Lisp Tutorial

Jan Winkler

Institute for Artificial Intelligence
Universität Bremen

May 12, 2014
Summer Term

Today's Schedule

- Macros in Lisp
  - What are Macros (difference between Lisp and other languages)
  - Usecases
  - Writing Macros
  - Pitfalls
- ROS Basics
  - Architecture
  - Tools
  - References
- Tutorials on ROSLisp

Macros in Lisp

What a Lisp Macro is not

Macros in other languages commonly follow a different concept:

- **C++**: `#define x_plus_5(x) x + 5`

  Simple Example:
  ```
  cout << (x_plus_5(10) / 2) << endl;
  ```

  Macro Expansion:
  ```
  cout << (10 + 5 / 2) << endl;
  ```

  Expected Result: 7.5
  Actual Result: 12.5

C++-Macros can be confusing and misleading (leads to *macrophobia*).
They are just **source level text replacements**.

Macros in Lisp

What is a Lisp Macro then? – A Motivating Example

Macros in Lisp also replace code with code, but need to be **inherently complete functions**. Starter Example:

Simple Macro Example

```
> (defmacro x-plus-5 (x)
    `(+ x 5))

> (/ (x-plus-5 10) 2)
```

Macro Expansion

```
> (macroexpand '(/ (x-plus-5 10) 2))
(/ (+ 10 5) 2)
```

Macros in Lisp

Standard Lisp Macro – `when`

The default `if` implementation has a major shortcoming:

- The `then` and `else` clauses are limited to one form

The standard `when` macro resolves this:

Sample `when` implementation

```
> (defmacro when (condition &body forms)
    `(if condition
         (progn
           ,@forms)))
```

More standard Macros: `http://www.gigamonkeys.com/book/macros-standard-control-constructs.html`

Macros in Lisp

Lisp Macros Extend the Language – `&body`

New Macro Parameter specifier: `&body`

Practically the same as `&rest`, but can be a hint to smart editors (code purpose).

```
(defmacro name lambda-list &rest body)
```

All body forms given to the macro call go into 'body'.

Macros in Lisp

Lisp Macros Extend the Language – &body

Example:

The print-body macro

```
> (defmacro print-body (&body body)
    (dotimes (i (length body))
      (format t "~a:␣~a~%" i (nth i body))))

> (print-body
    (test)
    (format t "~a~%" 5))
0: (TEST)
1: (FORMAT T ~a~% 5)
```

Macros in Lisp

Lisp Macros Extend the Language – The Comma ‘ , ‘ Operator

Does not only work in Macros:

Backquote

```
> (let ((a 5))
    `(test a ,a))
Result: (TEST A 5)
```

The result is a **list**. Only expressions with a comma are evaluated. Only works in backquote'd lists, not in high comma lists:

High Comma

```
> (let ((a 5))
    '(test a ,a))
Failure: Comma not inside backquote
```

Macros in Lisp

Lisp Macros Extend the Language – Splicing with ,@

Inserting a forms list as single form instances:

Splice

```
> (let ((lst '(a b c d e f)))
    `(,@lst))
```

More meaningful example:

Splice

```
> (defmacro my-map (parameters &body body)
    `(let (,@(mapcar (lambda (x y)
                       `(,x ,y))
                     ',parameters
                     (loop for i from 0 below 10
                           collect i)))
       ,@body))
```

Macros in Lisp

Lisp Macros Extend the Language – An extended `let`

Assuming that the `let` environment does not allow multiple variables to be declared in one call, the following macro allows this (recursively):

Splice

```
> (defmacro my-let (definitions &body body)
    (if definitions
        `(let ((,(first (first definitions))
                ,(second (first definitions))))
           (my-let ,(rest definitions) ,@body))
        `(progn ,@body)))
```

Here, all presented techniques for macro writing are used:

- The &body parameter: Arbitrary forms to evaluate
- The ’‘’ backquote and comma operators: Partial evaluation in lists
- The ’@’ splice operator: Removing one list level

ROS – Robot Operating System

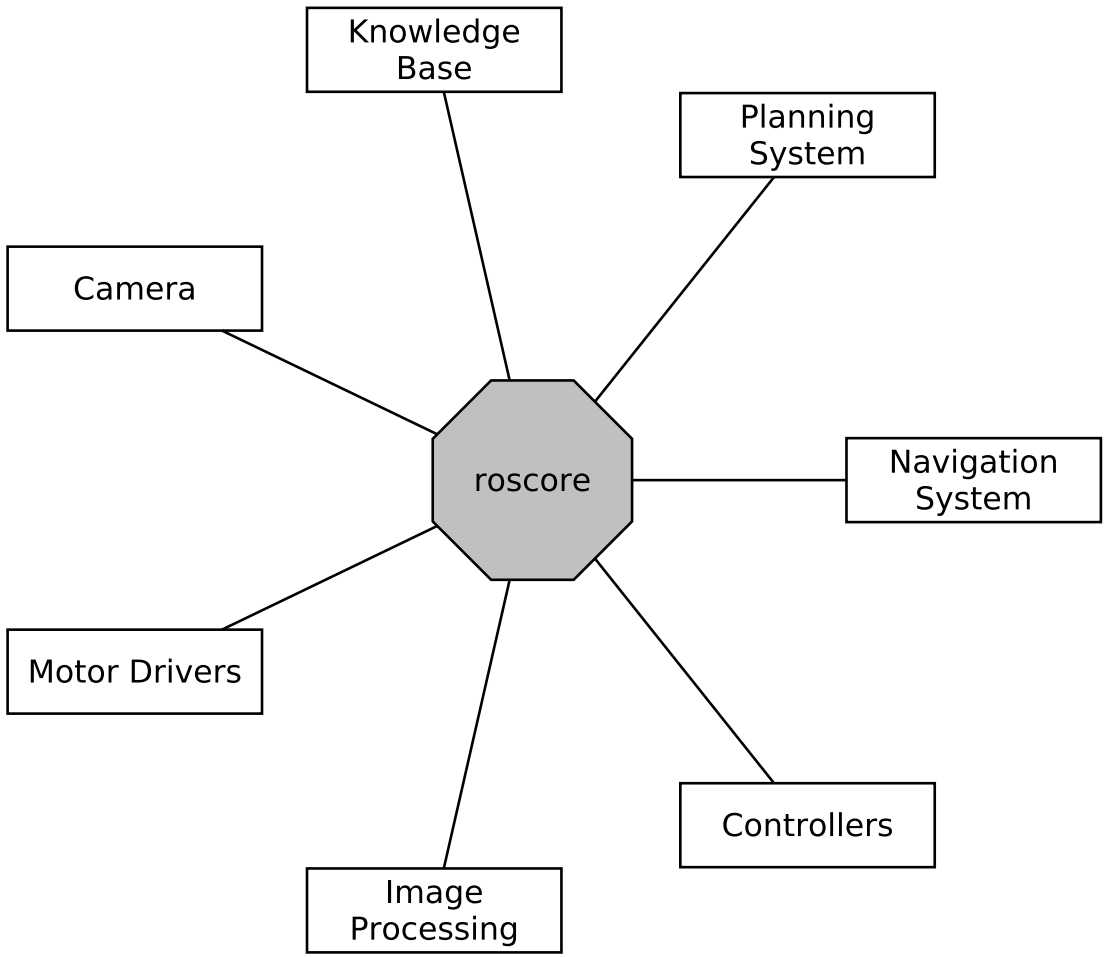
The Basic Idea

What the rough goals were and still are:

- Getting different programming languages together
- Distributing processing load over multiple computers
- Allowing different architectures to communicate (linux, windows, OS X, android, embedded, ...)
- Preventing researchers from inventing the wheel over and over again (writing middlewares)

ROS – Robot Operating System

Architecture – All Nodes Register with the Core



ROS – Robot Operating System

roscore

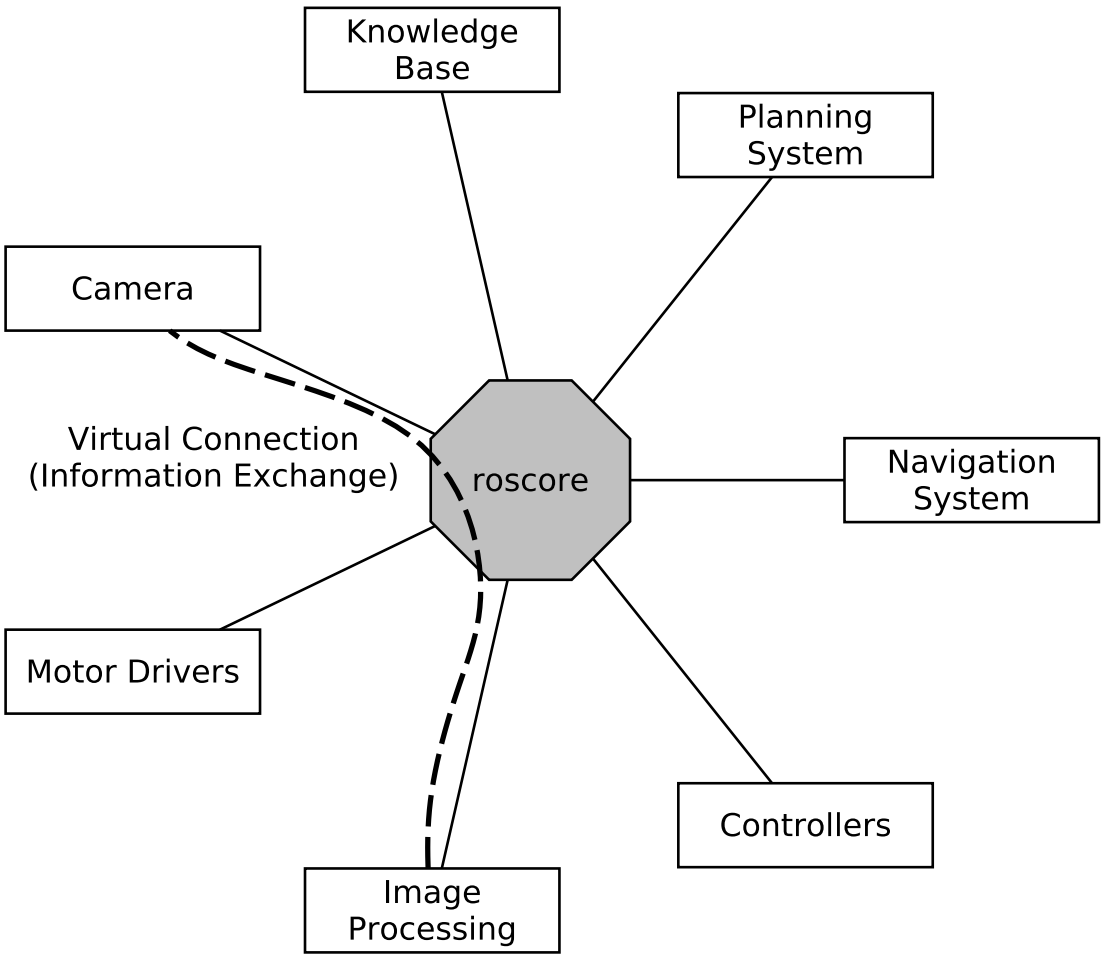
Central Component for Node Communication

Tasks:

- Bookkeeping of Node Information
  - Name, Host
  - Offered Services
  - Published, Subscribed Topics
- Message Exchange Initialization between Hosts

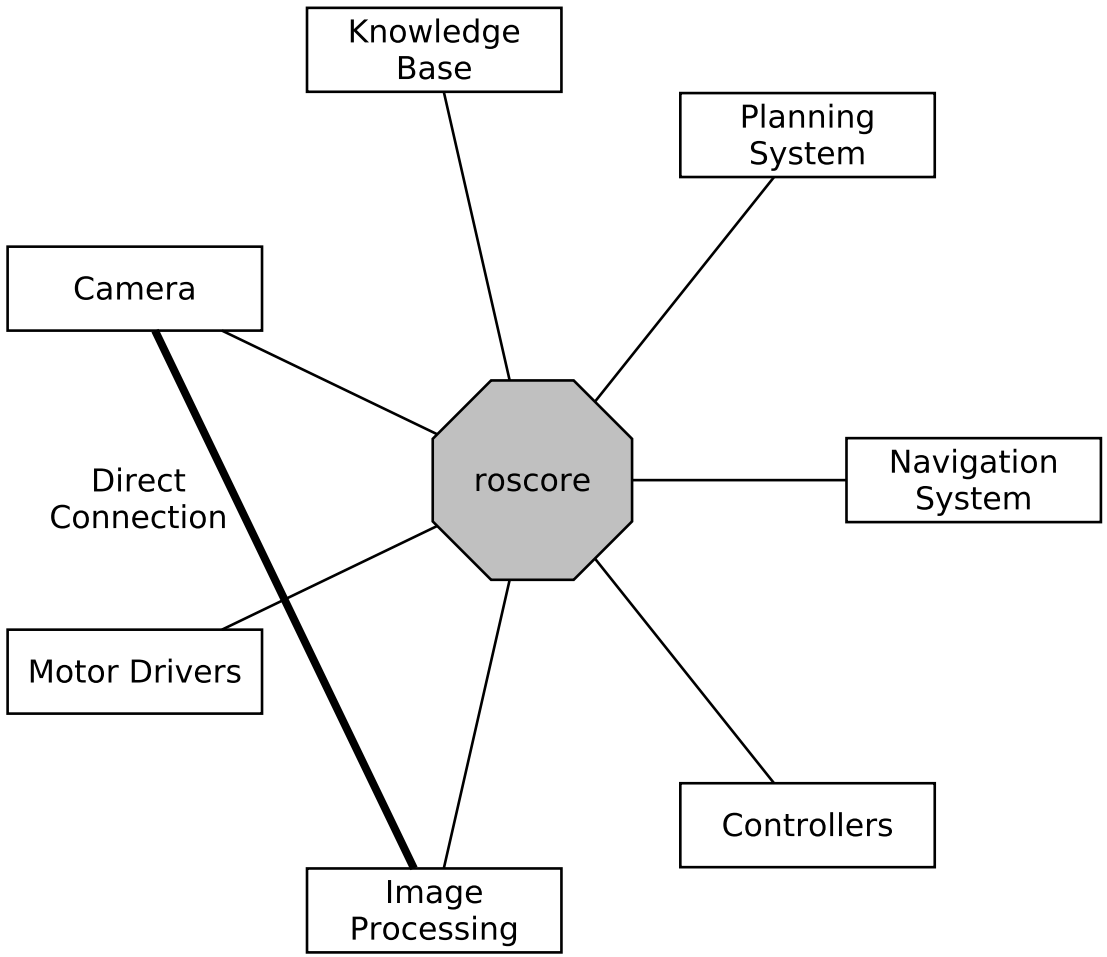
ROS – Robot Operating System

Architecture – Node Information Exchange through the Core



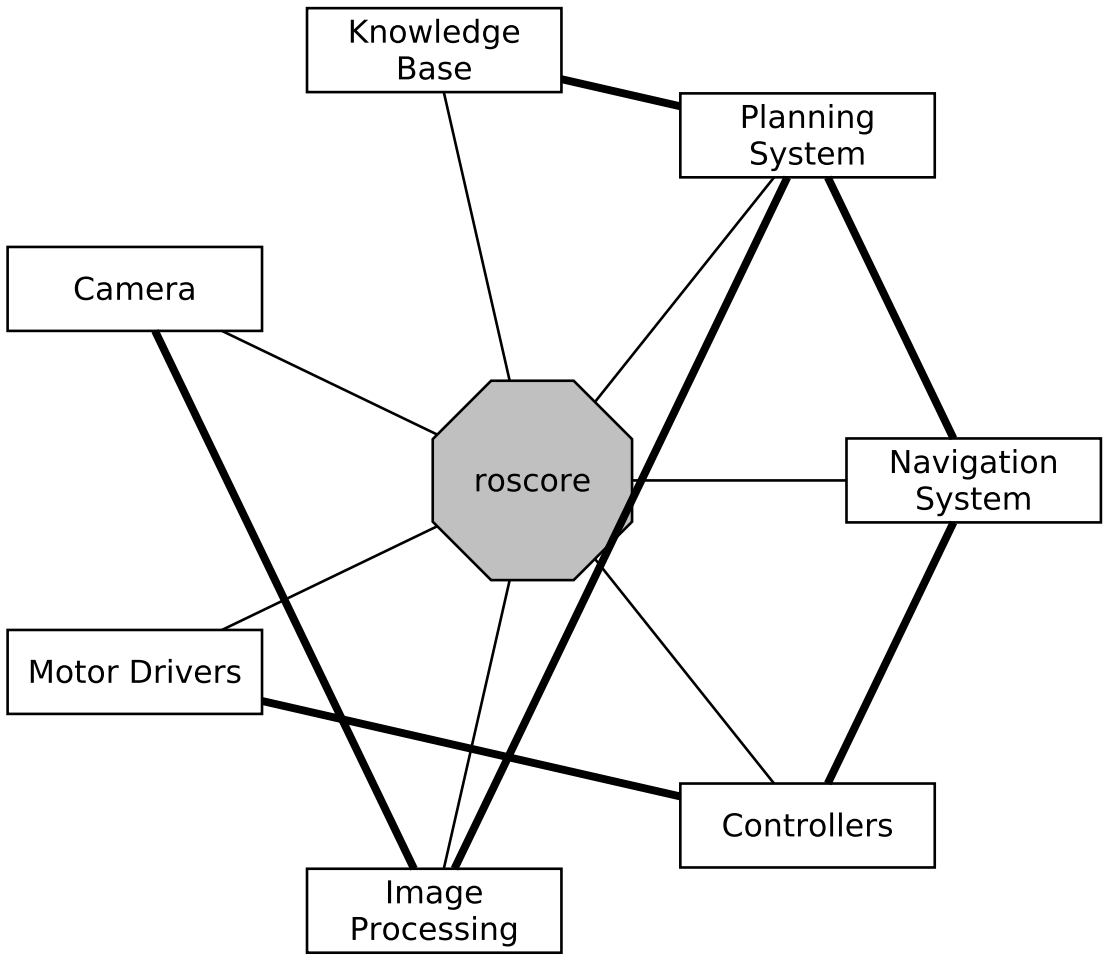
ROS – Robot Operating System

Architecture – Direct Network Connection for Data Exchange



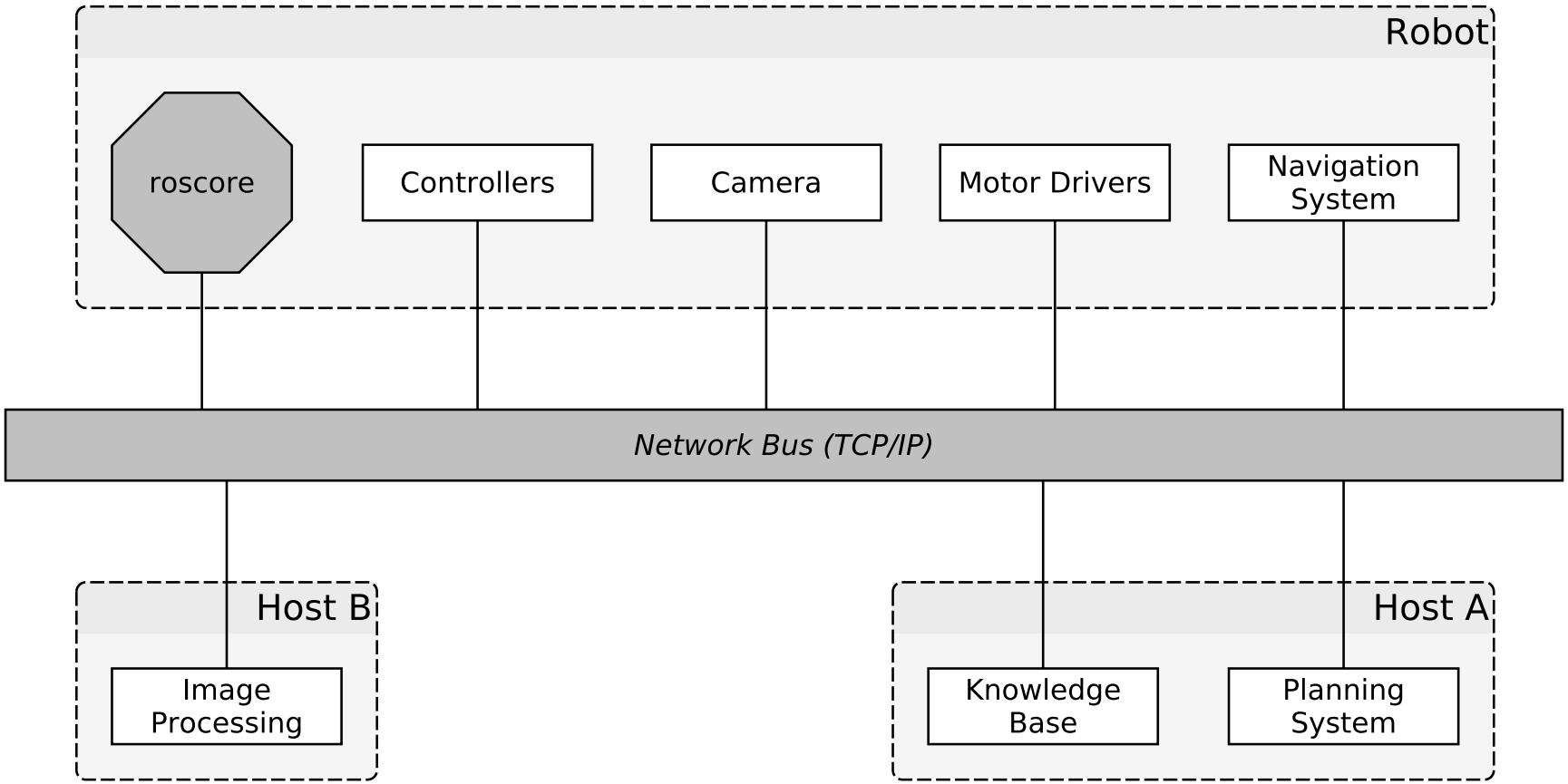
ROS – Robot Operating System

Architecture – Meshed Network Architecture



ROS – Robot Operating System

Architecture – Distributed Hosts on Network Bus



ROS – Robot Operating System

Tools

ROS offers a set of tools for development and debugging.

Syntax:

```
$ <tool-name> <command-name> [parameters]
```

ROS – Robot Operating System

Tools – rosnode

Operate on nodes on the current ROS Master. Able to:

- `ping`: Check if node is alive (or zombie)
- `list`: List all node names present on master
- `info`: Display information about node
  - Name, machine
  - Published and subscribed topics
  - Advertised services
- `machine`: List nodes on machine/machines in (ros-)network
- `kill`: Tell a running node to shut down
- `clean`: Let master clean up unreachable nodes

Central tool for ROS debugging and development

ROS – Robot Operating System

Tools – rostopic

Identifies topics of a running node. Able to:

- `bw`: Bandwith used by this topic
- `echo`: Raw output of messages on topic
- `find`: Find topic by (message-)type
- `hz`: Publishing rate (in Hertz) on topic
- `info`: Information (message-type, subscribers, publishers)
- `list`: List all (active) topics
- `pub`: Publish a message on a topic
- `type`: Print topic type

ROS – Robot Operating System

Tools – rosbag

Records and plays bag message sequences on topics. Most important:

- `record`: Records a number of topics
- `play`: Plays back a recorded file

Can be used to **loop** the playback.

Very useful for development and debugging when

- the same situation needs to be analyzed over and over again
- access to the real robot is restricted by time

ROS – Robot Operating System

Tools – rosservice

Similar to `rostopic`, but for ROS services. Most important:

- `list`: List all active ROS services on all nodes
- `find`: Find a service by its message type
- `call`: Call a service manually

A service message definition consists of two *normal* ROS messages:

- Request message
- Response message

Service calls are **blocking** (opposing concept: `actionlib`).

ROS – Robot Operating System

Tools – rosmsg, rossrv

List information about available ROS messages / service messages:

- `list`: List all defined (service) messages
- `show`: Show details about specific (service) messages

`rosmsg` and `rossrv` work alike.

Messages can be **nested** (but only different types). `rosmsg` and `rossrv` display the **expanded** message tree.

ROS – Robot Operating System
Bridging Different Programming Languages

ROS Nodes can currently be written in:

- C++ (`roscpp`)
- Python (`rospy`)
- Java (`rosjava`)
- **Lisp**: **Common Lisp** (`roslisp`), EUSLisp (`roseus`), Clojure, ...
- Lua (`roslua`)
- Matlab M-Code / Simulink
- Probably more...

Most commonly used at the moment: C++, Python, Java

ROS – Robot Operating System

Links, References, Tutorials

Look up more details on these websites:

- Basic ROS Tutorials: `http://wiki.ros.org/Tutorials/`
- ROSLisp Tutorials: `http://wiki.ros.org/roslisp`

ROSLisp
What is ROSLisp?

ROSLisp is the bridge between ROS and Lisp

Offers subscribing, publishing, communicating with other ROS nodes

From outside, is a transparent ROS node (you don't know from the outside in what language a node is written)

The rest for ROSLisp is interactive