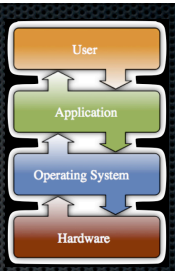
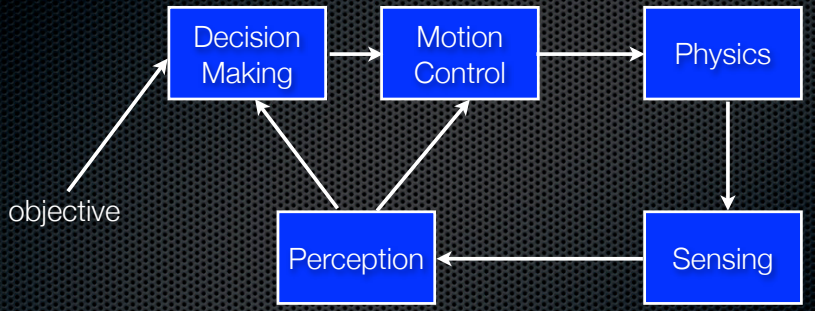
Robot middleware (ROS)

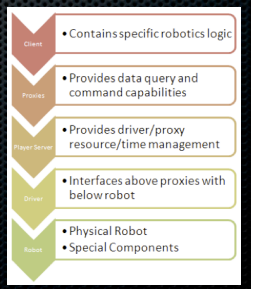


Robot control loop



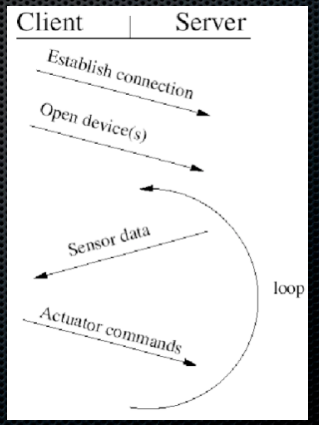
Robot middleware

- Provide an abstraction layer and drivers between computation and embodiment
  - Analogy: hardware abstraction layer in an operating system
- A number of open-source frameworks
  - Player: client/server model (1999-2009)
  - ROS: peer-to-peer model (2008-date)



Player (playerstage.sourceforge.net)

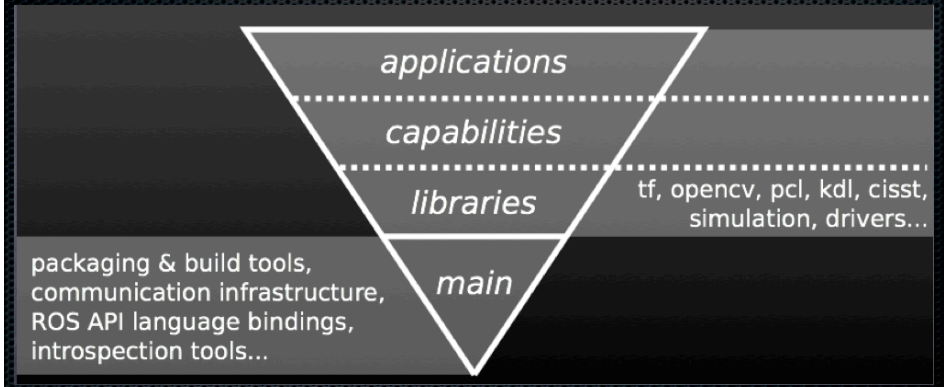
- Client-server architecture
- Publish-subscript messaging
- Server runs on robot
- Clients (applications) access server
  - subscribe to robot proxies



ROS (ros.org)

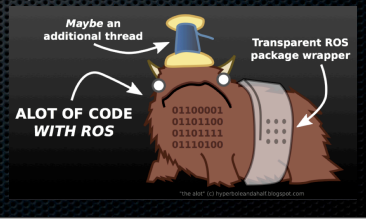
- Robot operating system
  - Created by Morgan Quigley and Willow Garage
- Peer-to-peer architecture over networks
- Software function modularized as ROS
  - Run-time system: nodes communicate over IP
  - Packaging system: modes organized into distributable packages

ROS as a development ecosystem



ROS Component

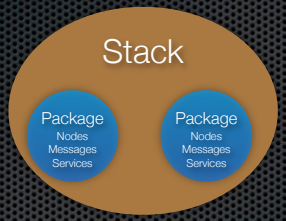
- ROS node: core element
  - Defined in terms of messages and services
- Nodes subscribe to and publish topics as a stream
- Nodes are wrappers around code
- Package (node, message,services)



ROS packaging system

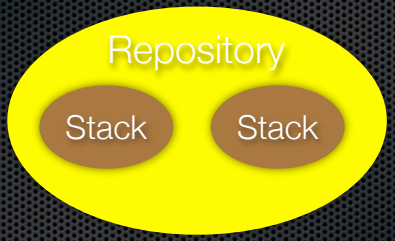
Individual packages are grouped together into stacks

ROS packaging system



Multiple stacks become a repository

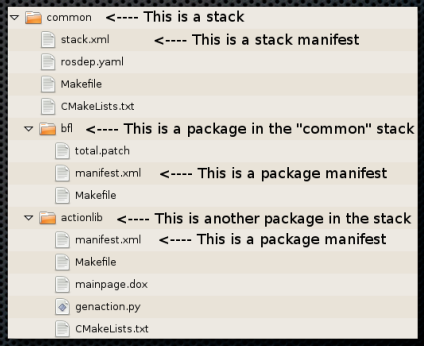
ROS packaging system

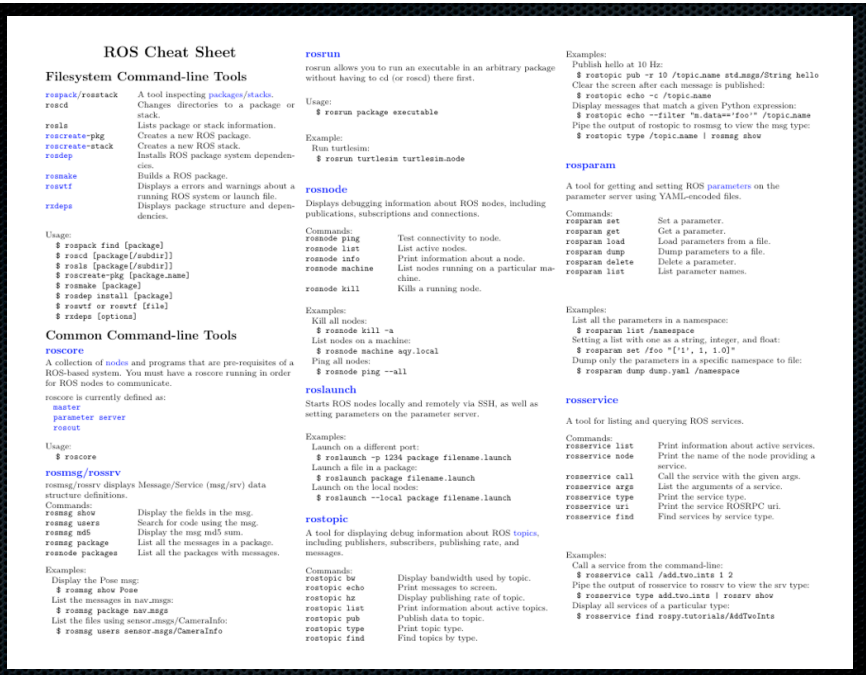


Multiple stacks become a repository

ROS operation

- Key commands are command-line driven
- Packages are specially constructed directories
- Nodes end up written in Python or C++ (other options, but not as common)





ROS Cheat Sheet

Filesystem Command-line Tools

rospack/rosstack	A tool inspecting packages/stacks.
roscd	Changes directories to a package or stack.
rosls	Lists package or stack information.
roscreate-pkg	Creates a new ROS package.
roscreate-stack	Creates a new ROS stack.
rosdep	Installs ROS package system dependencies.
rosmake	Builds a ROS package.
roswtf	Displays a errors and warnings about a running ROS system or launch file.
rxdeps	Displays package structure and dependencies.

Usage:

```
$ rospack find [package]
$ roscd [package[/subdir]]
$ rosls [package[/subdir]]
$ roscreate-pkg [package_name]
$ rosmake [package]
$ rosdep install [package]
$ roswtf or roswtf [file]
$ rxdeps [options]
```

Common Command-line Tools

roscore

A collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate.

roscore is currently defined as:

```
master
parameter server
rosout
```

Usage:

```
$ roscore
```

rosmsg/rossrv

rosmsg/rossrv displays Message/Service (msg/srv) data structure definitions.

Commands:

rosmsg show	Display the fields in the msg.
rosmsg users	Search for code using the msg.
rosmsg md5	Display the msg md5 sum.
rosmsg package	List all the messages in a package.
rosnode packages	List all the packages with messages.

Examples:

Display the Pose msg:
```
$ rosmsg show Pose
```
List the messages in nav_msgs:
```
$ rosmsg package nav_msgs
```
List the files using sensor_msgs/CameraInfo:
```
$ rosmsg users sensor_msgs/CameraInfo
```

rosrun

rosrun allows you to run an executable in an arbitrary package without having to cd (or roscd) there first.

Usage:
```
$ rosrun package executable
```

Example:

Run turtlesim:
```
$ rosrun turtlesim turtlesim_node
```

rosnode

Displays debugging information about ROS nodes, including publications, subscriptions and connections.

Commands:

rosnode ping	Test connectivity to node.
rosnode list	List active nodes.
rosnode info	Print information about a node.
rosnode machine	List nodes running on a particular machine.
rosnode kill	Kills a running node.

Examples:

Kill all nodes:
```
$ rosnode kill -a
```
List nodes on a machine:
```
$ rosnode machine aqy.local
```
Ping all nodes:
```
$ rosnode ping --all
```

roslaunch

Starts ROS nodes locally and remotely via SSH, as well as setting parameters on the parameter server.

Examples:

Launch on a different port:
```
$ roslaunch -p 1234 package filename.launch
```
Launch a file in a package:
```
$ roslaunch package filename.launch
```
Launch on the local nodes:
```
$ roslaunch --local package filename.launch
```

rostopic

A tool for displaying debug information about ROS topics, including publishers, subscribers, publishing rate, and messages.

Commands:

rostopic bw	Display bandwidth used by topic.
rostopic echo	Print messages to screen.
rostopic hz	Display publishing rate of topic.
rostopic list	Print information about active topics.
rostopic pub	Publish data to topic.
rostopic type	Print topic type.
rostopic find	Find topics by type.

Examples:

Publish hello at 10 Hz:
```
$ rostopic pub -r 10 /topic_name std_msgs/String hello
```
Clear the screen after each message is published:
```
$ rostopic echo -c /topic_name
```
Display messages that match a given Python expression:
```
$ rostopic echo --filter "m.data=='foo'" /topic_name
```
Pipe the output of rostopic to rosmsg to view the msg type:
```
$ rostopic type /topic_name | rosmsg show
```

rosparam

A tool for getting and setting ROS parameters on the parameter server using YAML-encoded files.

Commands:

rosparam set	Set a parameter.
rosparam get	Get a parameter.
rosparam load	Load parameters from a file.
rosparam dump	Dump parameters to a file.
rosparam delete	Delete a parameter.
rosparam list	List parameter names.

Examples:

List all the parameters in a namespace:
```
$ rosparam list /namespace
```
Setting a list with one as a string, integer, and float:
```
$ rosparam set /foo "['1', 1, 1.0]"
```
Dump only the parameters in a specific namespace to file:
```
$ rosparam dump dump.yaml /namespace
```

rosservice

A tool for listing and querying ROS services.

Commands:

rosservice list	Print information about active services.
rosservice node	Print the name of the node providing a service.
rosservice call	Call the service with the given args.
rosservice args	List the arguments of a service.
rosservice type	Print the service type.
rosservice uri	Print the service ROSRPC uri.
rosservice find	Find services by service type.

Examples:

Call a service from the command-line:
```
$ rosservice call /add_two_ints 1 2
```
Pipe the output of rosservice to rosrv to view the srv type:
```
$ rosservice type add_two_ints | rosrv show
```
Display all services of a particular type:
```
$ rosservice find rospy_tutorials/AddTwoInts
```

ROS tutorials

- Many online ones.
- Just to be different, lets do one differently
- Want to create a package to deal with the superscouts
- Could do this in lots of ways
  - Build on the robot, then do all control in ROS
  - Build a wrapper for the current software <- do this here

Superscouts

- Run a reactive control architecture
- Robot has
  - a pose (x,y,theta)
  - a local goal (x,y,theta)
- Continually tries to maintain the goal while avoiding obstacles
- Uses a reactive control architecture to do this

So lets build a package..

- (1) Need to define a package to hold things
- roscreate-pkg superscout std_msg rospy roscpp
  - Creates a package called superscout
  - Defines standard messages, and provides building structures for Python and C++
- rosmake superscout
  - Builds the package (almost nothing at this point)

Defining messages

- Are defined in msg subdirectory of package
- Text file .msg
- Lines define fields in a message

```
float 32 x
float32 y
float32 theta
```

Pose.msg

Makefile
#rosbuild_genmsg()

Will define a type Pose()

Defining a service

- Defines a protocol between nodes
- Defined in terms of messages (can be unidirectional)
- Defined in a text file .srv in a directory ‘srv’

```
# get pose
---
Pose state
```

GetPose.srv

```
Post state
---
```

SetPose.srv

```
Makefile
#rosbuild_gensrv()
```

Now the magic

- ROS will deal with serialization, etc. It will define types based on the msg files and defines the node-node protocols based on the srv files
- Your job is to write the code to do this.
  - Examples here in Python

rospy.init_node(‘node-name’)
- create node

rospy.spin()
- sleep

rospy.Service(‘service’,serviceName, handler)
- define the service<->handler linkage
- serviceName is defined by .srv file
- service is the ‘name’ the service will be defined as

```
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Defined in .srv file

A sample from the ROS tutorials

```python
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Type defined in .msg file

A sample from the ROS tutorials

```python
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning [%s + %s = %s]"%(req.a, req.b, (req.a + req.b))
    return AddTwoIntsResponse(req.a + req.b)

def add_two_ints_server():
    rospy.init_node('add_two_ints_server')
    s = rospy.Service('add_two_ints', AddTwoInts, handle_add_two_ints)
    print "Ready to add two ints."
    rospy.spin()

if __name__ == "__main__":
    add_two_ints_server()
```

Servicing being provided

A sample from the ROS tutorials

Requesting a service

- my_package/src/Foo.srv ->
  - my_package.srv.Foo
  - my_package.srv.FooRequest
  - my_package.srv.FooResponse

rospy.ServiceProxy('service_name', my_package.srv.Foo)

```python
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')

import sys

import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Wait for this service

```python
#!/usr/bin/env python
import roslib; roslib.load_manifest('beginner_tutorials')

import sys

import rospy
from beginner_tutorials.srv import *

def add_two_ints_client(x, y):
    rospy.wait_for_service('add_two_ints')
    try:
        add_two_ints = rospy.ServiceProxy('add_two_ints', AddTwoInts)
        resp1 = add_two_ints(x, y)
        return resp1.sum
    except rospy.ServiceException, e:
        print "Service call failed: %s"%e

def usage():
    return "%s [x y]"%sys.argv[0]

if __name__ == "__main__":
    if len(sys.argv) == 3:
        x = int(sys.argv[1])
        y = int(sys.argv[2])
    else:
        print usage()
        sys.exit(1)
    print "Requesting %s+%s"%(x, y)
    print "%s + %s = %s"%(x, y, add_two_ints_client(x, y))
```

Link to this