Contenido

Comandos de terminal de ROS

Creación de los paquetes

catkin_create_package "nombre del paquete" rospy→ Crea un paquete con rospy

catkin_make → Compila todos los paquetes

catkin_make --only_pkg_with_deps “nombre_del_paquete”→ Compila un paquete

Ejecución de nodos

roscore → Inicia el nodo maestro de ROS

rosrun "nombre del paquete" "nombre del archivo" → ejecuta un programa de Ros

Comando para paquetes

rospack list → Devuelve la lista de paquetes ROS

Comandos para los topics

rostopic list →Para ver la lista de topics activos

rostopic echo "topic" → Para ver el contenido de un topic

rostopic info "topic" → Para ver la información de un topic

rostopic pub “mensaje” → Se publica un valor en un topic

Comandos para nodos

rosnode list → Devuelve la lista de nodos activos

rosnode info "nombre del nodo" → Devuelve la información de un nodo

Comando para mensajes personalizados

rosmsg list → Devuelve la lista de todos los mensajes

rosmsg show "Nombre del mensaje" → Devuelve la información de un mensaje

rosmsg list | grep “Mensaje” → Devuelve la ruta de un mensaje

Comando para servicios

rosservice list → Devuelve la lista de servicios activos

rosservice info "servicio" → Devuelve la información de un servicio

rosservice call “servicio” “mensaje” → Se le envía una petición a un servicio

rossrv show "paquete/mensaje" → Devuelve la información de los argumentos de un servicio

Comando para acciones

rostopic list → Devuelve la lista de topics y las acciones

* "acción"/cancel → Cancela la acción
* "acción"/feedback → Muestra el feedback
* "acción" goal → Muestra el objetivo
* "acción"/result → Muestra el resultado
* "acción"/status → Muestra el estado de la acción

rostopic pub "nombre acción/acción" → Ejecuta la acción

rostopic info "acción" → Devuelve la información de la acción

rosmsg list | grep "acción" → Muestra la lista de mensajes de las acciones

* "acción"/MensajeAction →
* "acción"/MensajeActionFeedback →
* "acción"/MensajeActionGoal →
* "acción"/MensajeActionResult →
* "acción"/MensajeFeedback →Mensaje de feedback de una acción
* "acción"/MensajeGoal → Mensaje de petición de una acción
* "acción"/MensajeResult →Mensaje de resultado de acción

rosmsg show "acción"/Mensaje___ " → Para devolver información del mensaje

Python

Comandos de Rospy

Nodos

rospy.init_node(" ") → Creación de un nodo

Topics

"nombre" = Publisher("nombre", Tipo de mensaje, queue_size=10) → Creación de un publicador

"nombre".publish(mensaje) →Publica el mensaje en el topic

rospy.Subscriber("nombre", Tipo de mensaje, callback) → Creación de un suscriptor

Mensajes

Tipos de mensajes básicos : Int32,Int64, String …

"nombre" = Int32()→ Creación de un mensaje

"nombre"."variable del mensaje" = "valor" → Asignación de un valor al mensaje

Bucles

rospy.spin()→Bucle infinito

rospy.is_shutdown()→ Se usa en un While not para finalizar el bucle con **Control+C**

"nombre"=rospy.Rate("frecuencia")→ Crea una frecuencia para un bucle

"nombre".sleep()→ Pausa para tener la frecuencia del bucle

Servicios

Servidor

"nombre"= rospy.Service(Nombre, Tipo de mensaje, callback)

Clientes

rospy.wait_for_services("nombre") → Espera a que el servidor este activo

"nombre_servicio"= rospy.ServiceProxy("/servicio", Tipo de mensaje) → Creación de un cliente para un servicio

“nombre” = “nombre_servicio”(“petición”) → Envía la petición al servicio

Acciones

Servidor

“nombre”= actionlib.SimpleActionServer("nombre",”mensaje”,self.goal_callback,False)→ Crea el servidor

“nombre”.start() → Inicia el servidor

self.”nombre”.is_preempt_requested() → Comprueba si hay una cancelación anticipada

self.”nombre”.set_preempted() → Establece el estado como cancelado anticipadamente

self.”nombre”.publish_feedback(self.__retroalimentacion) → Envia el feedback

self.”nombre”.set_succeeded(self.__resultado) → Envia el resultado

self.”nombre”.set_aborted() → Establece el cestado como abortado

Cliente

“nombre”= actionlib.SimpleActionClient (“nombre”,Tipo de mensaje) →Crea un cliente

“nombre”.wait_for_server() → Espera por el servidor de acción

“nombre”.send_goal(“mensaje”,done_cb=“callback_resultado”,feedback_cb=callback_feedback)→ envia el mensaje

“nombre”.cancel_goal() → Cancela el servidor anticipadamente

Status = “nombre”.get_status() → Para obtener el estado de la acción con el siguiente código: 0 = pendiente , 1 = ejecutándose , 2= finalizado , 3= warning, 4 = error , 5 = cancelado

“nombre”.wait_for_result() → Espera al que el servidor termine de ejecutar la acción

Topics

Publicador

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32

#Creamos el nodo principal
rospy.init_node("Primer nodo")

#Creamos el Publicador
pub = rospy.Publisher("laser", Int32, queue_size=10)
#Creamos el mensaje
a = Int32()
contador = 0
#Creamos la frecuencia del bucle
rate = rospy.Rate(10)

while not rospy.is_shutdown():
    contador += 1
    a.data = contador
    #Publicamos el mensaje
    pub.publish(a)
    #Esperamos para que se cumpla el tiempo del bucle
    rate.sleep()
```

Suscriptor

```python
#!/usr/bin/env python
import rospy
from std_msgs.msg import Int32

def callback(msg):
    """Funcion que se ejecuta cuando se recibe un mensaje    """
    print (msg)

#Creamos el nodo
rospy.init_node('lectura')

#Nos suscribimos al topic laser
rospy.Subscriber('nombre', Int32, callback)

#Creamos el bucle
rospy.spin()
```

Mensajes personalizados

1. Crear una carpeta msg en el paquete
2. Generamos un archivo “nombre”.msg y añadimos las variables del mensaje
3. Modificar el archivo CMakeLists.txt

    1) En la función **find_package ()** dentro de ella escribimos "**message_generation**" y también añadimos los **paquetes** que use el mensaje creado (En líneas diferentes)
    2) En la función **add_message_files ()** la descomentamos y añadimos el **nombre del archivo .msg** que creamos (En líneas diferentes)
    3) En la función **generate_messages ()** la descomentamos y **añadimos el nombre de los paquetes** que usamos en el mensaje creado
    4) En la función **catkin_package ()** la descomentamos y en la **línea de CATKIN_DEPENDS añadimos rospy y message_rutine,** pero dejamos comentado la línea del INCLUDE
    5) Comprobamos si la función **include_directories() esta descomentada y si ${catkin_INCLUDE_DIRS} esta dentro** de ella,

4.**Modificar package.xml**

1) En el final modificamos el archivo para que siga el siguiente esquema

<build_depend> rospy </build_depend>

<build_depend> message_generation </build_depend>

<build_export_depend> rospy </build_export_depend>

<build_export_depend> message_runtine </build_export_depend>

<exec_depend> rospy </exec_depend>

<exec_depend> message_runtine </exec_depend>

5. **Compilar el paquete** ejecutando **catkin_make --only_pkg_with_deps “nombre_del_paquete”** dentro de la carpeta catkin_ws

Servicios

Cliente

```python
#!/usr/bin/env python
import rospy
from paquete.srv import Mensaje, MensajeRequest

#Crea el nodo principal
rospy.init_node('cliente_servicio')

#Esperamos a que el servicio este disponible
rospy.wait_for_service('/nombre_servicio')

#Se crea la conexion con el servicio
servicio = rospy.ServiceProxy("/servicio", "nombre del servicio")

#Creamos la petición del servicio
peticion = MensajeRequest()
peticion.dato = "dato"

#Enviamos la petición al servicio
resultado = servicio(peticion)

#Se imprime el resultado
print(resultado)
```

Servidor

```python
#!/usr/bin/env python
import rospy
from std_srvs.srv import Empty, EmptyResponse

def mi_callback(request):
    """Funcion que sera ejecutada cuando se llame al servicio mi_servicio"""
    print("Peticion recibida")
    return (EmptyResponse) #Retorno un mensaje vacio

#Inicializa el nodo principal
rospy.init_node('servidor')

mi_servicio = rospy.Service("mi_servicio", Empty, mi_callback) #Crea el
servicio. Argumentos: Nombre del servicio, Tipo de mensaje, Funcion que se
ejecuta cuando se recibe una peticion

#Mantiene el nodo activo
rospy.spin()
```

Mensajes personalizados

1. Crear una carpeta srv en el paquete del servidor del servicio
2. Creamos y editamos el fichero "Nombre".srv con el siguiente esquema

```
#Petición
int32 dato
---
#Respuesta
bool éxito
```

3. Modificamos el CMLists.txt para que se compile el servicio
   1) En la función **find package()**, la descomentamos y **añadimos message_generation** y añadimos las **dependencias** del archivo
   2) En la función **add_service_files añadimos el nombre del archivo** de mensaje
   3) En la función **generate_messages**, la descomentamos y **añadimos las dependencias del archivo**
   4) En la función **catkin_package añadimos rospy y message_runtime en la línea de CATKIN_DEPENDS**

4. **Modificar package.xml**

   1.En el final modificamos el archivo para que siga el siguiente esquema.

   <build_depend> rospy </build_depend>

   <build_depend> message_generation </build_depend>

   <build_export_depend> rospy </build_export_depend>

   <build_export_depend> message_runtine </build_export_depend>

   <exec_depend> rospy </exec_depend>

   <exec_depend> message_runtine </exec_depend>

5. **Compilamos** el paquete usando **catkin_make --only_pkg_with_deps “nombre_del_paquete”** dentro de la carpeta catkin_ws

Acciones

Cliente

```
#!/usr/bin/env python3
import rospy
import actionlib
from action_server.msg import MensajeAction, MensajeActionGoal,
MensajeActionResult , MensajeActionFeedback

def mi_funcion(feedback):
    """Funcion que se ejecuta cuando el servidor envia feedback"""
    print(feedback)

def acabado(result):
 """Funcion que se ejecuta cuando el servidor termina de ejecutarla accion"""
    print("El resultado es: ", result.resultado)

#Inciamos el nodo de accion_cliente
rospy.init_node("accion_cliente")

#Creamos el cliente de accion
cliente = actionlib.SimpleActionClient("accion_servidor",MensajeAction)
#Esperamos al servidor de acciones
cliente.wait_for_server()

#Creamos el mensaje goal
goal = MensajeGoal()
goal.ciclos = 10

#Enviamos el mensaje goal
cliente.send_goal(goal,done_cb = acabado, feedback_cb=mi_funcion)
rospy.spin()
```

Servidor

```
#!/usr/bin/env python3
import rospy
import actionlib
from servidor_action.msg import MensajeAction, MensajeActionResult,
MensajeActionFeedback


class accion_simple(object):
    #Creamos dos variables privadas que seran mensajes
    __retroalimentacion = MensajeFeedback()
    __resultado = MensajeResult()
```

```
    def __init__(self):
        #Creamos el servidor
 self.__act_serv=actionlib.SimpleActionServer("accion_servidor",MensajeAction,
self.goal_callback,False)
        #Iniciamos el servidor
        self.__act_serv.start()

    def goal_callback(self,datos):
        """Funcion que se ejecuta cuando se pide una accion al servidor
        RETURN: MensajeResult"""
        #Creamos la variable con los datos que llegan a la accion
        Datos = datos.goal

        #Definimos la variable del estado de la accion
        succes = False

        #Definimos una variable par el estado de cancelado
        cancelado = False
        #Creamos un bucle
        while condicion :
            #<--> Acciones <-->
            #Comprobamos que no hay una cancelacion anticipateda
            if self.__act_serv.is_preempt_requested():
            #Fijamos el estado de la accion como cancelada anticipatedamente
                cancelado = True
                self.__act_serv.set_preempted()
                break
            #Definimos el valor del feedback
            self.__retroalimentacion.feedback = 1
            #Publicamos el valor del feedback
            self.__act_serv.publish_feedback(self.__retroalimentacion)
        #Comprobamos si el proceso esta finalizado
        if cancelado== False :
            if succes:
                #Definimos el valor del resultado
                self.__resultado.resultado = 1
                #Ponemos el estado como finalizado y devolvemos el resultado
                self.__act_serv.set_succeeded(self.__resultado)
            else :
 #Definimos el estado como abortado pero si no fue cancelada anticipatedamente
                self.__act_serv.set_aborted()

#FIXME:Modulo principal
rospy.init_node("accion_servidor")#Inciamos el nodo de accion_servidor
accion_simple() #Creamos la clase
rospy.spin()
```

Mensajes personalizados

1. Creamos la carpeta action
2. Creamos el archivo “nombre”.action y añadimos los valores del mensaje

#Goal

#Resultado

#Feedback

3. Modificamos el CMakelists.txt
   1) En la función **find_package REQUIRED COMPONETS** añadimos **actionlib_msgs** y **las librerias** de que depende el mensaje
   2) En la función **add_action_files** añadimos los archivos "nombre".action
   3) En la función **generate_messages** añadimos **actionlib_msgs** y las **librerías** que depende el mensaje
   4) Comprobamos la función **catkin_package** esta descomentada y **añadimos rospy** y dejamos INCLUDE comentado
   5) Comprobamos que la función **include_directories** esta descomentada

4. Modificaciones del package.xml
   1) Añadimos las siguientes línea

<build_depend>actionlib</build_depend>

<build_depend>actionlib_msgs</build_depend>

<build_export_depend>actionlib</build_export_depend>

<build_export_depend>actionlib_msgs</build_export_depend>

<exec_depend>actionlib</exec_depend>

<exec_depend>actionlib_msgs</exec_depend>

5. **Compilamos** el paquete usando **catkin_make --only_pkg_with_deps “nombre_del_paquete”** dentro de la carpeta catkin_ws

C++

Comando de C++

Nodos

ros::init(argc, argv ,”nombre”) → Crea el nodo

ros::NodeHandle nh;→ Crea el objeto NodeHandle

Topics

ros:: Publisher “nombre” = nh.advertise<”Tipo de mensaje”>(“nombre”, “tamaño de la cola”); → Crea un publicador

ros::Subscriber “nombre” = nh.subscribe("topic", “tamaño cola”, “Callback”); → Crea un suscriptor

Mensajes

“carpeta de mensaje”::”Tipo de mensaje” “nombre variable” → Crea un mensaje

“nombre”. “campo” → asigna un valor al campo

Bucles

ros::spin() → Crea un bucle infinita

ros::Rate loop_rate(“frecuencia”) → crea un bucle con frecuencia

ros::ok() → Comprueba en un bucle si se cancelo el proceso

ros::MultiThreadedSpinner spinner(n); → crea un n numero de hilos

Servicios

Cliente

ros::ServiceClient nombre = nh.serviceClient<Mensaje >("nombre del servicio");→ Crea un cliente

client = waitForExistence(); →Espera que el servicio este disponibles

client.call(“Mensaje”) → Se envia la petición(se añade en un if)

Servidor

ros::ServiceServer “nombre” = nh.advertiseService<MensajeRequest , MensajeResponse >("Servidor", callback); → Crea un servidor

Acciones

Cliente

actionlib::SimpleActionClient<Mensaje> “nombre” (“Nombre_servicio", true);→ Crea un cliente

cliente->waitForServer(); → Espera que la actividad este actividad

cliente->sendGoal(“peticion”, doneCB , activeCB, feedbackCB); → Envia el mensaje

bool rdo=cliente->waitForResult(ros::Duration(30)); → Espera un tiempo por el resultado

Servidor

actionlib::SimpleActionServer<MensajeAcction>servidor(nh,"Servidor",cbaccion ,false)→Crea un servidor

servidor.start();→ Inicial el servidor de acción

servidor->isPreemptRequested() → Comprueba que no se cancela

servidor->publishFeedback(feedback);→ Envia el Feedback

servidor->setSucceeded(result); → Envia el resultado y pone el estado como resuelto

servidor->setPreempted(result); → Envia el resultado y pone el estado como cancelado

Compilación de archivos

Para compilar un archivo de roscpp tenemos que modificar el Cmakelist.txt en el apartado Build

Modificaciones:

1. Descomentamos la función add_executable

Resultado: add_executable(“nombre binario” src/”nombre archivo”.cpp)

2. Descomentamos la función add_dependencies

Resultado: add_dependencies(“nombre binario” ${${PROJECT_NAME}_EXPORTED_TARGETS} ${catkin_EXPORTED_TARGETS})

3. Descomentamos la función target_link_libraries

Resultado:
```
target_link_libraries(“nombre binario”
        ${catkin_LIBRARIES}
)
```

4. Compilamos con el comando **catkin_make --only_pkg_with_deps “nombre_del_paquete”**
5. **Lanzamos** el binario con **rosrun “nombre paquete” “nombre binario”**

Topics

Publicador

```cpp
#include <ros/ros.h>
#include <std_msgs/Int32.h>
#include <iostream>
using namespace std;

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

    //Creamos el nodo principal del programa
    ros::init(argc, argv, "Nodo publicador");
    ros::NodeHandle nh;

    //Creamos el Publicador
    ros::Publisher pub = nh.advertise<std_msgs::Int32>(contador, 1);

    //Creacion del mensaje
    std_msgs::Int32 cont;
    cont.data = 0;

    //Creacion de la frecuencia de publicacion
    ros::Rate loop_rate(1);

    //Bucle de publicacion
    while (ros::ok()){
        pub.publish(cont);
        cont.data++;
        loop_rate.sleep();
    }

    return 0;
}
```

Suscriptor

```cpp
#include <ros/ros.h>
#include <std_msgs/Int32.h>

void Callback(const std_msgs::Int32::ConstPtr& msg)
{
    /*Funcion que se ejecuta cuando llega un mensaje por el topic subcriptor*/
    ROS_INFO("I heard: [%d]", msg->data);
}
```

```cpp
int mian(int argc, char **argv){

    //Creamos el nodo principal
    ros::init(argc, argv, "subcriptor");
    ros::NodeHandle nh;

    //Creamos el subcriptor
    ros::Subscriber sub = nh.subscribe("topic", 1000, Callback);

    //Creacion del bucle infinito
    ros::spin();

    return 0;

}
```

Mensajes personalizados

1. Creamos la carpeta msg
2. Creamos el archivo “nombre”.msg y añadimos los valores del mensaje
3. Modificamos el CMakelists.txt
    1) En la función **find_package REQUIRED COMPONETS** añadimos **message_generation**
    2) En la función **add_message_files** añadimos los archivos "nombre".msg
    3) En la función **generate_messages** añadimos las **librerías** que depende el mensaje
    4) Comprobamos la función **catkin_package** esta descomentada y **añadimos cpp y message_runtime**

4. Modificaciones del package.xml
    1) Añadimos las siguientes líneas al final

        <build_depend>message_generation</build_depend>

        <exec_depend>message_runtime</exec_depend>

5. **Compilamos** el paquete usando **catkin_make --only_pkg_with_deps “nombre_del_paquete”** dentro de la carpeta catkin_ws

Para poder usarla en un paquete al generar el paquete es necesario añadirlo

catkin_create_pkg “nombre paquete ” roscpp std_msgs “nombre del paquete”

Servicios

Cliente

```
#include "ros/ros.h"
#include "Servicio/Prueba.h"
#include "iostream"
using namespace std;

int main(int argc, char **argv){
    //Creamos el nodo principal
    ros::int_node("Cliente", argc, argv);
    ros::NodeHandle nh;
    //Creacion del cliente de servicios
    ros::ServiceClient client = nh.serviceClient<std_msgs::Int32>("Servidor");
    //Espremos para que el servidor este listo
    client = waitForExistence();
    ROS_INFO("Servidor listo");
    //Creamos el mensaje
    Servicio::Prueba peticion;
    peticion.request.valor_peticion = 5;
    //Llamamos al servicio
    if(client.call(msg)){
        //Si el servicio se ejecuto correctamente
        ROS_INFO("El servicio finalizo correctamente");
        //Imprimimos el resultado
        ROS_INFO("El resultado es: %d", msg.response.valor_respuesta);
        return 0
    }
    else{
        ROS_ERROR("Fallo al llamar al servicio");
        return 1;
    }
}
```

Servidor

```
#include <ros/ros.h>
#include <Servicio/Prueba.h>
#include <iostream>
using namespace std;
bool callback (Servicio::PruebaRequest &req, Servicio::PruebaResponse &res){
    /*Funcion que se ejecuta cuando llega una peticion al servidor*/
    //Ejemplo
    ROS_INFO("Recibido: %s", req.mensaje.c_str());
    //Enviamos la respuesta
    res.valores_respuesta = 0
    return true;
}
```

```
int main(int argc, char ** argv){
    //Creacion del nodo principal
    ros::init_node("Servidor", argc, argv);
    ros::NodeHandle nh;
    //Creacion del servidor de servicios
    ros::ServiceServer server = nh.advertiseService<Servicio::PruebaRequest ,
Servicio:PruebaResponse >("Servidor", callback);
    //Creamos varios hilos de escucha
    ros::MultiThreadedSpinner spinner(4);
    spinner.spin();
}
```

Servidor con clases

```
#include <ros/ros.h>
#include <Prueba/MensajeServicio.h>
class Servidor{
    /*Clase de un servidor de servicios*/
    private:
        //Creacion de variables privadas
        ros::NodeHandle nh;
        ros::ServiceServer server;
    public:
        //Creamos el constructor
        Servidor(){
           server = nh.advertiseService("servicio", &Servidor::callback, this);
        }

  bool callback(Preueba::MensajeServicioRequest &req,
     Prueba::MensajeServicioResponse &res){
        /*Función callback del servicio*/
        ROS_INFO("Recibido: %s", req.mensaje.c_str());
        //Enviamos la respuesta
        res.valor_respuesta = 0;
        return true;
    }
}

int main(int argc, char **argv){
    //Creo el nodo principal
    ros::init(argc, argv, "servidor_con_clase");
    //Creo el objeto del servicio
    Servidor servidor;
    ros::spin();
    return 0;
}
```

Masajes Personalizados

1. **Creamos una carpeta srv**
2. Creamos un **archivo "nombre".srv** y **lo modificamos** con el siguiente esquema

   #Peticion

   #Respuesta

3. **Modificamos el CmakeLsit.txt**
   1) En la **función find_package** añadimos **message_generation** y añadimos las dependencias del mensaje
   2) En la **función add_service_files añadimos** el nombre del archivo **“nombre”.srv**
   3) **Descomentamos** la función **generate_messages**
4. **Modificamos el package.xml** añadimos las siguientes líneas:

   <build_depend>message_generation</build_depend>

   <exec_depend>message_runtime</exec_depend>

5. **Compilamos el paquete**

Acciones

Clientes

```
//tenemos que añadir en el paquete actionlib
#include <ros/ros.h>
#include <Prueba/MensajeAccionAction.h>
#include <actionlib/client/simple_action_client.h>
#include <iostream>
using namespace std;

//Definicion de variables globales
actionlib::SimpleActionClient<Prueba::MensajeAccionAction> *cliente=NULL;

void doneCB(const actionlib::SimpleClientGoalState& state, const
Prueba::MensajeAccionResultConstPtr& result){
    /*Funcion que se ejecuta cuando se termina la accion*/
    ROS_INFO("Terminado y el resultado es: %d", result->resultado.resultado);
}

void feedbackCB(const Prueba::MensajeAccionFeedbackConstPtr& feedback){
    /*Funcion que se ejecuta cuando se envia feedback*/
    ROS_INFO("Progreso: %d", feedback->progreso);
    //Calcelacion de la accion
    if (feedback->progreso == 50){
        ROS_INFO("Cancelando accion");
        cliente->cancelAllGoals();
    }
}
```

```
void activeCB(){
    /*Funcion que se ejecuta cuando se activa la accion*/
    ROS_INFO("Accion activa");
}
int main(int argc, char ** argv ){
    //Creamos el nodo principal
    ros::init(argc, argv, "nodo_cliente");
    ros::NodeHandle nh;
    //Creamos el cliente
    cliente =new actionlib::SimpleActionClient<Prueba::MensajeAccionAction>
("Servidor_accion", true);
    //Esperamos a que el servidor se inicie
    cliente->waitForServer();
    ROS_INFO("Servidor_accion conectado");
    //Creamos el mensaje
    Prueba::MensajeAccionGoal goal;
    goal.numero = 5;
    // FIXME: (Opcion 1)
    // Enviamos el mensaje
    cliente->sendGoal(goal, doneCB , activeCB, feedbackCB);
    //Esperamos a que el servidor nos devuelva el resultado
    bool rdo=cliente->waitForResult(ros::Duration(30));
    if(rdo){
        ROS_INFO("El estado del servidor es: %s", cliente-
>getState().toString().c_str());
        ROS_INFO("El resultado es: %d", cliente->getResult()->resultado);
    }
    else{
        ROS_INFO("El servidor no ha devuelto el resultado");
    }
    //FIXME: (Opcion 2)
    //Esperamos a que el servidor nos devuelva el resultado
    action::SimpleClientGoalState estado = cliente->getState();
    while (estado == actionlib::SimpleClientGoalState::PENDING or estado ==
actionlib::SimpleClientGoalState::ACTIVE){
        ROS_INFO("El estado del servidor es: %s", estado->toString().c_str());
        estado = cliente->getState();
    }
    ROS_INFO("El resultado es: %d", estado.getResult()->result)
    return 0;
}
```

Servidor

```
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include <Prueba/MensajeAccionAction.h>
#include <iostream>
using namespace std;
//Definicion de variables globales
actionlib::SimpleActionServer<Prueba::MensajeAccionAction> *servidor=NULL;
void cbaccion(const Prueba::MensajeAccionGoalConstPtr &goal){
    /*Funcion que se ejecuta cuando se pide una accion al servidor */
    //Creamos los mensajes
    Prueba::MensajeAccionFeedback feedback;
    Prueba::MensajeAccionResult result;
    bool estado = true;
    //FIXME: Accion (Ejemplo)
    for(int i=0; i<goal->numero; i++){
        //Comprobamos si se ha cancelado la accion
        if (servidor->isPreemptRequested()){
            ROS_INFO("El cliente ha cancelado la accion");
            estado = false
            break;
        }
        //Enviamos el feedback
        feedback.progreso = 0;
        servidor->publishFeedback(feedback);
    }
    //Enviamos el resultado
    result.resultado = 0;
        //Comprobamos que no se ha cancelado la accion
    if (estado == true){
        servidor->setSucceeded(result);
    }
    else{
        servidor->setPreempted(result);
    }
}
int main(int argc, char ** argv ){
    //Creamos el nodo principal
    ros::init(argc, argv, "nodo_servidor");
    ros::NodeHandle nh;
    //Creamos el servidor
    servidor =new actionlib::SimpleActionServer<Prueba::MensajeAccionAction>
(nh, "Servidor_accion",cbaccion ,false);
    servidor.start();
    ROS_INFO("Servidor_accion iniciado");
    ros::spin();
```

```
    return 0;
}
```

Servidor con clases

```
#include <ros/ros.h>
#include <actionlib/server/simple_action_server.h>
#include <Prueba/MensajeAccionAction.h>
#include <string>
#include <iostream>
using namespace std;
class Servidor{
    private:
        //Definicion de las varaibles privadas
        ros::NodeHandle nh;
        actionlib::SimpleActionServer<Prueba::MensajeAccionAction> server;
        Prueba::MensajeAccionFeedback feedback;
        Prueba::MensajeAccionResult result;
        ros::Subscriber sub = nh.subscribe("topic", 1000, &Servidor::Callback,
this);
        string accion_name;

        void cbServidor(const Prueba::MensajeAccionGoalConstPtr &goal){
            /*Funcion que se ejecuta cuando se pide una accion al servidor */

            //Declaramos valocidad de ciclos
            ros::Rate rate(1);
            // FIXME: Accion (Ejemplo)
            for (int i = 0; i < goal->numero; i++){
                // Comprobamos si se ha cancelado la accion
                if (server.isPreemptRequested()){
                    ROS_INFO("El cliente ha cancelado la accion");
                    estado = false;
                    break;
                }
                // Enviamos el feedback
                feedback.progreso = 0;
                server.publishFeedback(feedback);
                rate.sleep();
            }

            // Enviamos el resultado
            result.resultado = 0;
            if (estado == true){
                server.setSucceeded(result);
            }
```

```
            else{
                server.setPreempted(reuslt);
            }
        }
    public:
        //Declaramos el contructor
        Servidor(string name): server(nh, name,
boost::bind(&Servidor::cbServidor, this , _1) ,false ),accion_name(name){
            //Inicializamos el servidor
            servidor.start();
        }
}
int main(int argc, char ** argv){
    //Creamos el nodo principal
    ros::init(argc, argv, "Servidor_clase");
    //Creamos el objeto servidor
    Servidor servidor("Servidor_accion");
    ros::spin();
    return 0;
}
```

Mensajes personalizados

1. Primero en el paquete añadimos el paquete actionlib_msgs
2. Creamos la carpeta action
3. Creamos el archivo "MensajeAction.action" y modificamos el archivo con el siguiente contenido:

#Goal

#Result

#Feedback

4. Modificamos el CMakeList.txt:
    1) Descomentamos add_action_files() y añadimos el nombre de los archivos que queremos compilar
    2) Descomentamos el generate_messages()
    3) Descomentamos el catkin_package()

5. Compilamos el paquete con catkin_make