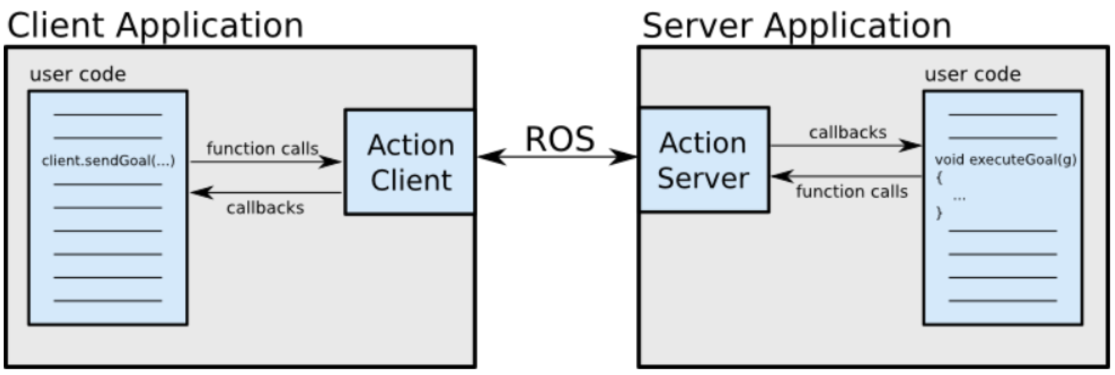
11.ROS action client

11.1 action overview

Action is a commonly used communication method in ros. There is an action client and an action server. It should be mentioned here that action and servcie are both communication methods with feedback, so what is the difference between them? An obvious difference is that action can feedback the current status at a certain frequency, while service can only feedback once.This is a very important point. For example, when executing a long-term task, you can use the action communication method to check the progress at any time, and you can also terminate the request.

11.2 action communication principle

Let’s look at the picture of ros-wiki,



Here it shows that the client sends a goal, which is sendGoal, and the server executes the corresponding program after receiving it, which is excuteGoal. This is a simple process, but in it, the client may also send feedback instructions according to the set frequency, and the server will send the feedback results back to the client after receiving it.

There are a few important concepts to know here:

- Goal

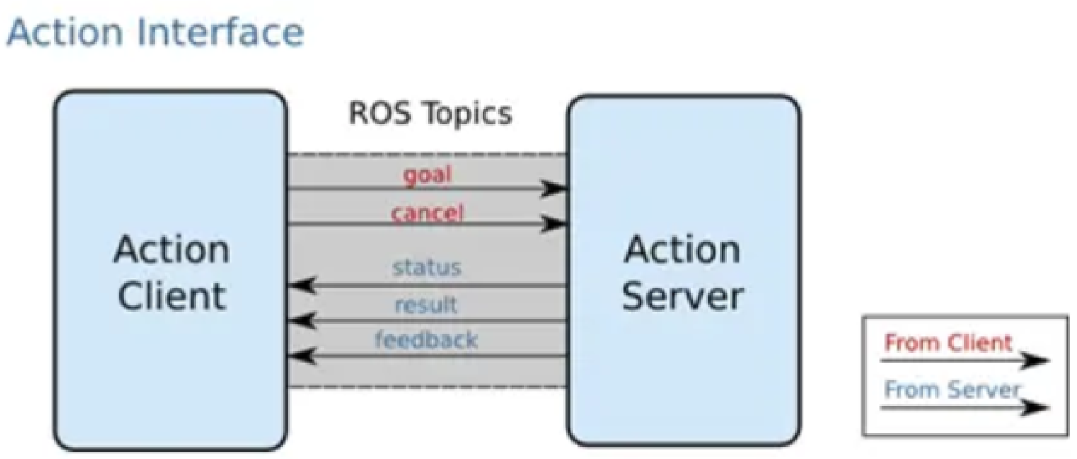
  When a robot performs an action, it should have clear moving target information, telling the server "where I want to go" or "what I want to do", including some parameter settings, direction, angle, speed, etc. This enables the robot to complete action tasks.

- Feedback

  During the action, the server should feed back real-time status information to the client, telling the client "where are we now" or "where have I done".
  It is also convenient for the client to choose to continue or give other instructions.

- Result

  When the movement is completed, the server sends the result data to the client, telling the client "I have arrived now" or "I have completed the action now", so that the client can obtain all the information of this action.For example, it may include the robot's movement duration, final posture, etc.



11.3 action file

Behavior specifications use `.action` files. The `.action` file has the goal definition, then the result definition, then the feedback definition, with each part separated by 3 hyphens (`---`). In the src directory of the workspace, create a new learn_action function package and enter the following instructions:

```
catkin_create_pkg learn_action std_msgs rospy roscpp actionlib actionlib_msgs
```

Then go back to the workspace to compile,

```
cd ~/ros_ws
catkin_make
```

Then create a new folder under the learn_action folder, named action, to store the .action file we defined and input from the terminal.

```
cd ~/ros_ws/src/learn_action
mkdir action
```

In the action folder, create a new file named DoDishes.action and copy the following code into it.

```
# Define the goal
uint32 dishwasher_id  # Specify which dishwasher we want to use
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

Modify the CMakeLists.txt file to the following content,

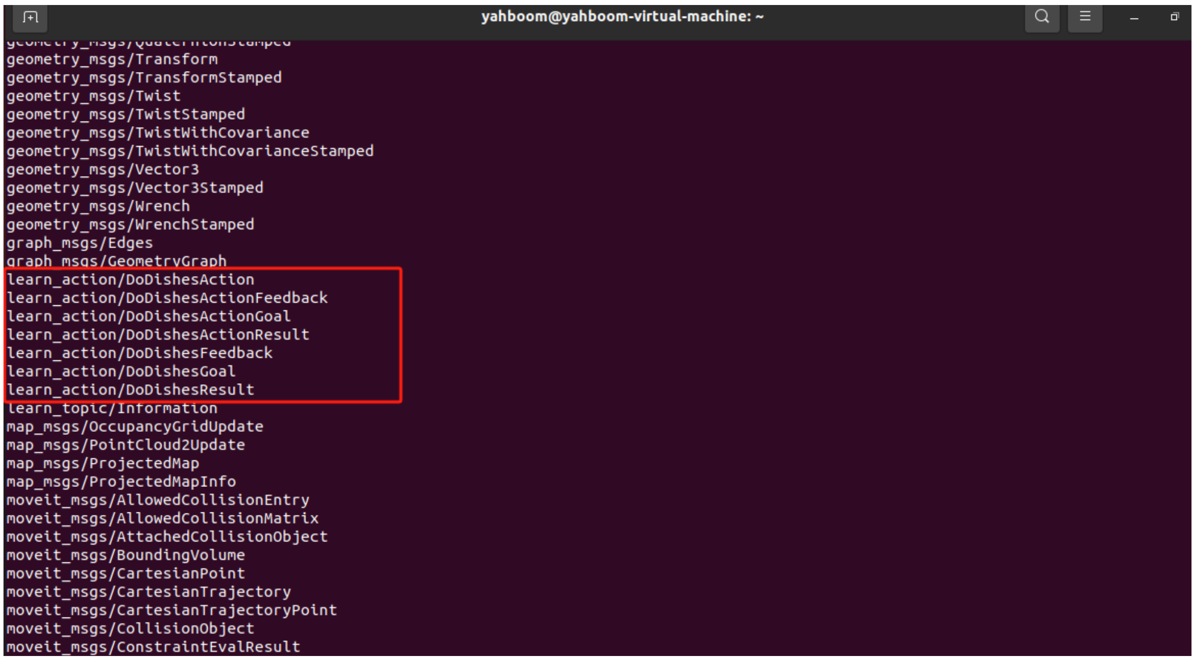
```
find_package(catkin REQUIRED COMPONENTS
  actionlib
  actionlib_msgs
  roscpp
  rospy
  std_msgs
  genmsg
)
add_action_files(
   DIRECTORY
   action
   FILES
   DoDishes.action
)
generate_messages(
   DEPENDENCIES
   actionlib_msgs
 )
```

Then, we go back to the workspace to compile and enter in the terminal,

```
cd ~/ros_ws
catkin_make
```

After successful compilation, for `DoDishes.action`, the following message is generated through `genaction.py` to generate the relevant msg file, which can be seen through rosmsg list and entered in the terminal,

```
rosmsg list
```



```
geometry_msgs/Transform
geometry_msgs/TransformStamped
geometry_msgs/Twist
geometry_msgs/TwistStamped
geometry_msgs/TwistWithCovariance
geometry_msgs/TwistWithCovarianceStamped
geometry_msgs/Vector3
geometry_msgs/Vector3Stamped
geometry_msgs/Wrench
geometry_msgs/WrenchStamped
graph_msgs/Edges
graph_msgs/GeometryGraph
learn_action/DoDishesAction
learn_action/DoDishesActionFeedback
learn_action/DoDishesActionGoal
learn_action/DoDishesActionResult
learn_action/DoDishesFeedback
learn_action/DoDishesGoal
learn_action/DoDishesResult
learn_topic/Information
map_msgs/OccupancyGridUpdate
map_msgs/PointCloud2Update
map_msgs/ProjectedMap
map_msgs/ProjectedMapInfo
moveit_msgs/AllowedCollisionEntry
moveit_msgs/AllowedCollisionMatrix
moveit_msgs/AttachedCollisionObject
moveit_msgs/BoundingVolume
moveit_msgs/CartesianPoint
moveit_msgs/CartesianTrajectory
moveit_msgs/CartesianTrajectoryPoint
moveit_msgs/CollisionObject
moveit_msgs/ConstraintEvalResult
```

As shown in the figure above, it is the msg file related to the generated DoDishes.action.

11.4 Write action client code

Under the learn_action function package, create a new file named action_client.cpp and copy the following content into it.

```
#include <actionlib/client/simple_action_client.h>
#include "learn_action/DoDishesAction.h"
```

```cpp
typedef actionlib::SimpleActionClient<learn_action::DoDishesAction> Client;

// When the action is completed, the callback function will be called once
void doneCb(const actionlib::SimpleClientGoalState& state,
        const learn_action::DoDishesResultConstPtr& result)
{
    ROS_INFO("Yay! The dishes are now clean");
    ros::shutdown();
}

// When the action is activated, the callback function will be called once.
void activeCb()
{
    ROS_INFO("Goal just went active");
}

// Callback function called after receiving feedback
void feedbackCb(const learn_action::DoDishesFeedbackConstPtr& feedback)
{
    ROS_INFO(" percent_complete : %f ", feedback->percent_complete);
}

int main(int argc, char** argv)
{
    ros::init(argc, argv, "do_dishes_client");

    // Define a client
    Client client("do_dishes", true);

    // Waiting for server side
    ROS_INFO("Waiting for action server to start.");
    client.waitForServer();
    ROS_INFO("Action server started, sending goal.");

    // Create an action goal
    learn_action::DoDishesGoal goal;
    goal.dishwasher_id = 1;

    // Send the action goal to the server and set the callback function
    client.sendGoal(goal,  &doneCb, &activeCb, &feedbackCb);

    ros::spin();

    return 0;
}
```

Modify CMakeLists.txt file and add the following content,

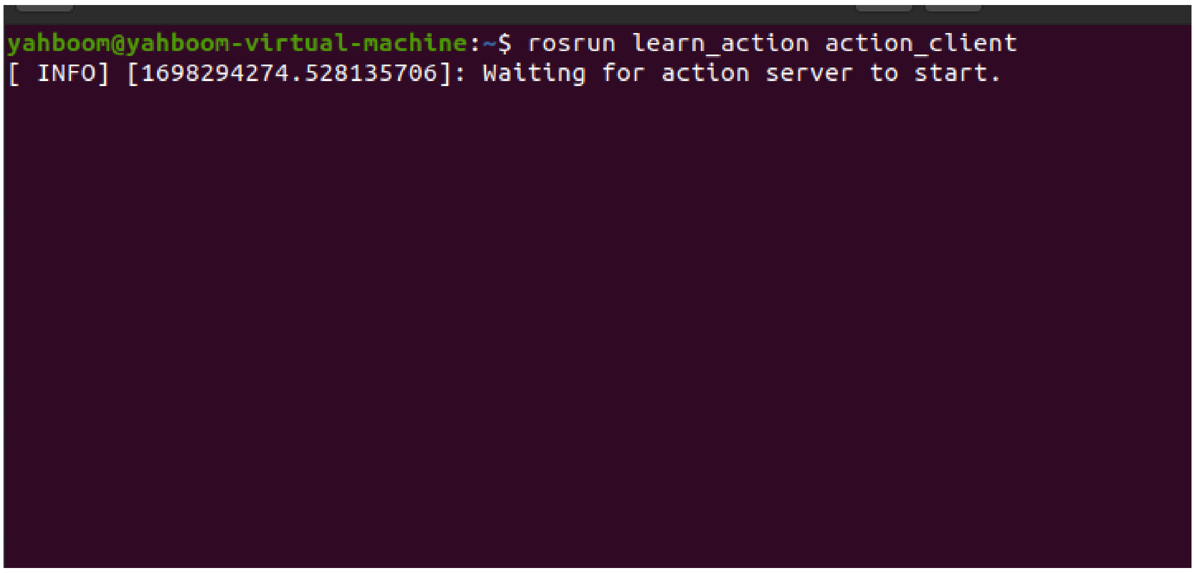
```cmake
add_executable(action_client src/action_client.cpp)
target_link_libraries( action_client ${catkin_LIBRARIES})
add_dependencies(action_client ${${PROJECT_NAME}_EXPORTED_TARGETS})
```

Return to the workspace to compile,

```
cd ~/ros_ws
catkin_make
```

After the compilation is completed, reopen the terminal or source, and then enter the following command to run,

```
rosrun learn_action action_client
```



```
yahboom@yahboom-virtual-machine:~$ rosrun learn_action action_client
[ INFO] [1698294274.528135706]: Waiting for action server to start.
```

Because the server is not running, no corresponding services are provided, or corresponding actions and feedback are performed. In the next section, we learn how to write the server side of the action. Let's look at the core code now.

```
// Define a client
Client client("do_dishes", true);
```

Here we first define a client with two pieces of information. The first is that the service sought by the client is "do_dishes", and the second is true which means starting a thread to provide services for the subscription of this operation.

```
client.waitForServer();
```

Waiting for the `"do_dishes"` service to start

```
learn_action::DoDishesGoal goal;
goal.dishwasher_id = 1;
// Send the action goal to the server and set the callback function
client.sendGoal(goal,  &doneCb, &activeCb, &feedbackCb);
```

Create a goal. The goal member is dishwasher_id, assign it a value of 1, and then sendGoal sends the goal to the server. It includes goal and three callback functions, namely doneCb, activeCb and feedbackCb.

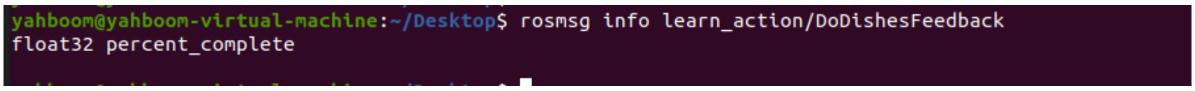
```
// When the action is completed, the callback function will be called once
void doneCb(const actionlib::SimpleClientGoalState& state,
        const learn_action::DoDishesResultConstPtr& result)
{
    ROS_INFO("Yay! The dishes are now clean");
    ros::shutdown();
}

// When the action is activated, the callback function will be called once.
void activeCb()
{
    ROS_INFO("Goal just went active");
}

// callback function called after receiving feedback
void feedbackCb(const learn_action::DoDishesFeedbackConstPtr& feedback)
{
    ROS_INFO(" percent_complete : %f ", feedback->percent_complete);
}
```

Here is the feedback member percen_complete. We can use rosmsg info to query the specific members and enter it in the terminal.

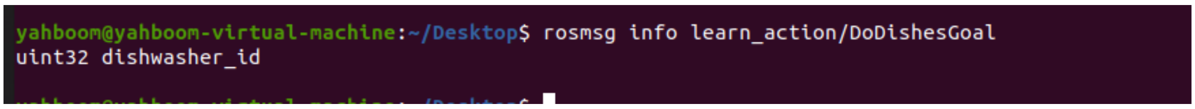
```
rosmsg info learn_action/DoDishesFeedback
```



```
yahboom@yahboom-virtual-machine:~/Desktop$ rosmsg info learn_action/DoDishesFeedback
float32 percent_complete
```

The same applies to the goal above. Enter in the terminal.

```
rosmsg info learn_action/DoDishesGoal
```



```
yahboom@yahboom-virtual-machine:~/Desktop$ rosmsg info learn_action/DoDishesGoal
uint32 dishwasher_id
```

Summary: First create a client to call the service, then wait for the service to start. After the service starts, define a goal, and then send the goal to the server. Please refer to the official documentation for the parameter API interface:

actionlib: actionlib::SimpleActionClient< ActionSpec > Class Template Reference (ros.org)