D2.1.3-4 The navigation capabilities of the robot platforms for the selected scenario and common API specifications for the robot platforms

Leo Nomdedeu (Universitat Jaume-I)

with contributions from:

Jorge Sales (Universitat Jaume-I), Enric Cervera (Universitat Jaume-I)

Abstract.
EU-IST STREP IST-2006-045269 Guardians
Deliverable D2.1.3-4 (WP2)
The aim of this document is to present an overview of the robot platforms used in the project, focusing on their navigation capabilites, A common Application Programming Interface (API) specification is also presented.
Keyword list: mobile robot, navigation, software specification.

Document Identifier	Guardians/2008/D2.1.3-4/v1.7
Project	Guardians EU-IST-2006-045269
Version	v1.7
Date	February 18, 2008
State	draft
Distribution	public

Guardians Consortium

This document is part of a research project funded by the IST Programme of the Commission of the European Communities as project number IST-2006-045269.

Sheffield Hallam University (SHU) - Coordinator
United Kingdom
Contact person: Jacques Penders
E-mail address: J.Penders@shu.ac.uk

Heinz Nixdorf Institute - University of Paderborn (HNI)
Germany
Contact person: Dr. Ulf Witkowski
E-mail address: witkowski@hni.uni-paderborn.de

TOBB University of Economics and Technology (ETU)
Turkey
Contact person: Dr. Veysel Gazi
E-mail address: vgazi@etu.edu.tr

Institute of Systems and Robotics - University of Coimbra (ISR-UC)
Portugal
Contact person: Dr. Lino Marques
E-mail address: lino@isr.uc.pt

K-Team (K-Team)
Switzerland
Contact person: Pierre Bureau
E-mail address: pierre.bureau@k-team.com

Space Application Services (SAS)
Belgium
Contact person: Jeremi Gancet
E-mail address: guardians@spaceapplications.com

Robotnik Automation (Robotnik)
Spain
Contact person: Roberto Guzman
E-mail address: rguzman@robotnik.es

Universitat Jaume-I de Castelló (UJI)
Spain
Contact person: Enric Cervera
E-mail address: ecervera@icc.uji.es

South Yorkshire Fire and Rescue Service
United Kingdom
Contact person: Neil Baugh

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed to writing parts of this document:

Sheffield Hallam University
Heinz Nixdorf Institute
TOBB University of Economics and Technology
Institute of Systems and Robotics - University of Coimbra
K-Team
Space Application Services
Robotnik Automation
Universitat Jaume-I de Castelló
South Yorkshire Fire and Rescue Service

Changes

Version	Date	Author	Changes
0.0	17.10.07	Enric Cervera	creation
0.1	10.12.07	Leo Nomdedeu	added content: s.2
0.2	11.12.07	Jorge Sales	added content: s.3
0.3	12.12.07	Jorge Sales	updated content: s.3
0.4	12.12.07	Leo Nomdedeu	added and updated content: s.2-3
0.5	13.12.07	Leo Nomdedeu	added and updated content: s.2
0.6	13.12.07	Jorge Sales	added and updated content: s.3 & annex
0.7	13.12.07	Leo Nomdedeu	added and updated content: s.2-3-4 & annex
0.8	13.12.07	Jorge Sales	added content: annex
0.9	14.12.07	Leo Nomdedeu	added and updated content: s.1
1.0	22.01.07	Leo Nomdedeu	added and updated content: s.1-2
1.1	24.01.07	Leo Nomdedeu	added and updated content: s.2
1.2	25.01.07	Leo Nomdedeu	added and updated content: s.2-4
1.3	28.01.07	Leo Nomdedeu	added and updated content: s.2-3
1.4	01.02.07	Leo Nomdedeu	added and updated content: s.3 & annex
1.5	12.02.07	Leo Nomdedeu	added and updated content: s.3
1.6	15.02.07	Leo Nomdedeu	added and updated content: s.3 & annex
1.7	18.02.07	Leo Nomdedeu	added and updated content: s.2 & s.3 & s.4 & annex

Executive Summary

The aim of this document is to present an overview of the robot platforms used in the project, namely the Khepera, Erratic and Rescuer robots. We focus on their navigation capabilites, according to the scenario envisaged (small-scale / full-scale demonstrations). To cope with the diverse platforms, an homogeneous Application Programming Interface (API) specification based on the open-source Player robot software is presented.

Contents

Chapter 1

Overview of mobile robots

1.1 Introduction

Mobile robots have the capability to move around in their environment (not only ground, but also water, air or space) and are not fixed to one physical location. In contrast, industrial robots usually consist of a jointed arm (multi-linked manipulator) and gripper assembly (or end effector) that is attached to a fixed surface.

Mobile robots are the focus of a great deal of current research and almost every major university has one or more labs that focus on mobile robot research. Since the early 70s, much progress has been achieved, and ever-increasing computing performance has boosted their capabilities, thus mobile robots are routinely found nowadays in industry, space exploration, military and security environments. They also emerge as consumer products, for entertainment or to perform certain service tasks like vacuum cleaning or mowing [1].

This and more information regarding Mobile Platforms can be obtained in the previous deliverable 2.1.1 [MMFT07].

1.2 Mobile robot platforms in Guardians

The mobile platforms used by Guardians are composed by:

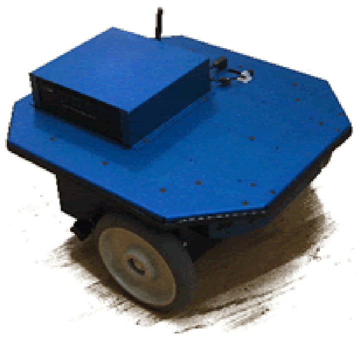
1. small platforms produced by SME partner K-Team, namely the Khepera-III robot.
2. medium-sized commercial platforms, namely Super Scouts, Pioneer, and Erratic platforms, owned by research partners.
3. the Rescuer tracked platform developed by SME partner Robotnik

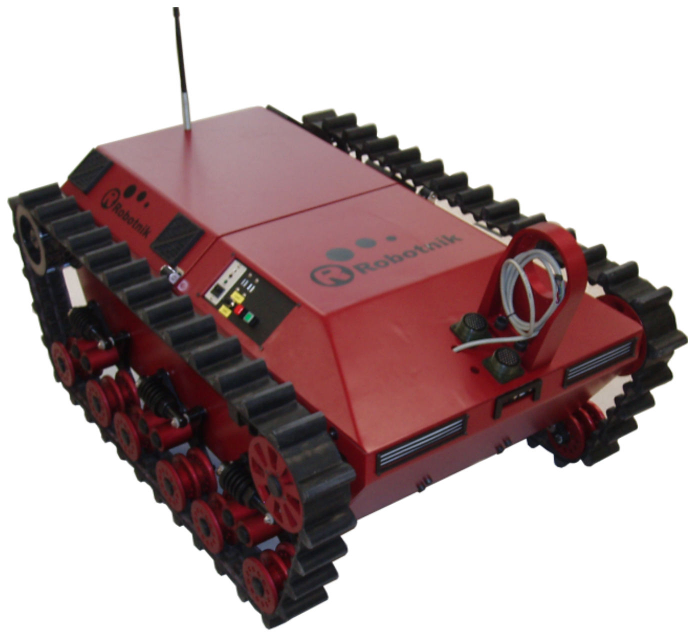
[1] `http://en.wikipedia.org/wiki/Mobile_robot`

Small robots will be used for real experimentation in lower scale models. Solutions will then be implemented in medium sized platforms for testing first in real scale laboratory environments, then in user scenarios.

The need for a common development environment arises from the use of different platforms. Program code developed for the small platforms should be compatible, with minor adjustments, with the bigger size platforms.

(a) Kepera III Mobile platform


(b) Erratic Mobile platform



(c) Rescuer Mobile platform

Figure 1.1: Mobile platforms used

Chapter 2

Specifications of the robot platforms

2.1 Khepera platform

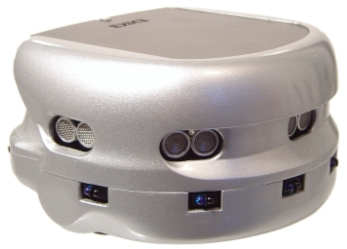


Figure 2.1: Khepera III

The Khepera III platform (Fig. 2.1) is a new system from K-Team. The full specs can be obtained at [1] while the Users manual can be obtained at [2].

Several researches have used Kephera as the mobile real platform for development and testing. A good amount of them are listed in it's manufacturer's web page[3]. We could highlight among them the works done in the field "Collective Robotics" [MM95] [KBK00] [AC07] [AAC07].

[1]`http://www.k-team.com/kteam/index.php?site=1&rub=3&upPage=200&page=197&version=EN`

[2]`http://ftp.k-team.com/Can/KIII/Kh3.Robot.UserManual.1.3.pdf`

[3]`http://www.k-team.com/kteam/index.php?site=1&rub=2&page=117&version=EN`

2.1.1 Navigation capabilities

Among other capabilities, this platform provides the basic navigation capabilities shown in table 2.1. The sensor layout can be appreciated in figures 2.2 and 2.3.

Motion	*2 DC brushed servo motors with incremental encoders (roughly 22 pulses per mm of robot motion)*
Speed	*Max: 0.5 m/s*
Size	*Diameter: 130mm*
	Height: 70mm
Weight	*Approx 690g*
Payload	*Approx 2000g*
Sensors	*9 Infra-red proximity and ambient light sensors with up to 25cm range*
	2 Infra-red ground proximity sensors for line following applications
	5 Ultrasonic sensors with range 20cm to 4 meters

Table 2.1: Khepera III navigation capabilities

2.1.2 Controller / Computer

The Khepera III basic setup comes with a DsPIC but K-Team allows us to buy a more powerful and versatile controller, based on a 400Mhz XScale called KoreBot.

Processor	*DsPIC 30F5011 at 60MHz. 400MHz XScale KoreBot Extension*
RAM	*4Kb on DsPIC, 64Mb on KoreBot Extension*
Flash	*66Kb on DsPIC, 32Mb on KoreBot Extension*

Table 2.2: Khepera III Controller specs

2.1.3 Electronics

The Khepera III comes with enough connectivity methods to allow us a wide range of posibilities.

I/O	*Several IO with KoreIO Extension*
	2 programmable LED
Power	*Power Adapater*
	Swapable Lithium-Polymer battery pack (1400 mAh)
Autonomy	*8 hours, moving continuously, without KoreBot.*
	Additional turrets will reduce battery life.
Comms	*Standard Serial Port, up to 115kbps*
	USB communication with KoreBot
	Wireless Ethernet with KoreBot and WiFi card
Ext. Bus	*Expansion modules can be added to the robot using the KB-250 bus.*

Table 2.3: Khepera III Electronics specs

2.1.4 Software

Among proprietary solutions, K-Team allows us to use free open-source software solutions with its Khepera III robot platform.

Simulator	*WEBOTS, Realistic 3D Simulator and robot programming (Windows & Linux).*
Dev. Env.	*GNU C/C++ compiler, for native on-board applications with KoreBot.*
	Freeware
Remote-	*LabVIEW® (on PC, MAC or SUN) using RS232.*
-Control-	*MATLAB® (on PC, MAC, Linux or SUN) using RS232.*
-Software	*SysQuake® (on PC, MAC, Linux or SUN) using RS232.*
	Freeware.
	Any other software capable of RS232 communication

Table 2.4: Khepera III Software capabilities

2.1.5 Accessories

K-Team provides a number of accessories that we could use to enhance the Khepera III basic capabilities.

TODO	

Table 2.5: Khepera III Available accessories

2.1.6 Tech Views

Here we have some technical views of this Khepera III platform.

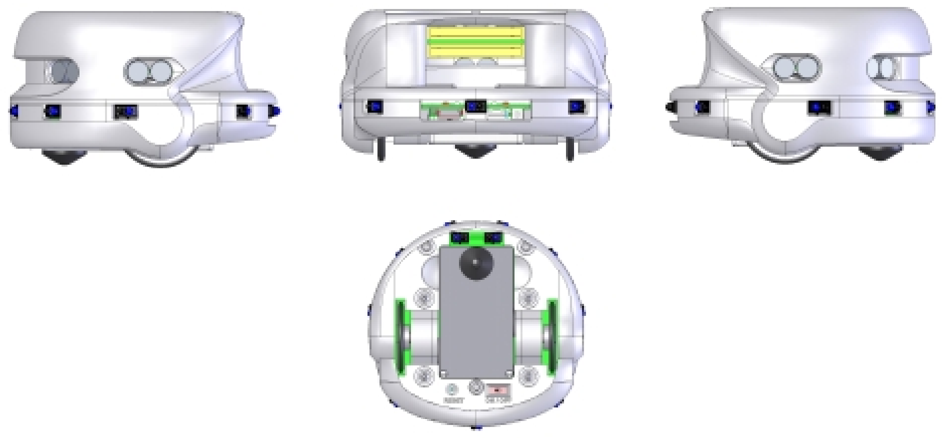


Figure 2.2: Khepera III technical views

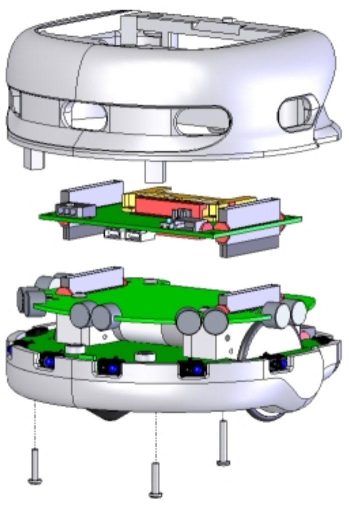


Figure 2.3: Khepera III insides

Despite its slow speed, the Khepera III platform is very usefull in or project for small scale testing. Obviously is not the end platform that will follow the fire-fighters into the flames.

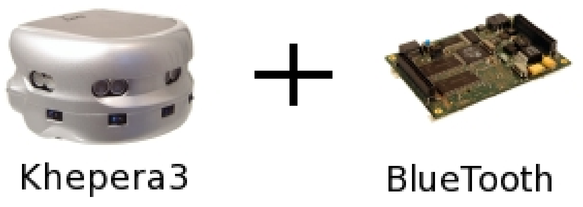
2.1.7 Configurations

The Khepera III is a very versatile platform which can be configured for many different experiments. This page is a summary of the most common configurations, but many others are possible to match exactly your requirement. Please send us an email for support to configure your robot.

Remote Control Operation

The Khepera III can be completely remote controlled from a Personal Computer using a serial communication protocol. The robot can be interfaced using a standard serial cable and a KoreConnect module, but it also supports BlueTooth wireless connection using a BlueTooth radio communication extension.

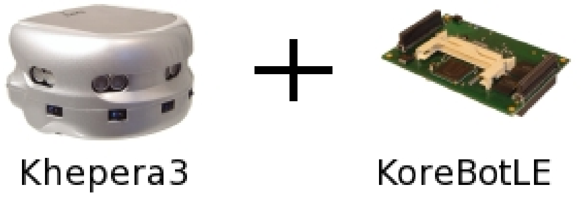
With such a configuration, no programming is required on the robot side. The robot is fully controlled from a remote operation program running on a PC. The KoreBot is unnecessary and not useful for this configuration, as the computing power and complex calculations are hosted on the PC.



Autonomous Operation

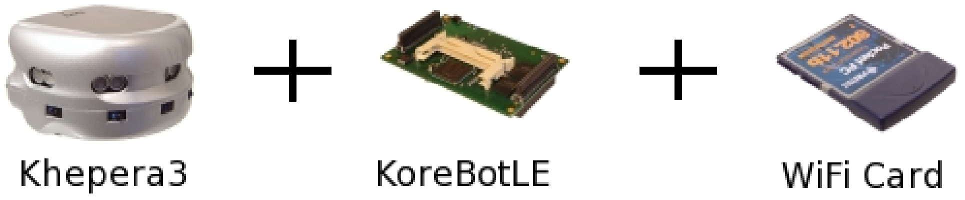
Combined with the KoreBot, the Khepera III provides a complete embedded Linux environment for autonomous robotic application development. This configuration is a powerful solution for fully autonomous algorithms, with all the calculations embedded on the robot.

Programming on the robot is performed using the KoreBot tools, using the GNU C/C++ cross-compiler for development and the libKoreBot as a base API for all applications.



Autonomous Operation with WiFi

The Khepera III is also able to include Wireless Ethernet network communication. This configuration is a perfect solution for applications requiring communication between two robots, or between robots and a Computer. Calculations for algorithm can be fully embedded, hosted on a remote PC, or a combination of the two options.



Such configuration is the most advanced solution for collective robotics experiments and swarm robotics projects. It provides high embedded computing power and high speed network communication between robots with all the features of TCP/IP networking.

2.2 Erratic platform

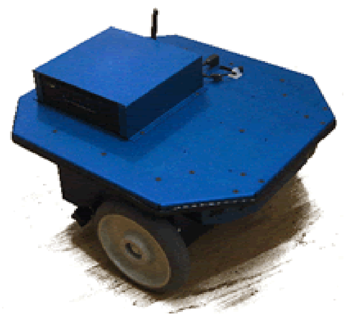


Figure 2.4: Erratic

The Erratic platform (Fig. 2.4) from Videredesign[4] is a middle size platform for educational an research proposes. It is customizable since it's controlled by a standard small-size PC.

The full specs can be obtained at [5] while the Users manual can be obtained at [6].

Our Erratic platform provides the navigation capabilities shown in table 2.6. The sensor capabilities are also explained along with it's specs. Images of the entirely platform setup can be seen in figure 2.6.

Although this is not the final platform that will follow the fire-fighters into the flames, it let us make real scale tests in real size scenarios.

2.2.1 Navigation capabilities

Among other capabilities, this platform provides the basic navigation capabilities shown in table 2.6.

[4] `http://www.videredesign.com`
[5] `http://www.videredesign.com/robots/era_mobi.htm`
[6] `http://www.videredesign.com/docs/ERA-Rev-E-manual.pdf`

`Motion`	*Differential, single rear caster*
	DC reversible with gearhead. 72 W continuous power
	Wheels: 12.5 cm diameter (driven), 6.25 cm diameter (caster)
`Speed`	*Max: 2.0 m/s*
`Size`	*Long: 400mm*
	Width: 410mm
	Height: 150mm
`Weight`	*Approx 12900g (base, batteries, computer, sonar)*
`Payload`	*Approx 20000g*
`Sensors`	*2 Infra-red Sharp GP2D15 ground proximity sensors*
	Precision optical encoders with 500 counts per motor revolution
	Hokuyo URG Laser range finder with range up to 4m, and 240 degrees field of view (Optional)
	8 MaxSonar EZ1 Ultrasonic sensors with maximum range of 6m (Optional)
	Other optional devices

Table 2.6: Erratic navigation capabilities

2.2.2 Controller / Computer

Erratic platforms comes with a 16bit microcontroller connected via USB-RS232 to the main processing unit (a small PC on top of it). VidereDesign allows us to upgrade this onboard PC to a more powerful one in its website.

`Controller`	*16 bit microcontroller. Integrated controller / motor driver*
`Processor`	*1.6 GHz Celeron M*
`RAM`	*512 MB memory*
`Hard Drive`	*40 GB hard drive*

Table 2.7: Erratic Controller specs

2.2.3 Electronics

As everybody can imagine, with an onboard PC the connection capabilities are endless.

`I/O`	*N/A*
`Power`	*12V, 7AH lead-acid batteries (x3)*
	5V, 12V, 19V power bus for peripherals
	5A charger
`Autonomy`	*>10 hours, PC only*
	4-5 hours with nominal movement
`Comms`	*FireWire, USB, wireless (802.11 b/g) and wired ethernet*
`Ext. Bus`	*N/A*

Table 2.8: Erratic Electronics specs

2.2.4 Software

This platform can be controlled and simulated with Player/Stage. In fact, P/S is the preffered middleware.

`Simulator`	*Player/Stage/Gazebo, Realistic 2D/3D Simulator and robot programming middleware (Linux).*
`Dev. Env.`	*Any IDE for C, C++, Java, Python, etc.*
`Remote-`	*Linux OS, Player/Stage installed*
`-Control-`	*Any other software capable of RS232 communication*
`-Software`	

Table 2.9: Erratic Software capabilities

2.2.5 Accessories

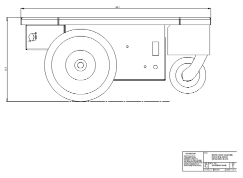
VidereDesign provides a number of accessories that we could buy from its website to get a more versatile platform.

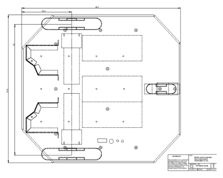
`TODO`	

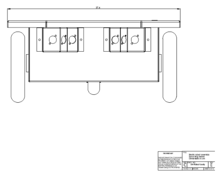
Table 2.10: Erratic Available accessories

2.2.6 Tech Views

Here we have some technical views of this Erratic platform.


(a)


(b)


(c)

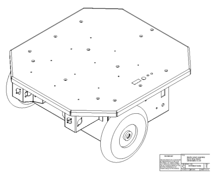
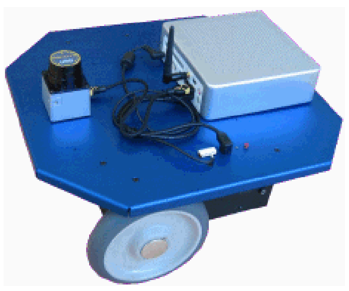
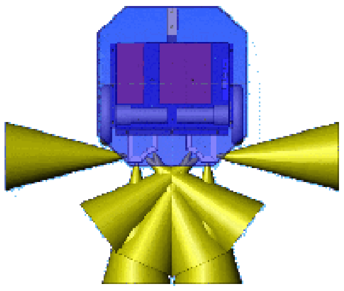

(d)

Figure 2.5: Erratic technical views

2.2.7 Configurations


(a) Mobile platform Erratic: H8S controller, Servo and analog/digital I/O, 3x 7A 12V batteries. Onboard PC: 1.4 GHz Celeron, 802.11b/g, USB 2.0, IEEE 1394b, 40 GB, 512 MB


(b) Sonar ring

(c) XSens MTi IMU device

(d) Hokuyo URG Laser Rangefinder: Power 2.5W, Weight 0.16 Kg

Figure 2.6: Mobile platform and sensors used

2.3 Rescuer platform

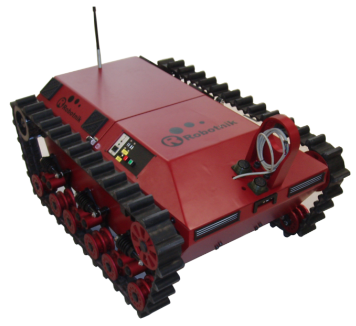


Figure 2.7: Rescuer

The Rescuer platform (Fig. 2.7) from Robotnik[7] is a big size platform for research, civil protection, and safety application proposes.

The product webpage is located at [8] while the specs can be found at [9].

2.3.1 Navigation capabilities

Among other capabilities, this platform provides the basic navigation capabilities shown in table 2.11.

Motion	*Belts with suspension system*
	Climbing stairs capability
	Different belt models on request
	2 axis, skid configuration motors
	2 x 750W
Speed	*Max: 1.25 m/s*
Size	*Long: 1100mm*
	Width: 780mm
	Height: 600mm
Weight	*Approx 250.000g*
Payload	*Approx 200.000g*
Sensors	*N/A*

Table 2.11: Rescuer navigation capabilities

[7]http://www.robotnik.es

[8]http://www.robotnik.es/automation/productos/agvs/robotnik-p01-s.html

[9]http://www.robotnik.es/automation/pdf/ROBOTNIK-Robot%20Movil%20Rescuer.pdf

2.3.2 Controller / Computer

The platform has a powerful enough onboard PC to carry out almost every dessired task.

Controller	*N/A*
Processor	*AMD Athlon K8*
RAM	*N/A*
Hard Drive	*HD Flash 2Gb*

Table 2.12: Rescuer Controller specs

2.3.3 Electronics

AS it hasn a powerful onboard PC the capabilities are endless.

I/O	*N/A*
Power	*12V, 100AH (x2)*
Autonomy	*8 hours in normal operation*
Comms	*2 serial, 8 USB 2.0/1.1, onboard RTL8100C wired ethernet (10/100)*
	RF WiFi/WiMan
Ext. Bus	*Distributed internal CAN bus net*

Table 2.13: Rescuer Electronics specs

2.3.4 Software

The system can be controlled via Player/Stage. The drivers are under developement nowadays.

Simulator	*Player/Stage/Gazebo, Realistic 2D/3D Simulator and robot programming middleware (Linux).*
Dev. Env.	*Any IDE for C, C++, Java, Python, etc.*
Remote-	*LinuxRT OS installed*
-Control-	
-Software	

Table 2.14: Rescuer Software capabilities

2.3.5 Accessories

We do not have very much information on this platform yet as it's under active development nowadays. We hope to be able to provide this information in future releases.

N/A

2.3.6 Tech Views

N/A

2.3.7 Configurations

N/A

2.4 Platform comparison

Here we present a brief navigation capabilities comparison of the three platforms shown above.

	Khepera III	*Erratic*	*Rescuer*
`Motion`	*2 DC brushed servo motors with incremental encoders (roughly 22 pulses per mm of robot motion)*	*Differential, single rear caster. DC reversible with gearhead. 72 W continuous power. Wheels: 12.5 cm diameter (driven), 6.25 cm diameter (caster)*	*Belts with suspension system. Climbing stairs capability. Different belt models on request. 2 axis, skid configuration motors. 2 x 750W*
`Speed`	*Max: 0.5 m/s*	*Max: 2.0 m/s*	*Max: 1.25 m/s*
`Size`	*Diameter: 130mm*	*Long: 400mm*	*Long: 1100mm*
		Width: 410mm	*Width: 780mm*
	Height: 70mm	*Height: 150mm*	*Height: 600mm*
`Weight`	*Approx 690g*	*Approx 12900g (base, batteries, computer, sonar)*	*Approx 250.000g*
`Payload`	*Approx 2000g*	*Approx 20000g*	*Approx 200.000g*

Table 2.15: Navigation capabilities comparison

Chapter 3

Software interface for navigation

As mentioned in the previous deliverable [CN07], the Player / Stage / Gazebo platform has been chosen for simulation and robot development in Guardians. In the following sections it is explained how Player interacts with a robotic platform and how a client application communicates with Player in order to control a robot.

3.1 The Player server

Player is a network server for robot control. Running on a robot, Player provides a clean and simple interface to the robot's sensors and actuators over the IP network. A client program talks to Player over a TCP socket, reading data from sensors, writing commmands to actuators, and configuring devices on the fly.

Player supports a variety of robot hardware. The original Player platform is the ActivMedia Pioneer 2 family, but several other robots (such as the Khepera and Erratic platforms used in Guardians) and many common sensors are supported. Player's modular architecture makes it easy to add support for new hardware (like in the case of the Rescuer platform), and an active user/developer community contributes new drivers.

Player runs on Linux (PC and embedded), Solaris and *BSD, but it is also designed to be language and platform independent. This means that the client program can run on any machine that has a network connection to the robot, and it can be written in any language that supports TCP sockets.

The Player project provides client-side utilities for C++, Tcl, Java, and Python languages. Further, Player makes no assumptions about how the programmer might want to structure the robot control programs. In this way, it is much more 'minimal' than other robot interfaces. In this way, the robot control program can be either a highly concurrent multi-threaded program, or a simple read-think-act loop.

Player allows multiple devices to present the same interface. For example the Khepera

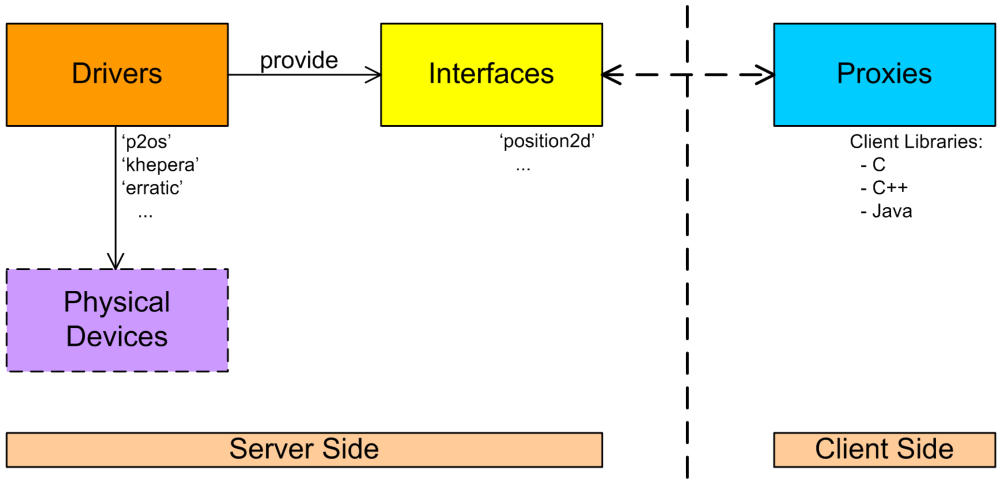


Figure 3.1: Player architecture: Drivers, Interfaces and Proxies.

and Erratic drivers both use Player's *position* interface to allow control of the robot's movement. Thus the same control code could drive both kinds of robot.

Player is also designed to support virtually any number of clients. Any client can connect to and read sensor data from (and even write motor commands to) any instance of Player on any robot. Aside from distributed sensing for control, Player can be used for monitoring of experiments. For example, while a C++ client controls a robot, a graphical visualization tool can be running elsewhere in order to show the current sensor data and a logger program can be monitoring data for later analysis. On-the-fly device requests allow client programs to gain access to differenct sensors and actuators as needed for the task at hand.

3.2 Player as a device interface

Player is a *robot device interface*, i.e., a *hardware abstraction layer* (HAL) for robotic devices, and act as an Operating System for the robot. Player defines a set of standard interfaces, each of which is a specification of the ways that you can interact with some class of devices (see Figure 3.1). For example the *position2d* interface covers ground-based mobile robots, allowing them to accept commands to make them move (either velocity or position targets) and to report their state (current velocity and position). Many drivers support the *position2d* interface, including *p2os*, *obot*, and *rflex*, each of which controls a different kind of robot. The job of the driver is to make the robot support the standard interface. This way, Player control code that works with one robot will work (within reason) on another robot.

3.3 Player interfaces

All Player communication occurs through interfaces, which specify the syntax and semantics for a set of messages. An interface specifies of how to interact with a certain class of robotic sensor, actuator, or algorithm. The interface defines the syntax and semantics of all messages that can be exchanged with entities in the same class. For each interface, the following is given:

- Relevant constants (size limits, etc.)
- Message subtypes:
  - Data subtypes : codes for each kind of data message defined by the interface.
  - Command subtypes : codes for each kind of command message define by the interfce.
  - Request/reply subtypes: codes for each kind of request/reply message defined by the interface. Also specified are the interaction semantics, such as when to send a null request and when to expect a null response. A 'null' request or response is a zero-length message.
- Utility structures : structures that appear inside messages.
- Message structures:
  - Data message structures : data messages that can be published via this interface.
  - Command message structures : command messages that can be sent to this interface.
  - Request/reply message structures : request messages that can be sent to this interface and reply messages that can be expected from it.

It can be the case that a given message can be sent as data or in response to a request. A common example is geometry. For many devices, geometry is fixed and so need only be requested once. For others geometry may change dynamically and so the device will publish it periodically.

3.4 The Position2d interface

This interface offers a simple yet powerful way of controlling the 2D robot motion. It lets you set the initial Pose and the robot geometry, the desired speed and turn rate, and even a desired destination pose to reach and let the underlying driver manage to get to the goal. Obviously it let you query any of this information at any given time.

`playerc_client_t * playerc_client_create (playerc_mclient_t *mclient, const char *host, int port)`	*Create a client object.*
`void playerc_client_destroy (playerc_client_t *client)`	*Destroy a client object.*
`int playerc_client_connect (playerc_client_t *client)`	*Connect to the server.*
`int playerc_client_disconnect (playerc_client_t *client)`	*Disconnect from the server.*
`void * playerc_client_read (playerc_client_t *client)`	*Read data from the server (blocking).*

Table 3.1: Most important functions contained in the C client API

3.5 Controlling a mobile robot using a client library

Libraries are available in various languages (C, C++, Java, etc.) to facilitate the development of TCP client programs. Any of these libraries can be used to develop your a robot control program.

Client libraries are based on a *device proxy model* (see Figure 3.1), in which the client maintains a local *proxy* for each of the devices on the remote server. Thus, for example, one can create local *proxies* for the position and laser devices. There is also a special client *proxy*, used to control the Player server itself.

3.5.1 C API

Libplayerc is a client library for the player server. The client `_playerc_client_t` data structure maintains the data connection with the Player server; it is responsible for reading new data, setting data transmission modes and so on. The client object must be created and connected before device proxies are initialized. Through the `playerc_client_create` and `playerc_client_connect` functions the client can connect to the server. The most important functions contained in the client API can be seen in Table 3.1.

Once the client is connected to the Player server, the `position2d` proxy provides an interface to a mobile robot allowing to control the respective *position2d* device. The most important functions to handle this proxy can be seen in Table 3.2. The `playerc_position2d_create` function creates de proxy and the `playerc_position2d_subscribe` function subscribes it to the server.

In Section 3.7.1, a C client library sample code is presented. This simple client program connects to the server and makes simple robot movements and sensor readings. The `playerc_position2d_enable` function enables the robot motors. Then, the `playerc_position2d_set_cmd_vel` function sets the desired robot speed.

`playerc_position2d_create (playerc_client_t *client, int index)`	*Create a position2d device proxy.*
`playerc_position2d_destroy (playerc_position2d_t *device)`	*Destroy a position2d device proxy.*
`playerc_position2d_subscribe (playerc_position2d_t *device, int access)`	*Subscribe to the position2d device.*
`playerc_position2d_unsubscribe (playerc_position2d_t *device)`	*Un-subscribe from the position2d device.*
`playerc_position2d_enable (playerc_position2d_t *device, int enable)`	*Enable/disable the motors.*
`playerc_position2d_get_geom (playerc_position2d_t *device)`	*Get the position2d geometry.*
`playerc_position2d_set_cmd_vel (playerc_position2d_t *device, double vx, double vy, double va, int state)`	*Set the target speed.*
`playerc_position2d_set_cmd_pose (playerc_position2d_t *device, double gx, double gy, double ga, int state)`	*Set the target pose (gx, gy, ga) in the odometric coordinate system.*
`playerc_position2d_set_odom (playerc_position2d_t *device, double ox, double oy, double oa)`	*Set the odometry offset.*

Table 3.2: Most important functions to handle `position2d` proxy in the C client API

After that, the `playerc_client_read` function reads multiple data from the server and stores the information in the client `_playerc_client_t` data structure, displaying it on screen.

3.5.2 C++ API

The C++ client is generally the most comprehensive library, since it is used by Player developers to test new features as they are implemented in the server. It is also the most widely used client library and thus the best debugged by the maintainers of the Player project.

This library has two kinds of proxies: the special server proxy `PlayerClient` and the various device-specific proxies. Each kind of proxy is implemented as a separate class. The user first creates a `PlayerClient` proxy and uses it to establish a connection to a Player server. Next, the proxies of the appropriate device-specific types are created and initialized using the existing `PlayerClient` proxy, such as the `Position2DProxy` proxy to control the robot's motors.

The `PlayerClient` object is used to control each connection to a Player server.

`PlayerClient (const std::string aHostname=PLAYER_HOSTNAME,`
`              uint aPort=PLAYER_PORTNUM,`
`              int transport=PLAYERC_TRANSPORT_TCP)`
Make a client and connect it as indicated.

`˜PlayerClient ()` *destructor*
`Read ()` *A blocking Read.*
`StartThread ()` *Start the run thread.*
`StopThread ()` *Stop the run thread.*
`RequestDeviceList ()` *Get the list of available device ids.*

Table 3.3: Most important PlayerClient class methods of the C++ client API

`Position2dProxy (PlayerClient *aPc, uint aIndex=0)` *constructor*
`˜Position2dProxy ()` *destructor*
`SetSpeed (double aXSpeed, double aYSpeed, double aYawSpeed)` *Send a motor command for velocity control mode.*
`GoTo (player_pose_t pos, player_pose_t vel)` *Send a motor command for position control mode.*
`RequestGeom ()` *Get the device's geometry; it is read into the relevant class attributes.*

Table 3.4: Most important methods to handle `Position2dProxy` proxy in the C++ client API

Contained within this object are methods for changing the connection parameters and obtaining access to devices (see Table 3.3).

The `Position2dProxy` class is used to control a *position2d* device. The most important public member methods can be seen in Table 3.4.

In Section 3.7.2, a C++ client library sample code is presented. This simple client program does the same as the sample code in Section 3.7.1.

3.5.3 Java API

The Java client library implementation is called *JavaClient*. Javaclient allows development of applications for Player/Stage using the Java programming language. The client

`PlayerClient(java.lang.String serverName, int portNumber)`	*The PlayerClient constructor.*
`close()`	*The PlayerClient 'destructor'.*
`requestInterface{InterfaceName} (int index, int access)`	*Request the InterfaceName device.*
`runThreaded(long millis, int nanos)`	*Start a threaded copy of Javaclient.*
`setNotThreaded()`	*Change the mode Javaclient runs to non-threaded.*
`requestData()`	*Configuration request: Get data.*
`readAll()`	*Read the Player server replies in non-threaded mode.*

Table 3.5: Most important PlayerClient class methods of the Java client API

implements all interfaces described in the Player manual, plus several various additions.

`PlayerClient` is the main Javaclient class. It contains methods for interacting with the player device (see Table 3.5). The player device represents the server itself, and is used in configuring the behavior of the server. There is only one such device (with index 0) and it is always open. Once the constructor is called, it will create a socket with the Player server running on a specified host and port number.

The `Position2DInterface` class is used to control a mobile robot in 2D. The most important public member methods can be seen in Table 3.6.

In Section 3.7.3, a Java client library sample code is presented. This simple client program does the same as the sample code in Section 3.7.1.

The `requestInterfacePosition2D` and `requestInterfaceSonar` lets the client program to manage the robot movements and read sonar values. The `runThreaded` method starts a threaded copy of Javaclient. After that, a read-compute-act cycle is started. If there is data available from sonar devices (`isDataReady` method), get data from them (`getData` method) and calculate new translational and rotational velocities for the robot, based on the sonar sensor values. With the `setSpeed` method of the `Position2DInterface`, new velocities are commanded to the robot.

`Position2DInterface(PlayerClient pc)` *Constructor for Position2DInterface.*
`setPosition(PlayerPose pos, int state)` *New position for the robot's motors.*
`setSpeed(float speed, float turnrate)` *New position for the robot's motors.*
`queryGeometry()` *Request/reply: Query geometry.*
`isDataReady()` *Check if data is available.*
`isGeomReady()` *Check if geometry data is available.*
`getData()` *Get the Position2D data.*
`getGeom()` *Get the geometry data.*

Table 3.6: Most important methods to handle `Position2DInterface` proxy in the Java client API

3.5.4 API comparison

Here we present a brief API comparison between the three presented programing languages.

	C	*C++*	*Java*
`Contructors`	*playerc_client_t * playerc_client_create (playerc_mclient_t *mclient, const char *host, int port) int playerc_client_connect (playerc_client_t *client)*	*PlayerClient (const std::string aHostname=PLAYER_HOSTNAME, uint aPort=PLAYER_PORTNUM, int transport=PLAYERC_TRANSPORT_TCP)*	*PlayerClient(java.lang.String serverName, int portNumber)*
`Destructors`	*int playerc_client_disconnect (playerc_client_t *client) void playerc_client_destroy (playerc_client_t *client)*	*PlayerClient ()*	*close()*
`Data read`	*void * playerc_client_read (playerc_client_t *client)*	*Read () //If you want to control every read operation StartThread () //If you want a thread for data reads to be launched*	*readAll() //If you want to control every read operation runThreaded(long millis, int nanos) //If you want a thread for data reads to be launched*
`Proxy`	*playerc_position2d_create (playerc_client_t *client, int index) playerc_position2d_destroy (playerc_position2d_t *device) playerc_position2d_subscribe (playerc_position2d_t *device, int access) playerc_position2d_unsubscribe (playerc_position2d_t *device)*	*Position2dProxy (PlayerClient *aPc, uint aIndex=0) Position2dProxy ()*	*Position2DInterface(PlayerClient pc) // Or playerClient.requestInterface[InterfaceName](int index, int access)*
`Speed motion`	*playerc_position2d_set_cmd_vel (playerc_position2d_t *device, double vx, double vy, double va, int state)*	*SetSpeed (double aXSpeed, double aYSpeed, double aYawSpeed)*	*setSpeed(float speed, float turnrate)*
`Goal motion`	*playerc_position2d_set_cmd_pose (playerc_position2d_t *device, double gx, double gy, double ga, int state)*	*GoTo (player*$_{p}ose_{t}pos, player_{p}ose_{t}vel$)	*setPosition(PlayerPose pos, int state)*

Table 3.7: API comparison

3.6 Example configuration

In this section we presetn an example configuration file for each robot. This configuration files specifies which interfaces they will provide.

For a Stage simulator configuration file, please refer to Deliverable D2.3.1 **??**.

3.6.1 Khepera III

In this section an example configuration file for the Khepera robot is presented. In this example, `position2d`, `sonar` and `ir` (infrared) interfaces are provided:

```
driver
(
  name "khepera"
  provides ["position2d:0" "sonar:0" "ir:0"]
)
```

3.6.2 Erratic

In this section an example configuration file for the Erratic robot is presented. In this example, `position2d`, `power`, `aio` (analogic input/output), `ir` (infrared) and `sonar` interfaces are provided:

```
driver
(
name "erratic"
    provides [ "position2d:0"
             "power:0"
             "aio:0"
             "ir:0"
             "sonar:0"
port "/dev/erratic"
)
```

3.6.3 Rescuer

In this section an example of configuration file for the Rescuer robot is presented. In this example, `position2d` and `sonar` interfaces are provided:

```
driver
(
  name "rescuer"
  provides ["position2d:0" "sonar:0" ]
)
```

As the development of this driver is still an ongoing work, the complete list of features is not available at the moment.

3.7 Example client application

Here we present the very same application written in the three presented languages using the different APIs.

This application does really nothing but connecting to a robot in host “localhost” and port 6665, retreaving the position2d proxy, sending a simple motion command and quering and printing 200 times the robot pose. After this it disconnects from the robot.

3.7.1 C client

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

int main(int argc, const char **argv)
{
  int i;
  playerc_client_t *client;
  playerc_position2d_t *position2d;

  // Create a client object and connect to the server; the server must
  // be running on "localhost" at port 6665
  client = playerc_client_create(NULL, "localhost", 6665);
  if (playerc_client_connect(client) != 0)
  {
    fprintf(stderr, "error: %s\n", playerc_error_str());
    return -1;
  }

  // Create a position2d proxy (device id "position2d:0") and susbscribe
  // in read/write mode
  position2d = playerc_position2d_create(client, 0);
  if (playerc_position2d_subscribe(position2d, PLAYERC_OPEN_MODE) != 0)
  {
    fprintf(stderr, "error: %s\n", playerc_error_str());
    return -1;
  }

  // Enable the robots motors
  playerc_position2d_enable(position2d, 1);

  // Start the robot turning slowing
  playerc_position2d_set_cmd_vel(position2d, 0, 0, 0.1, 1);

  for (i = 0; i < 200; i++)
  {
    // Read data from the server and display current robot position
    playerc_client_read(client);
    printf("position : %f %f %f\n",
           position2d->px, position2d->py, position2d->pa);
  }

  // Shutdown and tidy up
  playerc_position2d_unsubscribe(position2d);
  playerc_position2d_destroy(position2d);
  playerc_client_disconnect(client);
  playerc_client_destroy(client);

  return 0;
}
```

3.7.2 C++ client

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

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

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

  // Enable the robots motors
  pp.SetMotorEnable(TRUE);

  // Start the robot turning slowing
  pp.SetSpeed(0, 0, 0.1);

  for (i = 0; i < 200; i++)
  {
    // read from the proxies
    robot.Read();
    std:cout << "position : " << pp.GetXPos() << " " << pp.GetYPos() <<
      " " << pp.GetYaw() << std:endl;
  }
}
```

3.7.3 Java client

```
import javaclient2.PlayerClient;
import javaclient2.Position2DInterface;
import javaclient2.structures.PlayerConstants;

public class SimpleTest {

   public static void main (String[] args) {
      PlayerClient        robot = null;
      Position2DInterface posi  = null;

      // Connect to the Player server and request access to Position and Sonar
      robot  = new PlayerClient ("localhost", 6665);
      posi = robot.requestInterfacePosition2D (0, PlayerConstants.PLAYER_OPEN_MODE);

      robot.runThreaded (-1, -1);

      // Enable the robots motors
      posi.setMotorPower(1);

      // Start the robot turning slowing
      posi.setSpeed(0, 0.1);

      for (i = 0; i < 200; i++)
      {
        // read from the proxies
        robot.Read();
        System.out.println("position : " + posi.getXPos() + " " +
                           posi.getYPos() + " " + posi.getYaw() + "\n";
      }
```

```
    }
}
```

Chapter 4

Conclusion and future work

This document presents the three platforms involved in the GUARDIANS project. Each one will be used in its own scale environment and will provide valuable information for each step in the process.

In Section 1 we have presented an overview of mobile platforms, a basic the state of the art and some ideas of the uses of this platforms for tasks as the one we are involved in.

Section 2 describes the three actual platforms we plan to use in our project. The specifications of each of them as well as a basic comparison of their navigation capabilities are presented.

Section 3 talks about the software layer. Which interfaces from Player / Stage we are going to use, and how they will work. In addition to this Sections from 3.7.1 to 3.7.3 present a basic application in C, C++, and Java, while Section 3.6 draws basic Player / Stage configuration files in order to let the reader easily set up a testing environment, regardless if it is real or simulated.

Although we will not come up with a final platform, this three platforms will let us know, at each step, if we are in the right way. They will let us test each of the several pieces of the development at the right time, avoiding for example field tests in the first steps of the development. We hope it to speed up the development process.

Bibliography

[AAC07] G. Antonelli, F. Arrichiello, and S. Chiaverini. The null-space-based behavioral control for autonomous robotic systems. *International Journal of Intelligent Service Robotics*, 2007.

[AC07] G. Antonelli and S. Chiaverini. Linear estimation of the physical odometric parameters for differential-drive mobile robots. *Autonomous Robots*, 2007.

[CN07] Enric Cervera and Leo Nomdedeu. Simulated environment for navigation tasks. *Work Package 2. Deliverable 2.3.1*, 2007.

[KBK00] M. J. B. Krieger, J.B. Billeter, and L. Keller. Ant-like task allocation and recruitment in co-operative robots. *Nature*, 406:992–995, 2000.

[MM95] A. Martinoli and F. Mondada. Collective and cooperative group behaviours: Biologically inspired experiments in robotics. In O. Khatib and J. K. Salisbury, editors, *Proceedings of the Fourth International Symposium on Experimental Robotics ISER-95*, pages 3–10. Springer Verlag, June 1995.

[MMFT07] Lino Marques, Ali Marjovi, José Francisco, and Mohmoud Tavakoli. Compiled list of recommended sensors and sensor-carriers (robots). *Work Package 2. Deliverable 2.1.1*, 2007.