KTH ROYAL INSTITUTE
OF TECHNOLOGY

# Introduction to ROS

# What is ROS?

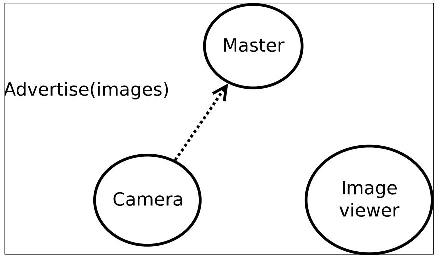
- "Middleware" designed to abstract away interprocess communication:
  - A ROS process is called a "node"
  - Nodes communicate by trading "messages" or calling "services". ROS manages the lower level side of things
  - Lots of convenient abstractions for Robotics
  - Provides several tools which help with debugging, logging, visualization, etc.

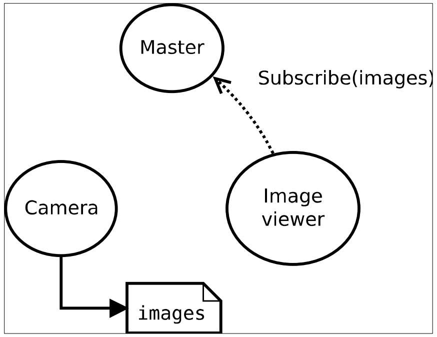
# What is ROS?

# Abstractions in ROS

# Abstractions in ROS

- ROS “master”: special process which maintains the ROS computation graph
- Acts as a DNS server: makes sure that Nodes can see each other



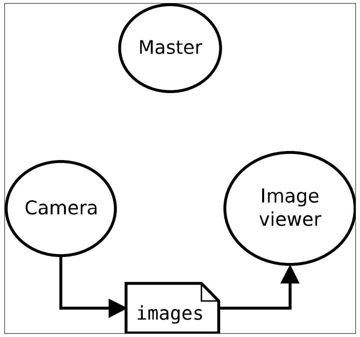




Image source: http://wiki.ros.org/Master?action=AttachFile&do=get&target=ROS_master_example_english_1.png
http://wiki.ros.org/Master?action=AttachFile&do=get&target=ROS_master_example_english_2.png
http://wiki.ros.org/Master?action=AttachFile&do=get&target=ROS_master_example_english_3.png

# Abstractions in ROS

- Basic interprocess communication: “nodes” exchange “messages” or call “services”
- “Messages” are “published” in “topics”
- To send messages, a node “publishes” in a topic
- To receive messages, a node “subscribes” to a topic

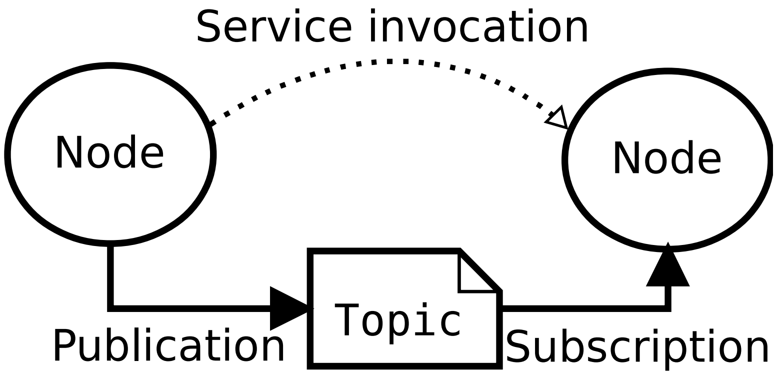



Image source: http://ros.org/images/wiki/ROS_basic_concepts.png

# Abstractions in ROS

- “Services” are server-client types of communication
- A client node “calls” a server node by sending a “Service message”
- The server node processes the message and sends a reply

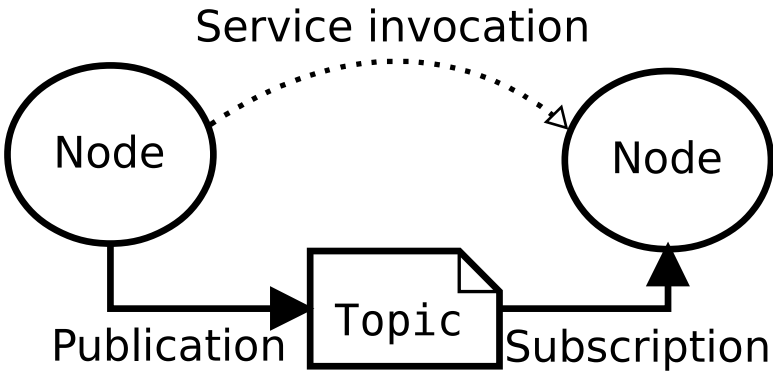



Image source: http://ros.org/images/wiki/ROS_basic_concepts.png

# Abstractions in ROS

- ROS software is organized in “packages”
- Packages contain code for nodes, libraries and/or messages (services definitions)
- Goal of a package is to offer specific functionality to a system
- Packages are built with *catkin* (more about this later)

```
➜ ~/Documents/example_ws/src/example_package
.
├── CMakeLists.txt
├── include
│   └── example_package
│       ├── library.hpp
│       └── node.hpp
├── package.xml
└── src
    ├── library.cpp
    └── node.cpp
```

# Example: write a ROS node

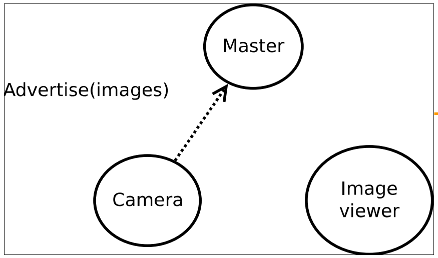
# Example: ROS node

Declare a new node

Handles communication
With the master

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example_node");
  ROS_INFO_STREAM(ros::this_node::getName() + " is alive");

  ros::NodeHandle nh;
  ros::Publisher pub;

  // advertise a new topic
  pub = nh.advertise<std_msgs::String>("example_topic", 1);
  std_msgs::String example_msg;

  ros::Rate loop_rate(2);
  int iter = 0;
  while (ros::ok()) // returns false when the node is killed
  {
    example_msg.data = "Hello world: " + std::to_string(iter++);
    pub.publish(example_msg);
    ROS_INFO_STREAM_THROTTLE(30, ros::this_node::getName() + " is still alive");
    loop_rate.sleep();
  }

  return 0;
}
```

# Example: ROS node




```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example_node");
  ROS_INFO_STREAM(ros::this_node::getName() + " is alive");

  ros::NodeHandle nh;
  ros::Publisher pub;

  // advertise a new topic
  pub = nh.advertise<std_msgs::String>("example_topic", 1);
  std_msgs::String example_msg;

  ros::Rate loop_rate(2);
  int iter = 0;
  while (ros::ok()) // returns false when the node is killed
  {
    example_msg.data = "Hello world: " + std::to_string(iter++);
    pub.publish(example_msg);
    ROS_INFO_STREAM_THROTTLE(30, ros::this_node::getName() + " is still alive");
    loop_rate.sleep();
  }

  return 0;
}
```

# Example: ROS node

ROS has a nice logging system. Allows you to easily filter messages based on severity.

```cpp
#include <ros/ros.h>
#include <std_msgs/String.h>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "example_node");
  ROS_INFO_STREAM(ros::this_node::getName() + " is alive");

  ros::NodeHandle nh;
  ros::Publisher pub;

  // advertise a new topic
  pub = nh.advertise<std_msgs::String>("example_topic", 1);
  std_msgs::String example_msg;

  ros::Rate loop_rate(2);
  int iter = 0;
  while (ros::ok()) // returns false when the node is killed
  {
    example_msg.data = "Hello world: " + std::to_string(iter++);
    pub.publish(example_msg);
    ROS_INFO_STREAM_THROTTLE(30, ros::this_node::getName() + " is still alive");
    loop_rate.sleep();
  }

  return 0;
}
```

# Example: ROS node

- To compile packages, ROS uses catkin
- Builds on top of cmake to recursively compile and link ROS packages
- More examples and really good tutorials (cpp and python): http://wiki.ros.org/ROS/Tutorials

# Useful ROS tools

# Useful ROS tools

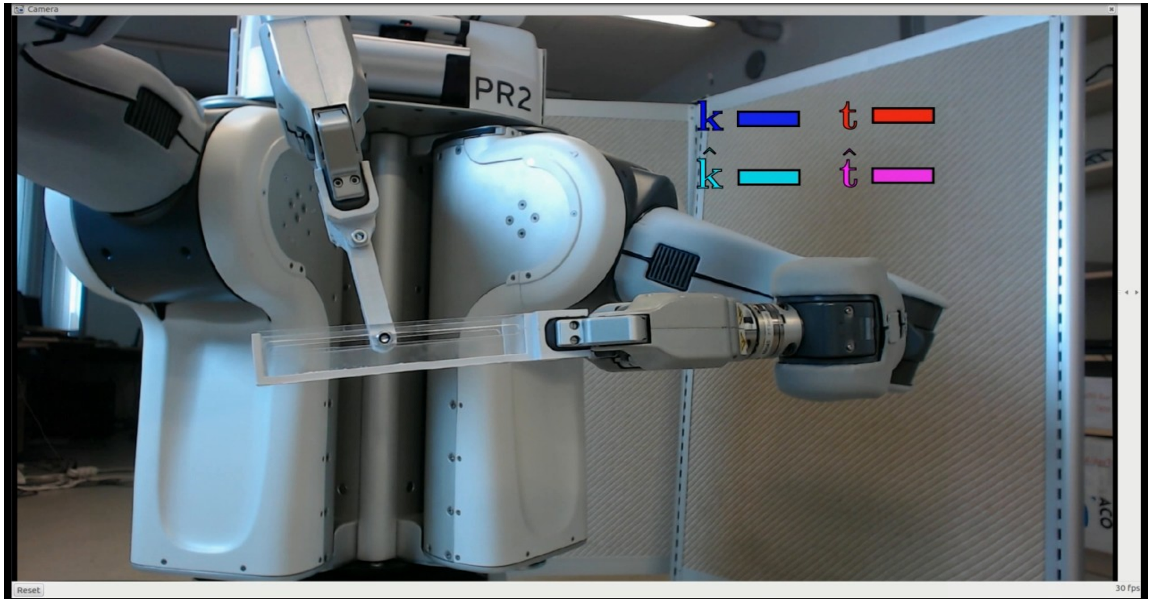
- Basic command line tools:
  - rostopic: allows to interact with topics without writing a node for it. Example: “$ rostopic echo topic_name” will print the messages published in ‘topic_name’ to the terminal
  - rosservice: sends a service request from the command line
  - roscd: changes your current directory to the requested package. Example: “$ roscd package_name”

# Useful ROS tools

- Launch files: **Very important** files which allows you to run a set of ROS nodes
- Rosbags: The ROS way to log data
  - You can save topics into a rosbag file
  - ROS provides code to ‘play’ a bag file. Allows you to use data recorded later as if it were being generated in the moment
- **Demo with publisher node**
- Parameter server: part of the ROS master node, it keeps parameters’ values which can be accessed from any ROS node. Very useful for configuring nodes

# Useful ROS tools

- Rviz: Very useful tool which displays graphical information generated by ROS nodes



# Useful ROS tools

- TF: Library which manages coordinate transforms in a robotic system

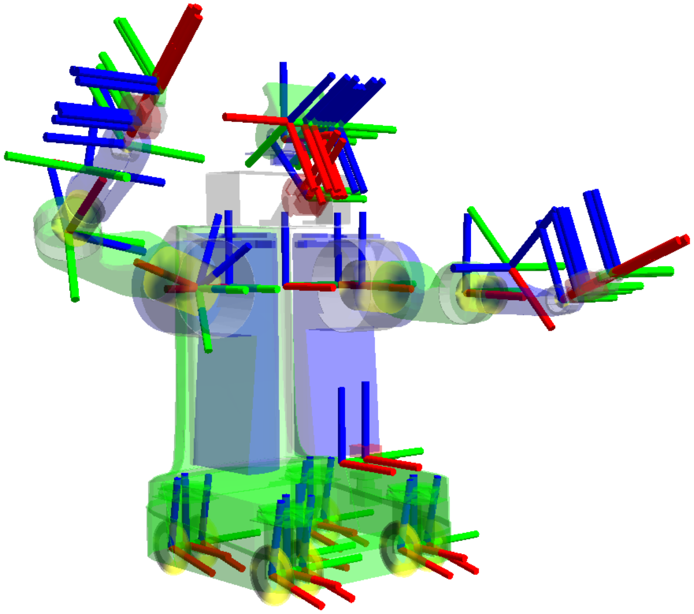


Image source: http://wiki.ros.org/tf?action=AttachFile&do=get&target=frames2.png

# Useful ROS tools

- Actionlib: Implements a protocol for interfacing with actions. Similar to services, but allows multiple goals and preemption.

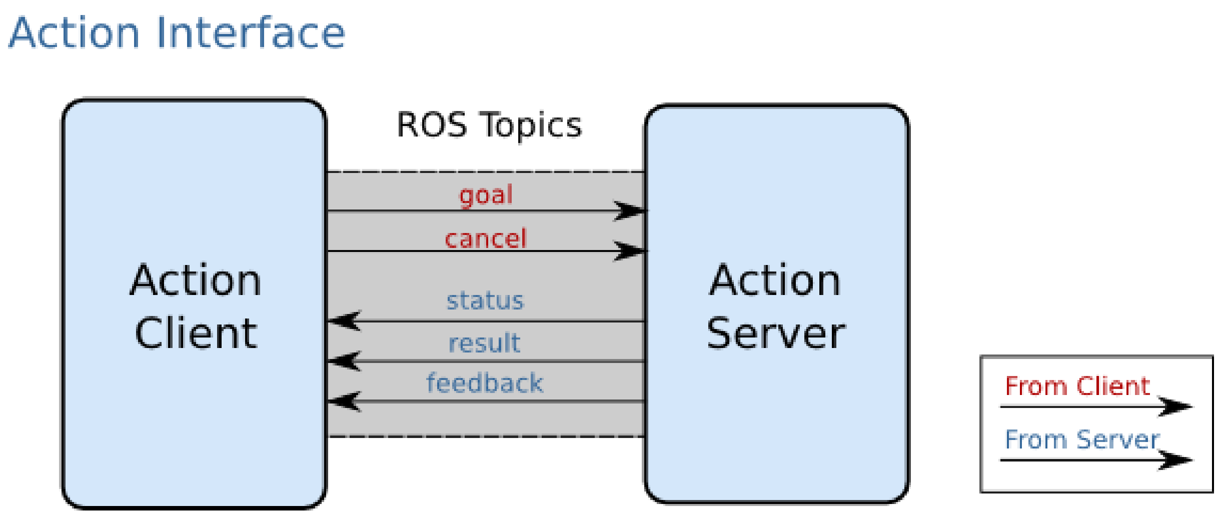



Image source: http://wiki.ros.org/actionlib/DetailedDescription?action=AttachFile&do=get&target=action_interface.png

# ROS Programming exercise

# Setting up your system

- Pre-requisite: Ubuntu 16.04
- Install script:

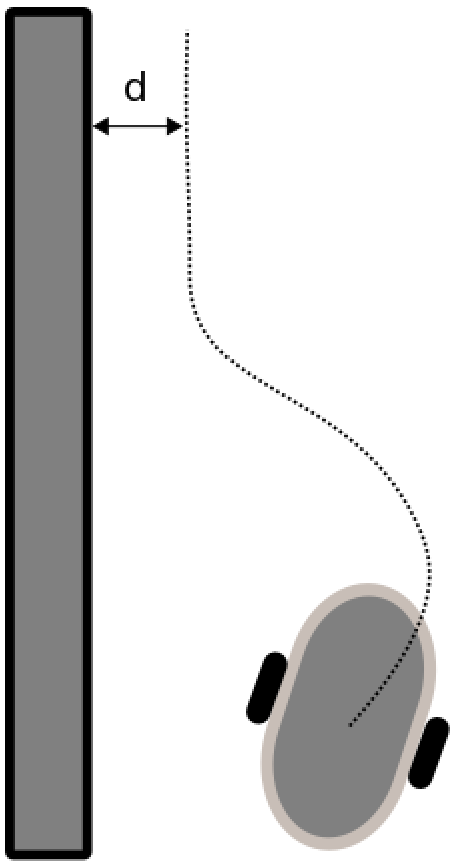
  $ wget https://raw.githubusercontent.com/KTH-RAS/ras_install/kinetic-2018/scripts/install_basic.sh

- Run the script. You should be able to launch the exercise environment through

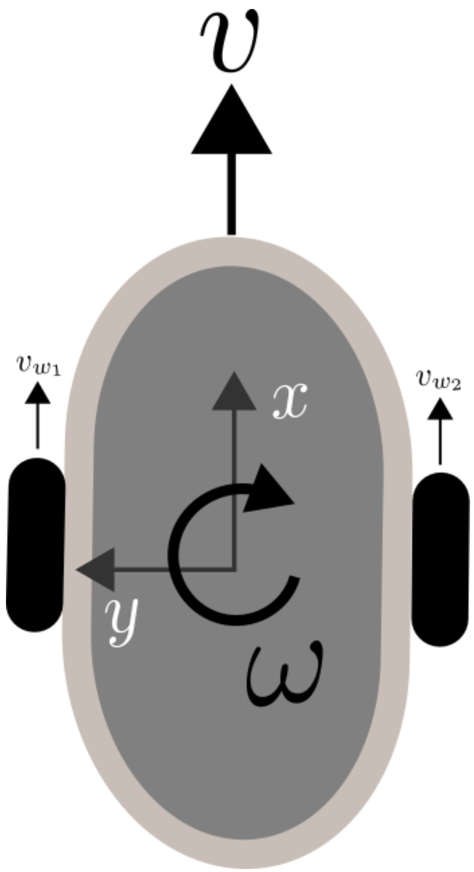
  $ roslaunch ras_lab1_launch kobuki_lab1.launch

# Programming exercise

- Problem: Drive a differential drive robot to follow a virtual wall
- Robot is equipped with encoders and distance sensors
- Need to control robot base speed to keep wall at distance d



# Robot kinematics



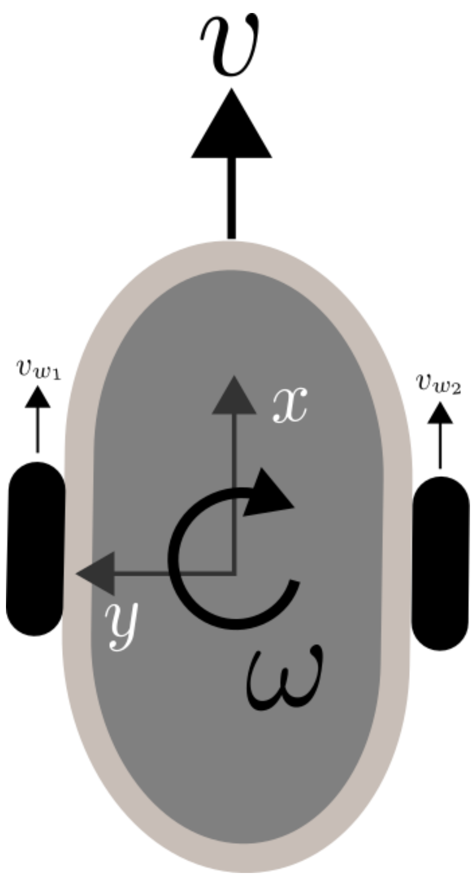

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

$$v = \frac{v_{w_1} + v_{w_2}}{2}$$

$$\omega = \frac{v_{w_2} - v_{w_1}}{2b}$$

$$v_{w_i} = \frac{2\pi r f \Delta_{\text{enc}}}{\text{ticks per rev}}$$

# Robot properties




$$\text{ticks per rev} = 360$$
$$b = 0.115$$
$$r = 0.0352$$

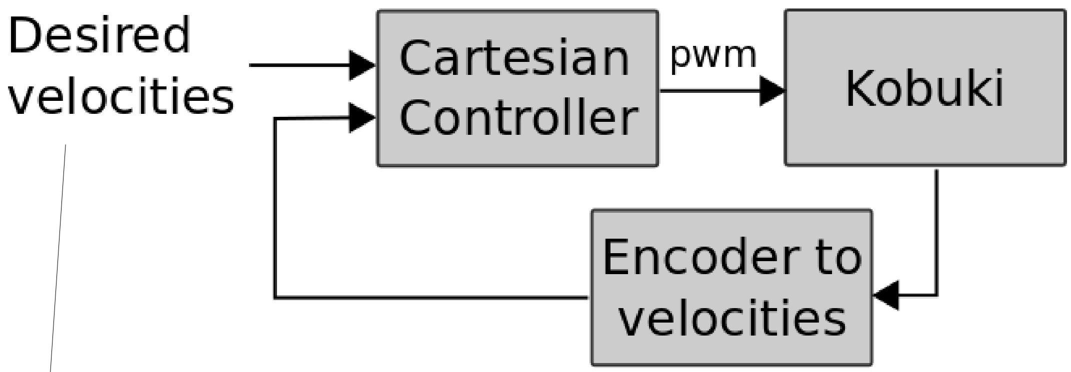
# Task 1: Send a command to the robot

- The robot receives a pwm signal on the topic /kobuki/pwm
- It sends encoder information over /kobuki/encoders
- Sensor information in /kobuki/adc
- Demo

```
➜  ~  rostopic list | grep kobuki
/kobuki/adc
/kobuki/encoders
/kobuki/pwm
```

# Task 2: Cartesian controller

- Given a desired robot velocity, control the wheels




$$\begin{aligned} v &= \frac{v_{w_1} + v_{w_2}}{2} \\ \omega &= \frac{v_{w_2} - v_{w_1}}{2b} \end{aligned}$$

# Task 2: Cartesian controller

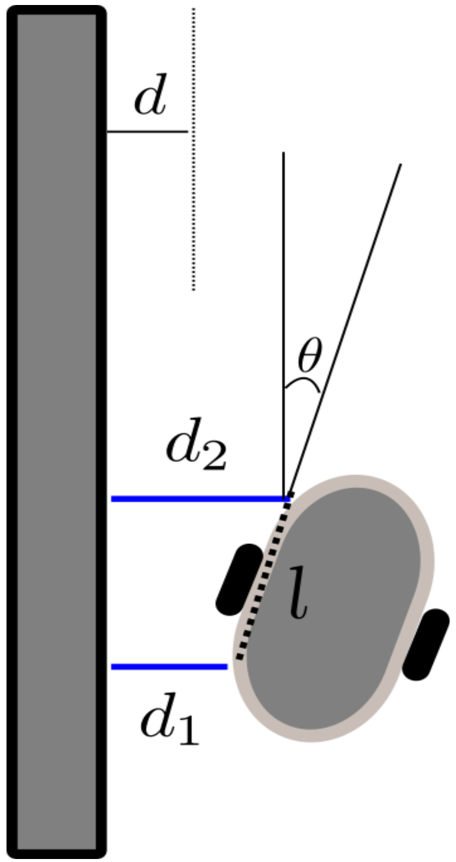
- The cartesian controller must subscribe to the topic */motor_controller/twist* of the type *geometry_msgs/Twist* (user input)
- It translates the desired linear and angular velocities into desired wheel velocities
- Closes a feedback loop between the commanded pwm signal and the encoder feedback
- Demo

# Task 3: Wall follower

- Close the wall following loop by generating a twist command based on the distance sensors' feedback

$$\theta = \arctan\left(\frac{d_1 - d_2}{\sqrt{(d_1 - d_2)^2 + l^2}}\right)$$

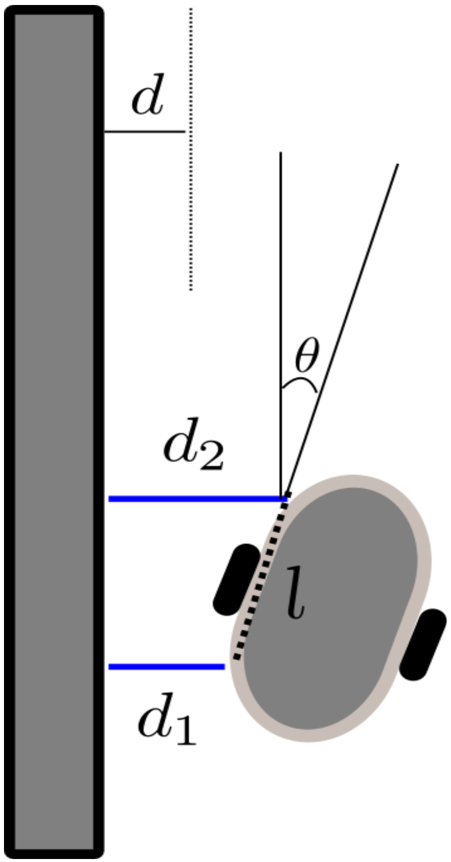
$$l = 0.2$$



# Task 3: Wall follower

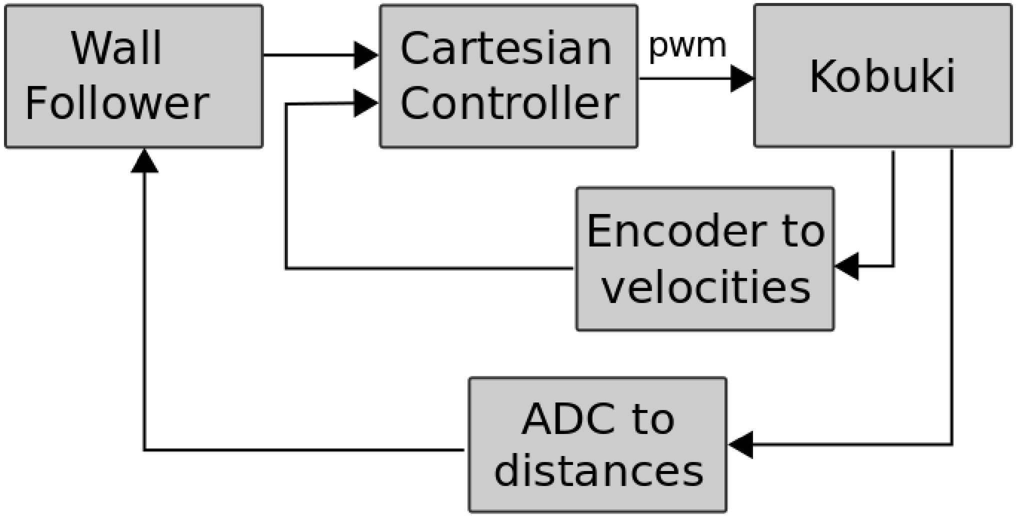
- Goal: Follow the wall at distance d
- Sensor readings are available in the */kobuki/adc* topic. Need to be converted to distance

$$d_i = 1.114e^{-0.004\mathrm{adc}}$$



# Task 3: Wall follower

- Cartesian velocity is published to the cartesian controller
- How to set the desired angle to the wall?
- Demo



# Tips

- The wall can be reset with respect to the robot by calling the service */reset_world*
- A PI controller works well for the pwm commands:

$$\mathrm{pwm}_i(t) = K_p e(t) + K_i \int_{t_i}^{t} e(\tau)\,\mathrm{d}\tau$$

- A P controller is enough for the wall follower errors
- Try changing the desired angle to the wall based on the distance between the robot and the wall