Entorno de desarrollo de robots Player/Stage/Gazebo

Gonzalo F. Perez Paina

Centro de Investigación en Informática para la Ingeniería
Univiversidad Tecnológica Nacional
Facultad Regional Córdoba

Octubre 2009

Índice

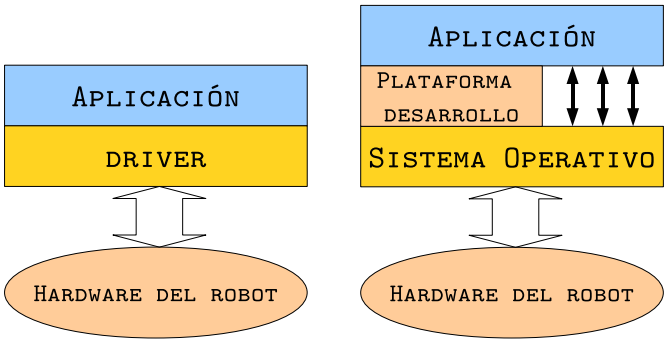
1. Entornos de desarrollo de robots (RDE) 2
1.1. Programación de robots . . . 2
1.2. Sistemas operativos y lenguajes de programación . . . 3
1.3. Entornos de desarrollos de robots . . . 3

2. RDE Player/Stage/Gazebo 4
2.1. Interfaz, Drivers y Dispositivos . . . 5
2.2. Drivers de Player . . . 7
2.3. Cliente Player . . . 7
2.3.1. Ejemplo de programa cliente en C++ . . . 8
2.4. Herramientas de Player . . . 9
2.5. Flexibilidad de Player . . . 9

3. Servidor Player 9
3.1. Archivos de configuración . . . 11
3.2. Dirección de dispositivo . . . 13
3.3. Player con Stage . . . 13
3.4. Player con Gazebo . . . 14

1. Entornos de desarrollo de robots (RDE)

La forma de programar los robots ha ido cambiado con el paso del tiempo. Anteriormente los desarrollos de robots era únicos por lo que su programación consistía en generar la aplicación específica utilizando directamente el acceso al hardware mediante los *drivers* de los dispositivos tanto para leer información de los sensores como para escribir comandos a los actuadores del robot, por lo que las "librerías de programación" eran las mínimas necesarias para acceder a este conjunto de drivers, figura 1(a). Con el surgimiento de diferentes fabricantes de robot y el trabajo cada vez mayor de los centros de investigación en robótica han ido apareciendo *plataformas de desarrollo* que simplifican la programación de la aplicación de control, lo que permite acceder a los dispositivos robóticos de una forma mas abstracta facilitando el desarrollo de aplicaciones cada vez mas complejas incrementando la funcionalidad del robot, figura 1(b).



(a) sobre drivers específicos de sensores y actuadores

(b) sobre una plataforma de desarrollo

Figura 1: Programación de robots

1.1. Programación de robots

El desarrollo de programas de robots es muy diferente al de las demás aplicaciones de software, puesto que los robots son sistemas complejos y además no es sencillo programar la actitud que debe tener el robot ante diversas situaciones, algunas consideraciones importantes en el desarrollo de programas para robots son

- Los programas de los robots están conectados directamente a la realidad física a través de sensores y actuadores, los sensores obtienen información de esta realidad y los actuadores la modifica; las consideraciones importantes aquí son la toma de decisión, respuesta en tiempo real, etc.
- Los robots deben responder a varias fuentes de actividad y objetivos a la vez, como el sensado del entorno, comandos a los actuadores, etc.
- Los programas de robots se enfrentan cada vez mas a una mayor heterogeneidad respecto al hardware utilizado
- Comprensión limitada de la forma de generar y descomponer el comportamiento del robot ante diversas situaciones

Además, las aplicaciones de robot necesitan ser distribuidas para mayor flexibilidad, y de una interfaz gráfica de usuario (GUI) para facilitar la depuración de los programas.

Una herramienta de principal importancia en el desarrollo de aplicaciones robóticas son los *simuladores*, permitiendo depurar los algoritmos en un entorno virtual controlado que simule las observaciones de los sensores y las ordenes a los actuadores. Los simuladores actuales son capaces de simular sensores tan complejos como la visión, en un entorno de dos dimensiones o aun mas realistas en tres dimensiones con comportamiento dinámico de los elementos del entorno. Dos de los simuladores mas utilizados son `SRIsim (MobileSim)` para los robots ActivMedia y los simuladores `Stage` y `Gazebo` de software libre.

1.2. Sistemas operativos y lenguajes de programación

Aunque ha habido intentos de generar sistemas operativos (SO) específicos para aplicaciones en robótica, no resultaron de gran aceptación lo que esta llevando cada vez mas a utilizar SO de propósito generales como MS-Windows o GNU/Linux, los cuales deben disponer de herramientas de desarrollo como compilador, editor, depurador, etc. Además, de posibilidad de programación multitarea como por ejemplo los hilos POSIX *Pthreads*, y medios de comunicación. El mecanismo de comunicación más utilizado en GNU/Linux son los *sockets* lo que brinda una abstracción de los detalles de la red, protocolo y acceso al medio, pudiendo funcionar con el protocolo IP para el nivel de red, y los protocolos TCP o UDP para el nivel de transporte.

En cuanto a los lenguajes de programación también han existido intentos de establecer lenguajes específicos para programar robots, como *Task Description Language* (TDL) o *Reactive Action Packages* (RAP) [1], pero no han sido nunca de uso general. El principal objetivo de estos lenguajes es incluir en la sintaxis mecanismos que resulten ventajosos para la programación de robots.

La creciente utilización de computadoras personales como procesador principal de robots ha obligado a utilizar lenguajes de programación estándares de alto nivel como C, C++, Java, Python, etc. Además, los entornos de desarrollos de robots obligan a que las aplicaciones se escriban en el lenguaje que han sido programadas. Sin embargo, cada vez hay mas plataformas de desarrollo de robots independientes del lenguaje, por ejemplo el entorno `Player/Stage/Gazebo` brinda su funcionalidad mediante mensajes de red a un servidor, permitiendo que la aplicación sea escrita en cualquier lenguaje siempre que respete el protocolo de comunicación.

1.3. Entornos de desarrollos de robots

En la actualidad los principales fabricantes de robots incluyen plataformas de desarrollo para los usuarios de sus productos, por ejemplo ActivMedia ofrece la plataforma ARIA [2], para sus robots Pioneer, PeopleBot, etc.; Evolution Robotics vende su plataforma ERSP [3], y Sony ofrece OPEN-R para sus Aibo. Además, muchos centros de investigación han creado sus propias plataformas de desarrollo como la suite de navegación CARMEN de Carnegie Mellon University, Player/Stage/Gazebo [4], OROCOS [5], Miro [6], y JDE [7]. Estas que generalmente son mas versátiles puesto que tratan de dar soporte a una gran variedad de robots de diferentes fabricantes.

Los entornos de desarrollo de robots ofrecen un acceso mas abstracto y simple a sensores y actuadores, además de funcionalidades de uso común como algoritmos de control, localización, navegación segura, construcción de mapas, etc [8].

El trabajo realizado por James Krammer y col. [9] analiza nueve *Entornos de Desarrollos de Robots* de código abierto y realiza una evaluación y comparación desde varios puntos de vista (utilizando un framework conceptual para comparación sistemática de RDEs).

Los RDEs evaluados son

1. TeamBots
2. Advanced Robotics Interface for Applications (ARIA)
3. Player/Stage
4. Python Robotics (Pyro)
5. Carnegie Mellon Robot Navigation Toolkit (CARMEN)
6. MissionLab
7. APOC Development Environment (ADE)
8. Middleware for Robots (Miro)
9. Mobile and Autonomous Robotics Integration Environment (MARIE)

2. RDE Player/Stage/Gazebo

`Player`, `Stage` y `Gazebo` son tres piezas de software originalmente desarrollado en el laboratorio de investigación de robótica de la *University of Southern California* (USC, Robotics Research Lab) por Brian P. Gerkey y Richard T. Vaughan [10]. Ahora es un proyecto activo de SourceForge.net usado por un gran número de investigadores alrededor del mundo. `Player` es un servidor de dispositivos utilizados en robótica basado en sockets que proporciona una interfaz simple a sensores y actuadores en redes TCP/IP, la abstracción de los sockets posibilita la independencia del lenguaje de programación y de la plataforma de trabajo. En esencia `Player` es un protocolo, cualquier programa que implemente el protocolo puede actuar como `Player`, la implementación de los autores está escrita en C++ y utiliza servicios de sistemas POSIX.

Una parte de `Player` funciona como *servidor de red* para el control de robots y corre abordo del robot actuando como una capa de abstracción de hardware (HAL) para dispositivos robóticos. Otra parte son las *librerías clientes* que brindan acceso a los dispositivos remotos, el proyecto oficial cuentas con librerías clientes en C, C++ y Python; además de otras creadas por terceros como Java, MATLAB, GNUOctave, etc. Los programas clientes utilizan objetos proxy definidos en las librerías clientes para leer/escribir datos desde/hacia los dispositivos.

La utilización de un servidor basado en sockets tiene tres ventajas principales, 1) distribución: los programas clientes tienen acceso a los sensores y actuadores en cualquier lugar sobre la red, 2) independencia: los programas clientes se pueden escribir en cualquier lenguaje de programación lo que da flexibilidad en la plataforma de desarrollo y 3) conveniencia: el servidor proporciona interfaces de abstracción unificada de los dispositivos conectados al mismo (ver figure 2).

`Player` soporta múltiples conexiones concurrentes de clientes a dispositivos, creando nuevas posibilidades de control y sensado distribuido y colaborativo, además incluye varias herramientas de soporte como visualización de información de sensores de forma gráfica. A diferencia de muchos entornos de desarrollos propietarios que vienen incluidos con kits de robots, `Player` no esta ligado a un sistema robótico en particular, y debido a su diseño modular los programadores pueden agregar soporte a nuevo hardware.

`Stage` es un simulador de múltiples robots del proyecto Player que simula una población de robots, sensores y objetos en un entorno bitmap 2D. `Stage` dispone de robots virtuales de modo que Player interactúe con el entorno simulado en lugar de los dispositivos físicos, además de varios modelos de sensores incluyendo sonares, sensores láser rangefinder, cámaras pan-tilt-zoom con detección de blobs de color y odometria. `Stage` es adecuado para investigaciones de sistemas

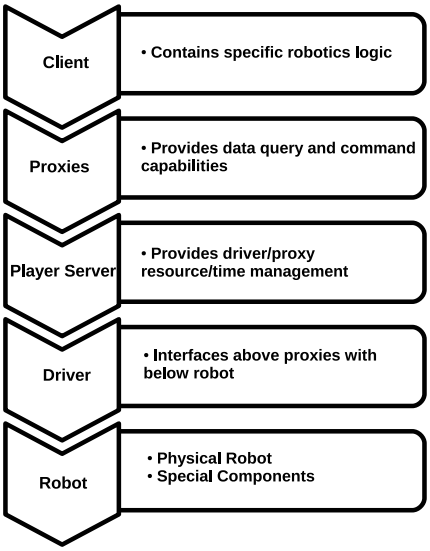


Figura 2: Niveles de abstracción de Player

autónomos multi-agente, puesto que proporciona un modelo simple de bajo requerimiento computacional de múltiples dispositivos en lugar de emular cada dispositivo con gran fidelidad.

`Gazebo` es un simulador de múltiples robots en entornos interiores y exteriores. Tal como Stage, es capaz de simular una población de robots, sensores y objetos pero en un entorno tridimensional. *Gazebo* genera realimentación realista de sensores e interacción física entre objetos, incluyendo simulación precisa de la física de cuerpos rígidos con su dinámica y detección de colisiones. Todos los objetos simulados tiene masa, velocidad, ficción, y otros atributos que les da un comportamiento realista al ser empujados, tirados, golpeados, etc.

A partir de la versión 2.0 de `Player` [11] donde se realizaron mejoras en la estructura interna del programa permitiendo mayor flexibilidad y simplicidad, el conjunto de herramientas `Player/Stage/Gazebo` se han convertido en un estándar de facto en la comunidad robótica "open source" [12]. Con estas mejoras `Player` es ahora un verdadero framework de robótica que permite reducir el tiempo y complejidad de desarrollo, además de brindar independencia de la plataforma, escalabilidad y promover la reutilización de software.

2.1. Interfaz, Drivers y Dispositivos

En *Player* hay tres conceptos claves (figura 3)

- **Interfaz:** Es una especificación de como interactuar con cierta clase sensores, actuadores o algoritmos de robots. Las *interfaces* definen la sintaxis y semántica de los mensajes que se puede intercambiar con entidades de la misma clase.
- **Driver:** Es una pieza de software (generalmente escrita en C++) que se comunica con los sensores, actuadores o algoritmos del robot, traduciendo sus entradas y salidas para adaptarse a una o mas *interfaces*. El *driver* esconde la especificación de cualquier entidad haciéndolo parecer el mismo para cualquier otra entidad de su misma clase.

- **Dispositivo:** Todos los mensajes en Player ocurren entre *dispositivos* a través de las *interfaces*. Los *drivers*, aunque realizan la mayoría del trabajo, nunca se acceden directamente.

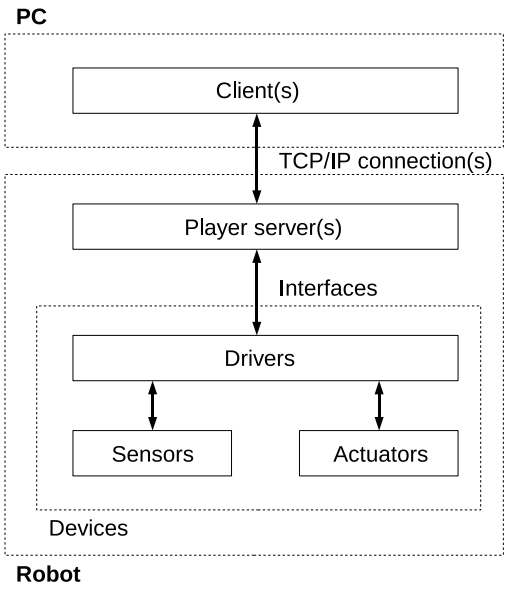


Figura 3: Interfaz, Driver y Dispositivos

Algunas interfaces incluidas en *Player* son

- **position2d** Planar mobile robot
- **laser** Laser range-finder
- **camera** Camera imagery
- **ranger** A range sensor
- **actarray** An array of actuators
- **gripper** Gripper interface
- **localize** Multi-hypothesis planar localization system
- **planner** A planar path-planner
- **blobfinder** A visual blob-detection system
- **fiducial** Fiducial (marker) detection
- **joystick** Joystick control
- **ptz** Pan-tilt-zoom unit
- **log** Log read/write control
- **imu** Inertial measurement unit
- **gps** Global positioning system
- **map** Access maps

Algunos dispositivos incluidos en *Player* son

- **Robots**

- Acroname Garcia
- iRobot Roomba
- Segway
- MobileRobots (ActivMedia) Pioneer

- **Hardware**
  - SICK LMS200 laser
  - Hokuyo URG laser
  - Sony EVID30/EVID100 pan-tilt-zoom camera
  - CMUcam2
  - IEEE1394 (Firewire) cameras
  - Camera supported by Video4Linux
  - DirectedPerception PTU-D46 pan-tilt unit
- **Software**
  - Color Machine Vision (CMVision) blog-traking software (Jim Bruce)
  - ActivMedia Color Tracking System (ACTS) blob-tracking software (Paul Rybsi)
- **Algorithms**
  - Vector Field Histogram (VFH+), goal-seeking obstacle avoidance algorithm (Ulrich & Borenstein)
  - Adaptive Monte Carlo Localization (AMCL) (Fox)
  - Wavefront propagation planner (Latombe)

2.2. Drivers de Player

`Player` incluye varias *interfaces* predefinidas para los algoritmos y dispositivos en robótica. El objetivo de un driver es crear un vínculo entre los dispositivos/algoritmos y la interfaz predefinida que mejor lo representa. Existen dos tipos de drivers en Player, el *driver normal* de Player y el *driver plugin*, este último tiene la ventaja de no necesitar recompilar Player para compilar el driver. En el caso de driver plugin este se enlaza con el servidor en tiempo de ejecución. El driver implementa métodos estándares para comunicarse con los proxies, un proxy es un estándar de comunicación definido por Player para un objeto dado, tal como un sonar, el movimiento de un vehículo, etc.

2.3. Cliente Player

Las librerías clientes están disponibles en varios lenguajes para facilitar el desarrollo de programas clientes TCP. Estas librerías se utilizan para desarrollar los programas de control de robot con el servidor Player ya sea con un hardware o simulado. Las librerías disponibles por el proyecto son

- `libplayerc` Librerías clientes C
- `libplayerc++` Librerías clientes C++
- `libplayer_py` Librerías clientes python

Existen además otras librerías de contribución que incluyen `MATLAB`, `Smalltalk`, `Java`, `GNUOctave`, etc.

2.3.1. Ejemplo de programa cliente en C++

El siguiente código muestra el manejo del proxy position2d para el robot RoMAA

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

int
main( int argc, char *argv[ )
{
 // Connect to the local player process on port 6665
 PlayerCc::PlayerClient romaa_robot( "localhost", 6665 );

  // Create a position2d proxy
  PlayerCc::Position2dProxy romaa_pos2d( &romaa_robot, 0 );

  romaa_pos2d.ResetOdometry();
  romaa_pos2d.SetSpeed( 1.0, 0 );

  for( int i = 0; i < 200; i++ )
  {
   romaa_robot.Read();

   std::cout << "i: "'<< i << " ---> ";
   std::cout << "px: " << romaa_pos2d.GetXPos() << " - ";
   std::cout << "py: " << romaa_pos2d.GetYPos() << " - ";
   std::cout << "pa: " << romaa_pos2d.GetYaw() << std::endl;

   usleep(50000);
  }

  romaa_pos2d.SetSpeed( 0, 0 );
  return 0;
 }
```

El siguiente código muestra el manejo del proxy laser

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

int main( int argc, char* argv[] )
{
  // Connect to the local player process on port 6665
  PlayerCc::PlayerClient romaa_robot( "localhost", 6665 );
  // Create a laser proxy
  PlayerCc::LaserProxy romaa_laser( &romaa_robot, 0 );

  romaa_robot.Read( );

  std::cout << "Laser data..." << std::endl;
  for( int i = 0; i < romaa_laser.GetCount( ); i++ )
  {
    std::cout << romaa_laser.GetRange( i ) << " ";
  }
  std::cout << std::endl;

  std::cout << "Laser bearing..." << std::endl;
  for( int i = 0; i < romaa_laser.GetCount( ); i++ )
  {
    std::cout << romaa_laser.GetBearing( i ) << " ";
  }
  std::cout << std::endl;
```

```
  return 0;
}
```

2.4. Herramientas de Player

Así como los sistemas operativos están provistos de servicios básicos para controlar la PC, Player provee de servicios básicos para controlar robots. Esto incluyen herramientas como

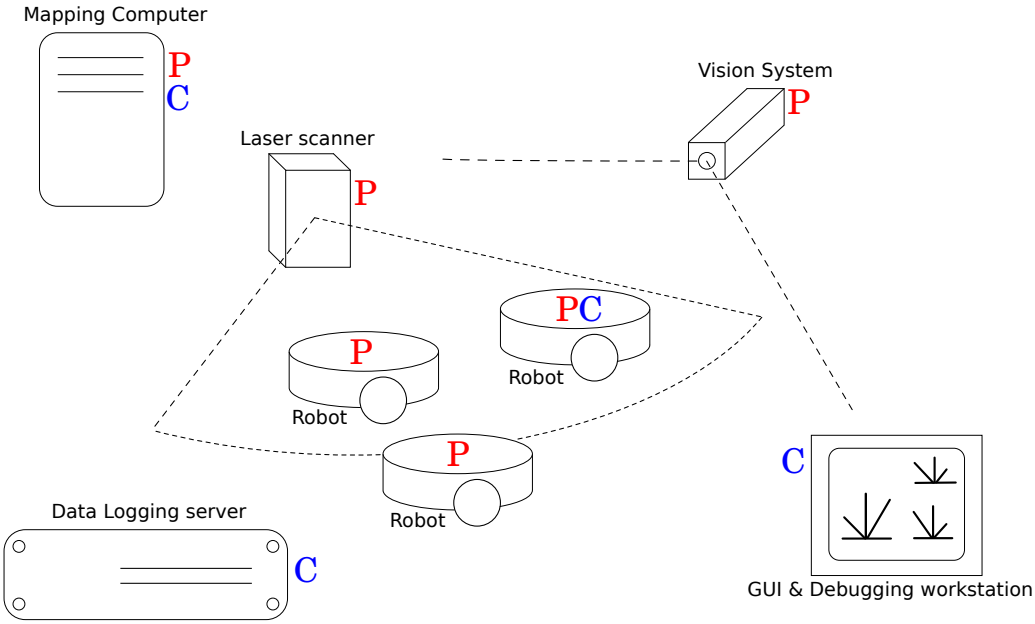
- **playerprint** Solicita e imprime datos de sensores a consola
- **playerv** Solicita y muestra gráficamente datos de sensores; también provee de teleoperación por medio de movimiento del mouse
- **playerjoy** Provee de teleoperación por medio de joystick
- **playervcr** Provee control remoto de logging de datos y playback
- **playernav** Unidad de control gráfica que provee de control sobre múltiples robots para localización y planificación de trayectoria (path-planning)
- **playerwritemap** Solicita mapa de grilla y vector (por ej. para un driver de SLAM) y lo escribe a disco
- **playercam** Muestra imagen de video remotamente desde cámaras montadas en robots

2.5. Flexibilidad de Player

La figura 4 muestra el uso de *Player* en una aplicación de sensado y control distribuido. En este esquema se representa un programa de *control de formación* que corre sobre uno de los robots, este se subscribe a los sensores de sonar y control de tracción de los otros robots para enviarles comandos de velocidad a las ruedas y mantener una distancia y orientación fija respecto del mismo. Mientras tanto la persona que realiza el experimento esta haciendo debug del controlador de formación; examinando las lectura de sonar de los robots desde un cliente GUI corriendo sobre una estación de trabajo. Un server de logging almacena las posiciones (ground-truth) para futuros trabajos, el logger se subscribe al dispositivo de tracking corriendo sobre un sistema de visión global. Simultáneamente, un colega de otra universidad con acceso a la red interna corre un cliente de mapeo on-line que se subscribe a cada dispositivo disponible, incluyendo un scaner laser montado en la pared y los tres sonares de los robots. La aplicación de mapeo no controla ningún actuador, así que el delay de 300ms de transmisión sobre Internet no es un problema. Este escenario, aunque es complicado se puede llevar a cabo mediante el protocolo y la implementación de *Player*.

3. Servidor Player

La estructura del servidor de `Player` esta diseñada como un conjunto de hilos (threads en inglés) asíncronos, que se muestra en la figura 5, la porción central es Player en sí mismo, a la izquierda están los dispositivos físicos y la derecha los clientes, cada cliente tiene un socket TCP de conexión con `Player`; si el cliente se está ejecutando sobre el mismo host que `Player` este socket simplemente es una conexión loopback; sino, existe una conexión física entre ellos. Del otro lado, `Player` se conecta a cada dispositivo por medio de cualquier método apropiado para el dispositivo (por ejemplo, RS232). El hilo principal escucha conexiones de nuevos clientes y utiliza un área de memoria compartida para intercambiar datos hacia los cliente o dispositivos; además tiene buffers de memoria de cada lado tanto para los comandos y datos con hilos para su manejo. Los programas clientes se comunican con el servidor para leer datos de sensores y escribir comandos a los actuadores.



"Most Valuable Player: A Robot Device Server for Distributed Control". Brian P. Gerkey, Richard T. Vaughan, Kasper Støy, Andrew Howard, Gaurav S Sukhatme, and Maja J Mataric´. In Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS 2001), pages 1226-1231, Wailea, Hawaii, October 29 - November 3, 2001

Figura 4: Escenario de ejemplo usando Player

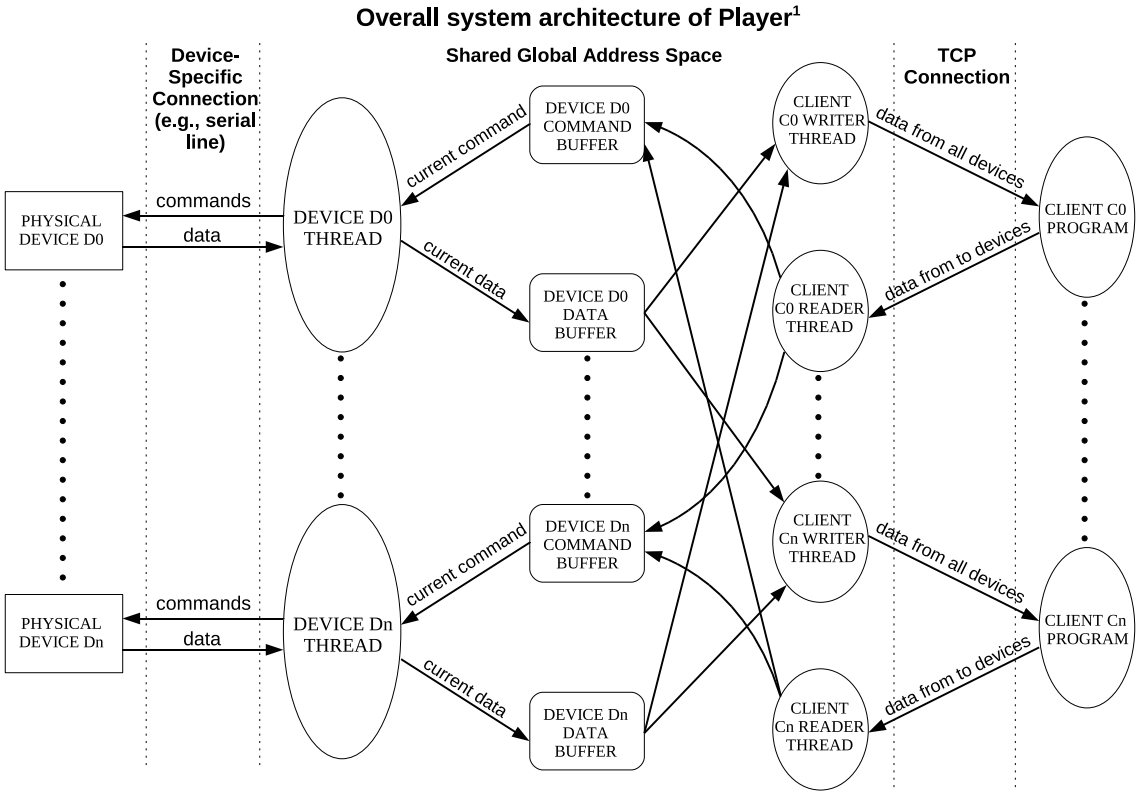


Figura 5: Arquitectura del servidor Player

El modelo de dispositivo en `Player` es de tipo UNIX [13] donde los dispositivos se tratan como archivos, hace también una diferenciación fuerte entre la interfaz de dispositivo y los drivers de dispositivos, donde las interfaces especifican como interactuar con cierta clase de sensores, actuadores o algoritmos de robots definiendo la sintaxis y semántica de los mensajes que se puede intercambiar con entidades de la misma clase; mientras que los drivers son una parte de software (generalmente escrita en C++) que se comunica con los sensores, actuadores o algoritmos del robot, traduciendo sus entradas y salidas para adaptarse a una o mas interfaces, el driver esconde la especificación de cualquier entidad haciéndolo parecer la misma para cualquier otra entidad de su misma clase.

3.1. Archivos de configuración

Como se mencionó anteriormente Player es una capa de abstracción de hardware que conecta el código de la aplicación con el hardware del robot funcionando como una aplicación Cliente/Servidor. Los *archivos de configuración* le indican al servidor Player cuales drivers utilizar y que interfaces usan estos drivers. Para cada modelo en la simulación o dispositivo real con el que se necesita interactuar se debe especificar un driver mediante un archivo de configuración `cfg` de la siguiente manera

```
driver
(
  name "driver_name"
```

```
  provides [device_address]
  # other parameters...
)
```

Los valores de `name` y `provides` son obligatorios, sin los cuales Player no sabe que driver utilizar (dado por `name`) y que tipo de información envía/recive el driver usado (dado por `provides`). El parámetro `name` tiene que ser un nombre de los drivers incluidos en Player. El parámetro `provides` le indica a Player que interfaz utilizar para poder interpretar la información dada por el driver (por ejemplo la información de sensores del robot), esta información que provee el driver se utiliza en el código de la aplicación de robótica. Este parámetro es del tipo "device address" que indica el puerto TCP y la interfaz donde se encuentra el dispositivo y se indica de la siguiente manera

```
provides [ "key:host:robot:interface:index"
           "key:host:robot:interface:index"
           "key:host:robot:interface:index"
           ...]
```

Los demás parámetros pueden ser `requires` y `plugin`, el primero se utiliza para drivers que requieren información de entrada tales como el driver `"vfh"` y es similar al parámetro `provides`, el segundo le indica a Player donde encontrar la información sobre el driver utilizado.

En el listado 1 se muestra el archivo de configuración de los robots Pioneer donde se ven las diferentes interfaces utilizadas como `position2d`, `sonar`, `power`, `bumper`, etc., además del campo `key` de la dirección de los dispositivos indicando la misma interfaz para diferentes dispositivos. El listado 2 muestra el driver del robot `RoMAA`, además de los parámetros estándar incluye otros particulares de este driver que permite ajustar el PID de los controladores de los motores. Los listados 3 y 4 muestra archivos de configuración para sensores láser `SICK` y `Hokuyo`, respectivamente.

Por último, el listado 5 muestra un ejemplo de archivo de configuración del driver software que implementa el método de navegación Vector Field Histogram (VFH) que realiza evasión de obstáculos en tiempo real y seguimiento de trayectoria en robots móviles.

Listado 1: pioneer.cfg

```
driver
(
  name "p2os"
  provides ["odometry:::position2d:0"
            "compass:::position2d:1"
            "gyro:::position2d:2"
            "sonar:0"
            "aio:0"
            "dio:0"
            "power:0"
            "bumper:0"
            "gripper:::gripper:0"
            "blobfinder:0"
            "sound:0"
           ]
  port "/dev/ttyS0"
)
```

Listado 2: romaa.cfg

```
driver
(
  name       "romaadriver"
  plugin     "libromaadriver"
  provides [ "position2d:0" ]
  port       "/dev/ttyUSB0"
  baudrate       38400
  motor_pid_kp 7
  motor_pid_ki 1
  motor_pid_kd 0
  vw_pid_ki      1
  vmultiplier    100
  wmultiplier    100
)
```

Listado 3: sick.cfg

```
driver
(
  name "sicklms200"
  provides ["laser:0"]
  port "/dev/ttyS2"
)
```

Listado 4: hokuyo.cfg

```
driver
(
  name      "urglaser"
  provides  ["laser:0"]
  port      "/dev/ttyS1"
  pose [0.0 0.0 0.0]
  min_angle   -115.0
  max_angle   115.0
  use_serial 1
  baud        115200
  alwayson    1
)
```

Listado 5: vfh.cfg

```
driver
(
  name "vfh"
  provides ["position2d:1"]
  requires ["position2d:0" "laser:0"]
  cell_size 0.1
  window_diameter 61
  sector_angle 1
  safety_dist_0ms 0.2
  safety_dist_1ms 0.4
  max_speed 0.3
  max_turnrate_0ms 75
  max_turnrate_1ms 50
  weight_desired_dir 5.0
  weight_current_dir 3.0
)
```

3.2. Dirección de dispositivo

La dirección de dispositivo le indica a Player donde presentar o recibir la información del driver y que interfaz se utiliza para acceder a dicha información. Se especifica mediante una cadena de la forma `key:host:robot:interface:index` donde cada campo se separa mediante dos puntos

- `key`: permite soportar múltiples interfaces del mismo tipo desde diferentes dispositivos
- `host`: dirección de la computadora host donde se encuentra el dispositivo (dirección IP)
- `robot`: puerto TCP en el que Player espera recibir datos desde una interfaz
- `interface`: interfaz utilizada para interactuar con los datos
- `index`: permite múltiples dispositivos del mismo tipo (misma interfaz), por ejemplo dos cámaras podrían ser `camera:0` y `camera:1`. Es diferente al campo `key` puesto que "tener un driver que soporta varias interfaces del mismo tipo" NO es lo mismo que tener múltiples dispositivos que usan la misma interfaz

3.3. Player con Stage

Las aplicaciones robóticas desarrolladas mediante programas clientes de Player se puede simular mediante `Stage` simplemente ejecutando el servidor Player con un archivo de configuración que

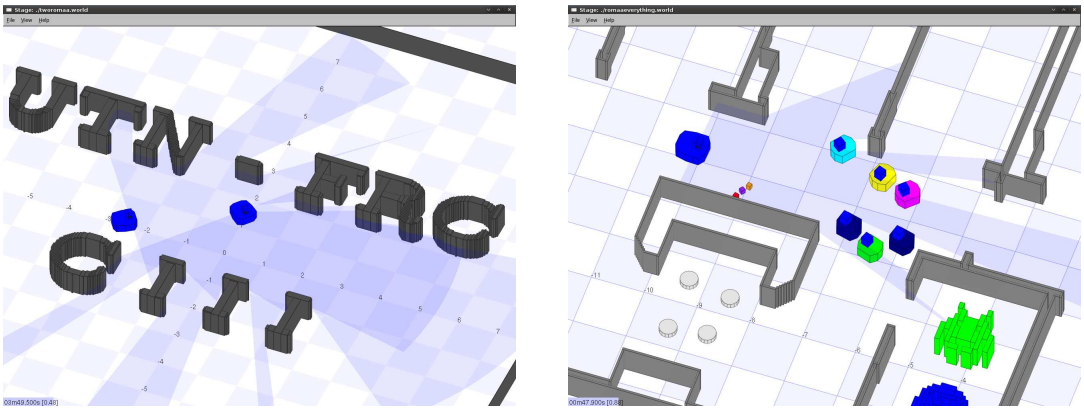
en lugar de cargar driver de dispositivos reales, ejecute el simulador. Un ejemplo de archivo de configuración se muestra en el listado 6

Listado 6: stage.cfg

```
driver
(
  name "stage"
  plugin "libstageplugin"
  provides ["simulation:0" ]
# load the named file into the simulator
  worldfile "empty.world"
)
```

Este archivo de configuración carga el "driver" cuyo nombre es `stage` del tipo driver plugin que se encuentra en `libstageplugin`, y provee como interfaz `simulation`, el parámetro `worldfile` indica un archivo que describe el entorno `.world` a simular. En el archivo tipo `.world` se configura la interfaz gráfica de Stage, se carga el mapa de bits que representa en entorno simulado y los diferentes modelos de dispositivos a utilizar como ser un robot móvil, sensores láser, sonar, etc. En [14] se presenta un tutorial de como utilizar Stage con Player y generar los archivos `.world` y modelos de robot.

La figura 6 muestra la interfaz gráfica del simulador Stage.



(a) Dos robots RoMAA (b) Robots RoMAA y Pioneer

Figura 6: Escenarios del simulador Stage

3.4. Player con Gazebo

En `Gazebo` al igual que con `Stage` se utilizan archivos `.world` para describir el entorno a simular, este archivo describe el diseño de robot, sensores, fuente de luz, componentes de la interfaz de usuario, etc. Además, también se puede utilizar para controlar algunos aspectos del motor de simulación, tales como la fuerza de gravedad o el paso temporal de la simulación. Los archivos `.world` de Gazebo están escritos en XML lo que permite crearlos y modificarlos usando un editor de texto.

Para utilizar Gazebo con Player primero se tiene que ejecutar

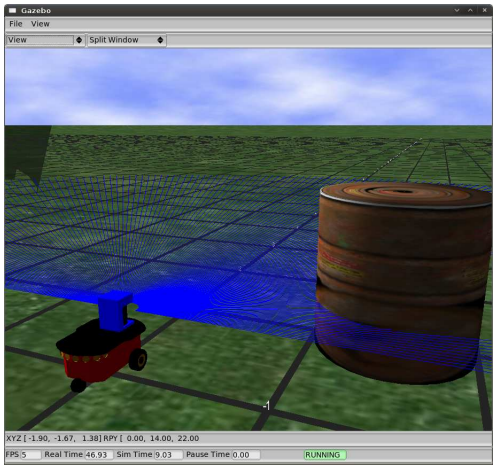
`$>gazebo worldfile.world`

y en otra terminal

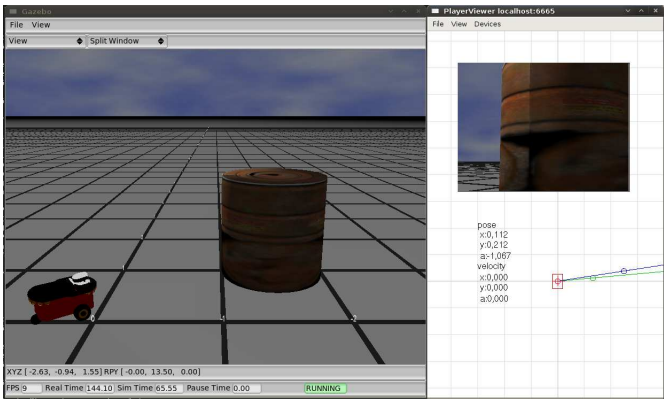
`$>player cfgfile.cfg`

que indique las interfaces de los controladores instanciados en el archivo `.world`.

La figura 7 muestra la interfaz gráfica del simulador Gazebo.



(a) Robot Pioneer con laser SICK-LMS200



(b) Robot Pioneer con cámara PTZ

Figura 7: Escenarios del simulador Gazebo

Referencias

[1] Geoffrey Biggs and Bruce Macdonald. A survey of robot programming systems. In *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, page 27, 2003.

[2] Advanced robotics interface for applications. `http://www.activrobots.com/SOFTWARE/aria.html`.

[3] Robotic development platform oem software by evolution robotics. http://www.evolution.com/products/ersp/.

[4] Player project. http://playerstage.sourceforge.net/.

[5] The orocos project. http://www.orocos.org/.

[6] Miro - middleware for robots. http://miro-middleware.berlios.de/.

[7] Jde middleware. http://jde.gsyc.es/index.php/Main_Page.

[8] J.M. Cañas, Matellán V, and R. Montúfar. Programación de robots móviles. pages 99–110, 2006.

[9] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Auton. Robots*, 22(2):101–132, 2007.

[10] Brian P. Gerkey, Richard T. Vaughan, Gaurav S. Sukhatme, Kasper Stoy, Andrew Howard, and Maja J. Mataric. Most valuable player: A robot device server for distributed control, 2001.

[11] Toby H. J. Collett, Bruce A. Macdonald, and Brian P. Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA)*, Sydney, Australia, 2005.

[12] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. On device abstractions for portable, reusable robot code. In *In IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2421–2427, 2003.

[13] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *In Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, 2003.

[14] Jennifer Owen. How to use player/stage. http://playerstage.sourceforge.net/doc/playerstage_instruc July 2009.