Reusable Robot Software and the Player/Stage Project

Richard T. Vaughan[1] and Brian P. Gerkey[2]

[1] School of Computing Science, Simon Fraser University, Canada vaughan@sfu.ca
[2] Artificial Intelligence Center, SRI International, USA gerkey@ai.sri.com

1 Introduction

The authors of several well-known robot software systems met at the ICRA 2005 workshop on the *Principle and Practice of Software Development in Robotics*. The meeting was held to examine the role of software engineering concepts and methods in experimental robotics applications. Everyone at the workshop agreed that extensive reuse of robot software should help to make robot development faster, easier and more efficient, and that this was highly desirable. There exist many robot programming tools and frameworks designed to promote this idea, some of which have been actively developed for several years using very fine software engineering techniques. However, very few supposedly reusable systems are extensively used outside their home institution or their immediate collaborators. Many well-engineered systems are never used at all. This suggests that there is more to getting code widely reused than nice code design, however principled.

In terms of enabling experimental robotics, the most elegant, powerful, reusable code objects are of no use until someone actually uses them. After the (excellent but simple and inflexible) Lego Mindstorms system, probably the most reused robot code in the world comes from the Player/Stage Project. We think this is because it solves some common problems and has relatively low barriers to reuse, as well as having a sensible code design.

Based on our experience as authors of Player/Stage, we suggest that to be effectively enable and promote efficiency in robotics software, the practice of "Software Engineering for Experimental Robotics" needs to include a broad consideration of all the barriers to code reuse. Some of these are technical issues of modularity or interoperability, but some, such as licensing, cost, distribution, documentation and support, are not.

This chapter reviews the Player/Stage robot development system (P/S), describing its key models and abstractions, and identifying the opportunities for code reuse presented by its several layers of interfaces. We also discuss some barriers that make code reuse difficult, and describe how P/S has avoided or solved some of these problems to become widely reused.

Before proceeding, it should be made clear that there is already a great deal of reused code running on laboratory robots and workstations every day, but almost all of it is general-purpose computing infrastructure such as Linux, GCC, glibc, GSL, MATLAB, Java, Python and Windows[1]. Some specialized code for computer vision is also commonly used, for example OpenCV[2]. None of this code is specific to robot applications, but we can learn from these successful code reuse examples.

2 Why Reuse Robot Software?

The problem of how to program intelligent robots includes nearly all the subproblems of AI, from perception, control, and reasoning under uncertainty, to planning, scheduling, and coordination. In addition, robot programming entails solving systems problems, including physical dynamics and real-time constraints, computation, memory and bandwidth limitations, and unreliable communications. Many students and researchers are attracted to the particular challenge presented by this complex combination of AI and systems. These properties, combined with the community's emphasis on demonstrating working systems mean that the products of research can often be quickly and usefully applied in the real world.

However, there are certain systems aspects of robotics work that are simply tedious. Consider the wide variety of wheeled mobile robots used in research labs. Despite the obvious similarity in their functionality and capabilities, the differences in size, shape, kinematics, and communication protocol mean that code developed for one robot (usually) must be ported or otherwise adapted to work on another robot. Simple tasks, like teleoperating a robot or visualizing its sensor state, require a significant amount of work. When they are available, interface libraries for robots often restrict the choice of programming language and/or style. Well-understood algorithms are re-implemented over and over again in laboratories around the world.

In addition to lost time as researchers duplicate engineering tasks, there is a negative impact on the quality of the resulting work. We have little shared equipment, few shared data[3], no shared environments, few shared tasks, and little shared code. As a result, robotic systems are not directly comparable,

[1] Respectively, the GNU/Linux operating system, the GNU Compiler Collection, the GNU C Library, the GNU Science Library, the MATLAB computing environment from The MathWorks, the Java programming environment, the Python programming environment, and Microsoft Windows operating system

[2] `http://sourceforge.net/projects/opencvlibrary/`

[3] An exception is the Radish project [`http://radish.sourceforge.net`], a public repository for localization and mapping data sets.

and competing approaches are often evaluated only in a "trial by video" at workshops and conferences.

Recently, challenges such as RoboCup and the DARPA Grand Challenge have focused research efforts and provided clear metrics for evaluating performance. These challenges are not a panacea, however; the overhead involved in entering such a competition is considerable, as is the danger of overfitting solutions to the peculiarities of the competition environment.

The goal of the Player/Stage Project is to develop Free Software infrastructure that improves research practice and accelerates development by handling tedious tasks and providing a standard development platform. More than simply distributing this infrastructure as a set of tools to the research community, we invite members of the community to contribute to the project. By collaborating on a common system, we share the engineering burden and create a means for objectively evaluating published work. If you and I use a common development platform, then you can send me your code and I can replicate your experiments in my lab. The ability to perform methodical peer evaluation is expected in the natural sciences but lamentably absent from robotics today.

3 Code Reuse from the Player/Stage Project

Our claim that code from the Player/Stage Project is frequently reused needs some supporting evidence. Player's distribution terms, the GNU General Public License[4] allow anyone to use and distribute the sourcecode. This makes it impossible to obtain precise numbers of Player users. There is no reliable mechanism for user registration or reporting. We have four sources of documentary evidence that Player code is being used:

1. Download statistics;
2. Support forum messages and bug tracker items;
3. Submissions to our user laboratory list;
4. Acknowledgements in published robotics articles.

The Player/Stage Project has been hosted by Sourceforge[5] since December 2001. As of 24 July 2006, Sourceforge reported 56,757 downloads of P/S software, and a download rate of around 2,000 packages (2.2GB) per month.

Figure 1 is a graph of the number of downloads and bandwidth supplied by Sourceforge each month of the period December 2001 to December 2005. Downloads show faster-than-linear growth during 2002-4, and were roughly constant in 2005; a year that saw no major new releases. The subsequent release of Player-2.0 in March 2006 produced a noticeable spike of 3,270 downloads, around 150% of the typical number.

[4] `http://www.gnu.org/copyleft/gpl.html`
[5] `http://sourceforge.net`

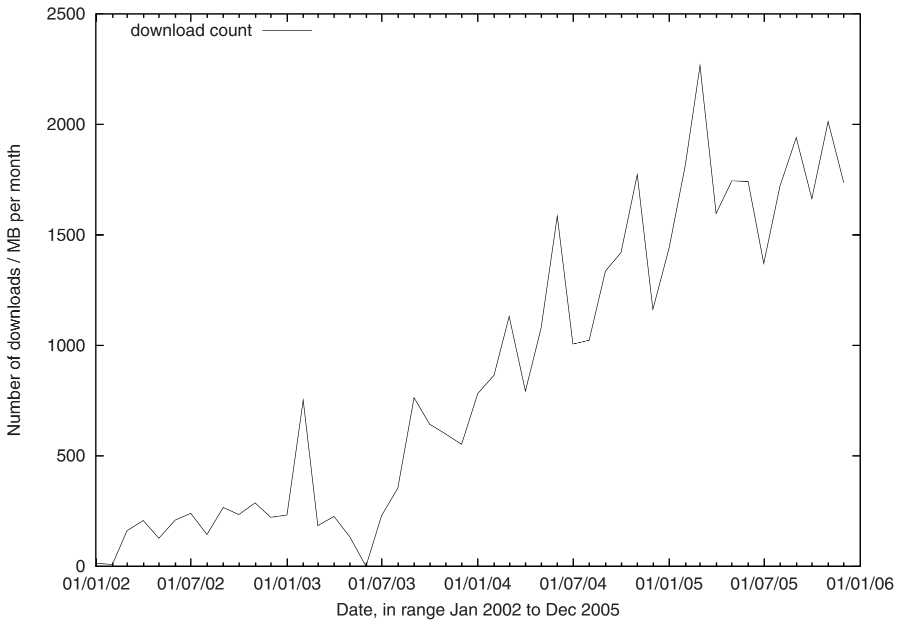


Fig. 1. Four-year download history for the Player/Stage Project

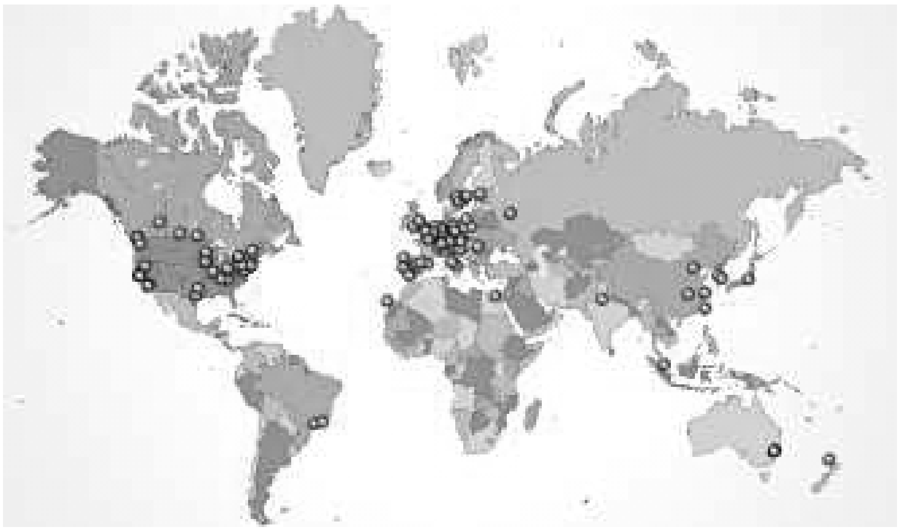


Fig. 2. Geographical distribution of downloads in December 2005

Table 1. Mailing list archive in July 2006

Mailing list	**Description**	**Subscribers**	**Messages**
playerstage-users	Player and Stage user support	293	4136
playerstage-developers	developer discussion	143	1864
playerstage-gazebo	Gazebo user support	121	1345
total messages			7345

Table 2. Issue-tracking database items (including closed items) in July 2006

Tracker	**Items**
Bugs	191
Patches	121
Feature requests	96
total items	408

Google's web traffic analytics tracking service[6] provides (among other things) a breakdown of the geographical location of visitors to the P/S web site. In the month of December 2005, a total of 70,920 web page requests were served. Figure 2 shows the locations of the visitors, demonstrating the global interest in Player. Unfortunately geographical data for actual software downloads (rather than web page requests) is not available, but we can assume that the distribution of software downloads is similar to that of the web requests.

3.1 Support Forum and Trackers

P/S uses Sourceforge's mailing list and issue-tracker services to communicate with users and developers. Activity on these services suggests active users. Table 1 shows the number of subscribers and messages on each list in July 2006. There were a total of 7,759 messages on the lists since their creation in January 2002; an overal average of 4.7 messages every day, with 12.5 messages per day in the first half of 2006. The total number of subscribers is not meaningful because many people subscribe to more than one list.

Another indicator of user activity is the issue-tracking database, a mechanism designed to manage issues that can not be immediately resolved on the mailing lists. Figure 2 shows the number of items on each list from January 2001 to July 2006: a total of 408. Notable here are the 121 code patches submitted by users to fix bugs or extend P/S software. Many of these patches have been applied to the tree to become 'official' code.

3.2 User Site List

Users are invited to submit the name of their laboratory for listing on the P/S Wiki site[7]. As of July 2006, the list contains 52 entries in 19 countries,

[6] `http://google.com/analytics`
[7] `http://playerstage.sourceforge.net/wiki/`

including the Boeing Company, the Australian Centre for Field Robotics, the Chinese National University of Defense Technology, the Technical University of Munich, the NARA Intitute of Science and Technology, Georgia Institute of Technology, the Mexican National Institute of Astrophysics, Optics and Electronics, and Rochester Institute of Technology's DARPA Grand Challenge Autonomous Race Team.

3.3 References in Scientific Articles

Many authors not connected with P/S development or each other have published papers that acknowledge the use of Player/Stage Project code in their experiments. Examples include the papers [AALB05, HHFS05, TC05, iSW05] all from the IEEE International Conference on Intelligent Robots and Systems (IROS) in 2005. Others are cited elsewhere in this chapter.

Based on these figures, we conclude that Player/Stage is both a well-known and well-*used* source of robot code.

4 Design of the Player Robot Device Interface

All work, including robotics research, is impacted by the tools that are used. Good tools simplify common tasks, while bad tools complicate them. The assumptions that are built into a set of tools bias the researcher who uses them toward particular kinds of solutions.

Our design philosophy is heavily influenced by the operating systems (OS) community, which has already solved many of the same problems that we face in robotics research. For example, the principle function of an operating system is to hide the details of the underlying hardware, which may vary from machine to machine. Similarly, we want to hide the details of the underlying robot. Just as I expect my web browser to work with any mouse, I want my navigation system to work with any robot. Where OS programmers have POSIX, we want a common development environment for robotic applications. Operating systems are equipped with standard tools for using and inspecting the system, such as (in UNIX variants) **top**, **bash**, **ls**, and **X11**. We desire a similar variety of high-quality tools to support experimental robotics.

Operating systems also support virtually any programming language and style. They do this by allowing the low-level OS interface (usually written in C) to be easily wrapped in other languages, and by providing language-neutral interfaces (e.g., sockets, files) when possible. Importantly, no constraints or normative judgments are made on how best to structure a program that uses the OS. We take the same approach in building robotics infrastructure. Though not strictly part of the OS, another key feature of modern development environments is the availability of standard algorithms and related data structures, such as **qsort()**, TCP, and the C++ Standard Template Library. We follow this practice of incorporating polished versions of established algorithms into the common code repository, so that each researcher need not re-implement, for example, Monte Carlo localization. Finally, an important but often overlooked aspect of OS design is that access is provided at all levels. While most C programmers will manage memory allocation with the library functions **malloc()** and **free()**, when necessary they can dig deeper and invoke the system call **brk()** directly. We need the same multi-level access for robots; while one researcher may be content to command a robot with high-level "goto" commands, another will want to directly control wheel velocities.

In summary, our approach to building tools for robotics research is to extend useful abstractions from the OS up *just enough* to enable robotic devices to be used as easily as normal computer devices such as mice and printers. Like an OS, we aim to provide resources as transparently as possible, extending and managing the hardware but otherwise keeping out of the user's way.

5 Abstractions

Player comprises four key abstractions: The Player Abstract Device Interface (PADI), the message protocol, the transport mechanism, and the implementation. Each abstraction represents a reusable and separable layer. For example, the TCP client/server transport could be replaced by a CORBA transport. Alternatively, an entirely different system could be built atop the PADI. These four abstractions, described in the following sections, are the result of a considerable design effort refined over several years of broad use and are in themselves opportunities for reuse, independent of their P/S software implementations.

5.1 The Player Abstract Device Interface

The central abstraction that enables portability and code re-use in Player is the PADI specification. The PADI defines the syntax and semantics of the data that is exchanged between the robot control code and the robot hardware. For ease of use, the PADI is currently specified as a set of C message structures; the same information could instead be written in an Interface Definition Language (IDL), such as the one used in CORBA systems.

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 goal of the PADI is to provide a virtual machine that is rich enough to support any foreseeable robot control system, but simple enough to allow for an efficient implementation on a wide array of robot hardware. The key concepts used in the PADI, both borrowed from the OS community, are the character device model and the driver/interface model.

The Character Device Model

The "device-as-file" model, which originated in Multics [FO71] and was popularized by UNIX [RT74], 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.

Player uses the character device interface to access its hardware devices. For example, to begin receiving sensor readings, the appropriate device is first opened, after which data can be read from it. Likewise to control an actuator, the appropriate device is opened, after which commands can be written 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 [GVS01, GVH03], Player does not currently implement exclusive access to devices, so multiple modules can simultaneously control the same device. We are considering adding exclusive access modes, which would be analogous to file-locking mechanisms in operating systems.

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 below.

Apart from the assumption of sequential access (supplemented 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 Player for years. We suggest that the character device model is a suitable foundation for a robot device control standard.

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.

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 controller 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: `position2d` and `sonar`. These two devices can be opened, closed, and controlled independently, relieving the user 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.

This Pioneer-specific device model was extensible, but did not encourage code reuse or portability. When code was added to provide access to a new device, that device presented a unique, device-specific interface that required device-specific support in control programs. As a result, 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 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 `position2d` 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.

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 `fiducial-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 under-utilized, 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. Consider Player's Monte-Carlo localization driver **amcl**; it can support both the sophisticated `localization` interface that includes multiple pose hypotheses, and the simple `position2d` interface that includes one pose and is also used by robot odometry systems.

5.2 The Player Protocol

The **Player Protocol** implements the PADI along with some additional structures and rules for multiplexing, ordering, sending and receiving collections of PADI messages, and commands for inspecting and controlling the behavior of Player itself. For example, part of the protocol specification states that data and messages are asynchronous and not acknowledged, while configurations are synchronous: every request is guaranteed an acknowledgment (positive or negative) in response. The protocol also defines a generic header structure that precedes every message and contains the meta-data necessary to unambiguously interpret the message. Note the the protocol does not specify the manner in which packets are serialized, addressed, or transmitted; those details are handled by the transport layer.

5.3 Transport Mechanisms

The PADI and Player Protocol together are sufficient for building a single-process robot control system. Drivers can be instantiated and bound to interfaces, and the resulting devices can exchange PADI-defined messages (e.g., by function calls) according to the rules of the protocol. However, if we want the ability to move messages among devices or other modules that are in different processes or on different machines, then we need a **transport** mechanism.[8] The job of a transport mechanism is two-fold: handle the addressing and routing of packets, and perform packet serialization and deserialization (also called data marshaling). Some transports may also provide higher-level functionality, such as resource discovery.

TCP Client/Server Transport

Historically Player has relied on a TCP client/server transport, in which devices reside in a *server* and a control program is a *client* to the server. To control a robot, the user first starts the **player** server, which listens on a particular TCP port (by default 6665), on the robot. Then a client program, such as a joystick controller or data visualization GUI, is started and establishes a TCP socket connection to the server. The client can run on-board the robot or on any other machine that has network connectivity to the robot. One client can connect to many servers and many clients can connect to one server. Importantly, clients can be written in any programming language with support for TCP sockets.

In order to safely send messages from one machine to another over a TCP socket, a data marshaling scheme must be defined. The marshaling rules specify the bit-level details of how, for example, numbers are represented on the

[8] With respect to the OSI Network Model [Tan96], we refer collectively to the Network Layers (Transport and below) and the packet-shuffling machinery in the Session Layer as the *transport*.

wire. Earlier versions of Player used a custom encoding of messages as packed C structs that contained integers in network byte-order. Player now uses an open standard called eXternal Data Representation, or XDR [Net87]. The XDR specifies an efficient, platform-independent encoding for commonly-used data types, including integers and floating point values. To reduce the occurrence of marshaling bugs, the library that performs the XDR data marshaling is generated in an automatic fashion directly from the header file that specifies the PADI.

Other Transports

The client/server model has served Player (and many other distributed systems) well for many years, but it is not the ideal transport mechanism for every robot system. In order to allow for the use of other transports, Player was redesigned to be transport-independent. The TCP client/server transport is still available (and will likely continue to be the mostly widely-used), but other transport mechanisms can be substituted in its place. The drivers, message structures, and other underlying details remain unchanged.

For example, we have developed a JINI-based transport for Player. JINI is a Java-based architecture for building distributed systems in a network-centric manner [Wal99]. JINI has been used in many distributed systems, including the largest multi-robot system deployed to date [KOV04]. JINI offers a number of advantages over the TCP client/server approach, including: robustness to network delays and dropouts, automatic resource discovery, and effortless data marshaling using Java's built-in object serialization mechanism. On the other hand, it is non-trivial to install, configure, and run the JINI infrastructure; and of course control programs must be written in Java.

Other trade-offs exist for other popular transport mechanisms, such as CORBA [Obj02], IPC [SW97], and ACE/TAO [SLM98]. It is highly unlikely that the robotics community will agree on a single transport for all robotics software, nor should they; the requirements and constraints exhibited by any given application area will make particular transports more or less appropriate, and it is important to allow the system designer to choose the best one. Because the manual construction of a new Player transport layer is a tedious and bug-prone process, we have taken care with the the PADI and core Player libraries so as to facilitate the automatic generation of transport code. For example, the JINI transport layer uses automatically-generated Java bindings for Player. We expect to see other transports developed similarly.

The choice of transport is critical for a robotics application as it is the transport that determines the real-time peformance of the system. Assuming that the robot software has been designed to avoid any logically unecessary time delays, the transport and the underlying OS that implements it are the only source of timing delays due to buffering, message transmissionn, etc. For control of most slow-moving, statically-stable wheeled robots, TCP over

Ethernet (802.3) or WiFi (802.11b/g) is found to have acceptable timing performance and its ubiquity makes it a good choice of general purpose transport.

5.4 Implementations

We have implemented the PADI, the Player Protocol, TCP client/server transport (with XDR data marshaling), and many device drivers as a set of reusable C/C++ libraries. The most common use of these libraries is in an executable server, called **player**. This server is used to parse configuration files, instantiate drivers, and service client connections to devices. It is customary when using Player to assign to each robot a server that contains all the drivers used in that robot's control system. The user's control program is written as a client that executes outside of the server (the situation is essentially the same, with different terminology, when using JINI instead of TCP).

The server represents a privileged space in which modules have better, faster access to hardware and to each other. Control code on the client side experiences greater latency and a somewhat lessened ability to interrogate devices. On the other hand, the client code has fewer constraints with respect to programming language and structure, and it is relieved of the drivers' burden of behaving properly to avoid crashing the rest of the system.

In this way, Player is analogous to a monolithic kernel operating system, in which a privileged kernel space is separated from a non-privileged user space [SGG05]. Most operating systems, including Linux, employ monolithic kernels. An alternative approach, used by operating systems such as QnX, is the microkernel, in which there is no privileged kernel space, but rather a collection of system processes and a mechanism for efficiently passing messages between them. There are advantages and drawbacks to each approach and the topic has been debated, without resolution, in the OS community for decades. An example of a microkernel-like robot control system is CARMEN [MRT03]. It is worth noting that although Player more naturally operates as a monolithic kernel, a microkernel system can be constructed by connecting multiple servers together, each with a single driver (server-server communication operates exactly like client-server communication).

As an alternative to our C/C++ system, an equivalent implementation of Player could be done in, for example, Java. A Java-only Player would be useful for the many embedded computing platforms that execute Java bytecode natively. Of course the existing C/C++ drivers could not be used on a native Java system. Other possible re-implementations include using strictly C (for systems without C++ runtime support) and removing the reliance on POSIX threads (which Player uses extensively).

6 Higher-Level Drivers

While Player's primary purpose is to provide portable and nearly transparent access to robot hardware, an increasing number of drivers encapsulate sophisticated algorithms that are removed by one or more steps from the physical hardware. These *higher-level drivers* use other drivers, instead of hardware, as sources of data and sinks for commands. The **amcl** driver, for example, is an adaptive Monte Carlo localization system [TFBD00] that takes data from a **position2d** device, a **laser** device, and a **map** device, and in turn provides robot pose estimates via the **localize** interface (as mentioned above, **amcl** also supports the simpler **position2d** interface, through which only the most likely pose estimate is provided). Other Player drivers perform functionality such as path-planning, obstacle avoidance, and various image-processing tasks.

The development of such higher-level drivers and corresponding interfaces yields three key benefits. First, we save time and effort by implementing well-known and useful algorithms in such a way that they are immediately reusable by the entire community. Just as C programmers can call **qsort()** instead of reimplementing quicksort, robotics students and researchers students should be able to use Player's **vfh** driver instead of reimplementing the Vector Field Histogram navigation algorithm [UB98]. The author of the driver benefits by having her code tested by other scientists in environments and with robots to which she may not have access, which can only improve the quality of the algorithm and its implementation. Second, we create a common development environment for implementing such algorithms. Player's C++ **Driver** API clearly defines the input/output and startup/shutdown functionality that a driver must have. Code that is written against this API can enter a community repository where it is easily understood and can be reused, either in whole or in part. Finally, we create an environment in which alternative algorithms can be easily substituted. If a new localization driver implements the familiar **localize** interface, then it is a drop-in replacement for Player's **amcl**. The two algorithms can be run in parallel on the same data and the results objectively compared.

7 APIs

We have described above the various interfaces to Player's components. In practice, the majority of user code will interact with Player through a *client library*; a language-specific interface that the user compiles (or loads, depending on the language) into their client program. Each client library presents an 'Application Programming Interface' (API). The most commonly used client libraries (and hence APIs) are the libplayerc and libplayerc++ libraries, for C and C++ respectively. Several other libraries are thin wrappers around one of these. For example the Python client library is automatically generated from libplayerc using the SWIG (Simplifiedf Wrapper and Interface Generator) tool [Bea96]. Using SWIG, changes to the libplayerc interface are correctly propogated through to APIs in several languages without having to tediously edit each by hand. As well as saving time, this removes a common source of bugs.

There are several advantages for users in using the client libraries instead of talking to the Player server directly; first, client libraries hide the details of the client/server communications almost completely, so the user can largely ignore the socket-level Player protocol, marshalling and serializing data, etc. Secondly, each API is designed to in a way that is natural way for its language; for example libplayerc++ presents client-side proxy objects that correspond to server-side devices. Thus the user can manipulate the proxies as fully-fledged objects, inherit from them, etc. As required by its language, the C client library has a similar proxy-based design, but structures and function calls are used instead.

Client library external APIs must change only when the PADI changes: they need not change with the client/server protocol, though of course the libraries must change internally to communicate with the server. The APIs can often be backwards compatible, for example when the PADI is extended but existing specifications are not changed the client API will still work, it will simply not provide access to the newly-defined structures.

As a 'Presentation Layer' interface in OSI terms, the client library APIs are almost independent of the transport layer in that they hide most details of the transport. However, some high-level transport-related details may leak through this abstraction. For example, all the client libraries that target the Player TCP server must be supplied with the hostname and port number of a running Player server on initialization. After initialization, the TCP connection is completely transparent as user code interacts with Player through client-side proxy objects (C++, Java, Python) or structures and function calls (C).

The popularity of the client libraries, particularly libplayerc++ and libplayerc, means that their design is very important. As the most-used interface anywhere in the Player/Stage system, the quality of their design plays a large part in the utility of the whole system. If the APIs are too complex, or inconsistent, or poorly documented, users will be quickly frustrated. Libplayerc was carefully designed to be as consistent and transparent as the authors could reasonably make it. It was intended to replace libplayerc++ which was written originally to test the Player TCP server and not intended for end-users. The fact that almost all early users used the C++ library instead of talking to the server took us by surprise.

For most users, the internal design of the server is of no interest at all, so long as she is protected from it by a client library. However, code correctness is equally important throughout the system: a bug anywhere upstream will eventually appear to the user through this interface. The existence of clean, maintainable server code is justfied by the increased probability of code correctness alone.

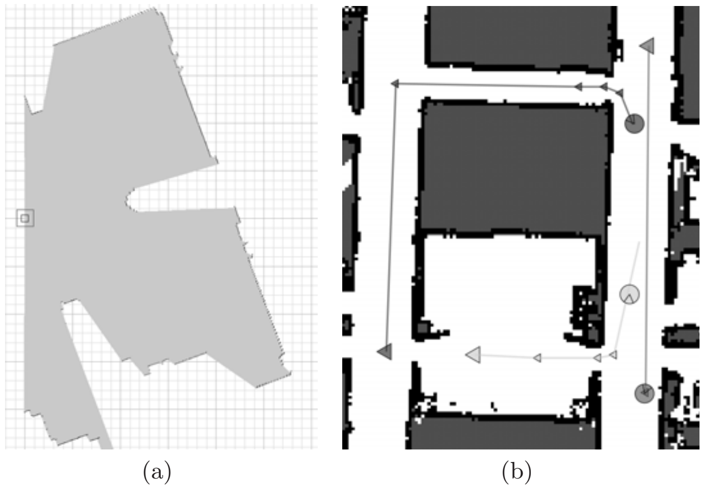


Fig. 3. Screenshots of: (a) *playerv* displaying range data from a laser-equipped robot; and (b) *playernav* showing poses and planned paths for a team of robots.

8 Tools

As an OS provides the basic services needed to control a computer, Player provides the basic services necessary to control a robot. An OS is also bundled with useful tools for common tasks, like listing files and displaying the process table. Similarly, Player is bundled with several tools:

- **playerprint** : Fetches and prints sensor data to the console.
- **playerv** : Fetches and graphically displays sensor data; also provides teleoperation by mouse movement (Figure 3(a)).
- **playerjoy** : Provides joystick teleoperation.
- **playervcr** : Provides remote control of data logging and playback.
- **playernav** : A graphical operator control unit that provides control over localization and path-planning for multiple robots (Figure 3(b)).
- **playerwritemap** : Fetches grid and vector maps (e.g., from a SLAM driver) and writes them to disk.
- **playercam** : Remotely displays video imagery from robot-mounted cameras.

Third-party tools have also been developed, including a tool similar to **playercam** and an OpenGL-based 3-D application that combines some of the functionality of **playerv** with some of **playernav**. The development of such tools by users outside of the P/S/G project is a testament to the reusability of the system and the extensibility of the architecture.

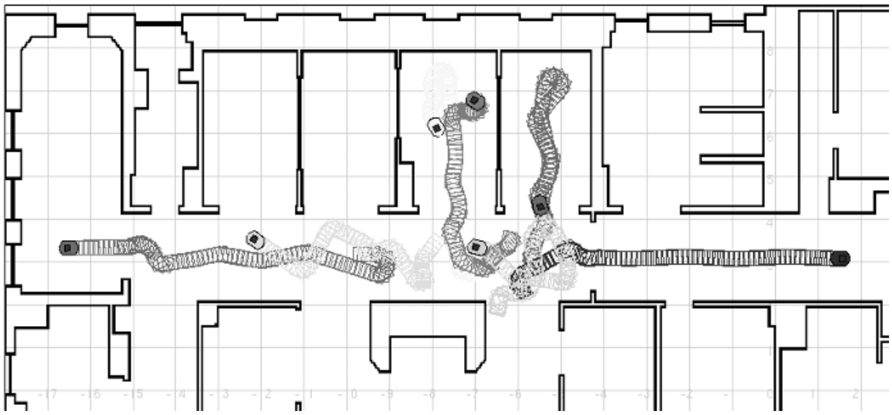


Fig. 4. A screenshot from the Stage multiple-robot simulation, showing several robots leaving trails as they explore a small section of a hospital floorplan.

9 Simulation

A significant contribution of the Player/Stage project is to provide robot simulators. The main benefits to the user of using a simulation over a real robot are convenience and cost: simulated robots are usually easy to use, their batteries need not run out, and they are very much cheaper than real robots.

9.1 Stage

After Player, the next most reused component of the Player/Stage project is the Stage robot simulation engine. Stage provides a virtual world populated by mobile robots and sensors, along with various objects for the robots to sense and manipulate. Designed with multi-agent systems in mind, it provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity. This design is intended to be useful compromise between conventional high-fidelity robot simulations, the minimal simulations described by Jakobi [Jak97], and the grid-world simulations common in artificial life research [Wil85]. We intend Stage to be just realistic enough to enable users to move controllers between Stage robots and real robots, while still being fast enough to simulate large populations. We also intend Stage to be comprehensible to undergraduate students, yet sophisticated enough for professional reseachers.

Stage provides several sensor and actuator models, including sonar or infrared rangers, scanning laser rangefinder, color-blob tracking, fiducial tracking and mobile robot bases with odometric or global localization. Figures 4 and 5 show screenshots from a running simulation.

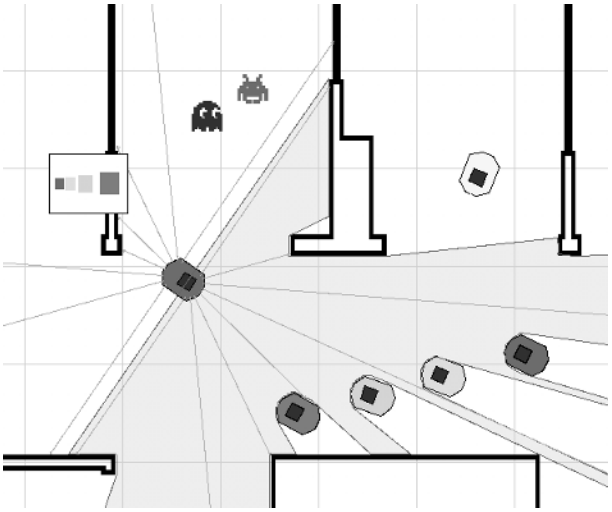


Fig. 5. A close-up screenshot from the Stage multiple-robot simulation, showing rendered laser and sonar data and several robots.

Stage is most commonly used with Player to form the Player/Stage system. Robot controller client programs interact only with Player, so simulated Stage devices appear identical to real devices. Client programs do not need to be rewritten or even recompiled to switch from simulated to real devices: a very convenient feature. Stage is implemented as a Player plugin driver (`libstageplugin`), loaded as Player starts up. This allows Stage to be developed and released on a schedule independent of Player.

```
#include "stage.h"

int main( int argc, char* argv[] )
{
  stg_init( argc, argv );
  stg_world_t* world = stg_world_create_from_file( argv[1] );

  while( (stg_world_update( world,TRUE )==0) )
    {
      /* use the simulation */
    }

  stg_world_destroy( world );
  return 0;
}
```

Fig. 6. Creating a complete multiple-robot simulation in C with libstage.

The Stage simulation engine is implemented as the standalone C library `libstage`; libstageplugin is a wrapper around libstage that connects it to Player's driver architecture. Libstage can be used to very easily create a robot simulation in user code. This is useful for users who wish to have more control over the internals of a simulation than they can get through Player's `simulation` interface. It also allows for perfectly repeatable experiments, without the unpredictable timing inevitably introduced by Player's TCP transport.

Using Stage directly without Player also saves a little computational overhead for those with very high performance requirements. The main downside is the loss of Player's high-level drivers (VFH, AMCL, etc.). Using the libstage in this way is very simple, as illustrated Figure 6 showing the C code required to instantiate a complete multiple robot simulator, similar to that shown in the screenshot above.

Stage is probably the most-used robot simulator, with research papers acknowledging the use of Player/Stage appearing in most major journals and conferences. The first published paper to use libstage directly was [ZV06], but libstage has also the basis of a the commercial product "MobileSim" from MobileRobots Inc. since 2005. This is noteworthy because none of the Stage maintainers have any relationship with the company or MobileSim, but MobileRobots have been active in contributing patches back to libstage.

9.2 Gazebo

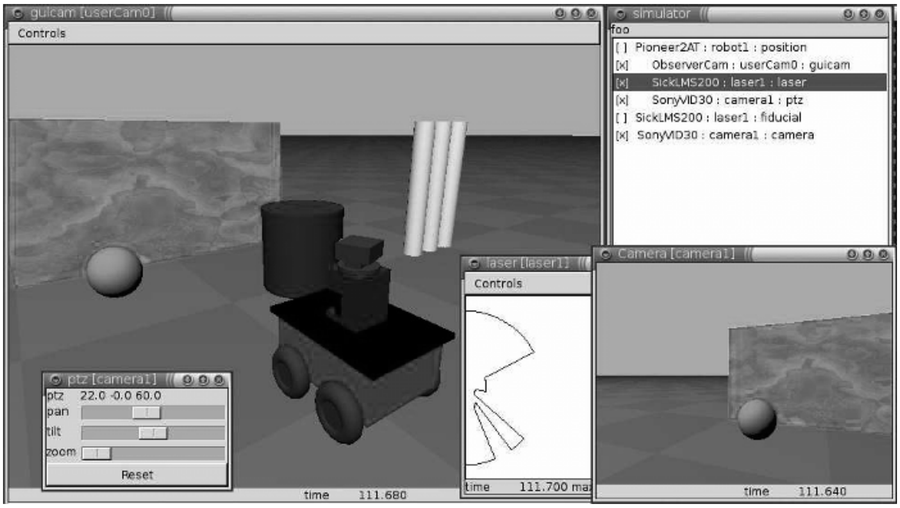
Gazebo is also a robot simulator that works with Player. Unlike Stage, it provides realistic kinematics and dynamics in three-dimensional environments. Figure 7 shows a screenshot from Gazebo. Client programs written using one simulator can usually be run on the other with little or no modification. Gazebo is more realistic than Stage, but much more computationally intensive. Stage is designed to simulate a large robot population with low fidelity, and Gazebo is designed to simulated a fairly small population with high fidelity. Thus the two simulators are complimentary, and users may switch between them without cost due to the common Player interface.

Like Stage, Gazebo is implemented as a standalone library and can be used without Player. Due to its complexity, Gazebo is currently more difficult to install and run than Stage, and is less frequently used.

10 Barriers to Reuse

Why are most robot code tools used by no one but their authors? Robot infrastructure code will not be widely used if it is

1. very buggy, or otherwise of poor quality;
2. undocumented;



```
foo
[ ] Pioneer2AT : robot1 : position
[x]    ObserverCam : userCam0 : guicam
[x]      SickLMS200 : laser1 : laser
[x]      SonyVID30 : camera1 : ptz
[ ] SickLMS200 : laser1 : fiducial
[x] SonyVID30 : camera1 : camera
```

```
ptz   22.0 -0.0 60.0
time      111.680
time      111.700
time      111.640
```

Fig. 7. A screenshot from the Gazebo 3D robot simulator, showing a model of a Pioneer robot carrying a SICK laser scanner and pan/tilt/zoom camera, with some environmental objects.

3. too expensive;
4. tied to a specific robot platform.
5. not solving the problems that lots of people really have;
6. difficult for the user to express their intentions;
7. overwhelmingly complex or cumbersome to use;
8. not distributed.

The first three reasons are self-explanatory, but the others may require some elaboration.

Several well-known pieces of software are produced and/or distributed by robot manufacturers to add value to their products. They deliberately only work with that company's robots. The market for robots is small and diverse, and many robots are still custom-built. Robot companies often (perhaps usually?) go out of business. When evaluating the software from a vendor, the researcher has to decide if the software is worth the investment in time and money (if the software is not free) to invest in software that restricts her choice of robot, and for which the support from someone with access to sourcecode could disappear at any time. These problems are the same with all commmercial software, but the small size and volatility of the research robot business makes this particularly risky.

The next two reasons are closely related. Producing software that does not solve anyone's real problems is a trap that is often fallen in to. For example, one feature that is repeatedly proposed to make robot programming easier is graphical programming, i.e. building systems by connecting boxes with lines using some spatial layout tool. This seems to ignore the fact that the population of robot programmers are overwhelmingly graduates of computer science or engineering, most of whom have no difficulty expressing themselves in code. Being artificially constrained to thinking about a problem *only* graphically, can be frustrating. Some common code structures, such as loops and recursion, are difficult to represent graphically. As anyone who has used Simulink knows, complex programs quickly lead to cluttered screens and so lots of time is spent arranging objects spatially: an arrangement that has no meaning at all for the code. Great complexity is also a problem: users are quick to abandon a system if they feel overwhelmed or can not get their robot equivalent of "Hello World" working in a few hours.

Another problem is of overwhelmingly complex or cumbersome software. Many of the proposed solutions to robot programming problems require infrastructure that may take more time to install and debug than they can save. The application of middleware such as CORBA, TAO and JINI, attractive though it is to software engineers, must be carefully justified to a robot programmer on a tight time budget, with no one in the lab with past experience in these systems. Even world-famous laboratories have "rapid" robot development systems that are so complex that the in-bouse engineers will not use them for real projects.

The final, and very basic reason that many systems are not widely used is that they are not distributed. This is usually due to institutional rules that prevent code from being given away free. The robot software market is very small and few people will pay to license software, so it stays within its development team only.

Notice that of these eight reasons, only the first two would automatically be addressed by conventional software engineering techniques. The rest are strategic, marketing and political problems that can not be solved by applying object-oriented techniques or client-server, publish-subscribe design patterns.

11 Conclusion

The most obvious way is to use code from the Player/Stage Project is to use the software packages as distributed. We have presented evidence above that this is happening frequently. However, the resources developed in and around the Player/Stage Project can be re-used in several different ways:

1. use the software packages as provided;
2. extend Player with new drivers and/or interfaces;
3. extend Stage or Gazebo with new simulation models;
4. use the simulation, driver or server libraries as part of custom software;
5. use the interface specification (PADI) as is, or as a guideline for creating custom interfaces
6. use the non-code resources, such as the Stage environment bitmaps that have become well-known;
7. use Player as middleware

This last option will become important in the near future. The drivers, PADI and Player Protocol specifications in Player are very valuable resources. Player's strength is the transparency it provides by trying to avoid placing constraints on the robot programmer. A Player-equipped robot is a blank slate, with a loose collection of easily accessible, commonly-used devices. A very suitable use for Player is as a substrate, or middleware layer, for more structured robot programming frameworks. If such a framework targetted Player instead of robot hardware directly, it would automatically inherit the large base of supported robots and implemented algorithms, saving the authors a great deal of development time.

Code has a good chance of being widely reused if it is

1. solving a user's problems
2. supported by their robots, or easy to port;
3. easy enough to use;
4. easy to obtain;
5. good enough quality;
6. well documented;
7. affordable;
8. supported by a knowledgable group of people;

We believe that Player/Stage is popular and widely used because it meets these criteria better than any other current system. It is not perfect, it is not finished, and it is not the right choice for every application. It is the product of a community of robot programmers, and it works for us.

References

[AALB05] G. Alankus, N. Atay, C. Lu, and B. Bayazit, *Spatiotemporal query strategies for navigation in dynamic sensor network environments*, Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), 2005.

[Bea96] David M. Beazley, *Swig : An easy to use tool for integrating scripting languages with c and c++*, Fourth Annual USENIX Tcl/Tk Workshop (Livermore, California), USENIX, July 1996.

[FO71] R.J. Feiertag and E.I. Organick, *The Multics input/output system*, Proc. of the Symposium on Operating Systems Principles (New York), October 1971, pp. 35–41.

[GVH03] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard, *The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems*, Proc. of the Intl. Conf. on Advanced Robotics (ICAR) (Coimbra, Portugal), June 2003, pp. 317–323.

[GVS01] 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*, Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS) (Wailea, Hawaii), October 2001, pp. 1226–1231.

[HHFS05] A. Hassch, N. Hofemann, J. Fritsch, and G. Sagerer, *A multi-modal object attention system for a mobile robot*, Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), 2005.

[iSW05] Junzhi Yu inyan Shao, Guangming Xie and Long Wang, *A tracking controller for motion coordination of multiple mobile robots*, Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), 2005.

[Jak97] Nick Jakobi, *Evolutionary robotics and the radical envelope of noise hypothesis*, Adaptive Behavior **6** (1997), no. 2, 325–368.

[KOV04] Kurt Konolige, Charlie Ortiz, Regis Vincent, Benoit Morisset, Andrew Agno, Michael Eriksen, Dieter Fox, Benson Limketkai, Jonathan Ko, Benjamin Stewart, and Dirk Schulz, *Centibots: Very large scale distributed robotic teams*, Proc. of the International Symp. on Experimental Robotics (ISER) (Singapore), June 2004.

[MRT03] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun, *Perspectives on standardization in mobile robot programming: The Carnegie Mellon Navigation (CARMEN) toolkit*, Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS) (Las Vegas, Nevada), October 2003, pp. 2436–2441.

[Net87] Network Working Group, Sun Microsystems, Inc., *RFC 1014 – XDR: External data representation standard*, June 1987.

[Obj02] Object Management Group, Inc., *The Common Object Request Broker: Architecture and Specification, Version 3.0*, July 2002.

[RT74] Dennis M. Ritchie and Ken Thompson, *The UNIX Time-Sharing System*, Communications of the ACM **17** (1974), no. 7, 365–375.

[SGG05] Avi Silberschatz, Peter Baer Galvin, and Greg Gagne, *Operating system concepts*, Seventh ed., J. Wiley & Sons, Inc., New York, 2005.

[SLM98] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, *The design of the TAO real-time object request broker*, Computer Communications **21** (1998), no. 4, 291–403.

[SW97] Reid Simmons and Gregory Whelan, *Visualization tools for validating software of autonomous spacecraft*, Proc. of the Intl. Symp. on Artificial Intelligence, Robotics, and Automation in Space, July 1997.

[Tan96] Andrew S. Tannenbaum, *Computer networks*, Third ed., Prentice Hall PTR, Upper Saddle River, New Jersey, 1996.

[TC05] E. Topp and H. Christensen, *Tracking for following and passing persons*, Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems (IROS), 2005.

[TFBD00] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, *Robust monte carlo localization for mobile robots*, Artificial Intelligence **128** (2000), no. 1-2, 99–141.

[UB98] Iwan Ulrich and Johann Borenstein, *VFH+: Reliable Obstacle for Fast Mobile Robots*, Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) (Leuven, Belgium), May 1998, pp. 1572–1577.

[Wal99] Jim Waldo, *The Jini Architecture for Network-Centric Computing*, Communications of the ACM **42** (1999), no. 7, 76–82.

[Wil85] Stuart W. Wilson, *Knowledge growth in an artificial animal*, Proc. Int. Conf. Genetic Algorithms and their applications (ICGA85), Pittsburgh PA. (Hillsdale NJ.) (J. J. Grefenstette, ed.), Lawrence Erlbaum Associates, 1985.

[ZV06] Yinan Zhang and Richard Vaughan, *Ganging up: Team-based aggression expands the population/performance envelope in a multi-robot system*, Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA), 2006.