PLAYER-STAGE BASED SIMULATOR FOR SIMULTANEOUS MULTI-ROBOT EXPLORATION AND TERRAIN COVERAGE PROBLEM

K.S. Senthilkumar and K. K. Bharadwaj

School of Computer and Systems Sciences
Jawaharlal Nehru University, New Delhi 110067
ksskumar16@yahoo.com, kbharadwaj@gmail.com

ABSTRACT

One of the possible ways of offering assistance without risking additional human lives during hazardous situations is by deploying a robot team, equipped with various sensors and actuators. Working with intelligent robotics requires a large investment in both money and time. There is a general purpose, open source simulator called Player/Stage, which provides a hardware abstraction layer to several popular robot platforms, and is commonly used by robotics community in research and university teaching, today. This simulator tends to be very simple and task-specific. Player, which is a distributed device server for robots, sensors and actuators, can control either a real or simulated robot thus allowing direct application of developed algorithms to real-life scenarios. Hence, we believe that Player/Stage, when coupled with robust hardware, is a viable paradigm for our Simultaneous MSTC(S-MSTC) algorithm. This paper gives details of our experience in running Player/Stage during the implementation of our online S-MSTC algorithm for multi-robot being implemented in C++ and we use Player/Stage middleware for validation and testing. In addition, the experience with Player/Stage can help us to do research in more complicated situations.

KEYWORDS

Multi-robot, Exploration, Terrain Coverage, Player/Stage Simulator

1. INTRODUCTION

In recent past, there has been an increasing interest in the software side of robotics such as simulators. The general goal is to write programs that are not fixed to specific architectures and can be easily reused with different robotic platforms. Working with robots involves an interesting and some time challenging interaction between hardware and software components. It is necessary to have a fairly deep understanding of both software and hardware when attempting to successfully design and develop intelligent robotic solutions. Construction and maintenance of different environments are expensive, in terms of money and time. We are looking for an open robotic middleware capable of implementing multiple mobile robots. So we intend using a proven and tested open-source middleware environment known as *Player/Stage*, for implementing our algorithm. Openness of this software encourages transparency and replication, and facilitates easy modification of different experiments. Richard T. Vaughan, one of the founder developers of *Player/Stage* simulator, has mentioned in one of his papers that how massively this software is used in major research conferences and university course websites indicate its wide usage and usefulness.

Player is a device that provides a powerful and flexible interface to a variety of sensors and actuators [1]. 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

DOI : 10.5121/ijaia.2011.2410

connects 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 [2]. 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.

Communications 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; is quite easy usage; providing models of many of the common robot sensors; supporting multiple robots to share one world and can be run in many different platforms [3].

The rest of the paper is structured as follows: in section 2, we describe the background and related work on simulator. Section 3 gives an overview of the *Player/Stage* simulator; its features; advantages and disadvantages of using this simulator. Section 4 discusses how we implement our algorithm using *Player/Stage* simulator middleware, and shows the experiments results in different environments. Finally, section 5 gives the conclusion of the work.

2. 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 [1, 4, and 5]. 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.

2.1. Why we need a Simulater?

Generally, robots are very specialized devices and no two systems are alike. Buying a robot or making a robot is quite 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 creativitness is very important. We cannot make effort to experiment directly on harware 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 level we can expect little hardware failures only.

2.2. Different Simulators in Use

There are many simulators available in this field [6, 7], and every simulator has its own advantages and disadvantages. We have taken some of them here. Mobile Robot Simulator (MOBS) is a 3-dimensional simulation system for mobile robot systems [8]. 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. This software is close to functionality with *Player/Stage*. Using *Webots* [9] 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 [10]. One other commercially available simulator, called *Newtonium* works with the open source *RoboTalk* interface for real-time communication with their *RoboWorks* models [11]. 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. 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 new development is stopped in 2000 [12]. *SPADES* [13] is a middleware simulator for mobile AI robots that have the sense-think-act cycle interaction with the simulated world. *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 is given in [14, 15].

3. 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 [4]. From the robotic literature, we can notice that during the recent past, *Player/Stage* has been consistently and continuously evolving. According to the *Player/Stage* website [16], over 50 different research laboratories and institutions all around the world are currently involved in the active development of this simulator, and even more 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 [3]. *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. We can say, that *Player/Stage* middleware offers a combination of transparency, flexibility and speed that makes it the most

useful robot development environment available in the market [1, 3]. 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 [17], Python, LISP [18], Ada, Octave, Ruby, Scheme, etc. The *Player* server represents the abstraction layer between our code, and the commands necessary to interface with either real or simulated robot hardware. Any number of clients can connect to the *Player* server and access data, send commands or request configuration changes to an existing device.

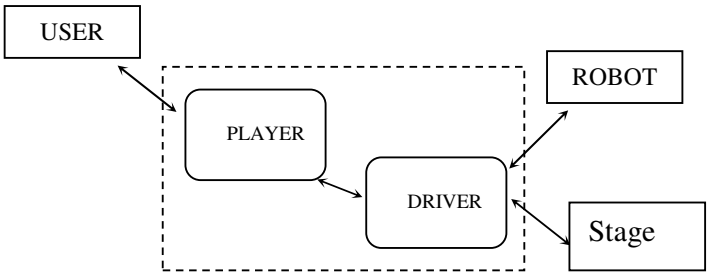


Figure 1. Player- Stage Architecture

3.1. Player

Player defines a set of standard, powerful and flexible interfaces which specify different ways of interacting with robotics devices. Figure 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 inter-faces 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. We can note 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.

3.1.1. Interface

The interface defines the syntax and semantics of all messages that can be exchanged with entities in the same class. E.g. the laser interface defines a format in which a planar range-sensor can

return range readings. *Player* serves as a robot device interface [16]. 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.

3.1.2. Driver and Device

Driver is piece of software that supports a specific hardware, which communicates with robotic sensors and actuators [19]. The driver's job is to make the robot to support the standard interface and to hide 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 [16]. Most of the drivers support the *position2d* interface, including *p2os*, *obot*, and *rflex*, each of which controls different kind 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*, transferring all messages occurs among devices, through interfaces. For example, the sicklms200 driver can create a device, which might have 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.

3.1.3. Transport mechanism

Player also provides transport mechanisms that allow data to be exchanged among drivers and control programs that are executing on different machines [16]. That means, a *player* server can be run by using sockets and a specified driver for the hardware involved, and any client programs may then 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.

3.1.4. Player as a software code repository

Currently there are many abstract drivers which have been developed and used instead of hardware [16]. 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 both the *position2d* interface, so it can be used directly in place of odometry, and, *player* becomes a common development platform and code repository for such algorithms.

3.2. Stage

Stage is the simulation of two-dimensional environment designed to interface with the *player* server and is fully customizable [15, 16]. 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" [3]. 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 highly accurate [1]. 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 clients developed using *stage* will work with little or no modification with the real robots and vice versa. In addition, *player* clients can not 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.

4. IMPLEMENTATION OF S-MSTC ALGORITHM

To use *Player* with *Stage* you need two main files: a *stage.world* file which defines the simulated world (environment or terrain) and a *player.cfg* configuration file which maps the simulated devices onto *player* devices. As mentioned above, the configuration file informs the *player* server, each device, its respective driver, and the port it is connected to. Coding directly towards a specific hardware platform is bad because it ties the code to that platform only. We must focus on the real task to control the robot behaviour from low level to a higher level implementation details. The figure 2 shows our *configuration* file. It declares drivers for a robot, sonar sensors and position sensors. There are two robots of which, the model P0 is for robot1 and model P1 is for robot2. *World* file contains the description of the environment, called from the *configuration* file.

```
Driver (  name "stage"
          provides ["simulation:0"]
          plugin "libstageplugin"
          worldfile "senthilb.world"         )
Driver (  name "stage"
          provides ["6665:position2d:0" "6665:sonar:0" "6665:laser:0" ]
          model "p0" )
Driver ( name "stage"
          provides ["6666:position2d:0" "6666:sonar:0" "6666:laser:0" ]
          model "p1"  )
```

Figure 2: Player configuration file.

In order to start an instance of the *Player* server on the robot itself we execute the following command at the *RedHat* shell. *player <config_file>.cfg –p <port_number>*

The port number is optional and default port is 6555. The *configuration* file is necessary to provide *player* with the information it needs to communicate with the *stage*.

```
size [20 20]
Window (  size [600 600]
          center [0.000 0.000]
          scale 0.03  )                    # 1 pixel = 0.03 meters
Map (  bitmap "bitmaps/myterrain.png"
           bitmap_resolution 0.03          #1 bitmap pixel is scaled to .03 meters
           size [20 20]                    #20 grids X 20 grids
           name "rooms"  )
Laserpioneer (  color "green"
           name "p0"
           pose [-4.5 -4.5 0]  )           #robot's position x,y,direction
Laserpioneer (  color "blue"
           name "p1"
           pose [-4.5 -6.5 0] )
```

Figure3. Stage World file

After a series of experiments, we have found out the suitable parameter values of the *configuration* file for our specific robot. *Player* offers a Transmission Control Protocol (TCP) sockets to connect to clients. In fact *player* supports multiple clients concurrently, each on a different socket. The socket abstract a client program to control any machine to which there is network connectivity. The figure 3 shows our *world* file. It declares size and map of the terrain, robots' initial location and obstacles' location if any. Here we describe two robots, two obstacles and the environment file "*myterrain.png*". The size of the terrain is 20x20 grid cells. There is a file with *.inc* extension providing the design of a robot and an obstacle, which is shown in figure 4, and 5. These files are called in the *world* file to place the map, robot and obstacles in the terrain. Here we describe our robot and its sensors.

```
define p2dx_sonar ranger (
 scount 11                    # define the pose of each transducer [xpos ypos heading]
 spose[0] [ -0.25 0.25 100 ]
 spose[1] [ 0.0 0.375 90 ]
 spose[2] [ 0.25 0.25 76 ]
 spose[3] [ 0.25 0.25 65 ]
 spose[4] [ 0.25 0.25 24 ]
 spose[5] [ 0.25 0.25 15 ]
 spose[6] [ 0.375 0.0 0 ]
 spose[7] [ 0.375 0.0 -50 ]
 spose[8] [ 0.375 0.0 -75 ]
 spose[9] [ 0.0 -0.375 -90 ]
 spose[10] [ -0.25 -0.25 -100 ]
 sview [0 3.8 22]          # sonar length, and corn angle
 ssize [0.01 0.01] )
```

Figure 4. Positions of the sonar rangers of our Robot

As discussed above, *Player* is supported by numerous client libraries some of which are built in to the existing installation. Numerous example files for this library can be found in the *player-*

2.1.2/examples/libplayerc++ directory. The general structure of a client program is given by the following outline:

1. Connect to robot by constructing a *PlayerClient* object
2. Create devices that are to be used in the program by requesting them from the *PlayerClient* object.
3. While (finishing Condition is Not True)

   {Read the data from devices based on received data determine actions}

```
define obstacles position (
  size [1.0 1.0]
  polygons 1
  polygon[0].points 4
  polygon[0].point[0] [  0.0 0.0 ]
  polygon[0].point[1] [  0.0 1.0 ]
  polygon[0].point[2] [  1.0 1.0 ]
  polygon[0].point[3] [  1.0 0.0 ]
  polygon[0].filled 1 )
```

(a). Obstacle

```
define pioneer2dx position (
        size [0.7 0.7]
        origin [0.0 0.0 0]
        gui_nose 1
        gui_boundary 0
        mass 15.0
        polygons 1
        polygon[0].points 8
        polygon[0].point[0] [  0.125 0.5 ]
        polygon[0].point[1] [  0.25  0.75 ]
        polygon[0].point[2] [  0.5   0.875]
        polygon[0].point[3] [  0.75  0.75]
        polygon[0].point[4] [  0.875 0.5 ]
        polygon[0].point[5] [  0.75  0.25 ]
        polygon[0].point[6] [  0.5   0.125]
        polygon[0].point[7] [  0.25  0.25 ]
        polygon[0].filled 1
        drive "diff"          # differential steering model
        p2dx_sonar() )    # use the sonar array defined above
```

(b). Robot

Figure 5. Define the size and shape of the obstacle and robot

Various devices are accessed in a control program through various Proxies which are requested in step 2. The proxies we used include: *SonarProxy* and *Position2dProxy.* Our client program is our online S-MSTC algorithm, which we discussed in our paper [20]. The figure 6 depicts our robot's shape and the sensor, and how it decomposes the terrain into cells using the sensor readings, the gray cells have obstacles. The robot is equipped with an array of eleven sonar sensors as shown in the figure 6**.**

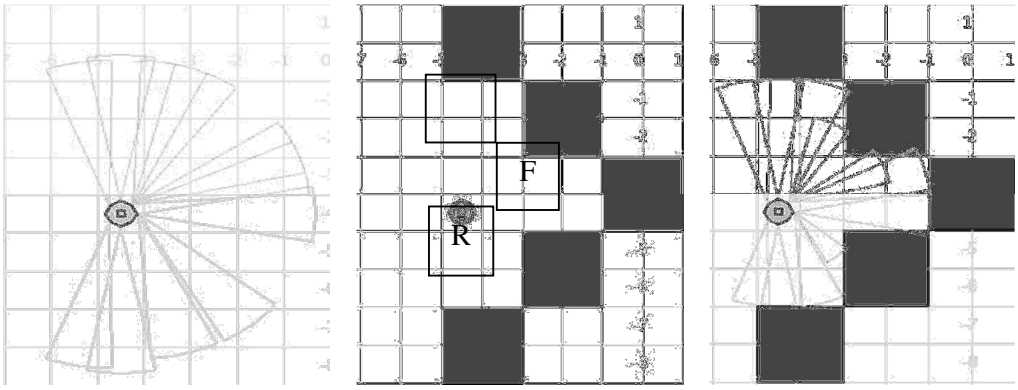


Figure 6. Robot and its Sensors

We use a "behavior based" algorithm to control the robot during the exploration and coverage task. The algorithm treats different sensor inputs separately, and then combines them together. We identify four behaviors: select cell, avoid going out of the terrain boundary, avoid an obstacle or robot and move as shown in figure 7.

1. Select cell to move- The robot receives its current location from *Position2dProxy*. The target cell is selected by our algorithm. The nearest unvisited free cell is selected first.
2. Avoid going out of the terrain boundary - The terrain boundaries are treated as static obstacles, and handled the same way like obstacle avoidance behavior.
3. Avoid an obstacle or robot - The sonar data contains an array of distances indexed by the view angle. If any distance is less than a certain threshold, behavior "Avoid an obstacle or robot" will be activated. The speed is reduced if the obstacle is getting closer, and turn direction is determined by which side has a longer clear distance.
4. Move – The robot moves to the target cell through the sub-cells by circumnavigating the spanning tree.

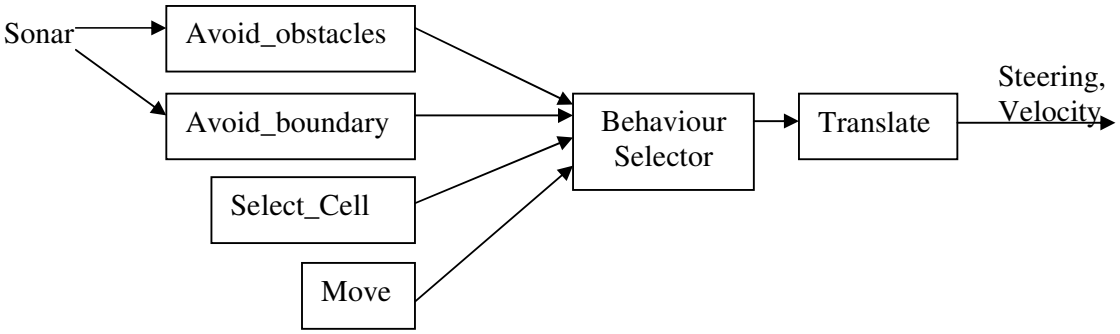


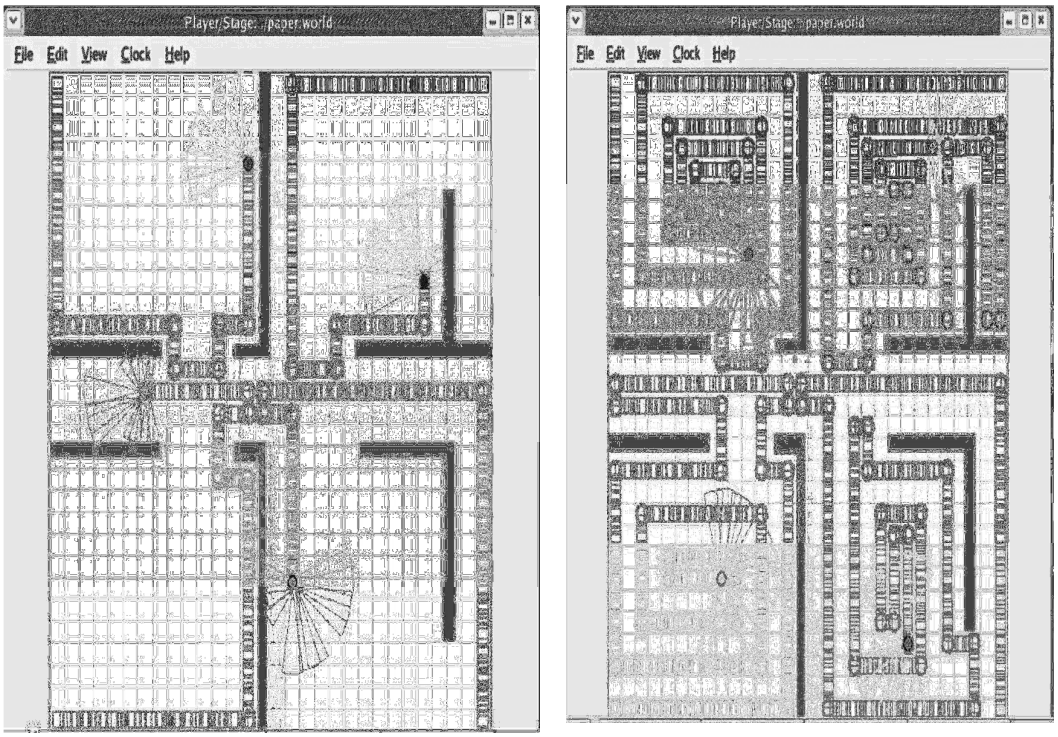
Figure 7. Behaviour based Autonomous Robot

We tested our algorithm using player-2.1.2 and stage-2.1.1 version on Linux operating system. We use SQL server to keep the visited cells information, to make the concurrent coordination among the multiple robots. The screenshots of the *Stage* simulator is shown in Figure 8, 9 and 10.

In this simulated experiments, four robots perform a cooperative exploration and coverage task in the terrain. The underlying rectangular grid pattern seen in the figure is added by *stage*, and each grid cell representing one square meter. The simulated sonar beams are shown in as thin lines radiating from the robot. The experiments presented in this paper are performed in two different simulated bitmapped environments. The first one is shown in figure 8 under the title "Simple Rooms". The second is shown in figure 9 under the title "out door" and figure 10 depicts an empty obstacle free environment. These environments are a modification of the bitmaps that come with the *stage* software.

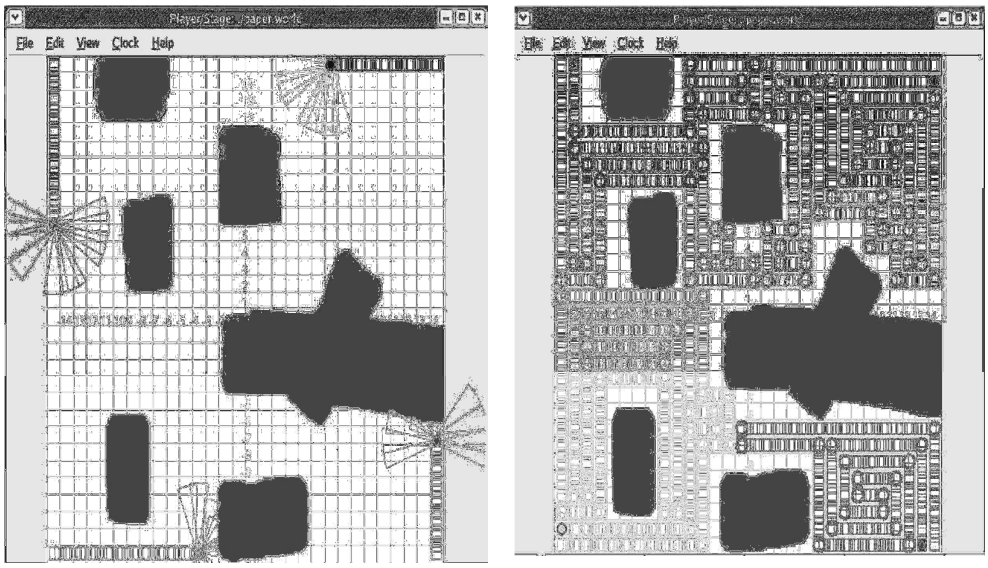
5. CONCLUSIONS

We have used the *Player/Stage*, an open source middleware for sensor/actuator systems, to implement our S-MSTC algorithm [20]. Some interesting aspects of our algorithm were investigated using *Player/Stage* middleware. We demonstrated how devices and our M-STC algorithms can be integrated into *Player/Stage* through experiments. And our opinion, the overall system, is more suitable in comparison to other existing middleware. Also currently *player/stage* development team is developing additional interfaces and drivers to include more wire-less sensor platforms, as well as semi automatic data processing tools, such as feature selection and learning models from acceleration data. So this simulator software will become more sophisticated in course of time and we are planning to utilize in our future work too.



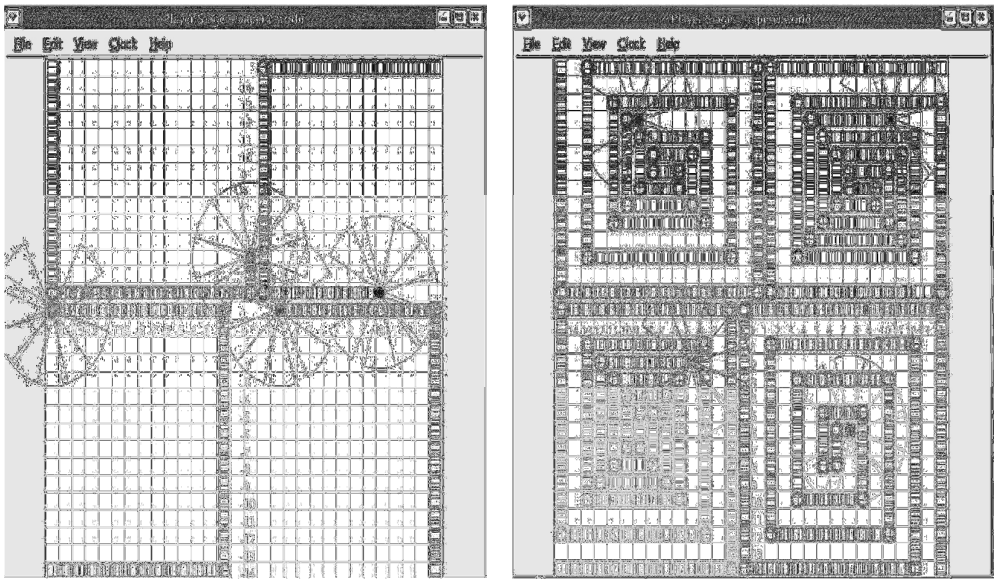
(a). Stage 1 (b). Stage 2

Figure 8. Indoor Environment – Simple Rooms



(a). Stage 1 (b). After completion

Figure 9. Outdoor Environment



(a). stage 1 (b). stage 2

Figure 10. An obstacle free Terrain

REFERENCES

[1]. B. P. Gerkey, R. T. Vaughan, and A. Howard, The Player/stage project: Tools for multi-robot and distributed sensor systems, in *Proceedings of the 11th International conference on advanced robotics,* 2003, pp. 317–323.

[2]. B. P. Gerkey, and R. T. Vaughan, Really Reusable Robot Code and the Player/Stage Project, *Software Engineering for Experimental Robotics, Springer Tracts on Advanced Robotics*, Springer, 2007.

[3]. B. P. Gerkey, R. T. Vaughan, K. Stoy , A. Howard, G. Sukhatme, and M. J. Matari´c, Most Valuable Player: A Robot Device Server for Distributed Control, in *Proceedings of the IEEE/RSJ International conference on Intelligent Robotic systems,* 2001, pp. 1226–1231.

[4]. T. H. Collett, B. A. MacDonald, and B. P. Gerkey, Player 2.0: Toward a Practical Robot Programming Framework, in *Proceedings of the Australasian Conference on Robotics and Automation (ACRA 2005)*.

[5]. R. T. Vaughan, B. P. Gerkey, and A. Howard, On Device Abstractions for Portable, Reusable Robot Code, in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robot Systems (IROS 2003),* 2003, pp. 2121-2427.

[6]. S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, USARSim: A Robot simulator for research and education, in *Proceedings of the IEEE international conference on robotics and automation,* 2007, pp. 1400–1405.

[7]. F. Mondada, G. C. Pettinaro, A. Guignard, I. Kwee, D. Floreano, J. L. Deneubourg, S. Nolfi, L. Gambardella, and M. Dorigo, SWARM-BOT: A new Distributed Robotic Concept, *Autonomous Robots*, *17*(2-3), 2004, pp. 193–221.

[8]. Cyberbotics: http://www.cyberbotics.com/products/webots/ (accessed July 2010).

[9]. Webots user guide release 5.1.0, 2005.

[10]. O. Michel, Webots: Professional Mobile Robot Simulation, *International Journal of Advanced Robotic Systems*, 1(1), 2004, pp. 39–42.

[11]. Newtonium: http://www.newtonium.com(accessed June 2010).

[12]. T. Balch, Behavioral diversity in learning robot teams, PhD thesis, College of Computing, Georgia Institute of Technology, 1998.

[13]. SPADES: http://spades-sim.sourceforge.net (accessed January 2010).

[14]. J. Craighead, R. Murphy, J. Burke, and B. Goldiez, A Survey of Commercial and Open Source Unmanned Vehicle Simulators, in *Proceedings of the IEEE International conference on robotics and automation,* 2007, pp. 852–857.

[15]. J. Kramer, and M. Schultz, Development Environments for Autonomous Mobile Robots: A Survey, *AutonomousRobots*, *22*(2), 2007, pp. 101–132.

[16]. Player/Stage Source Forge Homepage: http://playerstage.sourceforge.net (accessed June 2009).

[17]. Java Client: http://java-player.sourceforge.net/ (accessed January 2010).

[18]. LISP Client: http://www-robotics.cs.umass.edu/~bburns/software/player-lisp.html (accessed January 2010).

[19]. Player/Stage Drivers: Writing a Player Plug-in from the Penn State Robotics Encyclopedia RoboWiki : http://psurobotics.org/wiki/index.php?title=Player/Stage_Drivers. (accessed January 2010).

[20]. K. S. Senthilkumar, and K. K. Bharadwaj, Multi-Robot Terrain Coverage by Constructing Multiple Spanning Trees Simultaneously, *International journal of Robotics and Automation, ACTA Press*., vol. 25(3), 2010, pp. 195-203.

Authors

K. S. Senthilkumar received his B.Sc. and M.Sc degrees in Computer Science from University of Peradeniya, Sri Lanka in 1998 and 2003, respectively. Then, he received his M.Tech degree in 2006 and PhD degree in 2011 from Jawaharlal Nehru University, New Delhi, India. He did research on Multiple Robot Terrain exploration and coverage for his PhD degree in Computer Science. He is a lecturer in the Department of Computer Science, College of Art & Science, Wadi Al Dawaser, Alkharj University, Kingdom Of Saudi Arabia. His research area covers Robotics, Artificial Intelligence and Soft Computing. He has published five scientific papers in International conference proceedings and two papers in International Journals.

Kamal K. Bharadwaj is currently a professor in the School of Computer & Systems Sciences, Jawaharlal Nehru University (JNU), New Delhi, INDIA. He joined JNU in 1985 as an Associate Professor and since 1990 he is full Professor at JNU. Prior to JNU he has been a Faculty in the Computer Science Department, BITS, Pilani (Rajasthan), India. He received his M.Sc. degree in Mathematics from the University of Udaipur (Rajasthan), india and Ph.D degree in Computer Science from the Indian Institute of Technology (IIT), Kanpur, India. He has published widely in International journals and International conference proceedings. His current research interests include Machine Learning, Intelligent Systems, Knowledge Discovery in Databases (KDD), and Computational Web Intelligence