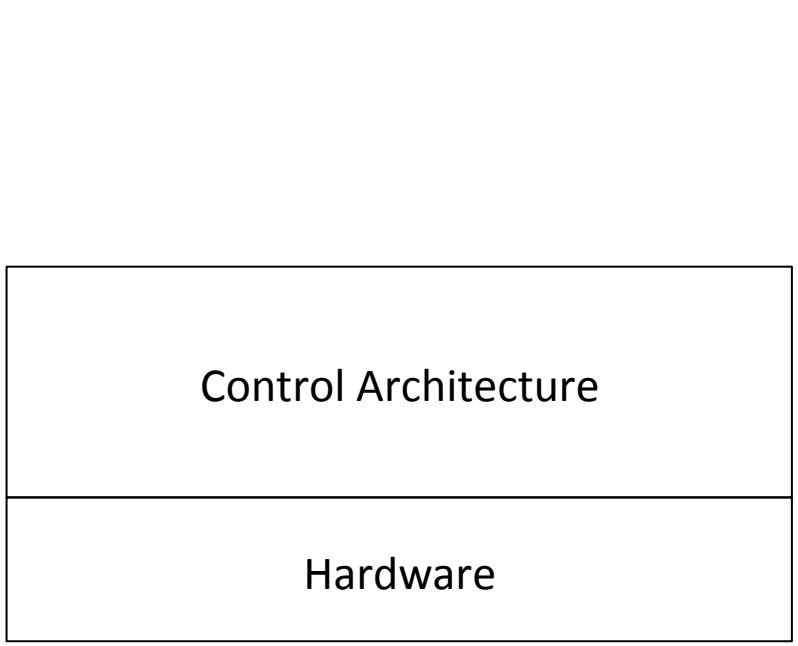
Introduction to ROS

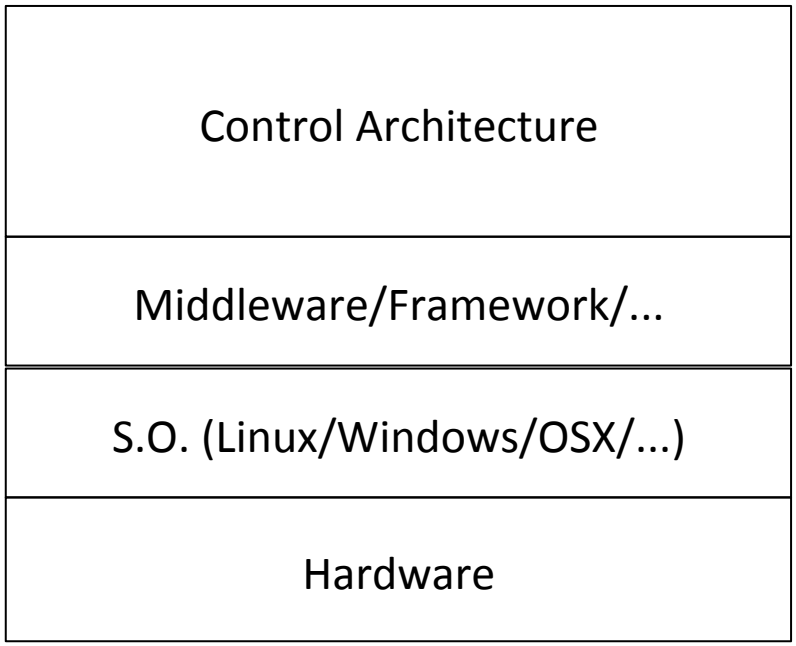
Narcís Palomeras
Universitat de Girona

How to program a robot?

- Different hardware
- Different uses

Layers of a robot





Generic Middlewares, Frameworks, ...

OROCOS

MOOS

Gostai
URBI

Microsoft®
Robotics
Developer
Studio

Common features

- **Simplify the creation and the addition of new components** being as uninvasive as possible when building the components, and minimizing the coupling with the system and the rest of the components.
- **Provide simple mechanisms for communication between the components.** Allowing the distribution of components among the nodes of a network. [*client/service, publish/subscribe, blackboard, ...*]
- **Enable component logging** features. [log files]
- **Allow enabling and disabling components** during the initialization and run-time. [initialization scrip ...]
- Allow components to **save information related to its configuration** in a persistent way. [config files, databases ...]
- **Provide extra tools** to simplify software development and testing. [*visualizers, simulators ...*]

ROS Basics

What is ROS?

The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source.

www.ros.org

- Simplifies the creation and the **addition of new components**
- Provides simple mechanisms for **communication between the components** in a network
- Provides **logging features** to save precious information in a safe and persistent way
- Provides **extra tools**, such as visualizers, simulators and much more
- **Open source**: Uses as a preferred license BSD

How to install ROS

- ROS currently supports **Ubuntu Linux** and has experimental support for other OSs, such as Debian, Arch Linux or OS X
- ROS is really **easy to install**. The **installation guide from the web** is complete:

http://wiki.ros.org/jade/Installation/Ubuntu

Supported:

Ubuntu

Ubuntu (armhf)

Source installation

Experimental:

OS X (Homebrew)

Gentoo Linux

Arch Linux

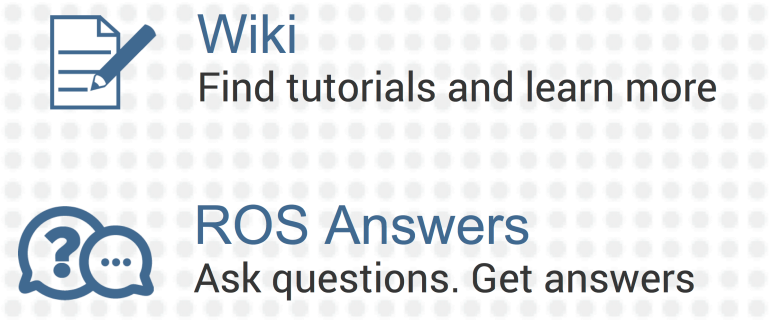
Android (NDK)

How to learn ROS

- There are many **tutorials in the web** that simplify the process of learning ROS:

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

- The tutorials have been organized in a logic and **progressive** way. There is also an indication of the **expected difficulty** of every tutorial
- If you have any problem or question regarding ROS:



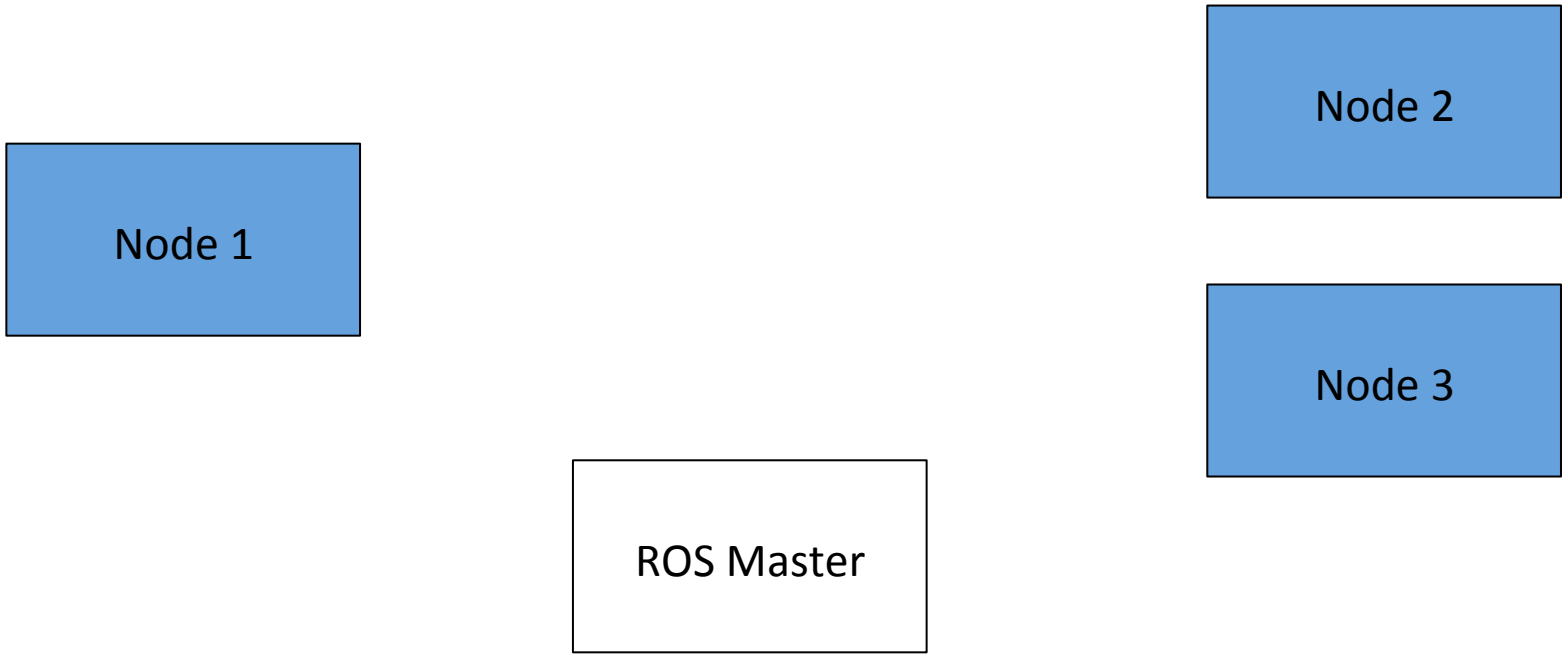
Basic concepts of ROS I

- **Node**: a node is *an executable* that uses ROS to communicate with other nodes.
- **Master (roscore)**: it is the ROS process that *enables the communication between all the other nodes* (i.e. it helps the nodes to find each other (kind of *name server*)).
- **Messages**: messages are ROS *data types* used to communicate information.
- **Topic**: nodes can *publish* messages to a topic as well as *subscribe* to a topic to receive messages. It is like a data stream.
- **Service**: a service is used when a *request/reply* (RPC) communication is needed.

Basic concepts of ROS II

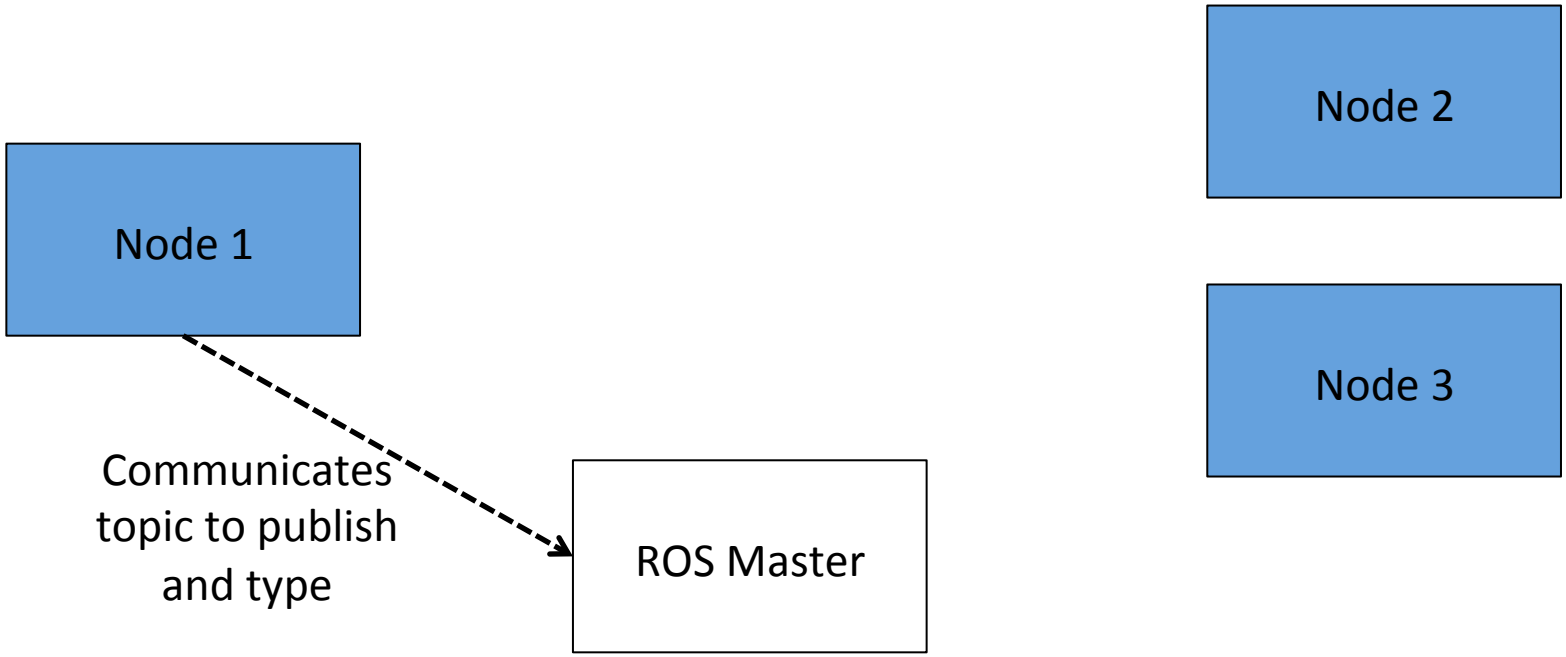
- **Parameter server**: the parameter server is a *shared*, multivariate *dictionary* that is accessible via network APIs.
- **Launch files**: roslaunch is a tool to *run multiple ROS nodes* locally or remotely via SSH as well as load parameters to the *param server*.
- **Package:** a package is a *group of nodes*, messages, services and configuration files.
- **Metapackage:** a metapackage is a *group of related packages.*
- **Workspace**: the workspace is the *main directory* in which all our packages and metapackages must be located.

Communications: Publish/Subscribe



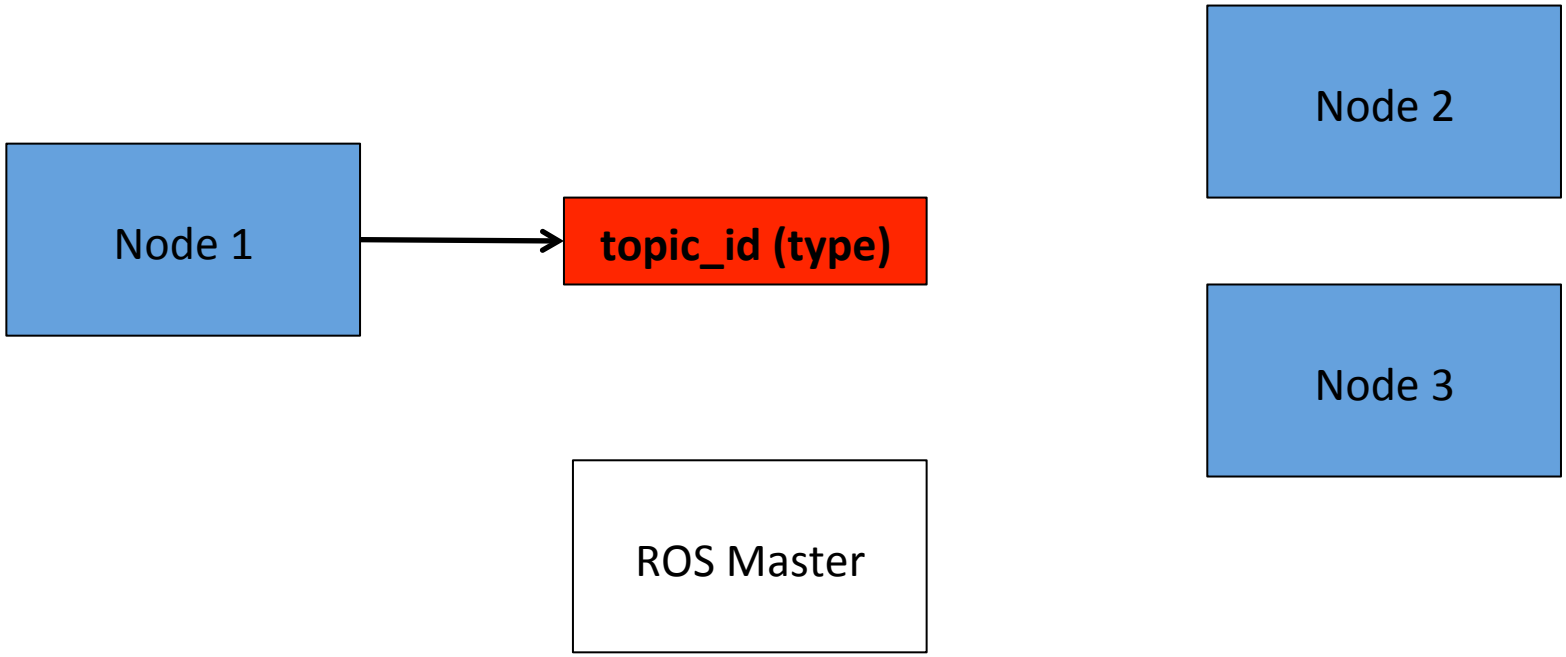
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



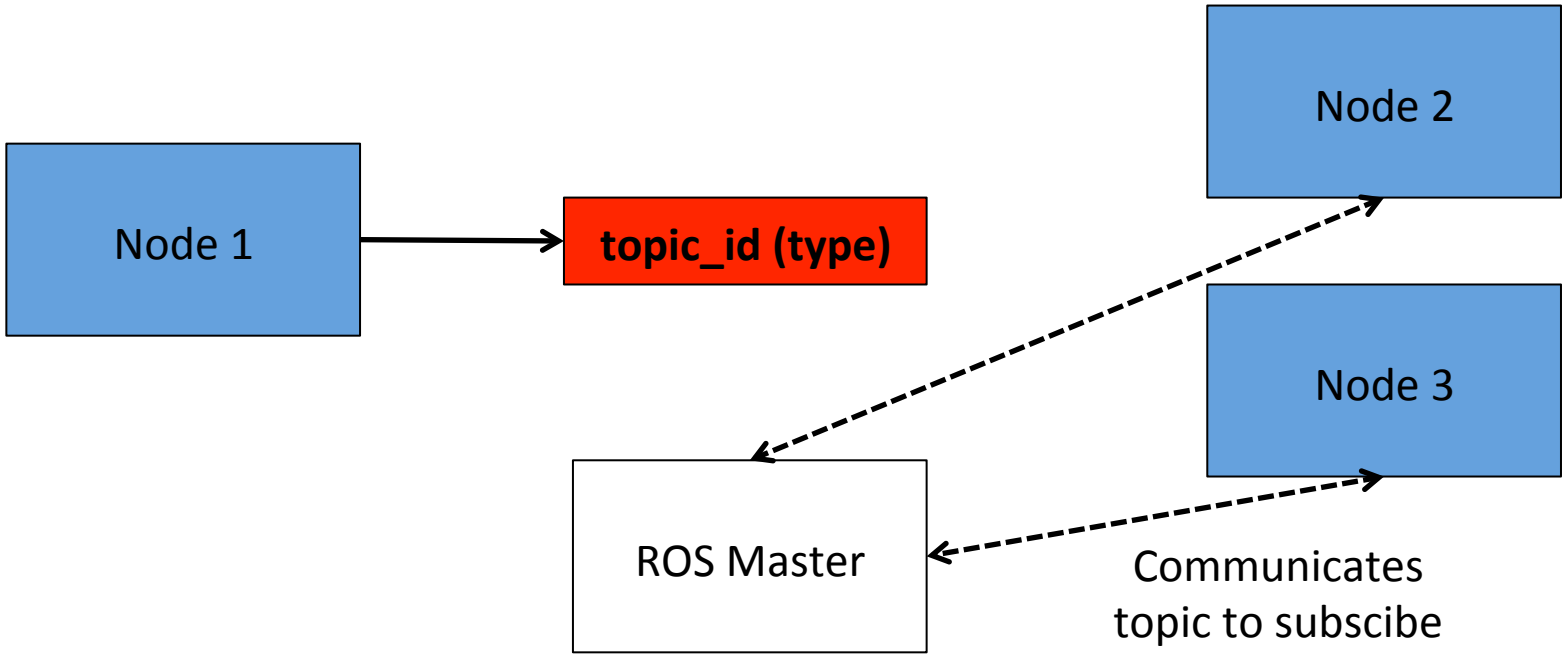
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



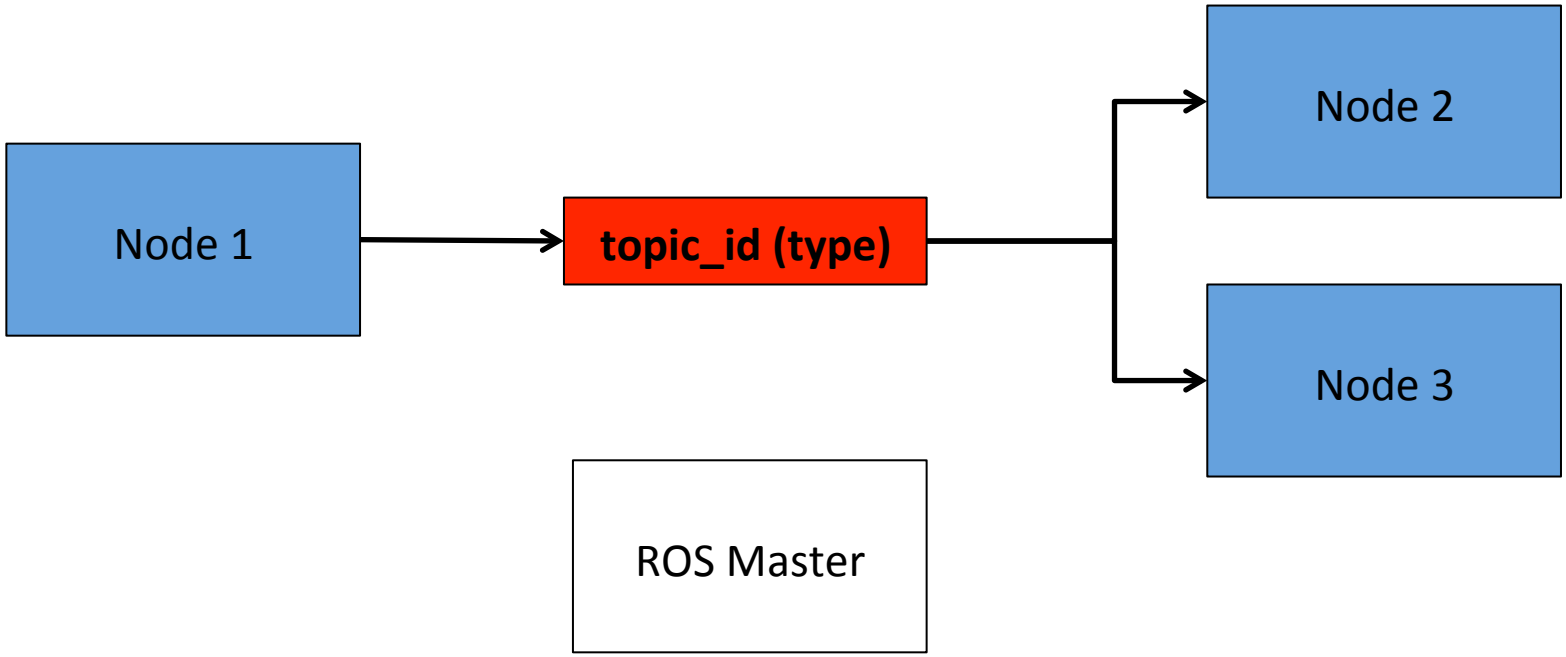
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



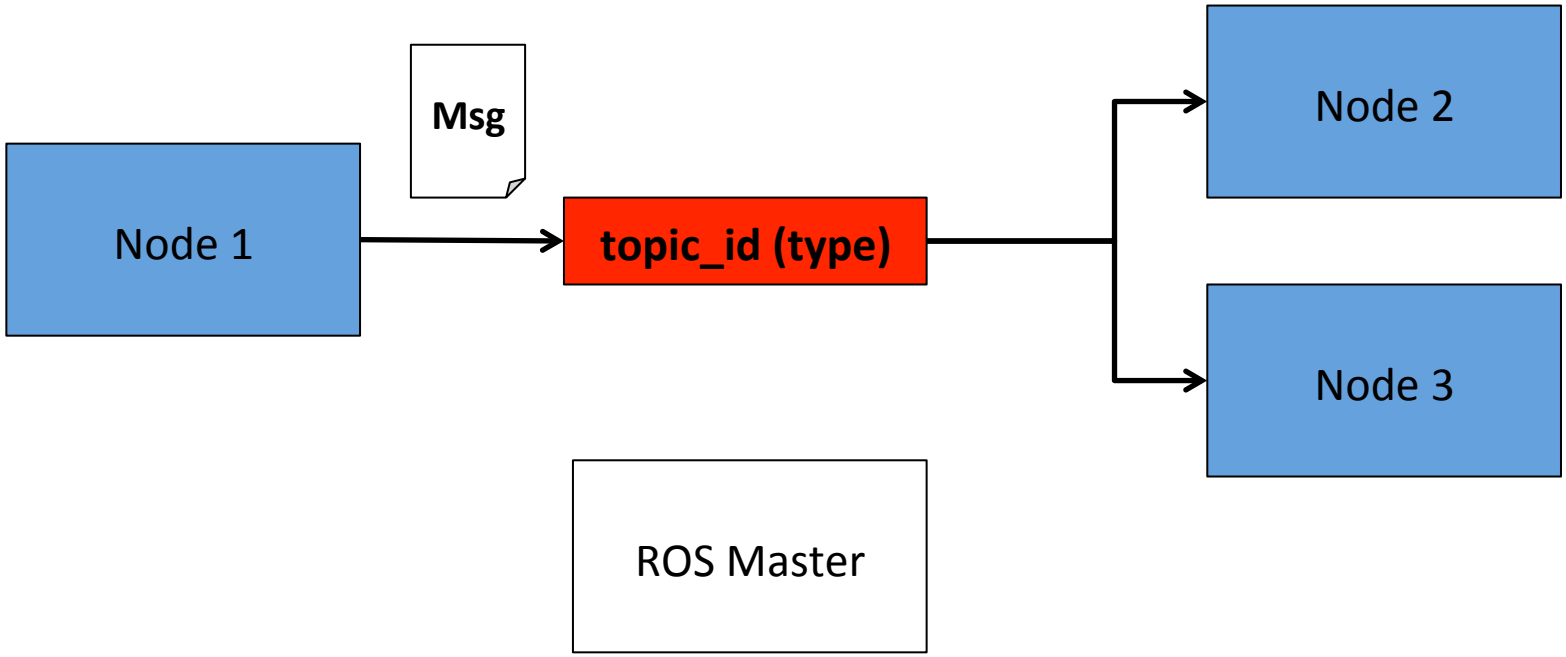
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



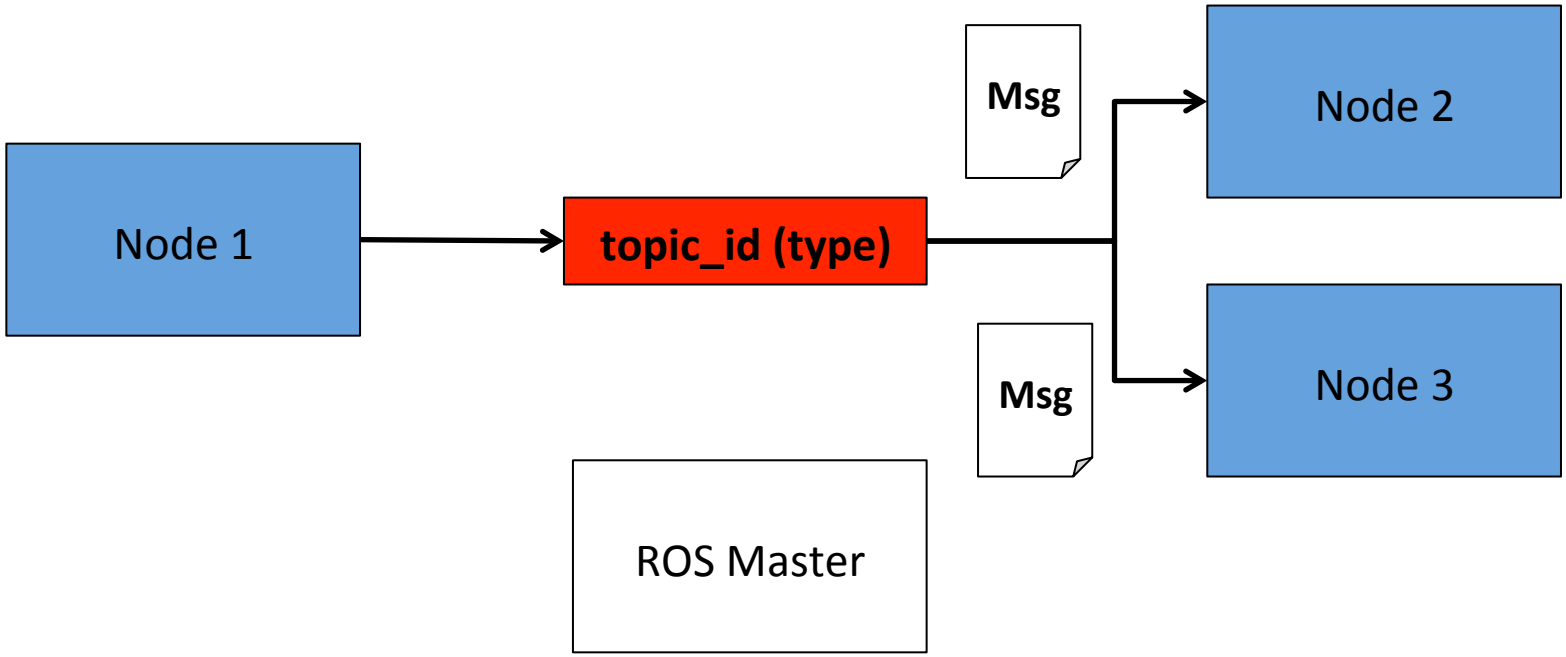
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



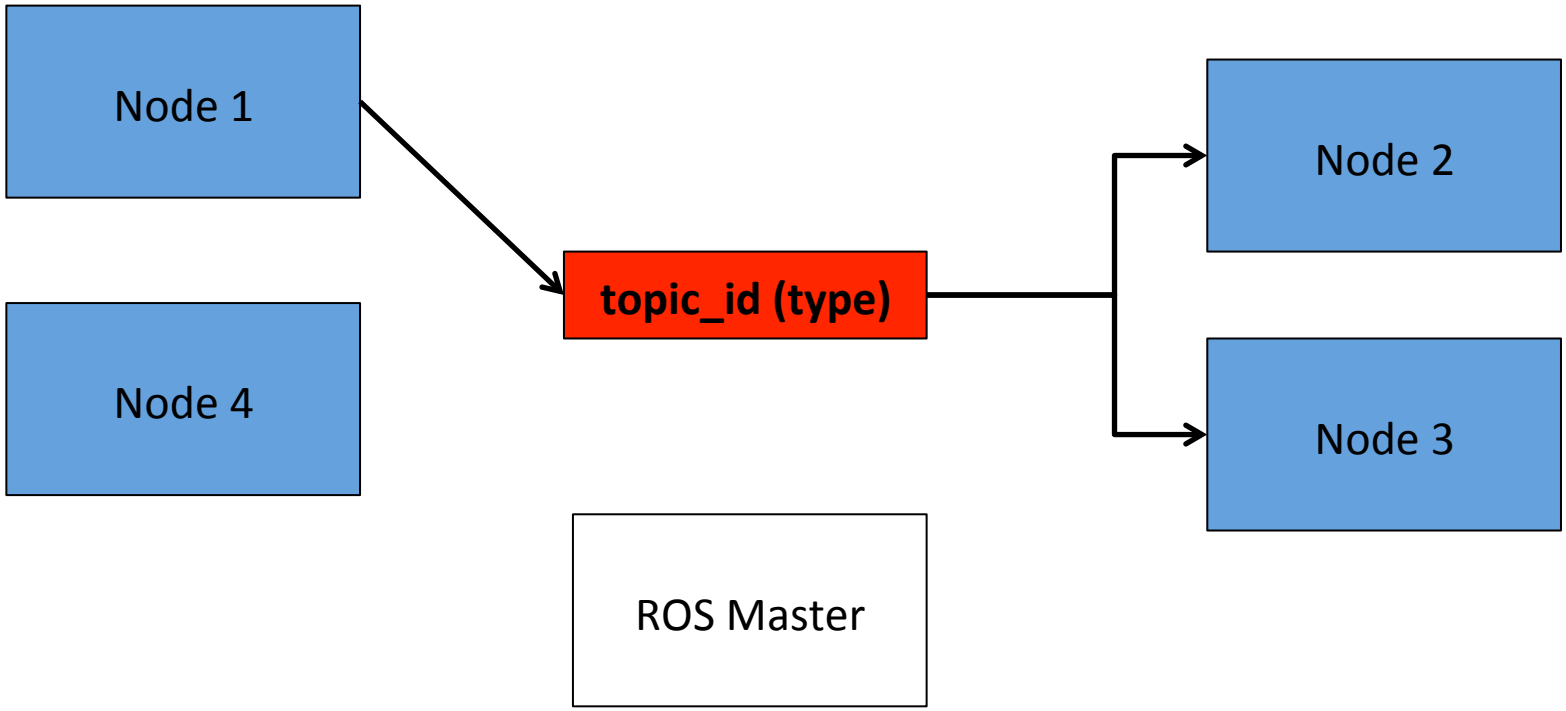
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



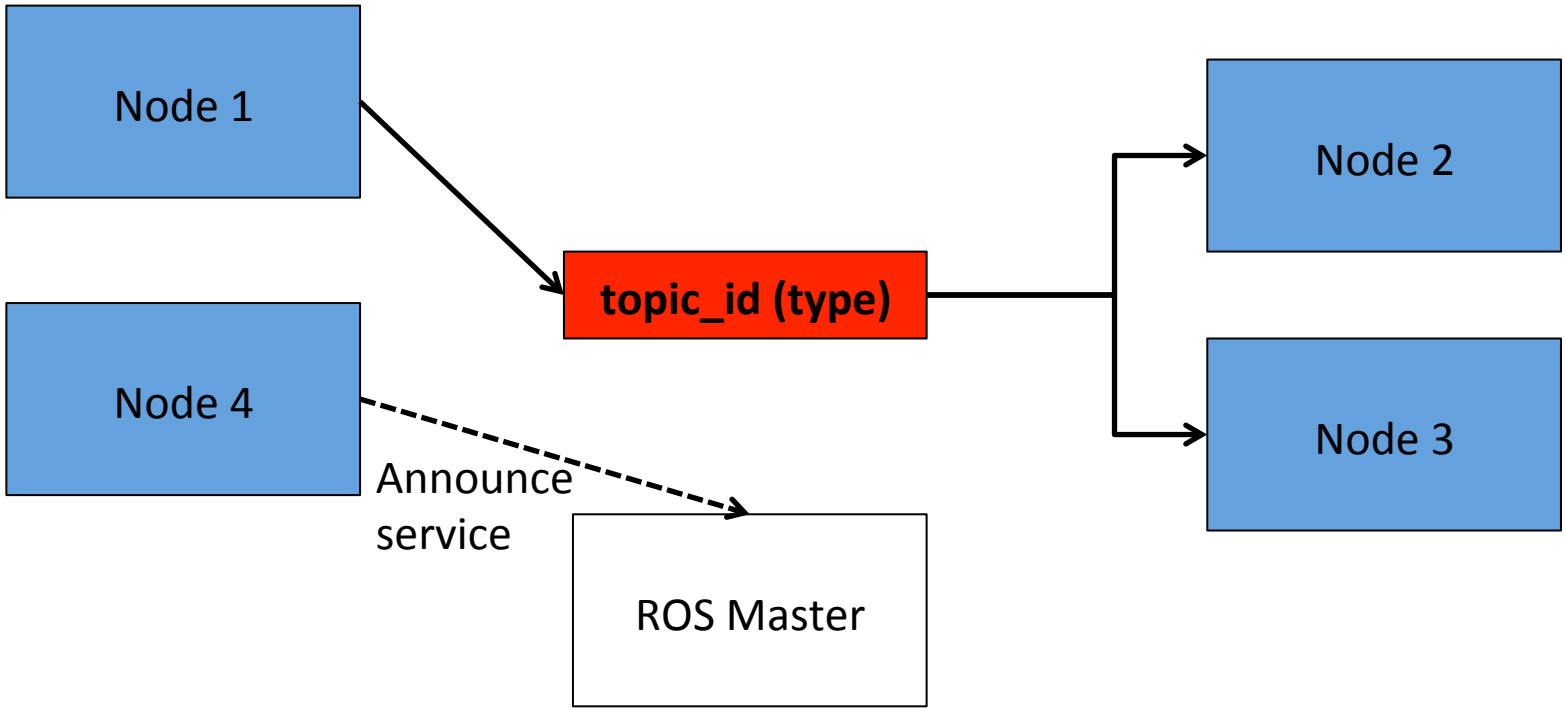
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



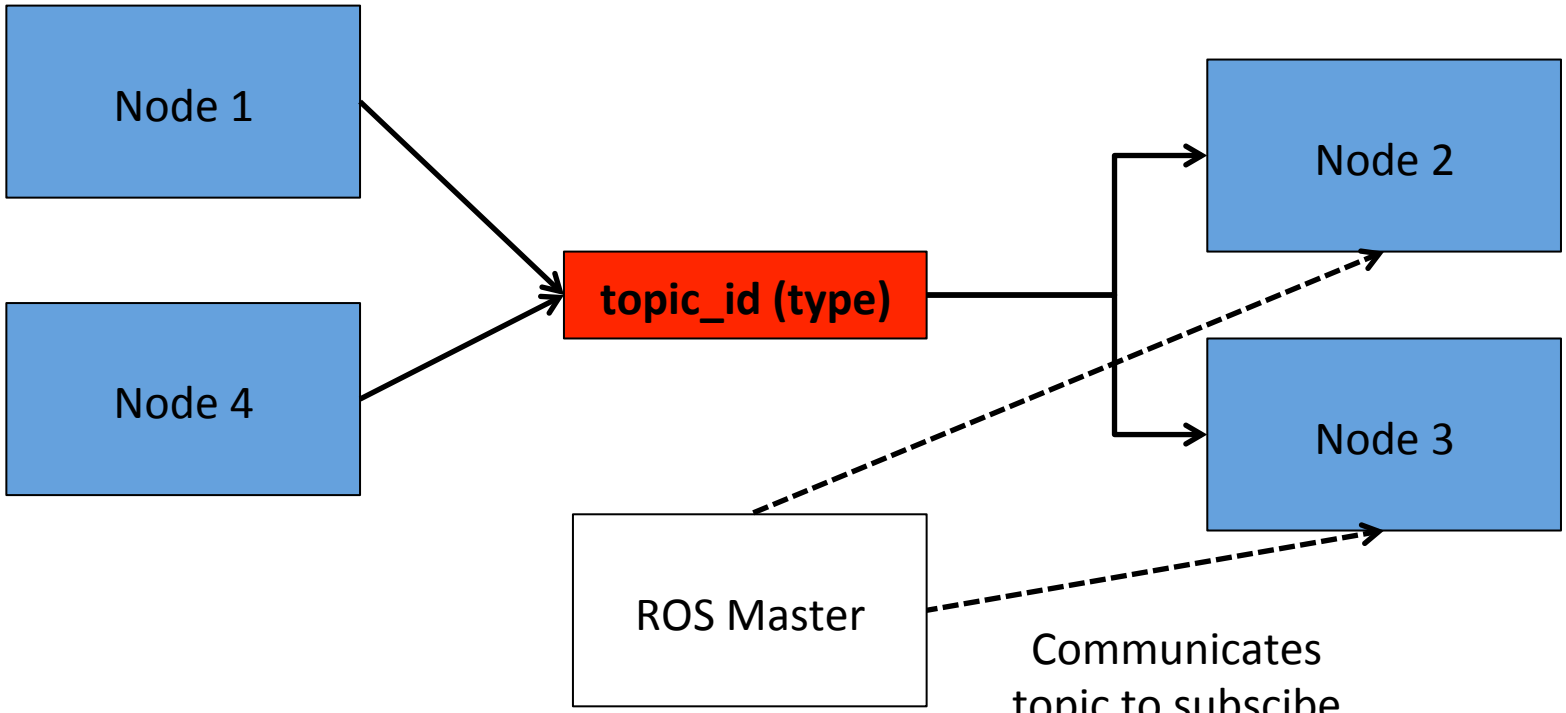
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



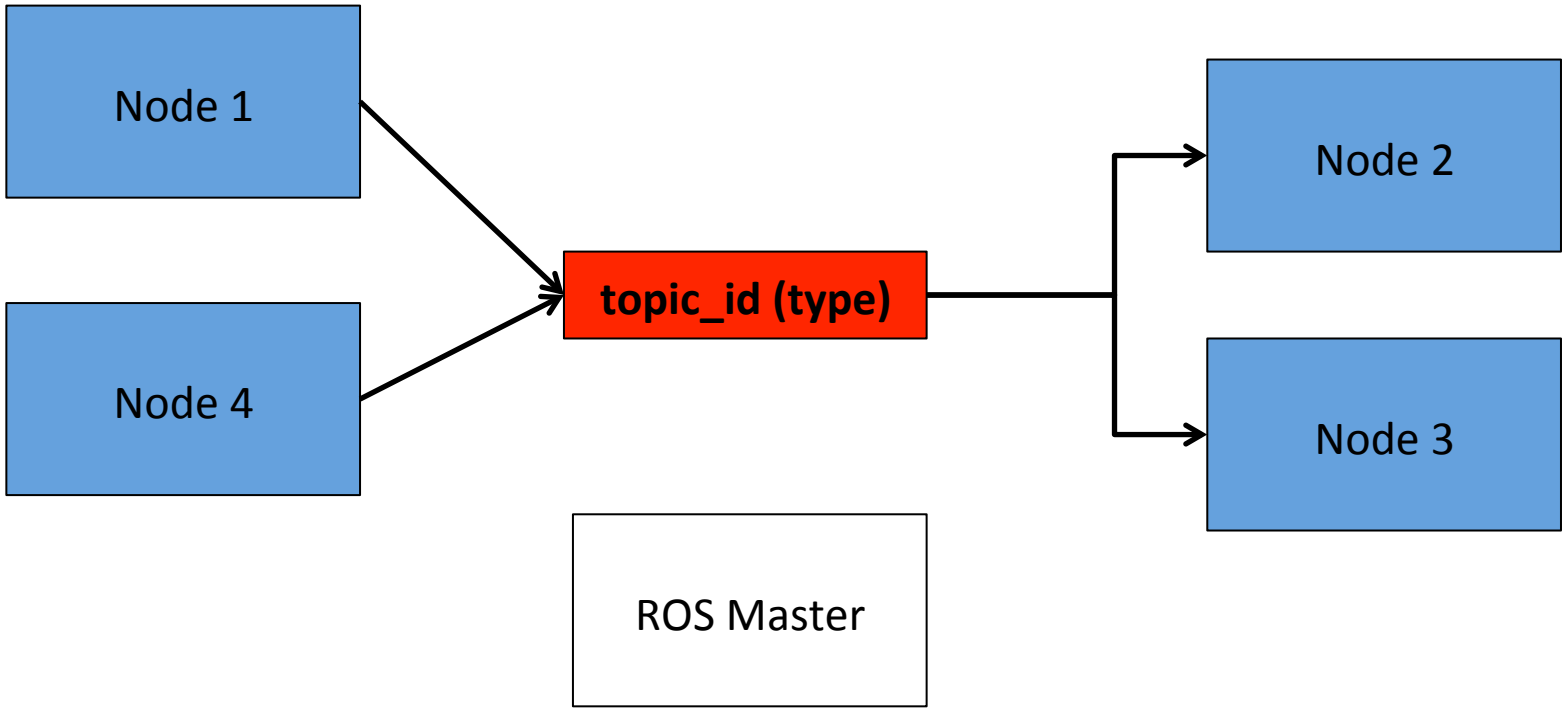
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



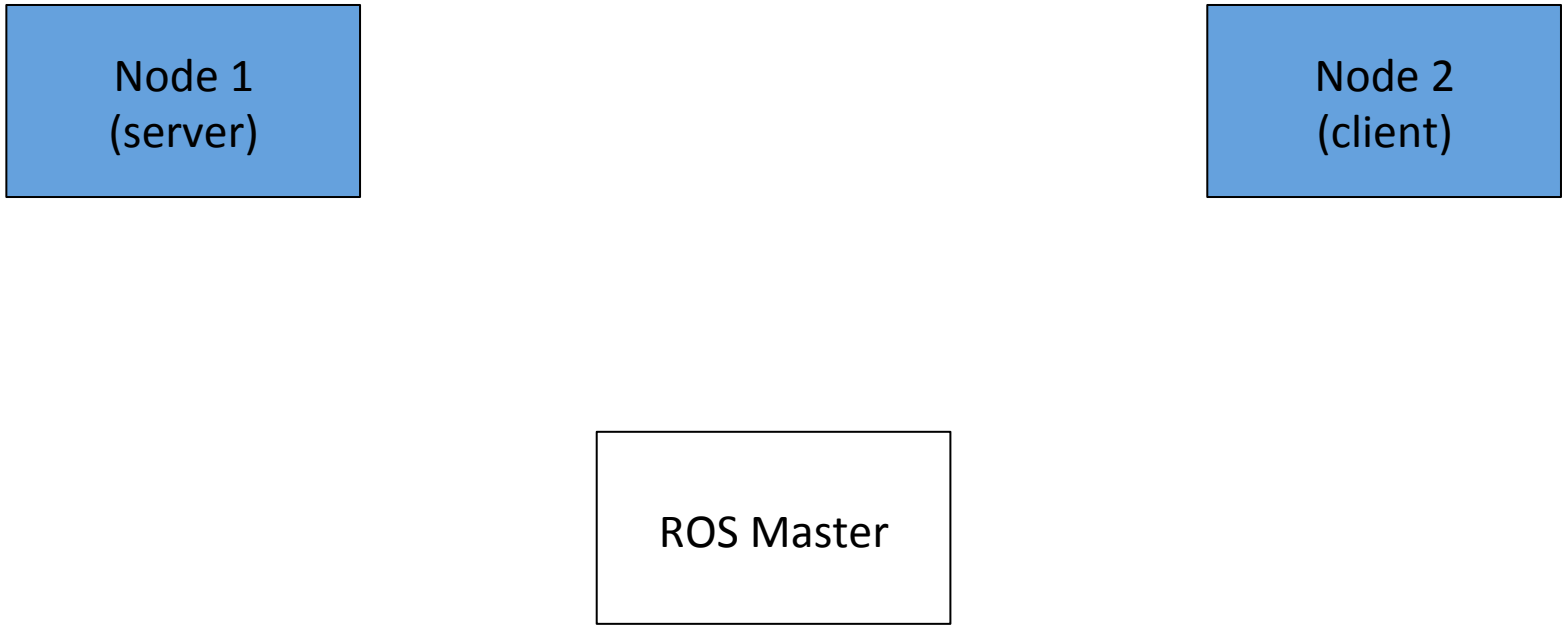
Asynchronous, many-to-many and one-way transport.

Communications: Publish/Subscribe



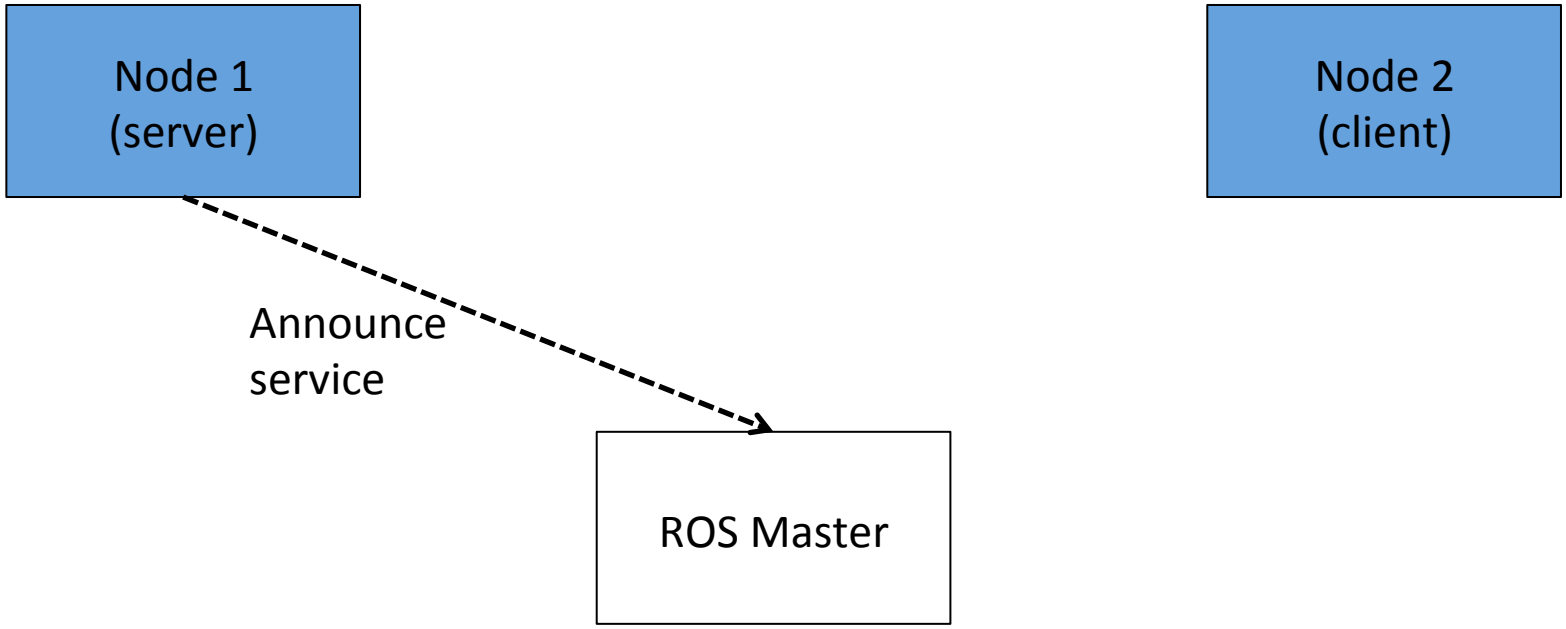
Asynchronous, many-to-many and one-way transport.

Communications: Client/Service



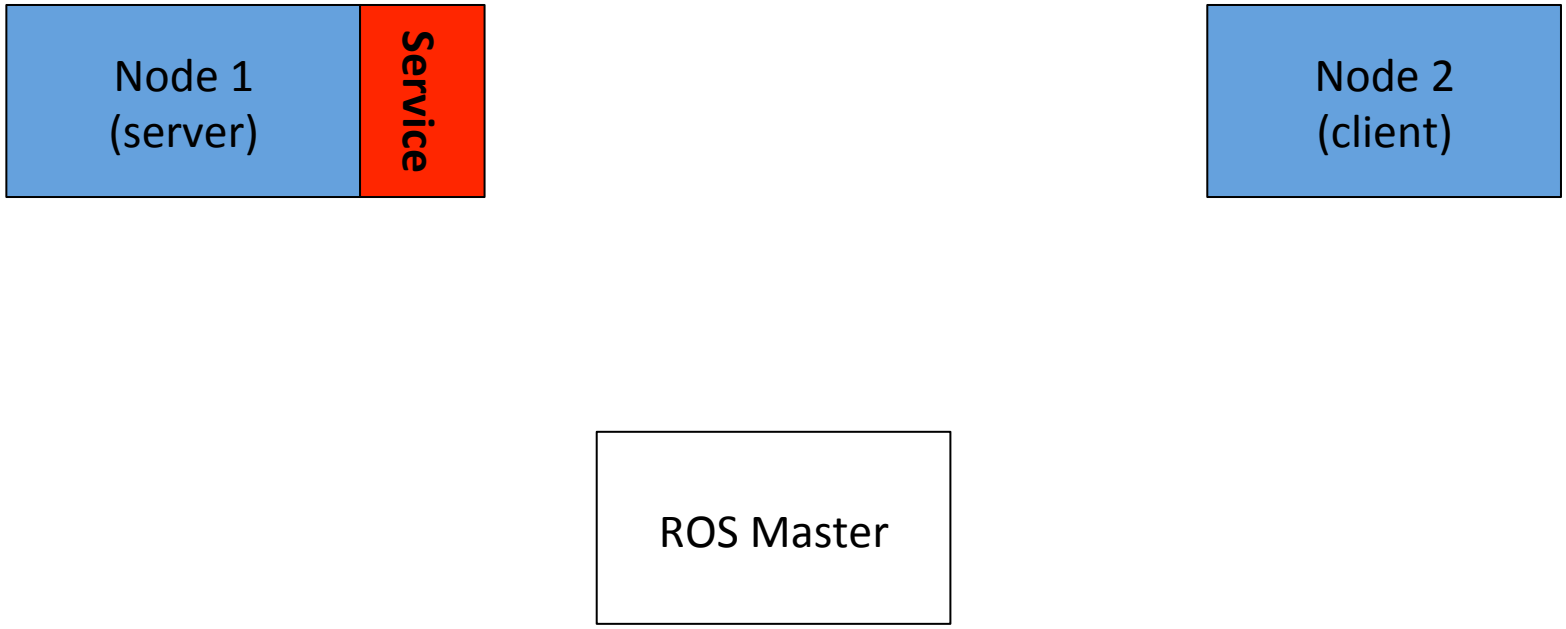
Synchronous, one-to-many

Communications: Client/Service



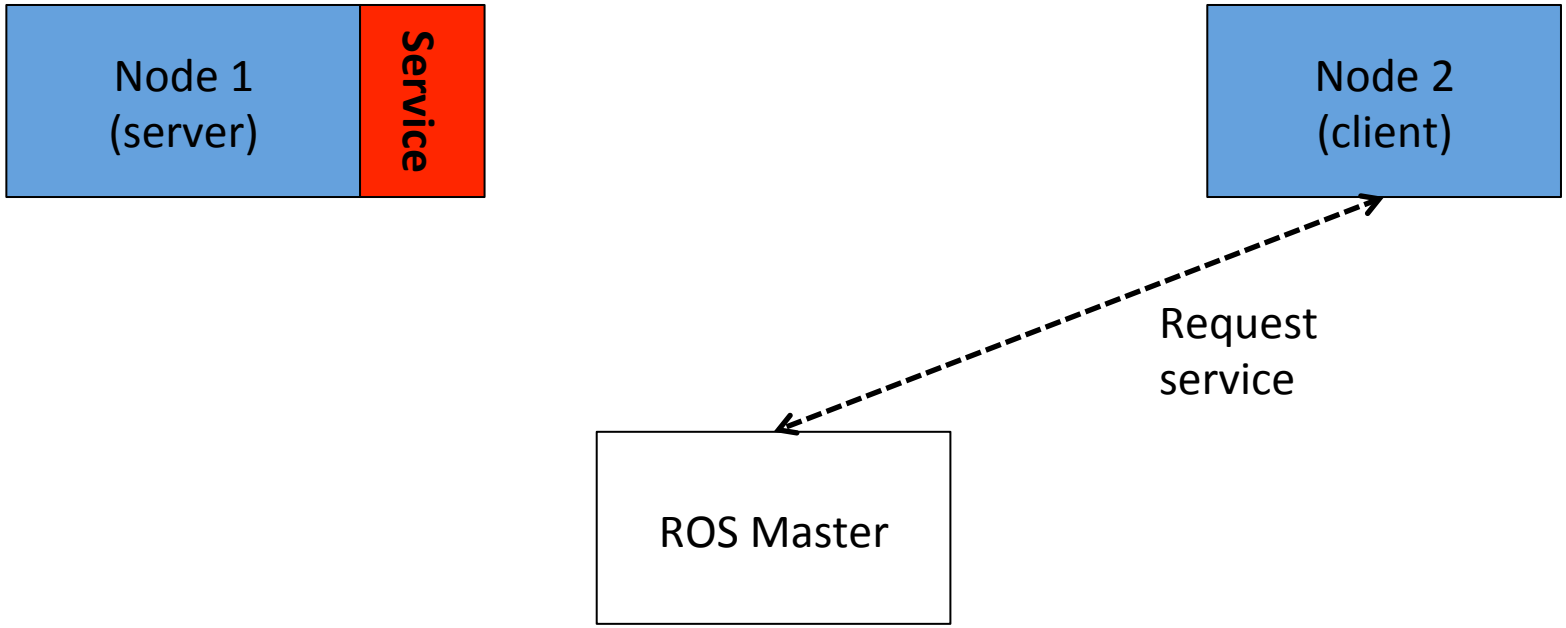
Synchronous, one-to-many

Communications: Client/Service



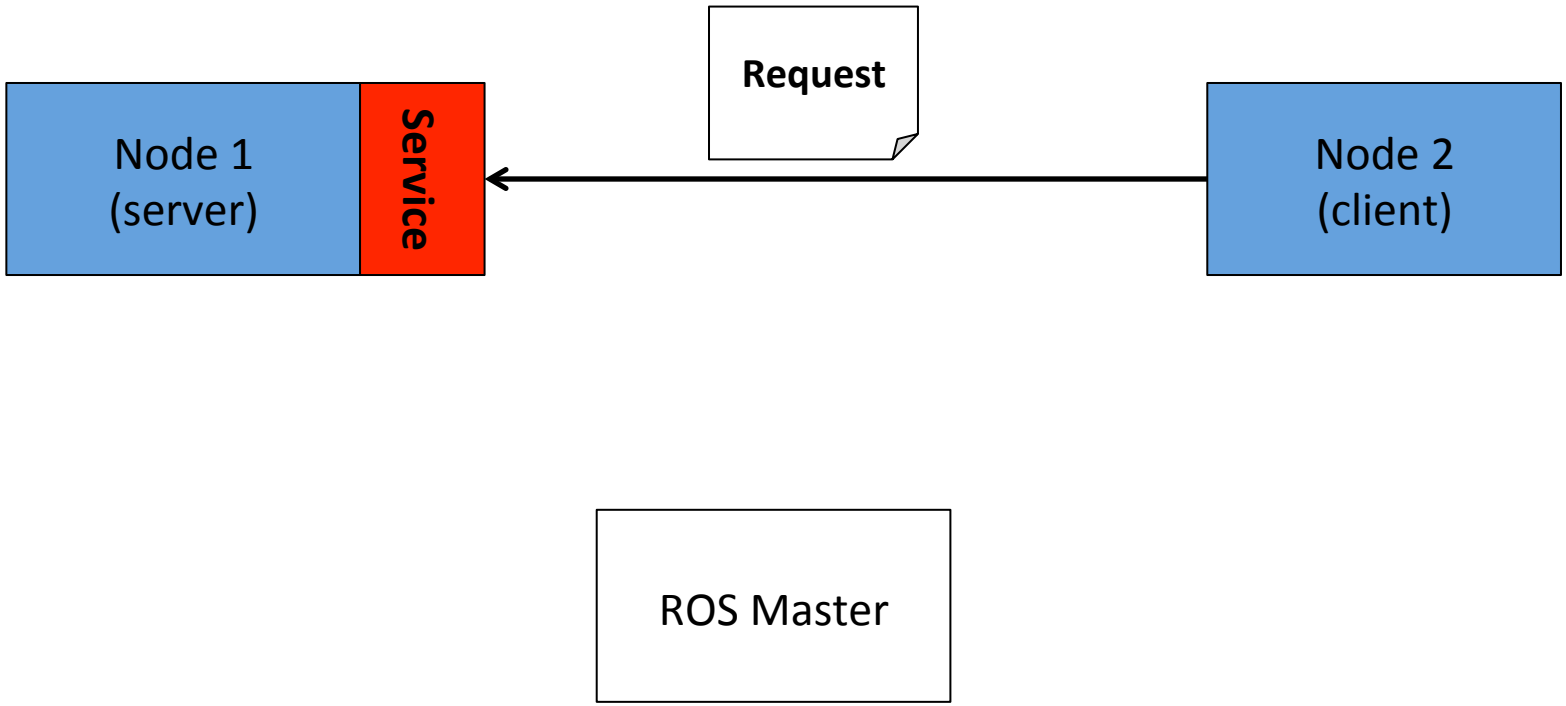
Synchronous, one-to-many

Communications: Client/Service



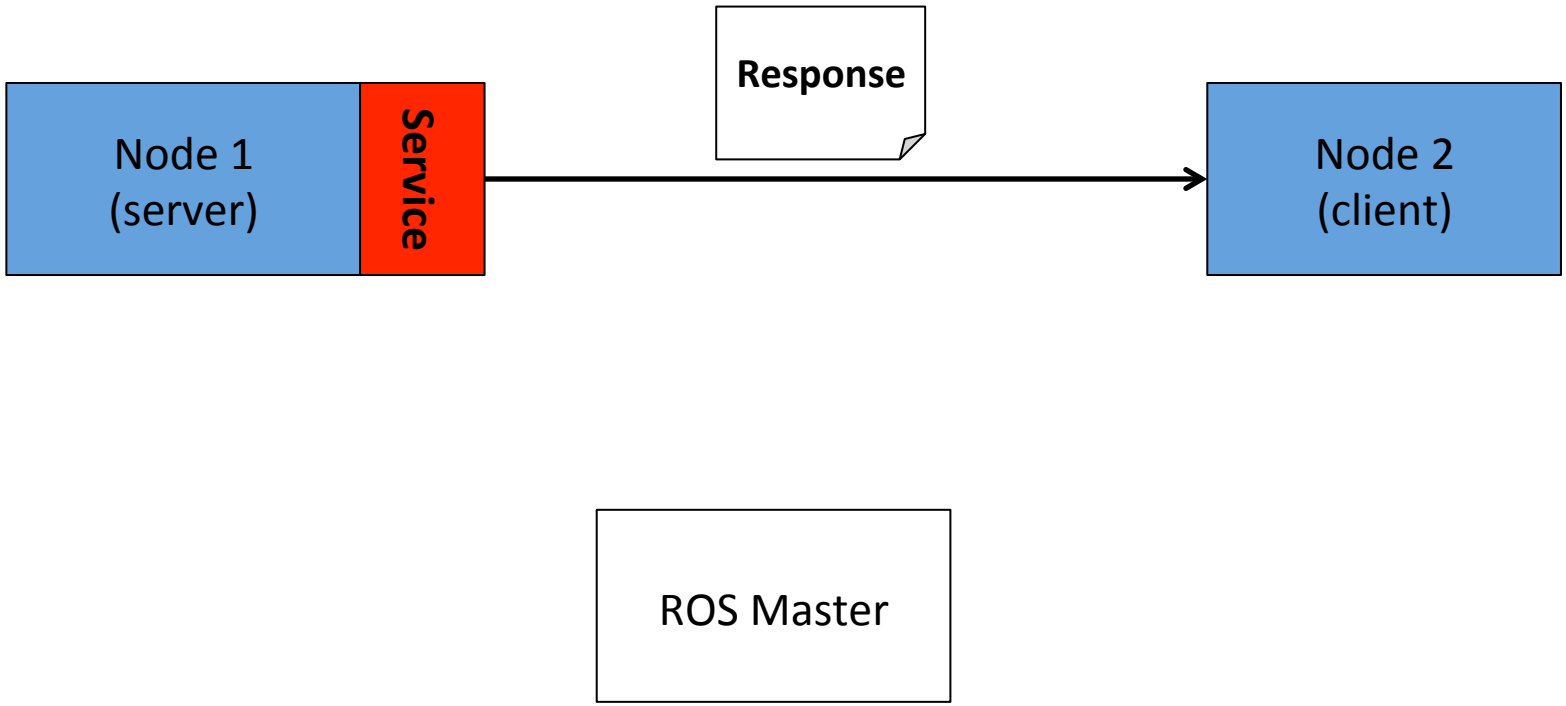
Synchronous, one-to-many

Communications: Client/Service



Synchronous, one-to-many

Communications: Client/Service



Synchronous, one-to-many

How to compile our ROS project

- ROS supports **C++** and **Python** programming languages:

- To compile our ROS code we use the **catkin** build system. To define how catkin has to compile our project we can define the dependencies between different packages in the files **CMakeLists.txt** and **packages.xml**

ROS basic commands

- **Exercise:** Test the following ROS commands

roscore
rosrun
roslaunch
rosnode
rostopic
rosservice

rosparam
rosmsg
rossrv
rosbag
catkin_make

ROS introduction
-- hands on --

Narcís Palomeras
Universitat de Girona

Creating nodes and packages

- The basic building blocks of a Robot Operative System (ROS) application are the nodes. Each node is an independent process that communicates with the others by several ways provided by the ROS framework. Nodes are organized in packages and packages have to be inside a workspace. Then, we need to create first our workspace and then the package that will contain the nodes that we will create.

ROS workspace

- If you have just installed ROS you need to source its setup.

```
# source /opt/ros/<distro>/setup.bash
```

- To create a catkin workspace do:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws/
$ catkin_make
```

ROS packages

- To navigate through ROS packages use **roscd**. It allows you to change directory directly to a package or a stack.

```
# roscd [locationname[/subdir]]
```

- For a package to be considered a catkin package it must meet a few requirements:

```
my_package/
  CMakeLists.txt
  package.xml
```

ROS packages

- To create a new package, navigate to your workspace and then use the **catkin_create_pkg** utility.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg beginner_tutorials std_msgs
rospy roscpp
```

- To build the new created package do:

```
$ cd ~/catkin_ws
$ catkin_make
$ . ~/catkin_ws/devel/setup.bash
$ rospack profile
```

Understanding ROS topics

- One of the mechanisms that ROS nodes use to communicate among them is called publish/ subscribe. A publisher node publishes a message (with a specific type) into a topic (a kind of address). A subscriber node is subscribed to a topic and every time that any node publishes a message to this topic a callback containing the message is executed inside the subscriber node.

Understanding ROS topics

- The publish/subscribe pattern is very useful to broadcast information asynchronously. This means that once a node receives some information (e.g., a compass sensor reads the current heading from a serial bus) it can send this data to several nodes without having to wait that all the nodes that are subscribed to this topic have received the information. Several nodes can publish to the same topic and a node can publish or be subscribed to several topics.

Writing a publisher node

- Create a **src** folder inside your *beginner_tutorials* package. And create a file named **talker.py** in it.

```
$ cd ~/catkin_ws/src/beginner_tutorials
$ mkdir src // if not exists
$ cd src
$ touch talker.py
$ chmod +x talker.py
```

- Copy the following code to it:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Writing a subscriber node

- Create a file named **listener.py** in the *beginner_tutorials/src* folder.

```
$ cd ~/catkin_ws/src/beginner_tutorials/src
$ touch listener.py
$ chmod +x listener.py
```

- Copy the following code to it:

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() +
                  "I heard %s", data.data)

def listener():
    rospy.init_node('listener', anonymous=True
    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this
    # node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

Build and run

- To build your code do:

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

- To run it execute these three lines in three different terminals:

```
$ roscore
-----------------------------------------
$ rosrun beginner_tutorials talker.py
-----------------------------------------
$ rosrun beginner_tutorials listener.py
```

- To see the publis/subscriber connection do:

```
$ rosru rqt_graph rqt_graph
```

Understanding ROS services

- Another mechanisms that ROS nodes use to communicate among them is the client/server model. A client node calls a service provided by a server node. When a server node is launched it announces the services that it provides and if any of their services is called, a callback containing the service request is executed inside the server node. Once the server finalizes its execution it returns a service response to the client node.
- The client/service model is very useful to synchronously request a particular computation/action from one node to another. Examples can be: go to a waypoint, enable a sensor, disable a controller, ...

Creating a ROS srv

- An **.srv** file describes a service. It is composed of two parts: a request and a response. Service files are stored in the *package_name/srv* folder.
- The field types you can use are:
  - int8, int16, int32, int64 (plus uint*)
  - float32, float64
  - string
  - time, duration
  - Header
  - other msg files
  - variable-length array[] and fixed-length array[C]

Creating a ROS srv

- Create a **srv** folder inside the *beginner_tutorials* folder

```
$ roscd beginner_tutorials
$ mkdir srv
$ cd srv
```

- Create a file named **AddTwoInts.srv** with the following content:

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

Creating a ROS srv

- Open the file **package.xml** and add these two lines:

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

- Open the file **CMakeList.txt** and add/uncomment:

```
find_package(catkin REQUIRED COMPONENTS
  roscpp rospy std_msgs message_generation
)
...
add_service_files(
  FILES
  AddTwoInts.srv
)
...
generate_messages(
  DEPENDENCIES
  std_msgs
)
```

Writing add_two_ints_server.py

```
#!/usr/bin/env python

from beginner_tutorials.srv import *
import rospy

def handle_add_two_ints(req):
    print "Returning “, 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()
```

Writing add_two_ints_client.py

```
#!/usr/bin/env python

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))
```

Build and run

- Make the scripts executable:

```
$ chmod +x add_two_ints_client.py
$ chmod +x add_two_ints_server.py
```

- Build the service message as usual:

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

- Run it as before:

```
$ roscore
-----------------------------------------------
$ rosrun beginner_tutorials talker.py
-----------------------------------------------
$ rosrun beginner_tutorials listener.py
```

ROS utilities

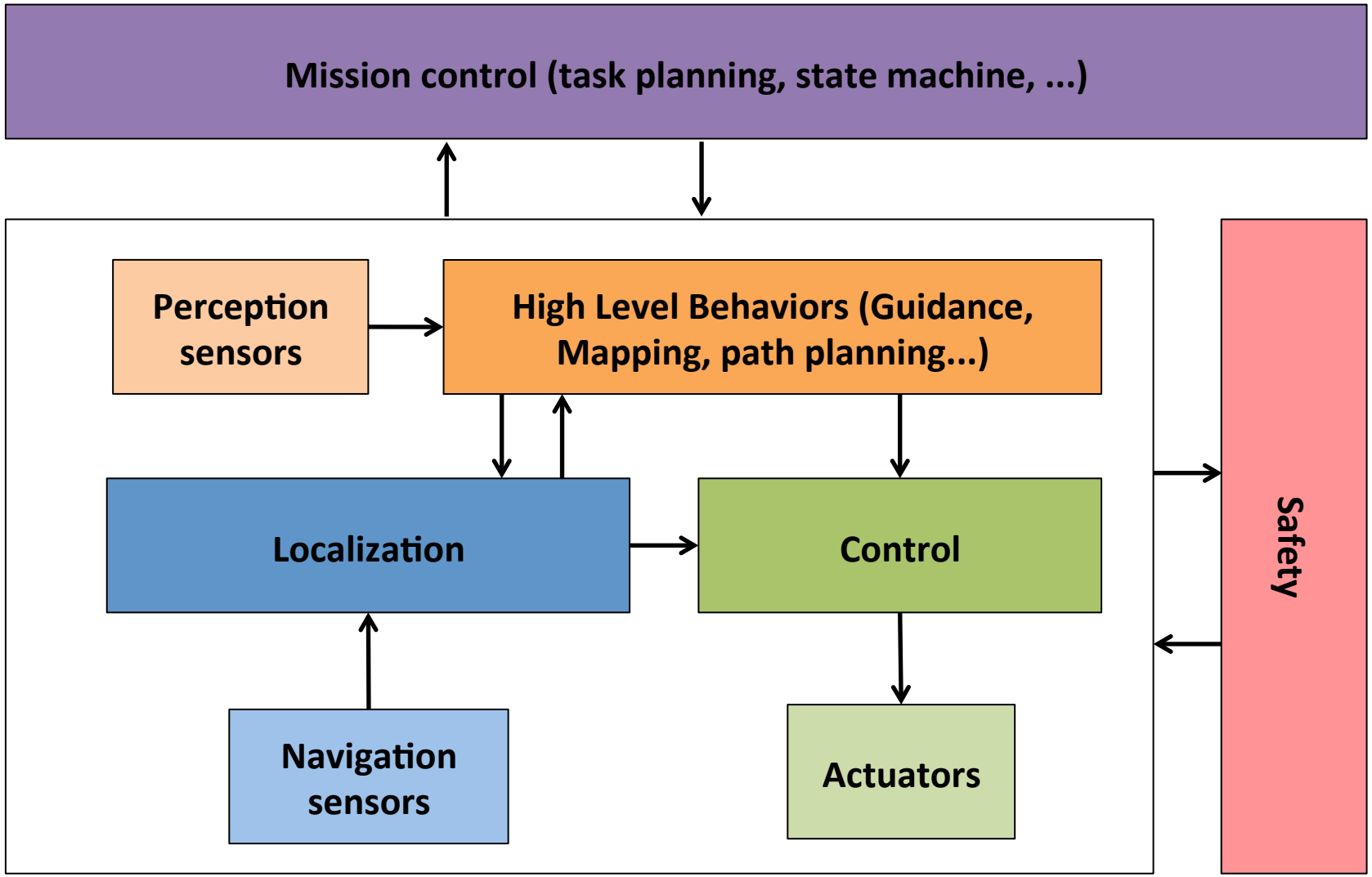
- ROS provides several utilities that are very helpful to develop, test, debug... your robot software:

  - rosmsg
  - rossrv
  - rosbag
  - rosparam
  - rqt_console
  - rqt_tf_tree
  - rqt_graph
  - rqt_bag

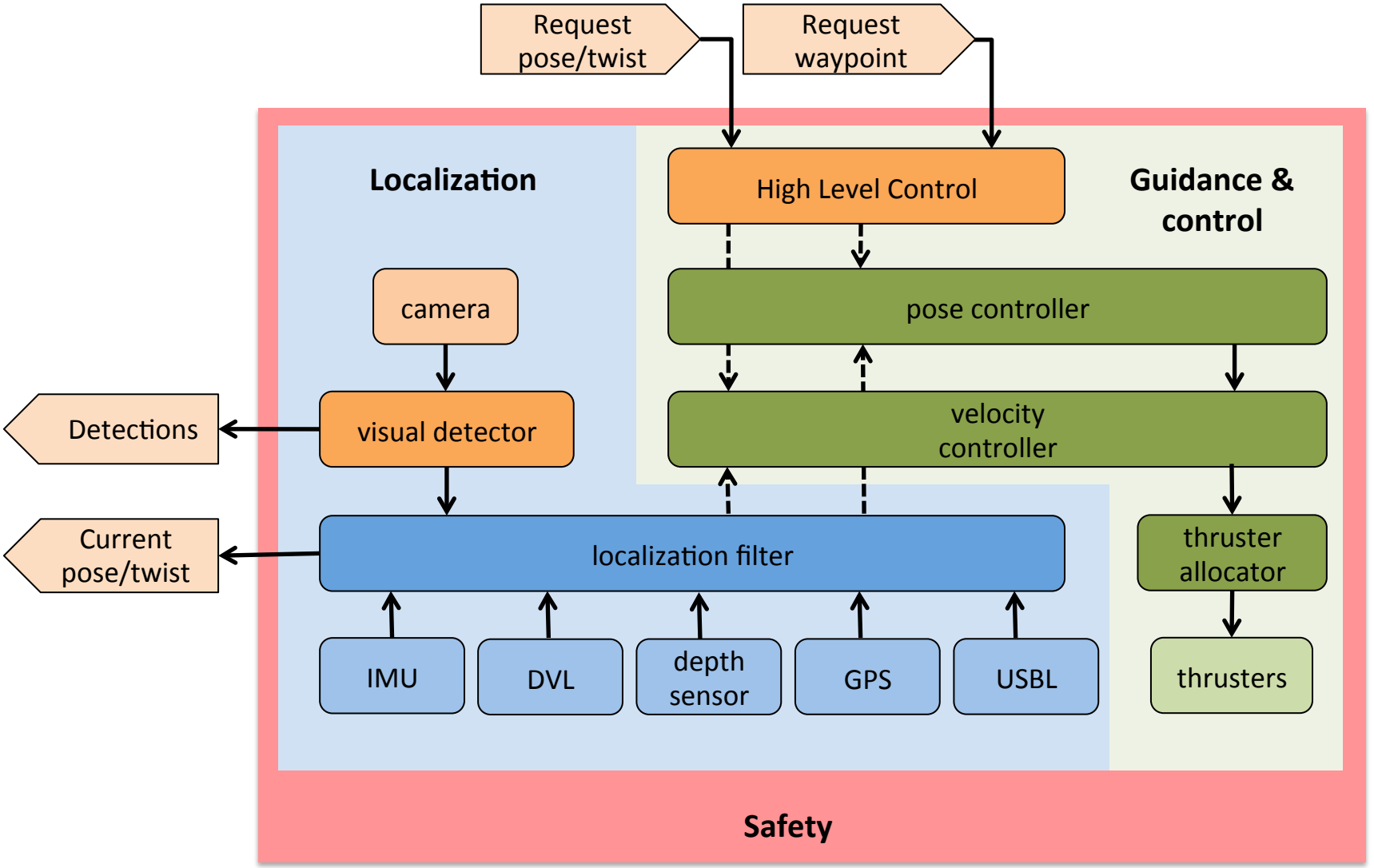
COLA2

Component Oriented Layered-based Architecture for Autonomy

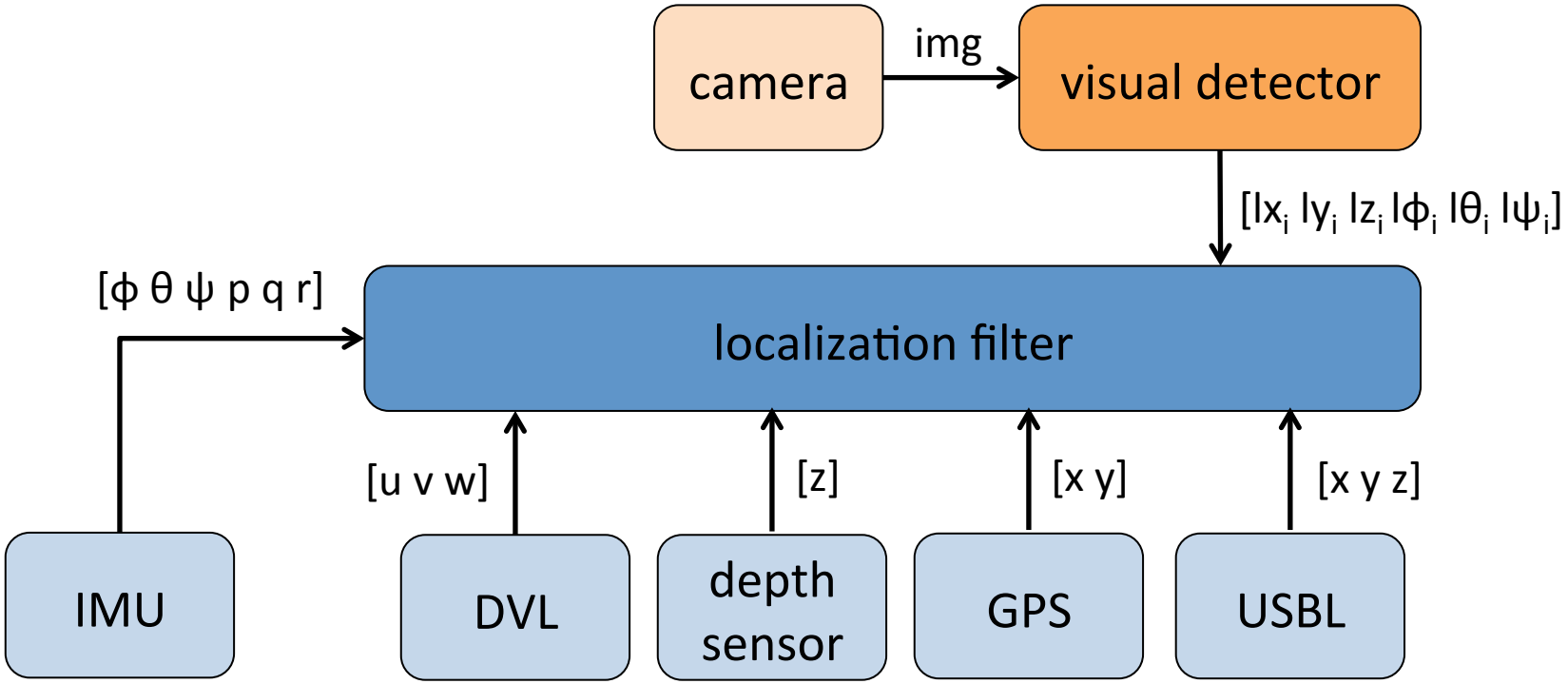
COLA2 Software architecture



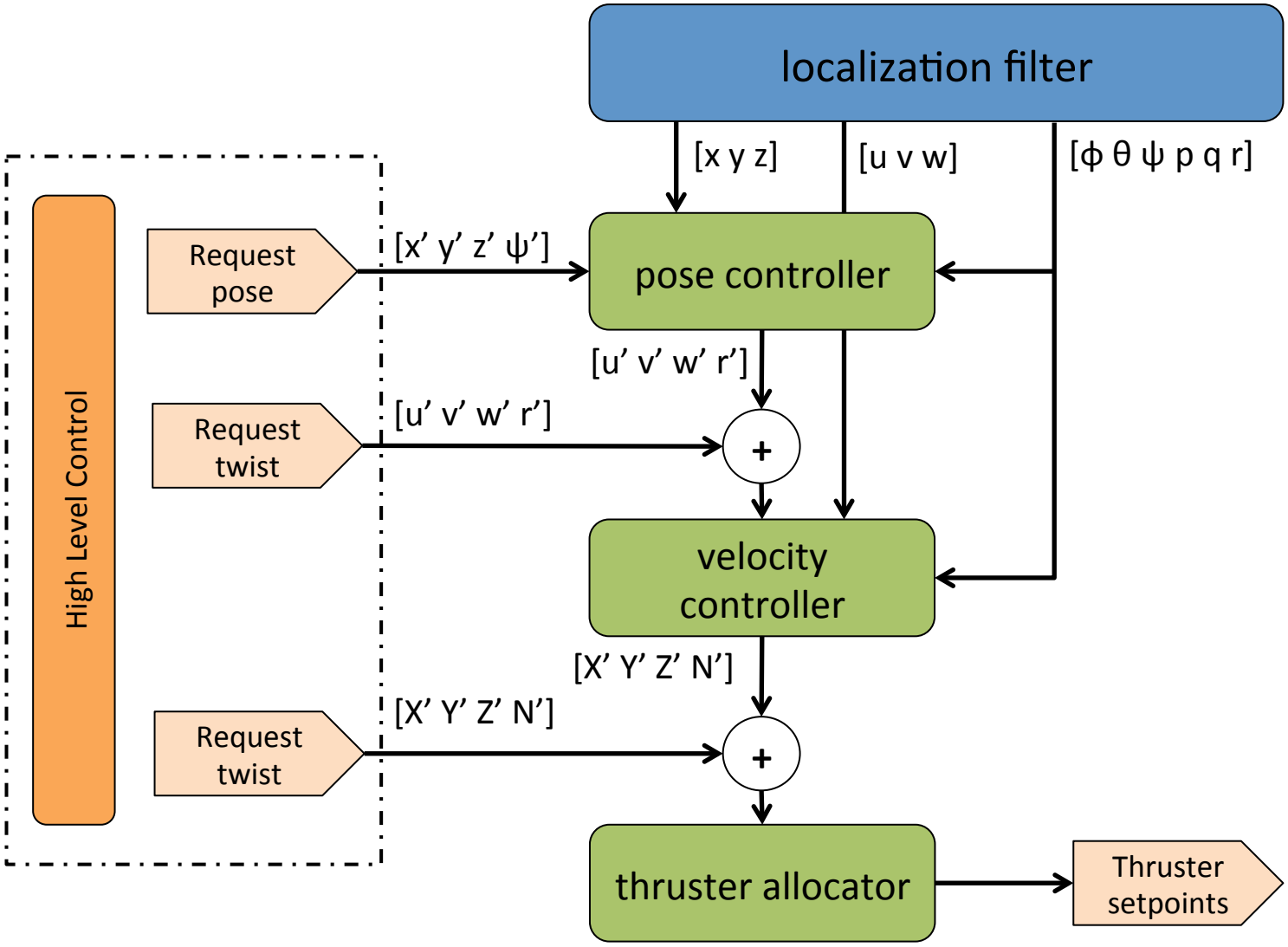
Architecture core



Localization schema



Control schema



Kalman Filter for navigation
-- introduction --

Narcís Palomeras - Univeritat de Girona

based on Greg Welch and Gary Bishop “An introduction to the Kalman Filter” slides in SIGGRAPH 2001

What is a Kalman Filter?

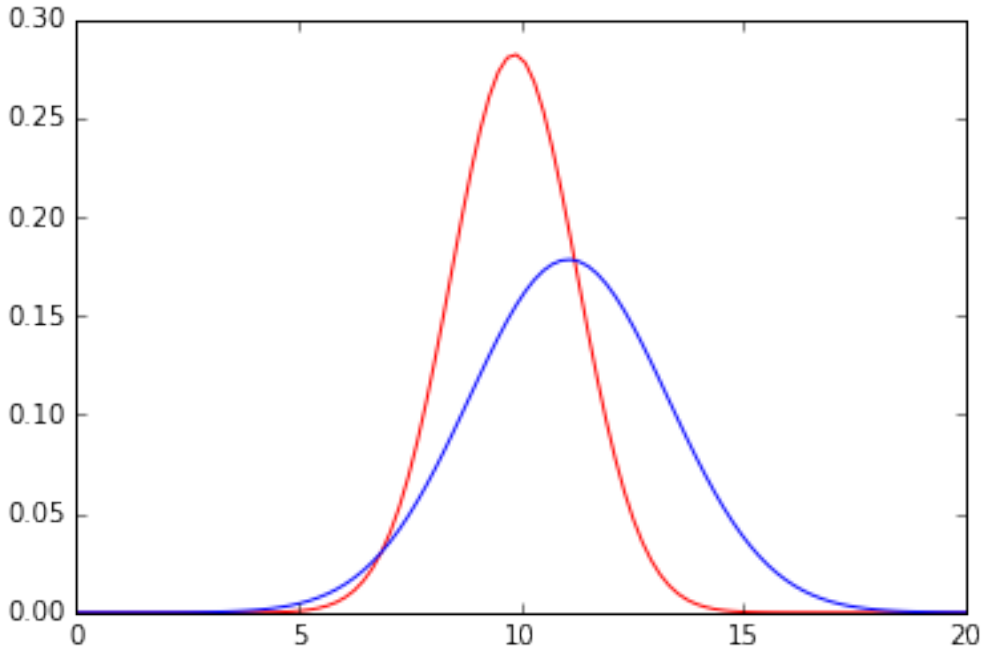
- Just some applied math.
- A linear system: f(a+b) = f(a) + f(b).
- Noisy data in hopefully less noisy out.
- But delay is the price for filtering...

What is it used for?

- Tracking missiles
- Tracking heads/hands/drumsticks
- Extracting lip motion from video
- Fitting Bezier patches to point data
- Lots of computer vision applications
- Economics
- **Navigation**

A simple example

- Alice and Bob measure a tension with their voltimeters. The nominal tension is 10v but Alice voltimeter, that has a variance of 2v, measures 9.85v while Bob voltimeter, that has a variance of 5v, measures 11.07.

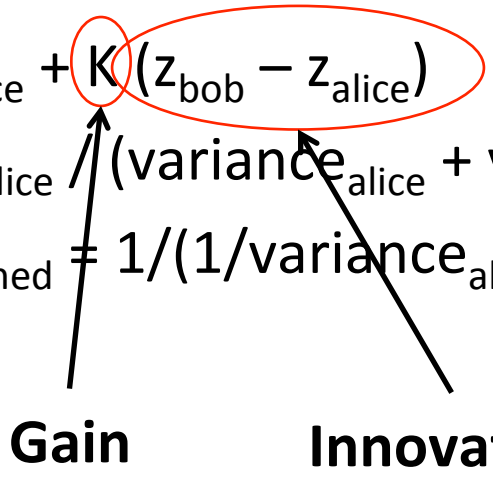


A simple example

- Can we combine both measures? Yes!
- To combine them we use:
  - $z_{combined} = z_{alice} + K\,(z_{bob} - z_{alice})$
  - $K = variance_{alice} / (variance_{alice} + variance_{bob})$
  - $variance_{combined} = 1/(1/variance_{alice} + 1/variance_{bob})$

A simple example

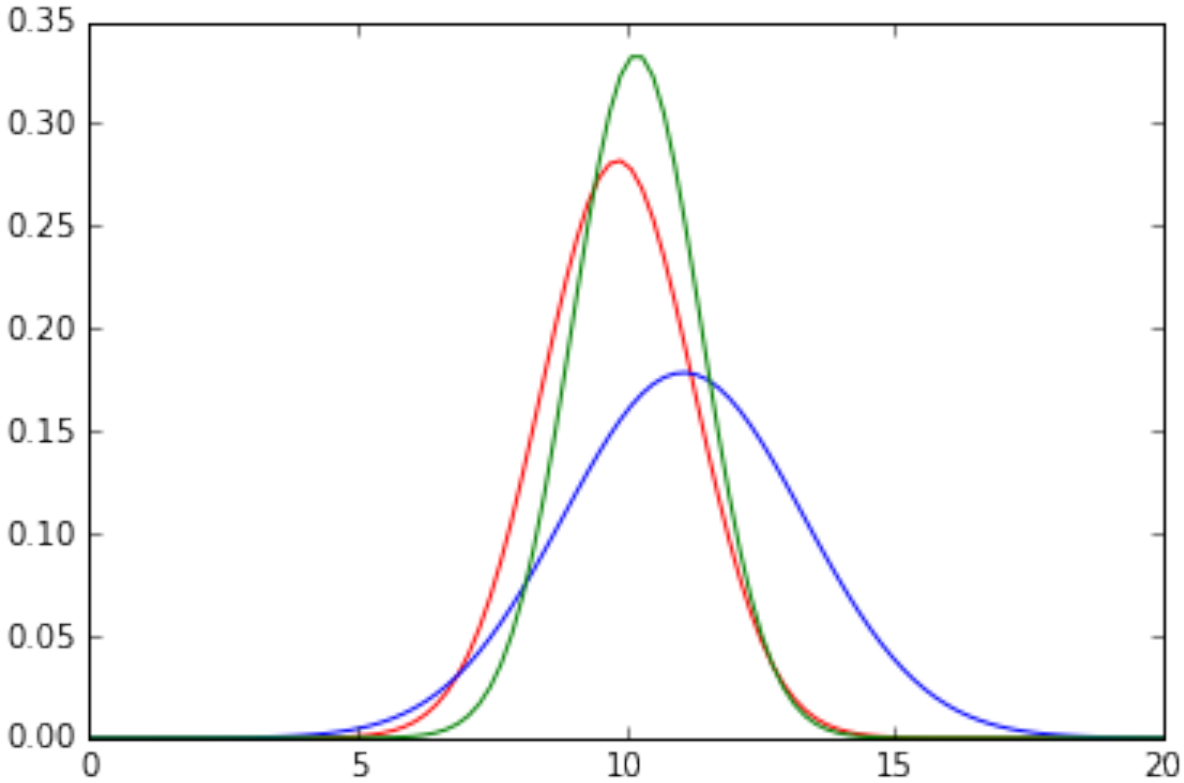
- Can we combine both measures? Yes!
- To combine them we use:
  - $z_{combined} = z_{alice} + K(z_{bob} - z_{alice})$
  - $K = variance_{alice} / (variance_{alice} + variance_{bob})$
  - $variance_{combined} = 1/(1/variance_{alice} + 1/variance_{bob})$



A simple example

- Can we combine both measures? Yes!
- To combine them we use:
  - $z_{combined} = z_{alice} + K\,(z_{bob} - z_{alice})$
  - $K = variance_{alice} / (variance_{alice} + variance_{bob})$
  - $variance_{combined} = 1/(1/variance_{alice} + 1/variance_{bob})$
- Doing the maths ...
  - K: 2/(2+5) = 0.29
  - z: 9.85 + 0.29*(11.07 – 9.85) = 10.2
  - v: 1/(1/2 + 1/5) = 1.43

- $z_{combined} = 10.2$
- $variance_{combined} = 1.43$



Suppose that we measure motion

- Not all the difference is error
- Some may be motion
- KF can include a motion model
- Estimate velocity and position

Process Model

- The Model describes how the *state* changes over time
- The *state* for the first example was scalar
  - The *process model* was “nothing changes”
- For a simple motion example:
  - State is a 2-vector [position, velocity]
  - $\text{position}_{n+1} = \text{position}_n + \text{velocity}_n * \text{time}$
  - $\text{velocity}_{n+1} = \text{velocity}_n$

Two Steps: Predict →Correct

- KF operates by
  - Predicting the new state and its uncertainty
  - Correcting with the new measurement

Motion example

- We want to filter the motion of a point that moves in the plane.
- We have a noisy measure (x, y) every second.
- The velocity (dx/dt, dy/dt) is more less constant with small variations.

Preparation

- Define state vector:
  - x_K = [px py vx vy]

where:

px = x position

py = y position

vx = dx/dt

vy = dy/dt

Preparation

- Define state vector:
  - x_K = [px py vx vy]
- Define model
  - $px_K = px_{K-1} + vx_{K-1}*dt$
  - $py_K = py_{K-1} + vy_{K-1}*dt$
  - $vx_K = vx_{K-1}$
  - $vy_K = vy_{K-1}$

Preparation

- Define state vector:
  - $x_K = [px\ py\ vx\ vy]^T$
- Define model
  - $px_K = px_{K-1} + vx_{K-1} * dt$
  - $py_K = py_{K-1} + vy_{K-1} * dt$
  - $vx_K = vx_{K-1}$
  - $vy_K = vy_{K-1}$

Define **A** matrix:

$$\begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Process Model

$$\begin{bmatrix} px_K \\ py_K \\ vx_K \\ vy_K \end{bmatrix} = \begin{bmatrix} 1 & 0 & dt & 0 \\ 0 & 1 & 0 & dt \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} px_{K-1} \\ py_{K-1} \\ vx_{K-1} \\ vy_{K-1} \end{bmatrix} + \begin{bmatrix} \sim px_{K-1} \\ \sim py_{K-1} \\ \sim vx_{K-1} \\ \sim vy_{K-1} \end{bmatrix}$$

State $\bar{x}_k^-$

State transition **A**

State $\bar{x}_{k-1}$

Noise W_{k-1}

$$\bar{x}_k = A\bar{x}_{k-1} + \bar{w}_{k-1}$$

Measurement Model

- Define measure:

$$z_K = [z_x \; z_y]$$

- Define **H** matrix:

$$\mathbf{H} = \begin{bmatrix} H_x & 0 & 0 & 0 \\ 0 & H_y & 0 & 0 \end{bmatrix}$$

Measurement Model

$$\begin{bmatrix} z_x \\ z_y \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} px_K \\ py_K \\ vx_K \\ vy_K \end{bmatrix} + \begin{bmatrix} \sim z_x \\ \sim z_y \end{bmatrix}$$

Measurement	Measurement matrix	State	Noise
z_K	**H**	x_K	v_K

$$\bar{z}_k = H\bar{x}_k + \bar{v}_k$$

Prepare noise matrices

- Process Noise Covariance:

$$Q = E\left\{\overline{w} * \overline{w}^T\right\}$$

- Measurement Noise Covariance:

$$R = E\left\{\overline{v} * \overline{v}^T\right\}$$

Predict

$$\bar{x}_k^- = A\bar{x}_{k-1}$$

$$P_k^- = AP_{k-1}A^T + Q$$

transition

uncertainty

Correct

$$\bar{x}_k = \bar{x}_k^- + K\left(\bar{z}_k - H\bar{x}_k^-\right)$$

$$P_k = \left(I - KH\right)P_k^-$$

actual

predicted

$$K = P_k^- H^T \left(HP_k^- H^T + R\right)^{-1}$$

"denominator"
(measurement space)

Summary

$$\bar{x}_k^- = A\bar{x}_{k-1}$$

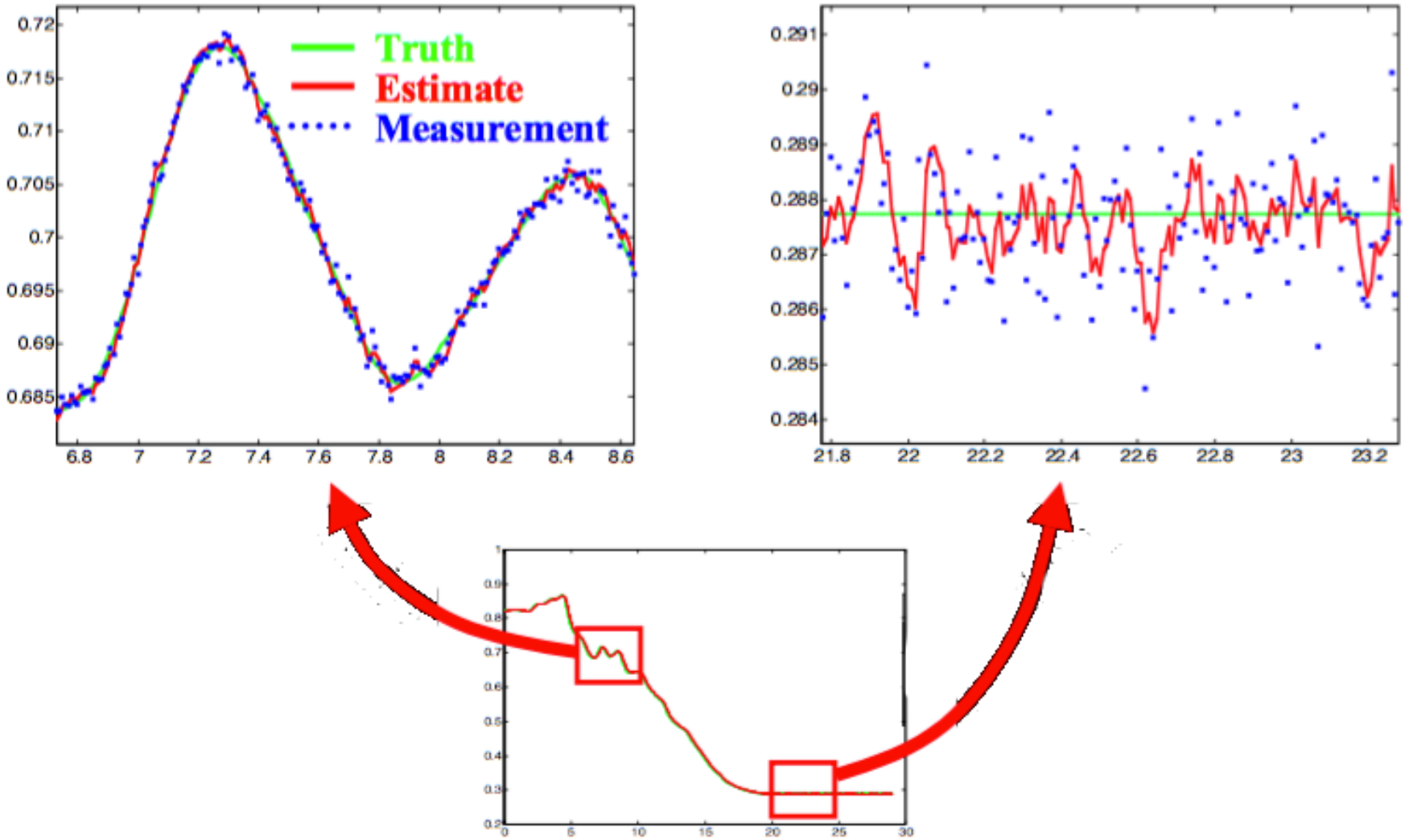
$$P_k^- = AP_{k-1}A^T + Q$$

$$K = P_k^- H^T \left(HP_k^- H^T + R \right)^{-1}$$

$$\bar{x}_k = \bar{x}_k^- + K\left(\bar{z}_k - H\bar{x}_k^- \right)$$

$$P_k = \left(I - KH \right) P_k^-$$

Results

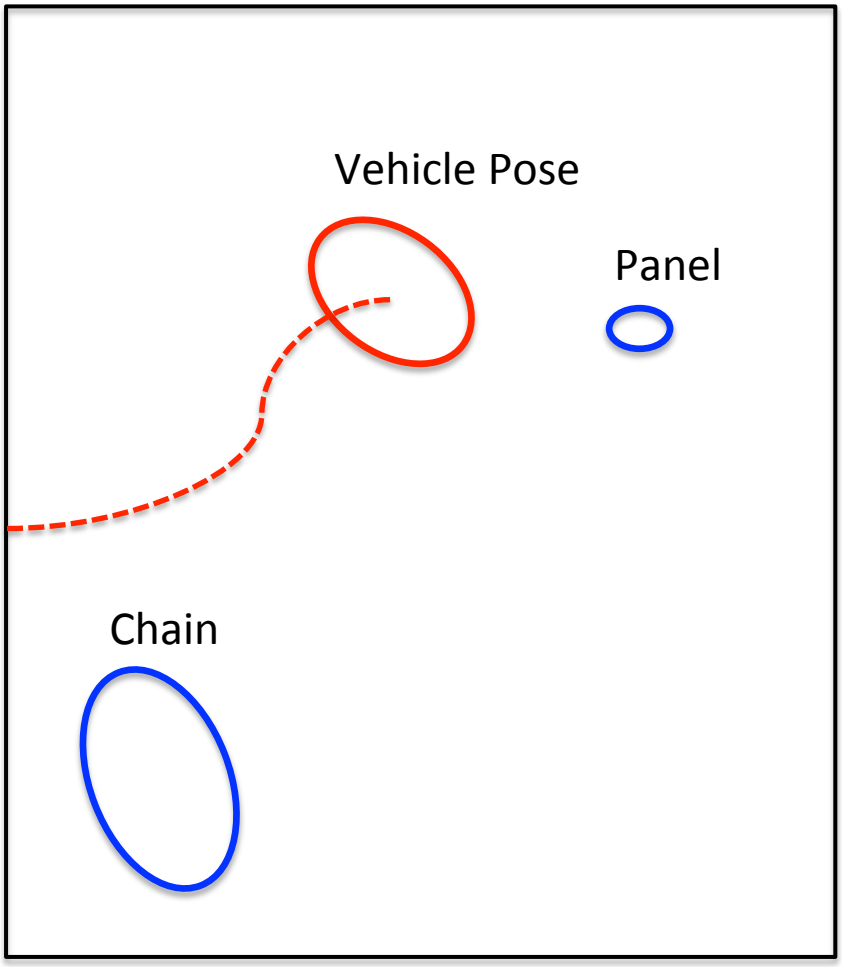

Truth
Estimate
Measurement
0.72
0.715
0.71
0.705
0.7
0.695
0.69
0.685
6.8
7
7.2
7.4
7.6
7.8
8
8.2
8.4
8.6
0.291
0.29
0.289
0.288
0.287
0.286
0.285
0.284
21.8
22
22.2
22.4
22.6
22.8
23
23.2

Extended Kalman Filter for localization and SLAM

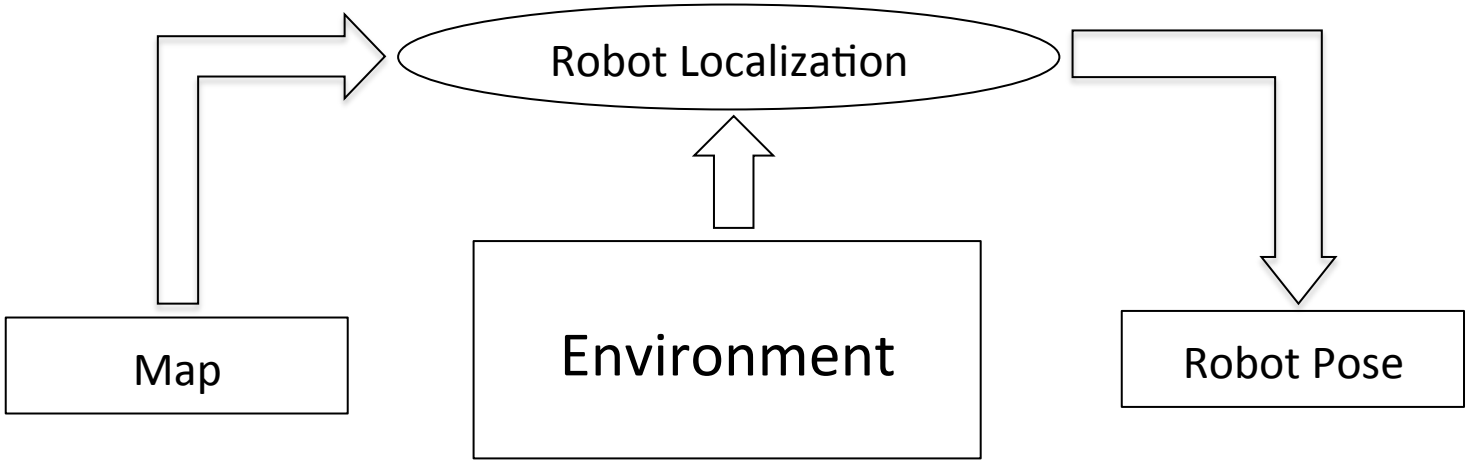
COLA2 implementation

Motivation

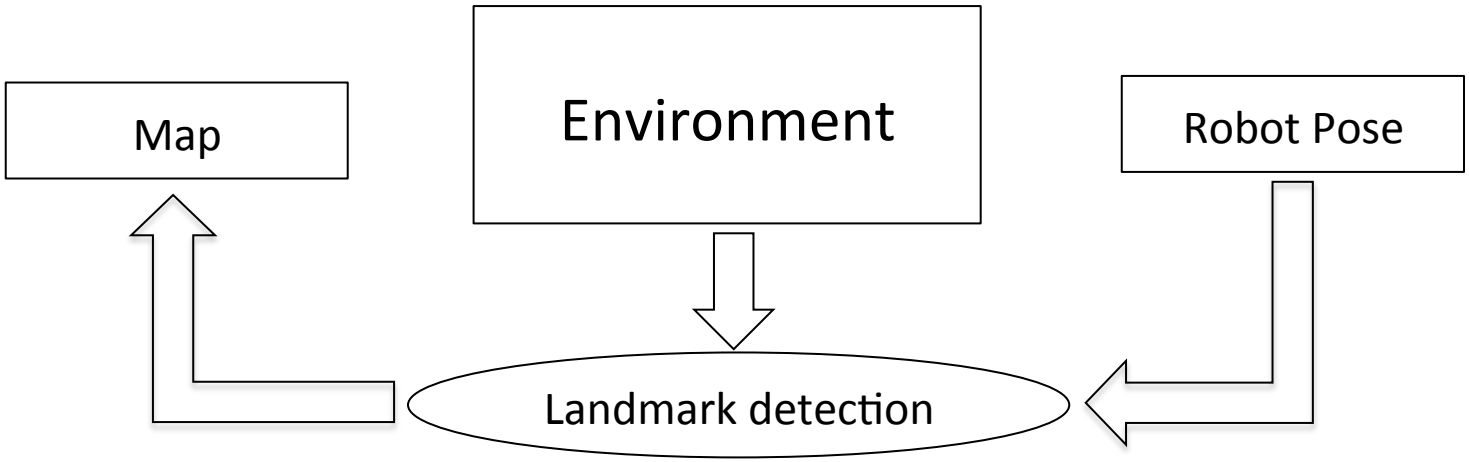
Motivation



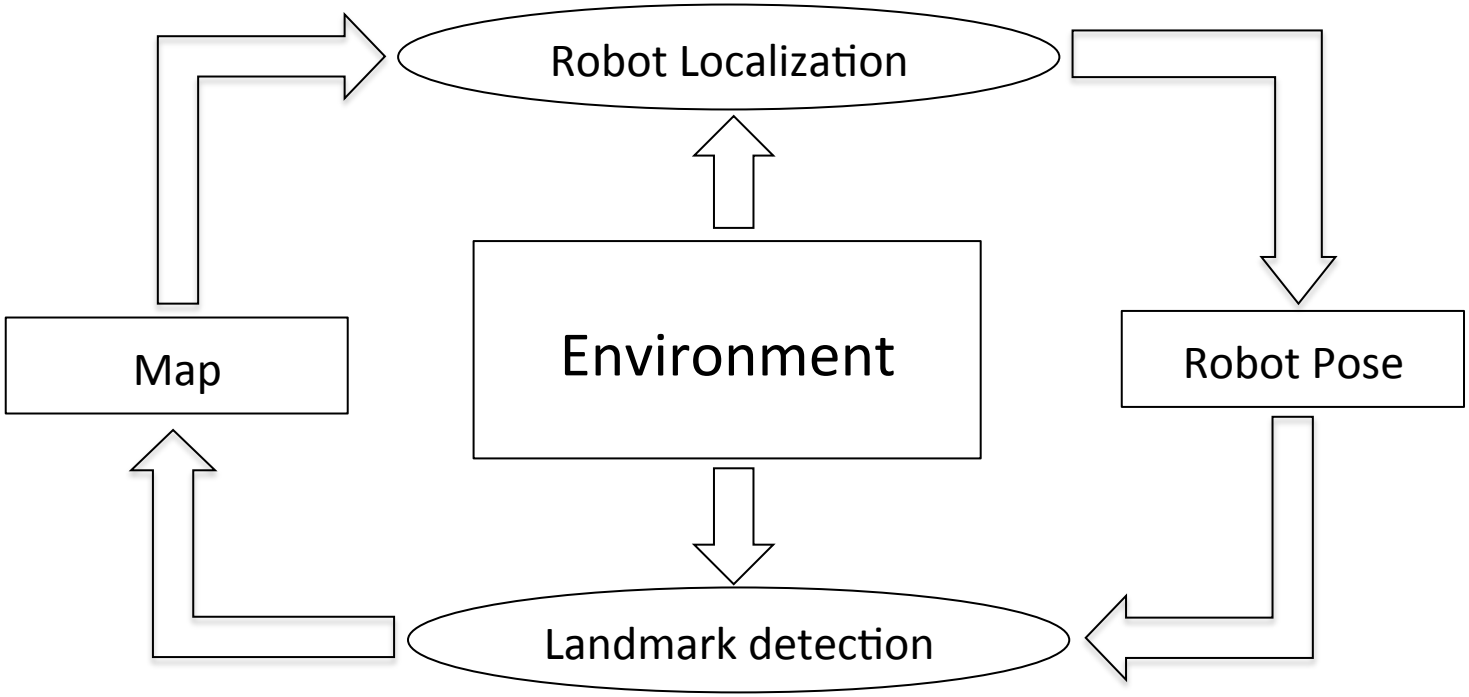
Robot Localization



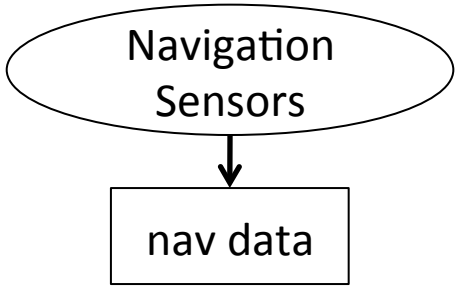
Mapping



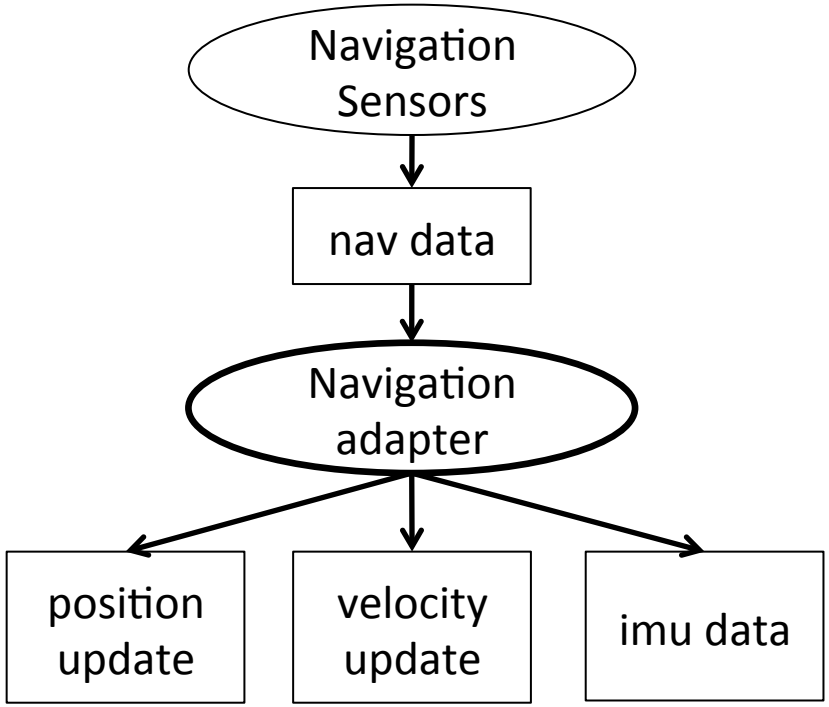
Simultaneous Localization And Mapping



Modular & generic

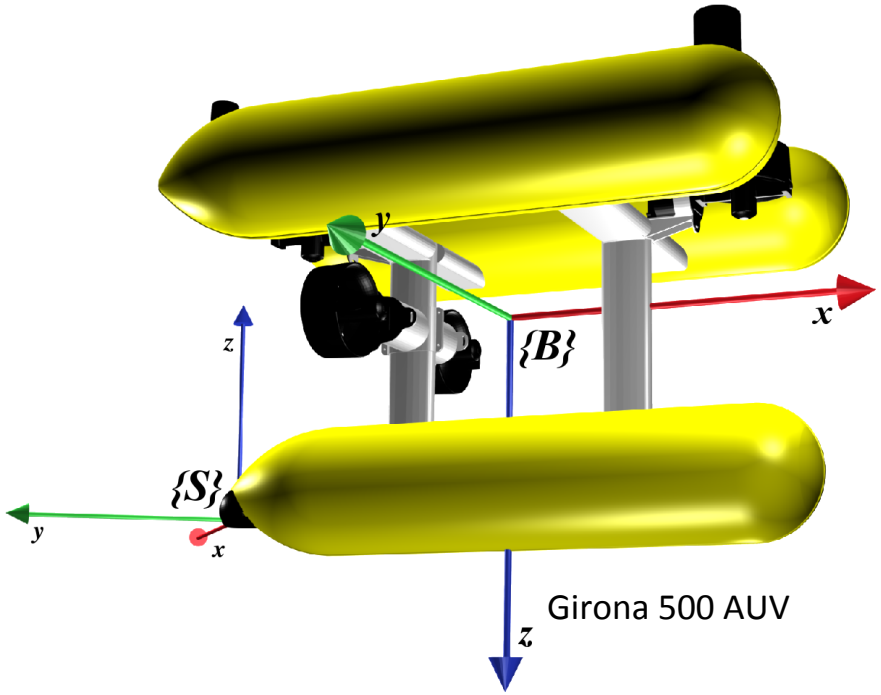


Modular & generic



Navigation adapter

- Transform units to metric system
- Right-handed reference frame
- Transform sensor measurements from {S} to {B}.



Position

$$\mathbf{p}_B = \mathbf{p}_S - \mathbf{RPY} \cdot \mathbf{t}_S$$

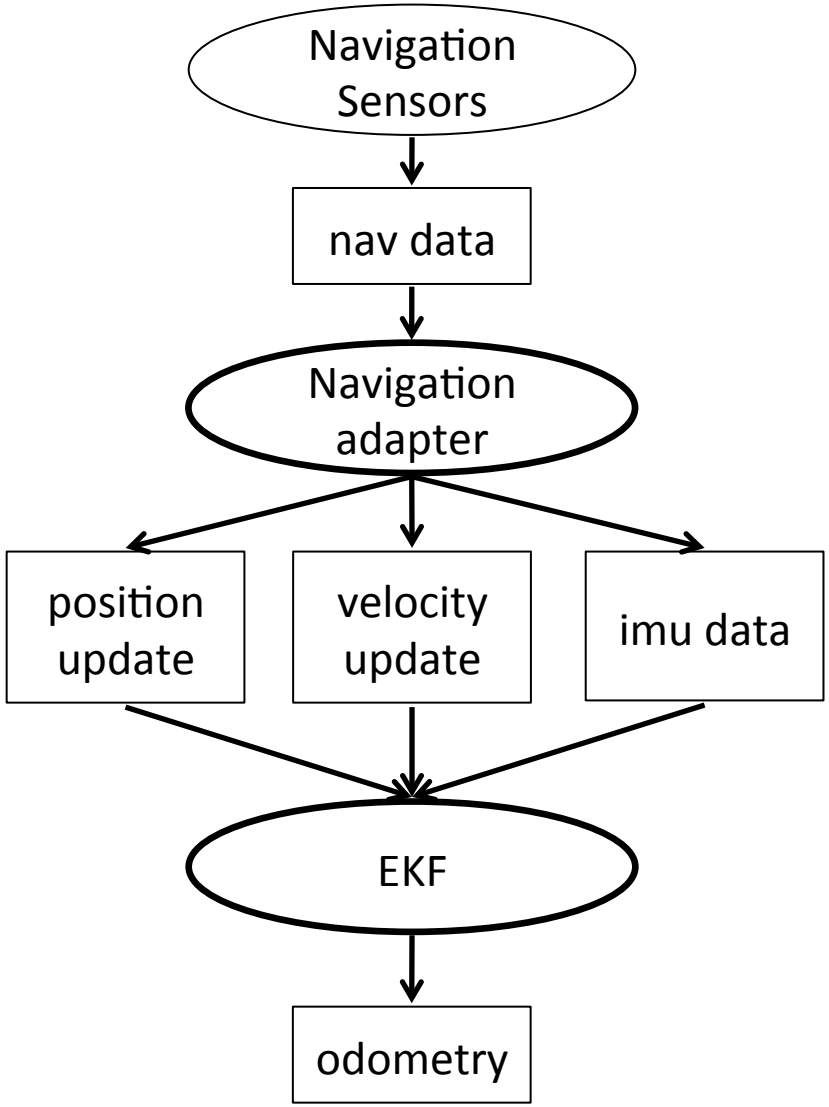
$$\mathbf{Pp}_B = \mathbf{Pp}_S + \mathbf{Jp}_B \cdot \mathbf{P}_{\mathbf{RPY}} \cdot \mathbf{J}^T\mathbf{p}_B$$

Velocity:

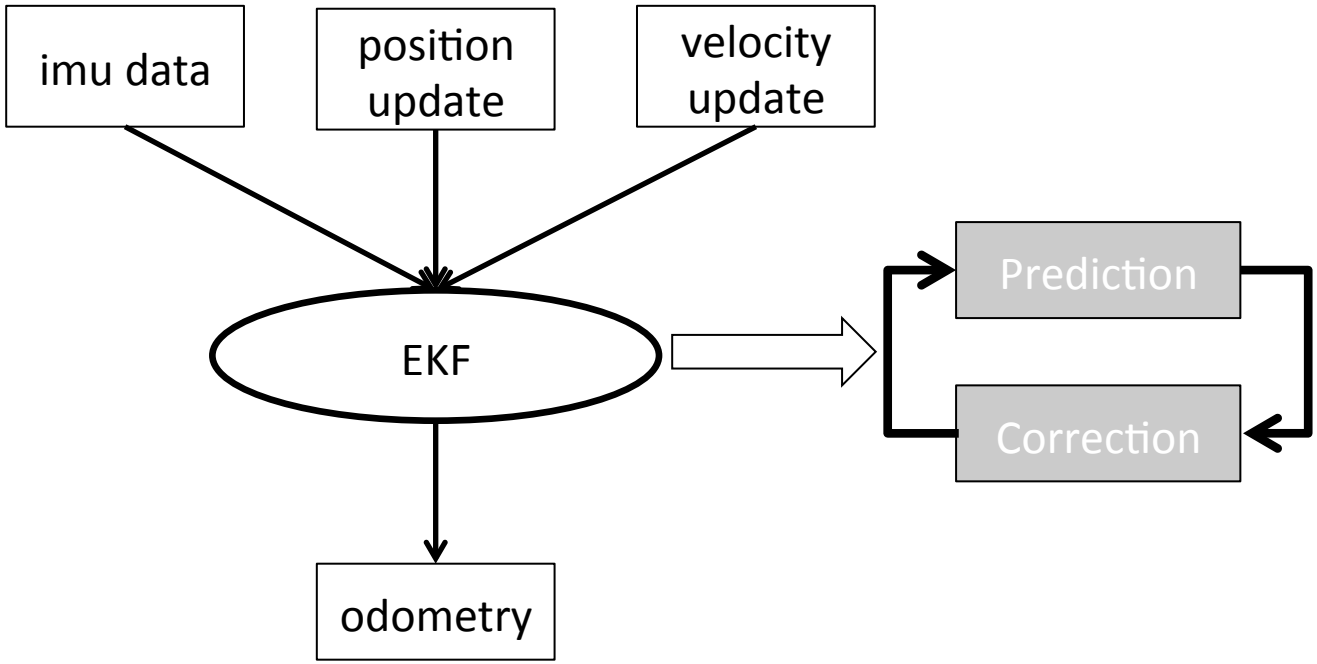
$$\mathbf{v}_B = \mathbf{rot}_S \cdot \mathbf{v}_S - (\mathbf{w}_B \otimes \mathbf{t}_S)$$

$$\mathbf{Pv}_B = \mathbf{rot}_S \cdot \mathbf{Pv}_S \cdot \mathbf{rot}^T_S + \mathbf{J}(\mathbf{w}_B \otimes \mathbf{t}) \cdot \mathbf{Pw}_B \cdot \mathbf{J}^T(\mathbf{w}_B \otimes \mathbf{t})$$

Modular & generic



EKF Localization



EKF localization: Prediction

- Estimates vehicle position [x y z] and linear velocity [u v w].

State vector

$\mathbf{x}_k$ = [x, y, z, u, v, w]

Model

- Constant velocity model

Noise

Zero-mean white Gaussian acceleration noise

$$\mathbf{Q} = \begin{bmatrix} \sigma_{n_u}^2 & 0 & 0 \\ 0 & \sigma_{n_v}^2 & 0 \\ 0 & 0 & \sigma_{n_w}^2 \end{bmatrix}$$

EKF localization: Prediction

$$\mathbf{x}_k^- = \begin{bmatrix} \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ z_{k-1} \end{bmatrix} + \mathbf{R}(\phi_k \theta_k \psi_k) \left(\begin{bmatrix} u_{k-1} \\ v_{k-1} \\ w_{k-1} \end{bmatrix} t + \begin{bmatrix} n_{u_{k-1}} \\ n_{v_{k-1}} \\ n_{w_{k-1}} \end{bmatrix} \frac{t^2}{2} \right) \\ u_{k-1} + n_{u_{k-1}} t \\ v_{k-1} + n_{v_{k-1}} t \\ w_{k-1} + n_{w_{k-1}} t \end{bmatrix}$$

A = Jacobian matrix of *f(x)*

	dx_{K-1}	dy_{K-1}	dz_{K-1}	du_{K-1}	dv_{K-1}	dw_{K-1}
x_K	1	0	0	...	...	...
y_K	0	1	0	...	...	...
z_K	0	0	1	...	...	...
u_K	0	0	0	1	0	0
v_K	0	0	0	0	1	0
w_K	0	0	0	0	0	1

EKF localization: Prediction

$$\mathbf{x}_k^- = \begin{bmatrix} \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ z_{k-1} \end{bmatrix} + \mathbf{R}(\phi_k \theta_k \psi_k) \left(\begin{bmatrix} u_{k-1} \\ v_{k-1} \\ w_{k-1} \end{bmatrix} t + \begin{bmatrix} n_{u_{k-1}} \\ n_{v_{k-1}} \\ n_{w_{k-1}} \end{bmatrix} \frac{t^2}{2} \right) \\ u_{k-1} + n_{u_{k-1}} t \\ v_{k-1} + n_{v_{k-1}} t \\ w_{k-1} + n_{w_{k-1}} t \end{bmatrix}$$

Jacobian matrix of f(x)

```
A = Eigen::MatrixXd::Identity(6, 6);
A(0,3) = cos(pitch)*cos(yaw)*t;
A(0,4) = -cos(roll)*sin(yaw)*t + sin(roll)*sin(pitch)*cos(yaw)*t;
A(0,5) = sin(roll)*sin(yaw)*t + cos(roll)*sin(pitch)*cos(yaw)*t;
A(1,3) = cos(pitch)*sin(yaw)*t;
A(1,4) = cos(roll)*cos(yaw)*t + sin(roll)*sin(pitch)*sin(yaw)*t;
A(1,5) = -sin(roll)*cos(yaw)*t + cos(roll)*sin(pitch)*sin(yaw)*t;
A(2,3) = -sin(pitch)*t;
A(2,4) = sin(roll)*cos(pitch)*t;
A(2,5) = cos(roll)*cos(pitch)*t;
```

EKF localization: Prediction

- Instead of

  $\overline{x}_K = A \cdot x_{K-1}$ and $\overline{P}_K = A * P_{K-1} * A^T + Q$

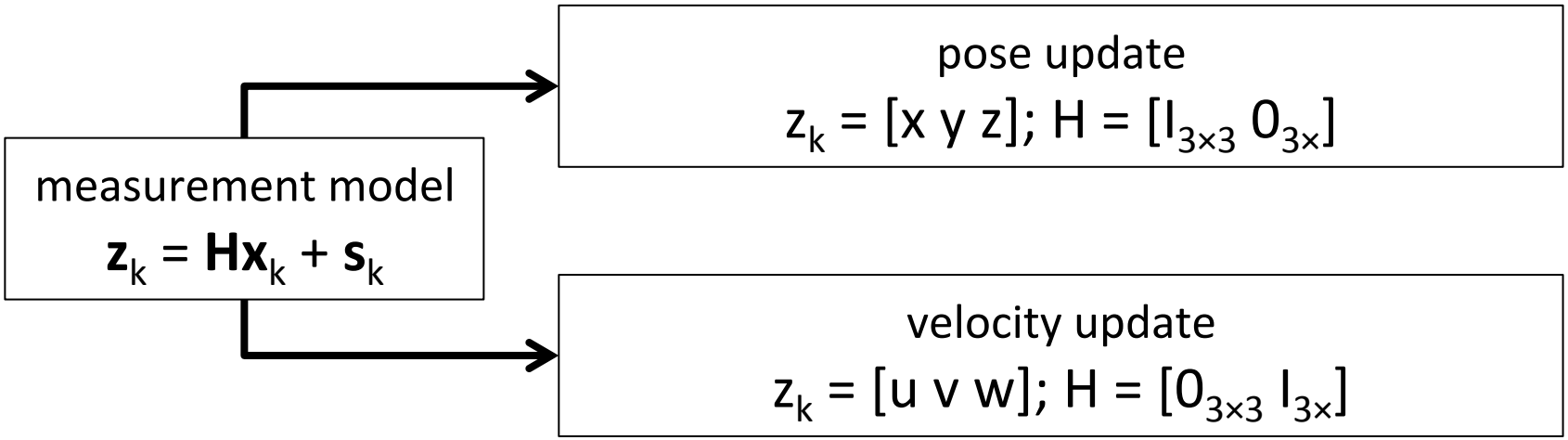
- we have

  $\overline{x}_K = f(x_{K-1})$ and $\overline{P}_K = A * P_{K-1} * A^T + Q$

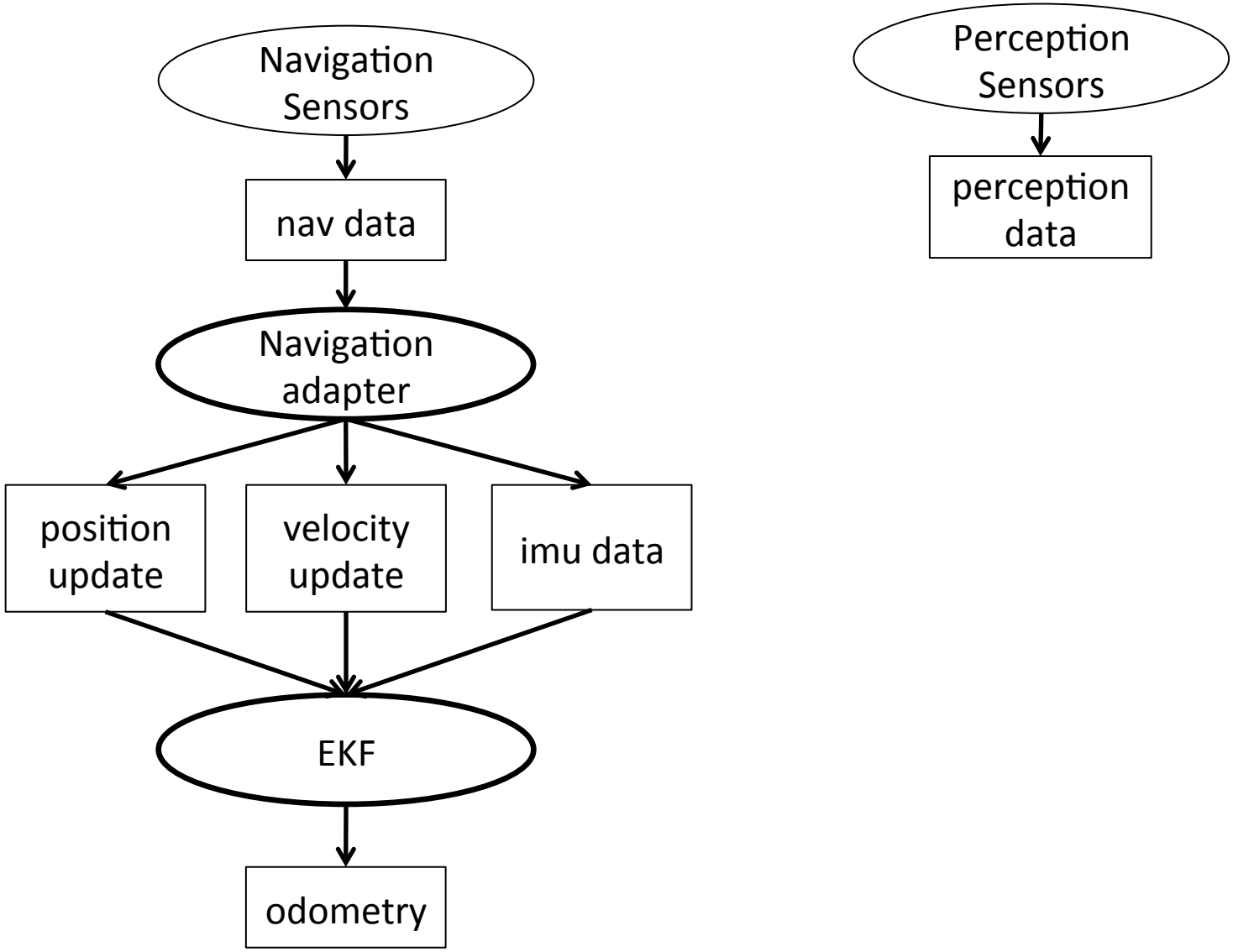
  where A = J(x)

EKF localization: Correction

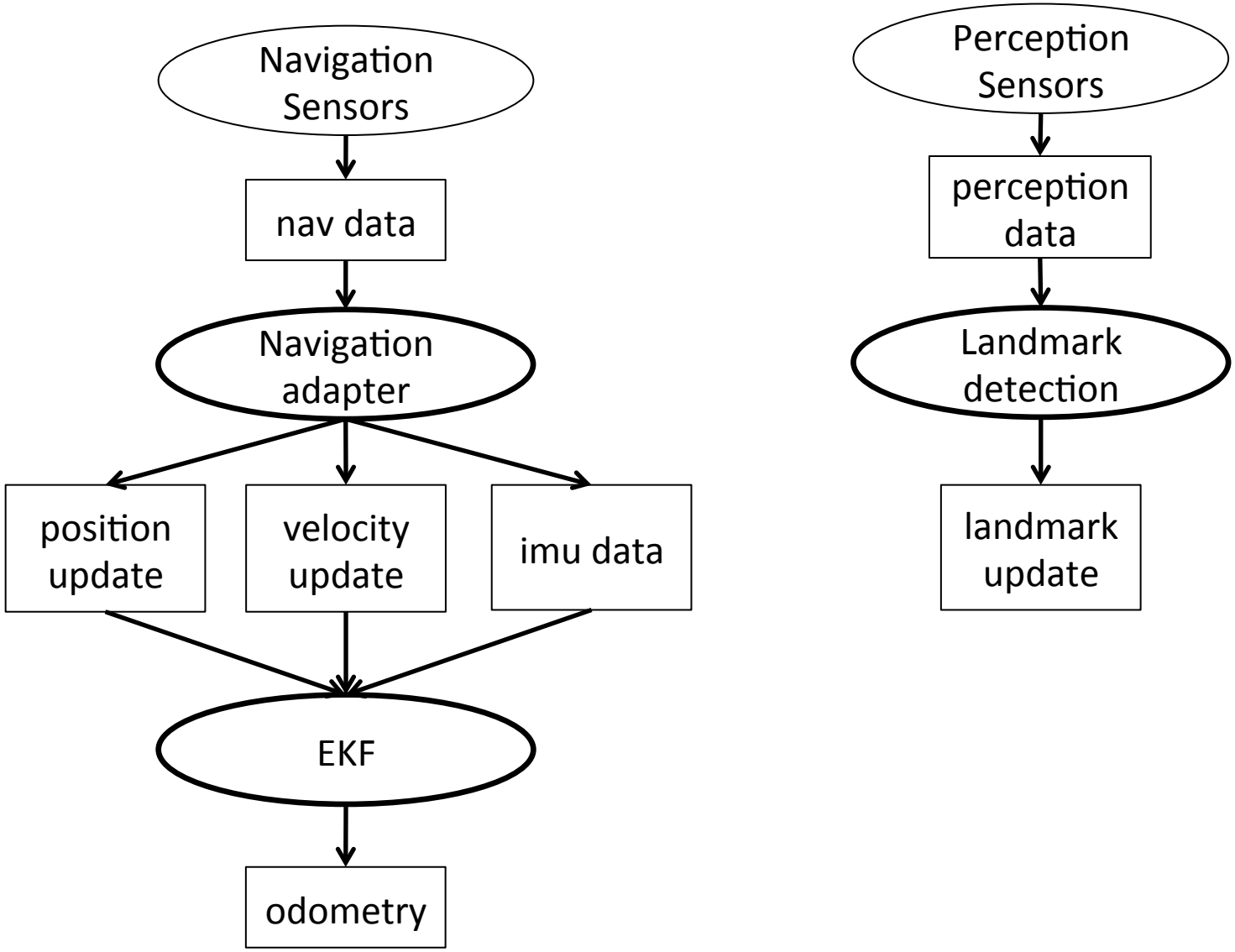
- Two **linear** measurement updates are applied in the filter



Modular & generic

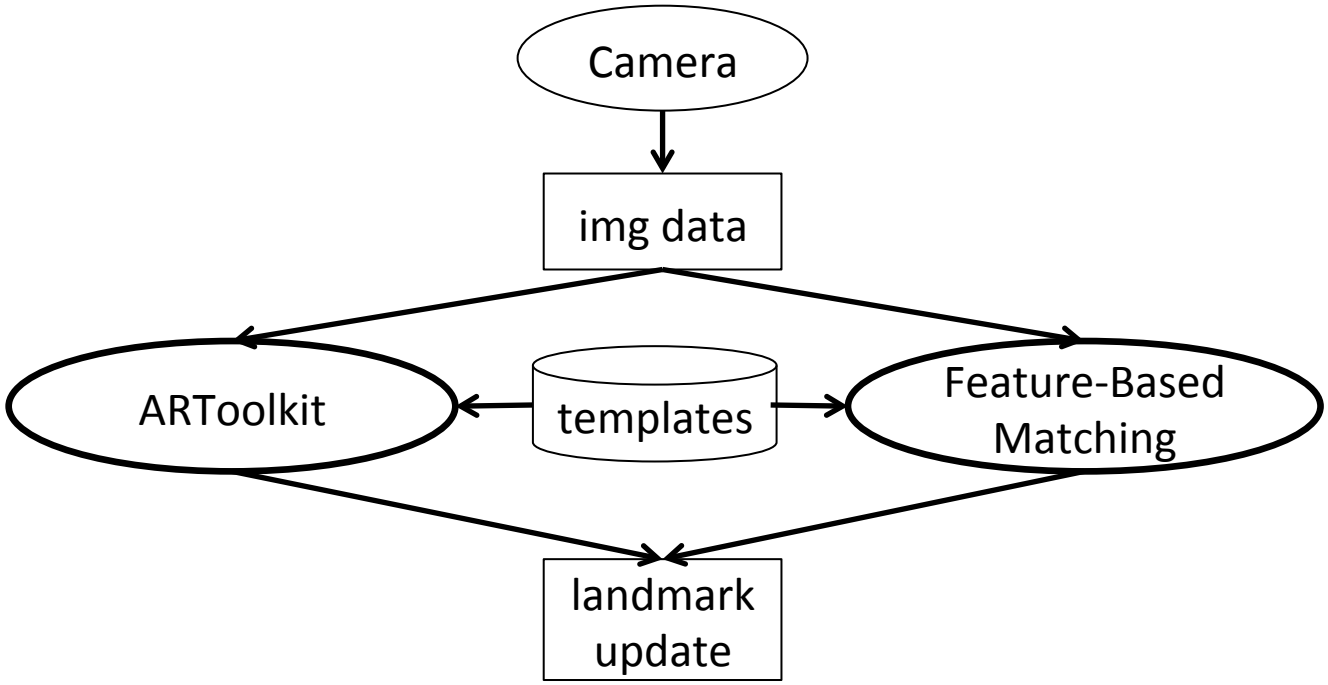


Modular & generic



Vision-based landmark detectors

- Compute landmarks pose, by comparing the images from the camera against an *a priori* known template.
- Two proposed methods: ARToolkit & feature-based matching

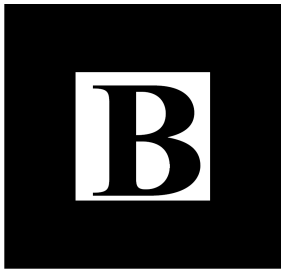


Vision-based landmark detectors

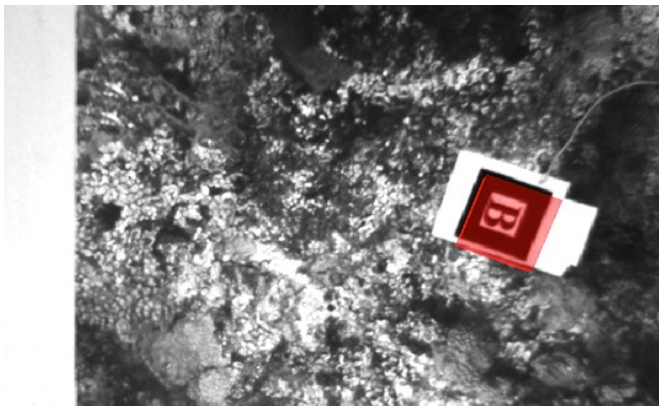
ARToolKit

- Grey scale monocular camera
- Lower resources
- Need to add extra markers
- Bad covariance estimation:
  $\mathbf{R} = (\mathbf{I}_{3\times3} \cdot d^2) \cdot \mathbf{c}$

Template
20x20 cm



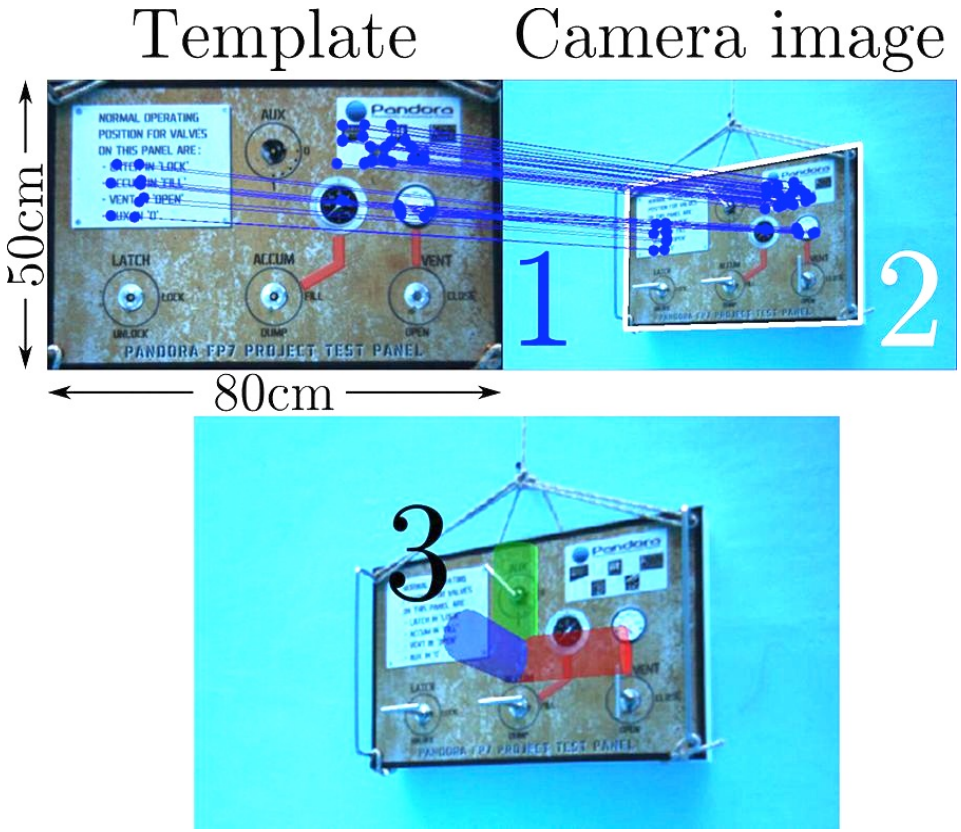
Camera image



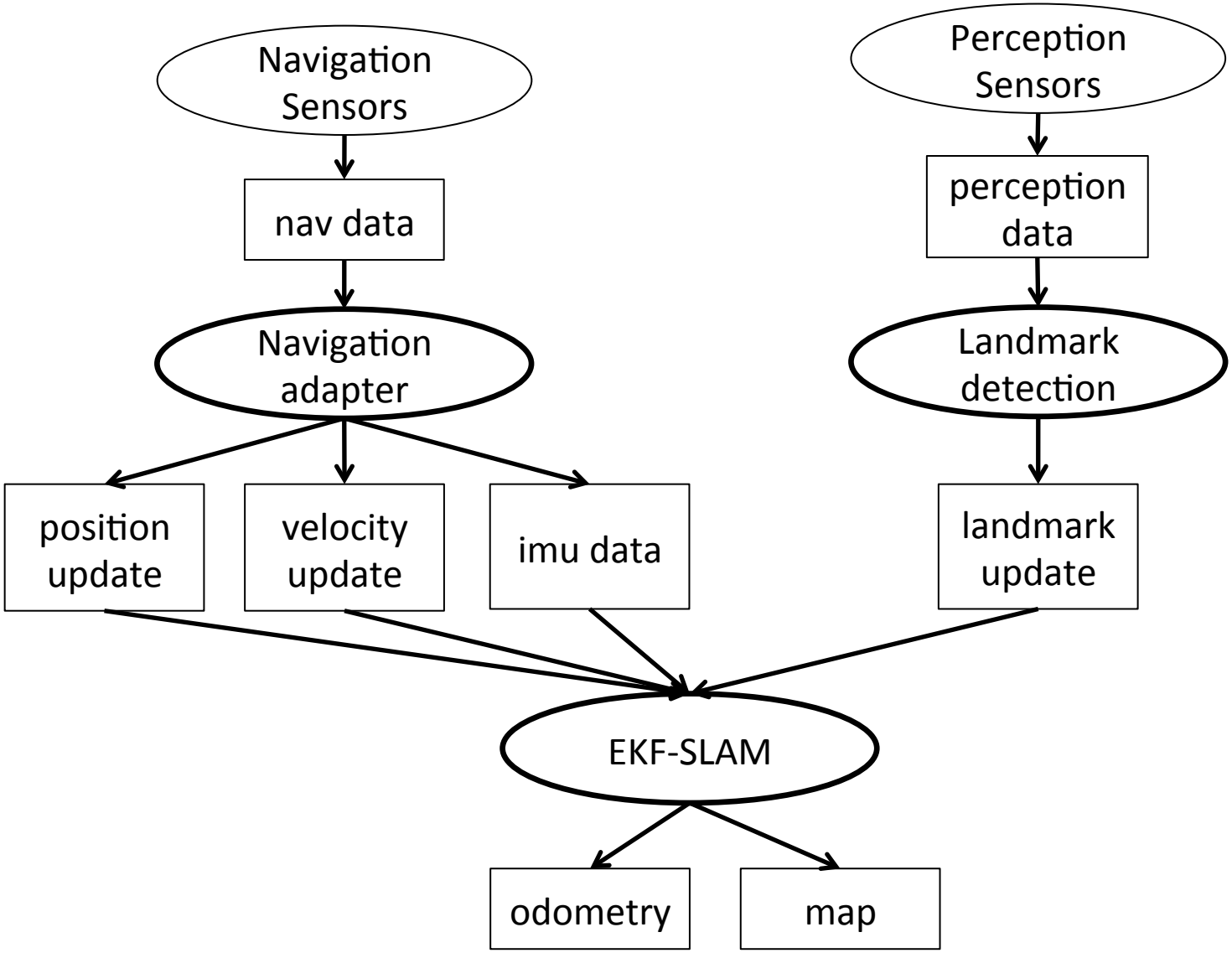
Vision-based landmark detectors

Feature-based Matching

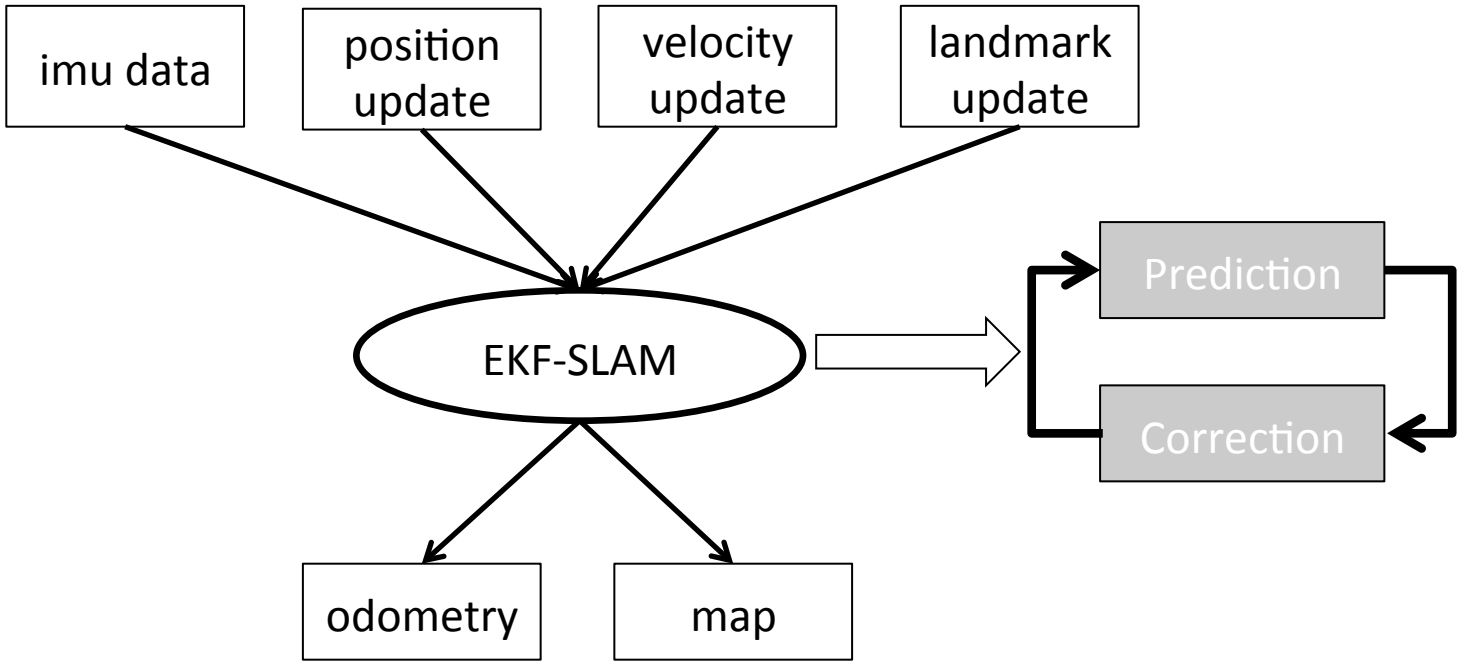
- Grey scale monocular camera
- Uses ORB feature detector
- No need of extra markers
- More complex but real-time



Modular & generic



EKF-SLAM Localization



EKF-SLAM algorithm: Prediction

- Estimates vehicle position [x y z] and linear velocity [u v w], and maps the position of landmarks [lx_i ly_i lz_i] identified by a visual detector.

State vector

$\mathbf{x}_k = [x, y, z, u, v, w, lx_0, ly_0, lz_0 \ldots, lx_n, ly_n, lz_n]$

Model

- Constant velocity for the vehicle
- Static position for the landmarks

Noise

Zero-mean white Gaussian acceleration noise

$$\mathbf{Q} = \begin{bmatrix} \sigma_{n_u}^2 & 0 & 0 \\ 0 & \sigma_{n_v}^2 & 0 \\ 0 & 0 & \sigma_{n_w}^2 \end{bmatrix}$$

EKF localization: Model

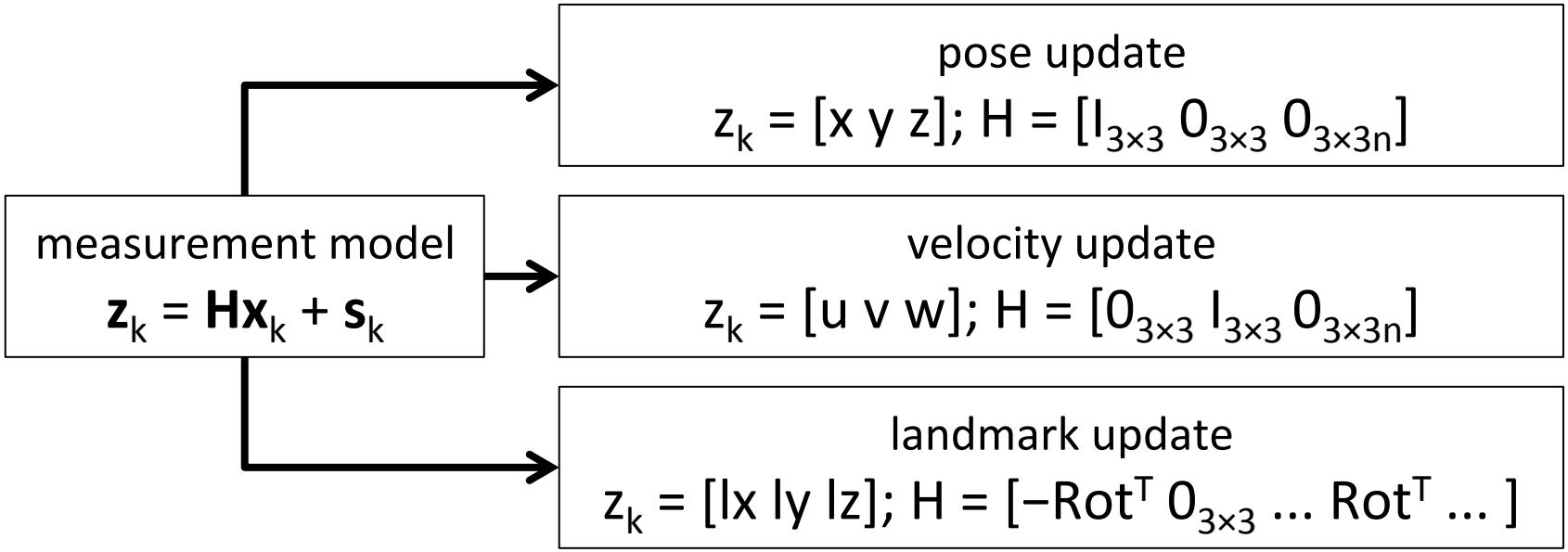
$$\mathbf{x}_k^- = \begin{bmatrix} \begin{bmatrix} x_{k-1} \\ y_{k-1} \\ z_{k-1} \end{bmatrix} + \mathbf{R}(\phi_k \theta_k \psi_k) \left(\begin{bmatrix} u_{k-1} \\ v_{k-1} \\ w_{k-1} \end{bmatrix} t + \begin{bmatrix} n_{u_{k-1}} \\ n_{v_{k-1}} \\ n_{w_{k-1}} \end{bmatrix} \frac{t^2}{2} \right) \\ u_{k-1} + n_{u_{k-1}} t \\ v_{k-1} + n_{v_{k-1}} t \\ w_{k-1} + n_{w_{k-1}} t \\ lx_{1_{k-1}} \\ ly_{1_{k-1}} \\ lz_{1_{k-1}} \\ \dots \end{bmatrix}$$

Jacobian matrix of f(x)

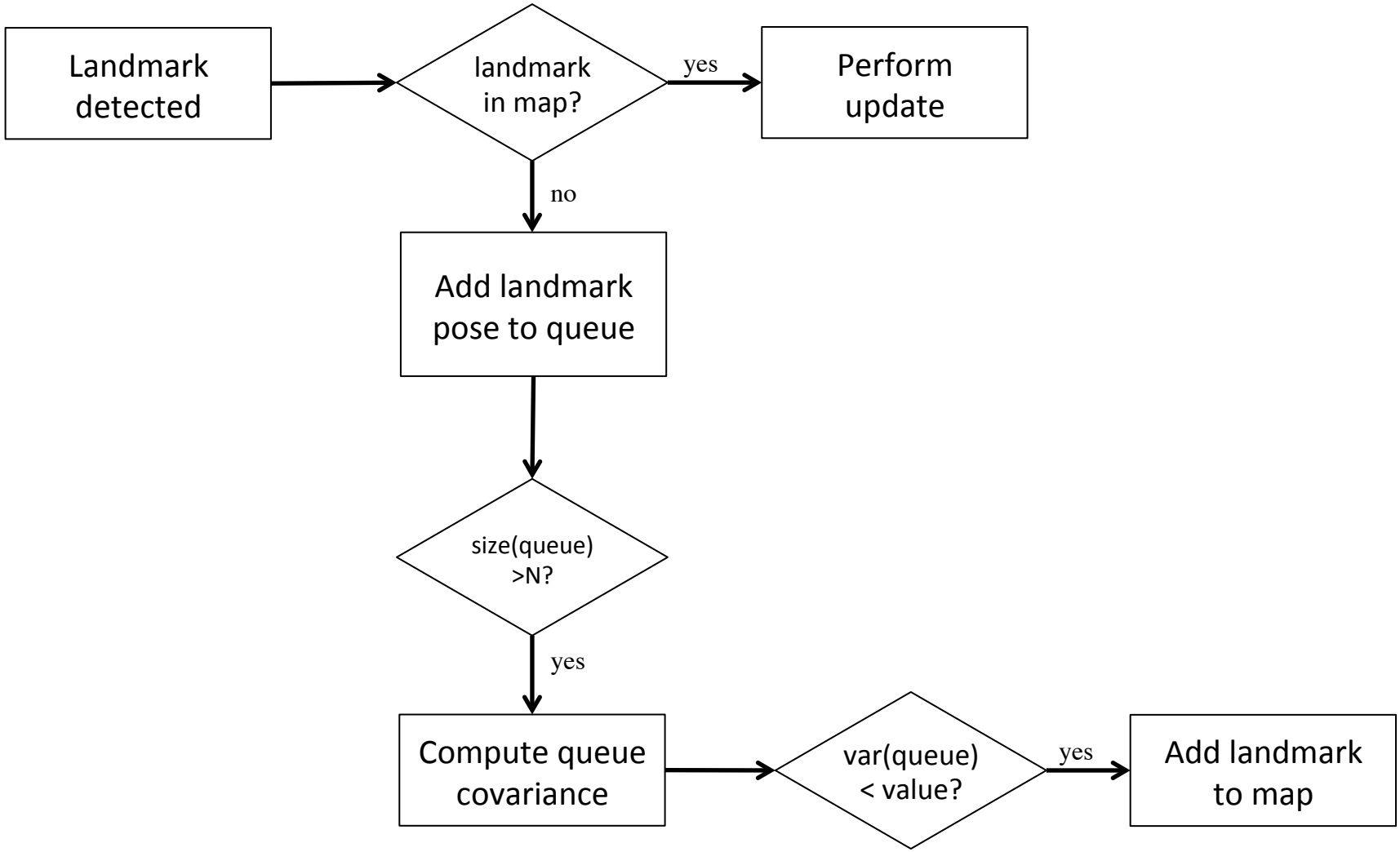
	dx_{K-1}	dy_{K-1}	dz_{K-1}	du_{K-1}	dv_{K-1}	...
x_{K-1}	...					
y_{K-1}		...				
z_{K-1}			...			
u_{K-1}				1		
v_{K-1}					1	
...						...

EKF-SLAM navigation: Correction

- Three **linear** measurement updates are applied in the filter

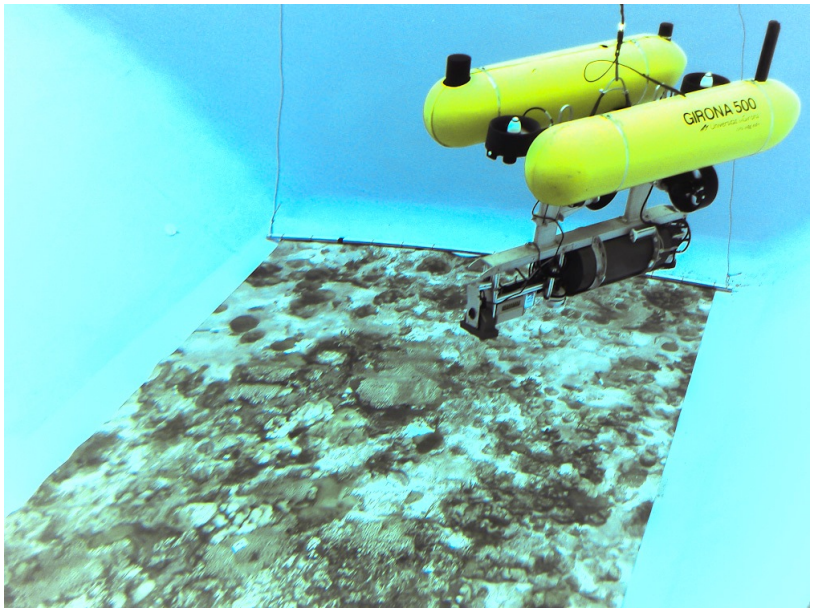


Filter initialization



Experiments setup

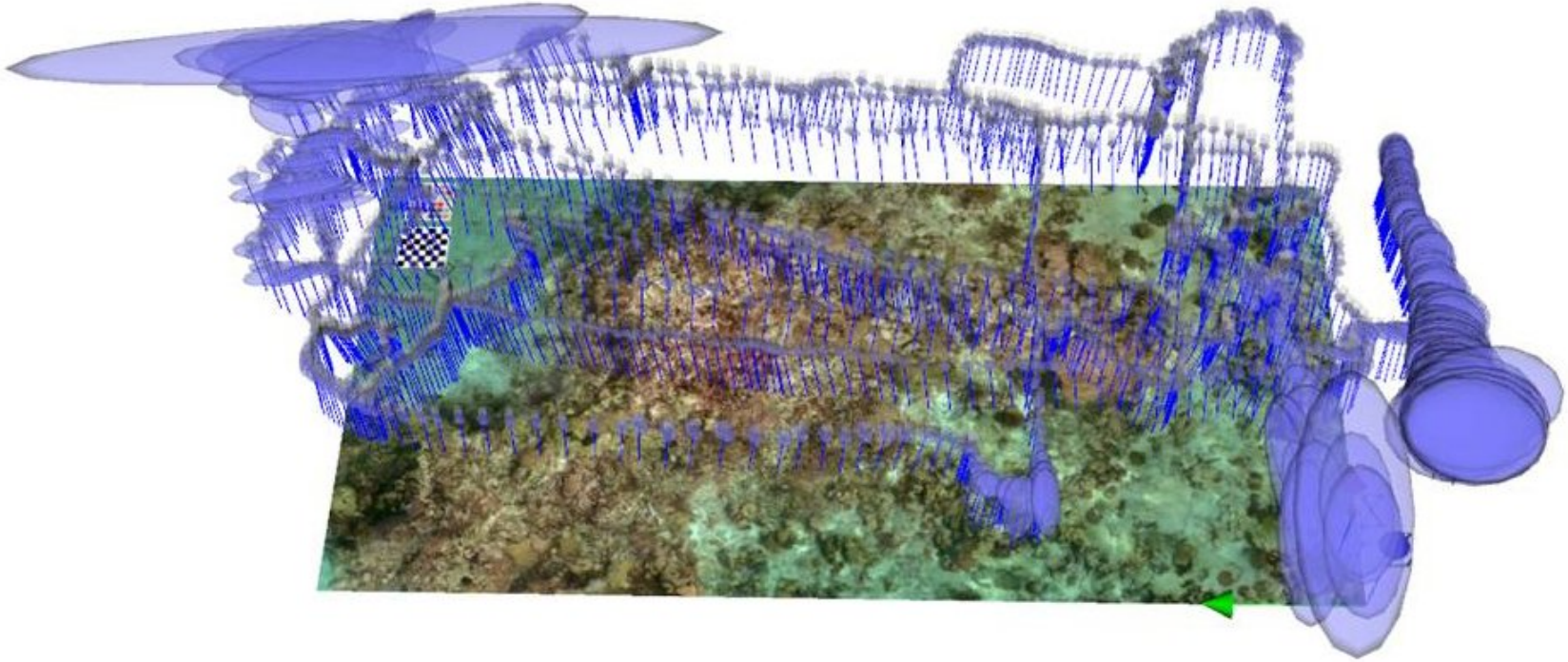
- Vehicle sensors: IMU, DVL, depth sensor and camera.
- Water tank of 16 x 8 x 5m
- Two landmarks
- Ground-truth poster at the bottom of the water tank



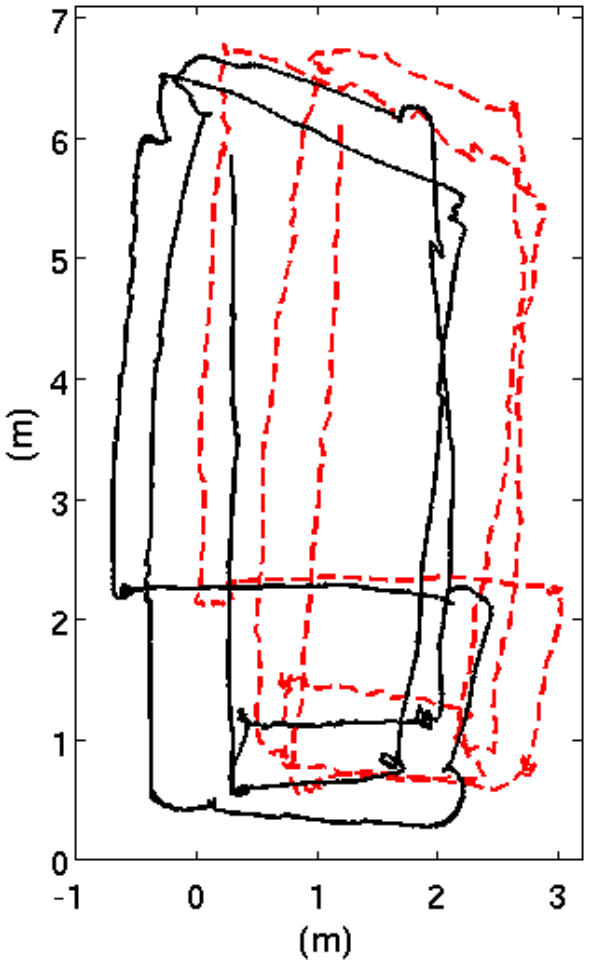
Girona 500 AUV was teleoperated for about 10 minutes. Visual detector algorithm and EKF-SLAM were running on-board the vehicle.

Ground truth

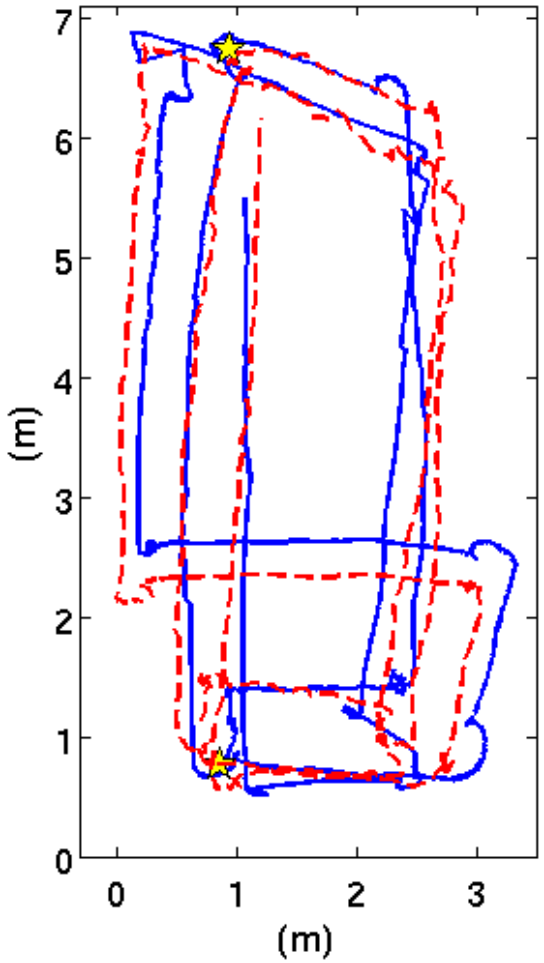
- To obtain estimates of absolute camera poses, together with their uncertainty, camera images are registered to the original image of the poster.



Trajectory results

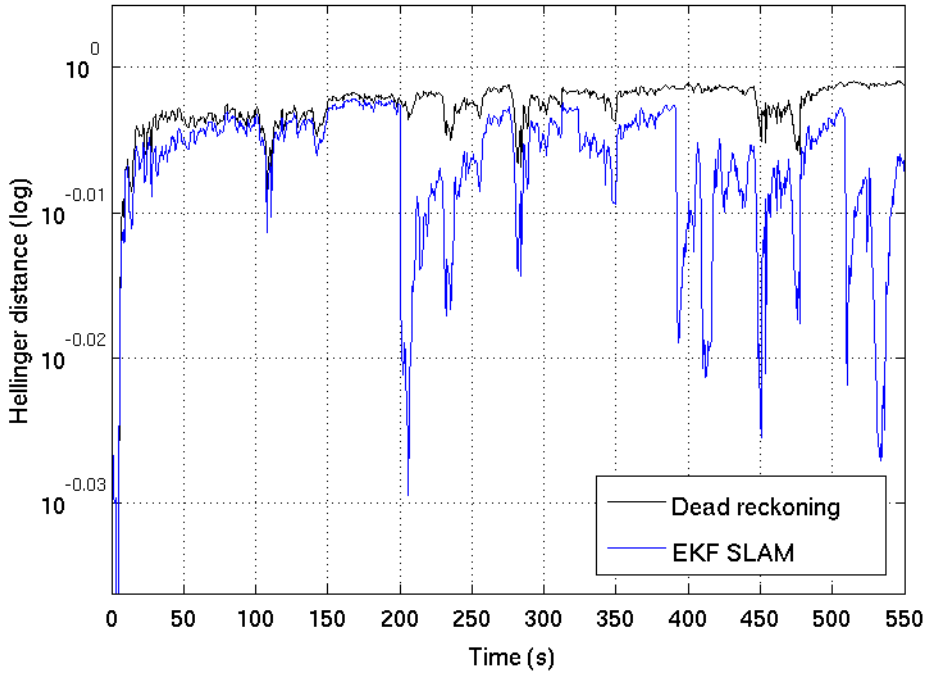


Dead reckoning Vs.
Ground-truth (dashed)



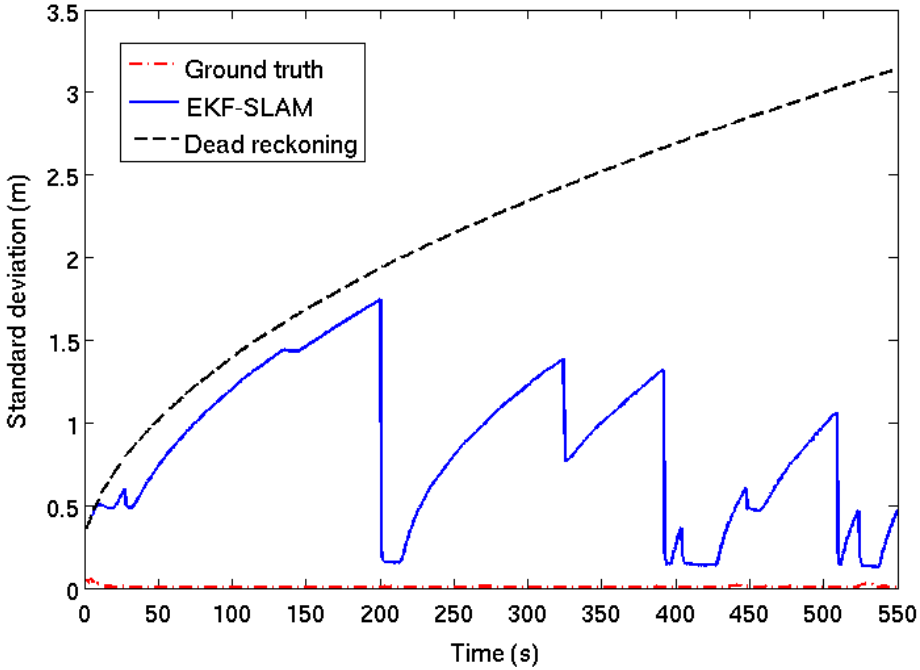
EKF-SLAM Vs.
Ground-truth (dashed)

Error comparison



Hellinger distance in the estimated trajectories from the dead reckoning and EKF-SLAM

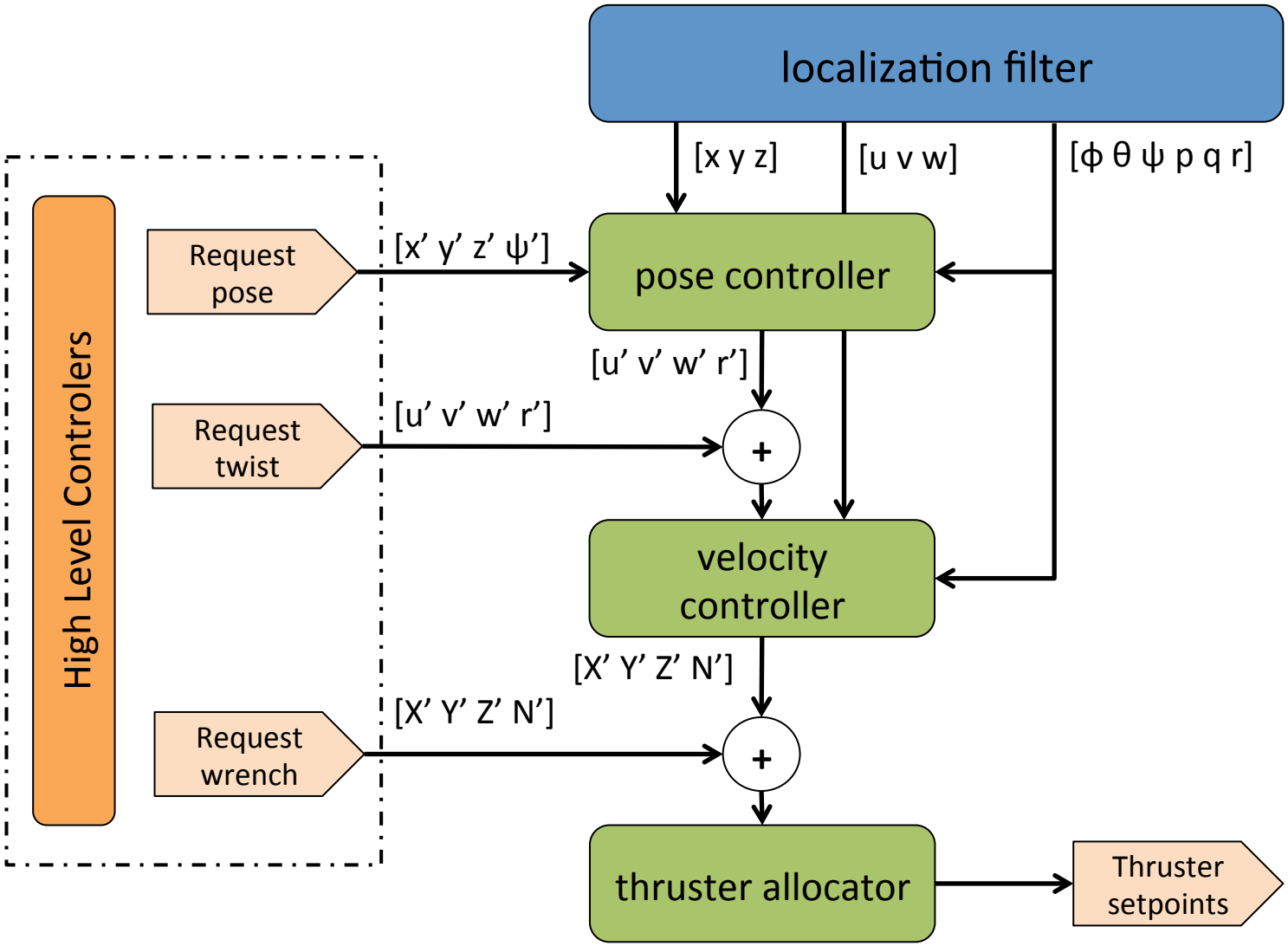
Standard deviation for axis X over time between dead reckoning, EKF-SLAM and the ground-truth.



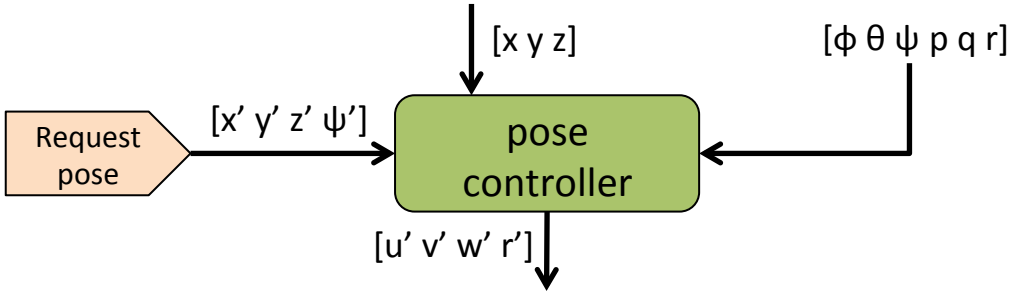
COLA2 control scheme

Narcís Palomeras
Universitat de Girona

COLA2 control scheme



Low level controller: position control



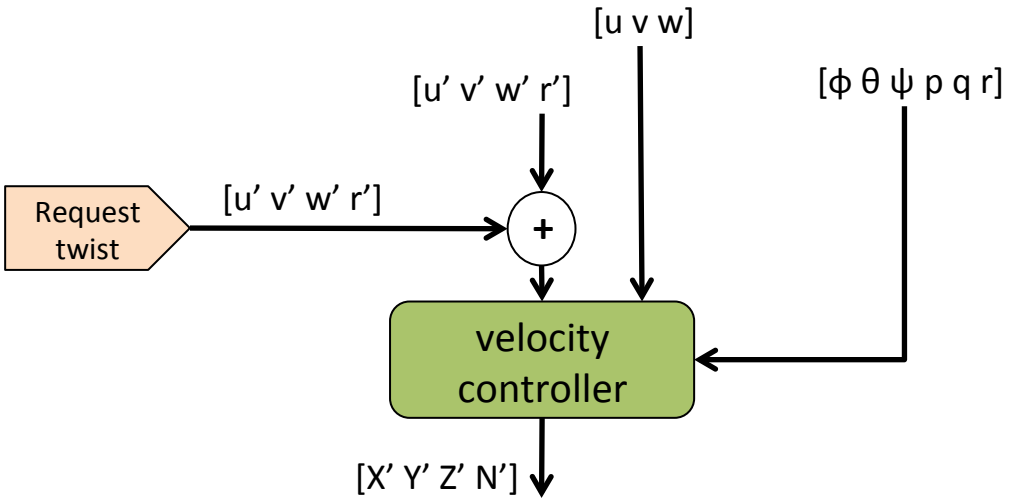
$$v'(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{d}{dt} e(t) \right)$$

$$\begin{bmatrix} e_x \\ e_y \\ 1 \end{bmatrix} = \begin{pmatrix} \mathbf{R}(\psi)^T & -\mathbf{R}(\psi)^T & \begin{bmatrix} x \\ t \end{bmatrix} \\ \mathbf{0}_{1\times 2} & 1 \end{pmatrix} \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

$$e_z = z' - z$$

$$e_\psi = normalized(\psi' - \psi)$$

Low level controller: velocity control

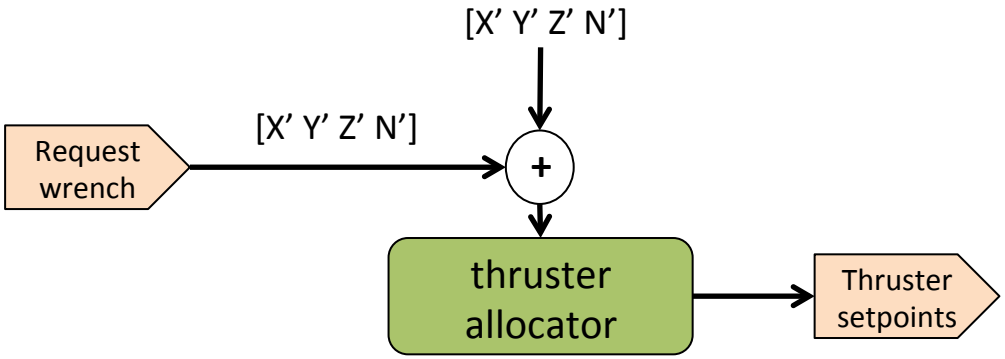


$$\tau' = PID(\nu, \nu') + Model(\nu')$$

$$v'(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{d}{dt} e(t) \right)$$

$$\tau_n = \mathbf{A}*v'^2 + \mathbf{B}*v' + \mathbf{C}$$

Low level controller: thruster allocator



Thruster allocation matrix

$[th_1\ th_2\ th_3\ th_4\ th_5] = [X'\ Y'\ X'\ N'] *$

	th_1	th_2	th_3	th_4	th_5
X	1	1	0	0	0
Y	0	0	0	0	1
Z	0	0	1	1	0
N	0.4	-0.4	0	0	0

+ Thruster model to transform force per thruster to setpoint

Combine requests

1. All requests are accumulated in a queue
2. If the queue already contains a previous request from the same node, the old one is erased
3. Requests are combined at 10Hz (low controller frequenzy):
   a. Requests older than 0.15s are deleted
   b. Remaining requests are sorted by priority
   c. Requests are combined

Combine requests

req: teleoperation
priority: 40
time: ...

	v	u	w	p	q	r
value	0.5	0	0	0	0	0.3
disabled_axis	F	T	T	T	T	F

+

req: altitude_control
priority: 10
time: ...

	v	u	w	p	q	r
value	0	0	0.2	0	0	0
disabled_axis	T	T	F	T	T	T

=

req: teleoperation
priority: 40
time: ...

	v	u	w	p	q	r
value	0.5	0	0.2	0	0	0.3
disabled_axis	F	T	F	T	T	F

Combine requests

req: teleoperation
priority: 40
time: ...

	v	u	w	p	q	r
value	-1.	0	0.4	0	0	0
disabled_axis	F	T	F	T	T	T

+

req: goto
priority: 10
time: ...

	v	u	w	p	q	r
value	0.4	0	0.2	0	0	0.1
disabled_axis	F	T	F	T	T	F

=

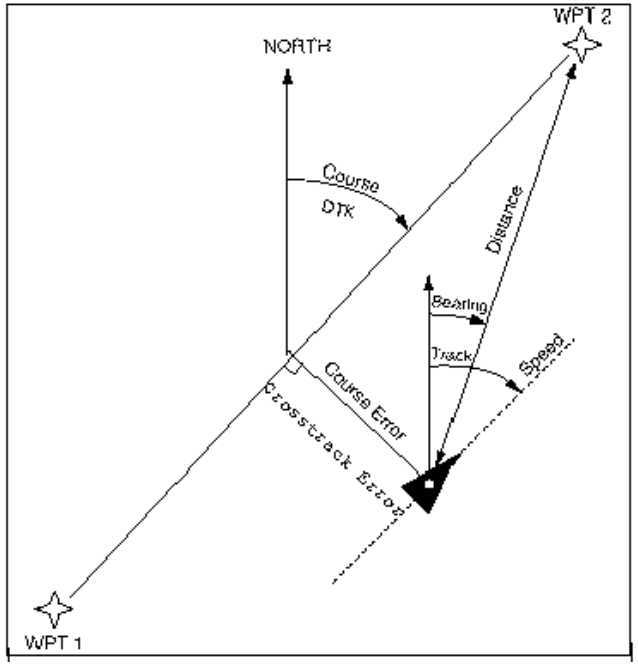
req: teleoperation
priority: 40
time: ...

	v	u	w	p	q	r
value	0.5	0	0.4	0	0	0
disabled_axis	F	T	F	T	T	T

High Level Controllers

- Move to position [x y z Ψ] → pose req [x y z Ψ]
- Go to waypoint [x y z] → pose req [z Ψ] & twist req [u]
- Line of sight with cross tracking error [x y z]
- Follow trajectory

  $\{[x_0\ y_0\ z_0], \ldots [x_n\ y_n\ z_n]\}$
- Altitude controller → pose req [z]
- …



1. Software installation

It is required to have a computer with UBUNTU 14.04 LTS and ROS Jade already installed to not loose time preparing the system. To follow the course it will be also necessary to install the UWSim simulator and the COLA2 software architecture:

* To install UBUNTU 14.04
  http://releases.ubuntu.com/14.04/
* To install ROS
  http://wiki.ros.org/ROS/Installation
* To install UWSim:
  sudo apt-get install ros-jade-uwsim
  Once done, run UWSim for the first time. It will download some extra content.
  roscore & rosrun uwsim uwsim
* To install dependencies and additional libraries:
  sudo apt-get install ros-jade-rosbridge-server ros-jade-octomap-ros ros-jade-ompl

* To install COLA2 architecture:
  Once your system is ready (UBUNTU and ROS installed), you have to create a catkin workspace (http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment) to install the packages that form the COLA2 control architecture. Next, install the follow packages:
  cd ~/catkin_ws/src/
  git clone https://bitbucket.org/udg_cirs/auv_msgs.git
  git clone https://bitbucket.org/udg_cirs/cola2_core.git
  git clone https://bitbucket.org/udg_cirs/cola2_s2.git

* To install Mapping and Path Planning modules:
  cd ~/catkin_ws/src/
  git clone https://bitbucket.org/udg_cirs/excellabust_croatia_16.git

To check that everything has been correctly installed do:
  cd ~/catkin_ws/
  catkin_make

2. How to run mapping and planning modules

After installing all the dependencies and repositories previously mentioned, there are two test scenarios to visualize, simulate and test different functionalities that are part of the software architecture. The first scenario resembles a natural-like environment composed of seamounts that form an underwater canyon. The second one is a 3-dimensional (3D) model of a breakwater structure composed of a series of concrete blocks. In order to simulate the SPARUS-II AUV in either scenario, it is necessary to execute the corresponding launch file:

roslaunch croatia16_planning start_s2_croatia16_test_canyon.launch # (canyon)
or
roslaunch croatia16_planning start_s2_croatia16_test_blocks.launch # (breakwater structure)

One of those functionalities that can be tested is the path-planning module, which uses the Open Motion Planning Library (OMPL)[1] to solve start-to-goal queries. In order to use it, once the test scenario and the AUV have been loaded (UWSim and RViz), there is a configuration file that

[1] http://ompl.kavrakilab.org/

includes different parameters for the path planner, including the Configuration Space (C-space) boundaries, the start and goal positions (query), the planner used to solve the query (sample-based methods), the computation time dedicated to attempt to solve the query, the depth (constant) at which the AUV navigates and the AUV used (Girona500 or SPARUS-II). To modify such configuration file:

rosed croatia16_planning tests_canyon_croatia16.yaml # (canyon)
or
rosed croatia16_planning tests_blocks_croatia16.yaml # (breakwater structure)

Once the path-planning parameters have been defined, it is necessary to run the planner to generate the mission file, which contains the collision-free path that solves the start-to-goal query. The planner uses a mapping module which provides an Octomap[2], which works as a 3D representation (map) of the environment. For this demonstration, we assume all the environment as explored, thus a complete map is available a priori, as visualized in Rviz. To solve query using such map, it is necessary to run the corresponding launch file in a different terminal:

roslaunch croatia16_planning croatia16_test_canyon_planning.launch # (canyon)
or
roslaunch croatia16_planning croatia16_test_blocks_planning.launch # (breakwater structure)

Doing so, the planner will attempt to solve the query and will visualize a solution if there is one. Now, in order to send the AUV to conduct the mission and follow the resulting path, it is required first to load the mission and then to tell the pilot (software architecture) to follow the mission, which is specified as a series of waypoints. To do this, execute in a different terminal the following commands:

To load the mission (replace path_to_catkin_ws according to your local workspace installation

rosservice call /cola2_control/load_trajectory "mystring:
'/path_to_catkin_ws/src/excellabust_croatia_16/croatia16_planning/missions/mission.yaml'"

#To execute the mission
rosservice call /cola2_control/enable_trajectory

3. Exercises

Once the basic planning module has been tested, there is a series of exercises to extend its functionality.

3.1. Understanding the planning module

The planning module uses OMPL to solve a start-to-goal query. The basic interface to solve such type of queries is coded and documented in offline_planner_R2.cpp. This file contains the class OfflinePlannerR2, in which Offline means that the environment is assumed as explored and the path is calculated offline (i.e., before conducting the mission)[3] and R2 refers to the dimension of the C-Space (RxR, for x and y coordinates). The class constructor

[2] https://octomap.github.io/

[3] For an example of online mapping and planning using Octomaps and OMPL take a look to our work http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7139336&tag=1, https://www.youtube.com/watch?v=A5-8LTBxbHQ

basically defines the ROS publishers needed to visualize the resulting path and load the parameters from the configuration file (e.g. tests_canyon_croatia16.yaml).

Apart from the constructor and the methods that visualize the path and create the mission file, there is another important method planWithSimpleSetup(), which attempts to solve the start-to-goal query using a SimpleSetup (a class that defines and solves a path-planning problem). planWithSimpleSetup() defines the C-Space (RxR), creates an instance of the desired planner (RRT, RRT*, EST, PRM, etc., specified in the configuration file) and then creates an instance of the state validity checker. This latter is a user-defined set of routines (state_validity_checker_octomap_fcl_R2.cpp/hpp) that verifies whether a configuration is valid or not, which in the simplest case, as the one used in our example, means that the state (configuration or position) is collision-free. After creating the state validity checker, the start and goal positions (loaded from the configuration file in the constructor) are provided to the SimpleSetup. The final step is to attempt to solve the query for a specified time (also given in the configuration file).

3.2. Subscribing to the vehicle's odometry

There are some parts of the code that are commented and incomplete. The first of them is a subscriber to the vehicle's odometry. Subscribe to /cola2_navigation/nav_sts and store the vehicle x and y positions in order to continuously have updated information of vehicle's position.

3.3. Creating a service to solve a query

The provided code attempts to solve the query once the instance of the planner has been created, as observed in the main procedure offline_planner.planWithSimpleSetup(). Create (complete) the service /planning/solve and its respective callback in order to solve the query only when such service is called.

3.4. Creating a service to solve a query from current position

Now that a subscriber is obtaining updated information of vehicle's position, create (complete) the service /planning/solve_from_curr_pos and its respective callback in order to solve the query using the current position as the start position. The goal will remain as the one specified in the configuration. (Note: in order to modified the query start position provide start_state_[0] and start_state_[1] with the appropriate values).

3.5. Creating a service to solve a query from current position to a given goal

Now create (complete) the service /planning/solve_from_curr_pos_to_giv_goal and its respective callback in order to solve the query using the current position as the start position and a given goal. For this it is necessary to create the message SetGoal.srv that permits a service to receive the goal specified as a x and y values. (Note: in order to modify the query start and goal positions provide start_state_[0], start_state_[1], goal_state_[0], goal_state_[1] with the appropriate values. Also do not forget to modify the CMakeLists to compile the message and include the header in the source file).

3.6. Creating a service to solve a query from a given start to a given goal

Now create (complete) the service /planning/solve_from_giv_start_to_giv_goal and its respective callback in order to solve the query using given start and goal positions. For this it is necessary to create the message SetStartGoal.srv that permits a service to receive the

start and goal specified as a start_x, start_y, goal_x and goal_y values. (Note: in order to modify the query start and goal positions provide start_state_[0], start_state_[1], goal_state_[0], goal_state_[1] with the appropriate values. Also do not forget to modify the CMakeLists to compile the message and include the header in the source file).