Intermediate concepts in ROS

Prof. V. Muthukumar

Concepts Covered

- Messages (custom Messages)
- Client and Services
- Actions Server
- Parameter Server
- TF

Custom Messages

- Create a custom msg file
- Define and use msg
- Build msg file
  - Modify package.xml
  - Modify CMakeLists.txt

```
#Standard metadata for higher-level flow data types
#sequence ID: consecutively increasing ID
uint32 seq
#Two-integer timestamp that is expressed as:
# * stamp.secs: seconds (stamp_secs) since epoch
# * stamp.nsecs: nanoseconds since stamp_secs
# time-handling sugar is provided by the client library
time stamp
#Frame this data is associated with
string frame_id
```

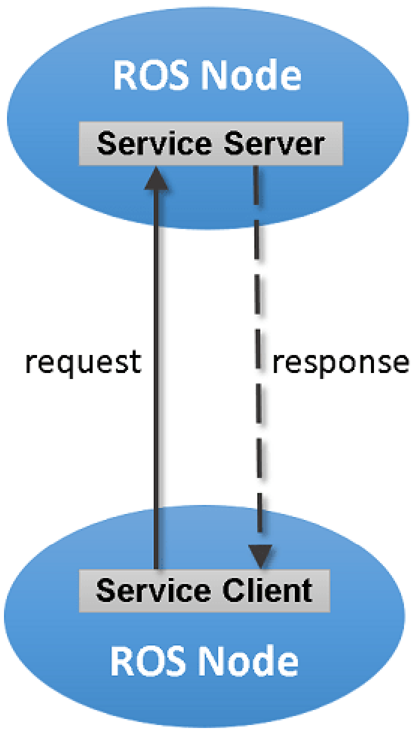
```
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```

```
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
)
```

Custom Messages Demo

ROS Client & Server

A simple



Service Name: **/example_service**
Service Type: **roscpp_tutorials/TwoInts**

Request Type: **roscpp_tutorials/TwoIntsRequest**
Response Type: **roscpp_tutorials/TwoIntsResponse**

ROS Service-Client model:
•**Client**: A node that requests a service from another node.
•**Server**: A node that provides a service, performs a task, and returns a result.
This model uses the **request-response** mechanism:
•A **Service** is defined by a pair of messages: one for the request and one for the response.
•The client sends a request message to the server, which then processes the request and sends back a response.

ROS Service-Client is best for simple, quick, and synchronous tasks where immediate results are needed.

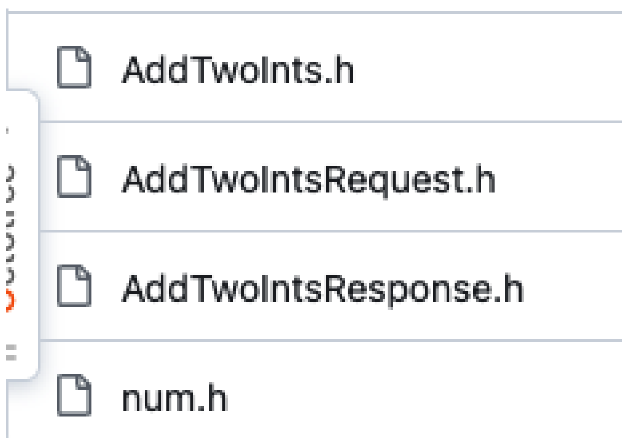
Generate messages in the 'msg' folder
add_message_files(DIRECTORY msg FILES Num.msg)

Generate services in the 'srv' folder

add_service_files(DIRECTORY srv FILES AddTwoInts.srv)

Will autogenerate the following files @:

$ ros_workspace/devel/include/pkg_name/



Simple Example - Service

src/add_two_ints_server.cpp

```
#include "ros/ros.h"
#include "beginner tutorials/AddTwoInts.h”

bool add (beginner tutorials: :AddTwoInts: :Request &req,
beginner tutorials: :AddIwoInts:: Response &res
{
res. sum req.a + req.b;
ROS_INFO ("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
ROS_INFO ("sending back response: [%ld]", (long int)res.sum);
return true;
}
int main (int arge, char **argv)
{
ros: :init (arge, argv, "add two ints server");
ros:: NodeHandle n;
ros: :ServiceServer service = n. advertiseService ("add _two_ints", add);
ROS_INFO ("Ready to add two ints.");
ros::spin();
return 0;
}
```

Simple Example – Client

src/add_two_ints_client.cpp

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

#include "beginner_ tutorials/AddTwoInts.h"

#include <cstdlib>

int main (int argc, char **arqv)

ros::init (argc, argv, "add_two_ints_client");

if (argc != 3)

ROS_INFO ("usage: add_two_ints_client X Y") ;

return 1;


ros: : NodeHandle n;

ros:: ServiceClient client n. serviceClient<beginner tutorials:: AddTwoInts>("add_two_ints");

beginner tutorials:: AddTwoInts srv;

srv.request.a = atoll(argv(1]);

srv.request.b = atoll (argv[2]);

    if (client.call (srv))

    {

     ROS_INFO ("Sum: %ld", (long int )srv. response. sum) ;

    }

    else

    {

       ROS ERROR ("Failed to call service add_two_ ints");

       return 1;

    }

    return 0;

}
```

Running the simple action - service

- roscore
- rosrun beginner_tutorials add_two_ints_server
- rosrun beginner_tutorials add_two_ints_client 1 3
  - Call a service from the command-line:
    - $ rosservice call /add two ints 1 2
  - Pipe the output of rosservice to rossrv to view the srv type:
    - $ rosservice type add two ints | rossrv show
  - Display all services of a particular type:
    - $ rosservice find rospy tutorials/AddTwoInts

rosservice

A tool for listing and querying ROS services.

Commands:

`rosservice list`	Print information about active services.
`rosservice node`	Print the name of the node providing a service.
`rosservice call`	Call the service with the given args.
`rosservice args`	List the arguments of a service.
`rosservice type`	Print the service type.
`rosservice uri`	Print the service ROSRPC uri.
`rosservice find`	Find services by service type.

ROS Class

```
#ifndef EXAMPLE_ROS_CLASS_H_
#define EXAMPLE_ROS_CLASS_H_
#include <math.h>
#include <stdlib.h>
#include <string>
#include <vector>
#include <ros/ros.h>
#include <std_msgs/Bool.h>
#include <std_msgs/Float32.h>
#include <example_srv/simple_bool_service_message.h> // this is a pre-defined service message, contained in shared "example_srv" package


class ExampleRosClass
{
public:
   ExampleRosClass(ros::NodeHandle* nodehandle);
private:
   ros::NodeHandle nh_;
// we will need this, to pass between "main" and constructor
 // some objects to support subscriber, service, and publisher
   ros::Subscriber minimal_subscriber_;
//these will be set up within the class constructor, hiding these ugly details
   ros::ServiceServer minimal_service_;
   ros::Publisher  minimal_publisher_;


   double val_from_subscriber_;
   double val_to_remember_;
// member variables will retain their values even as callbacks come and go
   // member methods as well:
   void initializeSubscribers();
   void initializePublishers();
   void initializeServices();


   void subscriberCallback(const std_msgs::Float32& message_holder);
//prototype for callback of example subscriber
bool serviceCallback(example_srv::simple_bool_service_messageRequest& request,  example_srv::simple_bool_service_messageResponse& response);
}; // note: a class definition requires a semicolon at the end of the definition
#endif
```

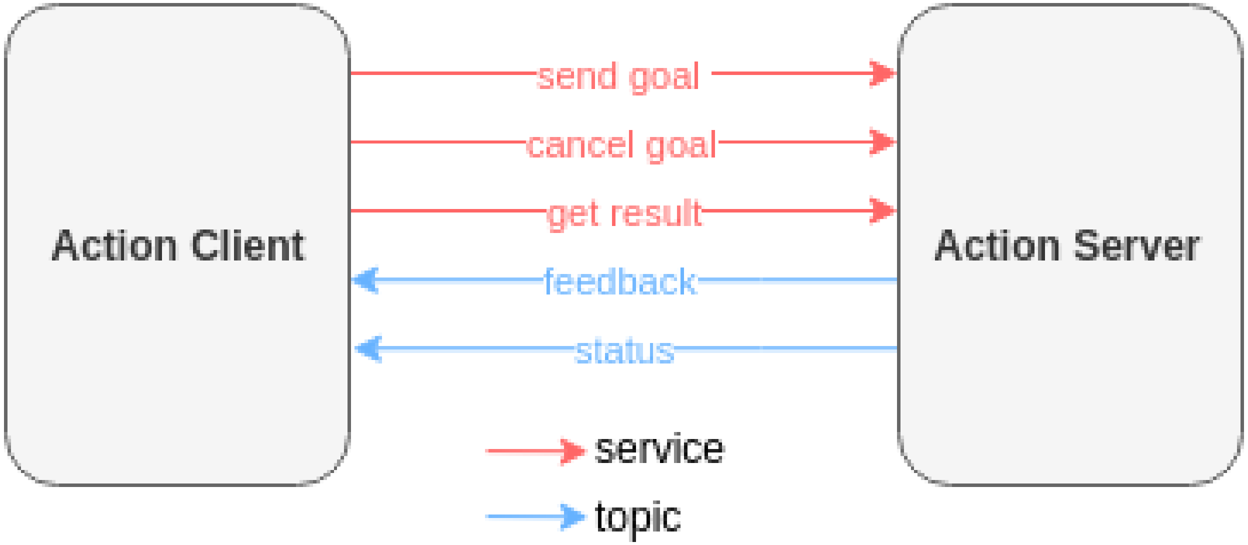
ROS Class

```
#include "example_ros_class.h"


ExampleRosClass::ExampleRosClass(ros::NodeHandle* nodehandle):nh_(*nodehandle)
{ // constructor
   ROS_INFO("in class constructor of ExampleRosClass");
   initializeSubscribers();
   initializePublishers();
   initializeServices();


   //initialize variables here, as needed
   val_to_remember_=0.0;
 }


void ExampleRosClass::initializeSubscribers()
{
   ROS_INFO("Initializing Subscribers");
   minimal_subscriber_ = nh_.subscribe("exampleMinimalSubTopic", 1,  \
                     &ExampleRosClass::subscriberCallback,this);
   // add more subscribers here, as needed
}
```

```
void ExampleRosClass::initializeServices()
{
    ROS_INFO("Initializing Services");
    minimal_service_ = nh_.advertiseService("exampleMinimalService",
                                            &ExampleRosClass::serviceCallback, this);
}

//member helper function to set up publishers;
void ExampleRosClass::initializePublishers()
{
    ROS_INFO("Initializing Publishers");
    minimal_publisher_ = nh_.advertise<std_msgs::Float32>("exampleMinimalPubTopic", 1, true);
}

void ExampleRosClass::subscriberCallback(const std_msgs::Float32& message_holder) {
    val_from_subscriber_ = message_holder.data
    ROS_INFO("myCallback activated: received value %f",val_from_subscriber_);
    std_msgs::Float32 output_msg;
    val_to_remember_ += val_from_subscriber_;
    output_msg.data= val_to_remember_;
    // demo use of publisher--since publisher object is a member function
    minimal_publisher_.publish(output_msg); //output the square of the received value;
}
```

```
bool ExampleRosClass::serviceCallback (example_srv::simple_bool_service_messageRequest& request, \
                        example_srv::simple_bool_service_messageResponse& response) {

    ROS_INFO("service callback activated");

    response.resp = true; // boring, but valid response info

    return true;

}

int main(int argc, char** argv)

{

    // ROS set-ups:

    ros::init(argc, argv, "exampleRosClass"); //node name


    ros::NodeHandle nh; // create a node handle; need to pass this to the class constructor


    ROS_INFO("main: instantiating an object of type ExampleRosClass");

    ExampleRosClass exampleRosClass(&nh);  //instantiate an ExampleRosClass object and pass in pointer to nodehandle for constructor
to use


    ROS_INFO("main: going into spin; let the callbacks do all the work");

    ros::spin();

    return 0;

}
```

ROS Action Client & Server



ROS Action Client-Server model:
•**Action Client**: A node that sends a request to the action server to perform a long-running task.
•**Action Server**: A node that performs the long-running task and sends feedback, as well as the final result.
This model uses the **goal-feedback-result** mechanism:
•The **Action Server** can report progress (feedback) and send a final result when the task is complete.
•The **Action Client** can also cancel a request (goal) if needed.

ROS Action Client-Server is ideal for long-running tasks where feedback, progress tracking, and task cancellation are important.

Client-Server Vs Action Client-Server

Major Differences Between ROS Client-Server and ROS Action Client-Server:

Feature	ROS Service-Client	ROS Action Client-Server
Communication Type	Synchronous (Blocking)	Asynchronous (Non-blocking)
Feedback Mechanism	No feedback after request	Supports feedback during task execution
Use Case	Short tasks with immediate responses	Long-running tasks with progress feedback
Result Retrieval	Immediate after task completion	Can receive result later, after task completion
Task Cancellation	Cannot cancel a task	Can cancel an ongoing task
Example Use Cases	Requesting sensor data, performing quick calculations	Robot navigation, trajectory execution
Task Duration	Suitable for short-duration tasks	Suitable for longer, more complex tasks

demo.action

<u>Demo.action</u>

```
#goal definition
#the lines with the hash signs are merely comments
#goal, result and feedback are defined by this fixed order, and separated by 3 hyphens
int32 input
---
#result definition
int32 output
int32 goal_stamp
---
#feedback
int32 fdbk
```

CMakeLists.txt

```
find_package(catkin REQUIRED COMPONENTS actionlib_msgs)

add_action_files( DIRECTORY action FILES demo.action )
generate_messages( DEPENDENCIES actionlib_msgs std_msgs  # Or other packages containing msgs )
catkin_package( CATKIN_DEPENDS actionlib_msgs )
```

Action Client & Server Example

```
#include<example_action_server/demoAction.h>


int main(int argc, char** argv) {
    ros::init(argc, argv, "demo_action_client_node"); // name this node
    int g_count = 0;
    // here is a "goal" object compatible with the server, as defined in example_action_server/action
    example_action_server::demoGoal goal;
    actionlib::SimpleActionClient<example_action_server::demoAction>
                                                    action_client("example_action", true);
    bool server_exists = action_client.waitForServer(ros::Duration(5.0)); // wait for up to 5 seconds
    //bool server_exists = action_client.waitForServer(); //wait forever

    if (!server_exists) {
        ROS_WARN("could not connect to server; halting");
        return 0; // bail out; optionally, could print a warning message and retry
    }
```

Action Client & Server Example

```
while (true) {
    // stuff a goal message:
    g_count++;
    goal.input = g_count; // this merely sequentially numbers the goals sent
    //action_client.sendGoal(goal); // simple example--send goal, but do not specify callbacks
    action_client.sendGoal(goal, &doneCb);
    // we could also name additional callback functions here, if desired
    //    action_client.sendGoal(goal, &doneCb, &activeCb, &feedbackCb); //e.g., like this
    bool finished_before_timeout = action_client.waitForResult(ros::Duration(5.0));
    //bool finished_before_timeout = action_client.waitForResult(); // wait forever...
    if (!finished_before_timeout) {
      ROS_WARN("giving up waiting on result for goal number %d", g_count);
      return 0;
    } else {
      //if here, then server returned a result to us
    }
  }
void doneCb(const actionlib::SimpleClientGoalState& state, const example_action_server::demoResultConstPtr& result) {
  ROS_INFO(" doneCb: server responded with state [%s]", state.toString().c_str());
  int diff = result->output - result->goal_stamp;
  ROS_INFO("got result output = %d; goal_stamp = %d; diff = %d", result->output, result->goal_stamp, diff);
}
```

Action Client & Server Example

```
class ExampleActionServer {
private:
…
      actionlib::SimpleActionServer<example_action_server::demoAction> as_;
   // here are some message types to communicate with our client(s)
   example_action_server::demoGoal goal_; // goal message, received from client
   example_action_server::demoResult result_;
   // put results here, to be sent back to the client when done w/ goal
   example_action_server::demoFeedback feedback_; // for feedback

public:
   ExampleActionServer(); //define the body of the constructor outside of class definition
   ~ExampleActionServer(void) {}
void executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction>::GoalConstPtr& goal);
};


ExampleActionServer::ExampleActionServer() :   as_(nh_, "timer_action",
boost::bind(&ExampleActionServer::executeCB, this, _1),false)
ROS_INFO("in constructor of exampleActionServer...");
   as_.start(); //start the server running
}
```

Action Client & Server Example

```
void ExampleActionServer::executeCB(const actionlib::SimpleActionServer<example_action_server::demoAction>::GoalConstPtr& goal) {
    ROS_INFO("in executeCB");
    ROS_INFO("goal input is: %d", goal->input);
    //do work here: this is where your interesting code goes
    ros::Rate timer(1.0); // 1Hz timer
    countdown_val_ = goal->input;
    //implement a simple timer, which counts down from provided countdown_val to 0, in seconds
    while (countdown_val_>0) {
       ROS_INFO("countdown = %d",countdown_val_);

       // each iteration, check if cancellation has been ordered
       if (as_.isPreemptRequested()){
          ROS_WARN("goal cancelled!");
          result_.output = countdown_val_;
          as_.setAborted(result_); // tell the client we have given up on this goal; send the result message as well
          return; // done with callback
                }

           //if here, then goal is still valid; provide some feedback
           feedback_.fdbk = countdown_val_; // populate feedback message with current countdown value
           as_.publishFeedback(feedback_); // send feedback to the action client that requested this goal
       countdown_val_--; //decrement the timer countdown
       timer.sleep(); //wait 1 sec between loop iterations of this timer
    }
    //if we survive to here, then the goal was successfully accomplished; inform the client
    result_.output = countdown_val_; //value should be zero, if completed countdown
    as_.setSucceeded(result_); // return the "result" message to client, along with "success" status
}
```

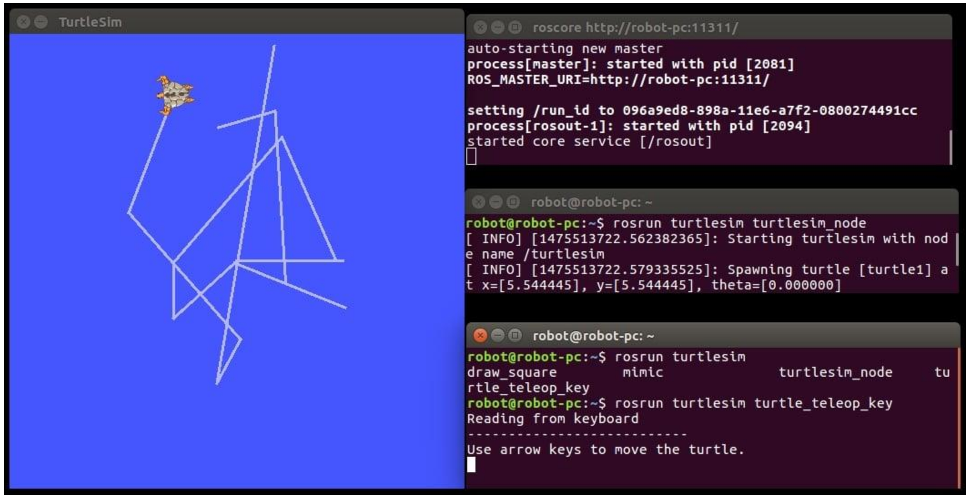
Action Client & Server Demo

Turtlesim

- Turtlesim is a lightweight simulator included with the Robot Operating System (ROS) that serves as a foundational tool for learning ROS concepts.
- **Basic Simulation Environment:**
  - It provides a simple 2D simulation environment where a virtual "turtle" robot can be controlled.
- **ROS Concepts:**
  - Turtlesim is designed to demonstrate fundamental ROS concepts such as: **Nodes:** Independent executable processes that perform specific tasks (e.g., the turtlesim_node itself, or a node controlling the turtle's movement).
  - **Topics:** Channels for communication between nodes, allowing data to be published and subscribed to (e.g., a topic for sending velocity commands to the turtle).
  - **Services:** Request-response communication mechanisms for specific actions (e.g., spawning a new turtle or clearing the simulation screen).
- **Interactive Control:**
  - The turtle can be controlled through various methods, including: **Keyboard Teleoperation:** Using a dedicated node like turtle_teleop_key to control the turtle with arrow keys.
  - **Programmatic Control:** Writing custom ROS nodes in languages like Python or C++ to publish commands to the turtle's velocity topic.

Turtlesim Demo

- Installation and Setup
- Launching Turtlesim
- Controlling the Turtle
- Introducing ROS Concepts
- ROS Graphical User Interface
- Introduction to tf2



Turtlesim – Client Example

```
#include <ros/ros.h>
//The srv class for the service.
#include <turtlesim/Spawn.h>

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

    ros::init(argc, argv, "spawn_turtle");
    ros::NodeHandle nh;

//Create a client object for the spawn service. This
//needs to know the data type of the service and its name.
    ros::ServiceClient spawnClient
            = nh.serviceClient<turtlesim::Spawn>("spawn");

//Create the request and response objects.
    turtlesim::Spawn::Request req;
    turtlesim::Spawn::Response resp;

    req.x = 2;
    req.y = 3;
    req.theta = M_PI/2;
    req.name = "Leo";

    ros::service::waitForService("spawn", ros::Duration(5));
    bool success = spawnClient.call(req,resp);

    if(success){
        ROS_INFO_STREAM("Spawned a turtle named "
                        << resp.name);
    }else{
        ROS_ERROR_STREAM("Failed to spawn.");
    }
}
```

Turtlesim – Service Example

```
#include <ros/ros.h>
#include <std_srvs/Empty.h>
#include <geometry_msgs/Twist.h>

bool forward = true;

bool toggleForward(
        std_srvs::Empty::Request &req,
        std_srvs::Empty::Response &resp){
        forward = !forward;
        ROS_INFO_STREAM("Now sending "<<(forward?
                "forward":"rotate")<< " commands.");
        return true;
}



int main(int argc, char **argv){
        ros::init(argc,argv,"pubvel_toggle");
        ros::NodeHandle nh;

        ros::ServiceServer server =
                nh.advertiseService("toggle_forward",&toggleForward);

        ros::Publisher pub=nh.advertise<geometry_msgs::Twist>(
                "turtle1/cmd_vel",1000);

        ros::Rate rate(2);
        while(ros::ok()){
                geometry_msgs::Twist msg;
                msg.linear.x = forward?1.0:0.0;
                msg.angular.z=forward?0.0:1.0;
                pub.publish(msg);
                ros::spinOnce();
                rate.sleep();
        }
}
```