ROS BASICS

Sören Schwertfeger / 师泽仁

https://robotics.shanghaitech.edu.cn

With lots of material from last years summer school by Ling Chen icheno@shu.edu.cn (Shanghai University) and with material by Levi Armstrong and Jonathan Meyer SwRI "ROS-Industrial Basic Developer’s Training Class 2016"
Outline

• Reviews

• Learning by Practice:
  • How to customize your own message and service
  • How to publish a topic
  • How to subscribe a topic
  • How to build a service server
  • How to build a client

• Actions

• Roslaunch
Robot Software: Tasks/ Modules/ Programs (ROS: node)

**Support**
- Communication with Micro controller
- Sensor drivers
- Networking
  - With other PCs, other Robots, Operators
- Data storage
  - Store all data for offline processing and simulation and testing
- Monitoring/ Watchdog

**Robotics**
- Control
- Navigation
- Planning
- Sensor data processing
  - e.g. Stereo processing, Image rectification
- Mapping
- Localization
- Object Recognition
- Mission Execution
- Task specific computing, e.g.:
  - View planning, Victim search, Planning for robot arm, …
Software Design

• Modularization:
  • Keep different software components separated
  • 🙄 Keep complexity low
  • 😊 Easily exchange a component (with a different, better algorithm)
  • 😊 Easily exchange multiple components with simulation
  • 😊 Easily exchange components with replay from hard disk instead of live sensor data
  • 😊 Multiple programming teams working on different components easier
  • Need: Clean definition of interfaces or exchange messages!
  • Allows: Multi-Process (vs. Single-Process, Multi-Thread) robot software system
  • Allows: Distributing computation over multiple computers
Review for ROS

• Different components, modules, algorithms run in different processes: **nodes**

• Nodes communicate using **messages** (and **services** …)

• Nodes **publish** and **subscribe** to **messages** by using names ( **topics** )

• **Messages** are often passed around as shared pointers which are
  • “write protected” using the const keyword
  • The shared pointers take the message type as template argument
  • Shared pointers can be accessed like normal pointers
Constant Variables

• Declare variables that do not change (anymore) in the code: `const`

• Works for variables and objects

• Const Objects:
  • Only methods that do not change any variable of the object may be called =>
  • Those methods have to be declared const

• Used for program-correctness

• Especially for multi-threading:
  • Share the data (e.g. image)
  • Make it read only via const
  • => no side-effects between different threads

1. `const int x = 5;` // x may not be changed
2. `int * someValue = &x;` // pointer – compilation error!!
3. `const int * pointy = &x;` // good
4. `*pointy = 8;` // error – pointing to const!
5. `int y = 4;`
6. `pointy = &y;` // from non const to const is always possible!
7. `const int * p2 const = &y;` // pointing to const variable and p2 is also const
8. `p2 = &x;` // error – p2 is const
C++ Templates

- Functions and classes that operate with generic types
- Function or class works on many different data types without rewrite
  - template <typename T> int compare(T v1, T v2);
  - Type of T is determined during compile time => errors during compilation (and not run-time)
  - Any type (type == class) that offers the needed methods & variables can be used
  - Usage: compare<string>( string("string number one"), "hello world" );
    - Explicit declaration: typename T = string
    - typename T can (most often) deducted by the compiler from the argument types

- Class template:
  - template <typename T> class myStuff{
      T v1, v2;
      myStuff(T var1, T var2){ v1 = var2; v2 = var2; }
  };
Template example

```cpp
template <typename Type>
Type max(Type a, Type b) {
    return a > b ? a : b;
}
```

```cpp
#include <iostream>

int main(int, char**) {
    // This will call max <int> (by argument deduction)
    std::cout << max(3, 7) << std::endl;
    // This will call max<double> (by argument deduction)
    std::cout << max(3.0, 7.0) << std::endl;
    // This type is ambiguous, so explicitly instantiate max<double>
    std::cout << max<double>(3, 7.0) << std::endl;
    return 0;
}
```
Shared Pointer

- C++ Standard Library (std): heavily templated part of C++ Standard (many parts used to be in boost library)
- Pointer: address of some data in the heap – in the virtual address space
- Space for data has to be allocated (reserved) with: new
- After usage of data it has to be destroyed to free the memory: delete
- Problem: Data (e.g.) image is shared among different modules/components/threads. Who is the last user – who has to delete the data?
  - Shared pointer: counts the number of users (smart pointers); upon destruction of last user (smart pointer) the object gets destroyed: called “Reference counting”
  - Problem: Shared pointer needs to know the destructor method for the pointer =>
  - Shared pointer is a templated class: Template argument: class type of the object pointed to
  - Shared pointer can also point to const object!
Shared pointer example

```cpp
std::shared_ptr<int> p1(new int(5));
std::shared_ptr<int> p2 = p1; // Both now own the memory.
p1.reset(); // Memory still exists, due to p2.
p2.reset(); // Deletes the memory, since no one else owns the memory.
```

- Earlier, `shared_ptr` used to be in boost
- Excerpt from ROS message of type “String”:
  ```cpp
typedef boost::shared_ptr<::std_msgs::String_<ContainerAllocator>> Ptr;
typedef boost::shared_ptr<::std_msgs::String_<ContainerAllocator> const> ConstPtr;
```
  
  - `typedef`: create another (shorter) name for a certain type
  - Our type: a shared pointer that points to a (complicated) String object

```cpp
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}
```
Messages

- Publisher does not know about subscribers
- Subscribers do not know publishers
- One topic name: many subscribers and many publishers possible, BUT: same message type (determined by the first publisher)!
- List all topics in the current system:
  - rostopic list
  - Other commands: rostopic echo, rostopic hz, rostopic pub , rostopic pub /test std_msgs/String “Hello World!”
Learning by Practice

How to customize your own message and service

- How to publish a topic
- How to subscribe a topic
- How to build a service server
- How to build a client
Creating your own package
Creating your own package

Create a new package
Creating your own package

Create a new package

cd ~/catkin_ws/src
catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
Creating your own package
Creating your own package

Make two folders for messages and services
Creating your own package

Make two folders for messages and services

```
$ roscd beginner_tutorials
$ mkdir msg
$ mkdir srv
```
Creating your own package

Make two folders for messages and services

```bash
$ roscd beginner_tutorials
$ mkdir msg
$ mkdir srv
```

In msg, create a file called AandB.msg, with content:

```plaintext
float32 a
float32 b
```
Creating your own package

Create two folders for messages and services:

```bash
$ roscd beginner_tutorials
$ mkdir msg
$ mkdir srv
```

In `msg`, create a file called `AandB.msg`, with content:

```plaintext
float32 a
float32 b
```

In `srv`, create a file called `AddTwoInts.srv`, with content:

```plaintext
int64 a
int64 b
---
int64 sum
```
Create own message: Text format

• **Types:**
  - int8, int16, int32, int64 (plus uint*)
  - float32, float64
  - string
  - time, duration
  - other msg files
  - variable-length array[] and fixed-length array[C]

• Save in folder “msg”, start with big letter, end with “.msg”
Create own Services

• ROS **service**: send a “message” or command to service provider, wait for reply

• Text format: First message for **request**
  • Separation: three dashes
  • Then message for **response**

• A call to a service blocks

• Either or both data blocks may be empty!

• The response always includes a boolean to indicate success!

```
float32 x
float32 y
float32 theta
string name
---
string name
```
Modify Package.xml and CMakeLists.txt
Modify Package.xml and CMakeLists.txt

Change package.xml.

Open package.xml, and make sure these two lines are in it and uncommented:

```xml
<build_depend>message_generation</build_depend>
<run_depend>message_runtime</run_depend>
```
Modify Package.xml and CMakeLists.txt

Change package.xml.
Open package.xml, and make sure these two lines are in it and uncommented:

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

Add message_generation dependency in CMakeLists.txt.

```cmake
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation)
```
Modify Package.xml and CMakeLists.txt

*Change package.xml.*

Open package.xml, and make sure these two lines are in it and uncommented:

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

*Add message_generation dependency in CMakeLists.txt.*

```cmake
find_package(catkin REQUIRED COMPONENTS roscom rospy std_msgs message_generation)
```

*Uncomment those lines:*

```cmake
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

*Also make sure you export the message runtime dependency.*

```cmake
catkin_package(
  ...
  CATKIN_DEPENDS message_runtime ...
)
Modify Package.xml and CMakeLists.txt
Modify Package.xml and CMakeLists.txt

Change CMakeLists.txt.

Find the following block of code:

```cpp
# add_message_files(
# FILES
# Message1.msg
# Message2.msg
# )
```

Uncomment it by removing the `#` symbols and change to this:
```cpp
add_message_files( FILES AandB.msg )
```
Modify Package.xml and CMakeLists.txt

Change CMakeLists.txt.

Find the following block of code:

```
# add_message_files(
#   FILES
#   Message1.msg
#   Message2.msg
# )
```

Uncomment it by removing the `#` symbols and change to this: `add_message_files(FILEs AandB.msg )`

Remove `#` to uncomment the following lines:

```
# add_service_files(
#   FILES
#   Service1.srv
#   Service2.srv
# )
```

And replace the placeholder Service*.srv files for your service files: `add_service_files(FILEs AddTwoInts.srv)`
Modify Package.xml and CMakeLists.txt

package.xml should look like:

```xml
<?xml version="1.0"?>
<package>
  <name>beginner_tutorials</name>
  <version>0.0.0</version>
  <description>The beginner_tutorials package</description>
  <maintainer email="ling@todo.todo">ling</maintainer>
  <license>TODO</license>

  <build_depend>message_generation</build_depend>
  <buildtool_depend>catkin</buildtool_depend>
  <run_depend>message_runtime</run_depend>

  <buildtool_depend>catkin</buildtool_depend>
  <build_depend>roscpp</build_depend>
  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <run_depend>roscpp</run_depend>
  <run_depend>rospy</run_depend>
  <run_depend>std_msgs</run_depend>

  <export>
  </export>
</package>
```
Modify Package.xml and CMakeLists.txt

CMakeLists.txt could look like:

```cmake
# CMakeLists.txt could look like:
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files( FILES AandB.msg )
add_service_files( FILES AddTwoInts.srv )
generate_messages( DEPENDENCIES std_msgs )
catkin_package( CATKIN_DEPENDS roscpp rospy std_msgs message_runtime )
include_directories( ${catkin_INCLUDE_DIRS} )
```

http://robotics.shanghaitech.edu.cn/static/ROS/
Learning by Practice

- How to customize your own message and service
- How to publish a topic
- How to subscribe a topic
- How to build a service server
- How to build a client
**ROS C++ Client Library**

*roscpp* is a ROS client implementation in C++

Library documentation can be found at:
- [http://docs.ros.org/api/roscpp/html/](http://docs.ros.org/api/roscpp/html/)

ROS header files can be found at: `/opt/ros/hydro/include`
- For example, `/opt/ros/hydro/include/ros/ros.h`

ROS core binaries are located at: `/opt/ros/hydro/bin`
- For example, `/opt/ros/hydro/bin/rosrunc`
ROS Init

A version of ros::init() must be called before using any of the rest of the ROS system

Typical call in the main() function:

```cpp
ros::init(argc, argv, "Node name");
```

Node names must be unique in a running system
**ros::NodeHandle**

- The main access point to communications with the ROS system.
  - Provides public interface to topics, services, parameters, etc.

- **Create a handle to this process’ node (after the call to ros::init())** by declaring:

  ```
  ros::NodeHandle n;
  ```

- The first `NodeHandle` constructed will fully initialize the current node
- The last `NodeHandle` destructed will close down the node
ros::Publisher

Manages an advertisement on a specific topic.

A Publisher is created by calling `NodeHandle::advertise()`
- Registers this topic in the master node

Example for creating a publisher:

```cpp
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("chatter", 1000);
```

- First parameter is the topic name
- Second parameter is the queue size

Once all Publishers for a given topic go out of scope the topic will be unadvertised
**ros::Rate**

A class to help run loops at a desired frequency.

Specify in the constructor the desired rate to run in Hz.

```cpp
ros::Rate loop_rate(10);
```

**ros::Rate::sleep() method**

- Sleeps for any leftover time in a cycle.
- Calculated from the last time sleep, reset, or the constructor was called.
ros::ok()

Call **ros::ok()** to check if the node should continue running

**ros::ok()** will return false if:

- a SIGINT is received (Ctrl-C)
- we have been kicked off the network by another node with the same name
- **ros::shutdown()** has been called by another part of the application.
- all ro::NodeHandles have been destroyed
#include "ros/ros.h"
#include "beginner_tutorials/AandB.h"

int main(int argc, char **argv) {
    ros::init(argc, argv, "talker"); // Initiate new ROS node named "talker"
    ros::NodeHandle n;
    ros::Publisher chatter_pub = n.advertise<beginner_tutorials::AandB>("chatter", 1000);
    ros::Rate loop_rate(10);

    int count = 0;
    while (ros::ok()) // Keep spinning loop until user presses Ctrl+C
    {
        beginner_tutorials::AandB msg;
        msg.a = 1.0;
        msg.b = 2.0;

        ROS_INFO("msg a: %.6f, msg b:%.6f", msg.a, msg.b);

        chatter_pub.publish(msg);
        ros::spinOnce(); // Need to call this function often to allow ROS to process incoming messages
        loop_rate.sleep(); // Sleep for the rest of the cycle, to enforce the loop rate
        count++;
    }
    return 0;
}
CMakeLists.txt

cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files(FILES AandB.msg)
add_service_files(FILES AddTwoInts.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package(CATKIN_DEPENDS roscpp rospy std_msgs message_runtime)
include_directories(${catkin_INCLUDE_DIRS})
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files( FILES AandB.msg )
add_service_files( FILES AddTwoInts.srv )
generate_messages( DEPENDENCIES std_msgs
catkin_package( CATKIN_DEPENDS roscpp rospy std_msgs:message_runtime )
include_directories( ${catkin_INCLUDE_DIRS} )
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
Building Your Nodes

Note the bottom line in the CMakeLists file:

```bash
add_dependencies(talker beginner_tutorials_generate_message_cpp)
```

- This makes sure message headers are generated before being used

After changing the CMakeLists file call catkin_make

```bash
$ cd ~/catkin_ws
$ catkin_make
```
Running the Node From Terminal

Make sure you have sourced your workspace's setup.sh file after calling `catkin_make`:

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

- Can add this line to your `.bashrc` startup file
- Now you can use `rosrun` to run your node:

```bash
$ rosrun beginner_tutorials talker
```
Debugging the Node

```bash
$ cd ~/catkin_ws/build
$ cmake ../src -DCMAKE_BUILD_TYPE=Debug
```

- Tell cmake to create debug symbols
- Find the executable in the devel folder:
  ```bash
  ~/catkin_ws/devel/lib/beginner_tutorials/talker
  ```
- cd ~/catkin_ws/devel/lib/beginner_tutorials
- Debug using gdb:
  ```bash
  gdb ./talker
  ```
Running the Node From Terminal

![Terminal Output]

```plaintext
[ INFO] [1396656488.733257493]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656488.833254496]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656488.933257089]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.033265121]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.13325912]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.23325740]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.333255214]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.433208006]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.53324030]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.633236163]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.733234654]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.833235280]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656489.933220974]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656490.033234669]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656490.133247883]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656490.233246691]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656490.333246143]: msg a: 1.000000, msg b: 2.000000
[ INFO] [1396656490.433246017]: msg a: 1.000000, msg b: 2.000000
```
Examine node talker

$ rostopic list
Examine node talker

```
$ rostopic echo /chatter
```
Learning by Practice

- How to customize your own message and service
- How to publish a topic

- How to subscribe a topic
- How to build a service server
- How to build a client
Create node listener

Add the new source file: listener.cpp, save it

```cpp
#include "ros/ros.h"
#include "beginner_tutorials/AandB.h"

void chatterCallback(const beginner_tutorials::AandB::ConstPtr& msg) {
  ROS_INFO("I heard: msg:a %f, msg:b %f", msg->a, msg->b);
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```
CMakeLists.txt

CMakeLists.txt should look like:

cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files( FILES AandB.msg )
add_service_files( FILES AddTwoInts.srv )
generate_messages( DEPENDENCIES std_msgs )
catkin_package( CATKIN_DEPENDS roscpp rospy std_msgs
  message_runtime )
include_directories( ${catkin_INCLUDE_DIRS} )
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})

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

add_dependencies(talker beginner_tutorials_generate_messages_cpp)
CMakeLists.txt

CMakeLists.txt should look like:

cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files( FILES AandB.msg )
add_service_files( FILES AddTwoInts.srv )
generate_messages( DEPENDENCIES std_msgs )
catkin_package( CATKIN_DEPENDS roscpp rospy std_msgs message_runtime )
include_directories( ${catkin_INCLUDE_DIRS} )
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})

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

add_dependencies(talker beginner_tutorials_generate_messages_cpp)
Building node

After changing the CMakeLists file call catkin_make

```
$ cd ~/catkin_ws
$ catkin_make
```

Or in Eclipse, use short cut “Ctrl + B” to build all packages in the workspace.
Running node listener

Open another terminal, short cut: Ctrl+Shift+T

$ rosrun beginner_tutorials listener
Learning by Practice

- How to customize your own message and service
- How to publish a topic
- How to subscribe a topic
- How to implement a service server
- How to build a client
Service

- Each Service is made up of 2 components:
  - Request: sent by client, received by server
  - Response: generated by server, sent to client

- Call to service blocks in client
  - Code will wait for service call to complete
  - Separate connection for each service call

- Typical Uses:
  - Algorithms: kinematics, perception
  - Closed-Loop Commands: move-to-position, open gripper
Service definition

- In srv/AddTwoInts.srv
- Catkin auto-generates C++ files for us…

```
#Add Integers
int64 a
int64 b
---
int64 sum
```
Go to eclipse, new source file: add_two ints_server.cpp

```cpp
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

bool add(beginner_tutorials::AddTwoInts::Request &req,
          beginner_tutorials::AddTwoInts::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 argc, char **argv)
{
  ros::init(argc, 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;
}
```

http://robotics.shanghaitech.edu.cn/static/ROS/
• **Service Server**
  - Defines associated **Callback Function**
  - Advertises available service *(Name, Data Type)*

```cpp
bool add(AddTwoInts::Request &req, AddTwoInts::Response &res) {
    res.sum = req.a + req.b;
    return true;
}

ros::ServiceServer service = n.advertiseService("add_two_ints", add);
```
CMakeLists.txt should look like:

cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files(FILES AandB.msg)
add_service_files(FILES AddTwoInts.srv)
generate_messages(DEPENDENCIES std_msgs)
catkin_package(CATKIN_DEPENDS roscpp rospy std_msgs message_runtime)
include_directories(${catkin_INCLUDE_DIRS})
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})

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

add_dependencies(talker beginner_tutorials_generate_messages_cpp)
CMakeLists.txt

CMakeLists.txt should look like:

cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files( FILES AandB.msg )
add_service_files( FILES AddTwoInts.srv )
generate_messages( DEPENDENCIES std_msgs )
catkin_package( CATKIN_DEPENDS roscpp rospy std_msgs message_generation include_directories( ${catkin_INCLUDE_DIRS} )
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
Building node

After changing the CMakeLists file call catkin_make

```
$ cd ~/catkin_ws
$ catkin_make
```
Running node add_two_ints_server

Open another terminal, short cut: Ctrl+Shift+T

```
$ rosrun beginner_tutorials add_two_ints_server
```

Open another terminal

```
$ rosservice list
$ rosservice args /add_two_ints
$ rosservice call /add_two_ints 1 2
```
Learning by Practice

- How to customize your own message and service
- How to publish a topic
- How to subscribe a topic
- How to build a service server
- How to build a client
Create node add_two_ints_client

Go to eclipse, new source file: add_two_ints_client.cpp

```cpp
#include "ros/ros.h"
#include "beginner_tutorials/AddTwoInts.h"

int main(int argc, char **argv)
{
  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;
}
```
CMakeLists.txt

CMakeLists.txt should look like:

cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
   roscpp
   rospy
   std_msgs
   message_generation
)
add_message_files( FILES AandB.msg )
add_service_files( FILES AddTwolInts.srv )
generate_messages( DEPENDENCIES std_msgs )
catkin_package( CATKIN_DEPENDS roscpp rospy std_msgs message_runtime )
include_directories( ${catkin_INCLUDE_DIRS} )
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)

add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
CMakeLists.txt

CMakeLists.txt should look like:

cmake_minimum_required(VERSION 2.8.3)
project(beginner_tutorials)
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)
add_message_files( FILES AandB.msg )
add_service_files ( FILES AddTwolInts.srv )
generate_messages( DEPENDENCIES std_msgs )
catkin_package( CATKIN_DEPENDS roscpp rospy std_msgs message_runtime )
include_directories( ${catkin_INCLUDE_DIRS} )
add_executable(talker src/talker.cpp)
target_link_libraries(talker ${catkin_LIBRARIES})
add_executable(listener src/listener.cpp)
target_link_libraries(listener ${catkin_LIBRARIES})
add_executable(add_two_ints_server src/add_two_ints_server.cpp)
target_link_libraries(add_two_ints_server ${catkin_LIBRARIES})
add_executable(add_two_ints_client src/add_two_ints_client.cpp)
target_link_libraries(add_two_ints_client ${catkin_LIBRARIES})
add_dependencies(talker beginner_tutorials_generate_messages_cpp)
Building node

After changing the CMakeLists file call catkin_make

$ cd ~/catkin_ws
$ catkin_make

Or in Eclipse, use short cut “Ctrl + B” to build all packages in the workspace.
Running node add_two_ints_client

Open another terminal, short cut: Ctrl+Shift+T

```
$ rosrun beginner_tutorials add_two_ints_client 1 2
```

```
[INFO] [1397426506.895109360]: Sum: 3
```

```
[INFO] [1397426477.889556278]: Ready to add two ints.
[INFO] [1397426506.894479672]: request: x=1, y=2
[INFO] [1397426506.894544834]: sending back response: [3]
```
Advanced: Actions

Actions manage Long-Running Tasks

Client

My Application

Server

Execute Motion

Robot Motion

Goal

Dest Pos

Feedback

Curr Pos

Cancel

Result

Success
Actions: Details

- Each action is made up of 3 components:
  - Goal, sent by client, received by server
  - Result, generated by server, sent to client
  - Feedback, generated by server

- Non-blocking in client
  - Can monitor feedback or cancel before completion

- Typical Uses:
  - “Long” Tasks: Robot Motion, Path Planning
  - Complex Sequences: Pick Up Box, Sort Widgets
Action definition

• Defines Goal, Feedback and Result data types
  • Any data block may be empty – they always receive handshakes
• Catkin auto-generates C++ files…

```cpp
// Calculate Pi
int32 digits
---
string pi
---
string pi
int32 iter
```
Action Server

- Defines Execute Callback
- Periodically Publish Feedback
- Advertises available action (Name, Data Type)

```c
void executeCB(const CalcPiGoalConstPtr &goal) {
    loop {
        if (as_.isPreemptRequested() || !ros::ok())
            as_.setPreempted();
        as_.publishFeedback(...);
    }
    as_.setSucceeded(result_);
}
SimpleActionServer<CalcPiAction> as_ ("calcPi", &executeCB);
```
Action Client

- Connects to specific Action (Name / Data Type)
- Fills in Goal data
- Initiate Action / Waits for Result

```c++
SimpleActionClient<CalcPiAction> ac("calcPi");
CalcPiGoal goal;
goal.digits = 7;
ac.sendGoal(goal);
ac.waitForResult();
```
# Message vs. Service vs. Action

<table>
<thead>
<tr>
<th>Type</th>
<th>Strengths</th>
<th>Weaknesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message</td>
<td>• Good for most sensors (streaming data)</td>
<td>• Messages can be dropped without knowledge</td>
</tr>
<tr>
<td></td>
<td>• One - to - Many</td>
<td>• Easy to overload system with too many messages</td>
</tr>
<tr>
<td>Service</td>
<td>• Knowledge of missed call</td>
<td>• Blocks until completion</td>
</tr>
<tr>
<td></td>
<td>• Well-defined feedback</td>
<td>• Connection typically re-established for each service call (slows activity)</td>
</tr>
<tr>
<td>Action</td>
<td>• Monitor long-running processes</td>
<td>• Complicated</td>
</tr>
<tr>
<td></td>
<td>• Handshaking (knowledge of missed connection)</td>
<td></td>
</tr>
</tbody>
</table>
roslaunch

- ROS is a Distributed System
  - Often 10s of nodes plus configuration data
  - Painful to start each node manually

- roslaunch is a tool for easily launching multiple ROS nodes, and setting parameters on the Parameter Server.

- It takes in one or more XML configuration files (with the .launch extension) saved in the ‘launch’ folders in packages.

- If roslaunch is used, roscore does not need to be run manually.
Launch Files are like **Startup Scripts**

**Launch File**

Load Perception

Load Robot

Parameter Server

**Nodes**

Camera Driver

Image Processing

Motion Planner

Robot Control
Launch file example

A launch file for launching a node with many parameters

Using `<param />` to set parameters

To run a launch file use:

```bash
$ roslaunch package_name file.launch
```

For the above example:

```bash
$ roslaunch cmd_vel_publisher cmd_vel_publisher.launch
```
Launch file example

A launch file for launching two or more nodes simultaneously
Launch file example

A launch file for launching two or more nodes by **including** another launch file

```xml
<launch>
  <include file="$(find fish_sonar)/launch/fish_sonar.launch" />
  <node pkg="fish_obstacle_avoid" type="fish_obstacle_avoid_node"
        name="fish_obstacle_avoid" output="screen">
    <param name="mode_str" type="string" value="manual" />
    <param name="amplitude" type="int" value="50" />
    <param name="debug" type="bool" value="true" />
    <param name="loop_rate_for_sonar" type="double" value="25" />
    <param name="loop_rate_for_gps_update" type="double" value="4.0" />
    <param name="threshold_obstacle" type="int" value="40" />
  </node>
</launch>
```
Advanced Launchfiles

- `<arg>` — Pass a value into a launch file
- `if= or unless=` — Conditional branching
  - *extremely limited. True/False only (no comparisons).*
- `<group>` — group commands, for if/unless or namespace
- `<remap>` — rename topics/services/etc.

```xml
<launch>
  <arg name="robot" default="sia20"/>
  <arg name="show_rviz" default="true"/>
  <group ns="robot">
    <include file="$(find lesson)/launch/load_$(arg robot)_data.launch"/>
    <remap from="joint_trajectory_action" to="command"/>
  </group>
  <node name="rviz" pkg="rviz" type="rviz" if="$(arg show_rviz)"/>
</launch>
```
Retrieving Parameters in c++ file

• There are two methods to retrieve parameters with NodeHandle:
  – `getParam(key, output_value)`
  – `param(key, output_value, default)` is similar to `getParam()`, but allows to specify a default value
  – `key: "~..."` is in the private namespace...

• Example: in the cpp file

```cpp
ros::NodeHandle n_local("~");
n_local.param("frequency", frequency, 1.0);
n_local.param("Max_constant_V", Max_constant_V, CONSTANT_V);
n_local.param("delta_v", delta_v, 0.05);
```
Try: Launch

• Use launch file to run two nodes with params
  — Run turtlesim and its velocity control

• Solution:

```xml
<launch>
  <!-- run turtlesim -->
  <node pkg="turtlesim" type="turtlesim_node" name="turtlesim"/>
</node>
  <!-- run turtle_teleop_key.launch -->
  <node pkg="turtle_teleop_key" type="turtle_teleop_key_node" name="turtle_teleop_key" output="screen">
    <param name="twist_name" value="/turtle1/cmd_vel"/>
  </node>
</launch>
```
Important ROS tools

- Rviz: show live data, including video, TF, Point Clouds, Maps, Robot Models …
- rosbag: record messages into a (bag-) file! Ability to replay those bagfiles!
- rqt_bag: visualize the contents of a bagfile
- rqt_graph: show in a GUI with which topics nodes are connected
- rqt_console: show debug, warning and error messages – good filters
- rosrun rqt_reconfigure rqt_reconfigure package: re-configure parameters on the fly using a GUI!
- roswtf: see if there are problems with your currently running ROS system
Recourses:

- C++: http://www.cplusplus.com/doc/tutorial/
- https://en.wikipedia.org/wiki/Smart_pointer
Cheat Sheets

Questions?