In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003)*
pages 2121--2427, Las Vegas, Nevada, October, 2003. (Also Technical Report CRES-03-009)

On device abstractions for portable, reusable robot code

Richard T. Vaughan[1] Brian P. Gerkey[2] Andrew Howard[2]

[1]Information Sciences Laboratory, HRL Laboratories LLC., Malibu CA 90265-4797 USA
[2]Robotics Research Laboratory, University of Southern California, Los Angeles CA 90089-0781 USA

***Abstract*— We seek to make robot programming more efficient by developing a standard abstract interface for robot hardware, based on familiar techniques from operating systems and network engineering. This paper describes the application of three well known abstractions, the *character device* model, the *interface/driver* model, and the *client/server* model to this purpose. These abstractions underlie Player/Stage, our Open Source project for rapid development of robot control systems. One product of this project is the Player Abstract Device Interface (PADI) specification, which defines a set of interfaces that capture the functionality of logically similar sensors and actuators. This specification is the central abstraction that enables Player-based controllers to run unchanged on a variety of real and simulated devices. We propose that PADI could be a starting point for development of a standard platform for robot interfacing, independent of Player, to enable code portability and re-use, while still providing access to the unique capabilities of individual devices.**

I. Introduction

> **Abstract**, n. 1. That which comprises or concentrates in itself the essential qualities of a larger thing or of several things. [Webster's Revised Unabridged Dictionary, 1998 MICRA, Inc., via Dictionary.com 2003]

The Player/Stage Project produces tools for rapid development of robot control code. Its two main products are the Player robot server, a networked interface to a collection of hardware device drivers, and Stage, a simulator that provides a population of virtual robot devices. Player and Stage are Open Source Software, released under the GNU General Public License. Source code is freely avalable from `http://playerstage.sourceforge.net/`.

The original goal of Player was to provide a simple, flexible interface to our Pioneer robots, and to allow code to run unchanged in simulation and on the real robots. The Player/Stage development system is now used for research projects in more than 20 labs around the world, and by many more individual students. As Player evolved, device drivers were written for the RWI robots (which ship with "Mobility"). The new drivers re-used some of the existing packet formats, as they did logically the same thing (reporting sonar range readings, etc.). There was a lot of overlap between interfaces.

This experience raised the possibility that a controller could run unchanged on *different* robot platforms. Our vision for the future development of Player and similar interfaces is that robot applications should produce similar behavior on a wide variety of robots, just as a web page can be rendered on a varietry of browsers.

This paper describes the abstractions used in our software, whereby Player "comprises and concentrates within itself the essential qualities" of robotics devices in order to afford code portability and re-use. We also make some proposals for the development of a standard platform for mobile robots, beyond the Player/Stage system.

II. The Player Abstract Device Interface

The central abstraction that enables portability and code re-use in Player is the **Player Abstract Device Interface** (PADI) specification. The PADI defines the syntax and semantics of the data that is exchanged between the robot control code and the robot hardware. The PADI's set of abstract robot control interfaces constitutes a virtual machine, a target platform for robot controllers that is instantiated at run time by particular devices.

The **Player Protocol** implements the PADI along with some additional structures and rules for multiplexing, buffering, sending and receiving collections of PADI packets, and commands for inspecting and controlling the behavior of the Player server. This protocol was designed from scratch to be suitable for mobile robot applications. While Player is a useful piece of software, the details of the shifting of packets are not fundamental to an effort to devise a standard robot programming environment. A different protocol could implement the PADI and the Player Protocol could manage a different abstract device interface. While there are many examples of special-purpose Session Layer (see section IV-A) protocols, the PADI is (we believe) a uniquely open abstract robot platform. As such, it is a candidate starting point for the development of a standard. Yet PADI is still immature and faces significant design challenges, some of which we indicate below.

III. Abstractions

As a development tool for robot control, the Player server has enjoyed significant success, becoming an important piece of enabling infrastructure for research projects in more than 20 academic and industrial labs around the world. However, Player's primary contribution derives from the abstractions it implements, rather than

from the details of its implementation. We recognize that the particular way in which Player is implemented is not fitting for all robotic application domains, but suggest that the principles of Player's design are instructive and widely applicable.

Putting aside the implementation details, which we discuss in Section VI, we can distill Player to three essential abstractions, which we describe in the following sections.

A. The character device model

The "device-as-file" model, which originated in Multics [1] and was popularized by UNIX [8], states that all I/O devices can be thought of as data files. A distinction is made between sequential and random access devices. Sequential devices such as terminals and tapes produce and consume streams of data one byte after another, and are called *character devices*, while random access devices such as disk drives can manipulate chunks of data in arbitrary orders, usually through a cache, and are known as *block devices*. The nature of sensors and actuators is to produce and consume data in time-extended streams: they are character devices.

The standard interface to character devices is through five well-defined operations. Access to devices is controlled by *open* and *close* operations. Data is collected from the device by a *read* operation, and sent to the device by a *write* operation. The asynchronous read and write are sufficient on their own for many devices, but a third transfer mechanism, the *ioctl* (input/output control) provides a synchronous request/reply channel, typically to access data that is persistent rather than sequential, such as setting and querying the configuration of the device. All five operations will indicate error conditions if they fail.

1) Player application: Player uses the character device interface to access its hardware devices. For example, to begin receiving sensor readings, the client opens the appropriate device and reads from it. Likewise, before controlling an actuator, the client opens the appropriate device and writes to it. An ioctl mechanism is used for device configuration and for atomic test/set and read/clear operations required by some devices. For reasons explained elsewhere [4], [3], Player does not currently implement exclusive access to devices, so multiple client programs can simultaneously control the same device. We are considering adding exclusive access modes, which would be analogous to file-locking mechanisms in operating systems.

2) Limitations: The character device model has some drawbacks. In particular, since there is no interrupt mechanism, clients must poll devices to receive new data. This is not the best approach for low-latency I/O with high-speed devices, which are usually interrupt-driven. Player was designed to support update rates of the order of 5-100Hz, covering the majority of research robots. This model is unlikely to suffice for devices that operate on the order of 1MHz. Also, the ioctl channel is often used in a way that breaks device independence and reduces portability, as discussed in Section IV-B.

3) Conclusion: The character device model defines five operations (open, close, read, write, ioctl) that are provided by the operating system to perform device I/O. Apart from the assumption of sequential access (suplemented with the ioctl), the character device abstraction is neutral with respect to programming language and style. Almost every programming language supports this model, and almost any robot control architecture can be (and likely has been) implemented atop the generic read/write/ioctl interface. This model has successfully supported UNIX-like operating systems for decades, and the Player server for years. We suggest that the character device model is a suitable foundation for a robot device control standard.

B. The interface/driver model

The character device model defines only the broadest semantics of its three channels (roughly: input, output and configuration), but imposes no other structure on the data streams. Each device could have its own unique data format, requiring controller code to be written specifically for each device. Another powerful abstraction, the *interface/driver* model determines the content of these streams and provides the device independence that is the key to portable code.

The interface/driver model groups devices by logical functionality, so that devices which do approximately the same job appear identical from the user's point of view. An *interface* is a specification for the contents of the data stream, so an interface for a robotic character device maps the input stream into sensor readings, output stream into actuator commands, and ioctls into device configurations. The code that implements the interface, converting between a device's native formats and the interface's required formats is called a *driver*. Drivers are usually specific to a particular device, or a family of devices from the same vendor.

Code that is written to target the interface rather than any specific device is said to be *device independent*. When multiple devices have drivers that implement the same interface, the controlling code is portable among those devices.

Many hardware devices have unique features that do not appear in the standard interface. These features are accessed by device-specific ioctls, while the read and write streams are generally device independent. Interfaces should be designed to be sufficiently complete so as to not require use of device-specific ioctls in normal operation, in order to maintain device independence and portability.

1) Player application: There is not a one-to-one mapping between interface definitions and physical hardware

components. For example, the Pioneer's native P2OS interface bundles odometry and sonar data into the same packet, but a Player client program that only wants to log the robot's position does not need the range data. For portability, Player separates the data into two logical devices, decoupling the logical functionality from the details of the Pioneer's implementation. The pioneer driver controls one physical piece of hardware, the Pioneer microcontroller, but implements two different devices: `position` and `sonar`. These two devices can be opened, closed, and controlled independently, relieving the client of the burden of remembering details about the internals of the robot.

Since Player was initially designed as an interface to our Pioneer 2-DX mobile robots, early versions of the server provided almost transparent access to specific components and peripherals of the Pioneer as it was used in the USC Robotics Lab. For example, each data packet from the sonars comprised 16 range readings, because the Pioneer has 16 sonar transducers. Likewise, command packets to the wheel motors comprised two velocities, because the Pioneer is a non-holonomic, differentially-driven robot (see Section IV-C for a further discussion of motor command formats).

This Pioneer-specific device model was extensible, but did not encourage code reuse or portability in the server or in client control programs. When code was added to the server to provide access to a new device, that device presented a unique, device-specific interface that required device-specific client-side support. As a result, client programs that controlled the second Player-supported mobile robot, the RWI B21r, used an API that was completely different from that used to control the Pioneer, despite the fact the two robots are functionally similar.

In order to more conveniently support different devices, we introduced the interface/driver distinction to Player. An interface, such as `sonar`, is a generic specification of the format for data, command, and configuration interactions that a device allows. A device driver, such as `pioneer-sonar`, specifies how the low-level device control will be carried out. In general, more than one driver may support a given interface; conversely, a given driver may support multiple interfaces. Thus we have extended to robot control the device model that is used in most operating systems, where, for example, a wide variety of joysticks all present the same "joystick" interface to the programmer.

As an example, consider the two drivers `pioneer-position` and `rwi-position`, which control Pioneer mobile robots and RWI mobile robots, respectively. They both support the `position` interface and thus they both accept commands and generate data in the same format, allowing a client program to treat them identically, ignoring the details of the underlying hardware. This model also allows us to implement more sophisticated drivers that do not simply return sensor data but rather filter or process it in some way. Consider the `lasercspace` driver, which supports the `laser` range-finder interface. Instead of returning the raw range values, this driver modulates them according to the dimensions of the robot, creating the configuration-space representation of free space in the environment.

2) Limitations: The primary cost of adherence to a generic interface for an entire class of devices is that the features and functionality that are unique to each device are ignored. Imagine a `ficudial-finder` interface whose data format includes only the bearing and distance to each fiducial. In order to support that interface, a driver that can also determine a fiducial's identity will be underutilitized, some of its functionality having been sacrificed for the sake of portability. This issue is usually addressed by either adding configuration requests to the existing interface or defining a new interface that exposes the desired features of the device. As an interesting example, consider Player's first Monte-Carlo localization driver; it can support both the sophisticated `localization` interface that includes multiple pose hypotheses, and the simple `position` interface that includes one pose and is also used by robot odometry systems.

C. The client/server model

The third abstraction provides a practical model for implementing a robot interface. Player is based on a client/server model, in which the user's control program, or *client*, is separated from the *server*, which executes low-level device control, by an external, standardized medium. In Player's case, that medium is a TCP socket. The most common alternative approach is the "direct link" model, in which only a function call separates the user from low-level device control [7].

Client/server is the dominant model for network and Internet applications. Many of the other models for moving data across networks, such as RPC, are themselves implemented on top of a client/server infrastructure.

A major advantage of the client/server model is that clients can be written in any programming language that supports the standardized communication medium used by the server. For this purpose Player uses a TCP socket, which is supported by almost every modern programming language. Thus Player is language-neutral, without the need to generate "wrappers" for different languages[1].

The client/server model also facilitates concurrent access to devices. Given a server that is capable of supporting a single client, relatively little extra work is required to support multiple clients simultaneously. Furthermore, if the medium that separates client and server is network-aware, as is the TCP socket, then clients can execute on

[1]However, it is common to use language-specific client libraries that encapsulate the handling of socket communication; see Section VI-A.

any machine with network connectivity to the machine on which Player is executing. In this way, Player is both platform- and location-neutral, exposing an identical interface to all clients, regardless of whether they are executing on the same machine or across the Internet.

1) Limitations: The principal drawback of the client/server model is that it introduces additional latency to device interactions, because data and commands are necessarily delayed as they pass through the server. Since Player provides data at a constant rate, it imposes an average delay of a half cycle on all data, in addition to network-related latencies. Although the data rate is configurable, Player is usually operated at 10Hz. Thus on average, new data sit in Player's buffers for 50ms, which is not significant for many robot control applications. This kind of latency can be reduced but not eliminated, for there are unavoidable costs resulting from the scheduler and network stack of the underlying operating system. An additional drawback of the client/server model is that it requires network hardware and software, which may not be available in extremely low-power embedded computing systems (e.g., robomotes [9]).

IV. Issues for a standard interface

Assuming that we employ the character device and interface/driver abstractions described in the previous sections, we must devise a set of useful interfaces.

These interfaces will define the classes of equivalent devices for robots, as concepts like "mouse" and "printer" do for workstations. Although each device in a given class may require a different driver, they all present the same interface and provide the same (or very similar) functionality. The PADI currently specifies 29 interfaces for a range of devices including range-finders, grippers, and probabilistic localization systems (see the Player Manual [2] for a full list). Although Player's interfaces are neither orthogonal nor spanning in the space of devices, we suggest from collective experience that the PADI specification provides a starting point for creating a standard set of interfaces.

An integral part of selecting the set of interfaces is deciding for each interface the content and semantics of the data, command, and configuration streams. In determining what is included in an interface, we trade off between generality across many devices and exposing the capabilities of any particular device. A similar tradeoff is made when deciding what is in the command and data streams, as opposed to the configuration channel.

The Player interfaces have evolved over time, but most quickly settle into quasi-static states. Interfaces usually become more general as more drivers are written, exposing limitations of the existing interfaces. The simulator Stage is an extreme example of this phenomenon, because it has none of the constraints of physical hardware.

A. The OSI Standard Model

We can define our recommended area of development in terms of the standard model for networking protocols and distributed applications. The International Standard Organization's Open System Interconnect (ISO/OSI) model distinguishes seven layers of functionality in a networked system, grouped into Application layers (5-7) and the Network layers (1-4) [10].

In terms of this model, Player uses TCP as a Transport Layer (4) which implements the character device model. The Player Protocol forms the Session Layer (5), defining the semantics of the packets moved by the Transport. The client libraries (see Section VI-A) form the Presentation Layer (6), converting data into a format appropriate for the Application Layer's (7) specific language and architecture.

Robot applications have no unique requirements for the Presentation Layer, which handles byte order, etc, so we recommend that some existing standard should be used to do the bulk of the work. The XDR External Data Representation [5] is a candidate specification. Instead, standardization effort should focus on the Session Layer, defining the commands, data and configurations supported by each interface, and thereby deciding what each class of device *does*.

B. ioctl issues

Since a major use of ioctls is for device-specific controls, managing them while maintaining device independence becomes an issue. Player applies the following policy, which we have found to work well. To support an interface, a driver may be required to support a certain set of ioctls. A driver may also support other ioctls. The set of ioctls defined for an interface is the union of all required and optional ioctls. If a driver supports an ioctl, whether required or optional, the driver must preserve the semantics for the ioctl as they are laid down in the standard.

A remaining question regarding ioctls is that of how to handle querying generic properties that cross interface boundaries, such as pose and geometry. It would be in keeping with many operating systems to leave it to each interface to define its own ioctls for these purposes (this is currently the case in Player). However, there is some benefit to be had, in terms of code re-use and usability of the standard, from defining and possibly requiring certain ioctls for all interfaces. For example, all drivers could be required, when queried, to return geometry information (or at least a standard "don't know") in the same format.

For maximum portability, a controller should use no device-specific ioctls. One simple mechanism to ensure this is to provide a "core" PADI specification that excludes all device-specific ioctls. Users would explicity enable non-portable extensions at run time, perhaps by setting a global flag, or on a per-device basis.

C. A challenge: position device interface

We have suggested that a standard set of interfaces could be designed that are widely useful for robot control. It is often straightforward to design an interface for a particular piece of hardware, but generalizing an interface over a range of devices is a non-trivial task, as the following example illustrates.

Many research robots are mobile, containing a device that allows them can change their position in the world. We'll attempt to generalize such a 'Position Device', and consider the definition of an abstract Position Device interface. A robot controller should be able to drive any mobile robot that presents the interface, with similar results.

The position of a rigid, non-articulated object can be completely specified in three dimensions relative to a fixed reference point by six numbers: latitude, longitude, altitude, roll, pitch and yaw, or (x, y, θ, a, b, c). For a wheeled robot on the plane, we have just three degrees of freedom (x, y, θ). So any Position Device can be commanded completely and unabiguously by sending it six numbers, and floor-travelling robots (the most common kind) require just three numbers.

Unfortunately, things are not so simple. Many devices and applications require velocity control rather than position control. We can extend our specification with an ioctl that switches the interpretation of the values between positions and velocities. A simple further extension allows us to select the n^{th} derivative of position, allowing acceleration control, etc. We can even allow a different derivative to be specified for each axis; for example we can command a robot to move to (x, y) while spinning at θ radians/second - a behavior that most wheeled robots can not achieve!

The HRL Pherobot is a useful example of a position device that is difficult to fit into the standard interface. The Pherobot uses the Descartes small, wheeled mobile robot platform. The Descartes's native command is (absolute heading, forward distance, speed percentage). That is, position control in θ in global coordinates, position control in x in local coordinates, and velocity control in a local frame. The velocity percentage is input into a Descartes' internal controller and does not map easily into any one coordinate system. Suffice to say, the Descartes, a simple wheeled mobile robot, has its own specific interface.

The Position Device illustrates the trade-off between a useful abstraction and a fully generalized interface. The reality of the robots' implementation makes it difficult to apply the most general abstractions, because it does not capture some important constraints imposed by real position devices.

In practice we observe that the Position Device drivers specify several device-specific ioctls, which change the meaning of the basic command packet considerably and dilute the generality of the interface. The design of a maximally general Position Interface is an interesting problem.

V. Simulation of Abstract Devices

The Player/Stage system includes the Stage simulator, which provides a population of virtual Player devices. The simulated devices interact with Player in exactly the same way as real device drivers, consuming command packets, generating data packets and responding to ioctls. There is a Stage model for each interface that requires interaction with the environment. Mirroring the development of Player, Stage models began by emulating specific devices, such as the Pioneer 2 and SICK LMS200 rangefinder, but have evolved with the interfaces to become more generic.

Designing a model to match an interface has been a useful test of the interface semantics. Removed from the constraints of a particular device, but attempting to implement the full functionality required by the logical interface, we have found that ambiguities or contradictions in the interface are exposed. Being forced to implement a general model makes explicit the key parameters, which are exactly those that must be exposed by the interface. We suggest that simultaneously designing the interface and a model is a useful methodology for developing a standard.

By operating with the same device abstractions as Player, Stage provides a drop-in replacement for the real devices. This enables convenient development of control code in simulation, and many controllers written this way have been shown to work unchanged on the real hardware. But this relationship is transitive; we can transparently replace Stage with another simulator that supports this interface. We are currently developing a new simulator "Gazebo" that models three-dimensional outdoor terrain (Stage uses a simple two-dimensional model in order to scale to large populations).

Mirroring the ioctl issues in Player, Stage must manage a growing number of device-specific ioctls. Should a generic model attempt to implement all the ioctls for a given interface? Probably not, since there is no requirement that the logic for ioctls for different devices can be consistent, merely that all the ioctls implemented by any given device be consistent, and that the behavior of a single ioctl is the same across devices. If we wish to model device-specific features, a reasonable approach would be to provide an extension layer to each model, corresponding to a particular device (e.g. an LMS200) or group of devices (e.g. P2OS-based robots). We have adopted this approach for future Stage development.

VI. Putting it together

A. APIs

It is conceivable that a single standard API for robot programming could be specified. However, our philosophy

has been to constrain the design of controllers as little as possible and experience suggests that a single API will not be practical to support the range of controller architectures and programming styles that are currently used for mobile robots.

The Player distribution comes with several "client libraries" in various languages, linkable code libraries that handle the details of packing and unpacking the device interface structures, converting from local to network-safe variable types, etc. The Player Protocol is open and documented, and users could implement controller clients that talk directly to Player through a socket. However in practice we observe that every Player client uses a client library, and the Player Protocol is only used by implementors of client libraries.

Each client library has a different design and presents a different API to the user, due partly to the differences between programming langages. Some common features have emerged, for example the C++ and Java libraries have a similar proxy-based design, but they do not match each other call-for-call. Just as the various programming languages provide different APIs to the POSIX environment, each in their native style, we believe that Player's various APIs, each targeting the PADI, are an important strength of the system. We consider the existence of multiple APIs for Player as *increasing* the portability and re-usability of the system by exposing the functionality of the set of device drivers in multiple programming languages and styles.

A standard abstract robot interface with diversity at the API level maximizes portability while interfering minimally with the design of controllers.

B. Transport Issues

This paper has so far played down the importance of the OSI Network Layers (Transport and below), and the packet-shuffling machinery in the Session Layer. For convenience, we'll refer to this whole communications implementation as the *transport*. This is because we believe the key abstractions unique to robotics are independent of any particular transport. Player is a socket server that uses TCP, while other robot interface systems use UDP [11], Remote Procedure Calls or CORBA [6] to do a similar job. Each approach has strengths and weaknesses in terms of ease of use, reliability and handling of errors, but these can be thought of as seperate from the logic of the interface. A wide variety of transports could present the same logical PADI environment to the robot application.

However, the dynamic properties of any implemented robot interface are highly significant. Unlike many computing applications, robot controllers have strong dynamic constraints. They may have specific requirements for bandwidth, latency, jitter and connection reliability. These are all attributes of the transport, not the abstract device descriptions, so the transport remains a significant factor in the design of a particular robot system.

We make a distinction, then, between the dynamic requirements of a device, determined by the application and provided by the transport, and the semantics of the device interface, determined by the PADI. The semantics of the (idealized) Position interface for a robot blimp and a robot hummingbird are the same, but the dynamic requirements of transport are very different. This implies that we should take care to leave *out* any specification of dynamic properties from the PADI.

The PADI specification does have some specific demands on the transport though, in that it must support the character device model. This is parsimonious with TCP and other stream socket protocols, but less so of RPC and CORBA approaches.

In systems that must be highly reliable, or that have very challenging communcation environments, the ability to detect and manage communication failures becomes very important. Player's TCP transport is highly reliable under typical loads and uses, but TCP's built in retries and long time-outs would provide very poor dynamic properties to clients under very heavy network loads or with intermittent connections.

These examples demonstrate that the PADI is by no means the whole story for designing a robot controller. It is, however, a sufficiently independent component that could be standardized and built into robot systems that are otherwise heterogeneous, thus permitting them to interoperate.

C. Resource Discovery

A key advantage of current Player-based robot code is that it is possible to use different hardware devices at run time. Ioctls provide a way to inspect the properties of a given device, but there is the question of how the controller establishes its connection to a device in the first place.

Player provides some basic resource discovery, in that the server can be queried for the set of devices it has available. However, client programs must know *a priori* the precise address of the server. A more powerful scheme would allow controllers to obtain addresses for available hardware on-the-fly; a resource-discovery mechanism. With this in place, controllers could probe for available hardware and choose the best sensor available for a task, optionally configuring it with ioctls. Alternatively, a controller could change its planned behavior according to the hardware resources available. This is beyond the current state of the art in robot control, and would offer interesting research opportunities. However, we can immediately suggest a further extension. The addition of a mechanism for processes to migrate between hosts would allow our robot controllers to graduate from being *portable* to being *mobile*, discovering local resources as they go. This would be an extraordinarily rich environment for future robot controllers, made feasible by a standard Abstract Device Interface.

VII. Conclusion

The specification of an abstract robot interface can usefully be broken up into three tasks. These are:

1) A collection of abstract device interaces, which specify a virtual machine that becomes the target for robotics applications. The Player Abstract Device Interface offers a starting point for this effort.
2) A Session Layer protocol for negotiating connections and shifting packets. The Player Protocol is an example.
3) Presentation Layer APIs, providing the abstract interface description in a covenient format for the client's chosen language and control style. Client libaries are available for Player in C, C++, Python, TCL, LISP, and Java, each with a distinct API.

We have referred extensively to Player, but we emphasize that we do not suggest Player as a panacea for robot control. Player has evolved from its beginnings as a platform-specific interface into its own platform, used by large projects as an integration tool. From the beginning, we have worked to make Player as open as possible in accordance with the principles of Open Source Software and in contrast to the goals of the commercial robot interfaces. This experience informed the ideas set out in this paper, which are already feeding back into Player/Stage development, and into our discussions with our colleagues in the Robotics Engineering Task Force, a body set up to examine the opportunities for standards in robotics `[http://robo-etf.org]`.

The Player system, based on the Player Abstract Device Interface, is the most open and portable platform for robot control currently available. Player-based robot control programs target a robot virtual machine, which can, in some cases, be implemented with a choice of hardware and software devices. We aim to increase Player's generality over time, tackling challenges like the Position Device interface along the way.

There is probably not One True Standard of Robot Abstract Device Interface waiting to be discovered, but there will be widely useful approximations. A community effort in this area will rapidly improve the portability and reuse of robot code.

Acknowledgments

The authors thank the Player/Stage developer and user community. A list of contributors is maintaind at `http://playerstage.sf.net/credits.html`. Thanks also to SourceForge.net for project hosting, and to Doug Gage at DARPA IPTO for his support. This work is supported in part by DARPA grant DABT63-99-1-0015 (MARS) at USC and contract N66001-99-C-8514 (SDR) at HRL.

VIII. References

[1] R.J. Feiertag and E.I. Organick. The Multics input/output systems. In *Proc. of the Symposium on Operating Systems Principles*, pages 35–41, New York, October 1971.

[2] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. *Player User Manual 1.3*. Player/Stage Project, http://playerstage.sourceforge.net, December 2002.

[3] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In *Proc. of the Intl. Conf. on Advanced Robotics (ICAR)*, pages 317–323, Coimbra, Portugal, June 2003.

[4] Brian P. Gerkey, Richard T. Vaughan, Kasper Støy, Andrew Howard, Gaurav S Sukhtame, and Maja J Matarić. Most Valuable Player: A Robot Device Server for Distributed Control. In *Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS)*, pages 1226–1231, Wailea, Hawaii, October 2001.

[5] Network Working Group. XDR: External Data Representation Standard. Internet RFC 1014, June 1987. http://www.faqs.org/rfcs/rfc1014.html.

[6] Gary Holness, Deepak Karuppiah, Subramanya Uppala, and Roderic Grupen. A Service Paradigm for Reconfigurable Agents. In *Proc. of the Second Intl. Workshop on Infrastructure for Agents, MAS, and Scalable MAS at Autonomous Agents*, Montreal, Canada, May 2001.

[7] James S. Jennings. Threaded servers enable thin robot clients. Technical report, Computer Science Dept., Tulane University, 1998.

[8] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, October 1974.

[9] Gabriel T. Sibley, Mohammad H. Rahimi, and Gaurav S. Sukhatme. Robomote: A Tiny Mobile Robot Platform for Large-Scale Ad-hoc Sensor Networks. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA)*, pages 1143–1148, Washington DC, May 2002.

[10] Andrew S. Tannenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, New Jersey, Third edition, 1996.

[11] Barry Brian Werger. Ayllu: Distributed port-arbitrated behavior-based control. In Lynne E. Parker, George Bekey, and Jacob Barhen, editors, *Distributed Autonomous Robotic Systems 4*, pages 25–34. Springer-Verlag, Knoxville, Tennessee, October 2000.