7.Client

In ROS communication, in addition to topic communication, there is also a type of service communication. Services include both client and server, where the client requests the service and the server provides the service. This section focuses on the client and explains how C++and Python can implement the client.

7.1 Preparation work

7.1.1 Establishing a Function Package

1. Switch to~/catkin_ In the ws/src directory,

```
catkin_create_pkg learning_server std_msgs rospy roscpp geometry_msgs turtlesim
```

2. Switch to~/catkin_ Under the ws directory,

```
catkin_make
```

7.2. C++Language Implementation

7.2.1 Implementation steps

1. Initialize ROS node
2. Create handle
3. Create a Client instance
4. Initialize and publish service request data
5. Wait for the response result after the server processes it

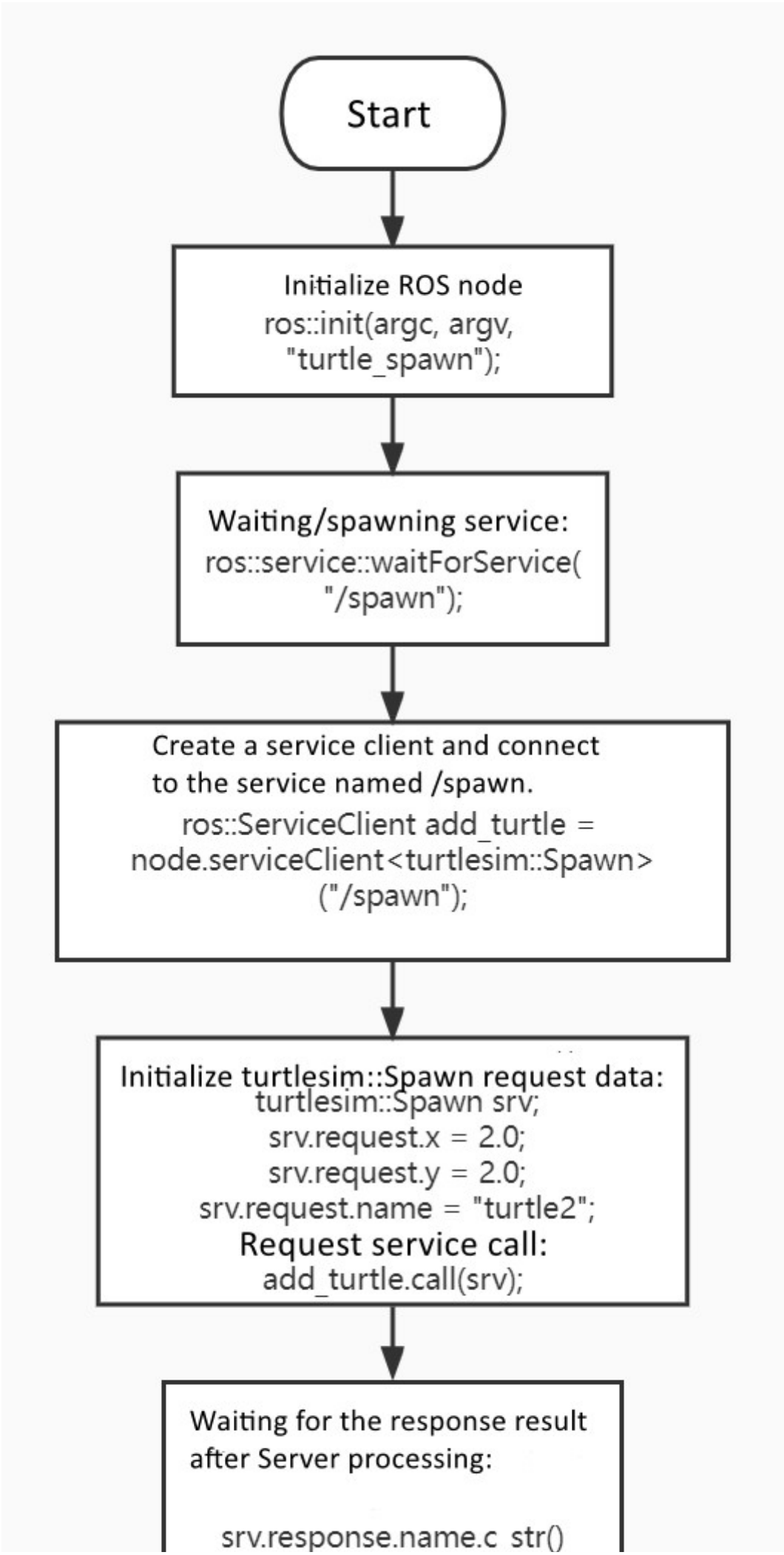
7.2.2. Switch to~/catkin_ ws/src/learning_ Create a new. cpp file under the server/src directory and name it a_ new_ Turtle, paste the following code inside

a_new_turtle.cpp

```
#include <ros/ros.h>
#include <turtlesim/Spawn.h>

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

    ros::init(argc, argv, "a_nes_turtle");// Initialize ROS nodes

    ros::NodeHandle node;

    ros::service::waitForService("/spawn"); // Waiting for/sawn service
```

```
    ros::ServiceClient new_turtle = node.serviceClient<turtlesim::Spawn>
("/spawn");//Create a service client to connect to a service named/sawn

    //Initialize the request data for turtlesim:: Spawn
    turtlesim::Spawn new_turtle_srv;
    new_turtle_srv.request.x = 6.0;
    new_turtle_srv.request.y = 8.0;
    new_turtle_srv.request.name = "turtle2";

    // Request the service to pass in xy location parameters and name parameters
    ROS_INFO("Call service to create a new turtle name is %s,at the
x:%.1f,y:%.1f", new_turtle_srv.request.name.c_str(),
        new_turtle_srv.request.x,
        new_turtle_srv.request.y);

    new_turtle.call(new_turtle_srv);


    ROS_INFO("Spwan turtle successfully [name:%s]",
new_turtle_srv.response.name.c_str());// Display service call results

    return 0;
};
```

1. Process Flow Chart


Start
Initialize ROS node
ros::init(argc, argv,
"turtle_spawn");
Waiting/spawning service:
ros::service::waitForService(
"/spawn");
Create a service client and connect
to the service named /spawn.
ros::ServiceClient add_turtle =
node.serviceClient<turtlesim::Spawn>
("/spawn");
Initialize turtlesim::Spawn request data:
turtlesim::Spawn srv;
srv.request.x = 2.0;
srv.request.y = 2.0;
srv.request.name = "turtle2";
Request service call:
add_turtle.call(srv);
Waiting for the response result
after Server processing:
srv.response.name.c_str()

2. Configure in CMakelist.txt, under the build area, add the following content

```
add_executable(a_new_turtle src/a_new_turtle.cpp)
target_link_libraries(a_new_turtle ${catkin_LIBRARIES})
```

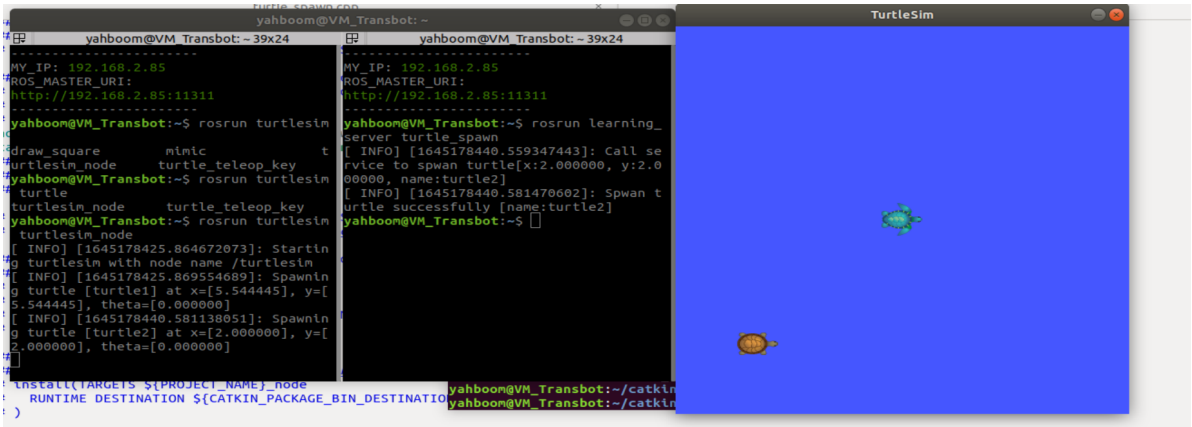
3. Compiling code under workspace directory

```
cd ~/catkin_ws
catkin_make
source devel/setup.bash
```

4. run a program

```
roscore
rosrun turtlesim turtlesim_node
rosrun learning_server a_new_turtle
```

5. Running effect screenshot



6. Program Description

After starting the node of Little Turtle, run a_ new_ The Turtle program will find that there will be another small turtle appearing in the screen, because the turtle's node provides a service/spa wn, which will generate another small turtle Turtle2. To view the services provided by the turtle, you can use the rossservice list command, as shown in the following figure

```
yahboom@VM_Transbot:~/catkin_ws/src/learning_server/src$ rosservice list
/clear
/kill
/reset
/rosout/get_loggers
/rosout/set_logger_level
/spawn
/turtle1/set_pen
/turtle1/teleport_absolute
/turtle1/teleport_relative
/turtle2/set_pen
/turtle2/teleport_absolute
/turtle2/teleport_relative
/turtle6/set_pen
/turtle6/teleport_absolute
/turtle6/teleport_relative
/turtlesim/get_loggers
/turtlesim/set_logger_level
```

You can view the parameters required for this service through rossservice info/spa, as shown in the following figure

```
yahboom@VM_Transbot:~/catkin_ws/src/learning_server/src$ rosservice info /spawn
Node: /turtlesim
URI: rosrpc://192.168.2.85:57303
Type: turtlesim/Spawn
Args: x y theta name
```

It can be seen that there are four parameters required: x, y, theta, and name, which are in a_ new_ There is initialization in turtle.cpp

```
srv.request.x = 6.0;
srv.request.y = 8.0;
srv.request.name = "turtle2";
```

7.3. Python Language Implementation

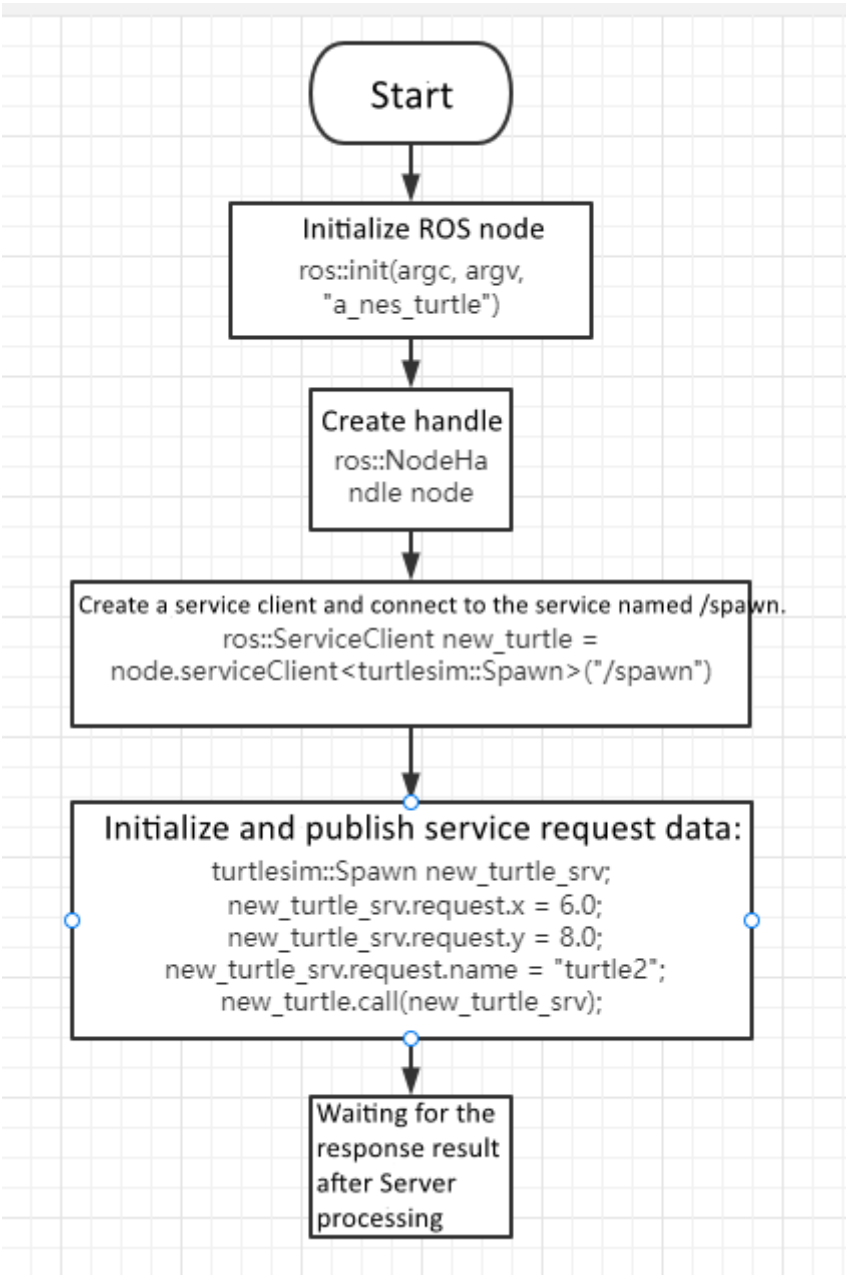
7.3.1. Switch to~/catkin_ ws/src/learning_ Under the server directory, create a new script folder, cut it in, and create a new py file named a_ new_ Turtle, paste the following code inside

a_new_turtle.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-


import sys
import rospy
from turtlesim.srv import Spawn

def turtle_spawn():

    rospy.init_node('new_turtle')# Initialize ROS nodes

    rospy.wait_for_service('/spawn')# Waiting for/sawn service

    try:
        new_turtle = rospy.ServiceProxy('/spawn', Spawn)
```

```
        response = new_turtle(2.0, 2.0, 0.0, "turtle2")# Input request data
        return response.name
    except rospy.ServiceException, e:
        print "failed to call service : %s"%e

if __name__ == "__main__":
    #Call the service and display the call result
    print "a new turtle named %s." %(turtle_spawn())
```

1. Program flowchart



2. run a program

```
roscore
rosrun turtlesim turtlesim_node
rosrun learning_server a_new_turtle.py
```

3. The program operation effect and program description are consistent with the implementation effect of C++. Here, we mainly discuss how Python provides the parameters required for the service,

```
response = add_turtle(2.0, 2.0, 0.0, "turtle2")
```

The corresponding parameters are x, y, theta, and name.