APPENDIX A

PLAYER-STAGE SIMULATOR

Player is a device that provides a powerful and flexible interface to a variety of sensors and actuators [Gerkey et al., 2003]. Each sensor is linked to a device through a specific driver and is accessed by a client program using the interface provided by the driver. Client-server architecture allows programs to access and control the physical devices. Client programs linked to the player server connect physically to every device in the system. In this way, Player acts as a hardware abstraction layer. To improve scalability, a simulation component should be available that uses the same interface as the robot. Ideally, simulator-based controllers can be reused as the hardware controller with minimal modifications [Gerkey & Vaughan, 2007]. In addition, Player-Stage provides a deeper understanding of kinematics, and has the capabilities of handling limitations of sensors, without any difficulty in dealing with actual hardware.

Communication in Player-Stage is carried out by using a Transmission Control Protocol (TCP) socket. Since communication follows that standard, player clients can be written in any language that supports socket. And also, Player is one of the most used frameworks in the robotics community. It allows the control and the simulation of several robotic platforms and incorporates various algorithms for robot navigation, localization, mapping, etc. Stage software has many important technical features such as cooperating
with popular hardware interface; quite easy usage; providing models of many
of the common robot sensors; supporting multiple robots to share one world
and running in different platforms [Gerkey et al., 2001].

Background and Related work

For solving the exploration and coverage problem, the tool used and its
limitation factors need to be known and understood. This section gives a brief
description of the background material on simulators. There are several other
software packages positioned closely to Player/Stage/Gazebo. Also there are
many publications that cover a lot of advantages and disadvantages of each
project from an architectural point of view [Collett et al., 2005; Vaughan et al.,
2003]. We can say that a very important aspect for any robotic simulator is
that it has to keep up with the rapid advancements in the field, and must be as
flexible as possible. We start with a description of the target simulator, and
then illustrate how we implement our algorithm using this simulator.

Why A Simulater is Needed?

Generally, robots are very specialized devices and no two systems are
alike. Buying a robot or making a robot is quite more expensive than buying
or developing a simulator. In common, robot developers do not directly get
into hardware design. In some cases such as complex robot behaviors and
multi robot environments, mechanical creativeness is very important. We
cannot make effort to experiment directly on hardware devices. So the
simulation is extremely important and since much of the development has to
happen on a software basis, it is often possible to build software simulators.
When we implement the algorithm in hardware, if robot works successfully in
simulation we can expect less hardware failures only.
Different Simulators in Use

There are many simulators available in this field [Carpin et al., 2007; Mondada et al., 2004], and every simulator has its own advantages and disadvantages. Some of them are taken for discussion here. Mobile Robot Simulator (MOBS) is a 3-dimensional simulation system for mobile robot systems [Appendix B]. The available sensors are sonar sensors, odometry, bumpers, and camera. MOBS has a complex sonar model, including multiple reflections, intensity calculations, and noise, which reflects real world behavior. This simulator can be connected to a robot application program even without re-compilation of the application program.

*Webots* is another high quality commercial mobile robotics simulator that provides a rapid prototyping environment for modeling, programming and simulating mobile robots [Appendix B]. This software is close to functionality with *Player-Stage*. Using *Webots* the user can design complex robotic setups, with one or more, similar or different robots, in a shared environment. A large choice of simulated sensors and actuators is available to equip each robot. In contrast to Player, no *Webots* driver exists to drive Amigobots, the robot used to develop the high level control system. Unfortunately, *Webots* is not a freely distributed simulator [Michel, 2004]. One other commercially available simulator, called *RoboWorks*, Newtonium works with the open source *RoboTalk* interface for real-time communication with their *RoboWorks* models [Appendix B]. Most of the analysis and visualization softwares such as Matlab, MathCad, and LabVIEW support graphics limited to charts and graphs. *RoboWorks* adds 3D modeling and animation capabilities to these software packages.

Another simple 2D Java-based multi-robot simulator called *TeamBots* was popularly used in and around 2000 [Appendix B]. This software is easy to use and freely available at that time, and the ability to run the same code in
simulation and on real robots. *TeamBots* is still available, but further development is stopped in 2000. *SPADES* is a middleware simulator for mobile AI robots that have the sense-think-act cycle interaction with the simulated world [Appendix B]. *SPADES* is a middleware system and is designed to handle some of the tricky parts of running a distributed simulation without being tied to any one particular simulation. The robot’s behavior is mainly tracked by the computation time required for given processes. Its strengths lie in out of order execution of simulation events, as well as being robust to variations in the network and machine load. A survey and comparison of robot development platforms and simulators are given in [Craighead et al., 2007; Kramer & Schultz, 2007].

**Player-Stage Simulation Software**

The work on the *Player-Stage* project started at the University of Southern California in the late nineties and moved to source-forge in 2001 [Collett et al., 2005; Vaughan, 2000]. From the robotic literature, it is noticed that during the recent past, *Player-Stage* has been consistently and continuously evolving. According to the *Player-Stage* website [Appendix B], over 50 different research laboratories and institutions all around the world are currently involved in the active development of this simulator, and even more people are using it. The sharing of huge amounts of sensor information among researchers becomes easy and practical by using this simulator platform. The data can be played back into the system via a virtual driver. This feature enables the researchers to test and develop their algorithms without the need for a real environment.

*Player-Stage* simulator is an open-source piece of software and open development methods supporting the academic tradition better than closed source products, since it encourages independent validation of experimental results. It is a multi-robot simulator and is written with portability in mind and
runs on almost any platform such as Linux, Ubuntu, Compaq ipaqs and Microsoft Windows.

*Player-Stage* is the ideal platform for specifying movement and search strategies for agents programmatically [Gerkey et al., 2001]. *Stage* provides only 2D- environment and the sensors available in stage allow the user to design and test *player* robot controllers without having to use the real robots. *Player-Stage* provides simulation drivers for a variety of sensors, making it the most realistic environment to encode strategies for autonomous agents. *Player-Stage* middleware offers a combination of transparency, flexibility and speed that makes it the most useful robot development environment available in the market. But sometime, direct human control over the robots is difficult in *Player-Stage* software. To facilitate our analysis we further subdivide the *player* into two main components, namely client library and server layer. Due to the standardized interfaces, and because of the fact that *Player-Stage* was designed to be language and platform independent, various client libraries exist for a large variety of programming languages: C, C++, Java, Python, LISP, Ada, Octave, Ruby, Scheme, etc [Appendix B]. *Player server* represents the abstraction layer between the code, and the commands necessary to interface with either real or simulated robot hardware. Any number of clients can connect the *Player* server to an existing device, for accessing data, sending commands or requesting configuration changes.

![Player-Stage Architecture](image.png)
Appendix A

Player

Player defines a set of standard, powerful and flexible interfaces which specify different ways of interacting with robotics devices. Fig. 4.1 depicts the Player-Stage architecture. It focuses exclusively on sensors and actuators and it tries to establish a client-server-based framework to permit TCP-based communications between robot-based devices. In this way, Player acts as a hardware abstraction layer. Player can run on many Unix-based computers providing a simple interface to a robot's sensors, actuators, and other devices. The software contains many powerful function calls to non-specific drivers allowing the programmer to reuse control programs on different robots without re-writing any code. There are standard interfaces which are supported by many drivers, one for each type of robot or sensor. It can therefore be reused and shared to a greater extent than code tied to certain hardware.

Currently player supports low level robot control protocols from five vendors. Development and compilation of player has to happen with gcc installed, which makes porting to different architectures fairly straightforward. The communications and control are pretty much platform and hardware independent, with only minimal sacrifices in terms of speed and efficiency. It is noticed, that there can be more than one server and a client may connect to multiple servers. For every server, each interface is provided with an index such that there can be multiple interfaces of the same kind in the server.

a) Interface

The interface defines the syntax and semantics of all messages that can be exchanged with entities in the same class. For example the laser interface defines a format in which a planar range-sensor can return range readings. Player serves as a robot device interface [Appendix B]. The interface can control linear and angular velocities and feed back odometry information.
Player defines a set of standard interfaces, each of which is a specification of how the user can interact with a certain class of robotic devices such as sensor, actuator, or algorithm. An example of player interface is position2d, which covers ground-based mobile robot to accept commands to move on the terrain. This interface is supported by several drivers, such as the p2os for Pionner robots and the reflex for RWI robots. A program that uses these interface, is able to control, without any change or recompilation, different kinds of robots. The only modification would be in a configuration file, used by the player server that contains the declaration of each device and its respective driver.

b) Driver and Device

Driver is a piece of software that supports a specific hardware, which communicates with robotic sensors and actuators [Appendix B]. The driver's job is to make the robot to support the standard interface. And it hides the specifics of any given entity by making it appear to be the same as any other entity in its class. For example in player, the sicklms200 driver controls a SICK LMS200, which is a particular planar range sensor. Most of the drivers support the position2d interface, including p2os, obot, and reflex, each of which controls different kinds of robots. So the driver also knows how to translate the retrieved data to make it match with the format defined by the laser interface. In player, transfer of all messages occurs among devices, through interfaces. For example, the sicklms200 driver can create a device, which has the following address: "localhost: 6665: laser: 0". The fields in this address correspond to the entries in the player_devaddr_t structure: host, robot (port number), interface, and index. The host and robot fields indicate where the device is located and the interface field indicates which interface the device supports, and how it can be used. The player allows us to access many devices concurrently, so we need to specify the above data to access multiple devices.
c) Transport Mechanism

*Player* also provides transport mechanism that allows data to be exchanged among drivers and control programs that are executing on different machines [Appendix B]. This means, a *player* server can be run by using sockets and a specified driver for the hardware involved, and any client programs may be run from a different location over the network through the specified sockets. The most common transport in use now is a client-server TCP socket-based transport. The *player* is executed with a configuration file that defines which drivers to instantiate and how to bind them to the specific hardware. The drivers run inside the *player*, and the user's control program runs as a client to that *player* server.

d) Software Code Repository

Currently there are many abstract drivers which have been developed and used instead of hardware [Appendix B]. The main use of abstract drivers is to encapsulate useful algorithms in a way that they can be easily reused. For example, the *amcl* driver is an implementation of adaptive Monte Carlo localization, a well-known algorithm for probabilistic localization of a mobile robot. This driver supports the *position2d* interface, so it can be used directly in place of odometry, and hence, *player* becomes a common development platform and code repository for such algorithms.

Stage

*Stage* is the simulation of two-dimensional environment, designed to interface with the *player server* and is fully customizable [Kramer & Schultz, 2007; Appendix B]. It supports the straightforward definition of new test environments and robot hardware variables. The *stage* package is designed to couple with the *player* interface in order to allow user to simulate robots and robot behaviors without the need for physical hardware. The robot models in *stage* are designed so that they repeat important physical attributes such as
robot size, and shape. *Stage* will run much faster in offline application than real time, which is useful for long or batch experiments. *Stage* provides fairly simple, computationally cheap models of lots of devices rather than attempting to imitate any device with great commitment. Low commitment simulation can actually be an advantage when designing robot controllers that must run on real robots, as it encourages the use of robust control techniques. It is stated by the developers that “agents developed in simulation will work with little or no modification on the real devices and vice-versa” [Gerkey et al., 2001]. Various sensor models are provided, including sonar, laser rangefinder, pan-tilt-zoom camera with color blob detection and odometry.

Virtual sensors and robot controllers can be tested without having to play around with a real robot; the *stage* client does not distinguish between the real thing and their software representations. The design of *stage* aims at simulations that are efficient and configurable rather than being highly accurate [Gerkey et al., 2003]. When we use virtual environments and virtual robots, it is possible to simulate devices that the researcher might not have or some time difficult to implement. When *stage* attempts to run, each simulated entity acts like its real counterpart. Models are updated at a fixed interval, so controllers are free to use real time clocks to synchronize the activity. But, there is no guarantee that experiments in *stage* are directly comparable with those in the real world. However, users have found that client programs developed using *stage* will work with little or no modification with the real robots and vice versa. In addition, *player clients* cannot tell the difference from real hardware to *stage*’s device simulations, so tests and real experiments can be done using the same programs. *Stage*’s open source license allows peer review of the simulation code, and encourages sharing of models, configurations and environments. Another important feature is that *stage* is scalable to large robot populations too.
APPENDIX B

WEBSITES OF ROBOT SIMULATORS

Java Client:

LISP Client:
http://www-robotics.cs.umass.edu/~bburns/software/player-lisp.html
(accessed on 15/03/2009)

MOBS - Mobile Robot Simulator: MOBS is a fully 3-dimensional simulation system for mobile robot systems. The simulator understands the same ASCII sequences as the "Robuter II" robot. The simulator can be connected to a robot application program even without re-compile of the application program. http://robotics.ee.uwa.edu.au/mobs/
(accessed on 11/05/2009)

Player/Stage Source Forge Homepage: Player is a network server for robot control, which run on your robot. It provides a clean and simple interface to the robot's sensors and actuators over the IP network. Player supports a variety of robot hardware and is designed to be language and platform independent. Player allows multiple devices to present the same interface and is also designed to support virtually any number of clients. Stage simulates a population of mobile robots, sensors and objects in a two-dimensional bitmapped environment. It is often used as a Player plugin module, providing populations of virtual devices for Player.
http://playerstage.sourceforge.net (accessed on 10/04/2009)

Player/Stage Drivers: Writing a Player Plug-in from the Penn State Robotics Encyclopedia RoboWiki:
(accessed on 10/04/2009)

RoboWorks Software: it is an easy to use software tool for 3D modeling, simulation and animation of any physical system.
Appendix B

SPADES: SPADES is a middleware system. It is designed to handle some of the tricky parts of running a distributed simulation for the artificial intelligence community without being tied to any one particular simulation.  

TeamBots: TeamBots is a Java-based collection of application programs and Java packages for multi-agent mobile robotics research. It supports prototyping, simulation and execution of multi-robot control systems.  
http://www.teambots.org/ (accessed on 18/03/2009)

Webots: Webots is a development environment used to model, program and simulate mobile robots. User can design complex robotic setups, with one or several, similar or different robots, in a shared environment using webots. A large choice of simulated sensors and actuators is available to equip each robot. The robot controllers can be programmed with the built-in IDE or with third party development environments.  