A Prototype Mobility Platform for a Ubiquitous Computing Testbed

Robert Grant
The University of Texas at Austin
bgrant@mail.utexas.edu

ABSTRACT

Ubiquitous computing is a relatively new field, and most UbiComp performance and usability claims are based on analytical results, simulation, or speculation. Though UbiComp applications are ambitious and attack complex problems, like all applications they need prototyping and testing to help take them from research concepts to deployable systems. This paper presents some first steps in creating a Ubiquitous Computing testbed based on the iRobot Create robotics platform. In it we detail building a mobile platform for nodes in such a system by equipping Creates with wireless-enabled embedded processors powered from the Create's rechargeable battery. We then describe programming the node to perform an example mobility model (Random Direction) and describe how this can be extended to more complex mobility patterns.

Categories and Subject Descriptors

H.1.2 [**Models and Principles**]: Miscellaneous
; H.4.3 [**Information Systems Applications**]: Communications Applications
; J.7 [**Computers in Other Systems**]: Command and Control

General Terms

Measurement, Experimentation, Design

Keywords

iRobot Create, mobile networking, testbed, robotics

1. INTRODUCTION

Currently most Ubiquitous Computing applications are still in the research stage or exploratory development stage, and many of their performance and usability claims are based on analytical results, simulation, or speculation. Unfortunately, due to the complexities of the wireless channel and interaction among heterogeneous software and hardware in the real world, it is hard to judge the realism of these published results and the accuracy of the accompanying claims. Though UbiComp applications are ambitious and attack complex problems, like all applications they need prototyping and testing to help take them from research concepts to deployable systems.

This paper presents the beginnings of a Ubiquitous Computing testbed. Specifically, we have developed a mobile node platform based on the iRobot Create [1]. After equipping the Creates with Gumstix processors running embedded Linux, the nodes become programmable mobile wireless devices capable of enacting complex mobility scenarios. As a proof-of-concept, we program a node to perform its part in a simple mobility model: Random Direction. Additionally, our mobility platform is easily programmable, and in the future it will support many more mobility models, carry payloads of many differing devices, and support prototyping and testing of varied UbiComp applications. In this way, we hope to scale this prototype up to a complex testbed.

The rest of this paper is organized as follows. In Section 2 we discuss construction of a testbed node. In Section 3 we describe mobility models we considered for an initial implementation and programming a node to enact one. We then discuss related and future work in Sections 4 and 5, and finally conclude in Section 6.

2. NODE CONSTRUCTION

In this section, we present our extensively revised technique for building a node that we have arrived at through much trial and error. In the near future we hope to provide detailed step-by-step procedure for constructing one of these nodes, but here we provide slightly higher-level view.

2.0.1 Node Hardware Setup

The hardware platform consists of an iRobot Create [1], a Gumstix Connex processor [2], a Robostix microcontroller [3], and a Wifistix 802.11g board [4]. A completely assembled node is shown in Figure 1. In this node, the Create is the controllable mobile platform, the Gumstix is a small form-factor computer running Linux that runs the controlling code, the Robostix allows the Gumstix to interface to the Create, and the Wifistix provides wireless connectivity via 802.11.

The essential procedure for assembling the node hardware is as follows:

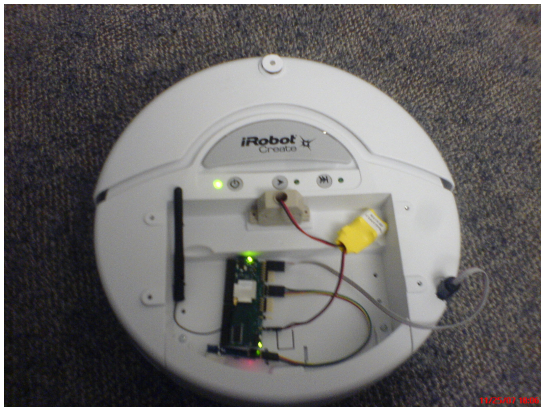

Figure 1: A complete node

1. Charge and test the Create using built-in demo functionality
2. Assemble and connect control hardware
   (a) Assemble the Gumstix stack
      i. The Gumstix fits onto the Robostix
      ii. The Wifistix fits onto the Gumstix
      iii. The antenna fits onto Wifistix
   (b) Build a serial cable to connect the Gumstix stack to the Create [5]
   (c) Connect the Gumstix stack to the Create via the serial cable
3. Connect the Gumstix stack to the Create's battery
   (a) Build a 25-pin connector to interface a voltage regulator with the Create battery through the Create's cargo bay connector
   (b) Connect the Gumstix stack to Create's battery through the voltage regulator (specifically, in this configuration you must connect the voltage regulator to the Robostix).
4. Power on the Create. LEDs on the Gumstix stack should light up.

2.0.2 Node Software Setup

The goal of our software setup is to get a Player server running on the Gumstix. Player is a piece of open source robotics software that acts as a Hardware Abstraction Layer for many different robotics platforms. It features a client/server architecture that allows you to run a Player server on a robot and then write a Player client to control the robot by communicating with the server over TCP. Because of the client/server architecture, the Player client can run on the robot along with the server or can run remotely on a PC with a TCP connection to the server. In our demonstration application we run the client on a remote machine.

To get the Player server onto the Gumstix, you must build a Linux filesystem image on a PC containing a version of Player cross-compiled for an ARM processor and then flash this filesystem into the Gumstix's memory. The basic procedure for doing this is enumerated below.

1. Test connectivity
   (a) Plug the USB adapter into the Gumstix stack, and connect a USB cable to it and a PC.
   (b) Configure Kermit, and connect to the serial port on which you are connected to the Gumstix (possibly ttyUSB0).
   (c) Boot the Gumstix, and you should see the Linux startup messages in Kermit.
   (d) Log into the Gumstix and setup Wifi connectivity in /etc/network/interfaces.
   (e) Test wirelessly connecting to the Gumstix over ssh.
2. Build a filesystem for the Gumstix on the PC
   (a) Log into the Gumstix and check the version of Buildroot that shipped on with it. Ours shipped with revision 1161 from the subversion repository, which also happens to be the last version people generally consider "stable" for the Gumstix Connex.
   (b) Check this version of Buildroot out of the project's Subversion repository.
   (c) Configure and build a base version of Buildroot with your desired configuration options and make sure there are no errors. We turned off all unnecessary software in the configuration script to save space, e.g. the embedded web server (boa).
   (d) Get the newest Player source, cross-compile it for ARM Linux, and install it in the target filesystem tree in Buildroot.
   (e) Build the Buildroot again, and it will produce the desired filesystem image in the toplevel of the source tree.
   (f) Using Kermit, transfer this filesystem image over to the Gumstix and flash it to its memory.
3. Prepare the Gumstix to act as a Player server controllable over a wireless network.
   (a) Log into the Gumstix and edit /etc/network/interfaces to bring up the wireless network interface by default. Also, configure it join your designated wireless network. Our configuration file is attached in Appendix B.
   (b) Add an script to /etc/init.d/ to start the Player server on bootup. An example is provided in Appendix C.
   (c) Add a roomba.cfg file to /root (or some other known location) for the Player server to read on startup. You must specify this location in your init script. Our roomba.cfg file is provided in Appendix A.

2.1 Node Construction Problems

We have completely constructed a single node as described above, and are able to control it via a Player client, which is discussed in Section 3.2. The discovery of the above procedure was not free of missteps, however.

We initially attempted to follow the procedure in the Robotics Primer [6] for setting up a Gumstix-controlled Create, but it was soon apparent that this workbook's procedure has some significant gaps in it.

2.1.1 Hardware Problems

As an example, the Robotics primer makes no mention of how to power the Gumstix stack while it's on the Create. Our initial testing of the Gumstix stack was done with it plugged into power via its AC adapter, but of course this is not acceptable for a platform intended to be mobile. We had assumed the Gumstix would receive power through the serial interface to the Create, but this proved incorrect. After some research we discovered a project out of the University of Alabama [7] that uses similar Create-based nodes for a robotics class. Their solution to the mobile power problem is to power the Gumstix stack through pins in the Create's cargo bay interface that connect to the Create's rechargeable battery. We have emulated their solution [8].

More specifically, the Robostix (where power must be attached in the Gumstix stack) takes 5V input and may supply over 500mA of current to its attached devices. This ruled out the convenient 5V, 100mA regulated voltage pin in the cargo bay connector due to insufficient current. However, the cargo bay connector also has pins that connect directly to the Create's rechargeable battery (12V, 1.5A) and can be used to power the Gumstix stack if regulated properly. This is why a voltage regulator [9] between the cargo bay connector and the Gumstix stack is needed.

To interface the voltage regulator to the cargo bay connector we cut off one of the regulator's connectors and soldered the leads to the appropriate pins on a 25-pin connector. This makes it easy to plug and unplug the control stack from the Create and ensures we always have a good connection. This detachable connector is very handy when moving a single Gumstix stack between different Creates for development and testing.

Fortunately, the construction of the serial cable needed to interface the Gumstix to the Create was well documented and went smoothly [5].

2.1.2 Software Problems

Additionally, the Robotics Primer Workbook makes no mention of how to actually get software onto the Gumstix. We overcame this by mining the Gumstix wiki [10], reading the UA Robotics website [11], and much trial and error.

Unfortunately, though the UA Robotics site was useful in getting an overall idea of what needs to be done, the particulars turned out to be wrong in many cases, at least for our setup. For example, their instructions say to check out the latest revision of Buildroot from the Subversion repository, but as it turns out, this version is not necessarily stable. In fact, it seems there ARE no stable versions of Buildroot except for a few that are available as downloads from Sourceforge that are also very old. After checking out, tweaking, and rebuilding the Buildroot many times to try to get it to build a kernel that supported our entire control stack (especially the wireless card), we stumbled upon this fact in some online forums.

It turns out that the best strategy is to not flash your operating system at all, but to find out what revision of Buildroot was used to build the operating system shipped on your hardware, check that out, build your filesystem with that, and not re-flash the OS itself at all (which also requires re-flashing the bootloader). This revision information can be found in the /etc/gumstix-release file on the Gumstix. Unfortunately, this means that you are stuck with the functionality of an old Buildroot, but this was not a problem for our setup. The version of the OS shipped with our Gumstix was built with the same very old version of Buildroot available on Sourceforge as a tar.gz (revision 1161). Apparently this is the only version considered "stable" for the Gumstix Connex, though there are newer "stable" versions for different hardware.

After clearing this hurdle, we ran into another problem. Though we were able to successfully build the basic Buildroot and verify that it was functional on the Gumstix, we were unable to cross-compile the newest version of Player successfully with its cross-compilation tools. The fix for this was to tweak some configuration settings for clibc to add RPC options.

Finally, yet another problem was that for some reason Buildroot was adding some extranous "-e"s in some of the files in the filesystem image, for example /etc/network/interfaces. This turned out to be a bug in Ubuntu Linux which was solved by relinking /bin/sh to /bin/bash (originally it was linked to a special, efficient system shell called dash).

Additionally, though not really a problem, we had to figure out how to make the Player server start automatically at boot time so we don't have to log into the Gumstix and manually start the process each time we power on the Create. This was just a matter of writing an init script and placing it in /etc/init.d/. We were able to use another script in that directory as a template. Our script is attached in Appendix C.

3. PROGRAMMING THE NODES

As a demonstration application, we have written a Player client in Python to control a node and have it move in a manner dictated by the Random Direction mobility model. For ease of experimentation, we run this client on a PC connected to the robot via an 802.11 wireless access point, but it would have been a simple matter to install Python on the node and run the client there.

3.1 Mobility Models

Because of the client/server architecture of Player, it is possible with this setup to centrally coordinate a group of nodes to enact a group mobility model, and it is likely this will eventually be done. However, as an initial step, our goal was to implement a simple model where each node acts independently. With this goal in mind, we considered several possible models for our initial implementation. Selected from a survey of mobility models [12]:

Random Walk: Each node chooses a direction, a speed, and either a constant distance or time for each "step". Until that distance has been travelled or that time

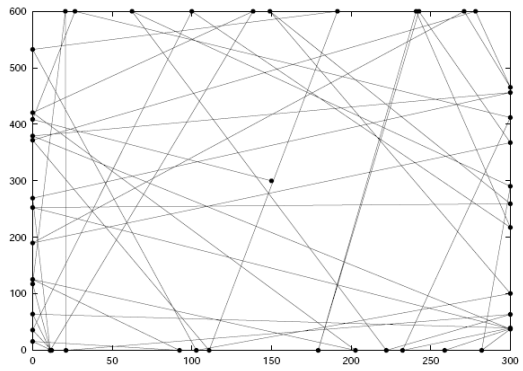

Figure 2: The Random Direction mobility model

has elapsed, the node travels in that direction at that speed. When the step is over, the node chooses a new direction and speed and repeats.

Random Waypoint: Like Random Walk, only at the end of each "step" the node pauses for a specified period. Also, this model usually specifies that a mobile node choose a new "location" rather than (direction, speed) after each step.

Random Direction: Similar to the previous two, except that in each step the node chooses a travel direction and then travels all the way to the edge of the boundary. This model was created to account for the density waves that occur in Random Waypoint. A visualization of this is shown in Figure 2.

Gauss-Markov: Adaptable to different levels of randomness via a tuning parameter. The new direction and speed at each step n is based on the direction and speed at step $n-1$. This can allow for a more realistic model by lessening the severity of direction and speed changes at each step if desired.

Except for maybe Gauss-Markov, these are all very simple independent-node algorithms for determining node paths. For simplicity we implemented Random Direction first, though it shouldn't be much of an extension to implement the others. These theoretical models do have problems, for example "speed decay" as noted in [13], but in practice the mobility patterns that real nodes enact is not equivalent to any of these theoretical models because in reality nodes collide with each other. This can be handled by beginning a new step when such a collision is detected, and this is what we do in our implementation of Random Direction.

It is unclear what significant effects node collisions have on the mobility pattern, and we have been unable to find any theoretical mobility papers addressing this issue. It should in some sense make the pattern more "realistic", since this is a real-world phenomenon.

Some issues may actually be more easily observed in a real implementation than in theory. For example, "realism" may be increased by placing actual obstacles within the movement area rather than through complex modelling of environmental obstacles as proposed in [14].

3.2 Client Code

As noted, to demonstrate the functionality of our constructed mobility platform we implemented a Player client in Python to control the node in enacting the Random Direction mobility model. This code is attached in Appendix D.

As seen in the Appendix, in this code we first abstract the Player interface to a slightly higher level. The Player API is designed to be flexible enough to make sense when used for controlling very different robotics platforms, but the cost of this flexibility is that it is not quite as elegant and tailored an API as we would like for controlling the iRobot Create. The interface is not complex, however, and we abstract our robot as a simple Node class which performs the necessary connections and initializations and then provides us with simple methods for controlling the robot. This script also contains the function do_random_direction() that instantiates a Node and uses our interface to have a Node perform the Random Direction model.

One advantage of using Python for the client software is that we can load this script from an interactive Python shell, instantiate Node objects, and run individual methods at will. This is a very convenient means of experimentation, and working at the level of this object model should allow very straightforward implementation of other mobility methods.

A disadvantage of using Python is that if we wanted to run our Python client on the Gumstix we would have to install the Python interpreter on it, which would take up 7MB more of valuable memory space. A C or C++ client script might be more appropriate for this scenario.

Also, it turns out that the Python API for Player is not very well documented. In fact, it's not documented at all. This API is generated from the C API using the Simplified Wrapper and Interface Generator (SWIG) [15], and until they document it you have to mentally translate from the C API (though you can also inspect modules and classes from within Python).

3.2.1 Testing

The independent-node mobility models are simple enough that it is easy to verify a node's behavior visually. However, as the behaviors we program become more complex it will be more difficult to do this. One solution is to record the Create's reported odometry data and analyze it offline for adherence to the constraints of our model. This is hard, because the odometry data reported by the Create is far from accurate, though perhaps it could be improved with calibration. A related idea is to do the same thing in a simulator using the same client code that we would run on a real node. The same open source project that produces Player also has a simulator called Stage that allows you to do just this.

4. RELATED WORK

For this initial prototype we have followed in the footsteps of other iRobot hobbyists and educators in creating our mobile nodes. In particular our nodes are similar to those outlined in the Robotics Primer Workbook [6] and in a project by the University of Alabama [7]. This former presents three documented options for experimenting with robotics on the

Create: using the iRobot Command module, using the Microsoft Robotics Studio and BAM, and using a stack of components (including the Gumstix and Wifistix) and the open source Player software [16]. For maximum capability and flexibility we have followed the third option for our initial node implementation. The University of Alabama project provided some additional information on powering the controlling hardware from the Create's battery [8].

In terms of testbeds, there are many, though they are often built for very different purposes. Some are mostly network and communications testbeds, some wired such as PlanetLab [17], some wireless but not mobile such as Hydra [18, 19, 20]. There have been many mobile wireless experiments using pedestrians to carry the nodes [21] or using people driving around in cars as mobile platforms. Because of the difficulty in organizing, funding, and repeating human-platform based mobile experiments, a robotic testbed is preferred over this method. Robotic platforms can also run for much longer periods of time than human-based experiments.

One sophisticated project that has created multiple remotely accessible testbeds is Emulab [22]. This project has a very mature wired testbed, and burgeoning fixed-position wireless, sensor-net, and robotic mobile testbeds. However, each of their networks are homogeneous, and it is the long term goal of this testbed to support heterogeneous network experiments.

Some very relevant related work is the MiNT project [23, 24], which has created a very sophisticated Roomba-based testbed. This testbed features a sophisticated GUI interface, node tracking though vision-based object tracking techniques, the ability to configure testbed experiments with the ns network simulator, the ability to "roll back" experiments to specific points in time, 24×7 operation (the Roombas recharge themselves when needed) and multiple wireless channels for communication between nodes and for feedback to a central system. However, this work was started before iRobot had a robot model meant for experimentation, so MiNT nodes are complex to set up since they involve hacking iRobot Roomba vacuums. Like Emulab, this testbed is also composed of homogeneous nodes.

5. FUTURE WORK

There is much potential for future work. In the long term, this testbed is meant to support Ubiquitous Computing experiments involving many nodes. The testbed will support each of these nodes running different hardware and software, allow them to be governed individually or in groups by different mobility models, and allow the testbed to be controlled locally or remotely.

Possible near term future work is to implement more independent-node mobility models and to construct more nodes. A step after that is to implement group mobility models which depend on coordination among multiple nodes; several group mobility models are also surveyed in [12]. Beyond that, more sophisticated testbed control features (inspired by the MiNT project) could be adapted to work on this testbed.

Once several nodes can be controlled reliably, a major goal is to start running UbiComp experiments involving mobile networks of heterogeneous nodes. These could involve ad hoc routing, resource discovery and access, or even high-level application or scenario experiments such as a model instrumented construction site. At a lower level, an interesting use of these mobile platforms would be to mount Universal Software Radio Peripherals (USRPs) and laptops on the nodes, allowing control of the entire communications stack as in the Hydra project but with the addition of control over mobility.

6. CONCLUSION

Real world experimentation is valuable for bridging the gap between theory and practice in any engineering endeavor. A robotic testbed such as this has the potential to provide the benefits of experimentation (realism) without the cost, limited operational time, and poor repeatability of using humans to carry mobile nodes.

In this paper we have demonstrated the feasibility of using iRobot Create robots as controllable mobile platforms for a Ubiquitous Computing testbed. We have constructed a Create-based node equipped with a Gumstix processor running a Player server, and using a Python Player client we programmed this node to perform the Random Direction mobility model. Given time and development, this demonstration indicates that this mobile platform is capable of supporting a complex and useful Ubiquitous Computing testbed.

7. REFERENCES

[1] iRobot Corporation: iRobot Create. http://www.irobot.com/sp.cfm?pageid=305.

[2] gumstix.com (gumstix product page). http://gumstix.com/store/catalog/product_info.php?products_id=136.

[3] gumstix.com (robostix product page). http://gumstix.com/store/catalog/product_info.php?cPath=31\&products_id=139.

[4] gumstix.com (wifistix product page). http://gumstix.com/store/catalog/product_info.php?products_id=171.

[5] Howto: Build a Mini-DIN to 4-pin TTL Serial Cable. http://roboticsprimer.sourceforge.net/workbook/Howto:_Build_a_Mini-DIN_to_4-pin_TTL_Serial_Cable.

[6] The Robotics Primer Workbook. http://roboticsprimer.sourceforge.net/workbook/Main_Page.

[7] Welcome to UA Distributed Autonomy Lab. http://robotics.cs.ua.edu/wiki/index.php/Main_Page.

[8] Connecting The Gumtix To The IRobot - UA Robotics. http://robotics.cs.ua.edu/wiki/index.php/Connecting_The_Gumtix_To_The_IRobot.

[9] 5.1V BEC Voltage Regulator - Use Li Poly/NiMH/NiCd Battery To Power RC Receiver & Servos. http://www.rctoys.com/rc-products/DF-51VR.html.

[10] Gumstix Support Wiki. http://docwiki.gumstix.org/Main_Page.

[11] Instructions for building player into the Gumstix OS for use with iRobot. http://robotics.cs.ua.edu/wiki/index.php/Gumstix.

[12] T. Camp, J. Boleng, and V. Davies. A survey of mobility models for ad hoc network research. *Wireless Communications and Mobile Computing*, 2(5):483–502, 2002.

[13] J. Yoon, M. Liu, and B. Noble. Sound mobility models. *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 205–216, 2003.

[14] A. Jardosh, E.M. Belding-Royer, K.C. Almeroth, and S. Suri. Towards realistic mobility models for mobile ad hoc networks. *Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 217–229, 2003.

[15] Welcome to swig. http://www.swig.org/.

[16] Player Project. http://playerstage.sourceforge.net/.

[17] Planetlab. http://www.planet-lab.org/.

[18] Hydra - a wireless multihop testbed. http://hydra.ece.utexas.edu.

[19] K. Mandke, S.H. Choi, G. Kim, R. Grant, R.C. Daniels, W. Kim, R.W. Heath Jr, and S.M. Nettles. Early Results on Hydra: A Flexible MAC/PHY Multihop Testbed. *Vehicular Technology Conference, 2007. VTC2007-Spring. IEEE 65th*, pages 1896–1900, 2007.

[20] K. Mandke, R.C. Daniels, S.H. Choi, S.M. Nettles, and R.W. Heath Jr. A MIMO demonstration of Hydra. *Proceedings of the the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 101–102, 2007.

[21] Robert S. Gray, David Kotz, Calvin Newport, Nikita Dubrovsky, Aaron Fiske, Jason Liu, Christopher Masone, Susan McGrath, and Yougu Yuan. Outdoor experimental comparison of four ad hoc routing algorithms. In *MSWiM '04: Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems*, pages 220–229, New York, NY, USA, 2004. ACM.

[22] Emulab - network emulation testbed home. http://www.emulab.net/.

[23] Mint. http://www.ecsl.cs.sunysb.edu/mint/.

[24] P. De, R. Krishnan, A. Raniwala, K. Tatavarthi, N. Syed, J. Modi, and T. Chiueh. MiNT-m: An Autonomous Mobile Wireless Experimentation Platform. *Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 124–137, 2006.

APPENDIX

A. THE ROOMBA.CFG FILE

```
driver
(
  name "roomba"
  provides ["position2d:0" "power:0" "bumper:0"]
  port "/dev/ttyS0"
  safe 1
)
```

B. THE /ETC/NETWORK/INTERFACES CONFIGURATION FILE

```
# Configure Loopback
auto lo
iface lo inet loopback

auto usb0
iface usb0 inet dhcp

iface bnep0 inet dhcp

auto eth0
iface eth0 inet dhcp

iface wlan0 inet dhcp

auto mwlan0
iface mwlan0 inet dhcp
pre-up /sbin/iwconfig $IFACE essid roombanet txpower 100mW
```

C. THE PLAYER SERVER INITIALIZATION SCRIPT

```
#!/bin/sh
#
# Starts player server.
#

# Make sure the roomba config file exists
[ -f /root/roomba.cfg ] || exit 0

start() {
  echo -n "Starting player server:"
start-stop-daemon --start --quiet --name player --exec /usr/local/bin/player /root/roomba.cfg &
echo "OK"
}
stop() {
echo -n "Stopping player server: "
start-stop-daemon --stop --quiet --name player
echo "OK"
}
restart() {
stop
start
}

case "$1" in
  start)
   start
;;
  stop)
   stop
;;
  restart|reload)
```

```
   restart
;;
  *)
echo $"Usage: $0 {start|stop|restart}"
exit 1
esac

exit $?
```

D. THE PLAYER CLIENT CODE

```
#!/usr/bin/env python

import math
import random
import functools
import time
from math import pi
from playerc import *

class Node:
    'Abstraction of an iRobot Create mobile node'

    def __init__(self, client_address):
        'Create a client object and subscribe to some sensor proxies'

        # Create a client object
        self.client = playerc_client(None, client_address, 6665)

        # Connect it
        if self.client.connect() != 0:
          raise playerc_error_str()

        # Create a proxy for position2d:0
        self.pos2d = playerc_position2d(self.client, 0)
        if self.pos2d.subscribe(PLAYERC_OPEN_MODE) != 0:
          raise playerc_error_str()

        # Create a proxy for the bumper
        self.bumper = playerc_bumper(self.client, 0)
        if self.bumper.subscribe(PLAYERC_OPEN_MODE) != 0:
          raise playerc_error_str()

    def disconnect(self):
        'Close down the connections'
        self.bumper.unsubscribe()
        self.pos2d.unsubscribe()
        self.client.disconnect()

    def velocity(self, speed, radians_per_second):
        'Set the linear and angular velocities'
        self.pos2d.set_cmd_vel(speed, 0.0, radians_per_second, 1)

    def stop(self):
        'Set the velocity parameters to stop the node\'s motion'
        self.velocity(0.0, 0.0)

    def turn_speed(self, radians_per_second):
        'Set the angular velocity of the node in radians/s.  Very inaccurate.'
        self.velocity(0.0, radians_per_second)

    def turn_angle(self, radians):
        '''Attempt to turn a specific anglular amount by turning at particular
```

```
        angular velocity for a certain amount of time.  Must be calibrated with
        internal angle_calibration_factor.'''
        angle_calibration_factor = 0.52

        self.read()
        original_angle = self.get_angle()
        speed = pi/8

        if radians >= 0:
            self.turn_speed(pi/8)
        else:
            self.turn_speed(-pi/8)

        time.sleep(abs(radians/speed * angle_calibration_factor))
        self.stop()

    def forward(self, speed):
        'Set a node\'s linear velocity only'
        self.velocity(speed, 0.0)

    def read(self):
        '''Command to pull sensor data from the node.  Must be polled constantly
        or node message queue will overflow and data will be lost.'''
        self.client.read()

    def get_bumper(self):
        'Get the last read() bumper data.'
        #self.bumper.bumpers returns a list of 32 values, but it looks like
        #only the first two are set
        return (self.bumper.bumpers[0], self.bumper.bumpers[1])

    def get_position(self):
        'Get the last read() position data (x pos, y pos, angular pos).'
        return (self.pos2d.px, self.pos2d.py, self.pos2d.pa)

    def get_angle(self):
        'Get the last read() angular position data.'
        return self.pos2d.pa

def random_angle():
    'Pick a turn angle in union([-pi/2,-pi],[pi/2, pi])'

    angle = random.uniform(-pi/2, pi/2)
    if angle < 0:
        angle -= pi/2
    else:
        angle += pi/2

    return angle

def do_random_direction():
    'Have a node perform the Random Direction mobility model'

    node = Node('192.168.1.2')
    node_speed = 0.2 # m/s

    try:
        # go forward until node runs into something
        node.forward(node_speed)

        while 1:
            node.read()
            print node.get_bumper()
```

```
            if sum(node.get_bumper()) != 0:
                # if the node runs into something, back up a bit to get
                # untangled
                node.forward(-node_speed)
                time.sleep(1)
                node.stop()

                # clear the next few sensor samples or else the node will think
                # it's still bumping against something and
                for x in range(5):
                    node.read()

                print node.get_bumper()

                # pick a new angle and go again
                node.turn_angle(random_angle())
                node.forward(node_speed)
    finally:
        # it seems not to hurt anything if we don't disconnect at the end, but
        # we should clean up anyway
        node.disconnect()

def main():
    '''Nothing in main() right now.  I\'ve been calling the functions from the
    interactive shell'''
    pass

if __name__ == '__main__':
    main()
```