Worksheet 2 – ROS Communication

This script covers the communication methodology between ROS nodes (processes), from custom message creation to their employment on synchronous, through topics, and asynchronous, by means of services, information exchange.

Message Creation

The messages' content is defined on a text file, .msg, describing each field edifying a message on separate lines through their type and name:

robot_id.msg

```
Header header
int8 id
string model
```

There is a multitude of field types at your disposal:

bool, *int8*, *uint8*, *int16*, *uint16*, *int32*, *uint32*, *int64*, *uint64*, *float32*, *float64*,

string, *time*, *duration*, *variable-length array[]*, *fixed-length array[C]*,

Including the abovementioned *ROS::Header*, which includes a *time stamp* and a *string frame_id*, frequent to other ROS messages, e.g., std_msgs, geometry_msgs or visualization_mgs, placing a message chronologically and spatially, respectively. For more information refer to http://wiki.ros.org/msg.

Let's start by creating a package that shall contain the definitions necessary to this class's work.

```
$ cd ~/trsa_ws/src
$ catkin_create_pkg t2_package std_msgs rospy roscpp
```

The package named t2_package is thus created and its dependency on foreign libraries readily declared. Those libraries include the aforesaid standard messages, *std_msgs*, and the ROS interface libraries with C++ and Python, respectively, *roscpp* and *rospy*.

```
$ roscd t2_package
```

The *catkin_create_pkg* script is in charge with making the package's directory where the package's properties, under the *package.xml* file (http://wiki.ros.org/catkin/package.xml), and CMake build definitions, on *CMakeLists.txt* (http://wiki.ros.org/catkin/CMakeLists.txt), are included.

Hence, to form a new custom message, start by creating the previous example's *robot_id.msg* file on a dedicated *msg* folder:

```
$ mkdir msg
$ nano msg/robot_id.msg
```

This opens a file in the nano editor, please add the msg file already defined.

```
Header header
int8 id
string model
```

To ensure that the proper C++, Python and other languages' source code files are generated from this message file, the package's configuration has to be changed.

- Open package.xml and uncomment two lines, as they specify the dependency on those packages at build and run time, respectively: message_generation e message_runtime.

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

- On CMakeLists.txt file it should also be added that there is a dependency on message_generation on the message's generation during the find_package() call, as well as ensuring the runtime dependency, message_runtime, is being exported catkin_package() call:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
...
catkin_package(
 CATKIN_DEPENDS message_runtime ...
 )
```

- Finally, make sure that both CMake knows it has to reconfigure the package's project after you add any message files, adding entries for those files on add_message_files()'s call, and generate_messages() is called in the end.

```
add_message_files(
  FILES
  robot_id.msg
)
...
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

To know for sure that everything went well on *robot_id*'s message creation and that ROS recognizes it, even before compiling the package, try:

```
$ rosmsg show t2_package/robot_id
```

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
int8 id
string model
```

Or, omitting the package it belongs to:

```
$ rosmsg show robot_id
```

```
[t2_package/robot_id]:
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
int8 id
string model
```

For more details on the rosmsg command see: http://wiki.ros.org/rosmsg.

Publishing a Topic

This section is dedicated to get a node publishing a determined topic, with a specific message's format. Start by defining a source code file, *publisher.cpp*, under a dedicated *src* folder:

```
$ roscd t2_package
$ nano src/publisher.cpp
```

On this file enter the following code:

```
#include "ros/ros.h"
#include "t2_package/robot_id.h"

unsigned int id = 106;
std::string model = "TRSABot";

int main(int argc, char **argv)
{
  ros::init( argc, argv, "t2_publisher_node");
  ros::NodeHandle nh;

  ros::Publisher pub = nh.advertise<t2_package::robot_id>("t2_robot_id_topic", 1000);

  ros::Rate loop_rate(10);
  t2_package::robot_id msg;

  while (ros::ok())
  {
    msg.header.stamp = ros::Time::now();
    msg.header.frame_id = "/base_link";
    msg.id = id;
    msg.model = model;

    ROS_INFO("Published robot_id message : %d %s", msg.id, msg.model.c_str());
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }

  return 0;
}
```

Let's now break the code into manageable chunks and go over the most important ones. First to publish a topic we need to advertise our topic in our ROS network. Hence, we should declare a new Publisher that will advertise a topic named t2_robot_id_topic of the t2_package::robot:id type.

```
  ros::Publisher pub = nh.advertise<t2_package::robot_id>("t2_robot_id_topic", 1000);
```

Then, we should declare a message of the t2_package::robot_id type and fill it with the desired information we want to pass to another node. In this case we just want to pass a timestamped message with an id and model.

```
   t2_package::robot_id msg;
  msg.header.stamp = ros::Time::now();
  msg.header.frame_id = "/base_link";
  msg.id = id;
  msg.model = model;
```

Finally, we call the publish function and pass to it the already filled message to send it to the ROS network.

```
pub.publish(msg);
```

Subscribing a Topic

To have a different node subscribe to that same topic and receive the previous node's messages, there is a need to create a new source file, *subscriber.cpp*, that simultaneously listens to what is being published from the other end. Create such a file under the package's *src* folder, and paste the following:

```
#include "ros/ros.h"
#include "t2_package/robot_id.h"

void callback(const t2_package::robot_id::ConstPtr& msg)
{
  ROS_INFO("Received Message from Robot %u model %s", msg->id, msg->model.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "t2_subscriber_node");

  ros::NodeHandle nh;

  ros::Subscriber sub = nh.subscribe("t2_topic", 1000, callback);

  ros::spin();

  return 0;
}
```

Like we did in the previous example, here we need to create a subscriber to be able to retrieve from the network the topic of our choosing

```
ros::Subscriber sub = nh.subscribe("t2_topic", 1000, callback);
```

This entails that we should also create a callback function that will be called when a message is received.

```
void callback(const t2_package::robot_id::ConstPtr& msg)
{
  ROS_INFO("Received Message from Robot %u model %s", msg->id, msg->model.c_str());
}
```

In this very simple case we just want to print to the screen using the ROS_INFO function the id, and model field of the message received.

Creating the Publisher and Subscriber Nodes

Now to create the executable for publisher and subscriber nodes, to be located at the package's *devel* space, some changes have to be made to the package's *CMakeLists.txt* file:

```
include_directories(include ${catkin_INCLUDE_DIRS})

add_executable(t2_publisher_node src/publisher.cpp)
target_link_libraries(t2_publisher_node ${catkin_LIBRARIES})
add_dependencies(t2_publisher_node t2_package_generate_messages_cpp)

add_executable(t2_subscriber_node src/subscriber.cpp)
target_link_libraries(t2_subscriber_node ${catkin_LIBRARIES})
add_dependencies(t2_subscriber_node t2_package_generate_messages_cpp)
```

Note that calling *add_dependencies(t2_publisher_node t2_package_generate_messages_cpp)* defines the target's dependencies, *t2_publisher_node*'s, on message generation output targets, *t2_package_generate_messages_cpp*, so message headers are duly generated before being used.

Finally, run *catkin_make*:

```
$ cd ~/trsa_ws
$ catkin_make
```

Running the Publisher and Subscriber Nodes

Now, to see everything working, the publisher node recursively sending messages and the subscriber node receiving them, lets stat by getting the publisher running:

```
$ rosrun t2_package t2_publisher_node
```

```
[INFO][…]: Published robot_id message : 106 TRSABot
```

The node has an output telling the message "Published robot id message : id model" has just been broadcasted. To actually sense the actual message being transmitted, try:

```
$ rostopic echo /t2_robot_id_topic
```

```
header:
  seq: …
  stamp:
    secs: …
    nsecs: …
  frame_id: /base_link
id: 106
model: TRSABot
```

The *rostopic* command is the corner tool to debug the information exchange between nodes through ROS topics. For more detailed information on its use: http://wiki.ros.org/rostopic.

Making sure the message is being passed over ROS environment, allows the possibility for subscriber nodes to receive those messages. So run the *t2_subscriber_node*:

```
$ rosrun t2_package t2_subscriber_node
```

The subscriber node has just been started, but somehow it seems has though nothing has been actually received. Since we know the publisher to be transmitting correctly, something must be wrong with the subscriber itself. To detect what the problem may be, try using the *rosnode info* command on both nodes:

```
$ rosnode info /t2_publisher_node
```

```
Node [t2_publisher_node]
Publications:
 * /t2_robot_id_topic [t2_package/robot_id]
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /t2_publisher_node/set_logger_level
 * /t2_publisher_node/get_loggers


contacting node http://...
Pid: …
Connections:
 * topic: /rosout
    * direction: outbound
    * transport: TCPROS
```

```
$ rosnode info /t2_subscriber_node
```

```
Node [t2_subscriber_node]
Publications:
 * /rosout [rosgraph_msgs/Log]

Subscriptions:
 * /t2_topic [t2_package/robot_id]

Services:
 * /t2_subscriber_node/set_logger_level
 * /t2_subscriber_node/get_loggers


contacting node http://...
Pid: …
Connections:
 * topic: /rosout
    * direction: outbound
    * transport: TCPROS
```

Here you will find the topic being published is named "t2_robot_id_topic" whereas the topic you find the subscriber node listening to is named "t2_topic". This inconsistency was deliberately promoted in the subscriber's source code to introduce some of these debug tools. On further use of the *rosnode* command see http://wiki.ros.org/rosnode.

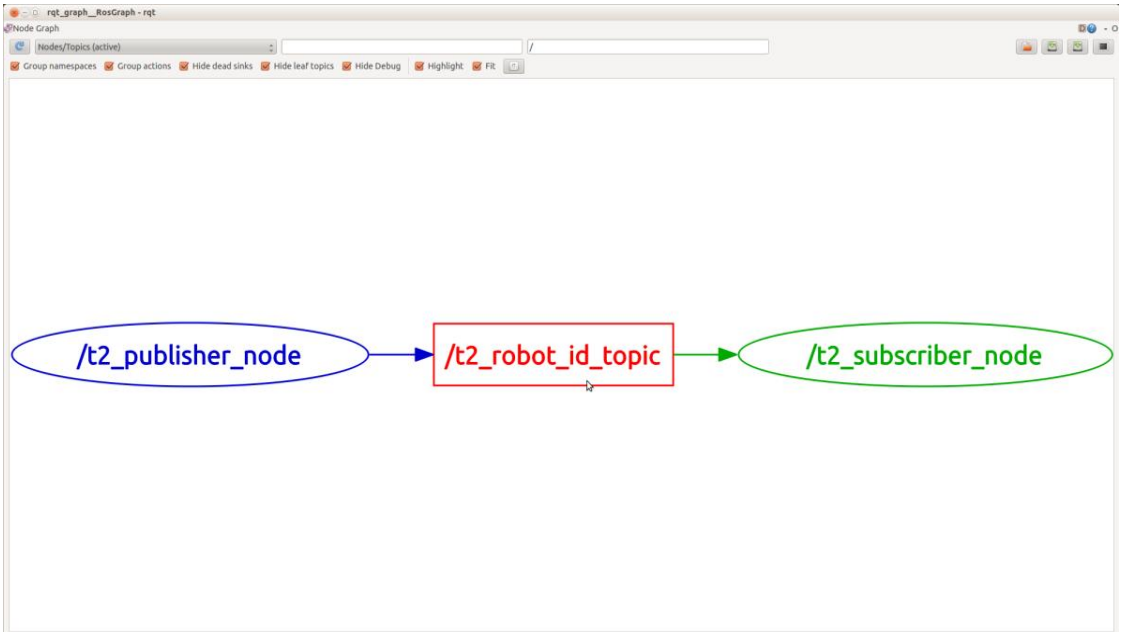
To overcome this problem you could go back to the source code and change the name of the topic, to match the publisher's topic name to the one on the subscriber, or vice-versa, or simply remap the argument in question on either node (http://wiki.ros.org/Remapping%20Arguments). So, stop the subscriber node, and rerun it like so:

```
$ rosrun t2_package t2_subscriber_node t2_topic:=t2_robot_id_topic
```

```
[INFO][…]: Received Message from Robot 106 model TRSABot
```

You can see now the subscriber node's output indicating that messages have been received. If you now run *rqt_graph*, you will see the topic "/t2_robot_id_topic" linking both nodes, whereas without the remapping, the nodes would appear disconnected.

```
$ rosrun rqt_graph rqt_graph
```



Service Creation

The service definition is very similar to creating a message. Normally, the service is composed just like a message, but on a .srv file, where the service request and response are defined. Create a *srv* directory on the package, and define the service as follows:

Set_Robot_Model.srv

```
string model
---
t2_package/robot_id robotID
```

The service node will then receive a string as the argument to request to change a robot's model, and the response is sent back with the updated robot information in the form of the above custom message *robot_id*.

```
$ mkdir srv
$ nano srv/Set_Robot_Model.srv
```

As for the messages, the dependencies have to be also referred on the *CMakeLists.txt* file:

```
add_service_files(
  FILES
  Set_Robot_Model.srv
)
```

Then, using the analogous command for ROS services: *rossrv*, you can make sure the service has been correctly formed:

```
$ rossrv show t2_package/Set_Robot_Model
```

```
string model
---
t2_package/robot_id RobotID
  std_msgs/Header header
    uint32 seq
    time stamp
    string frame_id
  int8 id
  string model
```

Setting up the Service Node

On this step we will change the current publisher node to provide the service, receiving the new model reference, on the request message, and then, besides returning the updated robot_id message to the service's client, update the robot_id topic message being already published.

Edit the *publisher.cpp* source file to include the service's generated header file

```
#include "t2_package/Set_Robot_Model.h"
```

add the service's callback routine, *service_callback*, and advertise the service:

```
bool service_callback(t2_package::Set_Robot_Model::Request  &req,
        t2_package::Set_Robot_Model::Response &res)
{
  model = req.model;
  ROS_INFO("Set_Robot_Model call: %s >> %s", model.c_str(), req.model.c_str());
  res.robotID.id = id;
  res.robotID.model = model;
  return true;
}
```

```
  ros::ServiceServer service = nh.advertiseService("Set_Robot_Model", service_callback);
```

In the end, your newly *publisher.cpp* file should look like this:

```
#include "ros/ros.h"
#include "t2_package/robot_id.h"
#include "t2_package/Set_Robot_Model.h"
unsigned int id = 106;
std::string model = "TRSABot";
bool service_callback(t2_package::Set_Robot_Model::Request  &req,
        t2_package::Set_Robot_Model::Response &res)
{
  model = req.model;
  ROS_INFO("Set_Robot_Model call: %s >> %s", model.c_str(), req.model.c_str());
  res.robotID.id = id;
  res.robotID.model = model;
  return true;
}
int main(int argc, char **argv)
{
  ros::init(argc, argv, "t2_publisher_node");
  ros::NodeHandle nh;
  ros::Publisher pub = nh.advertise<t2_package::robot_id>("t2_robot_id_topic", 1000);
  ros::ServiceServer service = nh.advertiseService("Set_Robot_Model", service_callback);
  ros::Rate loop_rate(10);

  while (ros::ok())
  {
    t2_package::robot_id msg;
    msg.header.stamp = ros::Time::now();
    msg.header.frame_id = "/base_link";
    msg.id = id;
    msg.model = model;
    ROS_INFO("Published robot_id message : %d %s", msg.id, msg.model.c_str());
    pub.publish(msg);
    ros::spinOnce();
    loop_rate.sleep();
  }
  return 0;
}
```

Then run *catkin_make*:

```
$ cd ~/trsa_ws
$ catkin_make
```

Running the Service provider Publisher Node

Firstly, stop your current publisher node, if still running, and then rerun it now its updated version:

```
$ rosrun t2_package t2_publisher_node
```

```
[INFO][…]: Published robot_id message : 106 TRSABot
```

Though it seems working the same way it will permit for an outsider caller to change the published *robot_id* message's model string through a call of its service *Set_Robot_Model*. To verify it is really being advertised you can use the *rosnode* command as before, and check the node's service list:

```
$ rosnode info /t2_publisher_node
```

```
Node [t2_publisher_node]
Publications:
 * /t2_robot_id_topic [t2_package/robot_id]
 * /rosout [rosgraph_msgs/Log]

Subscriptions: None

Services:
 * /Set_Robot_Model
 * /t2_publisher_node/set_logger_level
 * /t2_publisher_node/get_loggers


contacting node http://...
Pid: …
Connections:
 * topic: /rosout
    * direction: outbound
    * transport: TCPROS
```

Or use the *rosservice* command (http://wiki.ros.org/rosservice) just so:

```
$ rosservice info /Set_Robot_Model
```

```
Node: /t2_publisher_node
URI: rosrpc://…
Type: t2_package/Set_Robot_Model
Args: model
```

To see it in action you can promptly call the service directly from the *rosservice call* command. Try changing the published robot_id message's model string typing the following command.

```
$ rosservice call /Set_Robot_Model TurtleBot
```

```
robotID:
  header:
    seq: 0
    stamp:
      secs: …
      nsecs: …
    frame_id: /base_link
  id: 106
  model: TurtleBot
```

While on publisher's and subscriber's window you will see something like:

```
[INFO][…]: Published robot_id message : 106 TRSABot
[INFO][…]: Published robot_id message : 106 TRSABot
[INFO][…]: Published robot_id message : 106 TRSABot
[INFO][…]: Set_Robot_Model call: TRSABot >> TurtleBot
[INFO][…]: Published robot_id message : 106 TurtleBot
[INFO][…]: Published robot_id message : 106 TurtleBot
[INFO][…]: Published robot_id message : 106 TurtleBot
```

```
[INFO][…]: Received Message from Robot 106 model TRSABot
[INFO][…]: Received Message from Robot 106 model TRSABot
[INFO][…]: Received Message from Robot 106 model TRSABot
[INFO][…]: Received Message from Robot 106 model TurtleBot
[INFO][…]: Received Message from Robot 106 model TurtleBot
[INFO][…]: Received Message from Robot 106 model TurtleBot
```

Creating a Service Client Node

If another node is to request such a service, some arrangements have to be made specifically. In this exercise, create another source file (under the *src* folder, of course) named *client.cpp*. On that file submit the following code:

```
#include "ros/ros.h"
#include "t2_package/Set_Robot_Model.h"

int main(int argc, char **argv)
{
  ros::init(argc, argv, "t2_client_node");
  if (argc != 2)
  {
    ROS_INFO("usage: Set_Robot_Model model");
    return 1;
  }

  ros::NodeHandle nh;

  ros::ServiceClient client = nh.serviceClient<t2_package::Set_Robot_Model>("Set_Robot_Model");

  t2_package::Set_Robot_Model srv;
  srv.request.model = argv[1];
  if (client.call(srv))
  {
    ROS_INFO("New Robot ID Model: %u : %s", srv.response.robotID.id, srv.response.robotID.model.c_str());
  }
  else
  {
    ROS_ERROR("Failed to call service Set_Robot_Model");
    return 1;
  }

  return 0;
}
```

Yet again, do not forget to add the corresponding build calls for this source's node to the *CMakeLists.txt* file:

```
add_executable(t2_client_node src/client.cpp)
target_link_libraries(t2_client_node ${catkin_LIBRARIES})
add_dependencies(t2_client_node t2_package_gencpp)
```

Finally, *catkin_make*:

```
$ cd ~/trsa_ws
$ catkin_make
```

Running the Client Node

To run the recently created client node, like in *rosservice call* command, the argument model has to be submitted also:

```
$ rosrun t2_package t2_client_node PR2
```

```
[INFO][…]: New Robot ID Model: 106 : PR2
```

Meanwhile on publisher's and subscriber's window:

```
[INFO][…]: Published robot_id message : 106 TurtleBot
[INFO][…]: Published robot_id message : 106 TurtleBot
[INFO][…]: Published robot_id message : 106 TurtleBot
[INFO][…]: Set_Robot_Model call: TurtleBot >> PR2
[INFO][…]: Published robot_id message : 106 PR2
[INFO][…]: Published robot_id message : 106 PR2
[INFO][…]: Published robot_id message : 106 PR2
```

```
[INFO][…]: Received Message from Robot 106 model TurtleBot
[INFO][…]: Received Message from Robot 106 model TurtleBot
[INFO][…]: Received Message from Robot 106 model TurtleBot
[INFO][…]: Received Message from Robot 106 model PR2
[INFO][…]: Received Message from Robot 106 model PR2
[INFO][…]: Received Message from Robot 106 model PR2
```

Worksheet 2 – ROS Communication

PART II: Rock-Paper-Scissors Game

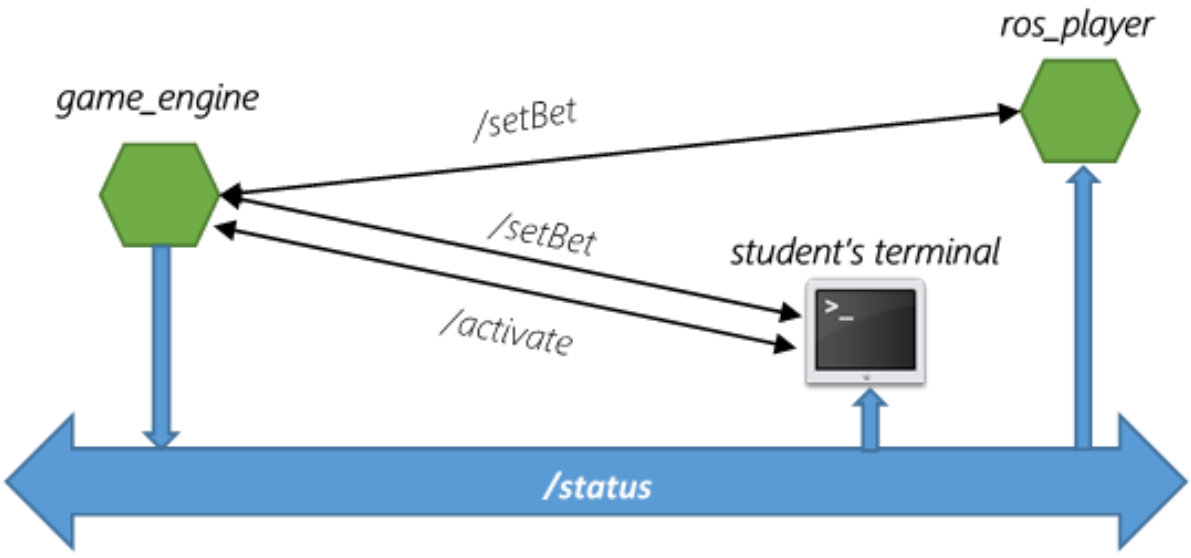
In order to consolidate the concepts about communication between ROS nodes, we're going to implement a version of the rock-paper-scissors game, adapted here for the ROS framework. This game requires that two players make a bet from a range of three options: rock, paper or scissor. The bets are evaluated from as follows:

- Rock defeats scissor by breaking it;
- Scissor defeats paper by cutting it;
- Paper defeats rock by wrapping it.

The player that wins the round adds one point to his(er) current score. In the case of defeat or tie, neither player sums points. The player that yields a higher score in a total of 5 bet rounds is the winner.

The game engine will be implement by a ROS node, labelled as "game_engine", which evaluates the registered bets made by the two players and, publishes messages with the game status on topic "/status" and the winner of the bet round (if any). The bets are registered through a service call on topic "/setBet", exposed by the "game_engine" node.

The student will dispute this game against a ROS node, named as "ros_player", and programmed to play autonomously. To initiate the game, the student must call the service advertised on topic "/activate". The desired system architecture is depicted as follows:



The first step begins by defining the contents of the aforementioned custom service and message. Therefore, inside a package named "t2_package" (you should know how to create a package by now), start by creating the "msg" and "srv" folders to place your custom messages and services. Then create the service SetBet.srv with the following content.

```
uint8 player_id
uint8 bet
---
bool result
```

Create the service **Activate.srv**:

```
bool active
---
bool result
```

Create the message **GameStatus.msg:**

```
# Constant Definition for game status
uint8 NEW_BET_SESSION  = 0
uint8 BET_SESSION_RESULT = 1
uint8 GAME_FINISHED  = 2

uint8 status
uint8[2] score       # score[0] = ROS Player;  score[1] = Human Player
int8 winner          # -1 = Not defined (tie); 0 = ROS Player; 1 = Human Player
```

Do not forget to make the necessary changes in the file CMakeLists.txt and package.xml in order to successfully compile and generate the C++ code of the custom message and service that we're going to use on our own ROS nodes.

Let's program now the "ros_player" node. Create a "player_node.cpp" file in "src" folder of package "t2_package" with the following code:

```
#include <ros/ros.h>
#include "t2_package/SetBet.h"
#include "t2_package/GameStatus.h"
#include <random>
#include <functional>

// Global vars (a necessary evil for now...)
t2_package::SetBet service_msg;
std::function<int()> generate_bet;

// TODO: Add here others global variables...

// TODO: Add here the callback function that will be executed upon receiving a GameStatus message

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

    // TODO: Add here the service client to register bets
    // TODO: Add here the subscriber for GameStatus messages

   // Initialise random generator to bet
   std::default_random_engine generator;
   std::uniform_int_distribution<int> distribution(0,2);
   generate_bet = std::bind ( distribution, generator );

   service_msg.request.player_id = 0;

    ros::spin();
    return 0;
}
```

Now it's time to implement the functionality of registering bets by creating a ros::ServiceClient object. This object is used to call ROS services, and in our case, we'll use it to call the service advertised on topic "/setBet" by the "game_engine" node. Now, make the following additions on the commented sections of the "player_node.cpp" file (i.e., comments that begin with the word "TODO"):

```
// HINT: this is a global variable
ros::ServiceClient bet_client;
```

```
// HINT:  the initialisation of this ServiceClient object is done in the main function
bet_client = node.serviceClient<t2_package::SetBet>("/setBet");
```

Do

```
// HINT: Callback to be called every time that a new GameStatus message is received by this node

void statusCIbk( const t2_ package::GameStatus& msg )
{
	switch( msg.status )
	{
		case t2_package::GameStatus::GAME_FINISHED:
			if ( msg.winner == 0 )
			ROS_INFO("I am the WINNER !!! :)");
			break;

		case t2_package::GameStatus::NEW_BET_SESSION:
			service_msg.request.bet = generate_bet();
			bet_client.call( service_msg );
			ROS_INFO("I sent my bet");
			break;

		case t2_package::GameStatus::BET_SESSION_RESULT:
			ROS_INFO( "My current score is %d" , msg.score[0] );
			break;
	}
}
```

Do the required changes in CMakeLists.txt file in order to compile the new ROS node and naming it as "ros_player" If you're having trouble to do that, please revise Worksheet 1.

Also in order to compile the "ros_player" node you should include in CMakeLists.txt the following flags:

```
SET (CMAKE_C_FLAGS      "-std=c++11")
SET (CMAKE_CXX_FLAGS    "-std=c++11")
```

In contrast of what was said for the case of "ros_player" node, the programming of "game_engine" node will be done by using the object-oriented paradigm. In this manner, we eliminate the use of global variables by accessing internal variables of classes. In our case, all the game's logic will be implemented in the *GameEngine* class.

NOTE: If the student doesn't have any experience in object-oriented programming, it's highly recommended that the student studies this programming paradigm by visiting and exploring the following website (for example) at http://www.learncpp.com/cpp-tutorial/81-welcome-to-object-oriented-programming/ or at other sources, like books, if required. This is the main programming paradigm that will be used during the practical classes.

To ease the student's task of programming this game, the source code files, although incomplete, are provided with this worksheet. The class declaration can be found in GameEngine.h (*header file*), which should be placed inside the "include" folder of package "t2_package". The respective implementation can be found in GameEngine.cpp file, which should be placed at "src" folder. Check and complete the provided source code in order to implement this game. Then, do not forget to edit the CMakeLists.txt file in order to compile the "game_engine" node. To test the code, the student may applies his(er) knowledge regarding what was learnt on the first part of this worksheet. Specifically, a new system's terminal can be used to manually "inject" calls to */activate* and */setBet* services:

```
rosservice call /activate "active: true"

rosservice call /setBet "player_id: 1
bet: <bet_integer_value>"
```

***NOTE**: Write **rosservice call /setBet** in the system's terminal and press TAB key. The terminal should present the respective service parameters for you. Now, you only have to fill the parameter player_id with value **1** and also write the bet value.*

Creation of a ROS Timer

In the first version of our game, the player that bets first will always have to wait for the opponent to place his(er) bet. Let's now make some changes in the code to impose a time limit for the remaining player to bet. When this time limit expires, the current bet session is cancelled if one of the players did not bet. If the two players place a bet within this time limit, the current bet session is immediately closed and evaluated. The results are announced and a new bet session starts over again when one of the players places a new bet.

To implement this new functionality, we'll add a timer (in GameEngine class) that executes, periodically, a class method. As you can see, this is similar to the use of callbacks as a dispatch mechanism for incoming service calls and messages, but in this case the callback is triggered by the internal counter of the timer instead of incoming data from the ROS network.

The ROS framework supports the creation and management of timers through the *createTimer* method of the *ros::NodeHandler* class. Just as in *ros::Subscriber* and *ros::ServiceClient* classes, this method accepts different arguments according to the nature of the callback function that we desire: a class method or a standard C-based function. In our case, the timer callback will be a method of GameEngine class:

```
ros::Timer timer = ros::NodeHandle::createTimer(  ros::Duration(seconds), <callback>, this,
                                                   bool oneshot = false, bool autostart = true );
```

The callback signature that is registered during the timer creation must be the following:

```
void callback( const ros::TimerEvent& );
```

Add to the GameEngine class, the following declaration (GameEngine.h) and implementation (GameEngine.cpp) of the ROS timer and its callback:

```
// HINT: place in GameEngine.h
ros::Timer m_sessionTimer;
void timerClbk ( const ros::TimerEvent& event );
```

```
// HINT: place in GameEngine.cpp
void GameEngine::timerClbk ( const ros::TimerEvent& event )
{
        m_playerBetted = -1;
        m_sessionTimer.stop();
        ROS_WARN( "The bet session expired... Broadcasting a new bet session" );

        t2_package::GameStatus msg;
        msg.status = t2_package::GameStatus::NEW_BET_SESSION;
        m_resultPub.publish( msg );
}
```

Finally, the timer must be initialised inside the class constructor, with a period of 15 seconds and the ***autostart*** disabled:

```
m_sessionTimer = node.createTimer( ros::Duration(15), &GameEngine::timerClbk, this, false, false );
```

Read now the proposed exercises to finalise the implementation of this game.

Final Exercise

1. The student must now define when to start or stop the timer. Do the necessary changes in the *GameEngine* class in order to correctly implement this timer's behaviour (*HINT: The required changes must be done inside the* ***reset*** *and* ***registerBet*** *class methods*);

2. Create a ROS launch file that executes all the required ROS nodes to play this game;

3. Change the bet session's time limit from 15 to 10 seconds;

Compile and test the new version of the game. Then, the updated "t2_package" package must be zipped and named as "T2_<NUMBER1>_<NUMBER2>.zip", where NUMBER is your student's number (i.e., T2_44500_45124.zip). This file must be submitted at Moodle platform, no later than October 6th, 23:55.