Generic interfaces for robotic limbs

Geoffrey Biggs and **Bruce MacDonald**
Department of Electrical & Computer Engineering, University of Auckland

Email: g.biggs at ec.auckland.ac.nz, b.macdonald at auckland.ac.nz

Abstract

Generic interfaces to robot hardware can improve the programming process by making it easier for developers to port applications between different robotic systems. The Player project provides abstract interfaces to aide porting, but until recently did not have any direct support for robotic limbs. This paper describes the design and implementation of three new generic interfaces for robotic limbs and grippers: a low-level interface for controlling limb joints, a high-level interface for controlling end-effector pose, and an interface for controlling grippers. The design of the interfaces allows robot applications to quickly be ported between different robotic limbs and different robots. The interfaces are implemented in the Player project, with support provided in the driver for Pioneer robots, allowing the interfaces to be used to control the Pioneer robot arm and gripper. Other drivers are expected to be developed in the future, providing support for other robotic limbs.

1 Introduction

Like mobile robots, robotic limbs are finding increasing use in service applications. Limbs are used to manipulate the world around the robot, in the form of arms with manipulating end effectors such as grippers, and to move the robot around the world, in the form of legs. Without limbs, robots would be considerably less capable of interacting with the world around them. However, as with all types of robot hardware, the need to use limbs brings with it the need to create software to control them. This paper describes generic interfaces developed for the Player project for controlling robotic limbs.

Previously Player did not provide direct support for any form of robotic limb and only had limited support for robotic grippers. Three new interfaces were created for these types of hardware: an actuator array interface to provide low-level control of the joints of robotic limbs, a limb interface to provide end effector position control, and a gripper interface to provide control of robotic grippers. These are described in Sections 3, 4 and 5. A comparison with other software interfaces to robotic limbs is given in Section 6.

The requirements for the interface designs were developed with help from the Player user and developer communities using Player's development mailing list[1]. Feedback from other Player users on the needs of the interfaces and needs of users led to improvements in the designs and the final, implemented interfaces.

2 The Player Project

Player is a distributed architecture for robot software based on the producer/consumer concept [Vaughan *et al.*, 2003]. It provides network transparency and hardware abstraction, with the goal of allowing software written for one robot to work on a robot with similar capabilities, even if the hardware or software implementation of that robot is very different. In its simplest form, it can be used for simple client/server programs.

The Player server provides access to robot hardware and the interface to client software. Robot hardware is controlled through a set of message interfaces that represent common robot functionality, such as a laser scanner, a position controller or a path planner. These interfaces provide the network communications protocol for the Player server. Drivers for specific hardware or for specific implementations of software algorithms are used to interface between robot hardware and software algorithms and the Player server [Collett *et al.*, 2005].

Client programs use the Player client libraries to access robots controlled by a Player server. There are client libraries for C, C++ and Java, and bindings for Python

[1] See `http://sourceforge.net/mailarchive/message.php?msg_id=12241168`

are automatically generated based on the C client library. The producer/consumer structure means that multiple clients written in different languages can connect to a single Player server at the same time. The client libraries hide the network interface from client software to provide a more useful robotics API. Messages are sent from client programs to drivers on the server to control robots, and messages are sent from drivers to client programs to provide data, for example sensor readings or state information.

Player uses interfaces to define how to interact with different types of hardware and software components. Each interface defines all the messages that a driver controlling a piece of hardware or software that supports its capabilities will understand and provide. There are three important message types:

Data messages are broadcast from drivers to clients to provide new information on, for example, the driver's state.

Command messages are sent from clients to a driver, instructing it to perform an action.

Request messages are sent from clients to a driver, usually to request a change in the driver's configuration or request some information. Unlike data and command messages, request messages expect to receive a response, which may indicate the success or failure of the request or contain the information requested.

In all cases, the client may be another driver or a client program written using one of the client libraries.

3 Actuator array interface

Robotic limbs can be viewed as an array of linear or rotary actuators, where linear actuators change the length of the link to the next actuator (for example, a hydraulic piston) and rotary actuators rotate about a joint axis. For example, a simple robot arm with one degree of freedom shoulder, elbow and wrist joints can be seen as an array of three rotary actuators.

The actuator array interface provides this low-level, direct control over the joints of robotic limbs. It was necessary to separate this functionality from the functionality of limb interface described in Section 4 because not all limbs will provide both interfaces, and because combining them would lead to an overly complex interface.

The actuator array supports both linear and rotary actuators. It allows the position of each actuator in the array to be set, or alternatively allows each actuator to be commanded to move at a given speed, in order to provide sufficient flexibility of control. It also enables moving actuators to a known "home" position, usually where it is safe to switch off the actuators. Clients must get data back describing the current position, speed and state of each actuator in the array. Finally, the correctly designed interface must provide enough information, including the geometry of the actuators in the array, for clients to implement their own inverse kinematics and make other calculations about the real robot arm.

Table 1: The messages of the actuator array interface.

Type	Name	Parameters
Data	State	(position, speed, state) for each actuator, number of actuators
Command	Position	Actuator number, position
	Speed	Actuator number, speed
	Home	Actuator number (-1 for all actuators)
Request	Geometry	Base pose, (type, offset from previous actuator, orientation, axis, minimum, centre, maximum, home position, configured speed, hasBrakes) for each actuator, number of actuators
	Power	Boolean – off or on
	Brakes	Boolean – off or on
	Speed	Actuator number, speed

The messages of the designed interface are shown in Table 1. Actuator states may be idle, moving, braked or stalled (blocked from moving to the requested position). Units for position values are either metres or radians, depending on whether the actuator is linear or rotary. Speeds are measured in units per second. All values are in each joint's coordinate space. It is important to note that this interface attempts to provide all the information necessary for clients to make local kinematics calculations.

A reference implementation of a driver supporting the actuator array interface was embedded in the multi-interface Player driver for Pioneer robots [MobileRobots, 2006]. It provides joint-level control over the Pioneer's 5DOF robotic arm. This arm, shown in Figure 1, has five degrees of freedom for controlling the end effector position and a gripper on the end of the arm.

The arm is controlled via the Pioneer's onboard controller, which receives commands over its serial or TCP link with the Player server in the form of a single byte value between 0 and 255, indicating the position of the servo that is being commanded. The actuator array driver converts these byte values into radians assuming the origin is at each servo's centre point, and using calibration data received from the onboard controller. It converts command values back from radians to the byte values before they are transmitted to the onboard controller.

The actuator array interface provides a device-

Figure 1: The 5DOF robotic arm for Pioneer 3-DX robots.

independant method of controlling the individual joints of robotic limbs at a low level. Software written using the interface can be moved between different hardware with ease, assuming the hardware provides the same configuration of joints. If the joint configuration changes, it is simpler to move code to the new hardware as a new method of controlling the joints does not need to be learned. The interface does not abstract away the need to alter software to handle changes in the joint configuration. The interface also has potential applications beyond the standard robotic limb. A snake robot, for example, is often constructed as an array of actuators.

4 Limb interface

Section 3 described the actuator array interface, used for individual joint control of a robotic limb. This section describes the high level interface used to control a limb with a built-in kinematics controller (in the driver or in the limb itself) or some other form of built-in direct control over the end position of the limb.

A robotic limb could be an arm with a gripper on the end, or it could be the leg of a humanoid robot. The limb interface is therefore designed to control the position and orientation of the end point of a robotic limb, whether this is an end effector or simply the end of the limb. It assumes the driver providing the interface will manage individual joint angles suitable to the hardware the driver is controlling.

To support a range of limb types, the interface must set both the position of the end of the limb without care for the orientation and for setting the position and the approach and orientation vectors of the end of the limb.

Table 2: The messages of the limb interface.

Type	Name	Parameters
Data	State	End effector position, approach vector, orientation vector, limb state
Command	SetPose	End effector position, approach vector, orientation vector
	SetPosition	End effector position
	VectorMove	Direction vector, length of move
	Home	–
	Stop	–
Request	Geometry	Position of limb's base
	Power	Boolean – off or on
	Brakes	Boolean – off or on
	Speed	Speed

The vectors are necessary for limbs with an end effector such as a gripper in order to approach an object to be grasped from the correct direction and at the correct orientation. To further support limbs in this style of task, the interface also must provide a "vector move," where the end effector is moved along a straight line without change in its orientation. This is used to move into the grasping position for picking up objects, for example. Finally, the interface must be able to send the limb to a safe "home" position and to stop the limb at its current position as a safety measure, for example, to stop the limb when obstructed.

The messages of the designed interface are shown in Table 2. The limb's state may be idle (awaiting commands), braked, moving, out-of-reach (indicates that the limb cannot move the end effector to the commanded pose or position due to constraints in its degrees of freedom), or in collision (indicates that the limb cannot move to the commanded pose or position because of an obstruction). The interface itself does not specify which coordinate space should be used for end effector geometry.

A driver has been implemented for the Pioneer arm, including forward and inverse kinematics via the limb interface, as part of the multi-interface Pioneer driver. The driver uses the kinematics calculations described by Gan *et al.* [2005]. Since this calculator requires full pose information, in the form of position plus vectors, the driver does not support the *SetPosition* command of the interface. It also does not currently support the *VectorMove* command. The driver uses the robot's coordinate space for all position information, so the end effector pose is set relative to the robot's origin.

The driver interfaces directly with the Pioneer's onboard controller, rather than going through the existing actuator array interface. This is done in the same way

Table 3: The messages of the original, Pioneer-specific gripper interface.

Type	Name	Parameters
Data	State	Gripper state (bit-mask), breakbeams state (bit-mask)
Command	Command	Command byte, argument byte
Request	Geometry	2D pose, 2D size

as the actuator array communicates with the onboard controller. See Section 3 for details.

The limb interface provides portability even beyond that of the actuator array interface. Because all that is important is the pose of the end of the limb, software can be moved between different hardware with ease, even if the configuration of the hardware changes. For example, two different robotic arms with sigificantly different joint configurations but with a similar reachable space could use identical code in client software, relying on the individual drivers to manage device-specific changes such as differences in kinematics calculations. In some applications, such as the arms of a service robot, changes in the reachable space of the arm in use may not be important because of the portability of the arm's base.

5 Gripper interface

The original Player gripper interface, shown in Table 3, was effectively just a pass-through interface to the gripper commonly used on Pioneer robots. It was capable of passing the byte commands used by this gripper's controller directly from a client program through to the robot. As such, it was not standard and could not be used to control any other robot grippers.

To rectify this, a new, standard gripper interface has been designed. All the functionality of the existing interface is available, as well as some new functionality. The gripper must be capable of opening, closing, and of providing information on where in the gripper an object is positioned, to allow clients to know when to close the gripper, a function provided by the *breakbeams* value in the original interface.

The Pioneer gripper includes a lift system, used to raise and lower the gripper in order to lift objects off the ground for carrying. However, because this functionality can be provided more suitably by the actuator array interface, it was left out of the new gripper interface.

One interesting consideration when designing the new interface was accommodating the wishes of artificial intelligence researchers who use the Player project's Stage simulator. The Pioneer gripper features a command to "store" the gripper, in other words to move it to a suitable position for driving the robot around. AI researchers had been using this command to cause the gripper to "swallow" whatever it was carrying, a useful function in AI simulations. In order to replicate this ability, and in consideration for the possibility of real gripper devices that could perform a similar function, the ability was added for storing and retrieving carried objects in some kind of repository in the gripper device.

Table 4: The messages of the new, standardised gripper interface.

Type	Name	Parameters
Data	State	Gripper state, breakbeams state, number of stored objects
Command	Open	–
	Close	–
	Stop	–
	Store	–
	Retrieve	–
Request	Geometry	3D Pose, outside dimensions (3D), inside dimensions (3D), number of breakbeams, storage capacity

It was decided to keep the use of breakbeams to indicate the position of objects in the gripper, as this is the standard method.

The messages of the designed interface are shown in Table 4. The gripper state may be open, closed, moving, or error to indicate some problem has occurred with executing the last command.

The new gripper interface was also added to the multi-interface Pioneer driver. A new Pioneer gripper interface was added, replacing the the one offered by the original gripper interface, and a second interface was added for the gripper on the end of the Pioneer robotic arm. Both grippers have a storage capacity of zero, and so do not support the *Store* and *Retrieve* commands.

The lift functionality of the original gripper interface was replaced with another actuator array interface. This particular instance of the interface is limited to just one linear actuator, and can only move between positions 0cm and 7cm. In this way, the functionality of the original gripper interface is preserved.

The lift hardware does not provide the current position of the lift, although it does allow the time taken for raising or lowering to be changed. This does not guarantee position accuracy as the distance moved in this time will change depending on the load being carried by the gripper. Position values of 0cm correspond to "down" and values of 7cm correspond to "up." Values between 0cm and 7cm are used to calculate an estimated travel time to reach that position, giving an approximate form of position control.

The gripper interface does not include information about the configuration of the gripper's fingers. This

is both a disadvantage and a benefit. It means that individual control of a gripper's fingers is not possible through the interface even if the gripper itself provides such control. However, it also means the interface is simpler. Knowing the available space inside the gripper allows for guaging if it can fit an object. This use of high level control over the whole gripper, rather than individual control of fingers, also allows for portability across many different grippers.

6 Other limb interfaces

Other robot programming systems provide their own interfaces for robotic limbs.

Industrial programming systems, such as those from ABB [2006] and KUKA [2006], are designed specifically for controlling the robot manipulators for industrial robots, and include a simple control language. The KUKA environment provides:

- instructions to use advanced motion control to move the end effector along a continuous geometrically defined path containing lines and arcs
- instructions to quickly move the end effector from its current point to another point, without defining the path the end effector takes and instead moving each joint to its destination position directly
- control over individual axis velocities and accelerations
- use of Programming by Demonstration to specify positions to move to
- support for involved coordinate spaces, including tool space, robot space and world space.

The highlight of such systems is their advanced motion control ability, which allows for moving the end effector along complex paths with a high level of accuracy. The interfaces created for Player do not provide the same level of functionality in this area. The only complex movement is provided by the limb interface's *VectorMove* command. However, Player is designed for research and service robotics applications, rather than industrial robotics, and so such functionality may not be as necessary. If it is found to be needed in the future, expanding the interfaces to add suitable commands would be a simple task. The industrial systems also feature a mix of both low- and high-level functionality, such as the direct control over individual joint positions and the advanced motion control. Player provides a similar mix, but separates it into the two separate interfaces, the actuator array interface and the limb interface.

The ARIA Application Programming Interface for robots from MobileRobots [ARIA, 2006], including the Pioneer 3DX, provides a class for controlling the Pioneer robotic limb. This class uses a set of functions and types for low-level joint control over the arm. Forward and inverse kinematic calculators are provided separately, for calculating joint and end effector positions. Player takes a different approach in that the kinematic calculators are hidden from clients behind the interface, allowing programmers to ignore the fact that a calculator is being used and concentrate instead on setting poses to move to, simplifying the use of kinematics. It also provides more flexibility in supporting a wide range of hardware, as some limbs may feature kinematics calculators built into their hardware controllers, others may need them in their Player drivers, and still others may rely on client-side calculators that control the limb via the actuator array interface.

ARCL [Corke and Kirkham, 1993] provides libraries and an API for the C language, that includes manipulators. It provides many commonly used data types, such as transformations. It uses a real-time trajectory generator in the background to control the robot in response to instructions from the program. Kinematic solutions are provided by plug-in libraries. The interface presented to the programmer is quite different from Player's, focussing on providing greater support for the data involved. Player's programmer interface in the client libraries, by contrast, closely mimics the messages of the network interface.

There are many similar projects to Player in development. Not all of them provide interfaces for supporting robotic limbs. The ORCA and CARMEN projects, for example, do not provide any specialised interfaces for controlling robotic limbs. In both cases, it would be possible to provide joint level or end-effector level control using existing interfaces for controlling position, but this lacks a strong semantic link to what is being controlled. It also lacks some of the functionality of Player's interfaces, such as the ability to command a vector move on the limb interface, or the simple open and close commands of the gripper interface.

7 Sample code

Listing 1 shows a small program written using the Python client library and RADAR [Biggs and MacDonald, 2006] to move the Pioneer arm to random positions every few seconds. The actuator array interface is used to provide low-level control of the arm by setting each joint's position individually. Note the selection of random joint values based on the information provided by the interface about the range of each joint (lines 5 to 14).

Listing 1: A program to move the Pioneer arm to random positions, using the actuator array interface for individual joint control.

```
1 geom = aa.get_geom ()
  while 1:
```

```
    robot.read ()
    for i in range (aa.actuators_count):
        min = geom[i][1]
        max = geom[i][3]

        if min > max:
            moveRange = min - max
        else:
            moveRange = max - min

        randomNum = random.random ()
        newPos = moveRange * randomNum
        if min < 0~rad:
            newPos += min
        else:
            newPos += max
        if i == 1 and newPos < -0.2~rad:
            newPos = -0.2~rad

        print 'Moving joint ' + str (i) + '
            to ' + str (newPos)
        aa.position_cmd (i, newPos)

    sleepTime = random.randrange (5) + 1
    time.sleep (sleepTime)
```

Listing 2 shows a program to pick up a block, move it to another position, and put it down. It uses the C++ client library, and demonstrates the use of the limb and gripper interfaces. Points of interest include:

10, 78 Moving the limb to a known safe position, its home position, at the start and end of the program.

42 Moving the end effector to a pose offset from the object, and aligning the end effector along a calculated approach vector, using the `SetPose` command.

46, 46 Checking if the limb was capable of reaching the requested pose by checking its state. If the requested pose was outside of the limb's reachable area, the limb would be in the out-of-reach state until a new command is issued.

66, 74 Using the simple gripper commands to grip and release the object.

Listing 2: A program to pick up and move a block. The limb and gripper interfaces are used to control a Pioneer arm.

```
double HandleHeight = 0.17; // Height above
    floor that the arm should grab at
double BlockRadius = 0.001; // Radius of
    circle about top of block thing

PlayerClient Robot(RobotHostname, RobotPort);
Robot.StartThread();
LaserProxy Laser(&Robot, 0);
LimbProxy Limb(&Robot, 0);
GripperProxy Gripper(&Robot,0);

Limb.MoveHome();
Limb.RequestGeometry();
Laser.RequestConfigure();
Laser.RequestGeom();
player_pose_t LaserPose = Laser.GetPose();

// Find the closest point in the laser scan
int MinIndex = 0;
for (uint ii=1; ii < Laser.GetCount(); ++ii)
{
    if (Laser[ii] < Laser[MinIndex])
        MinIndex = ii;
}
player_point_2d_t Closest =
    Laser.GetPoint(MinIndex);
Closest.px += LaserPose.px;
Closest.py += LaserPose.py;

// Calculate a point in the middle of the
    block
double PreRange = Laser[MinIndex]
    +BlockRadius;
double Bearing = Laser.GetMinAngle() +
    Laser.GetScanRes() * MinIndex;
player_point_2d_t CentrePoint;
CentrePoint.px = PreRange * cos(Bearing);
CentrePoint.py = PreRange * sin(Bearing);
CentrePoint.px += LaserPose.px;
CentrePoint.py += LaserPose.py;

// Calculate the approach vector
player_point_2d_t Orientation;
Orientation.px = CentrePoint.px - Closest.px;
Orientation.py = CentrePoint.py - Closest.py;

// Calculate a point about 5 cm above the
    target point.
Limb.SetPose(CentrePoint.px, CentrePoint.py,
    HandleHeight + 0.05, 0,0,-1,
    Orientation.px, Orientation.py, 0);
// Wait for it to get there (this could
    actually poll the arm to check)
sleep (10);

if (Limb.GetData().state ==
    PLAYER_LIMB_STATE_OOR)
{
    cout << "Target out of reach,
        terminating" << endl;
    Limb.MoveHome();
    return 0;
}

// Move to grasp position
Limb.SetPose(CentrePoint.px, CentrePoint.py,
    HandleHeight, 0,0,-1,Orientation.px,
    Orientation.py, 0);
// Wait for it to get there
sleep (3);

if (Limb.GetData().state ==
    PLAYER_LIMB_STATE_OOR)
{
    cout << "Target out of reach,
        terminating" << endl;
    Limb.MoveHome();
    return 0;
```

```
}

// Close the gripper to grasp
Gripper.Close();
sleep(3);

// Move arm to position over back
Limb.SetPose(0.119, -0.183, 0.234, -0.031,
    0.408, -0.912, 0, -0.91, -0.409);
sleep(5);

// Open the gripper to release the block
Gripper.Open();
sleep(3);

// Move to home position
Limb.MoveHome();
```

8 Conclusions

Generalised interfaces to robot hardware and software components can improve robot programming. They make it easier for developers to move between hardware and reuse existing code. The Player project provides facilities for generalised access to robots. New interfaces were added to Player to support hardware that was previously unsupported, specifically, robotic limbs and grippers. An interface to provide a method for low-level control of the individual joints of robotic limbs was created, as well as a separate interface providing a high level control of limbs via control over the position of the end of a limb. The gripper interface was completely reworked to change it from being specific to a single gripper device to instead being a general gripper interface that can work with any robot gripper.

The interfaces are part of the standard Player distribution.[2] They have been successfully used in a final year robotics paper programming assignment and in several postgraduate and final year projects at The University of Auckland. Higher level applications can use the interfaces, particularly the limb and gripper interfaces, to interact with robotic limbs without care for the specific hardware in use, in keeping with Player's concept of providing device abstraction. A humanoid robot could use the limb interface to specify feet positions, relying on the driver for the underlying hardware to manage specific joint angles. A service robot could use the same interface, and the gripper interface, to control its arms, irrespective of the arm hardware it is equiped with.

The interfaces provide suitable functionality for the limbs of service robots. They do not provide the advanced motion controls found in industrial manipulator controllers. They allow for low-level control of individual limb joints, higher-level control over the position of the end effector and movement of the effector in such a way as to move from a pre-grasp position to a grasp position. In this way, they provide a suitable level of control for the limbs of service robots in a general interface that increases portability of robot software.

[2]The source code is available at `playerstage.sf.net`

Acknowledgements

The designs were created based on initial concepts, then modified under the advice of the Player user and developer communities. The final designs were agreed upon by the community as being suitable to the needs of Player users. The limb/gripper sample code in Listing 2 was provided by Toby Collett.

References

[ABB, 2006] The ABB group. http://www.abb.com/, 2006.

[ARIA, 2006] Advanced Robotics Interface for Applications. `http://www.activrobots.com/SOFTWARE/aria.html`, 2006.

[Biggs and MacDonald, 2006] G.M. Biggs and B.A. MacDonald. Evaluation of dimensional analysis in robotics. In *Proc. IEEE Conference on Automation Science and Engineering*, Shanghai, China, October 2006.

[Collett *et al.*, 2005] Toby Collett, Bruce MacDonald, and Brian Gerkey. Player 2.0: Toward a practical robot programming framework. In *Proceedings of the Australasian Conference on Robotics and Automation*, University of New South Wales, Sydney, Australia, December 5–7 2005.

[Corke and Kirkham, 1993] P. Corke and R. Kirkham. The ARCL Robot Programming System. In *Proc. Int. Conf. Robots for Competitive Industries*, pages 484–493, 14–16 July 1993.

[Gan *et al.*, 2005] John Q. Gan, Eimei Oyama, Eric M. Resales, and Huosheng Hu. A complete analytical solution to the inverse kinematics of the pioneer 2 robotic arm. *Robotica*, 23(1):123 – 129, 2005.

[KUKA, 2006] KUKA Automatisering + Robots N.V. http://www.kuka.be/, 2006.

[MobileRobots, 2006] Mobilerobots P3-DX. `http://www.activrobots.com/ROBOTS/p2dx.html`, 2006.

[Vaughan *et al.*, 2003] R.T. Vaughan, B.P. Gerkey, and A. Howard. On device abstractions for portable, reusable robot code. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2421–2427, 2003.