Introduction to Player/Stage

Part II

Dario Cazzaro, Luca Invernizzi

Motion Lab, UCSB

October 16, 2008

Outline

1. Compiling
2. Standard Template Library
3. Connecting to the real robots
4. Simulating a wireless network
5. Solving the multi-hop dynamic routing problem

Compiling user clients

To compile a Player client you only need to include Player libraries and link your program with them. So, in your source code:

myclient.cpp

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

To compile and link it:

```
user@host$ gcc -o myclient ‘pkg-config --cflags playerc++‘
           ‘pkg-config --libs player c++‘ myclient.cpp
```

To run it:

```
user@host$ ./myclient
```

Outline

1. Compiling

2. Standard Template Library

3. Connecting to the real robots

4. Simulating a wireless network

5. Solving the multi-hop dynamic routing problem

Standard Template Library

The Standard Template Library (*STL*) is a C++ library which provides lots of data structures and algorithms you may need.
You can use library's algorithms instead of implementing them yourselves, avoiding boring coding-nights and time wasting errors.
Almost always, the STL algorithms are very efficient, both about time and memory use.
To make the *STL* as generic as possible, almost every component in it is a *template*.

What is a template?

Templates are a C++ feature that allow classes to work on different data types without being rewritten for each one.
When you instantiate a template class object, you have to define the data types you want your class will work on.

What is a template?

Templates are a C++ feature that allow classes to work on different data types without being rewritten for each one.

When you instantiate a template class object, you have to define the data types you want your class will work on.

For example, a *pair* is a simple container that lets you memorize a couple of objects. These objects may be of any type, both simple data or complex class objects.

What is a template?

To declare a *pair* object, you must define the types of the data you want to put in it. If you want a couple of integer, just pass *int* as *parameters* to the template:

```
pair < int , int > oldpair;
```

What is a template?

To declare a *pair* object, you must define the types of the data you want to put in it. If you want a couple of integer, just pass *int* as *parameters* to the template:

```
pair < int , int > oldpair;
```

If you want to store a char and a double:

```
pair < char , double > newpair;
```

What is a template?

First, let's assign some values to the pairs:

```
oldpair.first = 3;
oldpair.second = 14;
newpair.first = 'e';
newpair.second = 2.71828183;
```

What is a template?

First, let's assign some values to the pairs:

```
oldpair.first = 3;
oldpair.second = 14;
newpair.first = 'e';
newpair.second = 2.71828183;
```

Now, these two objects have the same names for their member data and functions:

```
cout << firstpair.first << secondpair.first;
```

What is a template?

First, let's assign some values to the pairs:

```
oldpair.first = 3;
oldpair.second = 14;
newpair.first = 'e';
newpair.second = 2.71828183;
```

Now, these two objects have the same names for their member data and functions:

```
cout << firstpair.first << secondpair.first;
```

Warning: this doesn't mean we can assign to an object any type we want, but only the type which it was declared with!

Containers

Containers are data structure used to store collection of data.
The STL provides many kinds of container, according to the data access method and the time/memory complexity to perform operations.
When you create a container, you must declare the type of the data you'll put in it.
They includes:

- Sequence containers
- Associative containers

Containers

Containers are data structure used to store collection of data.
The STL provides many kinds of container, according to the data access method and the time/memory complexity to perform operations.
When you create a container, you must declare the type of the data you'll put in it.
They includes:

- Sequence containers
    - Vector
    - List
    - Deque
- Associative containers

Containers

Containers are data structure used to store collection of data.
The STL provides many kinds of container, according to the data access method and the time/memory complexity to perform operations.
When you create a container, you must declare the type of the data you'll put in it.
They includes:

- Sequence containers
  - Vector
  - List
  - Deque
- Associative containers
  - Set
  - Map

Vector

The *vector* class implements a dynamic array, which behaves like a C array, but has the ability to resize itself if it's full.
So, when you initialize a *vector*, you don't have to worry about its size since it grows automatically when needed.
To access data, both for reading and writing, you can use the usual operator "[]"

Vector

The *vector* class implements a dynamic array, which behaves like a C array, but has the ability to resize itself if it's full.
So, when you initialize a *vector*, you don't have to worry about its size since it grows automatically when needed.
To access data, both for reading and writing, you can use the usual operator "[]"

Usage example

```
#include <vector>
...
vector<int> myvector;
for(int i=0; i<20; i++) myvector[i] = i;
```

Vector

Do you want a Matrix? It's simple: create a vector of vectors!

Vector

Do you want a Matrix? It's simple: create a vector of vectors!

Usage example

```
#include <vector>
...
vector<vector<int> > myvector;
for(int i=0; i<20; i++) myvector[i][i] = i;
```

Vector

Do you want a Matrix? It's simple: create a vector of vectors!

Usage example

```
#include <vector>
...
vector<vector<int> > myvector;
for(int i=0; i<20; i++) myvector[i][i] = i;
```

Warning: remember the space char between the "$>$" markers; the compiler could mistake them for the "$>>$" operator!

Iterators

An *iterator* is an object which points to an element of a container.
You can ask a container for an iterator and then use it to navigate through the data set

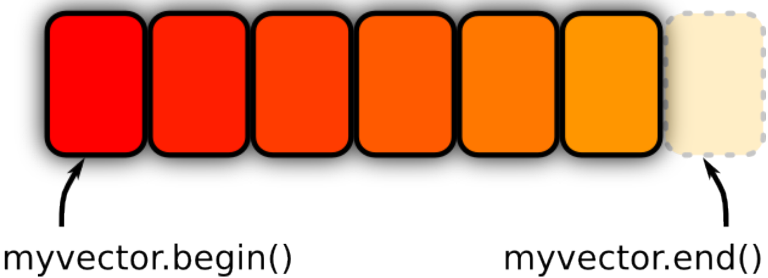
Iterators

An *iterator* is an object which points to an element of a container.
You can ask a container for an iterator and then use it to navigate through the data set

Usage example

```
#include <vector>
...
vector<int> myvector;
vector<int>::iterator i;
for(i = myvector.begin(); i != myvector.end(); i++)
(*i) = 1;
```

Iterators



Deque

A *Deque* is a container which looks like a vector, but it has better performances.
They both support random access to elements (with the operator "[]"), but the deque also supports constant time insertion and removal of elements at the beginning and at the end of the sequence.

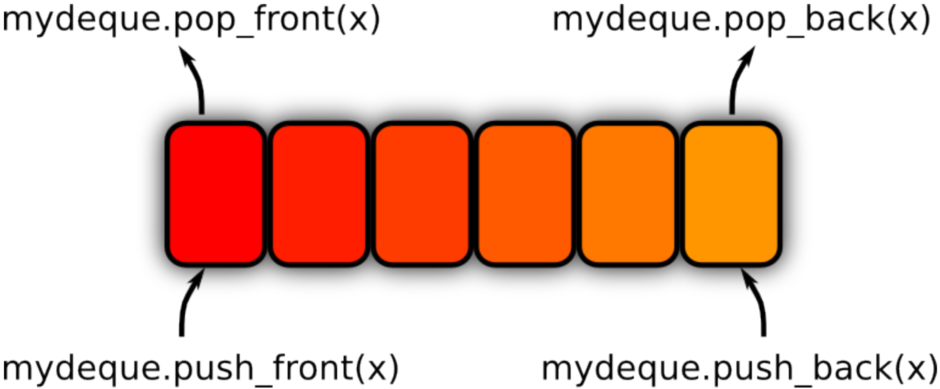
Deque

A *Deque* is a container which looks like a vector, but it has better performances.
They both support random access to elements (with the operator "[]"), but the deque also supports constant time insertion and removal of elements at the beginning and at the end of the sequence.

Usage example

```
#include <deque>
...
deque<int> mydeque;
mydeque.push_back(1); // [1]
mydeque.push_front(2); // [2 1]
mydeque.push_back(3); // [2 1 3]
mydeque.pop_front(); // [1 3]
int a = mydeque.back(); // a = 3
```

Deque


mydeque.pop_front(x)
mydeque.pop_back(x)
mydeque.push_front(x)
mydeque.push_back(x)

Map

A *Map* is an associative container which lets you refer to an element using a key.
As always, you can set both the type of the data and the type of the key.
Since the keys uniquely identify the elements, there cannot be multiple elements with the same key.

Map

A *Map* is an associative container which lets you refer to an element using a key.

As always, you can set both the type of the data and the type of the key.

Since the keys uniquely identify the elements, there cannot be multiple elements with the same key.

Usage example

```
#include <map>
...
map<string, int> ages;
map<string, int>::iterator i;
ages[‘‘John’’] = 9;
ages[‘‘Tom’’] = 30;
i = ages.find(‘‘Matt’’);
for(i = ages.begin(); i!=ages.end(); i++)
cout << (*i).first << (*i).second;
```

Algorithms

The STL also provides many useful algorithms, especially designed to be used on STL collections.
Most of them are applied to a couple of iterators: the beginning() end the end() iterators of the collection (or a subset of it).

Algorithms

The STL also provides many useful algorithms, especially designed to be used on STL collections.
Most of them are applied to a couple of iterators: the beginning() end the end() iterators of the collection (or a subset of it).
They includes:

- find
- count
- reverse
- unique
- sort
- merge
- min and max

Algorithms

Usage example

```
#include <vector>
#include <algorithm>
...
vector<int> v;
//fill the vector in some way
sort( v.begin(), v.end() );
```

Algorithms

Usage example

```
#include <vector>
#include <algorithm>
...
vector<int> v;
//fill the vector in some way
sort( v.begin(), v.end() );
unique( v.begin(), v.end() );
```

Algorithms

Usage example

```
#include <vector>
#include <algorithm>
...
vector<int> v;
//fill the vector in some way
sort( v.begin(), v.end() );
unique( v.begin(), v.end() );
vector<int>::iterator i;
i = find( v.begin(), v.end(), value);
```

Algorithms

Usage example

```
#include <vector>
#include <algorithm>
...
vector<int> v;
//fill the vector in some way
sort( v.begin(), v.end() );
unique( v.begin(), v.end() );
vector<int>::iterator i;
i = find( v.begin(), v.end(), value);
if ( binary_search( v.begin(), v.end(), value) )
cout << ‘‘found!’’;
```

Player/Stage example

In this example, we'll use STL containers to store Player map data

```
#include <set>
#include <algorithm>
#include <libplayerc++/playerc++.h>
...
MapProxy mp = new MapProxy(robot,0);
set< pair< int, int > > s;
pair< int, int> p;
for (int j = 0; j < mp->GetHeight(); j++)
for (int i = 0; i < mp->GetWidth(); i++){
p.first = mp->GetCellIndex(i, j);
p.second = mp->GetCellIndex(i+1, j);
s.insert(p);
}
```

Sky is the limit

Being all objects dynamically allocated, you don't have to worry about memory. So you can nest all these data structure as much as you want.

Sky is the limit

Being all objects dynamically allocated, you don't have to worry about memory. So you can nest all these data structure as much as you want.

```
map < pair< string , int>, pair< vector< vector<int> >,
list<double> > > hugedatastructure;
```

Sky is the limit

Being all objects dynamically allocated, you don't have to worry about memory. So you can nest all these data structure as much as you want.

```
map < pair< string , int>, pair< vector< vector<int> >,
list<double> > > hugedatastructure;
```

Sky is the limit

Being all objects dynamically allocated, you don't have to worry about memory. So you can nest all these data structure as much as you want.

```
map < pair< string , int>, pair< vector< vector<int> >,
list<double> > > hugedatastructure;
```

Sky is the limit

Being all objects dynamically allocated, you don't have to worry about memory. So you can nest all these data structure as much as you want.

```
map < pair< string , int>, pair< vector< vector<int> >,
list<double> > > hugedatastructure;
```

Sky is the limit

Being all objects dynamically allocated, you don't have to worry about memory. So you can nest all these data structure as much as you want.

```
map < pair< string , int>, pair< vector< vector<int> >,
list<double> > > hugedatastructure;
```

Notes and References

When you use the STL always put this line into your C++ code

```
using namespace std;
```

to tell the compiler you are using the *standard* namespace.

Notes and References

When you use the STL always put this line into your C++ code

```
using namespace std;
```

to tell the compiler you are using the *standard* namespace.

Here are some useful websites where you can find further information:

www.sgi.com/tech/stl/ Complete reference of the STL

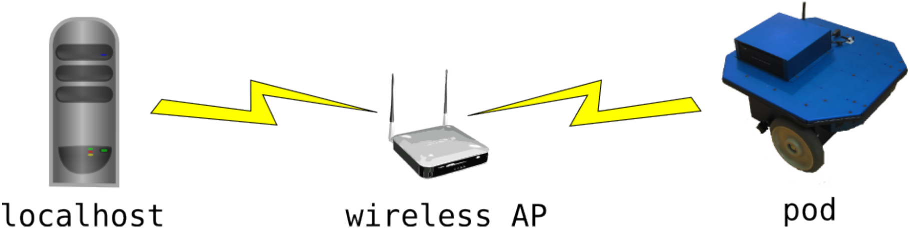
www.cppreference.com/wiki/stl/start Another good reference

www.cs.brown.edu/ jak/proglang/cpp/stltut/tut.html A brief tutorial

Outline

Connecting to the real robots

Connect your laptop to the wireless network (called "motionlab")



On linux

```
sudo iwconfig wlan0 essid "motionlab"
sudo dhclient wlan0
```

Connecting to the real robots

Log in the robot operating system

On linux, mac, windows

```
ssh  erratic@192.168.1.12
```

Connecting to the real robots

Log in the robot operating system

On linux, mac, windows

```
ssh  erratic@192.168.1.12
```

- username: erratic

Connecting to the real robots

Log in the robot operating system

On linux, mac, windows

```
ssh  erratic@192.168.1.12
```

- username: erratic
- hostname or ip address: 192.168.1.12

Connecting to the real robots

advanced options

Log in the robot operating system

On linux, mac, windows

```
ssh  erratic@pod
```

- username: erratic
- hostname or ip address: pod
  To use the hostname you have to manually add it to the local hostname list on your pc , since there the DNS does not provide them

on linux and mac

```
sudo echo "192.168.1.12 pod" >> /etc/hosts
```

Connecting to the real robots

advanced options

Log in the robot operating system

On linux, mac, windows

```
ssh -Y erratic@pod
```

- username: erratic
- hostname or ip address: pod
  To use the hostname you have to manually add it to the local hostname list on your pc , since there the DNS does not provide them

on linux and mac

```
sudo echo "192.168.1.12 pod" >> /etc/hosts
```

- use this option if you want to activate graphics forwarding (e.g. if you want to use directly the stereo camera)

Connecting to the real robots

Activating Player Server

Now that you have logged in the robot, execute:

On the robot shell

```
player ~/player/config/erratic.cfg
```

This command activates the PLAYER server on the robot (listening on the default port 6665, on the wireless interface).

Checking the connection

To check that everything is working as it should, try a simple connection to the robot

Command line

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

Checking the connection

To check that everything is working as it should, try a simple connection to the robot

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
```

PLAYERS that talk to each other

You'll probably want to connect the robot to the player server that's running on your laptop (e.g, if you're using both simulated and real robots at the same time).

PLAYERS that talk to each other

You'll probably want to connect the robot to the player server that's running on your laptop (e.g, if you're using both simulated and real robots at the same time).
To do that, you have three possible choices:

PLAYERs that talk to each other

You'll probably want to connect the robot to the player server that's running on your laptop (e.g, if you're using both simulated and real robots at the same time).

To do that, you have three possible choices:

- telling PLAYER on localhost to connect to the robot

PLAYERs that talk to each other

You'll probably want to connect the robot to the player server that's running on your laptop (e.g, if you're using both simulated and real robots at the same time).
To do that, you have three possible choices:

- telling PLAYER on localhost to connect to the robot
- using the passthrough driver

PLAYERs that talk to each other

You'll probably want to connect the robot to the player server that's running on your laptop (e.g, if you're using both simulated and real robots at the same time).
To do that, you have three possible choices:

- telling PLAYER on localhost to connect to the robot
- using the passthrough driver
- using port forwarding

PLAYERs that talk to each other

You'll probably want to connect the robot to the player server that's running on your laptop (e.g, if you're using both simulated and real robots at the same time).

To do that, you have three possible choices:

- telling PLAYER on localhost to connect to the robot
- using the passthrough driver
- using port forwarding

We'll go briefly through them

PLAYERs that talk to each other

telling PLAYER on localhost to connect to the robot

In this example, we run the VFH driver (i.e. obstacle avoidance) on localhost, telling PLAYER to get the position and the laser readings from pod.

PLAYER configuration

```
driver
(
  name "vfh"
  provides ["localhost:6665:position2d:1"]
  requires ["pod:6665:position2d:0" "pod:6665:laser:0"]
)
```

PLAYERs that talk to each other

telling PLAYER on localhost to connect to the robot

In this example, we run the VFH driver (i.e. obstacle avoidance) on localhost, telling PLAYER to get the position and the laser readings from pod.

PLAYER configuration

```
driver
(
  name "vfh"
  provides ["localhost:6665:position2d:1"]
  requires ["pod:6665:position2d:0" "pod:6665:laser:0"]
)
```

- PLAYER connects to pod to get the data

PLAYERs that talk to each other

telling PLAYER on localhost to connect to the robot

In this example, we run the VFH driver (i.e. obstacle avoidance) on localhost, telling PLAYER to get the position and the laser readings from pod.

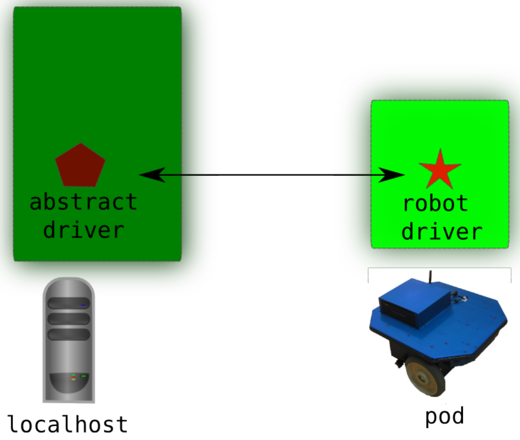
PLAYER configuration

```
driver
(
  name "vfh"
  provides ["localhost:6665:position2d:1"]
  requires ["pod:6665:position2d:0" "pod:6665:laser:0"]
)
```

- PLAYER connects to pod to get the data
- PLAYER provides the VFH driver on localhost

PLAYERs that talk to each other

telling PLAYER on localhost to connect to the robot



PLAYERS that talk to each other

telling PLAYER on localhost to connect to the robot

Pros:

- It's easy: just a change in the configuration file

Cons:

- It's transparent: if a client wants to read the laser, it will have to connect directly to the robot.
- It's inconvenient: you have to change the configuration file each time you switch from real robot to the simulated one (or keep two synchronized files, and that could be error prone)

PLAYERS that talk to each other

telling PLAYER on localhost to connect to the robot

Pros:

- It's easy: just a change in the configuration file

Cons:

- It's transparent: if a client wants to read the laser, it will have to connect directly to the robot.
- It's inconvenient: you have to change the configuration file each time you switch from real robot to the simulated one (or keep two synchronized files, and that could be error prone)

PLAYERS that talk to each other

telling PLAYER on localhost to connect to the robot

Pros:

- It's easy: just a change in the configuration file

Cons:

- It's transparent: if a client wants to read the laser, it will have to connect directly to the robot.
- It's inconvenient: you have to change the configuration file each time you switch from real robot to the simulated one (or keep two synchronized files, and that could be error prone)

PLAYERs that talk to each other

using the PASSTHROUGH driver

In the last version of PLAYER the developer have implemented a driver that works as a proxy: the PASSTHROUGH driver

PLAYER configuration

```
driver (
 name "passthrough"
 provides ["position2d:0"]
 requires ["position2d:0"]
 remote_host "pod"
 remote_port 6665
 remote_index 0
 access "a"
)
```

PLAYERs that talk to each other

using the PASSTHROUGH driver

In the last version of PLAYER the developer have implemented a driver that works as a proxy: the PASSTHROUGH driver

PLAYER configuration

```
driver (
 name "passthrough"
 provides ["position2d:0"]
 requires ["position2d:0"]
 remote_host "pod"
 remote_port 6665
 remote_index 0
 access "a"
)
```

- PLAYER connects to pod to get the data

PLAYERs that talk to each other

using the PASSTHROUGH driver

In the last version of PLAYER the developer have implemented a driver that works as a proxy: the PASSTHROUGH driver

PLAYER configuration

```
driver (
 name "passthrough"
 provides ["position2d:0"]
 requires ["position2d:0"]
 remote_host "pod"
 remote_port 6665
 remote_index 0
 access "a"
)
```

- PLAYER connects to pod to get the data (in read-write mode)

PLAYERS that talk to each other

using the PASSTHROUGH driver

In the last version of PLAYER the developer have implemented a driver that works as a proxy: the PASSTHROUGH driver

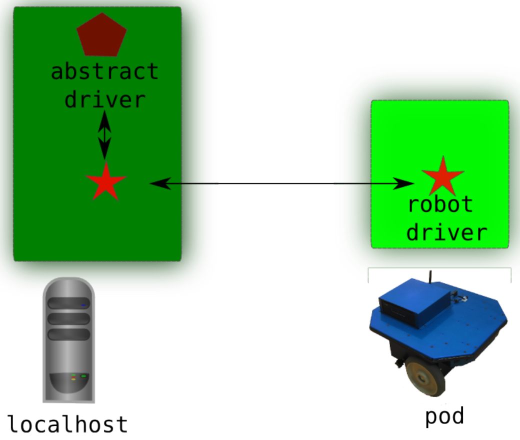
PLAYER configuration

```
driver (
 name "passthrough"
 provides ["position2d:0"]
 requires ["position2d:0"]
 remote_host "pod"
 remote_port 6665
 remote_index 0
 access "a"
)
```

- PLAYER connects to pod to get the data
- PLAYER provides the same driver on localhost

PLAYERS that talk to each other

using the PASSTHROUGH driver



PLAYERS that talk to each other

using the PASSTHROUGH driver

Pros:

- It's easy: just a change in the configuration file
- It's opaque: the client doesn't know if the robot is real or not: it always connects to localhost

Cons:

- It's inconvenient: you have to change the configuration file each time you switch from real robot to the simulated one (or keep two synchronized files, and that could be error prone)

PLAYERS that talk to each other

using the PASSTHROUGH driver

Pros:

- It's easy: just a change in the configuration file
- It's opaque: the client doesn't know if the robot is real or not: it always connects to localhost

Cons:

- It's inconvenient: you have to change the configuration file each time you switch from real robot to the simulated one (or keep two synchronized files, and that could be error prone)

PLAYERS that talk to each other

using the PASSTHROUGH driver

Pros:

- It's easy: just a change in the configuration file
- It's opaque: the client doesn't know if the robot is real or not: it always connects to localhost

Cons:

- It's inconvenient: you have to change the configuration file each time you switch from real robot to the simulated one (or keep two synchronized files, and that could be error prone)

PLAYERS that talk to each other

using port forwarding

This method requires a little more setup time, but it's a good way to avoid the need to change the configuration file in order to switch from simulation to the real robots and vice-versa.
The configuration of the localhost is always this:

PLAYER configuration

```
driver (
  name "vfh"
  provides ["6665:position2d:1"]
  requires ["6666:position2d:0" "6666:laser:0"]
)
```

PLAYERs that talk to each other

using port forwarding

This method requires a little more setup time, but it's a good way to avoid the need to change the configuration file in order to switch from simulation to the real robots and vice-versa.
The configuration of the localhost is always this:

PLAYER configuration

```
driver (
  name "vfh"
  provides ["6665:position2d:1"]
  requires ["6666:position2d:0" "6666:laser:0"]
)
```

- PLAYER connects to localhost on port 6666 to get the data

PLAYERs that talk to each other

using port forwarding

This method requires a little more setup time, but it's a good way to avoid the need to change the configuration file in order to switch from simulation to the real robots and vice-versa.
The configuration of the localhost is always this:

PLAYER configuration

```
driver (
  name "vfh"
  provides ["6665:position2d:1"]
  requires ["6666:position2d:0" "6666:laser:0"]
)
```

- PLAYER connects to localhost on port 6666 to get the data
- PLAYER provides the VFH driver on localhost

PLAYERS that talk to each other

using port forwarding

PLAYER configuration

```
driver (
  name "vfh"
  provides ["6665:position2d:1"]
  requires ["6666:position2d:0" "6666:laser:0"]
)
```

To simulate the robot, we just start another player with this configuration file:

PLAYER configuration

```
driver (
name "stage"
model "robot1"
provides ["6666:position2d:0" "6666:laser:0]
)
```

PLAYERs that talk to each other

using port forwarding

When we're dealing with a real robot, we would like to proxy the robot PLAYER port on localhost:6666.

.

PLAYERs that talk to each other

using port forwarding

When we're dealing with a real robot, we would like to proxy the robot PLAYER port on localhost:6666.
This can be done through port forwarding.

PLAYERs that talk to each other

using port forwarding

When we're dealing with a real robot, we would like to proxy the robot PLAYER port on localhost:6666.
This can be done through port forwarding.

Console command

```
user@host$ ssh -N -L 6666:pod:6665 erratic@pod
```

- Forwarding

PLAYERS that talk to each other
using port forwarding

When we're dealing with a real robot, we would like to proxy the robot PLAYER port on localhost:6666.
This can be done through port forwarding.

Console command

```
user@host$ ssh -N -L 6666:pod:6665 erratic@pod
```

- Forwarding
- from port 6665 on host pod

PLAYERS that talk to each other

using port forwarding

When we're dealing with a real robot, we would like to proxy the robot PLAYER port on localhost:6666.

This can be done through port forwarding.

Console command

```
user@host$ ssh -N -L 6666:pod:6665 erratic@pod
```

- Forwarding
- from port 6665 on host pod
- to localhost port 6666

PLAYERs that talk to each other

using port forwarding

When we're dealing with a real robot, we would like to proxy the robot PLAYER port on localhost:6666.

This can be done through port forwarding.

Console command

```
user@host$ ssh -N -L 6666:pod:6665 erratic@pod
```

- Forwarding
- from port 6665 on host pod
- to localhost port 6666
- without asking for a shell

PLAYERS that talk to each other

using port forwarding

When we're dealing with a real robot, we would like to proxy the robot PLAYER port on localhost:6666.
This can be done through port forwarding.

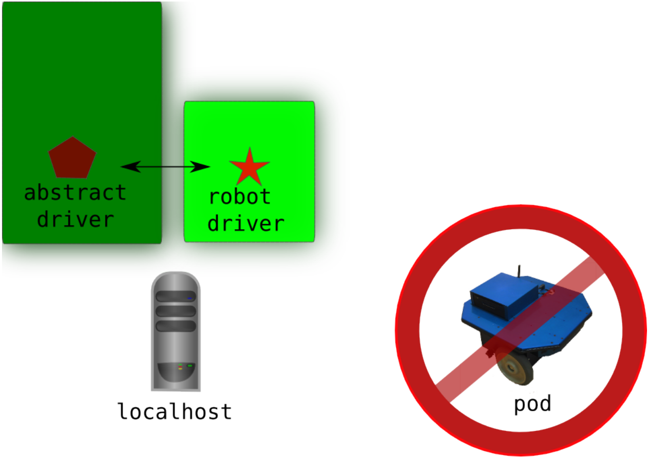
Console command

```
user@host$ ssh -N -L 6666:pod:6665 erratic@pod
```

- Forwarding
- from port 6665 on host pod
- to localhost port 6666
- without asking for a shell
- and logging in host pod with user erratic

PLAYERS that talk to each other

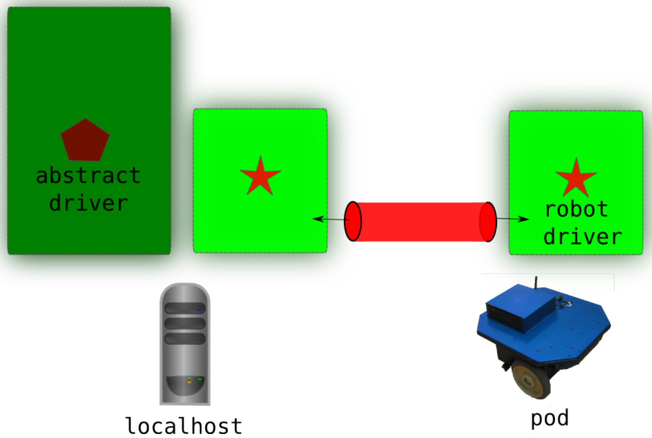
using port forwarding



Simulating

PLAYERS that talk to each other

using port forwarding



Working with hardware

PLAYERs that talk to each other

using port forwarding

Pros:

- It's opaque: the client doesn't know if the robot is real or not: it always connects to localhost
- It's convenient: the PLAYER configuration file does not change

Cons:

- It's not so easy: it needs a proper setup

PLAYERs that talk to each other
using port forwarding

Pros:

- It's opaque: the client doesn't know if the robot is real or not: it always connects to localhost
- It's convenient: the PLAYER configuration file does not change

Cons:

- It's not so easy: it needs a proper setup

PLAYERS that talk to each other

using port forwarding

Pros:

- It's opaque: the client doesn't know if the robot is real or not: it always connects to localhost
- It's convenient: the PLAYER configuration file does not change

Cons:

- It's not so easy: it needs a proper setup

Outline

1. Compiling
2. Standard Template Library
3. Connecting to the real robots
4. Simulating a wireless network
5. Solving the multi-hop dynamic routing problem

Simulating the wireless

Player does not have a built-in network simulation (it does have a WIFI driver, but that provides only signal strength).
In order to work at the same time with simulated and real robots, we developed a server that simulates wireless range-limited communication, and its client-side library.

Simulating the wireless

Server

The server is easy to start:

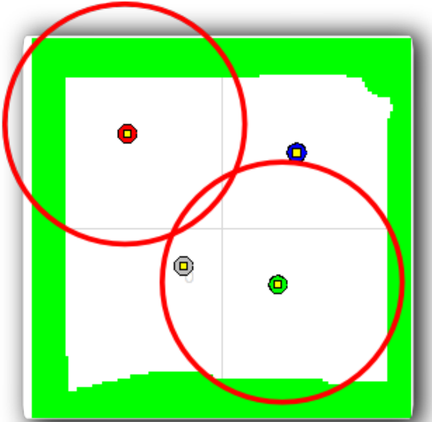
Command line

```
./wireless <listening_ip> <listening_port> <player_port>
```

The server uses the TCP protocol (i.e. the communication is reliable – so you don't have to deal with reordering the packets, packet losses..)

Simulating the wireless

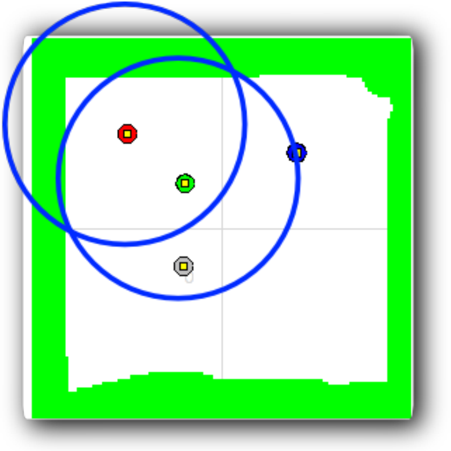
The graphical interface



Failed communication

Simulating the wireless

The graphical interface



Successful communication

Simulating the wireless

Server

To configure the server, just have a look at the file `wireless_config.h`:

- `#define WIRELESS_RANGE 2`: communication radius in meters
- `#define GRAPHICS_ACTIVE 1`: the server will draw the communication radii of each robot upon communication (in blue if the communication has taken place, in red if it has failed)

Simulating the wireless

Client - sending

This is how you send a message:

C++ code

```
wireless_client.write_to(dest, "hi");
```

Destination can be:

- a robot nickname
- "all" for broadcasting

Simulating the wireless

Client - receiving

This is how you check if there is a message, and receive it:

C++ code

```
if (wireless_client->receive(buffer) > 0) {
    printf("message received: %s", buffer);
} else {
    printf("no message");
}
```

The `receive ()` function is non blocking (there is a thread in the client library dedicated to handling the TCP socket, so you don't have to).

Outline

Routing algorithms

If you're deploying an algorithm implementation on real robot that have to travel far away or in different rooms, you'll have to deal with range-limited communication.

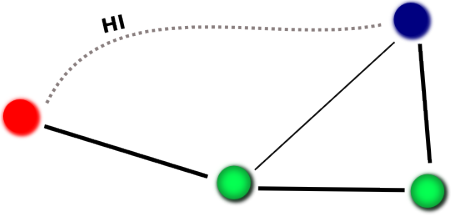
Routing algorithms

If you're deploying an algorithm implementation on real robot that have to travel far away or in different rooms, you'll have to deal with range-limited communication.

A way to ease up that problem is to use a routing protocol, so to obtain a wireless multi-hop ad-hoc network. You'll also want your routing to be dynamic, since your robots are going to move.

Routing algorithms

Example

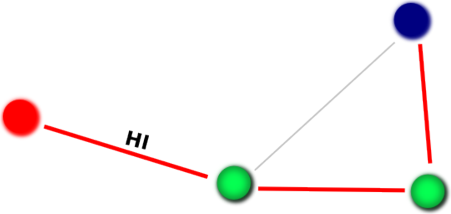


$$\text{RED} \quad \not\longmapsto \quad \text{BLUE}$$

Suppose that node Red wants to send a message to node Blue. Sadly, the nodes are not 1-hop neighbours, so it can't send it the message directly

Routing algorithms

Example

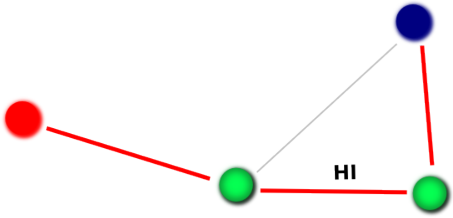


$$\text{RED} \qquad \longmapsto \qquad \text{GREEN1}$$

Red then decide to forward the message to the only node in its range. Red knows that that robot is in contact with blue because the routing table built by the routing software tell it so.

Routing algorithms

Example

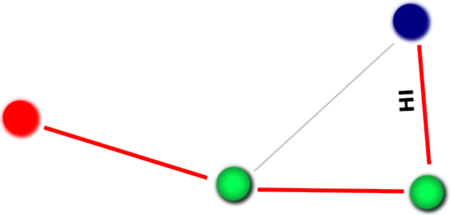


$$\text{GREEN1} \quad \longmapsto \quad \text{GREEN2}$$

Green1 decides to forward the packet to Green2 because its instance of the routing algorithm (that, being *proactive*, was flooding the network to test all the possible connections it could make) tells it that that route is more reliable

Routing algorithms

Example



$$\text{GREEN2} \quad \not\longmapsto \quad \text{BLUE}$$

Finally, the packet reaches its destination

Routing algorithms

The OPTIMIZED LINK STATE ROUTING PROTOCOL (OLSR, `http://en.wikipedia.org/wiki/OLSR`) is an IP routing protocol specified in RFC3626 (`http://www.ietf.org/rfc/rfc3626.txt`) which is optimized for mobile ad-hoc networks.

Routing algorithms

It has been implemented in two daemons, available for linux:

- OLSRD: `http://www.olsr.org/`
- BATMAND: `http://www.open-mesh.net/batman/`

These daemons basically monitor each possible path throughput and reliability, and choose the best path for each packet. The beauty of it it's that, since it works at networking leve l of the OSI Reference Model, it's completely transparent (with respect to the programmer).

Routing algorithms

AODV is but one of the existing algorithms to solve this problem: for example, there's AODV, used in the ZigBee networking protocols suite. That protocol is *reactive*, since it searches network paths only upon request

See `https://www.open-mesh.net/batman/doc/batmand_howto.pdf` for a discussion over those algorithms

The End