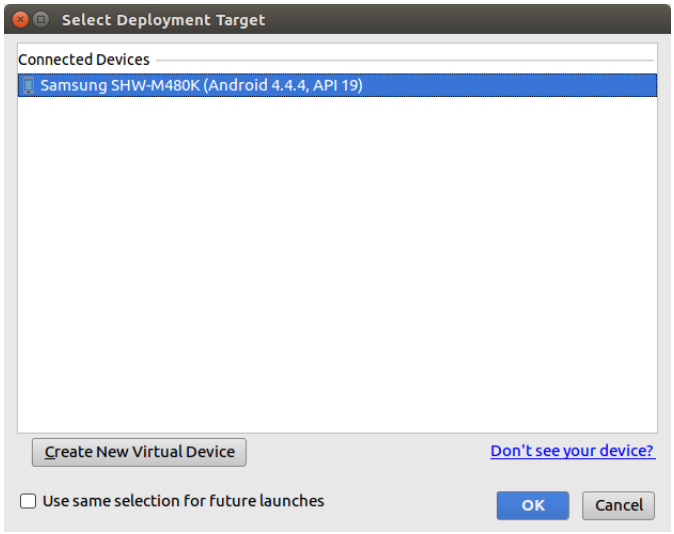

FIGURE 12-20 Device selection window

When the source download on the device is successfully completed, the IP of the ROS master (the computer on which roscore is running) is entered on the terminal as shown in Figure 12-21. Enter the appropriate IP and click Connect.

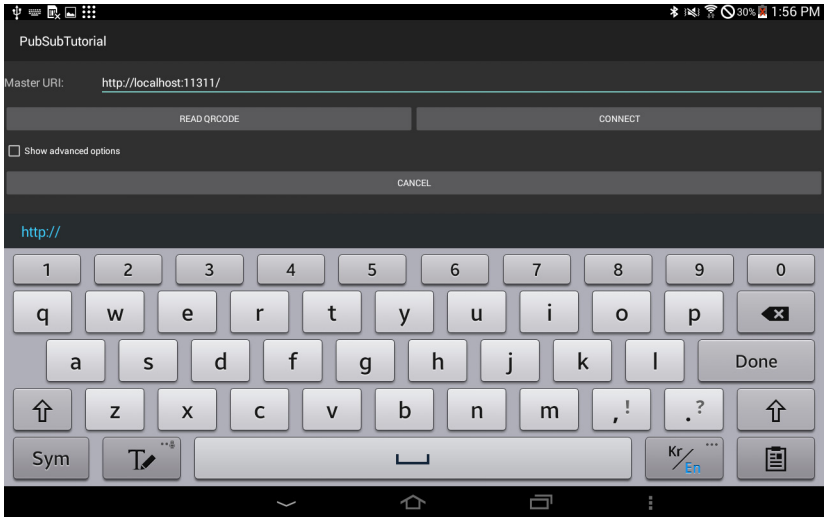

FIGURE 12-21 ROS IP Settings Window

When 'android_tutorial_pubsub' is executed, the device publishes 'Hello world!' followed by increasing number as a type of 'std_msgs::String'. Now, let's subscribe to a string that the device publishes on your computer. If you look at the '/chatter' topic using the 'rostopic echo' command as described previously, you can see that the topic is being published as follows.

```
$ rostopic echo /chatter
data: Hello world! 96
---
data: Hello world! 97
---
data: Hello world! 98
---
data: Hello world! 99
---
data: Hello world! 100
---
data: Hello world! 101
---
```

We looked at SLAM and service robot that have applied navigation in the previous chapter. As described in this chapter, it is not difficult to build a service robot. I close this chapter with hopes that this book helps you build a great robot.