SOUTH DAKOTA SCHOOL OF MINES & TECHNOLOGY

RIAS Robotics & Intelligent Autonomous Systems

Introduction to Mobile Robotics

Jeff McGough

Department of Computer Science
South Dakota School of Mines and Technology
Rapid City, SD 57701, USA

October 1, 2012

Robotics research is hard[1]

- Robots embody nearly all the problems of AI
  perception, control, reasoning under uncertainty, planning, scheduling, coordination

- But we also get many of the problems of systems
  real-time constraints, limited computation & memory, imperfect limited-bandwidth communication, distributed processing, physical dynamics

[1] Brian Gerkey

Robotics research is tedious[2]

- Wide variety of hardware, each robot a little different from the next
- Code must be ported from robot to robot
- Simple things, like visualizing the sensor state of the robot, require a lot of work
- Interface libraries (if youre lucky enough to have them) often restrict the choice of programming language and/or style
- Well-understood algorithms get re-implemented over and over and over and over

[2] Brian Gerkey

What do we need?[3]

- Good robotics infrastructure, just like we have good OS infrastructure
  - Such infrastructure should:
  - be agnostic about programming language, compute platform, control and coordination architecture
  - be portable across different robot hardware
  - implement standard algorithms
  - include development tools
  - support code re-use
  - be flexible
  - plausibly simulate a wide variety of robot systems
  - be extensible
- It should also be Open Source, aka Free -
  free as in speech (libre) and free as in beer (gratis)

[3] Brian Gerkey

What do we need?[4]

- What
  - Hide unnecessary details from controller programs
- Why
  - Much common structure at the device (sensor and actuator) level
  - Produce (semi-)portable code
  - Ignore irritating details e.g., hardware revisions
  - Make client code/binary more general
- How
  - Standards based approaches (e.g., JAUS)
  - What works based approach (e.g., Player)

[4] Brian Gerkey

Software

- Modeling real-world mechanisms is difficult.
- Unexpected behavior.
  Hard-to-debug numerical explosions, jitter, poor contact behavior and general unexpected weirdness.
- APIs force the user to learn arcane concepts.
  Many simulation primitives not intuitive angular velocity, inertia tensors.
- Too slow for big models.
- Force-based modeling is tricky.
- Too many numerical parameters to tune.
  Many modeling / numerical approximations used, all with their own tradeoff parameters. Little guidance available, need to experiment.

Software

- Matlab-Simulink
- Open Dynamics Engine
- Newton Game Dynamics
- Webots
- YARP
- Orocos
- ORCA
- Player
- Carmen
- MS Robotics Studio

Matlab

- Numerical tools
- Symbolic Algebra
- Programming language
- Toolboxes
- Controls toolbox
- Optimization toolbox
- Neural Network toolbox
- Signal processing toolbox
- Image processing toolbox
- Computer Vision toolbox
- Simulink
- and much much more ...

... commercial.

Open Dynamics Engine

ODE is an open source, high performance library for simulating rigid body dynamics.

It is fully featured, stable, mature and platform independent with an easy to use C/C++ API.

It has advanced joint types and integrated collision detection with friction. ODE is useful for simulating vehicles, objects in virtual reality environments and virtual creatures.

It is currently used in many computer games, 3D authoring tools and simulation tools.

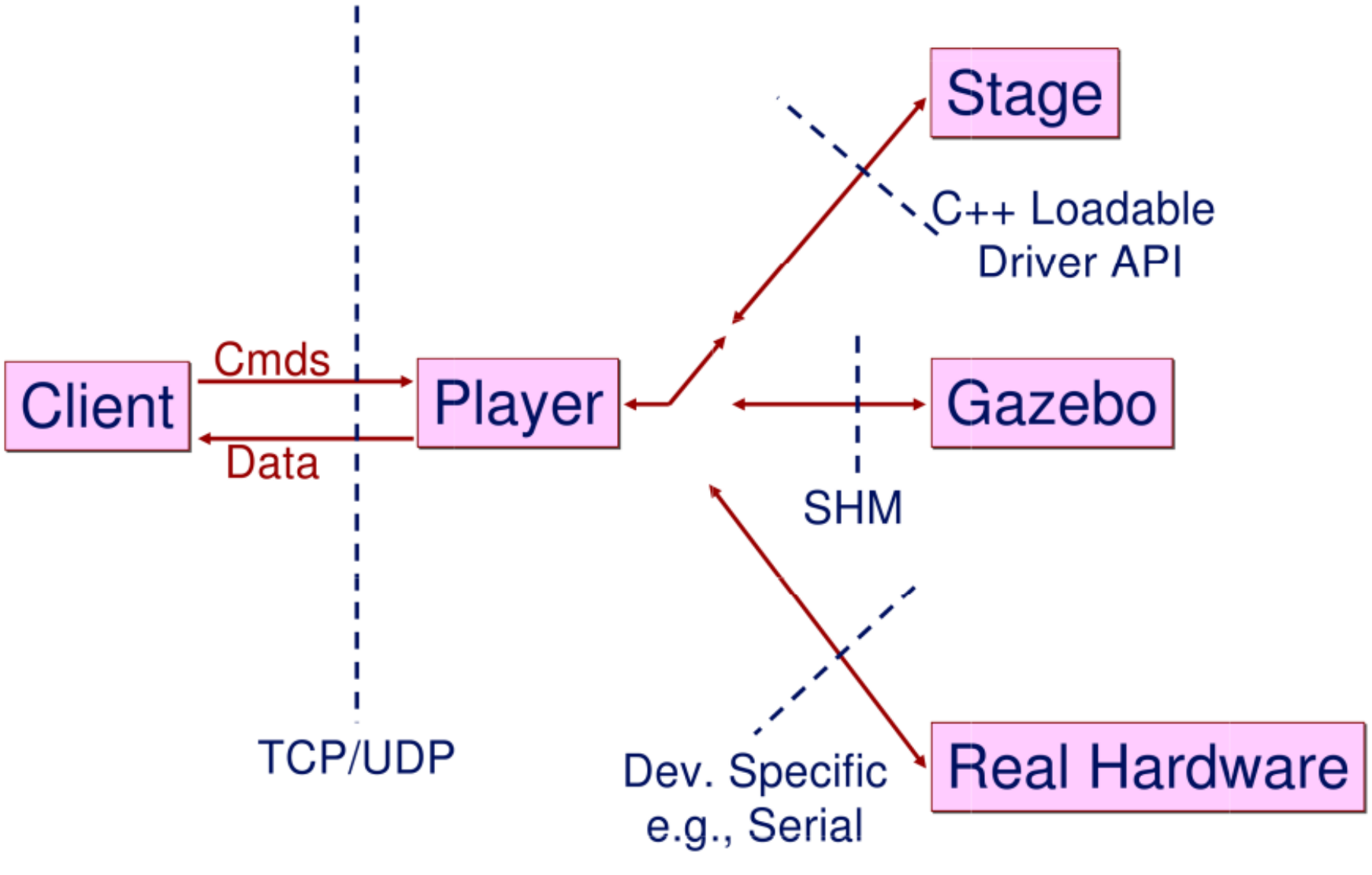
Player-Stage

Player is a robot framework or more accurately a Hardware Abstraction Layer. Stage is a robot simulation tool, which connects to Player.

That means that it talks to the bits of hardware on the robot (like a claw or a camera) and lets you control them with your code, meaning you don't need to worry about how the various parts of the robot work.

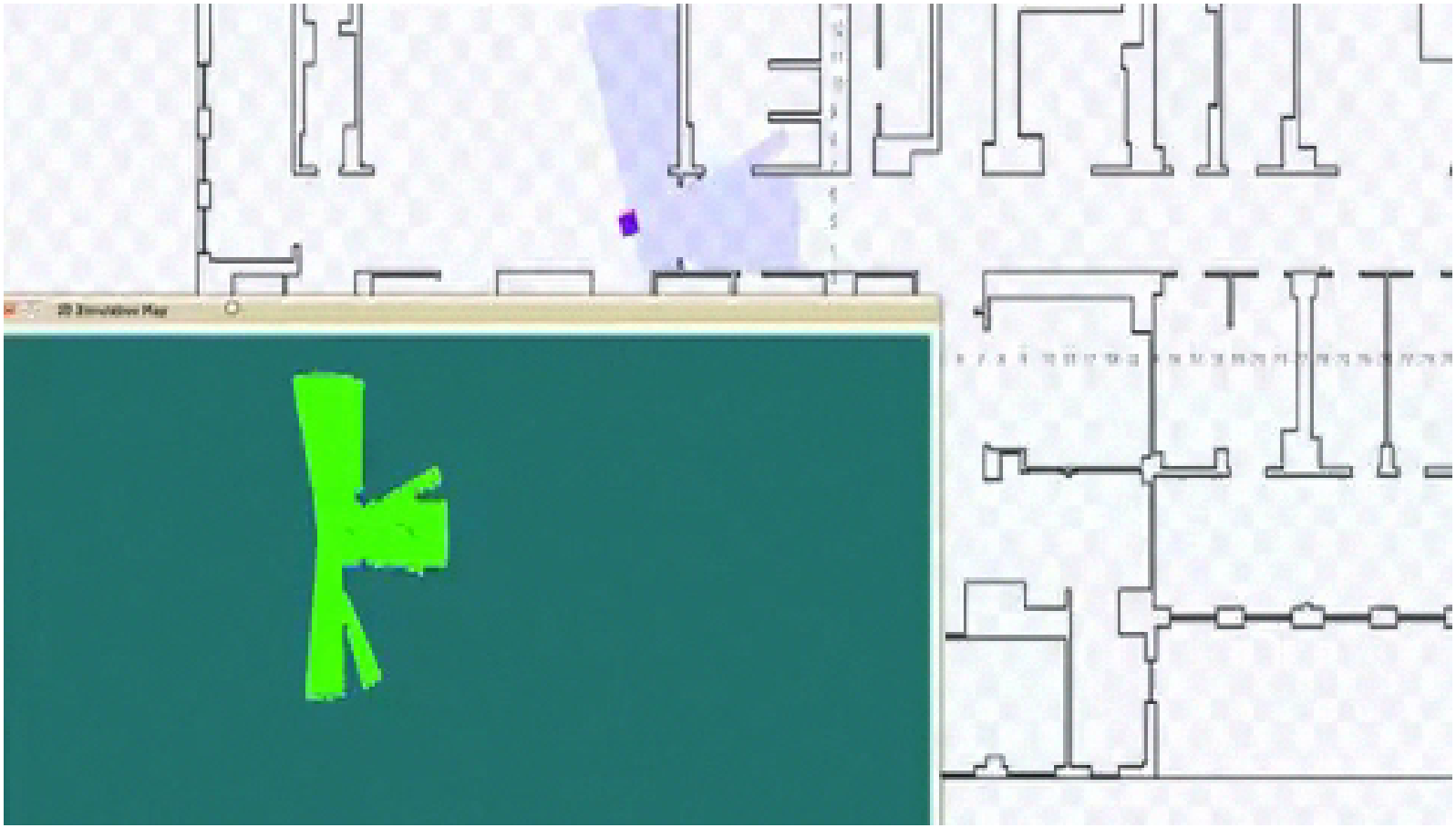
Stage is a plugin to Player which listens to what Player is telling it to do and turns these instructions into a simulation of your robot. It also simulates sensor data and sends this to Player which in turn makes the sensor data available to your code.

Player-Stage



Player-Stage

Player-Stage



Player-Stage

Player++

Player has been upgraded and it is called ROS

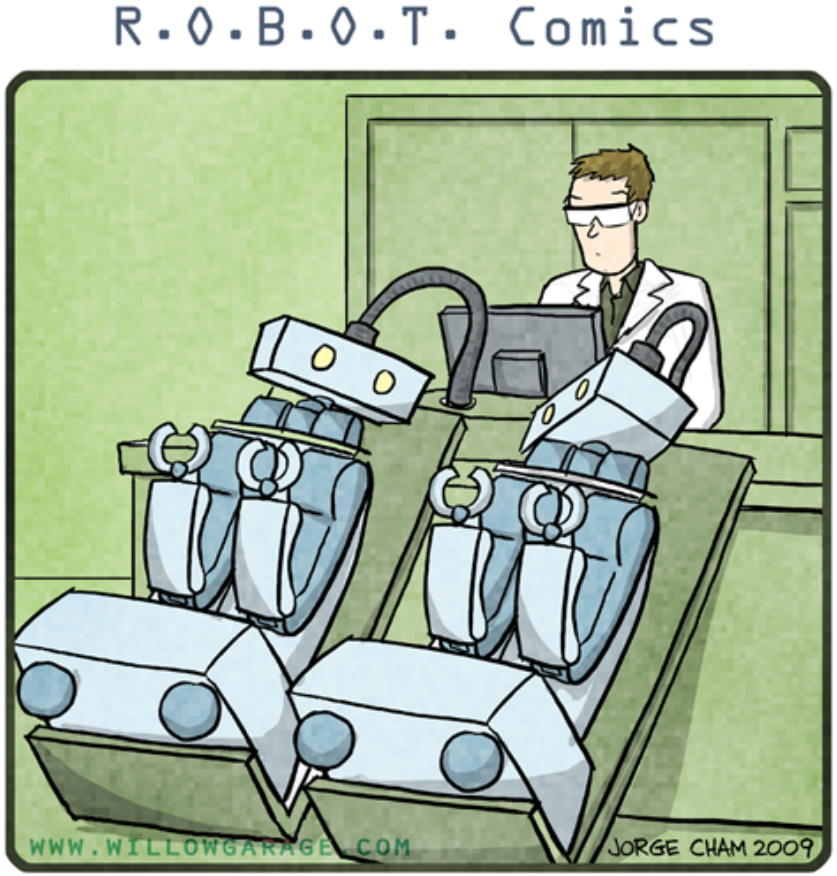
ROS = Robot Operating System

From now on, everything we said about Player holds true for ROS - we think ...

ROS Topics

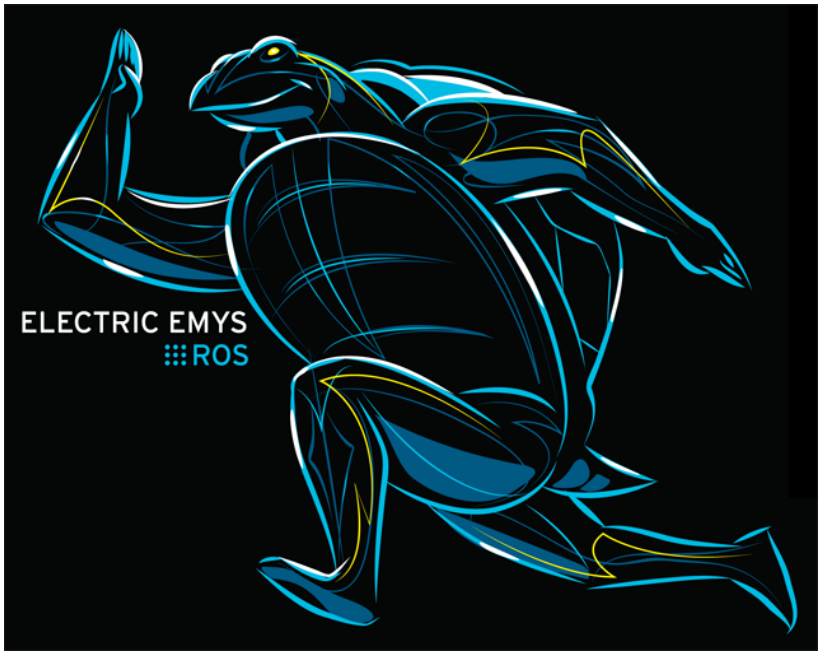
1. ROS Overview
2. Terminology
3. File System Structure
4. Command Line Tools

ROS Overview



What is ROS?

- Open-source, meta operating system for your robot.
- Provides tools, APIs, hardware abstraction, low-level device control, package management, message-passing between processes and multiple computers, and much more...
- Not a realtime framework, but it is possible to integrate ROS with realtime code.



ROS

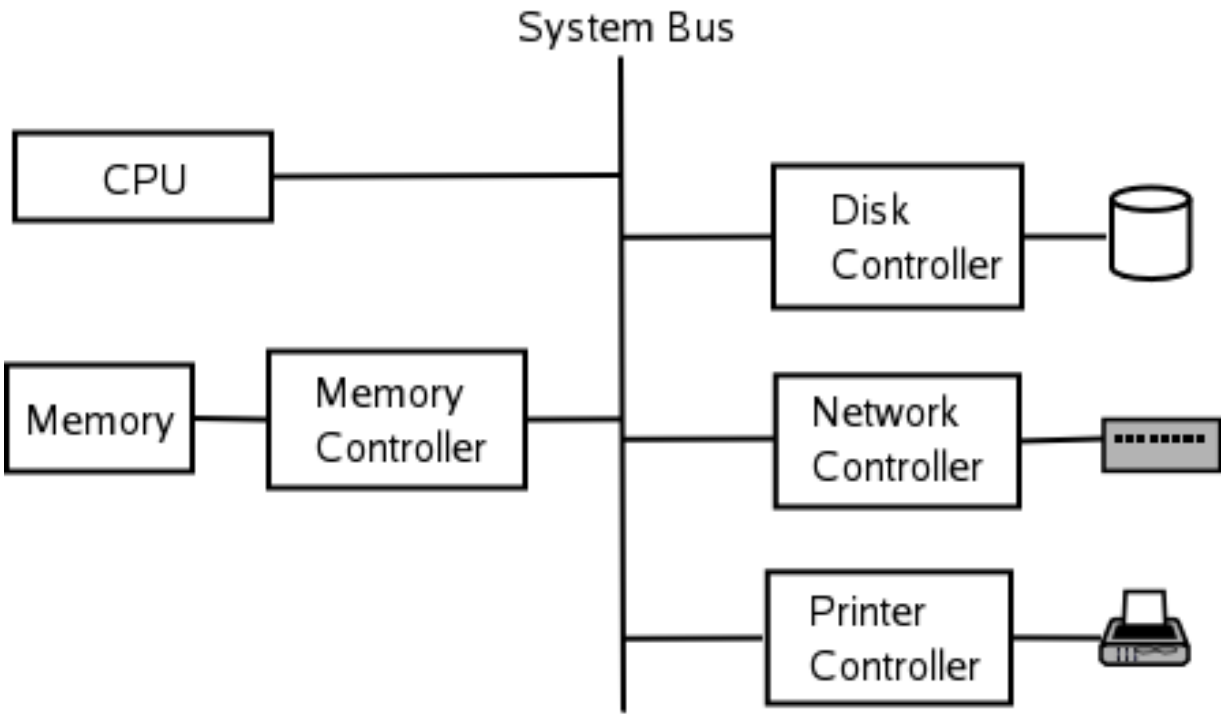
ROS Concept

- Not a realtime framework, but it is possible to integrate ROS with realtime code.
- Provide an abstraction layer and drivers between computation and embodiment
- Analogy: hardware abstraction layer in an operating system

Operating Systems

Operating Systems Overview

Recall the hardware ....

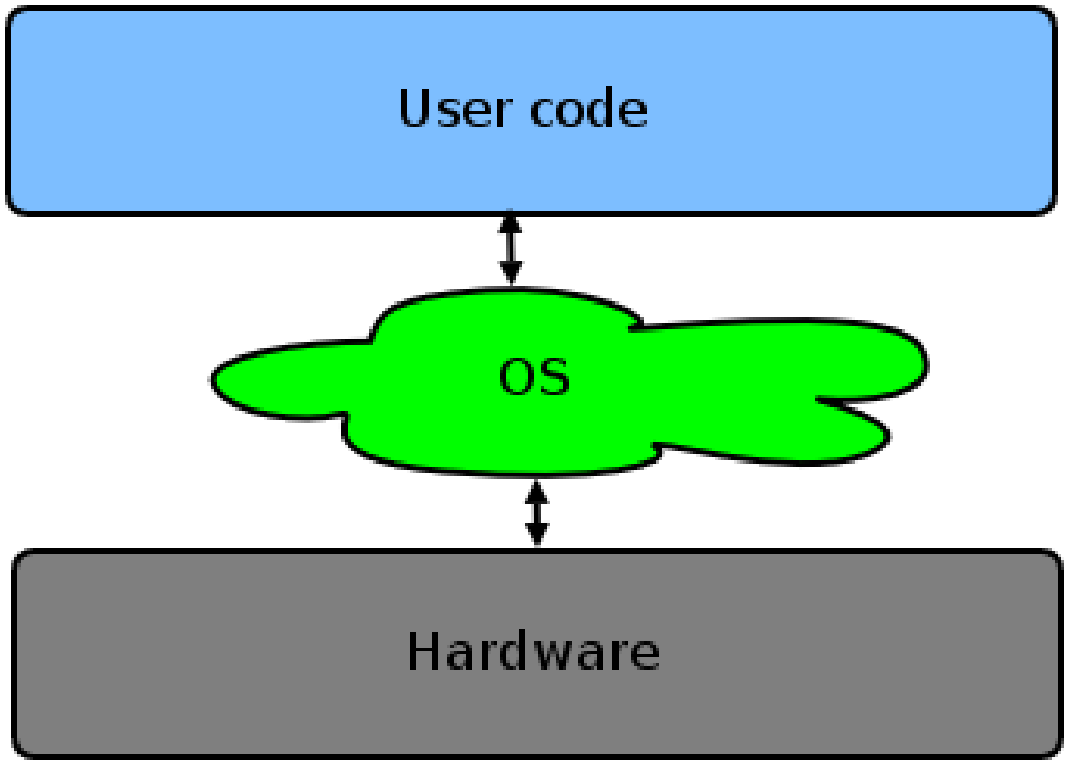


OS Requirements

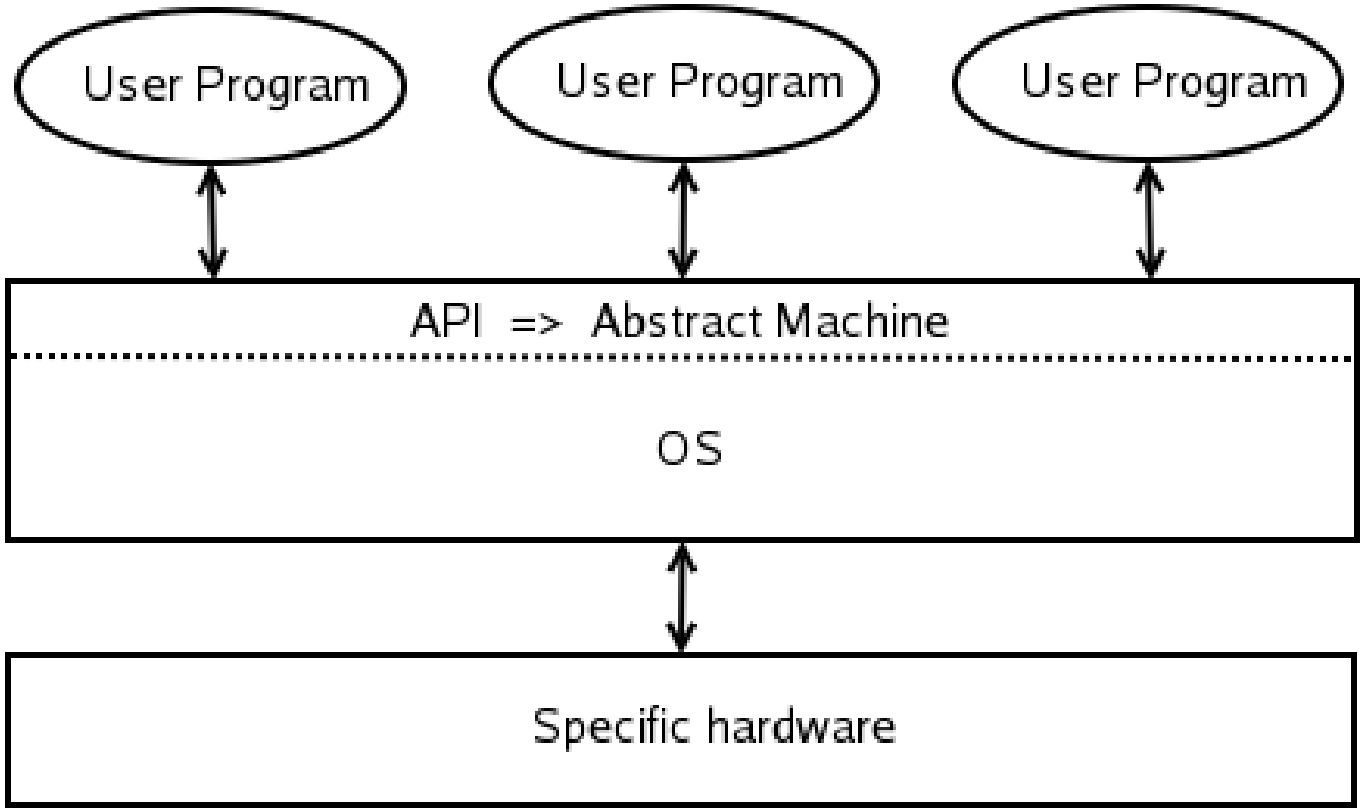
- What is an operating system?
  - Kernel?
  - Distribution?
- Computer system
  - Hardware
  - OS
  - Applications
  - Users
- What does it do?
  - Resource Allocator
  - Control Program
  - Resource Manager

Operating Systems

- OS is a resource manager

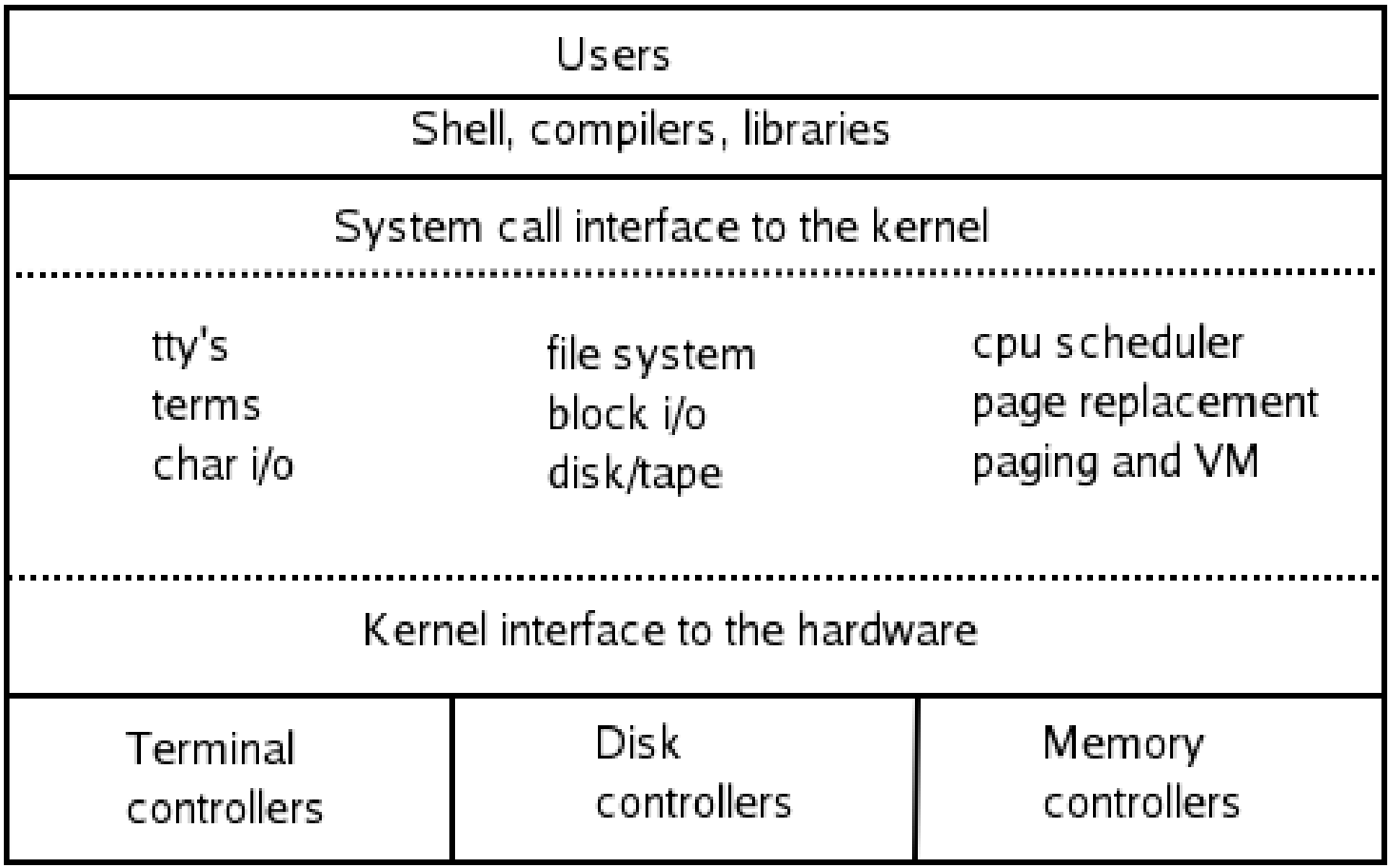


OS Design Abstraction



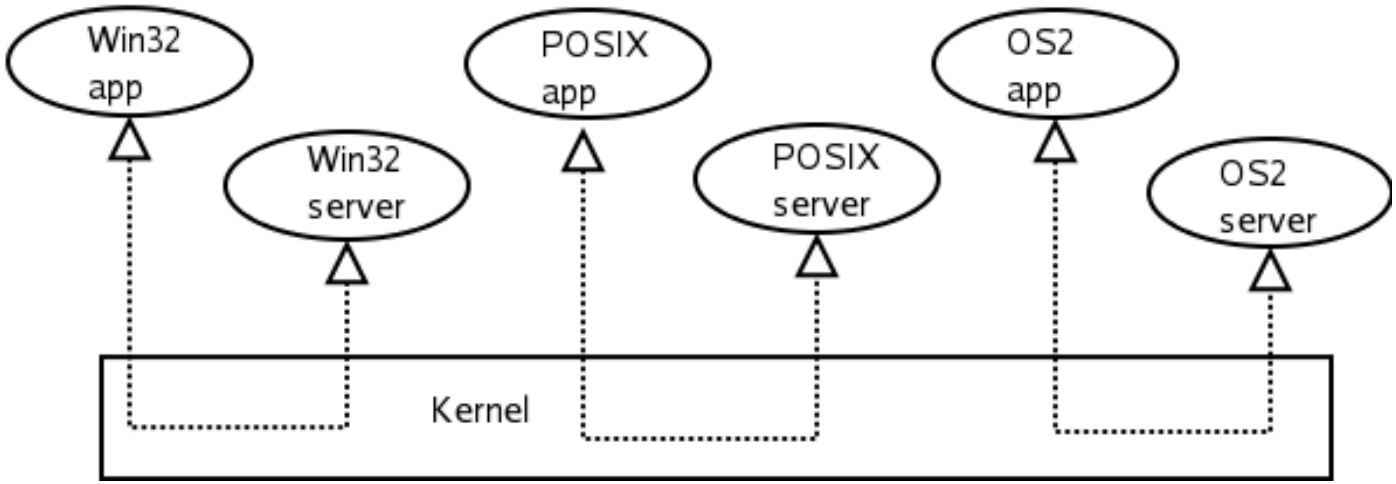
OS

- Linux monolithic kernel



Microkernel Design

- NT design concept



History

1999

University of Southern California Robotics Research Labs starts use of on-board TCP/IP and 802.11 wireless Ethernet as standard equipment.

2000

Unversity of SC releases Player Project. (Player/Stage/Gazebo is a robot simulation project that utilized TCP/IP protocol.)

History (Continued...)

2007

Stanford AI Lab develops ROS under the name *switchyard*, utilizing the Player-Stage model.

2008

Willow Garage, a robotics research institue/incubator, assumes development role.

2010

ROS officially gets released.

- 1999-2009 Player: client server model
- 2008 - Current ROS: peer to peer model

ROS

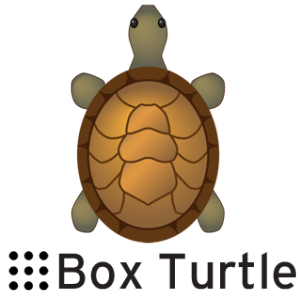
Design approach:

- Reduce reinvention, increase interoperability and reproducibility
- Inter-process communication for robots
  - Message-passing uses a format called XML/RPC, which allows messages to be sent in a XML format.
- Software functionality modularized as ROS nodes
- Run-time system: nodes communicate over IP net
- Packaging system: nodes organized into distributable packages

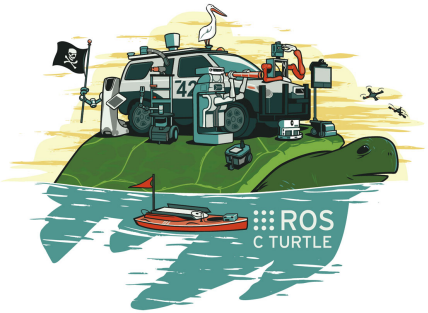
ROS Releases (Every 6 Months)

2010 Releases

- Box Turtle (March 2)



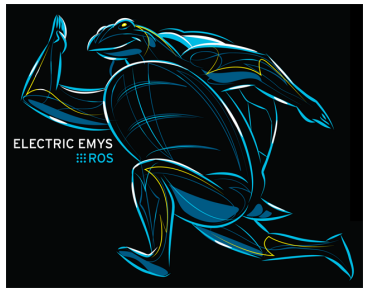
- CTurtle (August 2)



2011 Releases

- Diamondback (March 2)

- Electric Emys (August 30)



Why use it?

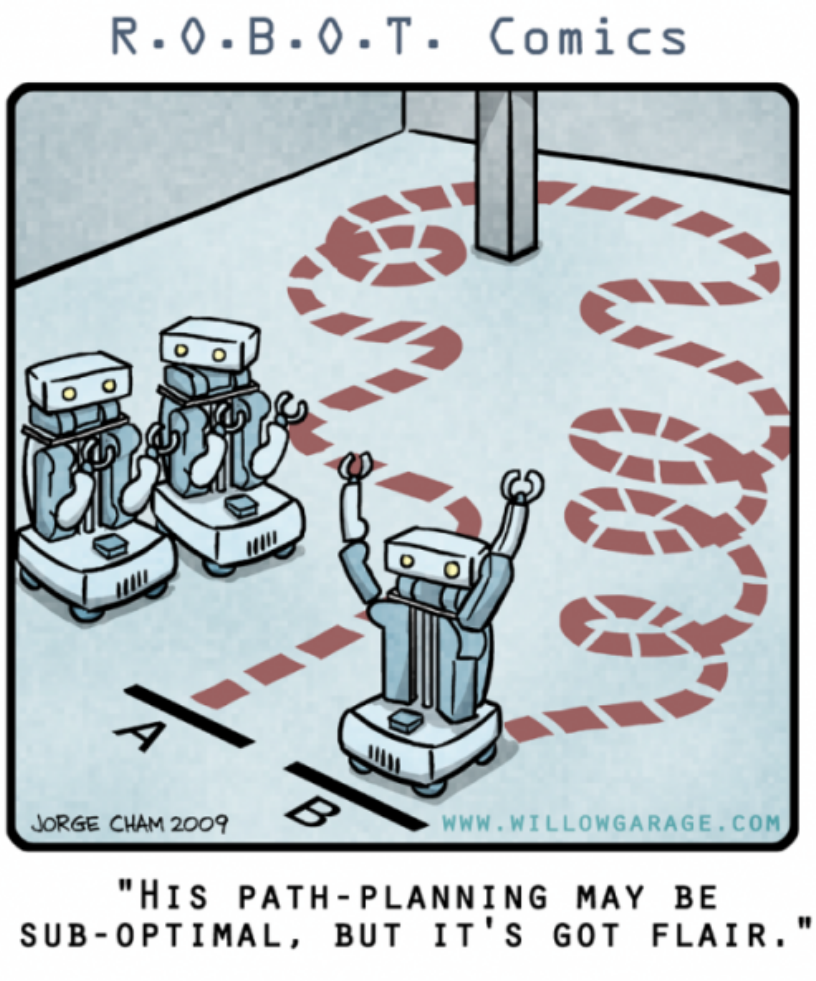
- Promotes code reusability.
- Programming Language independence.
  - Primarily programmed in C++ and Python.
  - Supports Lisp.
  - Experimental with Java and Lua.
- Operating System independence.
  - Primarily tested on Ubuntu and MacOSX. [ros.org]
  - Experimental on Fedora, Gentoo, and Arch Linux.
- Message-passing system.
  - Publish/Subscribe communication paradigm.

ROS Installation

Go to `http://www.ros.org/wiki/ROS/Installation`

follow the instructions.

Terminology (The Computation Graph Level)



Nodes

"The Clients"

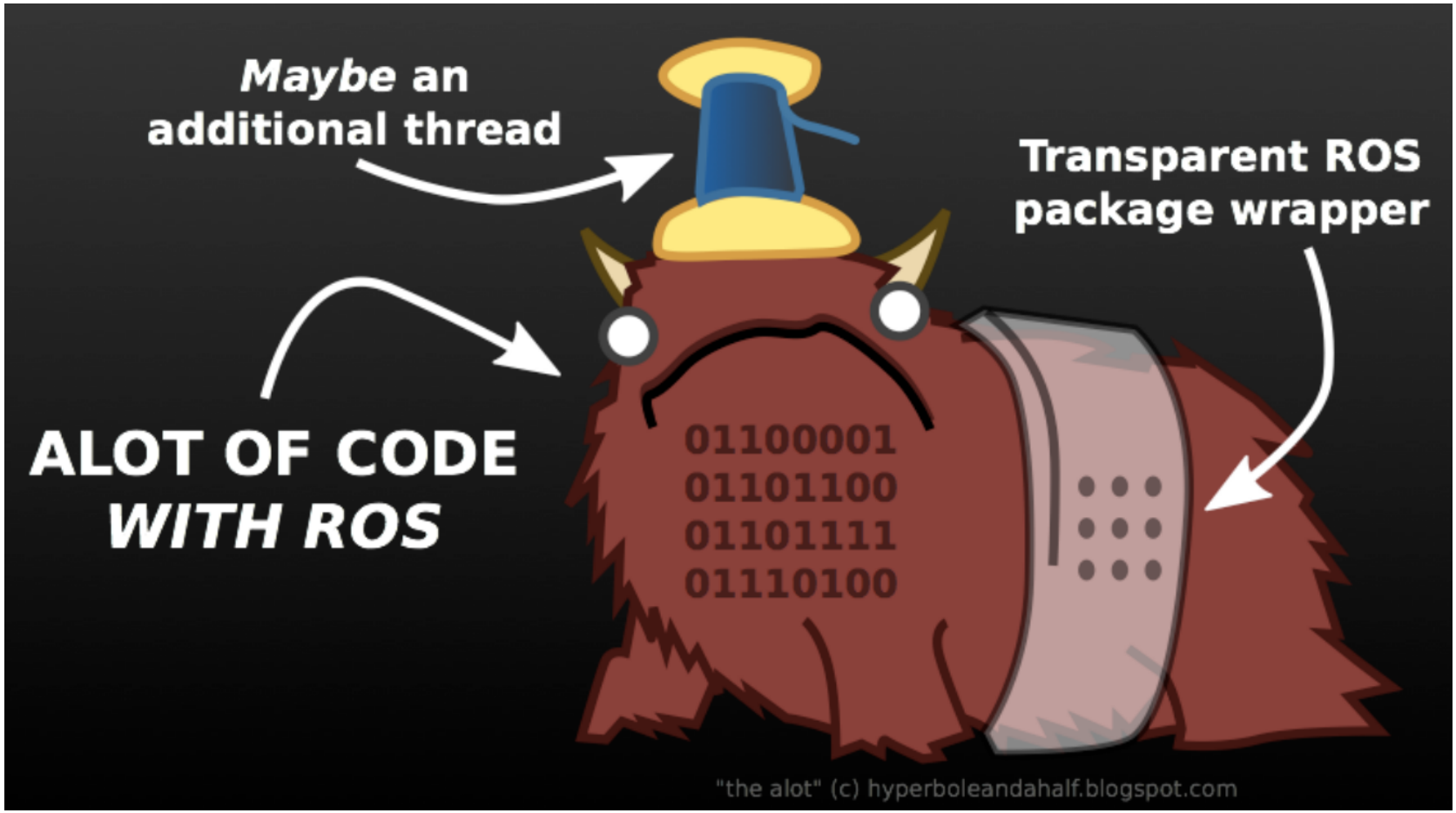
Node - a process that performs computation. [ros.org]

- i.e., Node = a simple program.

Isolation

- Allows for data processing to be broken up.
  - Reduces code complexity.
  - Reusable for other systems.
- Crashes do not affect the whole system.
- Nodes are unaware of each others existence.
- Communicate to make complicated systems.

ROS Nodes



ROS nodes are typically wrappers around a device, software library, or a "alot" of code.

ROS Nodes

Nodes are written as part of packages:

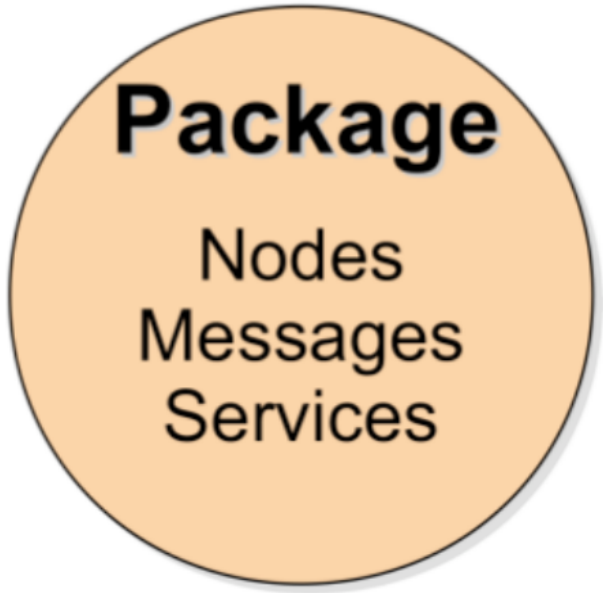
A package is a **building block** and implements a reusable capability

- Complex enough to be useful
- Simple enough to be reused by other packages

A **package** contains one or more executable processes (nodes) and provides a ROS interface:

Messages describe the data format of the in/output of the nodes. For example, a door handle detection node gets camera images as input and spits out coordinates of detected door handles.

Service and **topics** provide the standardized ROS interface to the rest of the system.



Master

"The Server"

Roles of the Master

- Provides registration and lookup.
- Allows nodes to locate one another.
- Allows nodes to communicate.
- Provides a Parameter Service.

Nodes are dependent on the master.

Messages

Message - a simple data structure. [ros.org]

Very similar to C structs

- Uses primitive data types (int, float, bool, etc.).
- Supports arrays.
- Nested structures (nested messages).

Data Broadcasting

- Many-to-many, one-way communication

Topics

Topics - named buses over which nodes exchange messages. [ros.org]

Topic Types

- Every topic has a type (the message type).
- Nodes must know the topic name and topic type.

Topic Transports

- TCP/IP (default transport)
- UDP (currently supported only in C++)

Services

Services - request and reply communication. [ros.org]

Defined by a pair of messages.

Services are stored in the Parameter Service.

- Parameter Service - runtime database.

Bags

Bag - a file format in ROS for storing ROS message data. [ros.org]

Allow you to record and playback messages.

- Useful for quick testing.

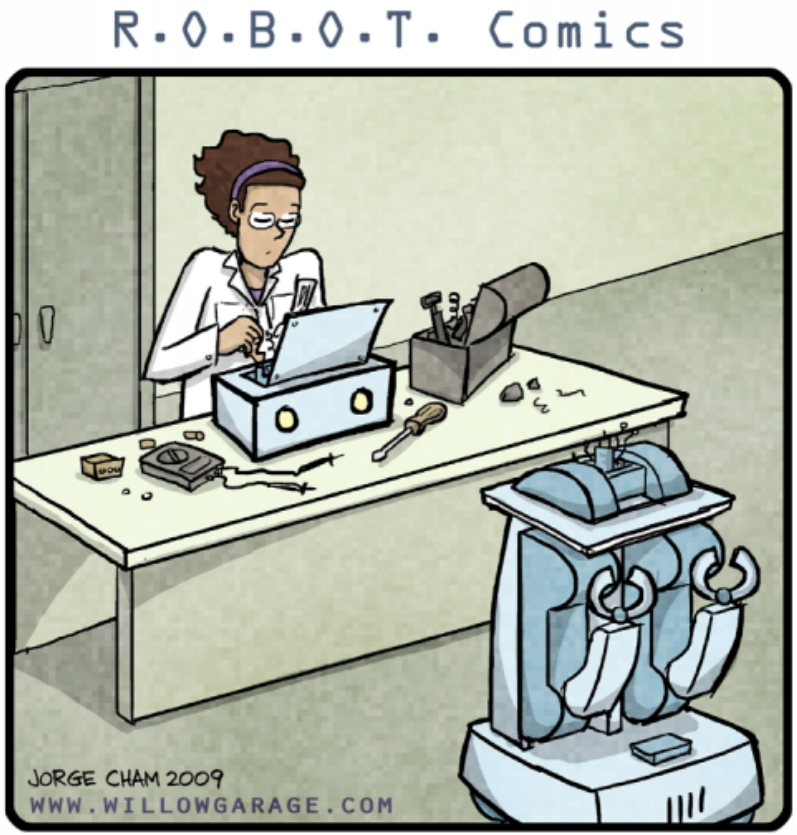
Recording

- Subscribes to a topic.
- Saves the data received into a file.

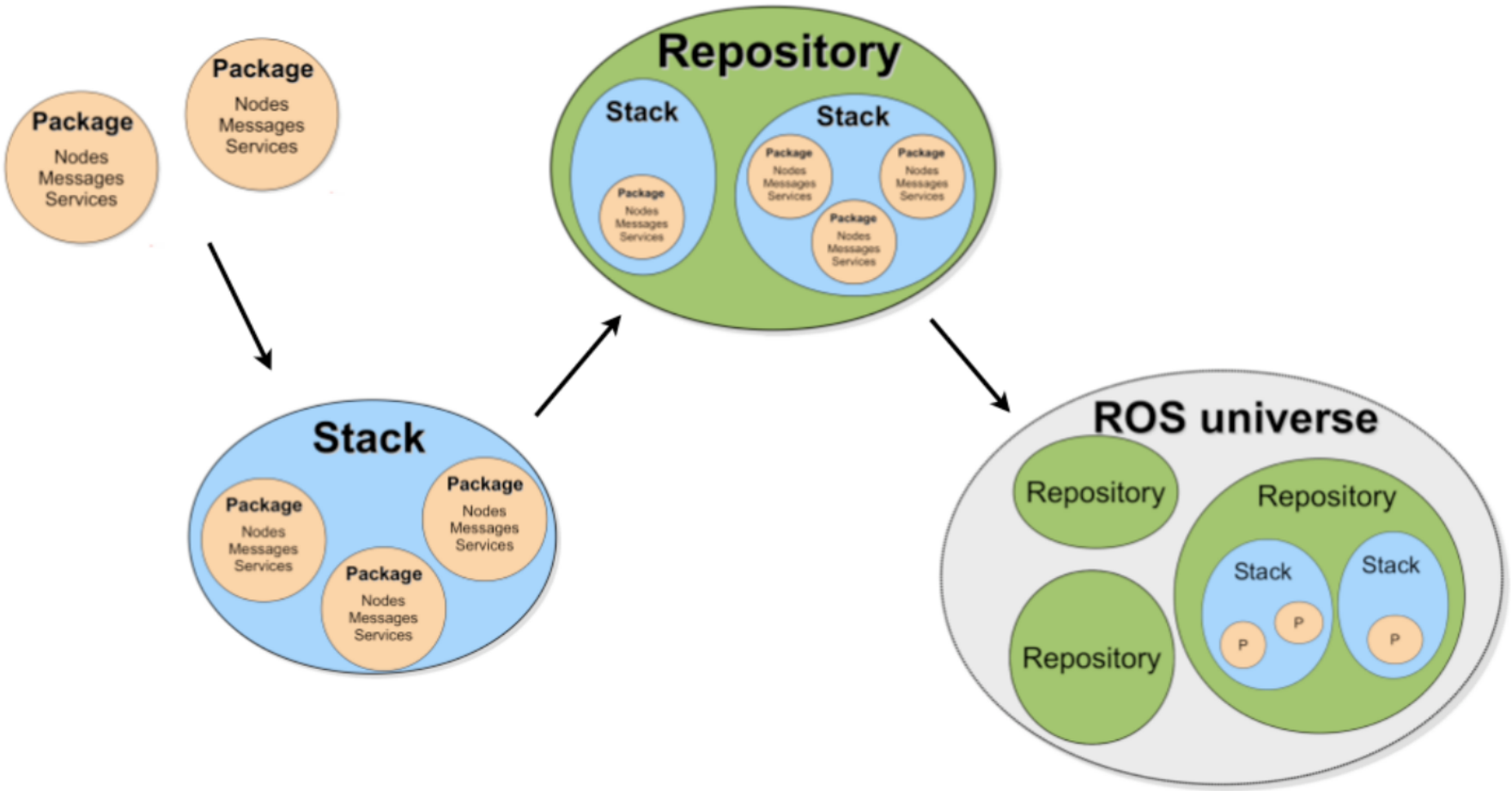
Playback

- Simulates the initial publisher.
- Publishes over same topic.

File System Structure (The Filesystem Level)



ROS File System



Packages

Software organizer

- Collection of nodes
- API library
- Datasets, configuration files, etc.

The "Goldilocks" principle:

- Packages provide "enough functionality to be useful, but not too much that the package is heavyweight and difficult use from other software." [ros.org]

Manifests (manifest.xml)

- Minimal specification of a package.
- States the package dependencies.

Stacks

Stacks - collection of packages that provide aggregate functionality. [ros.org]

Goals

- Organize packages.
- Ease code reuse.
- Simplifiy code sharing.

Stack Manifests (stacks.xml)

- Used for supporting distribution and installation.
- Provides information on the stack.
- States the stack's dependency on other stacks.

Message and Service Types

Simplified data description language.

Description Locations

- Message are stored in .msg files.
- Services are stored in .srv files.

Source code is automatically generated.

Message and Service Types (Examples)

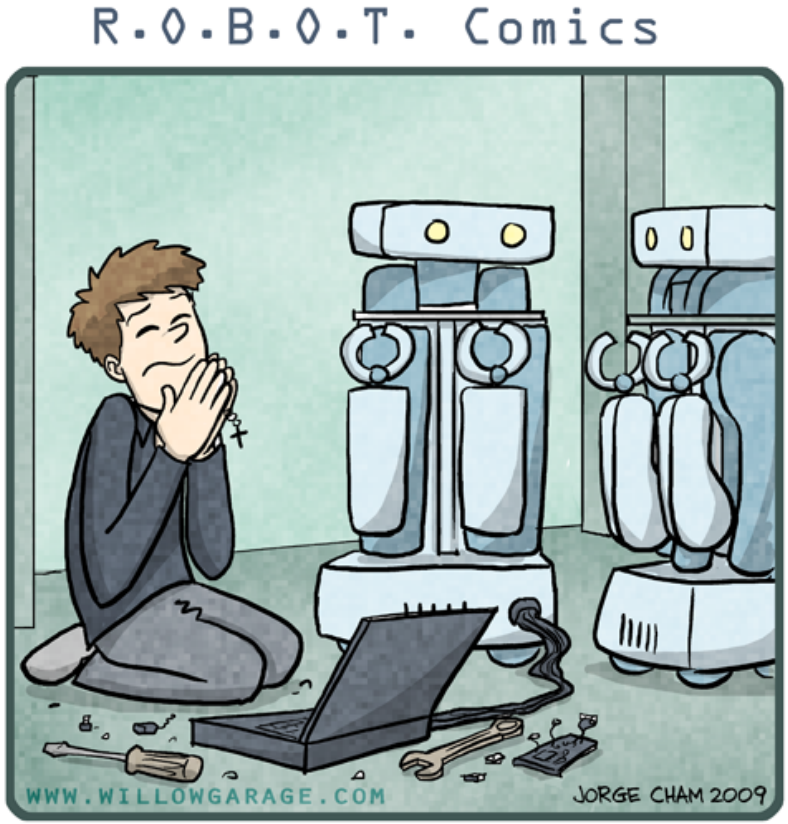
Image.msg:

```
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

GetPlan.srv:

```
geometry_msgs/PoseStamped start
geometry_msgs/PoseStamped goal
float32 tolerance
- - -
nav_msgs/Path plan
```

Command Line Tools



"THEY ALL SAY THEY'RE AGNOSTIC, UNTIL IT'S TIME FOR DIAGNOSTICS."

Running ROS

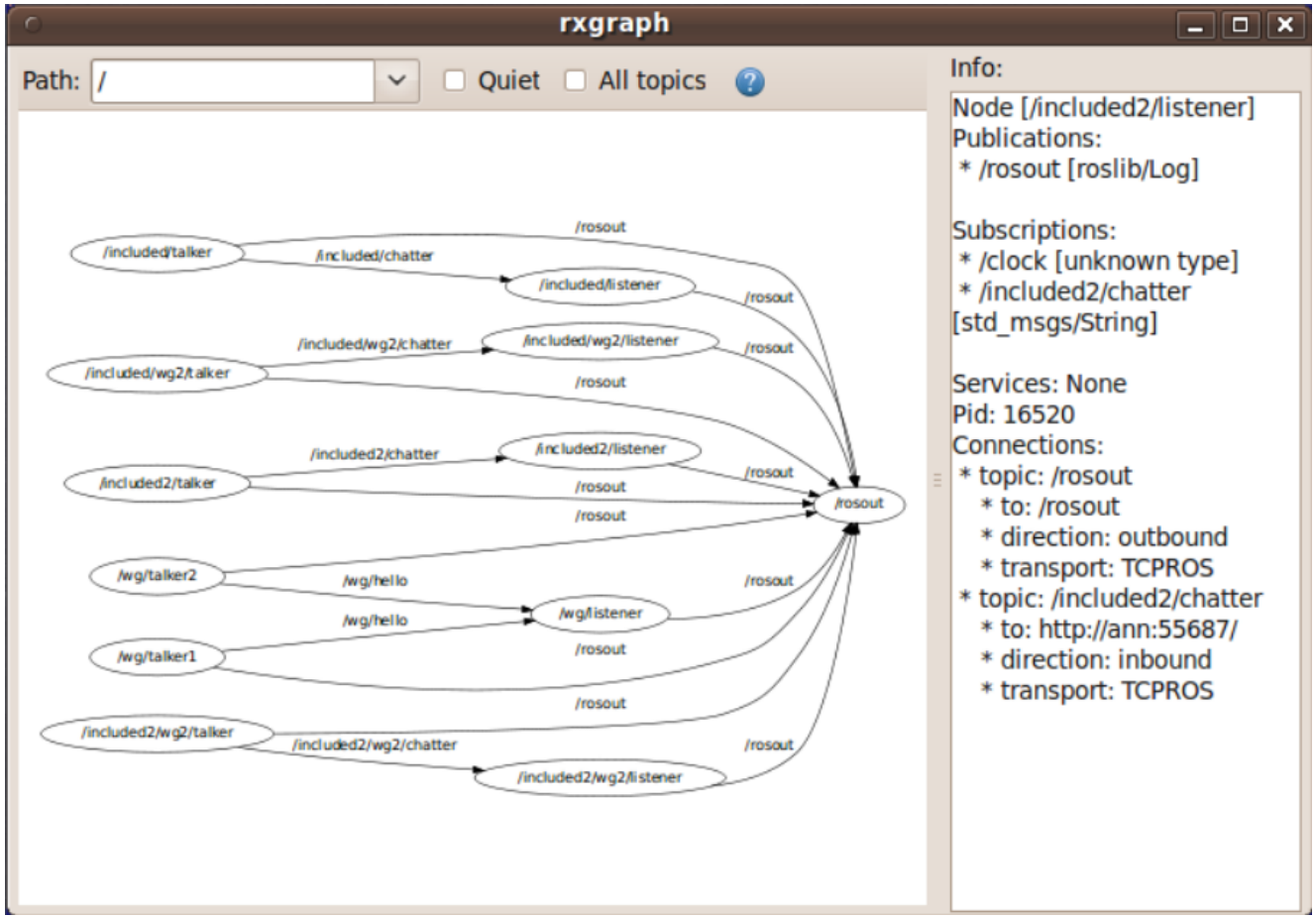
roscore	Starts the core system (the Master).
rosrun	Starts a single node. `rosrun package-name node-name`
roslaunch	Starts a collection of nodes using roslaunch xml files `roslaunch launch-file`

Interacting with ROS

rosnode	Provides information on a running node and its topics.
rostopic	Provides a list of all active topics.
rosparam	View and modify parameters.
rosservice	Provides information about running services.
rosmsg	View messages.
roserv	View services.

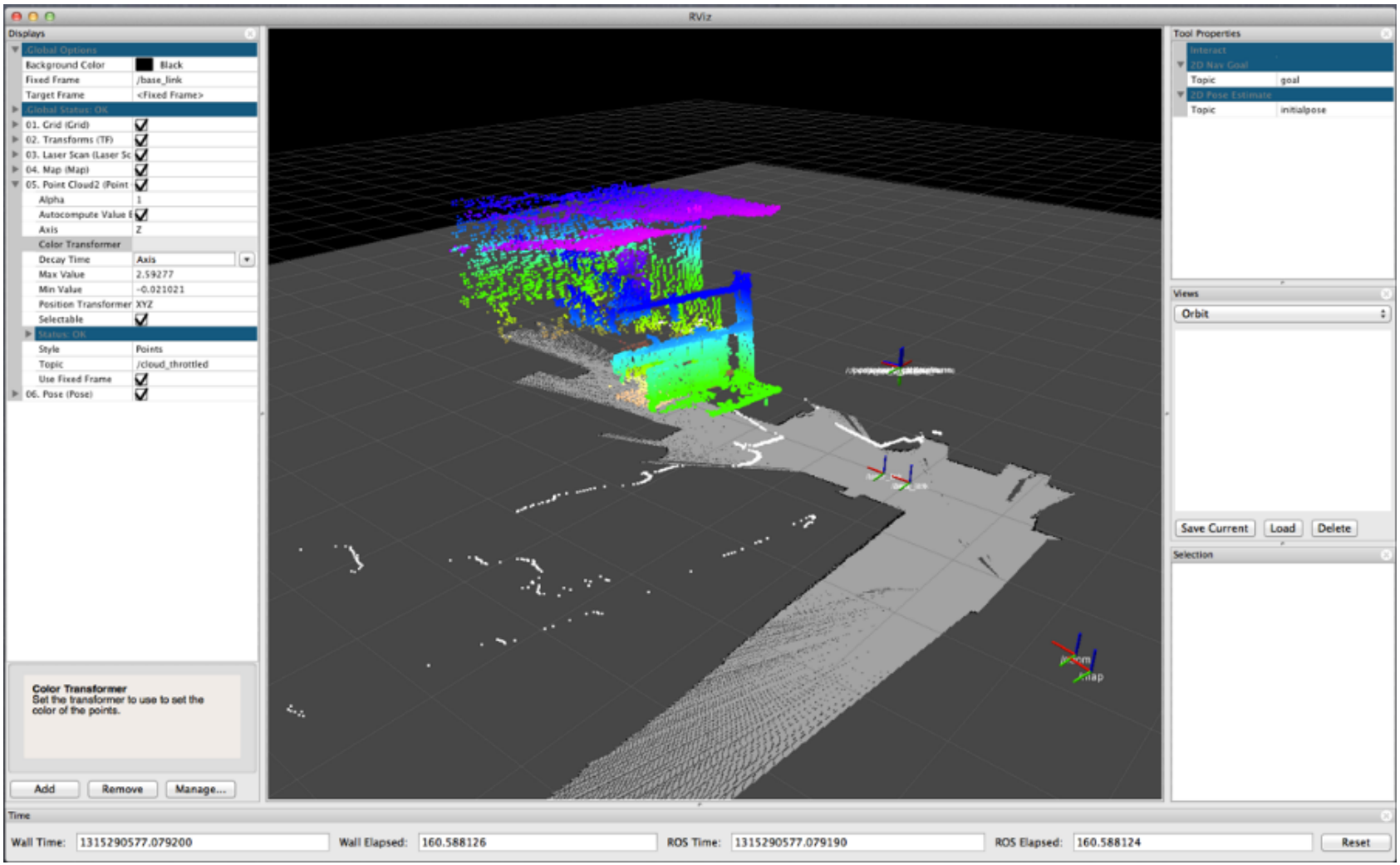
ROS Commands

rxgraph



ROS Commands

rviz



File System

Stacks

- rosstack.....Find and view information on a stack.
- roscreate-stack.....Creates user-defined stack.

Packages

- rospack.....Find and view information on a packages.
- roscreate-pkg.....Creates user-defined package.
- rosdep.....Install a package's dependencies.
- rosmake.....Compile nodes in a package.

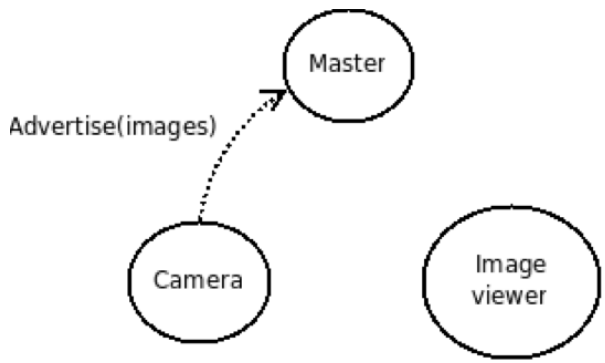
Other Commands

- roslocate.....Locate a package or stack.
- roscd.....Change to a package or stack directory.
- rosclean.....Check log file disk usage or delete log files.

Publish/Subscribe Communication

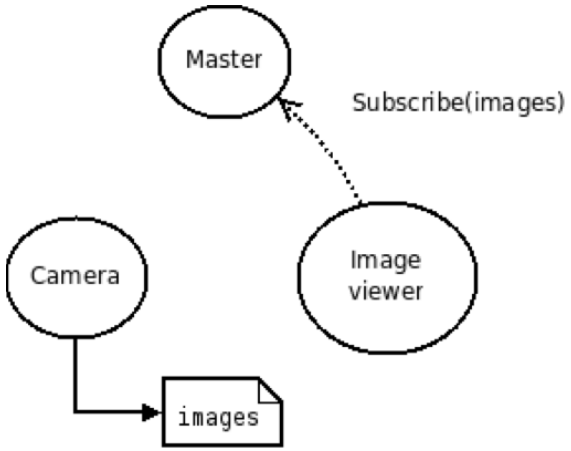
Publishing

1. A node advertises a message type over a topic name.
2. The node can then publish messages...
   - At a specific rate (in Hz) OR
   - At any time



Subscribing

1. A node subscribes to a topic name.
2. If no other node is publishing, the subscribe will not receive any data.



Note:

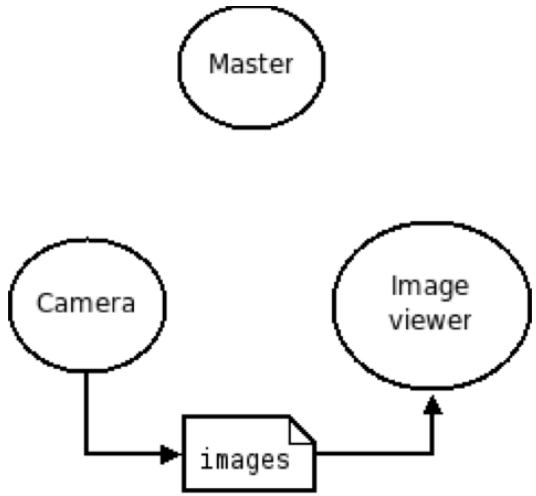
- The order of publishing/subscribing does not matter.

Connection

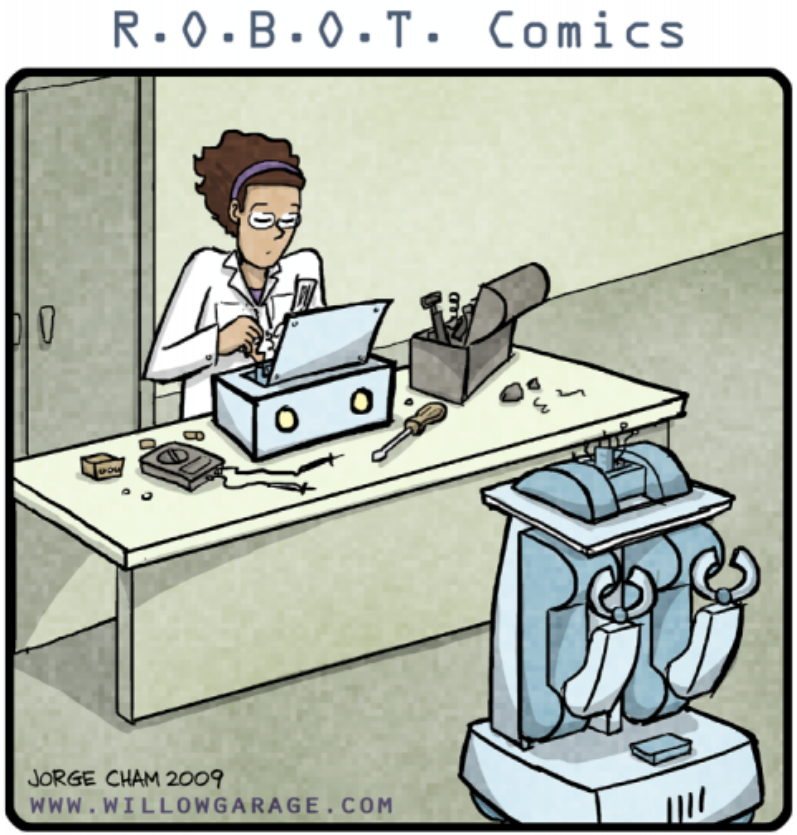
Communication is directly between nodes (not through the master).

Note:

- Topic names do not have to be unique.
- Message types do not have to be unique.
- Together, a topic name with a message type must be unique.



Node Development



Creating a Package

roscreate-pkg simple_nodes roscpp std_msgs

export ROS_PACKAGE_PATH=${HOME}:$ROS_PACKAGE_PATH

Edit CMakeLists.txt to build your nodes:

- rosbuild_add_executable(simple_node src/simple_node.cpp)
- rosbuild_add_executable(simple_subscriber src/simple_subscriber.cpp)
- rosbuild_add_executable(simple_publisher src/simple_publisher.cpp)

Simple Node

```
#include "ros/ros.h"
#include <string>
#include <sstream>

int main(int argc, char **argv) {
    ros::init(argc, argv, "simple_node");
    int count = 0;
    std::string message = "Hello world: ";

    ros::NodeHandle n;
    ros::Rate rate(1);
    while(ros::ok()) {
        std::stringstream ss;
        ss << message << count;

        ROS_INFO(%s, ss.str().c_str());

        count++;
        rate.sleep();
    }

    return 0;
}
```

Simple Subscriber

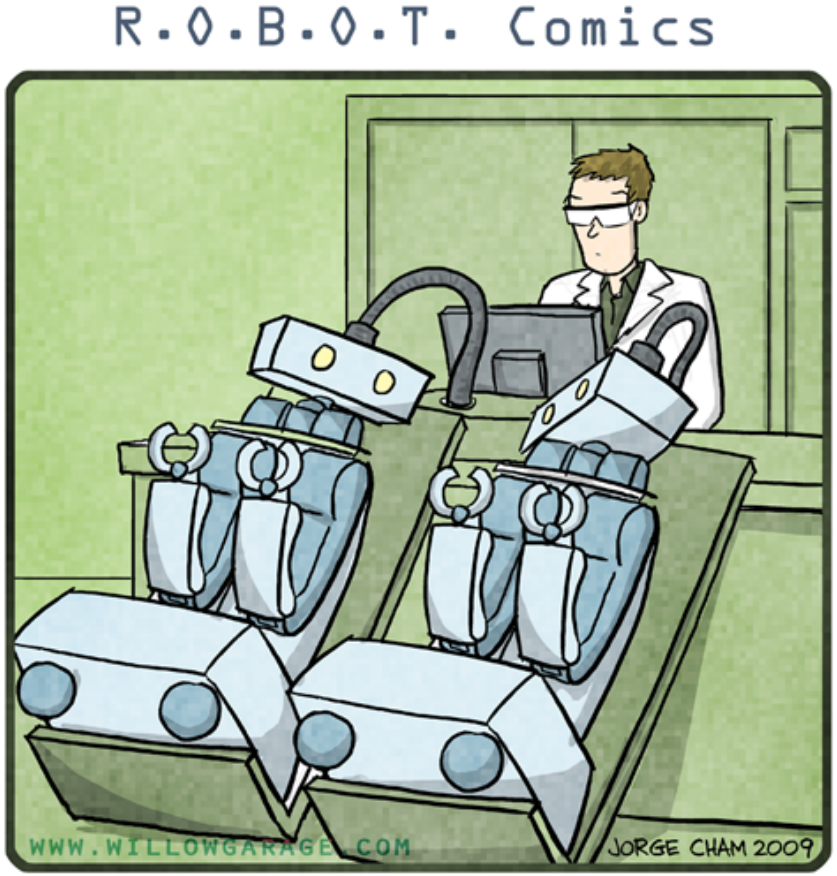
```
#include "ros/ros.h"
#include "std_msgs/String.h"

void listenerCallback( const std_msgs::String::ConstPtr& msg ){
    ROS_INFO("I heard: %s", msg->data.c_str() );
}


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

    ros::Subscriber sub = n.subscribe("talker_topic", 100, listenerCallback );

    ros::spin();

    return 0;
}
```

Simple Publisher

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>

int main( int argc, char **argv )
{
    ros::init( argc, argv, "talker" );
    ros::NodeHandle n;
    int count = 0;

    ros::Publisher pub =
    n.advertise<std_msgs::String>("talker_topic", 100
          );

    ros::Rate loop_rate(1); //in hertz

    while( ros::ok( ) )
    {
        std_msgs::String msg;
        std::stringstream cMsg;

        ss << "Hello " << count;n 0;
}
```

```
  msg.data = ss.str();

        ROS_INFO( "Publishing: %s", msg.data.c_str()
              );

        pub.publish(msg);

        ros::spinOnce();

        loop_rate.sleep();

        count++;
    }

    return 0;
}
```

Resource Names



"Do you ever feel like you're in The Matrix?"

Names

Everything in ROS has a name.

- nodes, topics, messages, etc.

Names help provide encapsulation (grouping related resources)

Graph Resource Names

Valid Graph Resource Names:

First character must start with...

- a..z, A..Z
- $\sim$ (tilde)
- / (forward slash)

Proceeding characters must be...

- 0..9
- a..z, A..Z
- _ (underscore)
- / (forward slash)

Graph Resource Names (Continued...)

Four types:

1. base - used for node names (cannot have '/' or '∼')
2. relative (default)
3. global - start with a '/' [5]
4. private - used for parameter passing (starts with a '∼')

[5] should be avoided

Graph Resource Names Examples

```
ros::init( argc, argv, "node_name" );
ros::NodeHandle n( "sdsmt" ); //relative name
ros::Subscriber sub = n.subscribe( "gps_data", ... );
```

Node's Name:

- node_name (base name)

Node's Topic Subscription:

- /sdsmt/gps_data

Package Resource Names

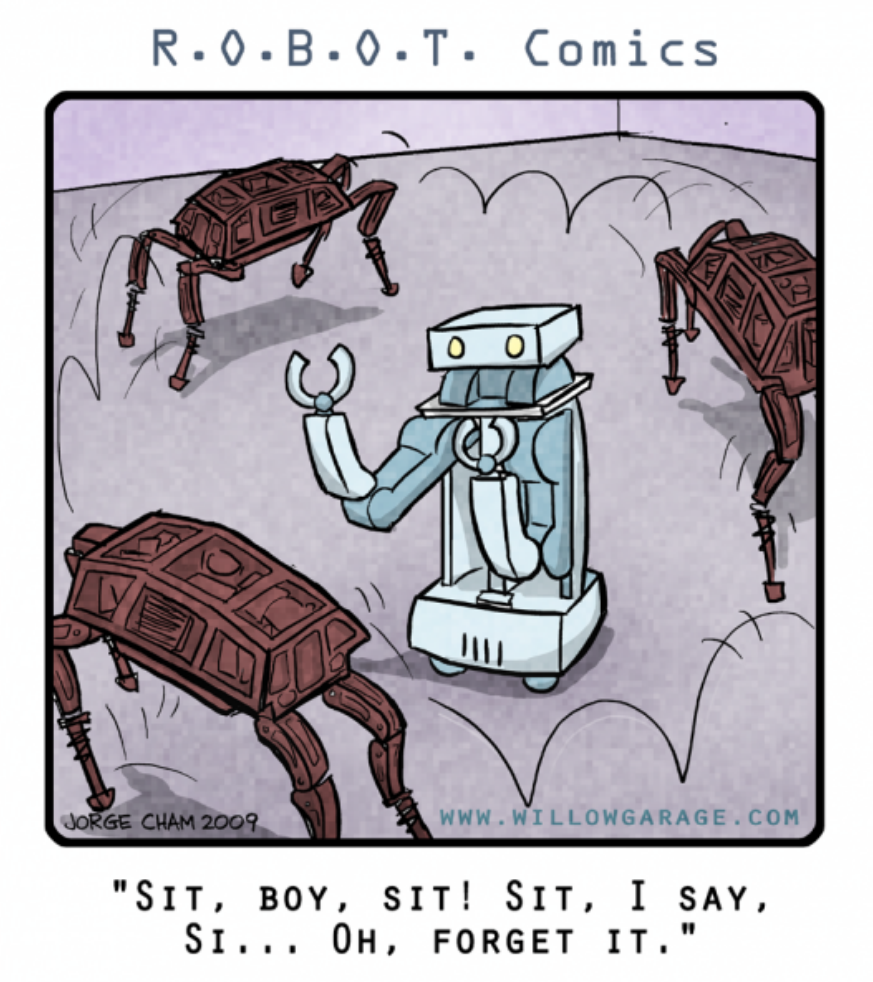
Refers to Packages and Stacks

Beware of Duplicate Names

Valid Package Resource Names

1. First character must start with...
   - a..z, A..Z
2. Procedding characters must be...
   - 0..9
   - a..z, A..Z
   - _ (underscore)
   - / (forward slash)
3. There can be at most one forward slash '/'.

Network Distribution



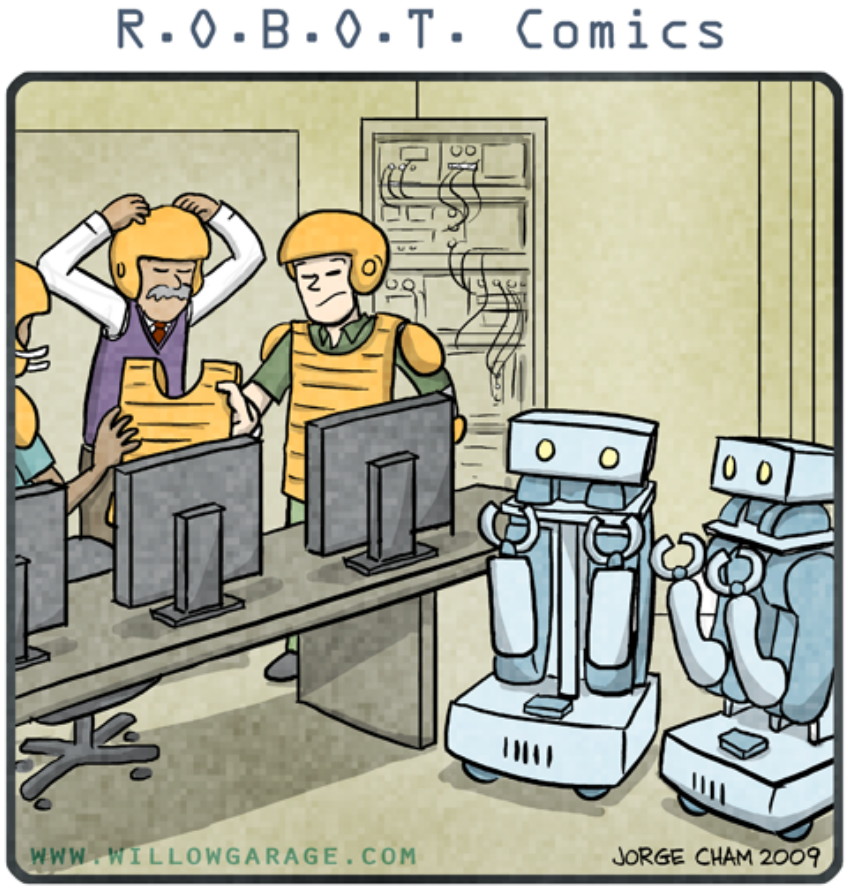
ROS Environment Variables

- ROS_MASTER_URI
  - A required setting that tells nodes where they can locate the master.
  - "The Server" IP address with port OR Name with port.
  - Required on both systems.
  - Usage:
    - export ROS_MASTER_URI=http://<server IP>:<port>
      OR
    - export ROS_MASTER_URI=http://<server name>:<port>
- ROS_IP
  - IP address of the current machine.
  - Usage:
    - export ROS_IP=<current machine IP>

ROS Environment Variables (Continued...)

- ROS_HOSTNAME
  - Specifying a host name on the current machine.
  - Usage:
    - export ROS_HOSTNAME=<name of current machine>
- ROS_NAMESPACE
  - Pushes a Node into a namespace.
  - Usage:
    - export ROS_NAMESPACE=<name of namespace>

Demonstration via Stage



Getting ROS-Stage

The best approach is to install the full desktop version of ROS first.

This will install Stage source, which then can be built.

Next, load in some sample files.

Finally, load a control program

ROS Setup - After a successful installation of ROS...

In your home directory:

```
mkdir ros
roscd
cd ..
sudo gedit setup.sh
```

In this file, we want to add a few lines. At the top, after the first line, add

```
export ROS_HOME=${HOME}/ros"
```

At the line with

```
ROS_PACKAGE_PATH
```

We need to concatenate the string with

```
:${ROS_HOME}
```

Save and exit, then type: `. setup.bash`

ROS

In a terminal window or tab, use these commands to start ROS:

```
export ROS_NAMESPACE=smp1
roscore
```

Stage

Grab the maze files from the homework and place them in a subdirectory

```
cd ~/ros
mkdir maze
```

save all of the maze files into `/ros/maze`.

Build Stage

```
export ROS_NAMESPACE=smp1
rosmake stage
rosrun stage stageros ~/ros/maze/maze.world
```

See what ROS is running

```
rostopic list
```

You should see a list of items running.

The exported variable sets an environment variable.

ROS treats all hardware like UNIX, everything is accessed via the filesystem.

So, it places the simulated robot under the name smp1 - this name is the mountpoint in the virtual filesystem.

A control code example

To download the SRS Basestation example:

```
svn co http://trac.mcs.sdsmt.edu/repos/commonbotcode
       #The code is now in your home directory.
cd commonbotcode
mv basestation ~/ros/
cd ~/ros
rospack profile
rosmake control_panel_v3 --rosdep-install
rosrun control_panel_v3 control_panel
```

One way to control the robot is with the SRS Basestation. This is currently just an alpha release and does not do much. You can control the robot with "w", "a", "s", "d" letters for motion.

Homework

Should I start on the homework now?

Or can I put this off until next week?

Homework

Let's think about this for a minute .....

... thinking ....

START NOW

Example - Stage

```
int main(int argc, char **argv)
{
  char c;
  set_conio_terminal_mode();

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

  ros::NodeHandle nh; //node handle
  ros::Publisher control_pub; //Publisher variable
  geometry_msgs::Twist cmd_vel;

  control_pub =
        nh.advertise<geometry_msgs::Twist>("cmd_vel",
        1);

  ros::Rate loop_rate(20);

  //At default we want the message to do nothing.
  cmd_vel.linear.x = 0.0; //ranges from [-1, 1]
        (negative to move reverse, positive to move
        forward)
  cmd_vel.linear.y = 0.0;
  cmd_vel.linear.z = 0.0;
  cmd_vel.angular.x = 0.0;
  cmd_vel.angular.y = 0.0;
  cmd_vel.angular.z = 0.0; //ranges from [-1, 1]
        (negative to turn left, positive to turn
        right)
```

```
  while (ros::ok())
    {
      cmd_vel.linear.x = 0.0;
      cmd_vel.angular.z = 0.0;
      if(kbhit()) {
         c = getchar();
         if(c =='w') cmd_vel.linear.x = 0.5;
         if(c =='s') cmd_vel.linear.x = -0.5;
         if(c =='a') cmd_vel.angular.z = 0.5;
         if(c =='d') cmd_vel.angular.z = -0.5;
         if(c =='p') return 0;
      }
      control_pub.publish(cmd_vel);
      ros::spinOnce();

      loop_rate.sleep();

    }


  return 0;
}
```

Laser example

```
#include "ros/ros.h"
#include "geometry_msgs/Twist.h" //this is to use
      the Twist message
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <termios.h>
#include <sensor_msgs/LaserScan.h>
#include <iostream>

using namespace std;

void laserCallback( const sensor_msgs::LaserScan&
      scan );

bool moveL=true, moveR=true, moveF=true;

/* Function: laserCallback
 * Author: Ryan W. Housh
 * Description:
 *         This function is used to grab the
       information from scan node
 *         and determines if the user is driving too
       close to an obstacle.
 */
```

```
void laserCallback( const sensor_msgs::LaserScan&
      scan ){
   double blockL = 0.5,
          blockR = 0.5,
          blockF = 0.8;

   for( int i = 0; i < scan.ranges.size(); i++ )
   {
      // Checks the laser scans up to 90 degrees
            from 0 degree,
      // which starts from right rear (rear right
            yellow mark on LiDAR).
      // Also we want to omit the scan ranges that
            are below range minimum.
      if( i < scan.ranges.size()/3.0 &&
            scan.ranges[i] > scan.range_min ){
         if( scan.ranges[i] <= blockR )
            moveR = false;
         else
            moveR = true;
      }
```

Laser Example

```
        //Checks laser scans from 91 degrees to 180
              degrees (forward yellow mark on LiDAR).
        else if( i < 2.0*scan.ranges.size()/3.0 &&
              scan.ranges[i] > scan.range_min ){
            if( scan.ranges[i] <= blockF )
                moveF = false;
            else
                moveF = true;
        }
        //Checks laser scans from 181 degrees to 270
              degrees (rear left yellow mark on
              LiDAR).
        else if( i < scan.ranges.size() &&
              scan.ranges[i] > scan.range_min ){
            if( scan.ranges[i] <= blockL )
                moveL = false;
            else
                moveL = true;
        }

        // printf("%f %f %f\n", scan.ranges[0],
              scan.ranges[scan.ranges.size()/2],
              scan.ranges[scan.ranges.size() - 1]);
    }
}
```

```
int main(int argc, char **argv)
{
  char c;
  set_conio_terminal_mode();

  ros::init(argc, argv, "talker");
  ros::NodeHandle nh, n3; //node handle
  ros::Publisher control_pub; //Publisher variable
  geometry_msgs::Twist cmd_vel;

  control_pub =
        nh.advertise<geometry_msgs::Twist>("cmd_vel",
        1);
  ros::Subscriber sub2 = n3.subscribe( "base_scan",
        5, laserCallback );
    ros::Rate loop_rate(20);

  //At default we want the message to do nothing.
  cmd_vel.linear.x = 0.0; //ranges from [-1, 1]
        (negative to move reverse, positive to move
        forward)
  cmd_vel.linear.y = 0.0;
  cmd_vel.linear.z = 0.0;
  cmd_vel.angular.x = 0.0;
  cmd_vel.angular.y = 0.0;
  cmd_vel.angular.z = 0.0; //ranges from [-1, 1]
        (negative to turn left, positive to turn
        right)
```

Laser Example

```
while (ros::ok())
   {
     ros::spinOnce();
     cmd_vel.linear.x = 0.0;
     cmd_vel.angular.z = 0.0;
     if(kbhit()) {
    c = getchar();
    if(c =='w' && moveF) cmd_vel.linear.x = 0.5;
    if(c =='s') cmd_vel.linear.x = -0.5;
    if(c =='a' && moveL) cmd_vel.angular.z = 0.5;
        if(c =='d' && moveR) cmd_vel.angular.z = -0.5;
        if(c =='p') return 0;
      }
     control_pub.publish(cmd_vel);
     loop_rate.sleep();
   }
  return 0;
}
```

ROS-Stage

A simulation is composed of three parts (Jenny Owen):

- Your code. This talks to ROS.
- ROS. This takes your code and sends instructions to a robot. From the robot it gets sensor data and sends it to your code.
- Stage. Stage interfaces with ROS in the same way as a robots hardware would. It receives instructions from ROS and moves a simulated robot in a simulated world, it gets sensor data from the robot in the simulation and sends this to ROS.

ROS-Stage

In ROS/Stage there are 2 kinds of files that you need to understand to get going with ROS/Stage:

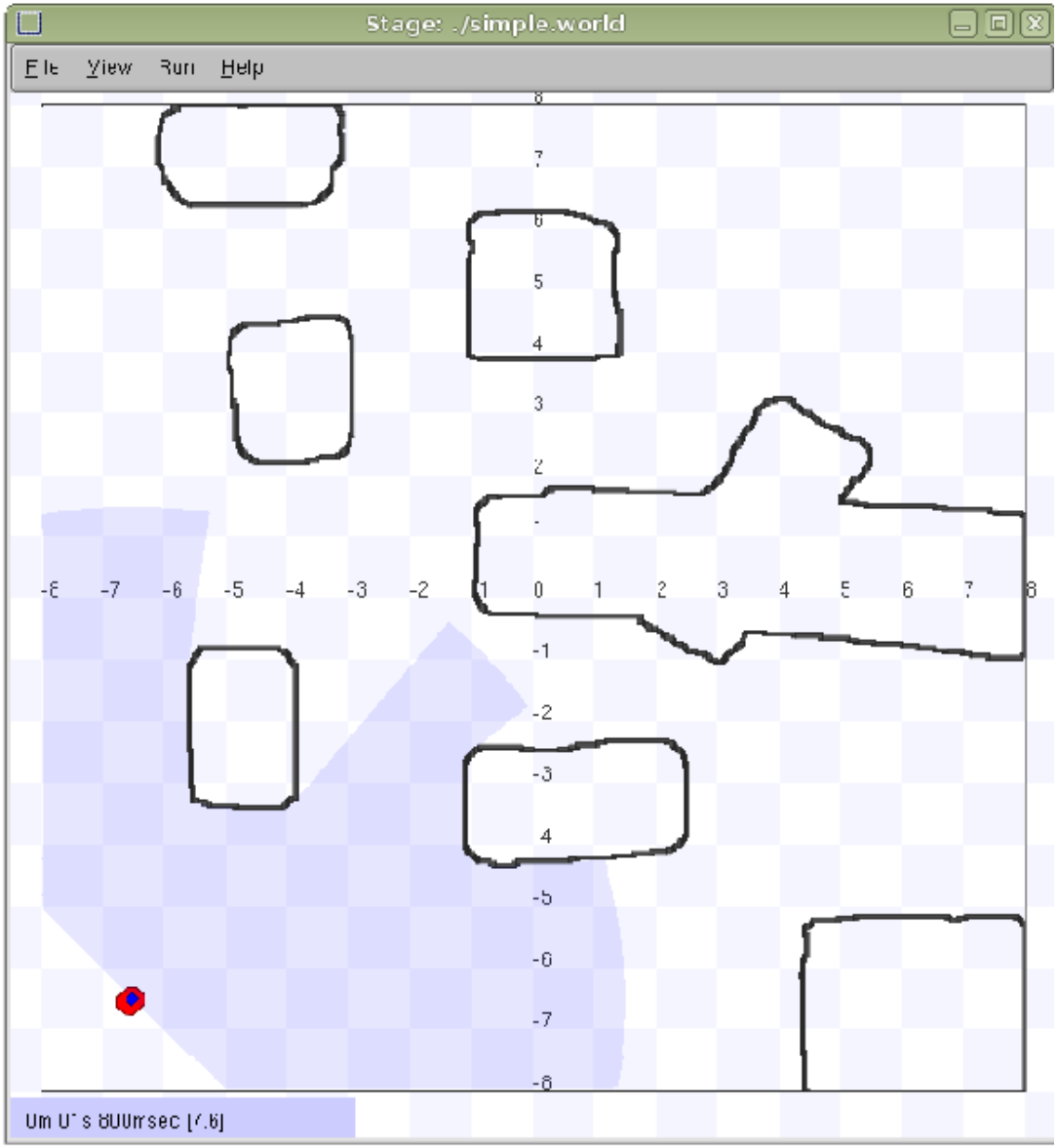
- .world file
- .inc (include) file

ROS-Stage

The .world file tells ROS/Stage what things are available to put in the world. In this file you describe your robot, any items which populate the world and the layout of the world.

The .inc file follows the same syntax and format of a .world file but it can be included. One can place common world file information here. Helps with code reuse and common robot descriptions.

Simple World



World File

A worldfile is basically just a list of models that describes all the stuff in the simulation. This includes the basic environment, robots and other objects. The basic type of model is called model, and you define a model using the following syntax:

```
define model_name model
(
# parameters
)
```

This tells ROS/Stage that you are defining a model which you have called model_name, and all the stuff in the round brackets are parameters of the model. Call this `empty.world`.

Floorplan

To begin to understand ROS/Stage model parameters, lets look at the **map.inc** file that comes with Stage, this contains the floorplan model, which is used to describe the basic environment of the simulation (i.e. walls the robots can bump into):

```
define floorplan model
(
  # sombre, sensible, artistic
  color "gray30"
  # most maps will need a bounding box
  boundary 1
  gui_nose 0
  gui_grid 0
  gui_move 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
  laser_return 1
)
```

Floorplan parameters

- **color**: Tells ROS/Stage what colour to render this model, in this case it is going to be a shade of grey.
- **boundary**: Whether or not there is a bounding box around the model. This is an example of a binary parameter, which means the if the number next to it is 0 then it is false, if it is 1 or over then its true. So here we DO have a bounding box around our "map" model so the robot can't wander out of our map.
- **gui_nose**: this tells ROS/Stage that it should indicate which way the model is facing.
- **gui_grid**: this will superimpose a grid over the model.
- **gui_move**: this indicates whether it should be possible to drag and drop the model. Here it is 0, so you cannot move the map model once ROS/Stage has been run.

Floorplan parameters

- **gui_outline**: indicates whether or not the model should be outlined. This makes no difference to a map, but it can be useful when making models of items within the world.
- **fiducial_return**: any parameter of the form some sensor return describes how that kind of sensor should react to the model. Fiducial is a kind of robot sensor which will be described later. Setting fiducial_return to 0 means that the map cannot be detected by a fiducial sensor.
- **gripper_return**: Like fiducial_return, gripper_return tells ROS/Stage that your model can be detected by the relevant sensor, i.e. it can be gripped by a gripper. Here gripper_return is set to 0 so the map cannot be gripped by a gripper.

Map Inc

To make use of the map.inc file we put the following code into our world file:

```
include "map.inc"
```

This inserts the map.inc file into our world file where the include line is.

Map Model modification

```
floorplan
(
   bitmap "bitmaps/helloworld.png"
   size [12 5 1]
)
```

What this means is that we are using the model floorplan, and making some extra definitions; both bitmap and size are parameters of a ROS/Stage model.

World File

Put all together:

```
include "map.inc"
# configure the GUI window
window
(
   size [700.000 700.000]
   scale 41
)
# load an environment bitmap
floorplan
(
   bitmap "bitmaps/cave.png"
   size [15 15 1.5]
)
```

References

- Gerkey, Brian P., et al. "Most Valuable Player: A Robot Device Server for Distributed Control." *IEEE International Conference on Intelligent Robots and Systems*, 2001
- Willow Garage. "ros.org" *www.ros.org/wiki*, 2011
- Wikipedia. "ROS (Robot Operating System)." *en.wikipedia.org/wiki/ROS_(Robot_Operating_System)*, 2011