ROS+GAZEBO

Part 1

July 2012

www.cogniteam.com

COGNITEAM

ROS (Robotic Operating System)

- Software framework for robot software development

- developed in 2007 under the name switchyard by the Stanford Artificial Intelligence Laboratory in support of the STAIR

- Based on graph architecture & is geared toward a Unix-like system

ROS is …

- Peer-to-peer
- Multi-lingual
- Tools-based
- Thin
- Free and Open-Source

Components
The File System
and
ROS Tools

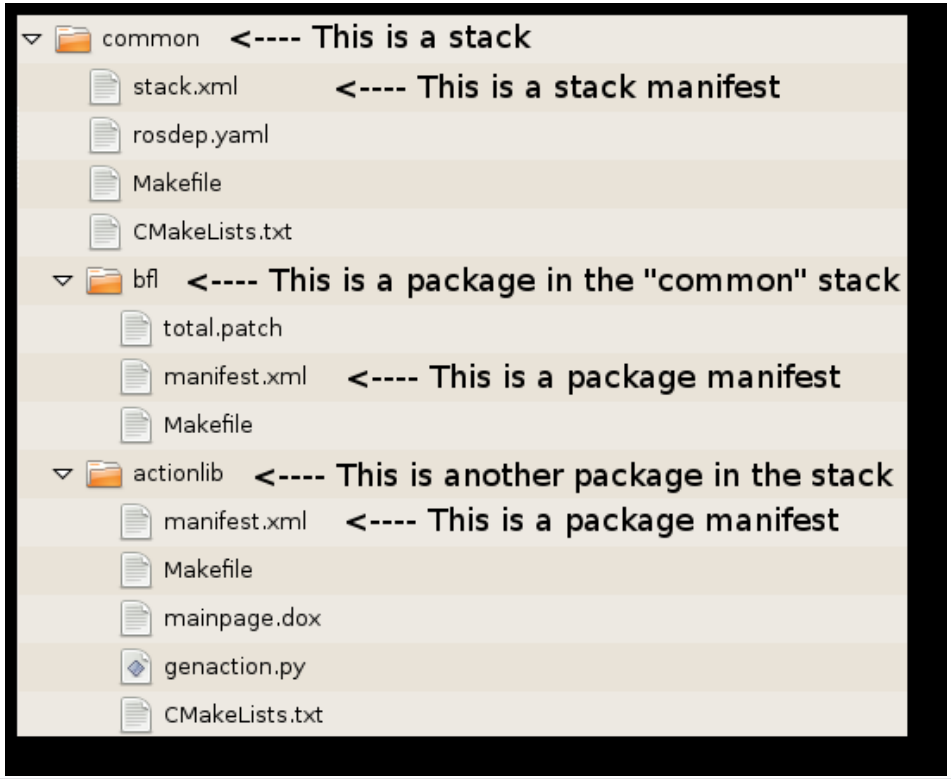
ROS File System

- Packages
  - Lowest level of ROS software organization.
  - Can contain libraries, tools, executable, etc.
- Manifest
  - Define dependencies between *packages*.
- Stacks
  - Collections of *packages*, higher-level library.
- Stack Manifest
  - Like normal *manifests*, but for *stacks*.

ROS File System

When you look at the filesystem, it's easy to tell *packages* and *stacks* apart:

- A package is a directory with a manifest.xml file.
- A stack is a directory with a stack.xml file.



File System tools

- **rospack** find option, returns path to package or stack

```
$ rospack find gazebo
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo
```

- **roscd** allows to change directory (cd) directly to a package or a stack.

```
$ roscd gazebo
/opt/ros/fuerte/stacks/simulator_gazebo/gazebo $
```

- **rosls** (is similar to **ls**)
- Tab completion also works

ROS package

- To create a new package

```
$ roscreate-pkg [package_name]
```

- You can also specify dependencies

```
$ roscreate-pkg [package_name] [depend1] [depend2]
```

- For example, in your home path try

```
$ roscreate-pkg tutorials1 std_msgs rospy roscpp
```

Is your path configured ?

If not, instructions are on the following page.

ROS path configuration

- ROS development should be in your home path
- Add a path to ROS_PACKAGE_PATH by prepending it. Go to the folder you wish to add to the path and type

```
$ export ROS_PACKAGE_PATH=$PWD:$ROS_PACKAGE_PATH
$echo $ROS_PACKAGE_PATH
/home/your_user_name/fuerte_workspace/sandbox:
/opt/ros/fuerte/share:/opt/ros/fuerte/stacks
```

- Make sure your package is now found. Use

```
$ rospack find
```

ROS dependencies

- rosdep is a tool you can use to install system dependencies required by ROS packages.

```
$ rosdep install turtlesim
```

- Dependencies are found in the manifest.xml

```
$ roscd turtlesim
$ cat manifest.xml
```

ROS make

- rosmake is just like the make command

```
$ rosmake tutorials1
```

- Notice that the dependencies are also built in the order they are found in the manifest file

Graph Concepts

- Nodes: A node is an executable that uses ROS to communicate with other nodes.
- Messages: ROS data type used when subscribing or publishing to a topic.
- Topics: Nodes can *publish* messages to a topic as well as *subscribe* to a topic to receive messages.
- Master: Name service for ROS (i.e. helps nodes find each other)
- rosout: ROS equivalent of stdout/stderr
- roscore: Master + rosout + parameter server (parameter server will be introduced later)

Components
ROS Nodes

ROS node

- A node is an executable file in a ROS package
- ROS nodes use a ROS client library to communicate with other nodes.
- Nodes can publish or subscribe to a Topic.
- Nodes can also provide or use a Service.

- Client Libraries
  - allow nodes written in different programming languages to communicate:
    - rospy = python client library
    - roscpp = c++ client library

ROS core

- roscore is the first thing you should run when using ROS.

```
$ roscore
```

Troubleshooting

- Python threading exceptions at this stage can be safely ignored (2.7 documented bug)
- Network errors can be resolved using

```
$ export ROS_HOSTNAME=localhost
$ export ROS_MASTER_URI=http://localhost:11311
```

ROS node

- rosnode displays information about the ROS nodes that are currently running.

- The rosnode list lists these active nodes

```
$ rosnode list
/rosout
```

ROS node

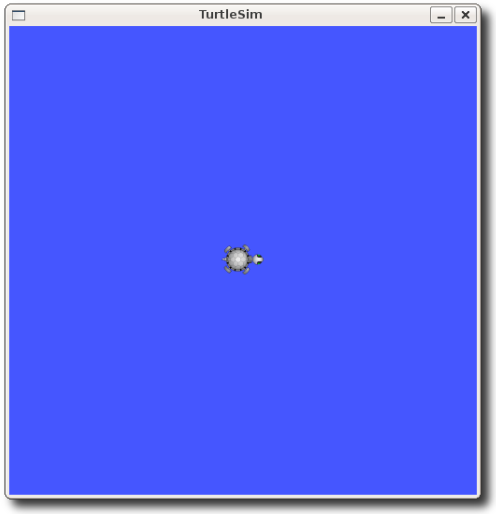
But what is rosout ?

```
$ rosnode info /rosout
--------------------------------------------------------------------------------
Node [/rosout]
Publications:
 * /rosout_agg [rosgraph_msgs/Log]
Subscriptions:
 * /rosout [unknown type]
Services:
 * /rosout/set_logger_level
 * /rosout/get_loggers
contacting node http://laptop:43643/ ...
Pid: 18019
```

ROS run

- rosrun allows you to use the package name to directly run a node within a package

```
$ rosrun [package_name] [node_name]
```




```
$ rosmake turtlesim
$ rosrun turtlesim turtlesim_node
```

- In a **new terminal**

```
$ rosnode list
/rosout
/turtlesim
$ rosnode ping turtlesim
xmlrpc reply from http://localhost:56037/     time=0.
 618935ms
```

Moving the turtle

- **in a new terminal**:

```
$ rosrun turtlesim turtle_teleop_key
```

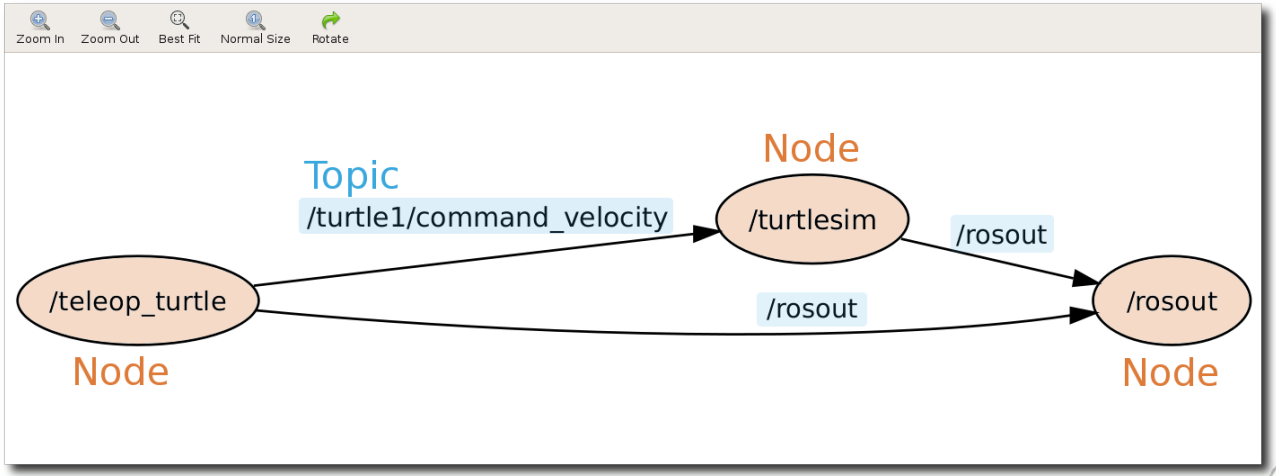
- Now you can move the turtle around using arrow keys

Components
ROS Topics

ROS topic

- The turtlesim_node and the turtle_teleop_key node are communicating with each other over a ROS **Topic**.
  - turtle_teleop_key is **publishing** the key strokes
  - turtlesim **subscribes** to receive the key strokes

- **rxgraph**

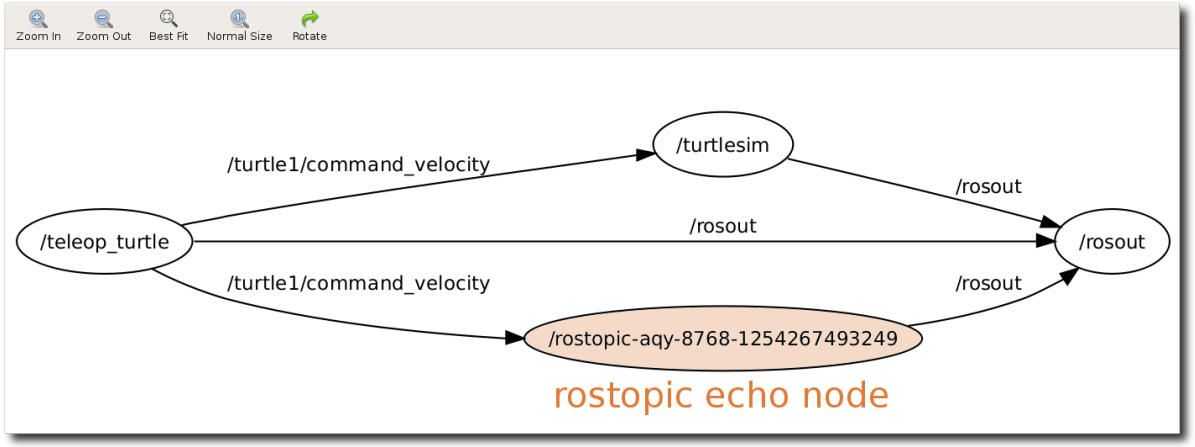


ROS topic

- The rostopic tool allows you to get information about ROS **topics**

```
$ rostopic echo [topic]
$ rostopic echo /turtle1/command_velocity
```

- You need to input values in order to see them
- Open rxgraph to see your echo node



ROS topic, messages

- In order to list of all topics currently subscribed to and published use

```
$ rostopic list -v
```

- Notice the line

```
/turtle1/command_velocity [turtlesim/Velocity] 1 publisher
```

- To get the type of the data transmitted use

```
$ rostopic type /turtle1/command_velocity
turtlesim/Velocity
```

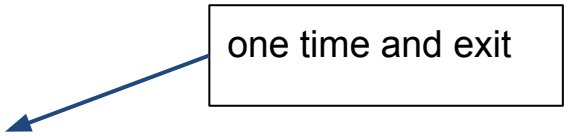
- And to see the ICD API for it

```
$ rosmsg show turtlesim/Velocity
float32 linear
float32 angular
```

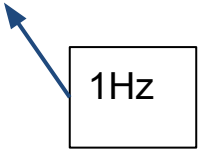
ROS topic, publishing

- rostopic pub publishes data on to a topic currently advertised.

```
$ rostopic pub [topic] [msg_type] [args]
```




```
$ rostopic pub -1 /turtle1/command_velocity
  turtlesim/Velocity  -- 2.0  1.8
$ rostopic pub /turtle1/command_velocity
  turtlesim/Velocity -r 1 -- 2.0  1.8
```



- rostopic can monitor frequencies also

```
$ rostopic hz /turtle1/command_velocity
```

Components
ROS Services

ROS services

Services allow nodes to send a **request** and receive a **response**.

- rosservice list
  - print information about active services
- rosservice call
  - call the service with the provided args
- rosservice type
  - print service type
- rosservice find
  - find services by service type
- rosservice uri
  - print service ROSRPC uri

ROS services

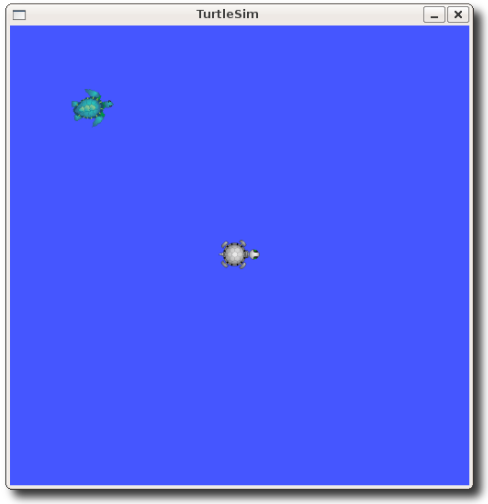
- Much like we did in messages, to get the ICD for the service API

```
$ rosservice list
$ rosservice type [service]
[a type]
$ rossrv show [a type]
float32 x
float32 y
---
string name
```

arguments

return value

Try spawning a second turtle !



- To call the service use for the example above

```
$ rosservice call [service] 2 2
```

Components

ROS ParameterServer

Logging and **Launchers**

ROS parameter server

rosparam allows you to store and manipulate data on the ROS Parameter Server.

- rosparam set — set parameter
- rosparam get — get parameter
- rosparam load — load parameters from file
- rosparam dump — dump parameters to file
- rosparam delete — delete parameter
- rosparam list — list parameter names

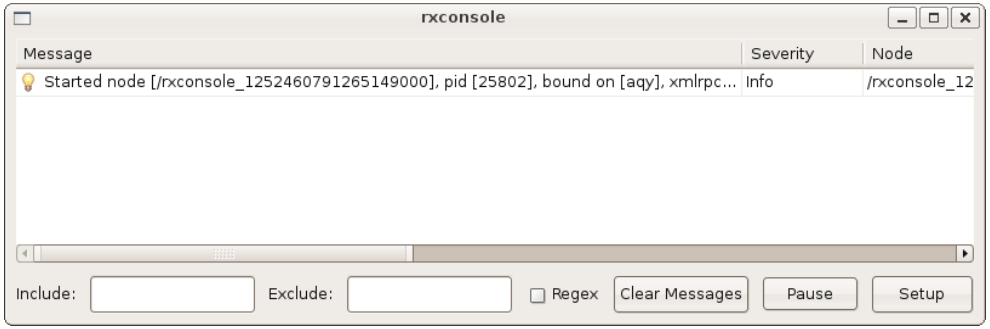
```
$ rosparam set [param_name]
$ rosparam get [param_name]
```

Try to set background_r (remember to "rosservice call clear" after you change the parameter)

ROS logging

- rxconsole attaches to ROS's logging framework to display output from nodes.
- rxloggerlevel allows to change the verbosity level (DEBUG, WARN, INFO, and ERROR) of nodes as they run.

```
$ rxconsole
$ rxloggerlevel
```



ROS launch files

- roslaunch starts nodes as defined in a launch file

```
$ roslaunch [package] [filename.launch]
```

- Launch files are in launch subdir (check path)
  - export ROS_PACKAGE_PATH=$PWD:$ROS_PACKAGE_PATH

[somename.launch]

```
<launch>
  <group ns="turtlesim1"> <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>
  <group ns="turtlesim2"> <node pkg="turtlesim" name="sim" type="turtlesim_node"/>
  </group>
  <node pkg="turtlesim" name="mimic" type="mimic">
    <remap from="input" to="turtlesim1/turtle1"/>
    <remap from="output" to="turtlesim2/turtle1"/>
  </node>
</launch>
```

- To launch tutorials1

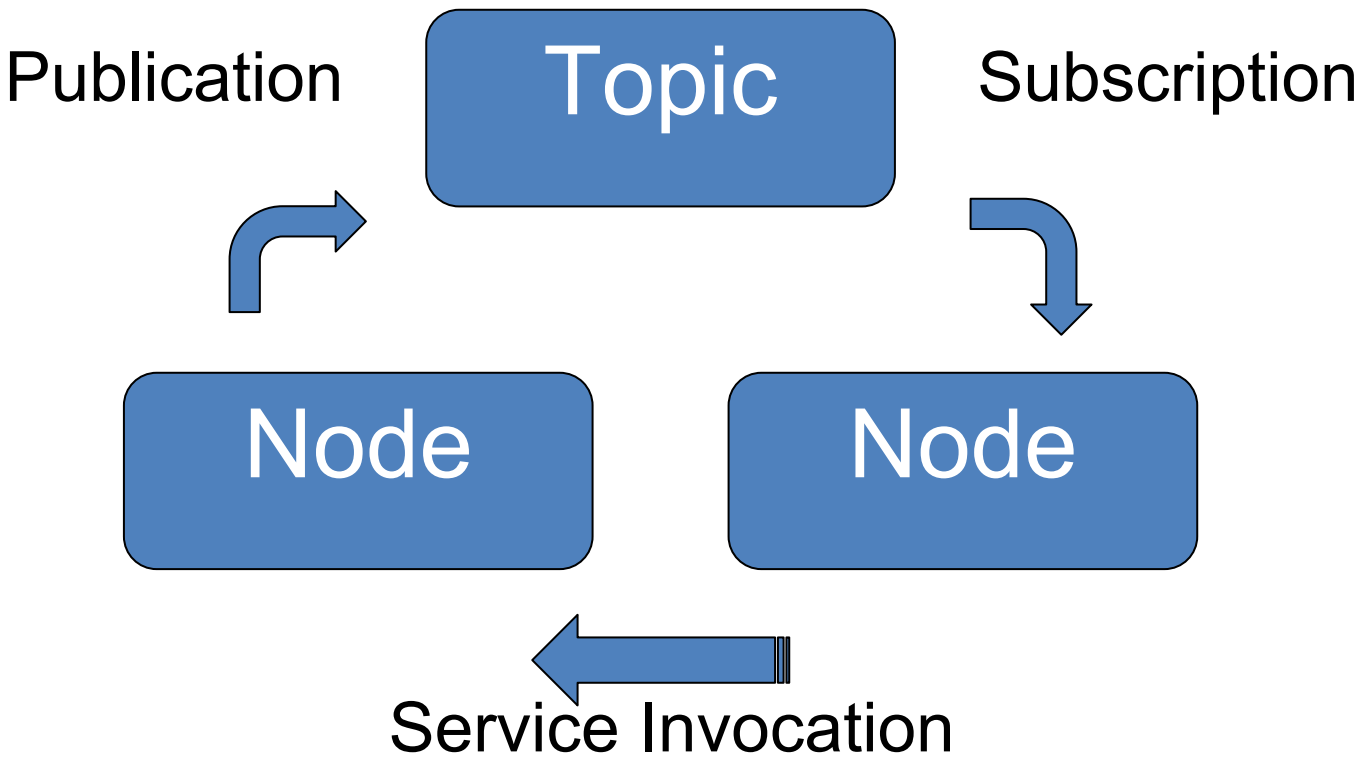
```
$ roslaunch tutorials1 somename.launch
$ rostopic pub /turtlesim1/turtle1/command_velocity
turtlesim/Velocity -r 1 -- 2.0  -1.8
```

Overview (reminder)

- Nodes can publish or subscribe to a Topic.
  - Messages are ROS data type used when subscribing or publishing to a topic.

- Nodes can also provide or use a Service.
  - Services allow nodes to send a **request** and receive a **response**.

ROS messaging



Overview

- Check out from Subversion repository
- Editing and Building with Eclipse IDE
- Creating Messages
  - for Topics
  - for Services
- Code writing and compilation
  - Publisher and Subscriber
  - Service and Client

Check out from Subversion
and
Build in Eclipse IDE

checkout ROBIL

- create somewhere ROS_ROBIL_ROOT

```
$ mkdir ROS_ROBIL_ROOT
$ cd ROS_ROBIL_ROOT
$ export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:$PWD
```

- check-out all ROBIL projects
  (inside of ROS_ROBIL_ROOT)

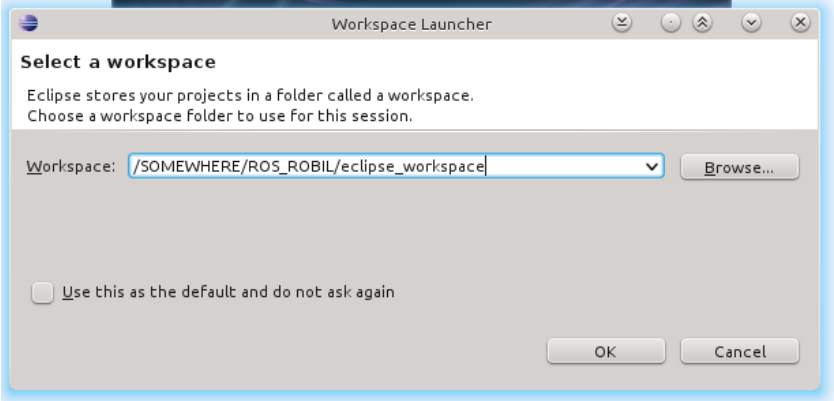
```
$ svn checkout svn+ssh:
//robil@COGNITEAM/var/lib/svn/ROBIL/ros_stack robil

$ rospack profile
```

ECLIPSE

- create eclipse project
  (inside of ROS_ROBIL_ROOT)

```
$ mkdir eclipse_workspace
$ roscd package_name
$ make eclipse-project
```

- open project in Eclipse IDE



ECLIPSE

- Import project into workspace
  - File -> Import...
  - General -> Existing Project into Workspace [*Next*]
  - Select root directory : /SOMEWHERE/ROS_ROBIL
  - Projects : [*Select All*]
  - IMPORTANT : don't check Copy project into workspace
  - [*Finish*]

- Build project
  - Project -> Build Project

Creating **Topic** and **Service** **Messages**

Creating msg and srv

msg and **srv** are simple text files that describe the fields of ROS message or service for code generator.

msg

```
Header header
string sensor_name
int64 pos_x
int64 pos_y
float32 heading
float64 value
```

srv

```
int64 a
int64 b
---
int64 sum
```

request

result

msg and srv types

- Primitives
  - int8, int16, int32, int64
  - float32, float64
  - string
  - time, duration
  - variable-length array[]
  - fixed-length array[*C*]
- Header
  - timestamp and coordinate frame information
- Other msg files
  - geometry_msgs/PoseWithCovariance

msg compilation

- Messages are usually under an mdg folder in your package
- go to your package msg folder or create it

```
$ roscd tutorials1
$ mkdir -p msg
$ echo "int64 num" > msg/MyMessage.msg
```

- Edit message file
- open CMakeList.txt and uncomment

```
#rosbuild_genmsg()
```

- run compilation

```
$ rosmake
```

srv compilation

- go to your package srv folder or create it

```
$ roscd tutorials1
$ mkdir -p srv
```

- In srv make a file called sum.srv that contains

```
int64 a
int64 b
---
int64 sum
```

- Open CMakeList.txt and uncomment

```
#rosbuild_gensrv()
```

run compilation

```
$ rosmake
```

rosmsg

rosmsg provides information about messages

- `rosmsg` show *MSG*
  - Show message description
- `rosmsg users` *MSG*
  - Find files that use message
- `rosmsg package` *PKG*
  - List messages in a package
- `rosmsg packages` *MSG*
  - List packages that contain messages

rossrv

rossrv provides information about services

- `rossrv show` *SRV*
  - Show message description
- `rossrv users` *SRV*
  - Find files that use message
- `rossrv package` *PKG*
  - List messages in a package
- `rossrv packages` *SRV*
  - List packages that contain messages

Code Writing:

Publisher

and

Subscriber

Package creation

- **roscreate-pkg** will create a default Makefile and CMakeLists.txt for your package

```
$roscreate-pkg pkgname depend1 depend2 depend3
```

- in our case :

```
$roscreate-pkg tutorials2 roslib roscpp std_msg
```

- in CMakeList.txt add the following at the bottom:

```
rosbuild_add_executable(talker src/talker.cpp)
rosbuild_add_executable(listener src/listener.cpp)
```

- add to Subversion

```
$svn add tutorials2
```

- create Eclipse progect

```
$cd tutorials2; make eclipse-project
```

Package creation

- Don't commit to SVN eclipse project files

```
$roscd tutorial2
$svn propedit svn:ignore .
.project
.cproject
.pydevproject
[^o][Enter][^x]
```

- Commit package
  (from parent folder of package)

```
$svn commit -m"NewPackage" tutorial2
```

Publisher - Subscriber

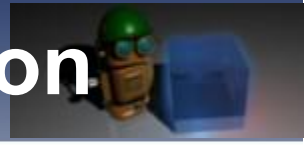
talker.cpp

- init ros and current node (connection point)
- advertise topic "chatter"
- in loop
  - create string message
  - publish message in topic

listener.cpp

- init ros and current node (connection point)
- subscribe to topic "chatter"
- get messages from "chatter"

talker.cpp - Includes , Initialization



```
//includes for ROS
#include "ros/ros.h"
#include "std_msgs/String.h"
int main(int argc, char** argv){

   //initialization of ROS
   ros::init(argc, argv, "talker");

   //initialization of current Node
   ros::NodeHandle n;
   //...
}
```

talker.cpp - Advertise

Tell ROS to publish on a given topic

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

topic name

size of message queue

talker.cpp - Publish

```
//sending frequencies
ros::Rate loop_rate(10);

//publishing
while (ros::ok()){

   std_msgs::String msg;
   msg.data = "Hello, world";
   chatter_pub.publish(msg);

   loop_rate.sleep();
}
```

listener.cpp - Includes, Initialization

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

int main(int argc, char** argv){
   //initialization of ROS
   ros::init(argc, argv, "listener");

   //initialization of current Node
   ros::NodeHandle n;
   //...
}
```

listener.cpp - Callback

```
void chatterCallback(
    const std_msgs::String::ConstPtr& msg)
{

    std::string msg_data = msg->data;
    //Logging to rosout
    ROS_INFO("I heard:[%s]",msg_data.c_str());

}
```

listener.cpp - Subscribe

```
ros::Subscriber sub =
   n.subscribe(
      "chatter",
      1000,
      chatterCallback
   );
```

topic name

message buffer size

callback function name

listener.cpp - main loop

- In order to pumping callbacks activate the main loop using

```
ros::spin();
```

```
ros::spinOne();
```

- Callbacks are blocking and in the same thread

Code Writing:
Service
and
Client

Package creation

- **roscreate-pkg** will create a default Makefile and CMakeLists.txt for your package

```
$roscreate-pkg pkgname depend1 depend2 depend3
```

- in our case :

```
$roscreate-pkg tutorials3 tutorials1
```

- in CMakeList.txt add the following at the bottom:

```
rosbuild_add_executable(summator src/summator_server.cpp)
rosbuild_add_executable(sum_client src/summator_clnt.cpp)
```

- add to Subversion

```
$svn add tutorials3
```

- create Eclipse progect

```
$cd tutorials3; make eclipse-project
```

Package creation

- Don't commit to SVN eclipse project files

```
$roscd tutorial3
$svn propedit svn:ignore .
.project
.cproject
.pydevproject
[^o][Enter][^x]
```

- Commit package
  (from parent folder of package)

```
$svn commit -m"NewPackage" tutorial3
```

Service - Client

summator_server.cpp

- init ros and current node (connection point)
- advertise service "add_two_ints_server"
- wait for new requests from clients

summator_client.cpp

- init ros and current node (connection point)
- connect to service
- send request and wait for answer

summator_server.cpp

```
//INCLUDES and INITIALIZATION
#include "ros/ros.h"
#include "tutorials1/sum.h"

int main(int argc, char** argv){
  //initialization of ROS
  ros::init(argc, argv, "summator");
  //initialization of current Node
  ros::NodeHandle n;
```

summator_server.cpp

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

summator_server.cpp

ROS request to be a server of service

```
//ADVERTISE
ros::ServiceServer service =
   n.advertiseService(
      "add_two_ints", add
   );
```

service name

service callback function

```
ros::spin();
```

wait for requests

summator_client.cpp

//INCLUDES and INITIALIZATION

```
#include "ros/ros.h"
#include "tutorials1/sum.h"
int main(int argc, char** argv){
  //initialization of ROS
  ros::init(argc, argv, "summator_clnt");

  //initialization of current Node
  ros::NodeHandle n;
```

summator_client.cpp

```
//CONNECT TO SERVICE
ros::ServiceClient client =
   n.serviceClient<sum>(
      "add_two_ints"
   );
```

service name

summator_client.cpp

```
//CREATE REQUEST, SEND and GET RESPONSE
sum srv;
srv.request.a = 1;
srv.request.b = 2;
if( client.call(srv) )
   long int res =
      (long int)srv.response.sum;
else
   ROS_ERROR("Failed to call service");
```

input

call

output

Simulation Environment

GAZEBO

Gazebo

- In order to run the simulation

```
$ rosmake gazebo_worlds
$ roslaunch gazebo_worlds empty_world.launch
```

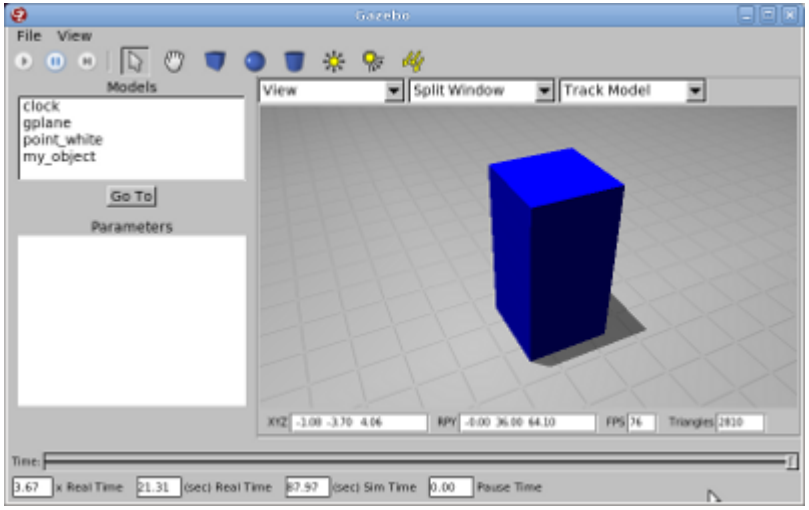
- Objects in the simulation are URDF files
  - http://ros.org/wiki/urdf
  - http://ros.org/wiki/urdf/Tutorials

- To spawn object.ordf at height z=1

```
$ rosrun gazebo spawn_model -file `pwd`/object.urdf -
urdf -z 1 -model my_object
```

object.urdf

```
<robot name="simple_box">
  <link name="my_box">
    <inertial>
      <origin xyz="2 0 0" />
      <mass value="1.0" />
      <inertia  ixx="1.0" ixy="0.0"  ixz="0.0"  iyy="100.0"  iyz="0.0"  izz="1.0"
/>
    </inertial>
    <visual>
      <origin xyz="2 0 0"/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </visual>
    <collision>
      <origin xyz="2 0 0"/>
      <geometry>
        <box size="1 1 2" />
      </geometry>
    </collision>
  </link>
  <gazebo reference="my_box">
    <material>Gazebo/Blue</material>
  </gazebo>
</robot>
```



A Box - Inertia

- A box geometry primitive
  - sizes 1m wide, 1m deep and 2m tall
- The 3x3 rotational inertia matrix is specified with the inertia element. It is symmetrical, can be represented by only 6 elements

  ixx ixy izz

  ixy **iyy iyz**

  ixz iyz **izz**

- In the example $i_{xx}=i_{zz}=1\ kg*m^2$ and $i_{yy}=100\ kg*m^2$
- If unsure what to put, the identity matrix is a good default

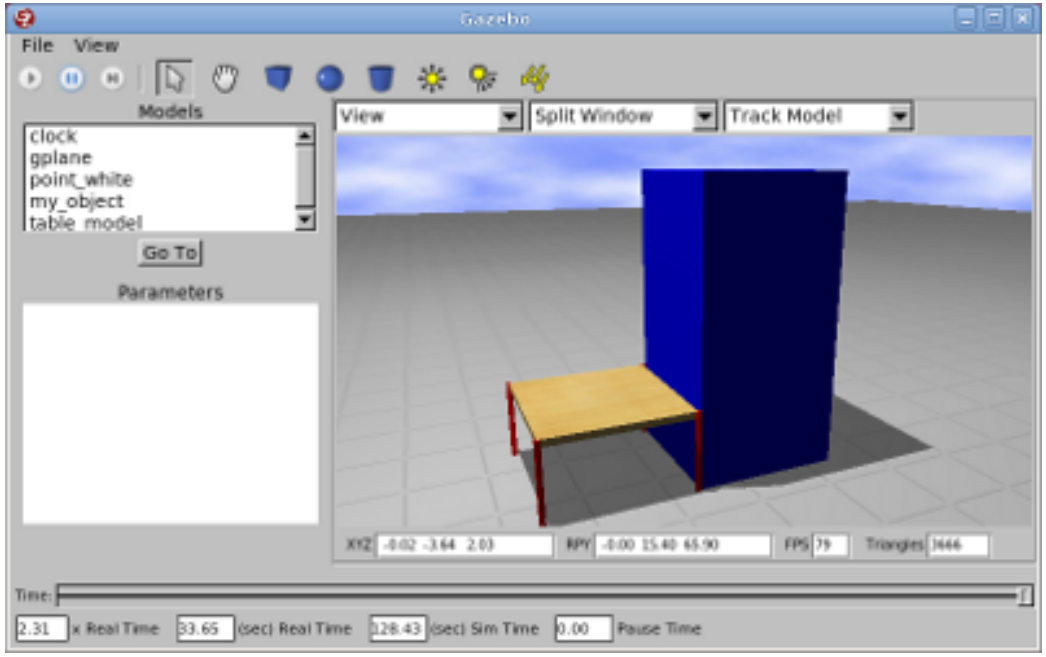
A Box - physical tags

- mu - The friction coefficient
- kp - Stiffness coefficient
- kd - Dampening coefficient
- friction - The physical static friction. For prismatic joints, the units are Newtons. For revolving joints, the units are Newton meters.
- damping - The physical damping value. For prismatic joints, the units are Newton seconds per meter. For revolving joints, Newton meter seconds per radian.

Spawning additional objects

- Several sample objects are provided in gazebo_worlds/objects.
- For example, you can spawn a desk by typing

```
$ roslaunch gazebo_worlds table.launch
```



ROS service and Gazebo

- Spawning a coke from gazebo_worlds/objects

```
$ rosrun gazebo spawn_model -file `pwd`/coke_can.urdf -
 urdf -z 1 -model my_object
```

- Deleting it

```
$ rosservice call gazebo/delete_model '{model_name:
 my_object}'
```

- To get the model state

```
$ rosservice call gazebo/get_model_state '{model_name:
 my_object}'
```

ROS service and Gazebo

- Change position

```
$ rosservice call /gazebo/set_model_state '{model_state: { model_name: my_object, pose: { position: { x: 0, y: 0 ,z: 1 }, orientation: {x: 0, y: 0.491983115673, z: 0, w: 0.870604813099 } }, twist: { linear: {x: 0.0 , y: 0 ,z: 0 } , angular: { x: 0.0 , y: 0 , z: 0.0 } } , reference_frame: world } }'
```

```
$ rosservice call /gazebo/set_model_state '{model_state: { model_name: my_object, pose: { position: { x: 0, y: 0 ,z: 1 }, orientation: {x: 0, y: 0.0, z: 0, w: 0.870604813099 } }, twist: { linear: {x: 0.0 , y: 0 ,z: 0 } , angular: { x: 0.0 , y: 0 , z: 0.0 } } , reference_frame: world } }'
```

- **More on wrenches and efforts can be found at**
  http://ros.org/wiki/simulator_gazebo/Tutorials/Gazebo_ROS_API

ROS Topics can also be used

Recording and Playing Data

Recording and Playing Data

rosbag - will record and playback data from topics

- Record

```
$ rosbag record -a
```

Record from all topics

```
$ rosbag record -O subset topic1 topic2
```

Record from 2 topics to file subset.bag

- Playing

```
$ rosbag play bagfile
```

- Recording information of bag file.

```
$ rosbag info bagfile
```

Distributed launch

Distributed launch

ROBOT

ROS_MASTER_URI=http://**MAIN**:11311/

camera.exe

MAIN

ROS_MASTER_URI=http://**localhost**:11311/

MASTER

video_viewer.exe

LAUNCH FILE

run video_viewer
on MAIN

run camera
on ROBOT

Distributed launch

ROBOT

ROS_MASTER_URI=http://**MAIN**:11311/

MAIN

ROS_MASTER_URI=http://**localhost**:11311/

MASTER

camera.exe

video_viewer.exe

camera

SSH

video_viewer

LAUNCH FILE

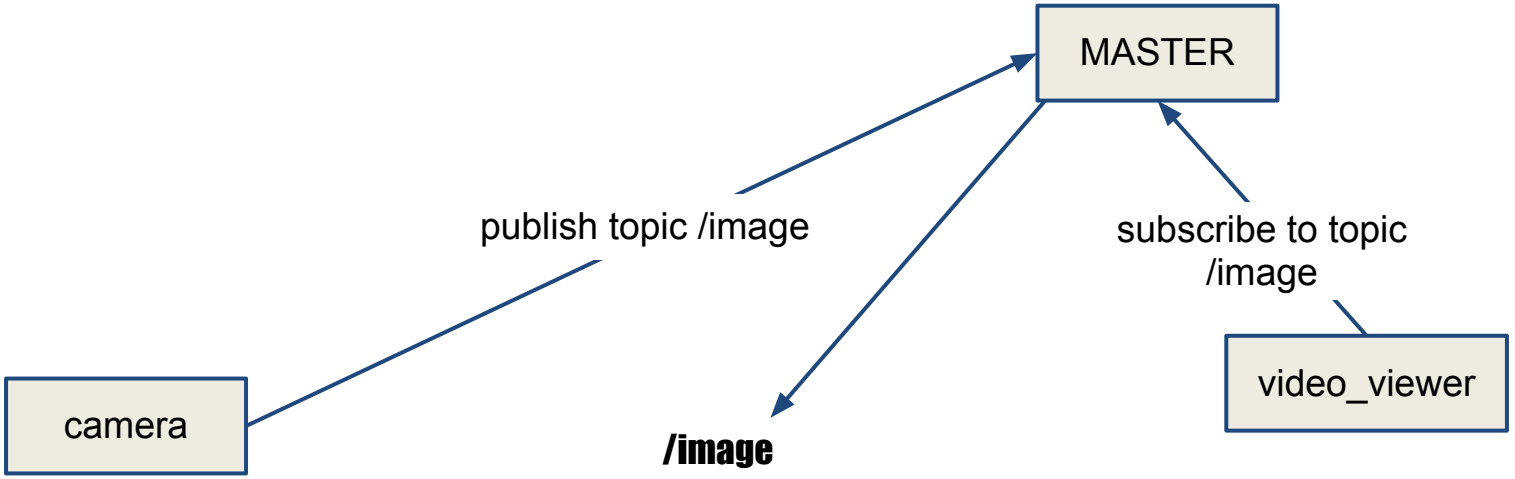
Distributed launch

ROBOT

ROS_MASTER_URI=http://**MAIN**:11311/

MAIN

ROS_MASTER_URI=http://**localhost**:11311/



Distributed launch

ROBOT

ROS_MASTER_URI=http://**MAIN**:11311/

MAIN

ROS_MASTER_URI=http://**localhost**:11311/

MASTER

camera

/image

video_viewer

Distributed launch

```
<launch>
    <machine name="ROBOT" address="10.0.0.15" />
    <machine name="MAIN" address="localhost" />
    <node
        name="camera" pkg="robil_vision"
        type="camera.exe" machine="ROBOT"
    />
    <node
        name="video_viewer" pkg="robil_vision"
        type="video_viewer.exe" machine="MAIN"
    />
</launch>
```

Distributed launch

```
<machine

   name="machine-name"
   address="10.0.0.15"

   env-loader="/opt/ros/fuerte/env.sh"
   default="true|false|never"


   user="username"
   password="*******"

/>
```

Required

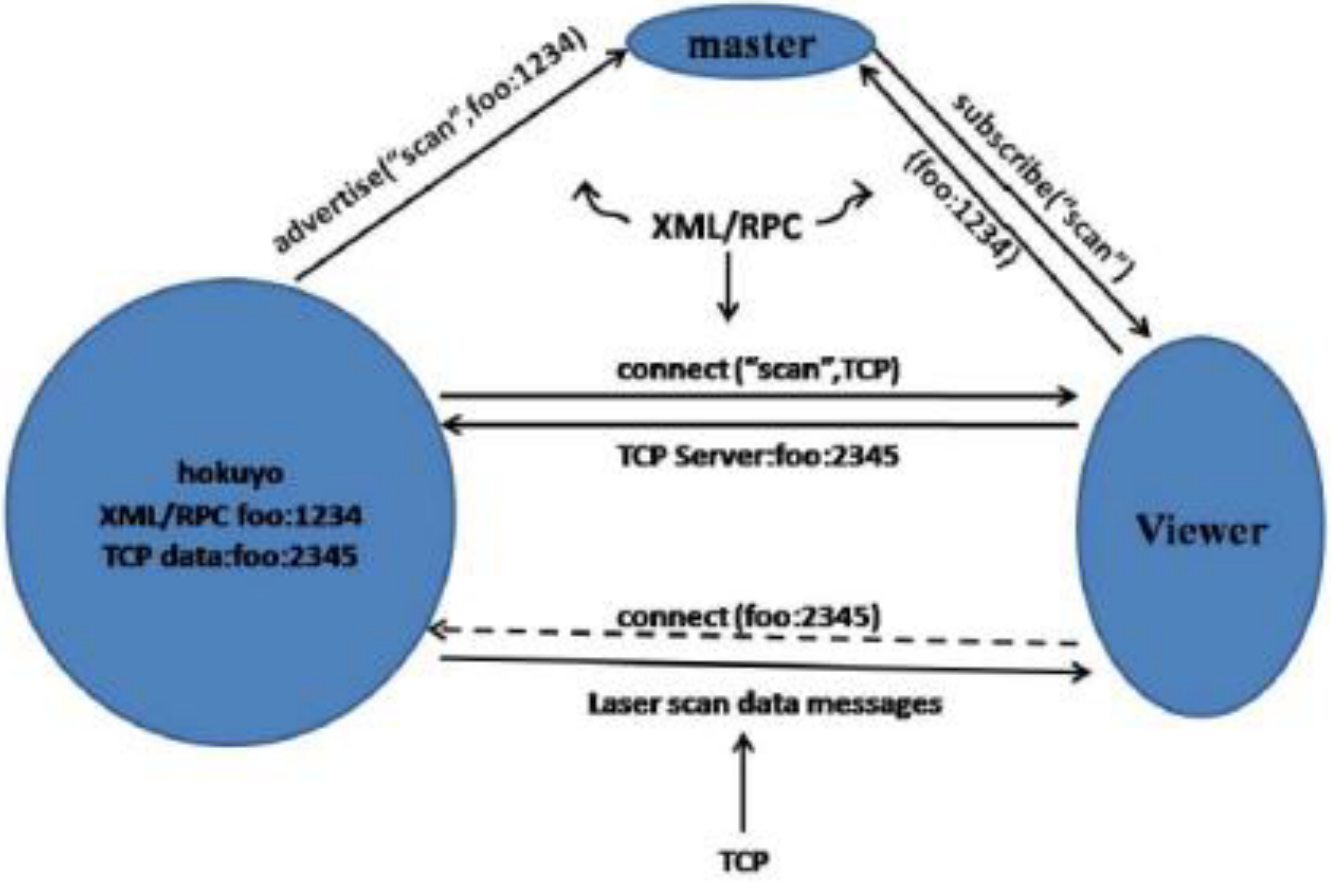
Optional

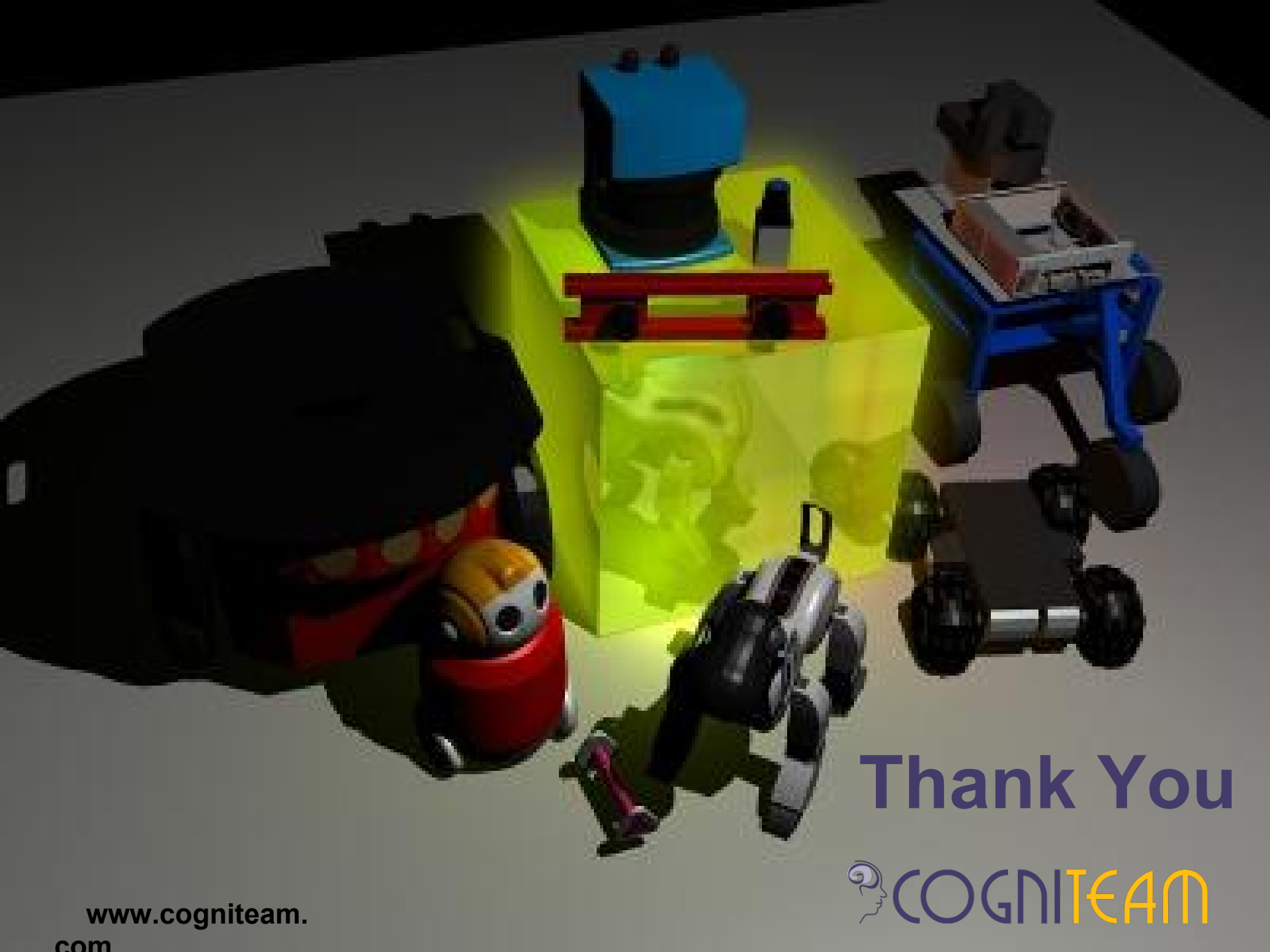
for SSH

Slide references

- http://www.ros.org/wiki/ROS/Tutorials

Messaging example





www.cogniteam.com