Introduction to Player/Stage

Dario Cazzaro, Luca Invernizzi

Motion Lab, UCSB

October 9, 2008

Outline

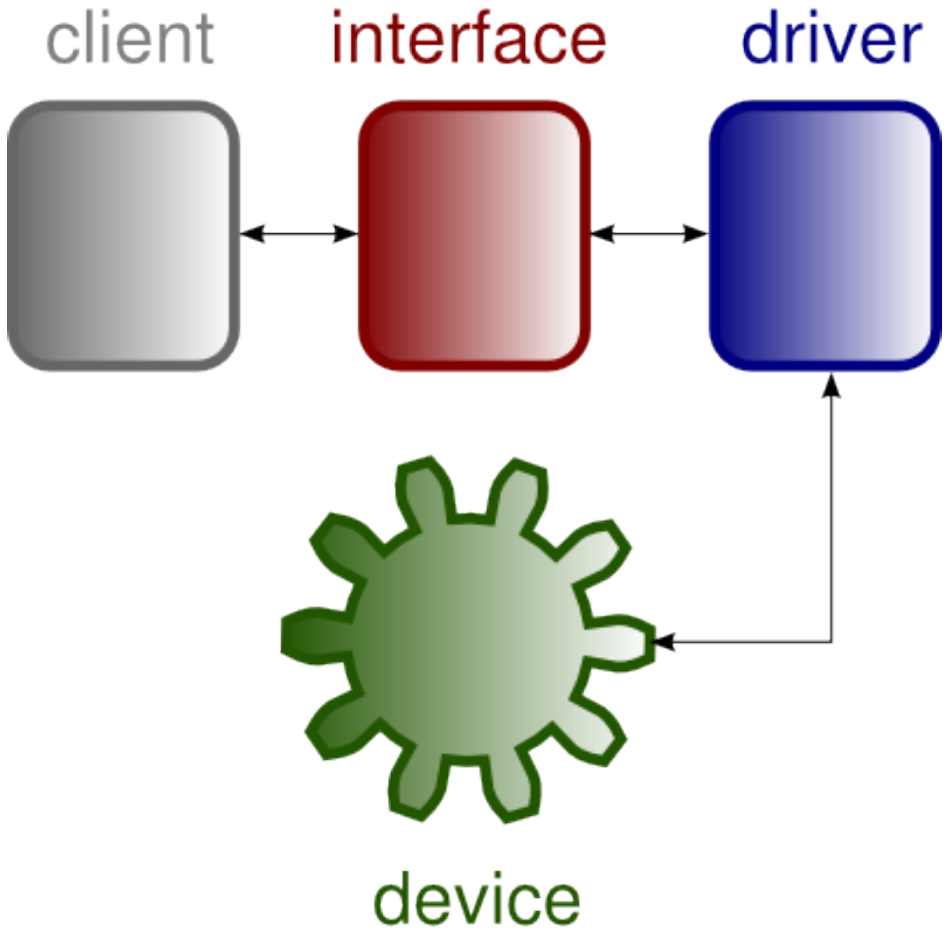
Interfaces and Drivers

In Player, there are 2 kind of entities you need to control an external device:

Interfaces: which describe a standard way to exchange data with the device

Drivers: which take care of the low level communication with the hardware

User's clients only need to connect to the Player interface, without any knowledge of the underlying hardware



Example: to connect to a laser scanner, you have to:

- define the binding between the *laser* interface and the driver (eg. the *sicklms200*) in the Player configuration file
- connect to the *laser* interface and get range data from it

In this way you can use your client on different robots with different devices, changing only the bindings between the standard interfaces and the actual drivers

Outline

Configuration files

A Player configuration file is composed of one or more *driver* sections, like this one:

robot.cfg

```
driver(
    name "sicklms200"
    provides ["laser:0"]
)
```

Configuration files

Here are some useful options to put in a *driver* section

name	The name of an existing driver
provides	This option lists the interfaces provided by this driver
requires	This option lists the interfaces that this driver needs
ad-hoc	Other driver-dependent options

Interfaces addresses are a 5 value string:

`key:host:robot:interface:index`

Where only `interface` and `index` are mandatory.

- **key** is needed to map interfaces of the same kind into different devices
- **host** is the network name of the host providing the interface
- **robot** is the port of the host providing the interface
- **interface** is the name
- **index** is needed to choose between interfaces of the same kind

Configuration files

Some drivers provide interfaces of the same kind, so you can use the *key* value to map them into different devices

robot.cfg

```
driver
(
  name "p2os"
  provides ["odometry::position2d:0"
            "compass::position2d:1"
            "gyro::position2d:2"]
)
```

Configuration files

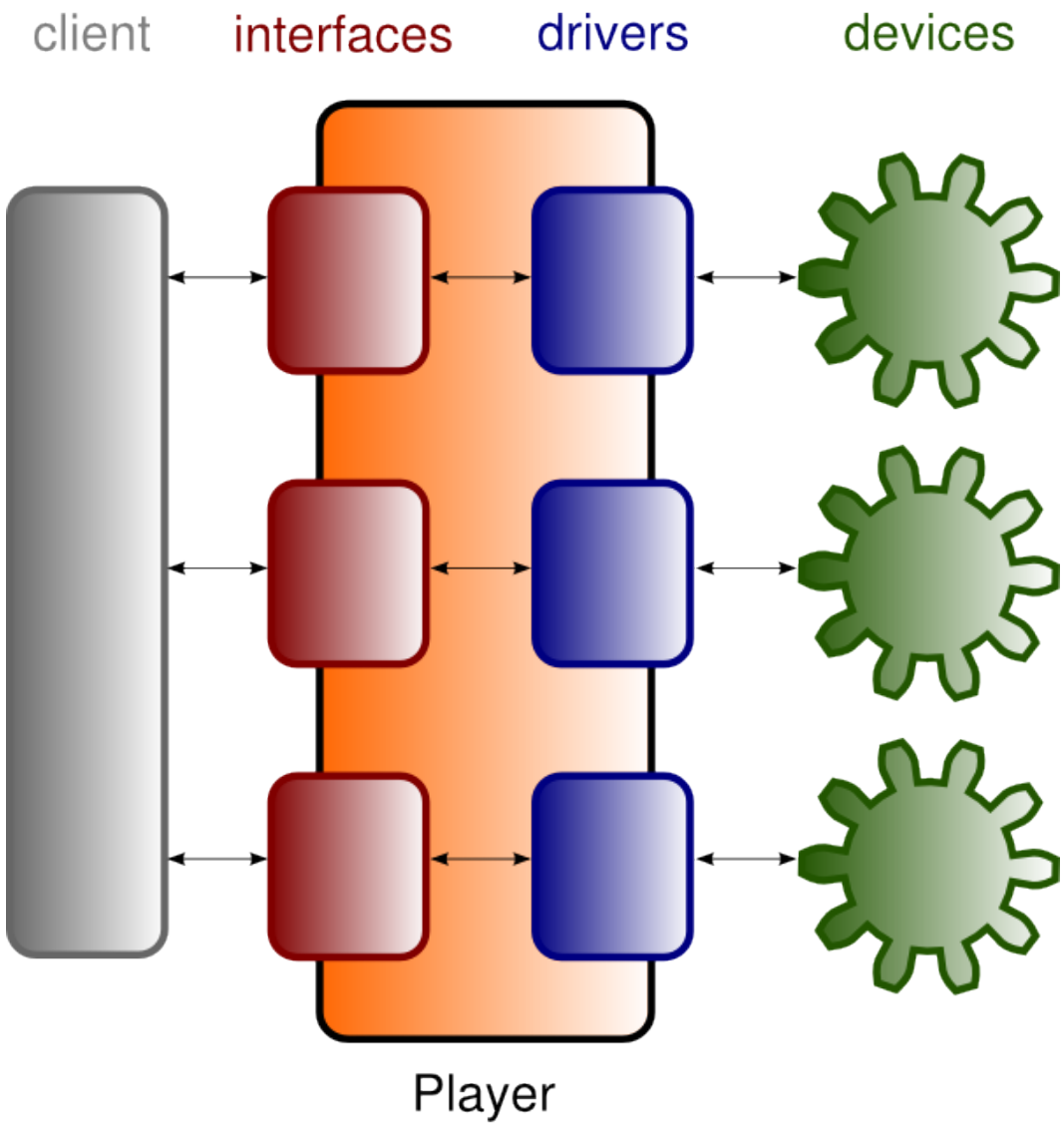
Some algorithms are implemented as drivers, so they can be used like devices through standard interfaces.

robot.cfg

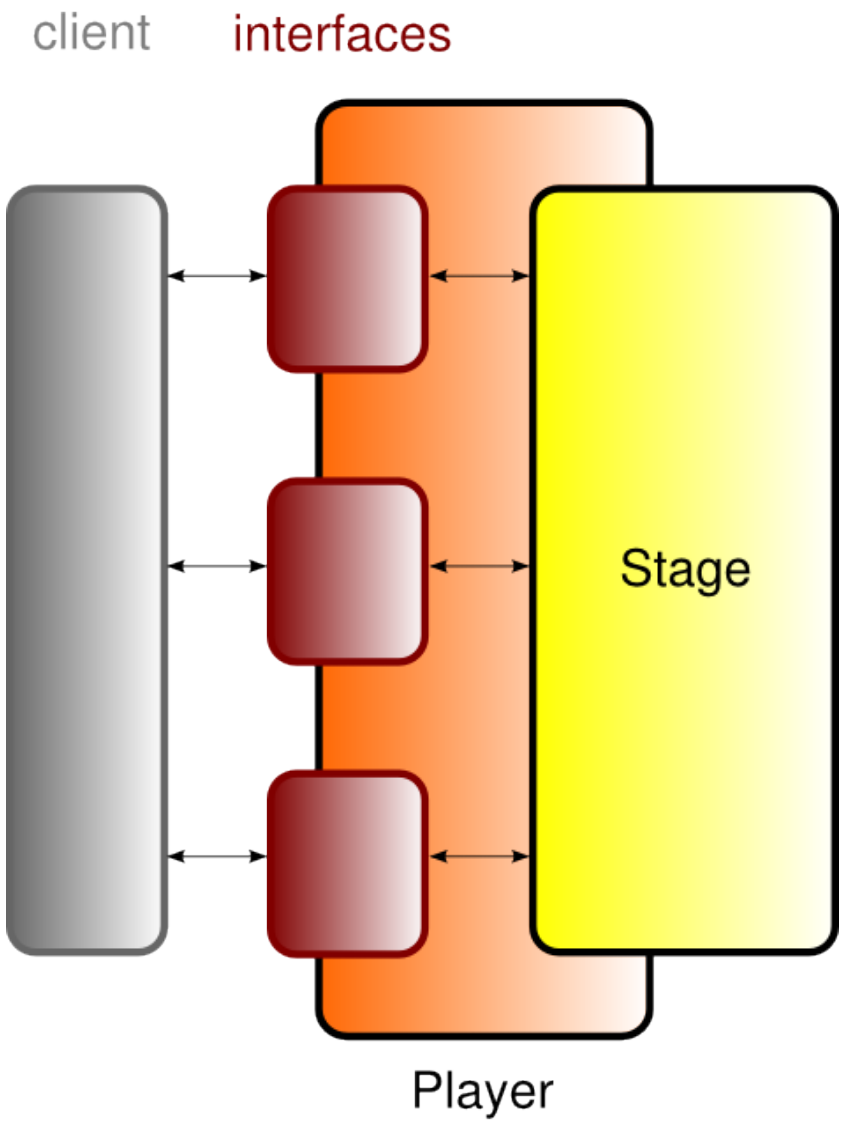
```
driver
(
  name "vfh"
  provides ["position2d:1"]
  requires ["position2d:0" "laser:0"]
)
```

Outline

The Player architecture



Working with Stage



In order to create a simulated world, you have to load the *stage* plugin, which is configured in the *worldfile*

robot.cfg

```
driver
(
  name "stage"
  provides ["simulation:0"]
  plugin "libstageplugin"
  worldfile "square.world"
)
```

World files

A *worldfile* contains a description of every object which take part in the simulated world

square.world

```
world
(
   name              "[filename of worldfile]"
   interval_real     100
   interval_sim      100
   gui_interval      100
   resolution        0.01
)
```

Populate the world

Before adding a robot to the world, you have to describe its model

square.world

```
include "circlebot.inc"
circlebot
(
  name "myrobot"
  color "red"
  pose [-0.5 0.5 0]
)
```

Once you have defined your model in the *worldfile*, you can define a *stage* driver and attach it to all the interfaces you need. Back in the Player configuration file:

robot.cfg

```
driver
(
  name "stage"
  provides ["position2d:0" "laser:0"]
  model "myrobot"
)
```

Outline

User's clients

After setting up the configuration files, you have to start the Player server

```
user@host$ player robot.cfg
```

and then you will be able to connect your client to Player

User's clients

Example: C++ client

client.cpp

```
#include <iostream>
#include <libplayerc++/playerc++.h>

int main(int argc, char *argv[]){
  using namespace PlayerCc;

  PlayerClient    robot("localhost");
  SonarProxy      sp(&robot,0);
  Position2dProxy pp(&robot,0);
```

User's clients

client.cpp

```
for(;;){
    double turnrate = dtor(10), speed = 0.1;

    // read from the proxies
    robot.Read();

    // print out sonars for fun
    std::cout << sp << std::endl;

    // command the motors
    pp.SetSpeed(speed, turnrate);
  }
}
```

Outline

Drivers

Do you remember what's a *driver*?

Drivers

Do you remember what's a *driver*?
It's an abstraction layer, that provides a standard interface of high-level commands.

Outline

Moving the robot

Suppose you want to move your robot.

Moving the robot

Suppose you want to move your robot.
You should consider one of the three positioning interfaces:

- position1d: moving on a line
- position2d: moving on a plane
- position3d: moving in a space

Moving the robot

Suppose you want to move your robot.
You should consider one of the three positioning interfaces:

- position1d: moving on a line
- position2d: moving on a plane
- position3d: moving in a space

Either one of these gives the robot the wanted new "ability" of moving

A practical example

In fact, if you connect with one of the robots in the lab (in this example, POD), you'll find that they give you this output.

A practical example

In fact, if you connect with one of the robots in the lab (in this example, POD), you'll find that they give you this output.

Command line

```
user@host$  playerv -h pod -p 6665
```

A practical example

In fact, if you connect with one of the robots in the lab (in this example, POD), you'll find that they give you this output.

Command line

```
user@host$  playerv -h pod -p 6665
PlayerViewer 2.0.5
Connecting to [pod:6665]
calling connect
done

Available devices: pod:6665
position2d:0      stage          ready
laser:0           stage          ready
```

A practical example

Command line

```
user@host$  playerv -h pod -p 6665
PlayerViewer 2.0.5
Connecting to [pod:6665]
calling connect
done

Available devices: pod:6665
position2d:0      stage          ready
laser:0           stage          ready
```

Note that :0

A practical example

Command line

```
user@host$  playerv -h pod -p 6665
PlayerViewer 2.0.5
Connecting to [pod:6665]
calling connect
done

Available devices: pod:6665
position2d:0      stage          ready
laser:0           stage          ready
```

Note that :0 : a robot can offer more than one driver of the same kind (e.g. a collision avoidance driver would provide the same position2d interface, since it's a smarter positioning system). That number is the way to select the wanted driver

A practical example (part II)

To easily drive the robot around, you have to:

- launch the PLAYER Server with a proper configuration file (shown later)

Command line

```
user@host$  player      positioning_example_1.cfg
```

A practical example (part II)

To easily drive the robot around, you have to:

- launch the PLAYER Server with a proper configuration file (shown later)
- launch the PLAYER Viewer, which is an easy utility to peek at the robot sensors and give commands to it

Command line

```
user@host$  player      positioning_example_1.cfg
user@host$  playerv
```

A practical example (part II)

To easily drive the robot around, you have to:

- launch the PLAYER Server with a proper configuration file (shown later)
- launch the PLAYER Viewer, which is an easy utility to peek at the robot sensors and give commands to it

Command line

```
user@host$  player -d 9 positioning_example_1.cfg
user@host$  playerv -h localhost -p 6665
```

These parameters are optional

A practical example (part II)

To easily drive the robot around, you have to:

- launch the PLAYER Server with a proper configuration file (shown later)
- launch the PLAYER Viewer, which is an easy utility to peek at the robot sensors and give commands to it

Command line

```
user@host$  player -d 9 positioning_example_1.cfg
user@host$  playerv -h localhost -p 6665
```

These parameters are optional, to make player more verbose

A practical example (part II)

To easily drive the robot around, you have to:

- launch the PLAYER Server with a proper configuration file (shown later)
- launch the PLAYER Viewer, which is an easy utility to peek at the robot sensors and give commands to it

Command line

```
user@host$  player -d 9 positioning_example_1.cfg
user@host$  playerv -h localhost -p 6665
```

These parameters are optional, to make player more verbose, or use a remote robot

A practical example (part II)

To easily drive the robot around, you have to:

- launch the PLAYER Server with a proper configuration file (shown later)
- launch the PLAYER Viewer, which is an easy utility to peek at the robot sensors and give commands to it

Command line

```
user@host$  player -d 9 positioning_example_1.cfg
user@host$  playerv -h localhost -p 6665
```

Let's see a video

A practical example (part III)

Let's look at the configuration file:

- it tells PLAYER to load the Stage driver (since we are simulating the robot)

positioning_example_1.cfg

```
driver
(
   name "stage"
   provides ["simulation:0" ]
   plugin "libstageplugin"
   worldfile "positioning_example_1.world"
)
```

A practical example (part III)

Let's look at the configuration file:

- it tells PLAYER to load the Stage driver (since we are simulating the robot), and where to find the environment and robot configuration

positioning_example_1.cfg

```
driver
(
   name "stage"
   provides ["simulation:0" ]
   plugin "libstageplugin"
   worldfile "positioning_example_1.world"
)
```

A practical example (part III)

Let's look at the configuration file:

- it tells PLAYER to load the Stage driver (since we are simulating the robot), and where to find the environment and robot configuration
- it tells PLAYER to simulate robot0 (which configuration is in the same configuration file).

positioning_example_1.cfg

```
driver
(
   name "stage"
   provides ["position2d:0"]
   provides ["laser:0"]
   model "robot0"
)
```

A practical example (part III)

Let's look at the configuration file:

- it tells PLAYER to load the Stage driver (since we are simulating the robot), and where to find the environment and robot configuration
- it tells PLAYER to simulate robot0 (which configuration is in the same configuration file).

positioning_example_1.cfg

```
driver
(
   name "stage"
   provides ["simulation:0" ]
   plugin "libstageplugin"
   worldfile "positioning_example_1.world"
)
```

A practical example (part III)

Let's look at the configuration file:

- it tells PLAYER to load the Stage driver (since we are simulating the robot), and where to find the environment and robot configuration
- it tells PLAYER to simulate robot0 (which configuration is in the same configuration file). In particular, it should provide its positioning and laser capabilities

positioning_example_1.cfg

```
driver
(
   name "stage"
   provides ["position2d:0"]
   provides ["laser:0"]
   model "robot0"
)
```

A practical example (part III)

Let's look at the configuration file:

- it tells PLAYER to load the Stage driver (since we are simulating the robot), and where to find the environment and robot configuration
- it tells PLAYER to simulate robot0 (which configuration is in the same configuration file). In particular, it should provide its positioning and laser capabilities

positioning_example_1.cfg

```
driver
(
   name "stage"
   provides ["position2d:0"]
   provides ["laser:0"]
   model "robot0"
)
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- It imports a robot type configuration

positioning_example_1.world

```
include "circlebot.inc"
include "urglaser.inc"
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- It imports a robot type configuration

positioning_example_1.world

```
include "circlebot.inc"
include "urglaser.inc"
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- It imports a robot type configuration
- It imports a laser type configuration

positioning_example_1.world

```
include "circlebot.inc"
include "urglaser.inc"
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- It imports a robot type configuration
- It imports a laser type configuration
- It loads an environment bitmap map

positioning_example_1.world

```
map
(
  bitmap "./bitmaps/lab-gmapping.png"
  size [6 6]
  name "map"
)
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- ...
- It create a *circlebot* robot

positioning_example_1.world

```
circlebot
(
  name "robot0"
  color "red"
  localization "gps"
  localization_origin [0 0 0]
  pose [2 2 0]
  urg_laser( laser_sample_skip 1 )
)
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- ...
- It create a *circlebot* robot ,which is going to be red in Stage

positioning_example_1.world

```
circlebot
(
  name "robot0"
  color "red"
  localization "gps"
  localization_origin [0 0 0]
  pose [2 2 0]
  urg_laser( laser_sample_skip 1 )
)
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- ...
- It create a *circlebot* robot ,which is going to be red in Stage, with a good localization system

positioning_example_1.world

```
circlebot
(
  name "robot0"
  color "red"
  localization "gps"
  localization_origin [0 0 0]
  pose [2 2 0]
  urg_laser( laser_sample_skip 1 )
)
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- ...
- It create a *circlebot* robot ,which is going to be red in Stage, with a good localization system.The robot will be centered in (2,2)

positioning_example_1.world

```
circlebot
(
  name "robot0"
  color "red"
  localization "gps"
  localization_origin [0 0 0]
  pose [2 2 0]
  urg_laser( laser_sample_skip 1 )
)
```

A practical example (part III)

Let's look at the world file, that defines the environment and the robots configuration:

- ...
- It create a *circlebot* robot ,which is going to be red in Stage, with a good localization system.The robot will be centered in (2,2) and it will have a laser

positioning_example_1.world

```
circlebot
(
  name "robot0"
  color "red"
  localization "gps"
  localization_origin [0 0 0]
  pose [2 2 0]
  urg_laser( laser_sample_skip 1 )
)
```

A practical example (part IV)

Now that we have a running simulation of a simple robot, how can we can connect to it in our program?

A practical example (part IV)

Now that we have a running simulation of a simple robot, how can we can connect to it in our program?
Suppose we want to write our program in C++: we will use the PLAYERC++ library

A practical example (part IV)

Now that we have a running simulation of a simple robot, how can we can connect to it in our program?

Suppose we want to write our program in C++: we will use the PLAYERC++ library

There are also libraries for C, PYTHON, ADA...

A practical example (part IV)

positioning_example_1.cc

```
#include <iostream>
#include <libplayerc++/playerc++.h>
#include <stdlib.h>
using namespace PlayerCc;
using namespace std;
```

A practical example (part IV)

positioning_example_1.cc

```
#include <iostream>
#include <libplayerc++/playerc++.h>
#include <stdlib.h>
using namespace PlayerCc;
using namespace std;
```

The namespace PlayerCc contains all PLAYER classes. It does not, however contain some PLAYER structures.

A practical example (part IV)

positioning_example_1.cc

```
#include <iostream>
#include <libplayerc++/playerc++.h>
#include <stdlib.h>
using namespace PlayerCc;
using namespace std;

  int main(int argc, char *argv[]) {
      PlayerClient    robot("localhost",6665);
      Position2dProxy posp(&robot,0);
```

The PlayerClient class is the interface between our program and the player server. It's able to connect to it and read the data it sends us (the constructor accepts the parameters *hostname* and *port* of the PLAYER server).

A practical example (part IV)

positioning_example_1.cc

```
int main(int argc, char *argv[]) {
    PlayerClient    robot("localhost",6665);
    Position2dProxy posp(&robot,0);
```

To move the robot, we have to get access to the Position2dProxy class, that's a proxy to the positioning driver. It takes, as arguments, a handle to one initialized PlayerClient class, and the driver index (that *:0* we spoke about before).

A practical example (part IV)

positioning_example_1.cc

```
int main(int argc, char *argv[]) {
    PlayerClient    robot("localhost",6665);
    Position2dProxy posp(&robot,0);

    posp.RequestGeom();
    robot.Read();
```

Then we need to populate the Position2dProxy class with the necessary geometric informations on the robot (e.g. wheel radius...), so we request them.

A practical example (part IV)

positioning_example_1.cc

```
int main(int argc, char *argv[]) {
    PlayerClient    robot("localhost",6665);
    Position2dProxy posp(&robot,0);

    posp.RequestGeom();
    robot.Read();
```

Then we need to populate the Position2dProxy class with the necessary geometric informations on the robot (e.g. wheel radius...), so we request them.
To receive them, however, we need to read the data that the server has sent to our program.

A practical example (part IV)

positioning_example_1.cc

```
posp.RequestGeom();
robot.Read();

posp.SetMotorEnable(1);
posp.GoTo(0.5,1.0,0);
```

Now we can enable the motors (that's only necessary when working with real robots).

A practical example (part IV)

positioning_example_1.cc

```
posp.RequestGeom();
robot.Read();

posp.SetMotorEnable(1);
posp.GoTo(0.5,1.0,0);
```

Now we can enable the motors (that's only necessary when working with real robots).
Finally, we can order the robot to go to the point $(0.5, 1.0)$ with final heading equal 0 radians.

A practical example (part IV)

positioning_example_1.cc

```
        posp.SetMotorEnable(1);
        posp.GoTo(0.5,1.0,0);

        for(;;) {
            cout <<"position ("<<posp.GetXPos() \
                <<","<<posp.GetYPos()<<")"<<endl;
            robot.Read();
        }
    }
```

If we want to get informations about the robot position, we need to keep reading what the server sends us

A practical example (part IV)

Since the positioning system is fundamental, let's give out some final considerations on it:

- it is possible to tune the maximum error allowed when reaching a target in the .cfg file (e.g., if we don't care about the robot final heading, we can use the directive angle_epsilon 360)

A practical example (part IV)

Since the positioning system is fundamental, let's give out some final considerations on it:

- it is possible to tune the maximum error allowed when reaching a target in the .cfg file (e.g., if we don't care about the robot final heading, we can use the directive angle_epsilon 360)
- there are a bunch of possible way to control the robot position: the lowest level is controlling its heading and speed (e.g. SetSpeed(1,0))

A practical example (part IV)

Since the positioning system is fundamental, let's give out some final considerations on it:

- it is possible to tune the maximum error allowed when reaching a target in the .cfg file (e.g., if we don't care about the robot final heading, we can use the directive angle_epsilon 360)
- there are a bunch of possible way to control the robot position: the lowest level is controlling its heading and speed (e.g. SetSpeed(1,0))
- in the .cfg file we analyzed before, the robot knows its initial position (since it equips an ideal gps). However, this is not the case of the robots in the lab: those robots usually starts up believing to be in the origin of their own local reference system

A practical example (part IV)

Since the positioning system is fundamental, let's give out some final considerations on it:

- it is possible to tune the maximum error allowed when reaching a target in the .cfg file (e.g., if we don't care about the robot final heading, we can use the directive angle_epsilon 360)
- there are a bunch of possible way to control the robot position: the lowest level is controlling its heading and speed (e.g. SetSpeed(1,0))
- in the .cfg file we analyzed before, the robot knows its initial position (since it equips an ideal gps). However, this is not the case of the robots in the lab: those robots usually starts up believing to be in the origin of their own local reference system
- it is also possible to update the position with odometry, and set its error range (the error itself is stochastic).

Outline

Finding out our position

Now we know how to move, at least in a very basic fashion.

But, where is our robot?

Finding out our position

But, where is our robot?

- If we have some kind of global positioning system, fine. We'll probably need to do some sensor fusion with the odometry readings, in order to lower the positioning error. However, PLAYER does not know how to do that, so you'll have to find a third-party library (see `http://babel.isa.uma.es/mrpt/index.php/Main_Page`), or implement it yourself.

Finding out our position

But, where is our robot?

- If we have some kind of global positioning system, fine. We'll probably need to do some sensor fusion with the odometry readings, in order to lower the positioning error. However, PLAYER does not know how to do that, so you'll have to find a third-party library (see `http://babel.isa.uma.es/mrpt/index.php/Main_Page`), or implement it yourself.
- If we have the map, the odometry from the wheels and a laser/sonar, we'll find our position on the map by performing a pattern matching with the sensor readings: we'll need to find the position where those readings "makes sense"

Finding out our position

But, where is our robot?

- If we have some kind of global positioning system, fine. We'll probably need to do some sensor fusion with the odometry readings, in order to lower the positioning error. However, PLAYER does not know how to do that, so you'll have to find a third-party library (see `http://babel.isa.uma.es/mrpt/index.php/Main_Page`), or implement it yourself.
- If we have the map, the odometry from the wheels and a laser/sonar, we'll find our position on the map by performing a pattern matching with the sensor readings: we'll need to find the position where those readings "makes sense"
- If we have just the laser/sonar and the odometry, we have to perform a simultaneous localization and mapping (SLAM) algorithm, in order to build the map and perform the localization

Finding out our position

But, where is our robot?

- If we have some kind of global positioning system, fine. We'll probably need to do some sensor fusion with the odometry readings, in order to lower the positioning error. However, PLAYER does not know how to do that, so you'll have to find a third-party library (see `http://babel.isa.uma.es/mrpt/index.php/Main_Page`), or implement it yourself.
- If we have the map, the odometry from the wheels and a laser/sonar, we'll find our position on the map by performing a pattern matching with the sensor readings: we'll need to find the position where those readings "makes sense"
- If we have just the laser/sonar and the odometry, we have to perform a simultaneous localization and mapping (SLAM) algorithm, in order to build the map and perform the localization

Now we'll analyze this situation.

Finding out our position

The amcl driver implements the Adaptive Monte-Carlo Localization algorithm.

Finding out our position

The amcl driver implements the Adaptive Monte-Carlo Localization algorithm.

from the PLAYER documentation

At the conceptual level, the amcl driver maintains a probability distribution over the set of all possible robot poses, and updates this distribution using data from odometry, sonar and/or laser range-finders. The driver requires a pre-defined map of the environment against which to compare observed sensor values.

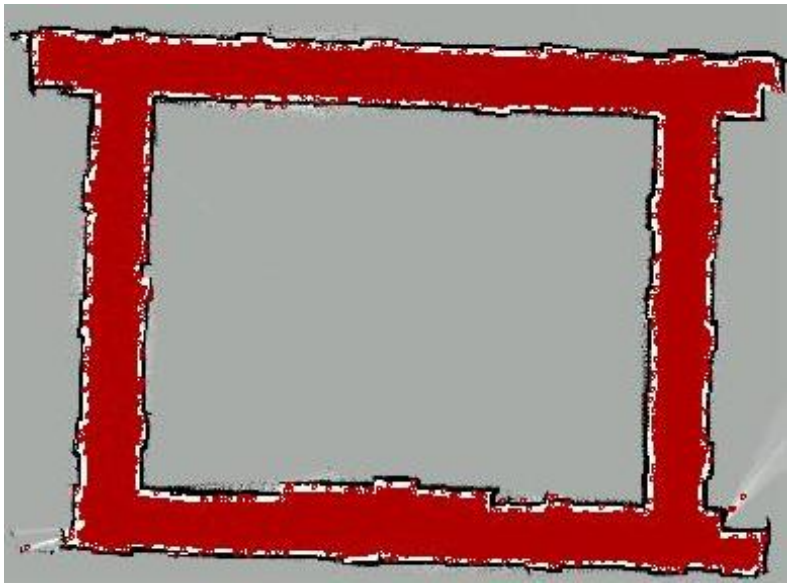
Finding out our position

The amcl driver implements the Adaptive Monte-Carlo Localization algorithm.

from the PLAYER documentation

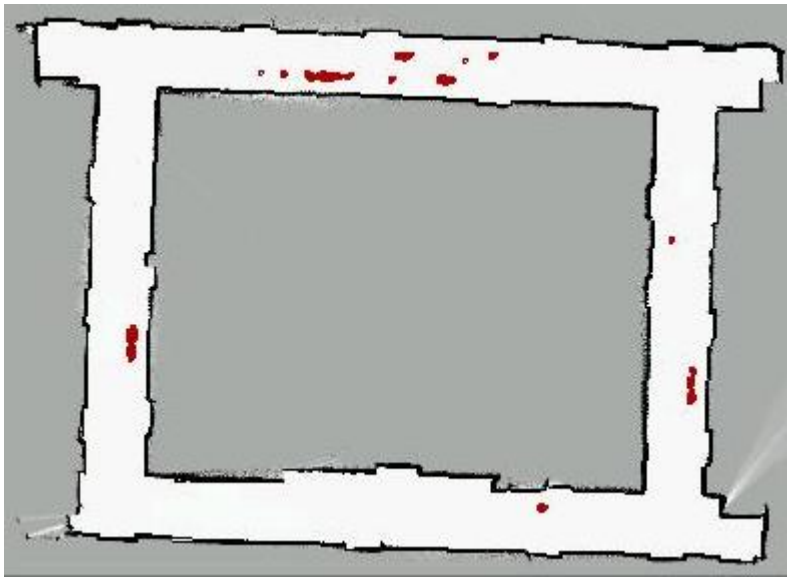
At the implementation level, the amcl driver represents the probability distribution using a particle filter. The filter is adaptive because it dynamically adjusts the number of particles in the filter: when the robot's pose is highly uncertain, the number of particles is increased; when the robot's pose is well determined, the number of particles is decreased. The driver is therefore able make a trade-off between processing speed and localization accuracy.

Finding out our position II



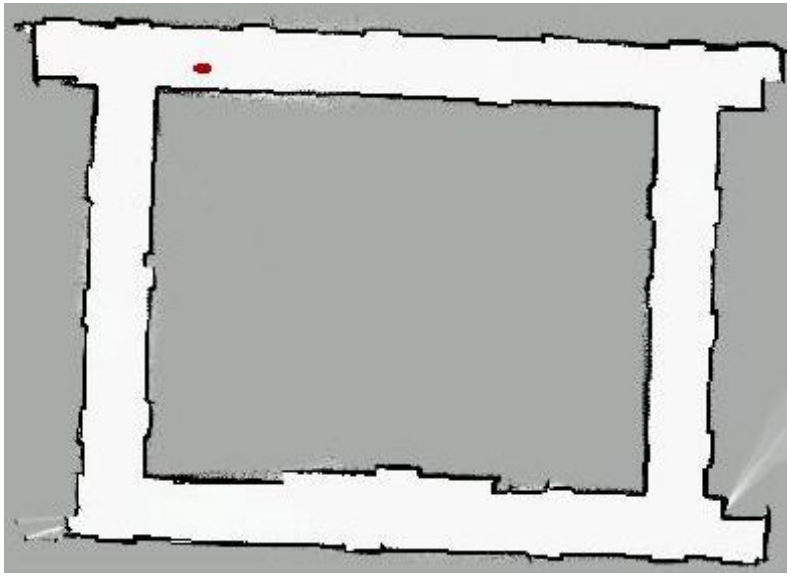
t = 1 sec, approx 100,000 particles

Finding out our position II



t = 40 sec, approx 1,000 particles

Finding out our position II



t = 80 sec, approx 100 particles

Finding out our position III

- If the robot's initial pose is specified as being completely unknown, the driver's estimate will usually converge to correct pose. This assumes that the particle filter starts with a large number of particles (to cover the space of possible poses), and that the robot is driven some distance through the environment (to collect observations).

Finding out our position III

- If the robot's initial pose is specified as being completely unknown, the driver's estimate will usually converge to correct pose. This assumes that the particle filter starts with a large number of particles (to cover the space of possible poses), and that the robot is driven some distance through the environment (to collect observations).
- If the robot's initial pose is specified accurately, but incorrectly, or if the robot becomes lost (e.g., by picking it up and replacing it elsewhere) the driver's estimate will not converge on the correct pose. Such situations require the use of more advanced techniques that have not yet been implemented.

Finding out our position III

- If the robot's initial pose is specified as being completely unknown, the driver's estimate will usually converge to correct pose. This assumes that the particle filter starts with a large number of particles (to cover the space of possible poses), and that the robot is driven some distance through the environment (to collect observations).
- If the robot's initial pose is specified accurately, but incorrectly, or if the robot becomes lost (e.g., by picking it up and replacing it elsewhere) the driver's estimate will not converge on the correct pose. Such situations require the use of more advanced techniques that have not yet been implemented.
- When the number of particles in the filter is large, data may arrive from the sensors faster than it can be processed. When this happens, data is queued up for later processing, but the driver continues to generate an up-to-date estimate for the robot pose.

Finding out our position IV

configuration

```
driver ( name "amcl" provides ["localize:0"                    ]
requires ["odometry:::position2d:0" "laser:0"
"laser:::map:0"]
)
```

- localize : this interface provides a (sort of) representative sample of the current pose hypotheses, weighted by likelihood.

Finding out our position IV

configuration

```
driver ( name "amcl" provides ["localize:0" "position2d:1"]
requires ["odometry:::position2d:0" "laser:0"
"laser:::map:0"]
)
```

- localize : this interface provides a (sort of) representative sample of the current pose hypotheses, weighted by likelihood.
- position2d (optional): this interface provides just the most-likely hypothesis, formatted as position data, which you can (at your peril) pretend came from a perfect odometry system

Outline

Moving in an unknown environment

Amcl solves the localization problem, but it needs a map.

How to obtain the map?

Moving in an unknown environment

Amcl solves the localization problem, but it needs a map.

How to obtain the map?

PLAYER/STAGE does not implement a proper SLAM algorithm (it used to have one, but it not being developed anymore, and the results are quite bad).

Moving in an unknown environment

Amcl solves the localization problem, but it needs a map.

How to obtain the map?

PLAYER/STAGE does not implement a proper SLAM algorithm (it used to have one, but it not being developed anymore, and the results are quite bad).

However, many SLAM algorithms have been implemented and are freely available on the internet (see `http://www.openslam.org/`).

Moving in an unknown environment

Amcl solves the localization problem, but it needs a map.

How to obtain the map?

PLAYER/STAGE does not implement a proper SLAM algorithm (it used to have one, but it not being developed anymore, and the results are quite bad).

However, many SLAM algorithms have been implemented and are freely available on the internet (see `http://www.openslam.org/`).

But, as far as I know, none works natively with PLAYER/STAGE.

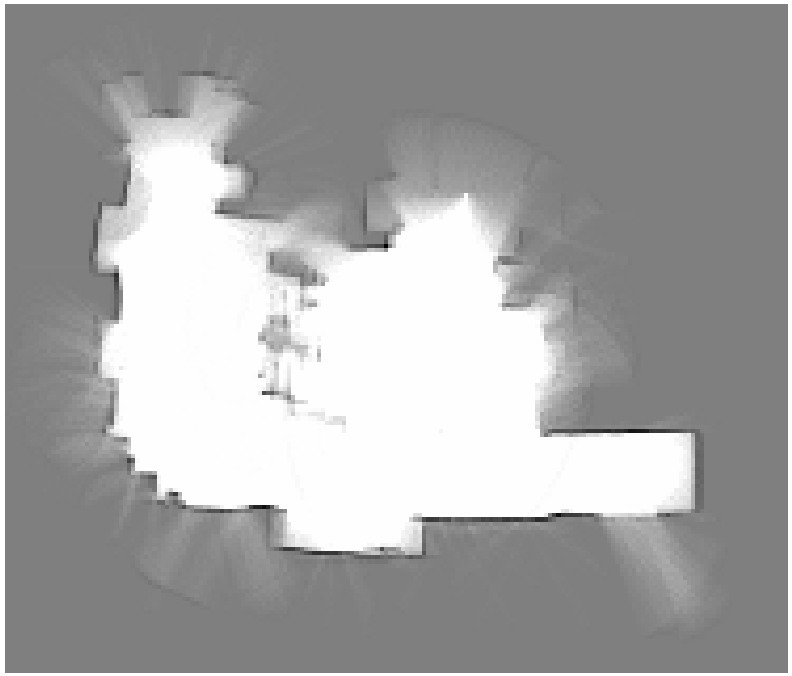
Moving in an unknown environment

We have created a script that make it possible to use GMAPPING using PLAYER logfile. It does that offline, since GMAPPING is quite slow, but it's also possible to perform slam online.

Moving in an unknown environment

We have created a script that make it possible to use GMAPPING using PLAYER logfile. It does that offline, since GMAPPING is quite slow, but it's also possible to perform slam online.
Otherwise, have a look in the MOBILE ROBOT PROGRAMMING TOOLKIT (`http://babel.isa.uma.es/mrpt/index.php/Main_Page`). You'll find several nice demos of (under-documented) C++ classes to perform SLAM

Moving in an unknown environment II



PLAYER/STAGE generated map of the lab

Moving in an unknown environment II

GMAPPING generated map of the lab

Outline

Unfolding the map

The map interface provides access to maps. Depending on the underlying driver, the map may be provided as an occupancy grid, or as a set of segments (or both).

Unfolding the map

The map interface provides access to maps. Depending on the underlying driver, the map may be provided as an occupancy grid, or as a set of segments (or both).
In all our examples, we'll use the occupancy grid approach.

Unfolding the map

The map interface provides access to maps. Depending on the underlying driver, the map may be provided as an occupancy grid, or as a set of segments (or both).
In all our examples, we'll use the occupancy grid approach.
In either case, the map is retrieved by request only (that means you have to request it from the PLAYER server like we did before with position2d).

Unfolding the map II

Map driver declaration in the .cfg file has been heavily modified in the last version of PLAYER/STAGE: if you plan to use it, just copy the proper .cfg section of one of the examples files in your PLAYER/STAGE version.

Unfolding the map III

Let's briefly see how to get the map.

Playerc++ library and map handling

```
PlayerClient robot("localhost",6665);
MapProxy mapp(&robot, 0);
```

Map proxy class instantiation is similar to the position2d proxy class we've seen before.

Unfolding the map III

Let's briefly see how to get the map.

Playerc++ library and map handling

```
PlayerClient robot("localhost",6665);
MapProxy mapp(&robot, 0);

mapp.RequestMap();
robot.Read();
```

Map proxy class instantiation is similar to the position2d proxy class we've seen before.
Here's how to get the map from the server

Unfolding the map III

Playerc++ library and map handling

```
int iMapWidth = mapp.GetWidth();
int iMapHeight = mapp.GetHeight();
 double fMapRes = mapp.GetResolution();
for(int i=0;i<iMapWidth ;i++){
    for(int j=0;j<iMapHeight;j++){
                cout <<(mapp.GetCell(i,j)>0?"X":" ");
            }
       cout <<endl;
     }
```

Here a sample code that gets the map width, height (both in cells) and resolution ($\frac{\text{meters}}{\text{cells}}$). Then, it prints on the console the map (as an occupancy grid)

Outline

Avoiding obstacles

Now we can pinpoint our location and move toward our objective.

Avoiding obstacles

Now we can pinpoint our location and move toward our objective.
But we are still going to blindly collide with anything that stands in our way.
What about avoiding it?

Avoiding obstacles II

PLAYER can do that for you. In particular, it has the VFH driver.

Avoiding obstacles II

PLAYER can do that for you. In particular, it has the VFH driver.

From PLAYER documentation

The vfh driver implements the Vector Field Histogram Plus local navigation method . VFH+ provides real-time obstacle avoidance and path following capabilities for mobile robots. Layered on top of a laser-equipped robot, vfh works great as a local navigation system .

Avoiding obstacles II

PLAYER can do that for you. In particular, it has the VFH driver.

From PLAYER documentation

The vfh driver implements the Vector Field Histogram Plus local navigation method . VFH+ provides real-time obstacle avoidance and path following capabilities for mobile robots. Layered on top of a laser-equipped robot, vfh works great as a local navigation system .

It is possible to configure it very deeply (min_turnrate, safety_dist . . .)

Avoiding obstacles II

PLAYER can do that for you. In particular, it has the VFH driver.

From PLAYER documentation

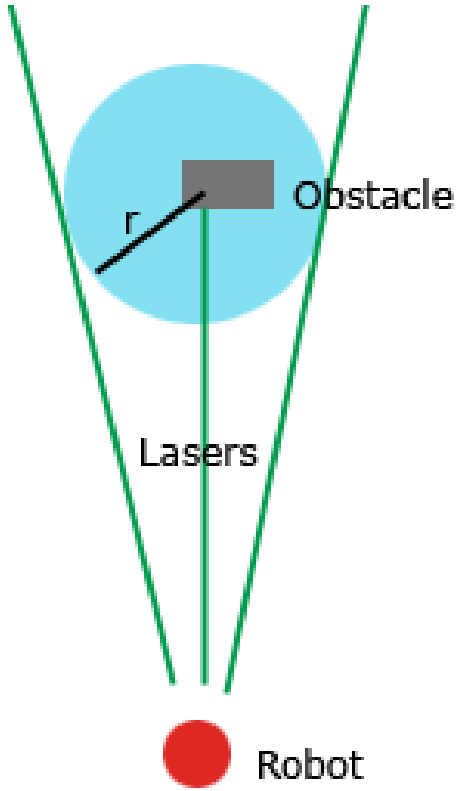
The vfh driver implements the Vector Field Histogram Plus local navigation method . VFH+ provides real-time obstacle avoidance and path following capabilities for mobile robots. Layered on top of a laser-equipped robot, vfh works great as a local navigation system .

It is possible to configure it very deeply (min_turnrate, safety_dist . . .) It is completely transparent: it provides a standard position2d interface

Avoiding obstacles III

Basic configuration

```
driver ( name "vfh"
requires ["position:1" "laser:0"]
provides ["position:0"]
)
```



for more information see `http://cgi.cse.unsw.edu.au/~cs4411/wiki/index.php?title=Path_Planning`

Avoiding obstacles IV

VFH is good in an ample environment, with few narrow passages.

Avoiding obstacles IV

VFH is good in an ample environment, with few narrow passages. That implicitly means that is bad in "crowded" environments.

Avoiding obstacles IV

VFH is good in an ample environment, with few narrow passages. That implicitly means that is bad in "crowded" environments.
The last version of PLAYER/STAGE offers the nd driver, that has an overall better behaviour, even if sometimes the robot heading trembles.

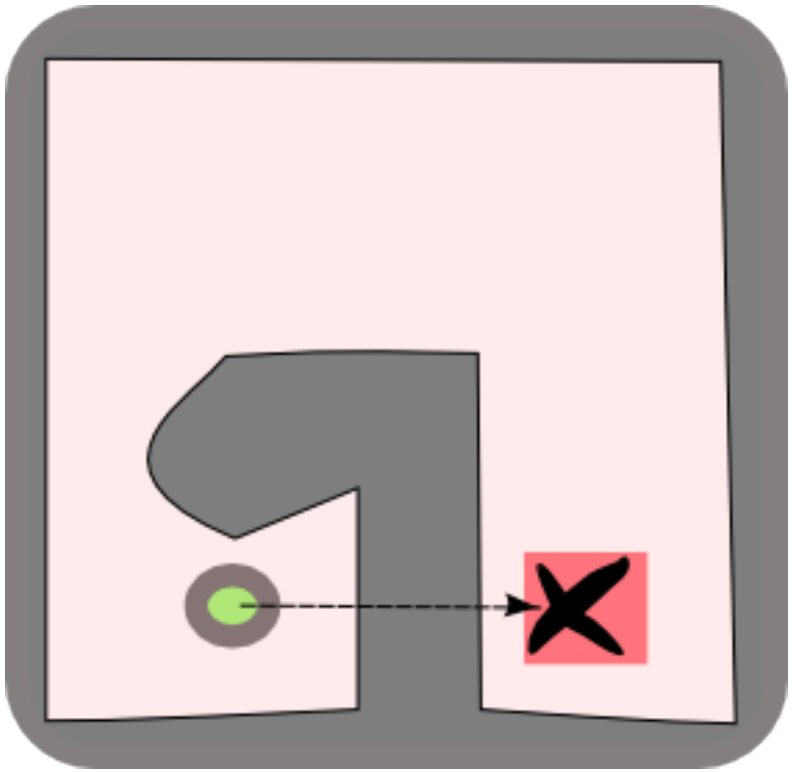
Avoiding obstacles IV

VFH is good in an ample environment, with few narrow passages. That implicitly means that is bad in "crowded" environments.
The last version of PLAYER/STAGE offers the nd driver, that has an overall better behaviour, even if sometimes the robot heading trembles.
And Joey has found a way to improve it.

Outline

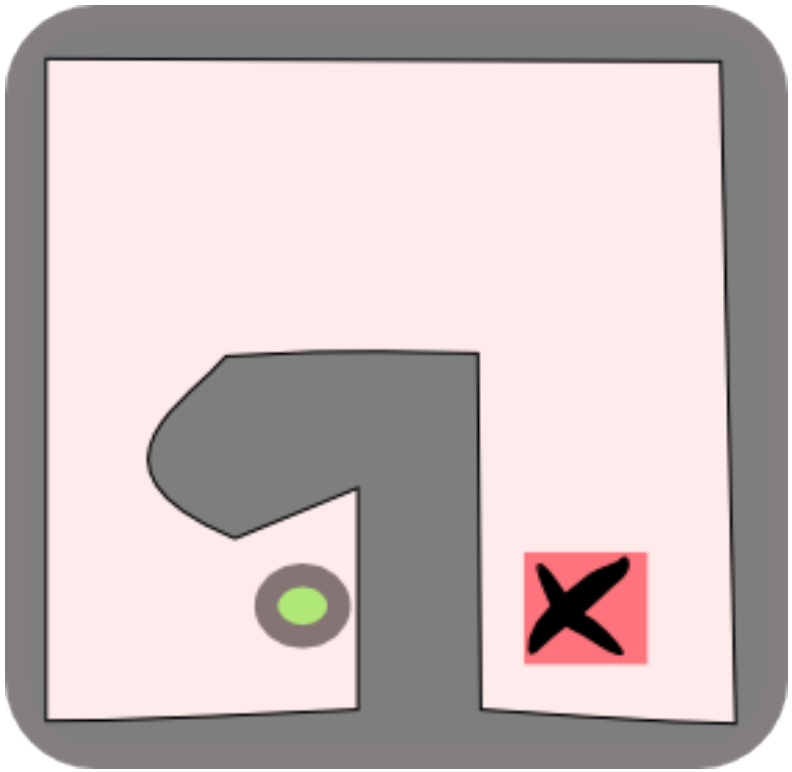
Path planning

Obstacle avoidance is not sufficient, since it's a greedy algorithm



Path planning

Obstacle avoidance is not sufficient, since it's a greedy algorithm, and can get trapped in a not through passage



Path planning II

PLAYER/STAGE offers the driver wavefront, that implements the wavefront algorithm for global path planning.

Path planning II

PLAYER/STAGE offers the driver wavefront, that implements the wavefront algorithm for global path planning.
It's usually layered on the top of vfh

Path planning II

Player/Stage offers the driver wavefront, that implements the wavefront algorithm for global path planning.
It's usually layered on the top of vfh
For more information on the algorithm see

- `http://cgi.cse.unsw.edu.au/~cs4411/wiki/index.php?title=Path_Planning`
- `http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__driver__wavefront.html`

Path planning III

Basic configuration

```
driver ( name "vfh"
provides ["6665:position2d:1"]
requires ["6665:position2d:0" "6665:laser:0"]
distance_epsilon 0.3
angle_epsilon 5
)
```

Path planning III

Basic configuration

```
driver ( name "vfh"
provides ["6665:position2d:1"]
requires ["6665:position2d:0" "6665:laser:0"]
distance_epsilon 0.3
angle_epsilon 5
)
driver ( name "wavefront"
provides ["6665:planner:0"]
requires ["output::6665:position2d:1"
"input::6665:position2d:1" "6665:map:0"]
safety_dist 0.1
distance_epsilon 0.5
angle_epsilon 10
)
```

Path planning III

Basic configuration

```
driver ( name "vfh"
provides ["6665:position2d:1"]
requires ["6665:position2d:0" "6665:laser:0"]
distance_epsilon 0.3
angle_epsilon 5
)
driver ( name "wavefront"
provides ["6665:planner:0"]
requires ["output::6665:position2d:1"
"input::6665:position2d:1" "6665:map:0"]
safety_dist 0.1
distance_epsilon 0.5
angle_epsilon 10
)
```

Path planning IV

Playerc++ library and map handling

```
PlayerClient robot("localhost",6665);
PlannerProxy planp(&robot, 0);
planp.SetGoalPose (10,20,0);
planp.SetEnable(true);
```

Outline

Of course, it is possible to get the laser readings and configure the laser.

Playerc++ library and map handling

```
PlayerClient robot("localhost",6665);
LaserProxy lasp(&robot, 0)));
for (int i=0;i<(int)lasp.GetCount();++i) {
    lasp.GetRange(i);
}
```

Outline

The erratic driver provides the following device interfaces:

- "odometry" position2d: This interface returns odometry data, and accepts velocity commands.

The erratic driver provides the following device interfaces:

- "odometry" position2d: This interface returns odometry data, and accepts velocity commands.
- power: Returns the current battery voltage (12 V when fully charged).

The erratic driver provides the following device interfaces:

- "odometry" position2d: This interface returns odometry data, and accepts velocity commands.
- power: Returns the current battery voltage (12 V when fully charged).
- aio: Returns data from analog and digital input pins

The erratic driver provides the following device interfaces:

- "odometry" position2d: This interface returns odometry data, and accepts velocity commands.
- power: Returns the current battery voltage (12 V when fully charged).
- aio: Returns data from analog and digital input pins
- ir: Returns ranges from IR sensors, assuming they're connected to the analog input pins

The erratic driver provides the following device interfaces:

- "odometry" position2d: This interface returns odometry data, and accepts velocity commands.
- power: Returns the current battery voltage (12 V when fully charged).
- aio: Returns data from analog and digital input pins
- ir: Returns ranges from IR sensors, assuming they're connected to the analog input pins
- laser: Returns ranges from laser sensors

The erratic driver provides the following device interfaces:

- "odometry" position2d: This interface returns odometry data, and accepts velocity commands.
- power: Returns the current battery voltage (12 V when fully charged).
- aio: Returns data from analog and digital input pins
- ir: Returns ranges from IR sensors, assuming they're connected to the analog input pins
- laser: Returns ranges from laser sensors
- ptz : control of the servos that pan and tilt

The erratic driver provides the following device interfaces:

- "odometry" position2d: This interface returns odometry data, and accepts velocity commands.
- power: Returns the current battery voltage (12 V when fully charged).
- aio: Returns data from analog and digital input pins
- ir: Returns ranges from IR sensors, assuming they're connected to the analog input pins
- laser: Returns ranges from laser sensors
- ptz : control of the servos that pan and tilt

For more information see:
`http://www.videredesign.com/robots/era_mobi.htm`

There are a lot of other drivers (e.g. drawing on the Stage interface...).
In the next slide there are reference for them.

There are a lot of other drivers (e.g. drawing on the Stage interface. . .).
In the next slide there are reference for them.
Sadly, the set of available drivers changes in different versions of stage (but it's not expanded: several important drivers have been dropped in the last version of Player/Stage)

There are a lot of other drivers (e.g. drawing on the Stage interface. . .).
In the next slide there are reference for them.
Sadly, the set of available drivers changes in different versions of stage (but it's not expanded: several important drivers have been dropped in the last version of PLAYER/STAGE)
For example: the graphic driver is not supported yet, and the map driver has changed completely (and that includes its configuration file)

Reference

- http://playerstage.sourceforge.net/doc/Player-2.0.0/player/modules.html
- http://playerstage.sourceforge.net/doc/Player-2.1.0/player/modules.html
- http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__drivers.html
- http://playerstage.sourceforge.net/doc/Player-2.1.0/player/group__drivers.html