USC Robotics Labs
University of Southern California
Los Angeles, USA

Player Robot Server

Version 0.7.4a User Manual

Brian Gerkey, Kasper Støy, and Richard T. Vaughan

{gerkey|kaspers|vaughan}@robotics.usc.edu

Technical Report IRIS-00-392
http://iris.usc.edu/~irislib

This report may not contain the most current documentation on
Player. For the latest documentation, consult the Player homepage:
http://fnord.usc.edu/player

November 22, 2000

Contents

1 Introduction **3**
- 1.1 Description . . . 3
- 1.2 Getting Player . . . 4
- 1.3 Bugs . . . 4
- 1.4 On the Name Player . . . 4
- 1.5 Acknowledgements . . . 4

2 Running Player **5**
- 2.1 Building and Installing Player . . . 5
- 2.2 Command Line Arguments . . . 5
- 2.3 Device Support Within Player . . . 5
- 2.4 Interaction With the Stage Simulator . . . 6

3 Socket Interface **7**
- 3.1 A Note on Data Types . . . 7
- 3.2 Connecting to the Server . . . 7
- 3.3 Requesting Access to Devices . . . 7
- 3.4 Sending Command to Devices . . . 8
- 3.5 Reading Data from Devices . . . 9
  - 3.5.1 ACTS Blob Data Format . . . 9
- 3.6 Configuring the Server . . . 10
- 3.7 Summary of Messages . . . 13

4 Architecture **14**
- 4.1 Server Structure . . . 14
- 4.2 Extending the Server . . . 17

A The C++ Interface **19**
- A.1 Hello World . . . 19
- A.2 Connecting . . . 19
- A.3 Request . . . 19
- A.4 Read . . . 21
- A.5 Command . . . 23
- A.6 Configuration . . . 24

B The Tcl/Tk Interface **25**

C The Java Interface **26**
- C.1 The **PlayerClient**-class . . . 26
- C.2 Motor commands . . . 27
- C.3 Sensor data . . . 27
- C.4 JAVA example, Obstacle avoidance using sonar . . . 29

1 Introduction

This document describes the Player robot server version 0.7.4a. It was developed mainly by Brian Gerkey and Kasper Støy; the Stage simulator interface (see Section 2.4) was written by Richard T. Vaughan. Player was created at the University of Southern California Robotics Research Labs, and is released under the GNU Public License.

We have tried to make this documentation as complete as possible. Hopefully there is sufficient information here for you to use Player and the provided clients as well as write your clients in your language of choice.

Questions and comments regarding Player should be directed to Brian Gerkey (`gerkey@robotics.usc.edu`). What do you like? What do you hate? Do you actually use it?

1.1 Description

What is Player? Player is a multi-threaded robot device server. It gives you simple and complete control over the physical sensors and actuators on your mobile robot[1] When Player is running on your robot, your client control program connects to it via a standard TCP socket, and communication is accomplished by the sending and receiving of some of a small set of simple messages.

Player is designed to be language and platform independent. Your client program can run on any machine that has network connectivity to your robot, and it can be written in any language that can open and control a TCP socket. We currently have client-side utilities available in C++, Tcl, and Java, with Python in the works. Further, Player makes no assumptions about how you might want to structure your robot control programs. In this way, it is much more "minimal" that other robot interfaces. If you want your client to be a highly concurrent multi-threaded program, write it like that. If you like a simple read-think-act loop, do that. If you like to control your robot interactively, try our Tcl client (or write your own client utilities in your favorite interactive language).

Player is also designed to support virtually any number of clients[2]. Have you ever wanted your robots to "see" through each others' eyes? Now they can. 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, you can also use Player for monitoring of experiments. For example, while your C++ client controls a robot, you can run a Tcl/Tk GUI client elsewhere that shows you current sensor data, and evens logs it for later analysis. Also, on-the-fly device requests allow your clients to gain access to different sensors and actuators as needed for the task at hand.

The behavior of the server itself can also be configured on the fly. See Section 3.6 for details on changing sensor data rate and other features.

Also, the interface to Player is identical to the interface to the Stage robot simulator; see Section 2.4 for details.

Last but not least, Player is free software, released under the GNU Public License. If you don't like some part of how it works, change it. And please send us your patch!

[1] Player is currently specifically implemented for the ActivMedia Pioneer 2-DX mobile robot, but the extensible server architecture (see Section 4) should allow simple porting to other platforms.

[2] There is a limit to the number of client service threads that the server will spawn. At the moment, that number is 16; at two service threads per client, Player can support 8 simultaneous clients. As far as I know, that number could be raised without problem.

1.2 Getting Player

The Player homepage is at:
`http://fnord.usc.edu/player`
Check there for the latest versions of the server and this document. You can download the latest tarball directly from:
`ftp://fnord.usc.edu/pub/player`

1.3 Bugs

This is new software, fresh off the presses. As such, it is bound to contain some bugs, despite our diligent testing. If you manage to break something, or if some aspect of Player's behavior seems wrong or non-intuitive, let us know. Email should be sent to Brian Gerkey (`gerkey@robotics.usc.edu`). Include as much information as possible, including at least Player version and OS version. A detailed description of what happened will enable us (hopefully) to repeat and analyze the problem. Of course, there is NO WARRANTY on this software, and we're not paid to maintain it, so there is no guarantee that we'll fix your problem. But we'll try :).

1.4 On the Name Player

Player was originally named Golem, and the Stage simulator was originally named Arena. However, we soon discovered that many, many, many pieces of robotics-related software already use those names. So, we had to make a change. We needed names that capture the now-integral relationship between the server and simulator, so we chose Player and Stage, as suggested by a living Englishman, in reference to a very dead Englishman.

From *As You Like It* Act II, Scene 7:

> "All the world's a stage,
> And all the men and women merely players:
> They have their exits and their entrances;
> And one man in his time plays many parts,"

From *Macbeth* Act V, Scene 5:

> "Life's but a walking shadow, a poor player
> That struts and frets his hour upon the stage
> And then is heard no more: it is a tale
> Told by an idiot, full of sound and fury,
> Signifying nothing."

1.5 Acknowledgements

This work is supported by DARPA grant DABT63-99-1-0015 (MARS), NSF grant ANI-9979457 (SCOWR), DARPA contract DAAE07-98-C-L028 (TMR), ONR Grants N00014-00-1-0140 and N0014-99-1-0162, and JPL Contract No. 1216961.

We thank those in the lab who have contributed to the development of the Player server and associated utilties and tools: Esben Østergård, Jakob Fredslund, Andrew Howard, and Boyoon Jung.

2 Running Player

2.1 Building and Installing Player

To build Player, you need C++ and POSIX threads. Actually, right now, you specifically need a recent version of Linux (any 2.2.x kernel should work, as well as any earlier versions that have the same pthread interface), because we use some Linux-specific pthread behavior that may or may not be portable to other pthread implementations (this will be cleaned up soon).

Open your tarball and read the README in the top-level directory; it will tell you what to do.

2.2 Command Line Arguments

Player accepts the command-line arguments shown in Table 1.

Argument	Meaning
-gp <port>	The Player server should listen on TCP port <port>. Default is `6665`.
-pp <port>	The P2OS device (the robot itself) is attached to serial port <port>. Default is `/dev/ttyS0`.
-lp <port>	The SICK laser rangefinder is attached to serial port <port>. Default is `/dev/ttyS1`.
-zp <port>	The Sony PTZ camera is attached to serial port <port>. Default is `/dev/ttyS2`. Note that Player will **not** control the Sony camera through the P2OS microcontroller, as is the standard configuration from ActivMedia.
-vp <port>	The ACTS vision server should listen on TCP port <port>. Default is `5001`. Well, actually, due to a bug in ACTS, it runs on 35091, and will not run anywhere else, regardless of what you tell it. Thus this argument has no effect right now.
-vc <path>	The ACTS vision server should use configuration file found at <path>. Default is `/usr/local/acts/actsconfig`.
-stage <path>	Run Player as a child process under the Stage simulator, and use memory-mapped IO through the file found at <path> to communicate with the simulator. Default is to communicate with the physical devices, not the simulator.
-debug	Print out various debug information (not recommended).
-exp	Enable various experimental features (definitely not recommended).

Table 1: Player command-line arguments

2.3 Device Support Within Player

Currently, Player is tailored to run on the ActivMedia Pioneer 2-DX mobile robot[3]. As such, Player supports the following specific accessories:

- Pioneer 2-DX wheels (motors and encoders) (via P2OS)
- Pioneer 2-DX compass (via P2OS)

[3]Other Pioneer 2 models may work, but likely we'll need to add some bits to our P2OS parsing code. Since we only have Pioneer 2-DXs, we really can't implement anything else right now.

- Pioneer 2-DX sonars (via P2OS)
- Pioneer 2-DX bumpers (via P2OS)
- Pioneer 2-DX gripper (motors and lift/paddle state; regrettably, there are no encoders) (via P2OS) *Actually, at the moment, gripper control is disabled...*
- Other Pioneer 2-DX state data; currently, only battery voltage, but soon digital and analog I/O (via P2OS)
- Sony PTZ camera (pan, tilt, & zoom commands and feedback) (directly via serial port)
- SICK LMS-200 laser rangefinder (at 38Kbps) (directly via serial port)
- ACTS color vision system (directly via TCP socket)

2.4 Interaction With the Stage Simulator

As extra incentive to use Player with your Pioneer robot, you should know that the TCP interface to Player is identical to the TCP interface to Stage, a lightweight robot simulator that currently simulates most of the devices available on the Pioneer 2-DX robot. In fact, Stage actually spawns a copy of Player for each simulated robot and feeds it simulated data. This means that a program that you develop in the simulator can be used to control a real robot with no changes[4]; if you include reasonable command-line arguments in your program for which host and port to connect to, you can actually run the same binary on the simulator or the real robot! Information about Stage can be found at:

`http://robot.usc.edu/stage`

[4]Since Stage does not completely faithfully simulate a Pioneer (no dynamics, for example), subtle changes might be required to fine-tune your controller.

3 Socket Interface

This section describes the TCP/IP socket interface to the Player server. See appendices for specific examples and source code.

3.1 A Note on Data Types

We are about to describe the protocol-level details of the socket interface to Player. As such, it is worth making clear two details regarding data types. First, the various messages that are sent between client and server are composed of fields of three different sizes, as listed in Table 2. They may be signed or unsigned, but they will always be the same size. The second important detail is that all data on the network is in network byte-order (big-endian)[5]. So, before sending a message to the server, the client must ensure that all `short`s and `int`s are in network byte-order. Analogously, before interpreting any messages from the server, the client must ensure that all `short`s and `int`s are properly ordered. Single characters need never be swapped.

Type	size (in bytes)
byte/character	1
short	2
int	4

Table 2: Data types

3.2 Connecting to the Server

First a connection to Player needs to be established. This is done by creating a TCP socket and connecting to Player on port number[6] 6665. Note that the server will not send any messages at this time; it is waiting for direction from the client.

3.3 Requesting Access to Devices

To get access to the robot resources the client sends a sensor request message. The format of this request is:

```
dr<size><device_name><access>...<device_name><access>
```

The leading `dr` (0x6472) tells Player that this is a device request. The `<size>` field is an unsigned short telling how many bytes follow. Each `<device_name>` is a single character identifying the device to which access is needed. The devices available and their codes are shown in Table 3(a). Each `<access>` is a single character code for the access needed to the preceding device. The codes and their meaning are shown in Table 3(b) (see Section 3.5 for details on each device). *Read* access means that the server will start sending data from the specified device. For instance, if read access is obtained for the sonar device Player will start sending sonar data to the client. *Write* access means that the client has permission to control the actuators of the device. There is no locking

[5] x86 machines are little-endian; thus clients running on them must byte-swap.

[6] This is the default port; Player can be configured to listen on a different port through a command-line option at startup. See Section 2.2.

ASCII Code	Device
l (0x6C)	Laser
s (0x73)	Sonar
p (0x70)	Position
g (0x67)	Gripper
m (0x6D)	Miscellaneous
v (0x76)	Vision
z (0x7A)	Pan-Tilt-Zoom Camera

(a)

ASCII Code	Access
r (0x72)	read
w (0x77)	write
a (0x61)	all (read and write)
c (0x63)	close (no access)
e (0x65)	error (only used by the server)

(b)

Table 3: (a) Devices and their character codes. (b) The access codes.

mechanism so different clients can have concurrent write access to the same actuators. *All* access is both of the above and finally *close* means that there is no longer any access to the device. Device request messages can be sent at any time, providing on the fly reconfiguration for clients that need different devices depending on the task at hand. Of course, not all of the access codes are applicable to all devices; for instance it does not make sense to write to the sonars. However, a request for such access will not generate an error; rather, it will be granted, but any commands actually sent to that device will be ignored. Note that several requests can be combined in one request packet. Also, although the devices can be specified in any order, the server will send sensor data back in the order originally requested.

When Player receives the request it sets up the appropriate devices (which might take a few seconds) and then responds to the client telling which requests went well. This is done by returning a reply of the following format:

```
r<size><device_name><access>...<device_name><access>
```

The leading `r` (0x72) indicates that this is a reply message. The next field, `size`, is an unsigned short telling how many bytes will follow. Then comes the same request string that the client sent to the server, but with an `e` (0x65) as access code for those devices that encountered some error [7]. Thus a simple check for successful device setup is to compare the server's reply (after the leading `r` (0x72) and `<size>`) with the client's original request.

3.4 Sending Command to Devices

When write access to a device has been obtained, the client can send commands for that device. The general format of the command message is:

```
c<device_name><size><command>
```

The leading `c` (0x63) tells Player that a command is following. `<device_name>` is a character code for the device to which the command is to be sent; the codes and their meanings are shown in Table 4. `<size>` is an unsigned short telling the size in bytes of the command field. The `<command>` field is device-dependent; consult Table 5 for the various formats.

[7] Although the response to the initial request will arrive before any sensor data, responses to later on the fly requests will be intermingled with any previously requested sensor data.

ASCII Code	Name	Description
p (0x70)	position	Wheel motor control
z (0x7A)	ptz	Pan-Tilt-Zoom camera motor control
g (0x67)	gripper	Gripper motor control (paddle and lift positioning) *Note: gripper control is currently disabled. Will be fixed soon.*

Table 4: The commandable devices and what they control

Device	Size	Fields	Meaning
position	4	short short	Speed (mm/sec; positive is forward) Turn rate (degrees/sec; positive is counterclockwise)
ptz	6	short short short	Absolute pan (-100 to 100 degrees; positive is counterclockwise) Absolute tilt (-25 to 25 degrees; positive is up) Absolute zoom (0 to 1023; 0 is wide and 1023 is telefoto)
gripper	2	2 unsigned bytes	These two bytes are sent directly to the gripper; refer to Table 3-3 page 10 in the Pioneer 2 Gripper Manual[2] for a list of commands. The first byte is the command. The second is the argument for the LIFTcarry and GRIPpress commands, but for all others it is ignored.

Table 5: Format of the `<command>` field for each device

In the interest of simplifying the protocol, the server does NOT respond to command messages. Badly formatted commands and commands to devices for which write permission was never established will only cause errors to be printed on the console from which Player was launched.

3.5 Reading Data from Devices

When read access has been requested for a device, sensor data from that device it is sent to the client. By default, the server continuously sends sensor data at 10Hz; see Section 3.6 for information on changing this behavior. The general format of the data is:

```
<device_name><size><data>
```

The `<device_name>` is a character code telling from which device the data is coming; Table 6 shows what kind of information each packet contains. The `<size>` field is an unsigned short telling the size of the `<data>` field in bytes. The `<data>` field contains the data. The format of this field is shown in Table 7.

3.5.1 ACTS Blob Data Format

Player interfaces with the ActivMedia Color Tracking System (ACTS) server; this software can be trained to recognize a variety of colors and it then outputs information about the size and location of any matching blobs that it finds in the camera image. Player currently uses ACTS v1.0; we will eventually upgrade to v1.2[8]

[8] Actually, we are using an interim version that lies somewhere between v1.0 and v1.2. The protocol is that of v1.0, but the number of channels (32) is that of v1.2.

ASCII Code	Name	Description
m (0x6D)	miscellaneous	Bumpers state and battery voltage
g (0x67)	gripper	Gripper data
p (0x70)	position	Time, position, speed, turnrate, heading, compass and stall
s (0x73)	sonar	Sonar data
l (0x6C)	laser	Laser data
v (0x76)	vision	Color blob information from the ACTS server
z (0x7A)	ptz	Encoder feedback from the pan-tilt-zoom camera

Table 6: The data that each device sends.

Player simply forwards blob data from the ACTS server, with no transformations. As such, it is worth explaining the format of an ACTS data packet here. And soon we will. In the meantime, for information regarding the ACTS data format, as well as instruction on using ACTS, consult the ACTS User Manual[1]. Specifically, page 21 in that document gives the data format. Note that number of channels given in that document is wrong; we actually have 32 channels and thus the ACTS header is 64 bytes.

3.6 Configuring the Server

In addition to device requests and actuator commands, there is another set of messages that the client may send to Player. These messages are configuration commands and they allow the client to change certain aspects of the behavior of the Player server by, for example, changing the rate at which sensor data is provided. Similar in structure to device command messages, configuration command messages take the following form:

```
x<device_name><size><config_code><args>
```

The leading single character `x` (0x78) (think "eXpert") tells Player that a configuration command follows. The next character, `<device_name>` denotes which device is being configured. In addition to the regular devices listed in Table 3(a), there is an extra configurable device, specified by the character code `y <config_code>`; this device represents the server itself. Configuration changes made to the `y` device only affect the behavior of the server with respect to the requesting client; changes to the other devices will affect the behavior of those devices with respect to all clients. The `size` field is an unsigned short which tells how many bytes of payload follow. The `config_code` field indicates which configuration request is being made of the device, and the `args` field, if present, is arguments for that request. Even though not all configuration requests have arguments, the `config_code` is always present, so the `<size>` field is always at least 1. The details of the various configuration command messages are given in Table 8.

Player will NOT respond to your configuration requests. This means that with the exception of the `y` device, another client could overwrite your configuration request before it is sent to the device. This is not so bad, since another client could always undo your configuration change at any time without your knowledge.

The default behavior of the server is to operate in continuous mode at a frequency of 10Hz; thus a client which makes no configuration changes will receive new sensor data approximately every 100ms. While this setup is likely to be generally useful for most clients, it will certainly

Device Name	Size	Data Type	Meaning
miscellaneous	3	unsigned byte	Lowest five bits are the front bumpers
		unsigned byte	Lowest five bits are the rear bumpers
		unsigned byte	Battery voltage in decivolts
gripper	2	unsigned byte	bit 0: Paddles open
			bit 1: Paddles closed
			bit 2: Paddles moving
			bit 3: Paddles error
			bit 4: Lift is up
			bit 5: Lift is down
			bit 6: Lift is moving
			bit 7: Lift error
		unsigned byte	bit 0: Gripper limit reached
			bit 1: Lift limit reached
			bit 2: Outer beam obstructed
			bit 3: Inner beam obstructed
			bit 4: Left paddle open
			bit 5: Right paddle open
position	21	unsigned int	Time in milliseconds since server started
		int	X-position in mm (the X-axis is the direction the robot is facing in when the position device is started; positive is forward)
		int	Y-position (the Y-axis is perpendicular to the X-axis; positive is to the left of the robot)
		unsigned short	Heading in degrees with respect to the initial heading. (0 to 360 degrees; increasing counterclockwise)
		short	Speed (mm/sec; positive is forward)
		short	Turn rate (degrees/sec; positive is counterclockwise)
		unsigned short	Compass heading (0 - 360 degrees; increasing counterclockwise)
		unsigned byte	Zero if no motor stall; otherwise non-zero
sonar	32	16 unsigned shorts	Distance readings (in mm) from the 16 sonars. The front sonars number from 0 at the front left around to 7 at the front right. The rear sonars number from 8 at the rear right to 15 at the rear left.
laser	722	361 unsigned shorts	Distance readings (in mm) from the SICK laser rangefinder. Readings number from 0 at the right side of the laser and increase counterclockwise to number 361 at the left side of the laser.
vision	varies	unsigned bytes	Color blob data as read from the ACTS vision system. See Section 3.5.1 for details on the format.
ptz	6	short	Current pan (-100 to 100 degrees; positive is counterclockwise)
		short	Current tilt (-25 to 25 degrees; positive is up)
		short	Current zoom (0 to 1023; 0 is wide and 1023 is telefoto)

Table 7: Format of the `<data>` field in the data packet from each device.

Device	Size	Config Code	Argument (type)	Meaning
y (0x79)	3	f (0x66)	frequency (unsigned short)	Set continuous data rate to given *frequency*
	2	r (0x72)	mode (unsigned byte)	If *mode* is non-zero, put server in request/reply mode; otherwise put server in continuous mode (the default)
	1	d (0x64)	NONE	When server is in request/reply mode, request a single data packet
p(0x70)	2	m (0x6D)	state (unsigned byte)	Set motor power to given *state*; zero is motors off and non-zero is motors on. USE WITH CAUTION!
	2	v (0x76)	mode (unsigned byte)	If *mode* is non-zero, use separate translational and rotational velocity control; otherwise use direct wheel velocity control (the default)
	1	R (0x52)	NONE	Reset robot's odometry to (0,0,0)

Table 8: The configuration commands and what they do.

not suit everyone. Thus the server can serve data faster[9] or slower (via the `f` request of the `y` device) and it can also operate in request/reply mode (via the `r` request of `y` device), in which it waits to receive the configuration request `d` of the `y` device and then responds with a single data packet containing data from all the sensors for which read access has been requested. Note that the preceding configuration commands will only change Player's behavior with respect to your client; that is, the server is independently configured by each client.

Further, the robot, by default, starts up with power to the motors off. Thus, after starting your client program, you have to push the white motor button on the robot to get it moving. The configuration command `m` of the `p` device is given as an alternative; the client can turn the motor power on and off remotely at will. Be VERY careful with this command! You're very likely to start the robot running across the room at high speed with the battery charger still attached.

The Pioneer robot offers two modes of velocity control: separate translational and rotational control and direct wheel control. When in the separate mode, the robot's microcontroller internally computes left and right wheel velocities based on the currently commanded translational and rotational velocities and then attenuates these values to match a nice predefined acceleration profile. When in the direct mode, the microcontroller simply passes on the current left and right wheel velocities. Essentially, the separate mode offers smoother but slower (lower acceleration) control, and the direct mode offers faster but jerkier (higher acceleration) control. Player's default is to use the direct mode; this can be changed with the `v` command of the `p` device.

The robot's odometry is reset to $(x, y, heading) = (0, 0, 0)$ when the first of the microcontroller-mediated devices (currently position, sonar, miscellaneous, and gripper) is opened. Thereafter, until all of these devices are closed, the odometry is integrated based on the robot's motor encoders. The

[9] Requesting data much faster than 10Hz probably won't help you because, at the moment, none of the sensors on the robots generate new data faster than that.

client can reset the odometry to $(0, 0, 0)$ at any time with the `R` command of the `p` device.

Note that for these last three commands (`m`, `v`, and `R` of the `p` device) to have any effect, they must be issued after one of the microcontroller-mediated devices has been opened.

3.7 Summary of Messages

For your convenience, Tables 9 and 10 summarize all the messages used in communicating with Player.

Name	Format	Meaning	Server Response
Device request	`dr<size><device_name><access>`	Request access to a device. Multiple device-access pairs may be concatenated in a single message.	Device response
Device command	`c<device_name><size><command>`	Send `<command>` to `<device_name>`	NONE
Config request	`x<device_name><size><config_code><args>`	Reconfigure server	NONE

Table 9: Summary of Client→Server Messages

Name	Format	Meaning
Sensor data	`<device_name><size><data>`	`<data>` is the latest data from `<device_name>`
Device response	`r<size><device_name><access>`	`<access>` has been granted for `<device_name>`. Multiple device-access pairs may be concatenated in a single message.

Table 10: Summary of Server→Client Messages

4 Architecture

Player was designed from the beginning to be easily extended by adding new devices and by adding new functionality to existing devices. In fact, Player is really a general-purpose device server; we just happen to use it for controlling our robots. You could use it to provide a simple, clean interface to any sensors or actuators you have. We *briefly* describe in the section the overall system architecture and how you would go about adding your own devices. After reading this section, you should consult the code for the existing devices as examples for writing your own.

4.1 Server Structure

Player is implemented in C++ and makes extensive use of the POSIX-compliant pthread interface for writing multi-threaded programs. The choice to create such a heavily multi-threaded server was guided by our desire to concurrently support many heterogeneous devices and many heterogeneous clients. Each device operates at some inherent frequency, and that frequency can vary wildly across different devices; for example, the SICK laser rangefinder returns a full scan at approximately 5Hz, while the Sony PTZ camera can give encoder feedback at almost 2500Hz. If we were to take the naive approach and poll each device in turn at the rate of slowest device, we would be discounting the full capabilities of the available resources. In most cases, what we want is to obtain data from and send commands to each device at the highest rate possible in order to fully exploit the hardware and maximize the responsiveness of the system. Analogously, each client operates at some inherent frequency; while a simple client written in C++ may be capable of consuming new data at 100Hz, a graphically intensive client written in Tk might operate at less than 1Hz. It should be possible for these two clients to be connected to Player simultaneously and to execute as fast as they want without interfering with each other.

Given the requirement to support interaction with external entities (i.e. clients and physical devices) that inherently operate at a variety of timescales, we designed Player as an asynchronous system. We achieved this asynchrony through the use of threads, which we spawn on demand, as shown in Figure 1. The main thread listens for new client connections on the selected TCP port. When a new connection is accepted, the main thread spawns two threads to service that connection and returns to listening for other connections. The two spawned threads, which we refer to as the client reader and client writer threads, read from and write to the client, respectively. When a client reader thread receives a request for a device that is not already setup, that thread calls the proper method (`Setup()`) in the object which controls the indicated device. The invocation of `Setup()` involves spawning yet another thread to communicate with that device[10]. So, in total, we have 1 main thread, 2 threads per client, and 1 thread per open device. While this may sound like a lot of overhead, it is important to remember that these threads all use blocking I/O and so spend almost their entire lives blocked, either on a client, a device, or a timer. We have found in practice no delay even when running Player with the maximum number of clients (currently 8), each connected to all the devices.

The overall system structure of Player is shown in Figure 2. The center portion of the figure is Player itself; on the left are the physical devices and on the right are the clients. As described above, each client has a TCP socket connection to Player. If the client is executing on the same host as Player, then this socket is simply a loopback connection; otherwise, there is a physical network in between the two. At the other end, Player connects to each device by whatever method is

[10]The only exception is for the sonar, miscellaneous, and position devices; they are simply logical divisions of what is physically a single device: the P2OS microcontroller. Thus, only the first of these devices that is opened actually calls `Setup()` and spawns the thread that talks to P2OS.

appropriate for that device. For most devices, including the laser, camera, and robot microcontroller, Player makes this connection via an RS-232 serial line. However, the connection to the ACTS vision server is via a TCP socket.

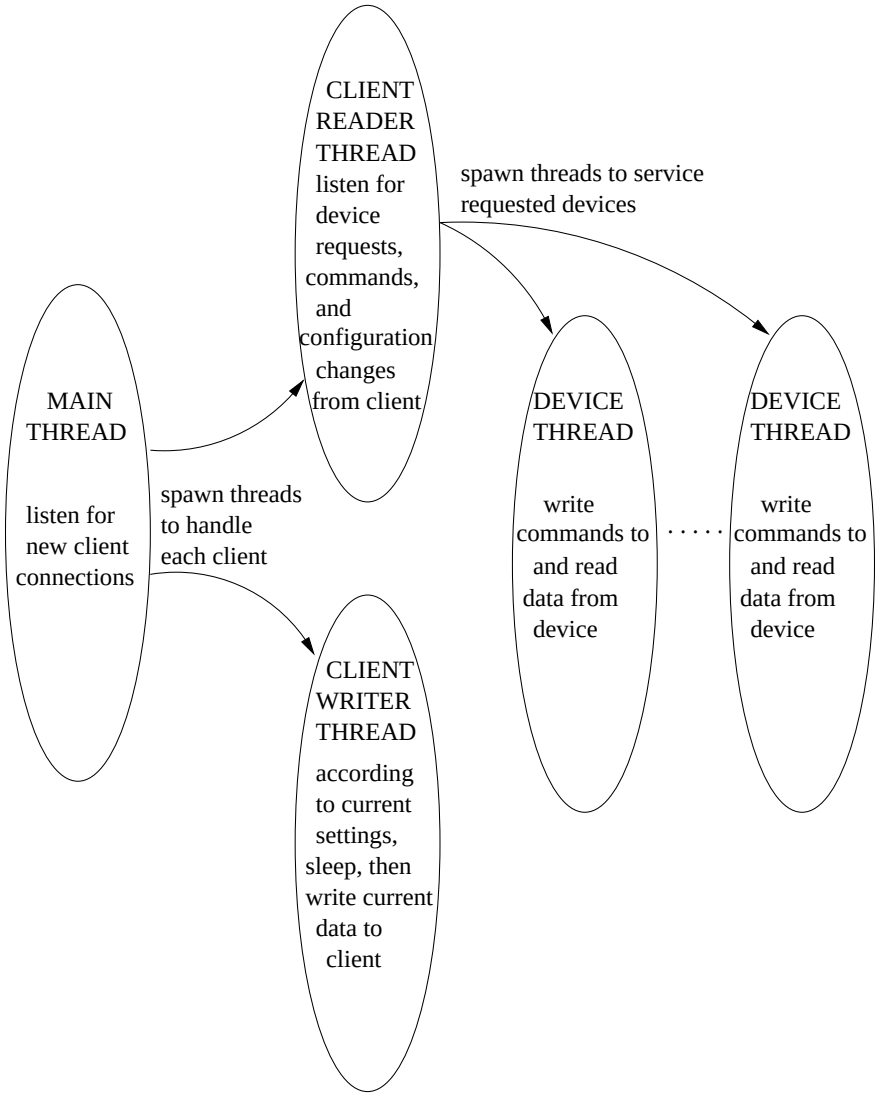

Figure 1: *Dynamic thread hierarchy of Player*

Within Player, the various threads communicate through a shared global address space. As indicated in Figure 2, each device has associated with it a command buffer and a data buffer. These buffers, which are each protected by mutual exclusion locks, provide an asynchronous communication channel between the device threads and the client reader and writer threads. For example, when a client reader thread receives a new command for a device, it writes the command into the command buffer for that device. At some later point in time, when the device thread is ready for a new command, it will read the command from its command buffer and send it on to the device. Analogously, when a device thread receives new data from its device, it writes the data into its data buffer. Later, when a client writer thread is ready to send new data from that device, it reads the data from the data buffer and passes it on to its client. In this way, the client service threads are decoupled from the device service threads (and thus the clients are decoupled from the devices).

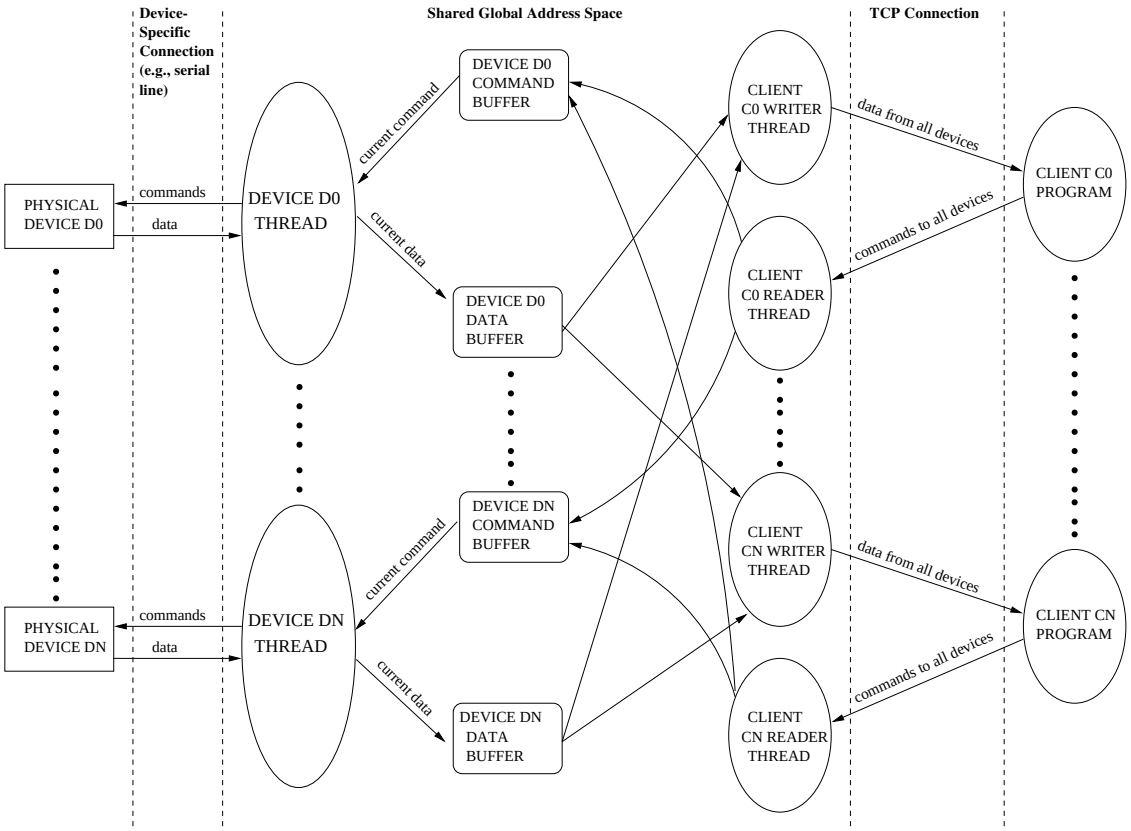

Figure 2: *Overall system architecture of Player*

Also, just by the nature of threads, the devices are decoupled from each other, and the clients are decoupled from each other.

One disadvantage to the decoupling of client from device is that there is no provision to inform the client of the "freshness" of some device's data. That is, a client writer thread will write new data for a device to its client whenever it is asked (either by a timer or by the client itself), independent of whether the device has actually produced new data. For example, with the default server configuration, a client will receive new data at 10Hz[11]; since the laser only produces a new scan at about 5Hz, the client will receive an old laser scan about half the time, and there is no way for the client to know the difference between an old scan and a new one. Similarly, the ACTS vision system can sometimes produce new data at almost 30Hz; in the default configuration, the client only receives every third set of vision data. We designed Player in this way for one reason: simplicity. By always transmitting the current state for all requested devices, regardless of the timescale of the device, we facilitate the writing of client programs. As a result, clients are able to use a simple blocking read loop to receive data from Player. If the client is multi-threaded (many are), the blocking read could be compartmentalized to a single thread, allowing the rest of the client program to proceed unhindered.

Of course, receiving data at 10Hz may not be reasonable for all clients; for these situations, we provide a method for changing the frequency, and also for placing the server in a request/reply mode (see Section 3.6). So, if a client wants the vision data at full frame rate, it can configure Player to send data at 30Hz, with the tradeoff it will also receive data from the other currently requested

[11] We chose this default because it is the maximum rate at which we can command the wheel motors on our robots.

devices at the higher rate. Alternatively, if there is a low-bandwidth connection between Player and a client using laser data (which is comparatively large), that client might lower the data rate to 5Hz in order to minimize message-passing and thus conserve bandwidth, with the tradeoff that data from other requested devices will also arrive more slowly. It is important to remember that even when a client receives data slowly, there is no backlog and it always receives the most current data; it has simply missed out on some intervening information. Also, these frequencies changes affect the server's behavior with respect to each client individually; the client at 30Hz and the client at 5Hz can be connected simultaneously, and the server will feed each one data at its preferred rate.

Having justified our design decisions regarding Player's data transfer paradigm, we do realize the utility of a mode of server operation in which a client only receives fresh data; we are currently investigating methods for implementing such a feature. The default behavior of the server will remain the same, so only those clients who need the freshness guarantee will be burdened with the extra complexity of reading data which arrives at the natural rate of each device.

Analogous to the issue of data freshness is the fact that there is no guarantee that a command given by a client will ever be sent to the intended physical device. Player does not implement any device locking, so when multiple clients are connected to a Player server, they can simultaneously write into a single device's command buffer. There is no queuing of commands, and each new command will overwrite the old one; the service thread for the device will only send to the device itself whatever command it finds each time it reads its command buffer. We chose not to implement locking in order to provide maximal power and flexibility to the client programs. In our view, if multiple clients are concurrently controlling a single device, such as a robot's wheels, then those clients are probably cooperative, in which case they should implement their own arbitration mechanism at a higher level than Player. If the clients are not cooperative, then the subject of research is presumably the interaction of competitive agents, in which case device locking would be a hindrance.

4.2 Extending the Server

Having described the internal workings of Player, we now give a short tutorial on how you would go about extending the server by adding a new device. As mentioned earlier, in lieu of a more complete prescription for creating devices, an examination of the code for the existing devices should provide you with sufficient examples. You should be familiar with C++, class inheritance, and thread programming.

The first step in adding a new device to Player is to create a new class for the device, which should inherit from `CDevice`, declared in `device.h`. That base class defines an interface that the new device must implement. Apart from the interface functions, you must also write a thread that will service the device when it is needed. There are a few subtleties here, including the use of a lock object of type `CLock` for protecting the data and command buffers; check existing code for guidance. After creating the class and thread for the new device, you need only modify `main.cc`. You should `#include` the header for your new class, and declare a global instance of it. If your device requires new command-line arguments (such as for specifying which to port the device is connected), add them to the argument parsing loop. Next, call `AddDevice()` with the appropriate identifying character for your device, the access mode that is possible (`'r'` for read-only, `'w'`, for write-only, or `'a'` for read-write), and the global instance of your class that will be used to control the device. For example, to add the laser device to the set of available of devices, we call:

```
deviceTable->AddDevice('l', 'r', laserDevice);
```

Finally, modify the `Makefile` to add your new device driver to the list of objects to be linked into Player, do `make` and you're done.

A The C++ Interface

A.1 Hello World

This is a short tutorial that explains how to use the Player robot server using the provided interface written in C++. The source code is available in playerclient.h and playerclient.cc and is included in the Player distribution.

The source code shown in Figure 3 is a complete program that makes the robot move around doing very limited obstacle avoidance. Let's go through the code step by step to get a feeling for what it does. All the code needed to interact with the Player server is wrapped into the C++ class CRobot.

The method `Connect("tanis")` sets up the connection to the server running on the host "tanis" using TCP/IP. This method will return non-zero if it fails to setup the connection; a meaningful error statement will be printed to stderr. After that we call `Request( "srpw" )`. This request tells Player what data we need access to. In this case we want read (`r`) access to the sonars (`s`) and write (`w`) access to the position device[12]. This tells the server to start sending sonar data to the client continuously[13] and accept position commands from the client. The program now goes into a read-act-write loop for one thousand iterations. `Read();` blocks and waits for the server to send the sonar data. The sonar data is then printed to the terminal using the `Print()` method. The client calculates and sets the variables `newturnrate` and `newspeed` in the robot object based on the sonar data. Finally `Write()` is called. This method writes the commands that the client has permissions to write to the server; in this case the newspeed and newturnrate. After the one thousand iterations has been reached another request is send. This time the request is to close ('c') the access to the position and sonar device. That's it.

A.2 Connecting

The connection is created using the `Connect` method. There are actually three forms of this method:

```
int CRobot::Connect();
int CRobot::Connect(char* host);
int CRobot::Connect(char* host, int port);
```

The first form, with no arguments, will connect to the Player server running on the host and TCP port specified by the public (and thus user-modifiable) fields `CRobot::host` and `CRobot::port` (defaults are "localhost" and 6665, respectively). The next two forms allow the caller to override the host and port.

If there is any error, `Connect()` will return non-zero, and the connection to the server has not been established. Otherwise, it returns zero.

A.3 Request

The request is made using the `Request()` method. See Section 3.3 for details on the format of the request string (which must be NULL-terminated).

```
int CRobot::Request(char* request_string)
```

[12] This is the device used to control the robots wheels

[13] The default rate is 10Hz.

```
#include <playerclient.h>

int main(int argc, char *argv[]) {
  CRobot robot;

  // where tanis is the hostname of the robot
  if(robot.Connect("tanis"))
    exit(1);

  if(robot.Request("srpw"))
    exit(1);

  for(int i=0;i<1000;i++)
  {
    if(robot.Read())
      exit(1);

    robot.Print();

    if((robot.sonar[0] + robot.sonar[1]) <
       (robot.sonar[6] + robot.sonar[7]))
      robot.newturnrate = -20; // turn 20 degrees per second
    else
      robot.newturnrate = 20;

    if(robot.sonar[3] < 500)
      robot.newspeed = 0;
    else
      robot.newspeed = 100;

    robot.Write();
  }

  /* close everything */
  robot.Request("scpc");

  return(0);
}
```

Figure 3: "Hello world" code for the Player robot server.

If there is any error, `Request()` will return non-zero; some parts of the request may have been accepted and processed by the server, but certainly at least one part was not. Otherwise, it returns zero.

A.4 Read

After requesting access to some sensors, the user can call the following method:

```
int CRobot::Read()
```

This method will block until it has received data from all the sensors for which read access was requested and granted. If there is some error, it will return non-zero; otherwise it will return zero and the received data is placed into various data structures in the `robot` object (see the source code for more details):

```
struct CGripper {
  /* gripper attributes */
  unsigned char paddlesOpen;
  unsigned char paddlesClose;
  unsigned char paddlesMoving;
  unsigned char paddlesError;

  unsigned char leftPaddleOpen;
  unsigned char rightPaddleOpen;

  unsigned char gripLimitReached;

  unsigned char liftUp;
  unsigned char liftDown;
  unsigned char liftMoving;
  unsigned char liftError;

  unsigned char liftLimitReached;

  unsigned char outerBeamObstructed;
  unsigned char innerBeamObstructed;
};

struct CMisc {
  /* misc attributes */
  unsigned char frontbumpers;
  unsigned char rearbumpers;
  unsigned char voltage;
};

struct CPtz {
  /* ptz attributes */
  short pan;
```

```
  short tilt;
  short zoom;
};

struct CPosition {
  /* Position information */
  unsigned int time;
  int xpos, ypos;

  unsigned short heading;
  short compass;
  short speed, turnrate;
  unsigned char stalls;
};

struct CBlob {
  /* data for a single color blob */
  unsigned int area;
  unsigned char x;
  unsigned char y;
  unsigned char left;
  unsigned char right;
  unsigned char top;
  unsigned char bottom;
};

struct CVision {
  /* this array, indexed by channel, tells the number of blobs detected
   * on that channel*/
  char NumBlobs[ACTS_NUM_CHANNELS];

  /* this array, indexed by channel, contains the blob data for each
   * blob detected on that channel */
  CBlob* Blobs[ACTS_NUM_CHANNELS];
};

class CRobot {
 /* (some parts left out here) */
 public:
  /* data from robot */

  /* sonar data is a 16-element array
   * the sonars number from 0 at the front left of the robot clockwise
   * around to number 15 at the back left */
  unsigned short *sonar;

  /* laser data is a 361-element array
```

```
    * the laser scans number from 0 at the right side of the laser
    * counterclockwise to 361 at the left side */
  unsigned short *laser;

  CPosition *position;
  CVision *vision;
  CGripper *gripper;
  CMisc *misc;
}
```

A.5 Command

The user is protected from all the byte manipulation needed to send commands. The user sets variables in the class corresponding to the commands and then when all command variables have been set the method `Write()` is used. `Write` checks what write permissions the client has and write the commands the client is allowed to write to Player.

```
class CRobot {
  /* (some parts left out here) */
  public:
  /* commands to robot */

  /* new forward velocity, mm/sec, positive is forward */
  short newspeed;

  /* new turnrate, deg/sec, positive is counterclockwise */
  short newturnrate;

  /* new absolute camera positioning
   *   pan  : -100 to 100 degrees, positive is counterclockwise
   *   tilt : -25 to 25 degrees, positive is up
   *   zoom : 0 to 1023, 0 is wide and 1023 it telefoto */
  short pan;
  short tilt;
  short zoom;

  /* new gripper command (see below) */
  short newgrip;
}
```

To aid the user in controlling the gripper, the following method is provided:

```
void CRobot::GripperCommand( unsigned char command )
void CRobot::GripperCommand( unsigned char command, unsigned char value )
```

This method is used to set the command field `newgrip` properly[14]. The first form is used when the desired command requires no argument; the second form is used for gripper commands that do require an argument. Examples (see the Gripper manual and read the source for more details):

[14]Note that `GripperCommand()` does not send any commands to the robot; the user must still call `Write()`.

```
robot.GripperCommand( GRIPPress, 20 );
robot.GripperCommand( GRIPclose );
```

A.6 Configuration

The CRobot class supports the following methods for configuring the server:

```
int CRobot::SetDataMode(data_mode_t mode);
int CRobot::SetFrequency(unsigned short freq);
int CRobot::RequestData();
int CRobot::ChangeMotorState(unsigned char state);
int CRobot::ResetPosition();
int CRobot::ChangeVelocityControl(velocity_mode_t mode);
```

These methods should only be called after a successful call to `Connect()`. In all cases, they return non-zero on error (the configuration change was not made) and zero otherwise.

The argument to `SetDataMode()` should be either `REQUESTREPLY` to put the server into request/reply mode or `CONTINUOUS` to put it into continuous mode. When in request/reply mode, `RequestData()` requests one round of sensor data from the server. When in continuous mode, `SetFrequency()` allows the user to set the continuous data rate to any frequency (in Hz). The default behavior of the server is to operate in continuous mode at a frequency of 10Hz.

Whereas the three previous methods will have their expected effect if they are called any time after a call to `Connect()`, the following three methods will only make a change if one of the microcontroller-mediated devices (currently position, sonar, miscellaneous, and gripper) has already been successfully requested. If the argument to `ChangeMotorState()` is zero, the motors are disabled (powered off); otherwise they are enabled (be VERY careful when turning the motors on from software!). The `ResetPosition()` method takes no arguments, and simply resets the robot's odometry to $(x, y, heading) = (0, 0, 0)$. The argument to `ChangeVelocityControl()` method should be either `DIRECTWHEELVELOCITY` to use direct-wheel velocity control or `SEPARATETRANSROT` to use separate translational and rotational velocity control; the default is to use direct-wheel velocity control.

B The Tcl/Tk Interface

Documentation on the Tcl/Tk client utilities will be available soon. In the meantime, consult the provided example programs (especially `viewer.tk`), and refer to the code itself (`playerclient.tcl`) when in doubt. Hopefully, the usage of the utilities will be straightforward for someone familiar with Tcl.

C The Java Interface

The JAVA-example[15] is in three files.

- **PlayerClient**, a class file that holds methods for making communication with the Player server easier.
- A file with an example program which uses the PlayerClient class for doing obstacle avoidance using the laser.
- An example program that uses the PlayerClient class for doing obstacle avoidance using the sonar.

Both of the obstacle avoidance programs work by getting the minimum distance for the sensors in each side of the robot, and then using this information to determine speed and rotation. The code should be self-explanatory, and will not be described in this text.

The **PlayerClient**-class is a ready-to-use example of a library for making communications with the pioneers easier. The **PlayerClient**-class will be briefly described in the next section.

C.1 The PlayerClient-class

The intended use of the **PlayerClient**-class is as follows:

```
1:  PlayerClient <robotName>=new PlayerClient(<pioneerName>,<portNumber>);
```

This line sets up communications to `<pioneerName>` on port number `<portNumber>`. This connection will from now on be available through the object `<robotName>`.

The second thing to do, is to request for devices used by the program.

```
2: <robotName>.request(``<requestString>'');
```

Where `<requestString>` is of the format described in section 3.3. This line determines which data will be available to this control program, and which motors will be accessible.

Now the communication between the Pioneer and your program is set up, and the Pioneer is ready for taking commands. The **PlayerClient**-class is intended to be used in a 10Hz cycle as shown below;

```
3: while (!<stopCondition>) {
4:   <robotName>.getData();
5:   <--- do stuff --->
          ....
x: }
```

This code will cause the program to run until the `<stopCondition>` is true. The `getData()` method called in line 4 will read all data requested in line 2, and store the data in data structures of the `<robotName>`-object. The `getData()` method does a blocking read, which causes the entire control loop (lines 3-x) to run at 100ms intervals, since the default update rate from the Player server is 10Hz.

[15] These Java client utilities were written by Esben Østergård and Jakob Fredslund; questions and comments should be sent to: `esben@robotics.usc.edu` or `jakobf@robotics.usc.edu`.

C.2 Motor commands

From line 5 your code fits in. Three methods are available for writing motor commands:

```
<robotName>.setSpeed( <translation>, <rotation> )
<robotName>.setVision( <pan>, <zoom>, <tilt> )
<robotName>.setGripper( <grippercommand> )
```

It is not possible to write motor commands for the devices for which write access has not been requested and accepted.

C.3 Sensor data

Sensor data for a device is available if read access has been requested and granted for that device. The data is structured like this:

```
/** Camera image data */
public class ColorChannel {
   public short noBlobs; // number of blobs in this color channel
   public ColorBlob[] blob = new ColorBlob[ACTS_MAX_NO_BLOBS];
}
public class ColorBlob {
   int area;
   short x, y, left, right, top, bottom; // really only unsigned bytes.
}
public int howManyColorsInCameraImage()
public ColorChannel[] channel = new ColorChannel[ACTS_NUM_CHANNELS];
public int[] activeChannel = new int[ACTS_NUM_CHANNELS];


/** Camera feedback data */
public class CameraFeedback {
  short pan, tilt, zoom; // camera feedback
}
public CameraFeedback camfeedback= new CameraFeedback();

/** Sonar */
public short sonar[]=new short[16];

/** Laser */
public short laser[]=new short[361];

/** Position data */
public int time,x,y;
public short heading,speed,turnRate,compass;
public boolean stalled=false;
```

```
/** Gripper data */
public byte gripperData1, gripperData2;

/** Misc. data */
public short battery;
public boolean frontBumper[] = new boolean[5];
public boolean backBumper[] = new boolean[5];
```

The camera image data comes as blobs in different colors, or channels. The ACTSServer camera system has 32 channels (`ACTS_NUM_CHANNELS`) and in each channel it will find a maximum of 10 blobs (`ACTS_MAX_NO_BLOBS`). The `PlayerClient` object has an array (`channel[]`) of 32 `ColorChannel` objects. Each such object holds the number of blobs currently visible in that color and an array (`blob[]`) of up to 10 `ColorBlob` objects. Each such object holds the area, the x and y coordinates of the blob's center, and the left, right, top, and bottom coordinates of its bounding box. To get the area of blob number 1 (they are numbered from 0, the 0th one being the biggest) in color channel 2, you could write

```
if (<robotName>.channel[2].noBlobs() > 1) // otherwise undefined
   int blobarea = <robotName>.channel[2].blob[1].area;
```

Also, the `PlayerClient` object has an array, `activeChannel[]`, that holds (from entry 0 and up) the indices of the channels that have at least one color blob. This means that you don't have to write code to find out which of the 32 colors have actually been found in the image; these are listed in `activeChannel[]`. E.g., you could write

```
if (<robotName>.howManyColorsInCameraImage() > 0)
   int blobarea = <robotName>.channel[<robotName>.activeChannel[0]].blob[0].area;
```

to get the area of the biggest blob of the first active color in the list. The number of active colors, i.e. the length of the portion of the array `activeChannel[]` actually used, can be found with the method `howManyColorsInCameraImage()`.

The camera sends feedback about its actual pan, tilt, and zoom position. This data is available as three `short` fields in the `camfeedback` object of `PlayerClient`, used as, e.g.,

`<robotName>.camfeedback.tilt`.

The bumper data is organized in two arrays numbering the front bumpers 0-4 clockwise from left to right, and the back bumpers 0-4 clockwise from right to left: `<robotName>.frontbumper[0]` is `true` if the left front bumper is pressed; `<robotName>.frontbumper[4]` is `true` if the right front bumper is pressed; `<robotName>.backbumper[0]` is `true` if the *right* back bumper is pressed; and `<robotName>.backbumper[4]` is `true` if the *left* back bumper is pressed.

For more information, look through the source file of the **PlayerClient**-class. See the two example programs for concrete information about how to use the **PlayerClient**-class.

C.4 JAVA example, Obstacle avoidance using sonar

```
public class ObstacleAvoidanceUsingSonar {
    static public void main(String args[]) {

// Setup communications with robot
PlayerClient ant=new PlayerClient("localhost",6665);

// Setup requests for 'ant'
ant.request("srpw"); // (Sonar, Read), (Position, Write)

// Do obstackle avoidance for 400 time steps (40 sec.)
int minR, minL;
for (int i=0; i<400; i++) {
    ant.getData(); // Blocking read of data
    minL=Integer.MAX_VALUE; minR=Integer.MAX_VALUE;
    for (int j=1; j<4; j++) {
if (minL>ant.sonar[j]) minL=ant.sonar[j];
    }
    for (int j=4; j<7; j++) {
if (minR>ant.sonar[j]) minR=ant.sonar[j];
    }
    int l=(100*minR)/300-100;
    int r=(100*minL)/300-100;
    if (l>150) l=150; if (r>150) r=150;
    ant.setSpeed(r+l,r-l);
}
    }
}
```

References

[1] ActivMedia Robotics. *ACTS User Manual.* http://robots.activmedia.com/docs/all_docs/ACTSman.pdf.

[2] ActivMedia Robotics. *Pioneer 2 Gripper Manual.* http://robots.activmedia.com/docs/all_docs/gripmanP2_3.pdf.