Construction of a Mobile Reconnaissance Robot

Derek Sewell

July 6, 2013

Chapter 1

Project Brief

1.1 Background Information

Robots are seeing an increased use in today's society. They are used in a variety of different areas, ranging from household cleaning (such as the Roomba[1]) to national defence. With the advent of cheaper and more accessible consumer electronics it has never been easier to construct your own robot. Using newly released electronics designed to make building things such as robots easier, I shall attempt to construct my own functioning robot and document this process.

1.2 Project Proposal

The first step was to decide what type of robot to make. The word *robot* is fairly ambiguous and thus it is difficult to sort it into different sub sections. *Robot* is defined as *a machine capable of carrying out a complex set of instructions automatically* [2]. This discounts remote control vehicles from being robots as they are not autonomous, so in order for my creation to be counted as a robot it must have at least some level of autonomy.

Since this is my first attempt at a project of this scale, I think it best that the robot should be kept simple yet functional. To achieve this I shall create small size box vehicle which operates remotely. This keeps the components of the robot inexpensive whilst providing enough service for the robot to be considered useful and thus worth creating.

1.2.1 Purpose

The main purpose of the robot other than proof of concept shall be for reconnaissance. Using cellular networks (more on that in the research section) the range that the robot could travel would only influenced by the battery life, availability of mobile phone signal and terrain. This enables the robot to be a vessel of exploration for the surrounding area. It could also be used to explore areas with dangerous hazards, so that human life is not risked in the process. However in practice I do not have access to such environments so this is a theoretical use of the robot only.

[1]An automated vacuum - http://www.irobot.com/us/learn/home/roomba.aspx

Chapter 2

Research

2.1 Computing platform

In order to transmit footage to the user, the robot will need to be equipped with a camera. The cheapest type of camera available for purchase would be a webcam. A webcam is typically used for visual contact whilst talking over VoIP[1]. Because of their limited functions webcams contain very little equipment apart from the camera. They have no storage mechanisms and typically no way to transmit their video other than through USB. In order to use these webcams I will need a computing platform which supports webcams and has the ability to connect to the internet.

The perfect platform for the job would be the Raspberry Pi (Referenced sometimes as just Pi). Released on the 29 February 2012 the Raspberry Pi sold out in about 15 minutes, amassing over two million expressions of interest or pre-orders[3]. The orders have died down since then and I was able to purchase one for the robot. The Raspberry Pi is essentially a credit card sized computer capable of performing many of the functions a much larger and more expensive desktop computer can. These functions include interfacing with a webcam and transmitting and receiving data via the internet. The Raspberry Pi was designed to run on Linux and the particular flavour of linux I shall be using will be Rasbian. Raspbian is based off the Debain operating system (another Linux Distribution) and is optimised for the Raspberry Pi.

2.2 Microcontroller

A microcontroller is a very small computer on a single integrated circuit which contains programmable input/output peripherals. Although the Raspberry Pi can function as a microcontroller it is not optimal. This is because the overhead of an operating system adds significant and unpredictable delay to commands sent to the IO ports.

What would be better is a dedicated microcontroller to control the motors and other devices which the robot may need. There are a number of available options for a microcontroller, but by far the most prolific of these is the Arduino. The Arduino is small and cheap microcontroller. Although it has not nearly the amount of computing power that the Raspberry Pi does (16 MHz clock speed compared to the Raspberry Pi's 700 MHz[1]) the Arduino does not have an

[1]Voice over Internet Protocol - Talking to people through the internet instead of on the phone, used in applications such as skype

operating system, instead of runs off a Atmel 8-bit AVR microcontroller. This enables for it to operate in real time and send instructions to the IO ports much faster than the Raspberry pi would be able to. The instructions are written in a high level language, in this case C, they are then translated to compact machine code for storage in the microcontroller's memory.

Using the Python programming language and the library PiSerial it is possible to control the Arduino using the Raspberry Pi. The reason for doing this is that the wireless networking facilities for the Arduino are expensive and difficult to configure. Using the Raspberry Pi's superior storage and processing speed it will be able to run long complex programs (for AI, ect) and send the necessary input to the Arduino so it can control the motors as needed.

2.3 Structural Components

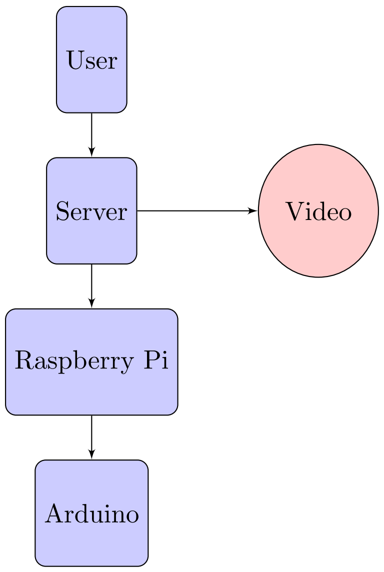
I will be using a combination of sheet metal and perspex plastic to construct the box like structure necessary to house the components listed above. The robot will use caterpillar tracks instead of tradition wheels. This is because the wheels tend to slip and slide a lot, and because of this it will be difficult to calculate the distance travelled, a necessary component for automation should I choose to add it sometime in the future. This may be an unnecessary precaution because the complexes and intricacies of AI design go well beyond the scope of this project.

I ordered the caterpillar tracks from I was pleased for the quality but not with the size. The small size only allows around a 11x5cm base which is going to make fitting the components inside the chassis a challenge.

2.4 Software

Software is a set of machine readable instructions, whilst hardware is the machine which runs the software. Examples of software include Internet Explorer and Firefox. The key difference between the two is that Firefox is open source, whilst Internet Explorer is closed source. Open source is when the code, the set of machine readable instructions is available to the public at no cost whatsoever. This is beneficial to the public as it allows them to inspect the software for themselves, to insure that it is suitable for their needs and has no security holes. It also allows people to modify and customise it to suite their own purposes. For example Raspbmc, as mentioned earlier is a modified version of the Debain operating system which in turn is built upon the Linux kernel. This would not be possible if Linux and Debain were closed source.

As mentioned before I shall be using Raspbmc on the Raspberry Pi. The Raspberry Pi will be hooked up to either a USB mobile broadband dongle or a WiFi dongle depending on whether a suitable Wifi hotspot is in range. In turn I will construct a program to communicate with a server, enabling the a user to connect to the server and issue the robot commands. The full process is described in the diagram below.



The reason that the user has to communicate to the Raspberry Pi through a server is because whilst roaming the Raspberry Pi will have a constantly changing IP address. This will make it impossible for the user to connect directly to the Raspberry Pi as the user would not know the IP address of the Raspberry Pi at any one time. Instead the Raspberry Pi will be constantly pinging a server with a static IP (that is, an IP which does not change). The user can then connect to the server and send commands to it which in turn the server will relay to the Pi, as shown in the diagram.

This will allow the user and the robot to communicate to each other without them having to know each others IP address. Due to the cost of hiring a dedicated server, I will be hosting the server from a home computer. This shall allow me full control over the server with no additional cost added to the project. Because the server is only temporary I will be using the Windows 7 operating system to host it, as this is the operating system already installed on the computer. If the server were to become more permanent I would install an OS better suited to hosting, such as the Linux distribution *Red Hat Linux*.

Chapter 3

Production

3.1 Programming

In order for the robot to function it must be able to carry out a specific set of instructions. Writing these instructions is called *programming*.

3.1.1 Server

The first program I will write will be for the server. The main aims of this program shall be:

1. Accept incoming ping's from the robot, establish connection to the robot.
2. Accept incoming commands from an external client. Relay these commands to the already connected robot.

This will allow the user to control the robot without having physical access to the machine or knowledge of its IP address. I started off with a few lines of code which will echo back any incoming messages. The following code is written in the language Python 1.7.

Listing 3.1: serverecho.py

```
"""
A server which echos any messages sent to it.
"""

import socket
host = ''
port = 25565
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host,port))
s.listen(backlog)

print "Listening"
while 1:
        client, address = s.accept()
        print "Found␣client"

        data = client.recv(size)

        if data:
                print "Revived␣data:␣" + str(data)
                client.send(data)

        client.close()
        print "Client␣closed"
```

I then changed this code to relay messages sent from one client (the user) to a different client (the robot) as described in the software section. The new code is as follows.

Listing 3.2: relayserver.py

```
"""
A server which echos any messages sent to it.
"""

import socket
import thread
import time
import sys


command = "0"
host = ''
port = 25565
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(backlog)


def listenOnClient(client):
    """Listens for incomming data from the
    client and overrides the command variable
    once data has been recived."""
    while True:
        data = client.recv(size)
        if data:
            print "Received:␣\"" + data + "\"␣from␣user"
            command = data


def relayOnClient(client):
    """Sends the last received command to the robot, effectively relaying the command."""
    while True:
        time.sleep(0.1)
        #Constantly sends the last received command to the client
        client.send(command)


print "Listening"

#Constantly listens for new clientswhile 1:
client, address = s.accept()
print "Found␣client␣at:␣" + str(address)
data = client.recv(size)

while True:
    if data == "user":
        while True:
            print "new␣user␣connected"
            #Starts a new thread for listening to the client
            thread.start_new_thread(listenOnClient(client), ())

    if data == "robot":
        print "new␣robot␣connected"
        #Starts a new thread for sending to the robot
        thread.start_new_thread(relayOnClient(client), ())
```

This works by obtaining connections to both the robot and the user. The program then waits on any messages sent by the user. When a message is recived it overwrites the `command` variable with the given message. Whilst it is doing this, the program constantly sends the `command` variable to the robot, every 0.1 seconds, as long as the robot is currently connected to the server.

3.1.2 Clients

Robot

I began as I did with the Server program. I created a simple program which will connect to a server, send the server a simple message and then wait to receive one back. Once the program has received the message it will print out the message before terminating. The program is as follows.

Listing 3.3: clientecho.py

```
"""
The client for serverecho.py
"""

import socket

host = 'derekbot.servegame.com'
port = 25565
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))
print "connected"
s.send('Hello,␣world')
data = s.recv(size)
s.close()

print 'Received:', data
```

I then modified the program to maintain a constant connection with the server instead of terminating once it has received a messaged. I also made the program send the message `user` once connected which will allow `relayserver.py` to initiate the `listenOnClient` thread.

Listing 3.4: clientconnect.py

```
"""
The client for serverecho.py
"""

import socket
import time

host = 'derekbot.servegame.com'
port = 25565
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))
print "connected"
s.send('user')
time.sleep(2)

while True:
    time.sleep(2)
    s.send('Ahoy')
```

The output of running `clientconnect.py` directly after running `clientconnect.py` is shown below.

Listing 3.5: echolog.txt

```
relayserver.py: Listening
clientconnect.py: Connected
relayserver.py: Found client at: ('31.52.72.62', 53943)
```

However the robot needs to identify itself as `robot` instead of `user`. This is simply changed by replacing `s.send('user')` with `s.send('robot')`. And because the robot only receives commands and does not give them, sending constant updates as shown `clientconnect.py` is pointless. Instead it would be much better to only receive messages and then act upon them. I revised the code to implement these changes.

Listing 3.6: robotconnect.py

```
"""
The client for serverecho.py
"""

import socket
import time

def move(string):
    pass

host = 'derekbot.servegame.com'
port = 25565
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

s.connect((host,port))
print "connected"
s.send('robot')
time.sleep(2)

while True:
    data = s.recv(size)
    if data:
        print "Received:␣\"" + data + "\"␣from␣user"
        move(data)
```

The program now needs to connect to the Arduino and pass on the given messages so that the Arduino can control the motors which in turn move the robot. In order to do this I will need to use programming library, or modules as they are called in Python. A library is a set of code which allows the programmer to easily accomplish something which would have taken a long time otherwise. A suitable analogy would be that cooking recipes do not list the instructions that the chef would need to carry out in order to make flour. The flour is already made and the chef does not need to know how flour is made or what it is composed of. A programming library is an aready made piece of code which simplifies the development of the rest of the program.

Most programs contain a set of stranded libraries which are included by default. However it would be impossible to include enough libraries that would-cover every possible situation. Instead the developers of the language decide upon which features would be most commonly needed and then design libraries in order to provide those features. Since communicating between the Pi and the Arduino via USB serial is a substandard case, a library to do such a thing is not included in Python's standard libraries. Instead I will use a user made library to accomplish this task. My library of choice is PySerial[4].

In order to run the following program the computers Python instillation must have the PySerial library installed. This is done simply on the Raspberry Pi by typing the following command in the terminal `pip install PySerial`. The process should be the same on all Linux based operating systems with Python 2.7 installed. PySerial allows for messages to be sent via USB Serial. I'm using this to send the messages that Raspberry Pi receives to the Arduino. I made this simple program which will receive incoming messages from the relay server and send them over the USB cable by using PySerial.

Listing 3.7: "piclient.py"

```
"""
The client for serverecho.py
"""

import socket
import serial

host = 'derekbot.servegame.com'
port = 25565
size = 1024

while True:
    try:
        print "Attempting␣to␣connect␣to␣Arduino."
        ser = serial.Serial('COM3', 9600)
        ser.setDTR(False)
        print "Connected␣to␣Arduino"
        while True:
            try:
                print "Attempting␣Connection␣to␣server"
                s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
                s.connect((host, port))
                print "Connected␣to␣Server"
                while True:
                    data = s.recv(size)
                    ser.write(data)
                    print "Received␣data:␣" + data
            except socket.error, (value,message):
                if s:
                    s.close()
                print "Connection␣Failed:␣" + message
    except serial.serialutil.SerialException, message:
        print message
```

I deigned the client in such a manner that if it fails to connect to the server it will try again and again as long as the program is running This is useful as most of the time I will not have physical accesses to the Raspberry Pi, so I need to insure that the program remains running. I accomplished this continuous running through the use Exceptions. By using exceptions one can "catch" an error and then tell the program to something special in regard to this error. In this case when the client failed to connect or it disconnected it threw a `connection failed` error. Normally this would result in the program crashing, but because the program now catches any errors the program continues running after closing any opened sockets. Because the connecting code in contained within the `while True:` block the program will attempt to connect again after reviving the disconnection error.

The messages sent to the Arduino are in the format listed in key.txt

Listing 3.8: "key.txt"

```
xyzzz
x = motor, 1 or 2
y = type:
        f = forwards
        b = backwards
        r = Release
zzz = power from 0-255
```

So if I wanted to move the first motor forwards at maximum power I would send a message to the Arduino saying "f 1 255".

In order to move the motors using the Arduino I used the Adafruit motor shield[1] and the accompanying library[2]. In order interpret the commands as listed above I used the example made by Mushfiq Sarker[6].

Listing 3.9: "arduinoserial.ino"

```
// Serial Read - Read commands and inputs from serial window
// Developed by: Mushfiq Sarker ( http://inventige.com )

/* How to use:
*  Send commands such as: 'a 10 1000'
*  The code will strip out command a and then individually the numbers.
*  Can be used to do a menu type system.
*/

/* To do:
*  Cannot handle negative numbers.
*  Cannot handle decimal (floating point) values.
*/

#include <AFMotor.h>


/************************* Private Variables****************/
unsigned long loopTime, currentTime;
const int serStrLen = 30;
char serInStr[ serStrLen ];  // array that will hold the serial input string
AF_DCMotor motor1(1, MOTOR12_64KHZ);
AF_DCMotor motor2(2, MOTOR12_64KHZ);
/***********************************************************/



/************ Setup ******************************/
void setup() {
  Serial.begin(9600);

  Serial.flush();
  Serial.println("Command␣example:␣␣a␣<input1>␣<input2>");

}
/************************************************/
```

[1]http://www.adafruit.com/products/81

[2]https://github.com/adafruit/Adafruit-Motor-Shield-library

```
/************* Main Arduino Loop ******************************/
void loop(){

  if( readSerialString() ) {

    unsigned long input1, input2;

    Serial.println(serInStr);  // Print the command for user to see.
    char cmd = serInStr[0];    // First character is the command
    char* str = serInStr;      // Save the remaining string for inputs.

    while( *++str == '␣' );    // After command, remove all spaces
    input1 = atoi(str); // Next command is input1, capture and save it.
    while( *++str == '␣' );    // After input1, remove all spaces
    input2 = atoi(str);  // Next command is input2, capture and save it.

    delay(30);                 // Delay to settle the UART buffer.

    /* input1, input2 CONTAIN the input numbers to be used throughout program.*/

    /*** Act on what was given *******/
    if(input1 == 1)
    {
        motor1.setSpeed(input2);
        Serial.println("Set␣motor␣1␣speed␣to:␣");
        Serial.println(input2);

        if(cmd == 'f') {
           motor1.run(FORWARD);
           Serial.println("Running␣motor␣1␣forwards");
        }

        if(cmd == 'b') {
           motor1.run(BACKWARD);
           Serial.println("Running␣motor␣1␣backwards");
        }

        if(cmd == 'r') {
           motor1.run(RELEASE);
           Serial.println("Running␣motor␣1␣release");
        }

    }

    if(input1 == 2)
    {
        motor2.setSpeed(input2);
        Serial.println("Set␣motor␣2␣speed␣to:␣");
        Serial.println(input2);

        if(cmd == 'f') {
           motor2.run(FORWARD);
           Serial.println("Running␣motor␣2␣forwards");
        }

        if(cmd == 'b') {
           motor2.run(BACKWARD);
           Serial.println("Running␣motor␣2␣backwards");
        }

        if(cmd == 'r') {
           motor2.run(RELEASE);
           Serial.println("Running␣motor␣2␣release");
        }

    }

  }
}
/*******************************************************************/

/************ EXTRA FUNCTIOONs ***********************************/

uint8_t readSerialString()
{
  if(!Serial.available()) {
    return 0;
  }
  delay(10);  // wait a little for serial data

  memset( serInStr, 0, sizeof(serInStr) ); // set it all to zero
  int i = 0;
  while(Serial.available() && i<serStrLen ) {
    serInStr[i] = Serial.read();   // FIXME: doesn't check buffer overrun
    i++;
  }
  return i;  // return number of chars read
}

/*****************************************************************************/
```

This program will receive messages from the Raspberry Pi running the `piclient.py` script and then move the motors in accordance with the message sent as detailed in `key.txt`. This is the last program which will run on the Raspberry Pi and it is absolutely critical that it runs without fault as the embedded nature of the platform will make it very difficult to change without physical accsess to the robot.

User

The user will connect to the server through a website. The website will use websockets to connect to the server. Websockets are a relatively new protocol. They are part of the JavaScript programming language and enable client side communication over the internet. In order for the user to use the website it must have a GUI, a graphical user interface. This is done through the use of HTML and CSS. Because my HTML/CSS skills are lacking somewhat I opted to use the interface designed by Andy Harris[**?**]. But with some adjustments to accommodate for the messaging the relay sever and thus robot with the right commands.

Listing 3.10: usersite.html

```
<script language="javascript" type="text/javascript">

    var output;
    var websocket;
    function init() {
        output = document.getElementById("output");
    } // end init
    function connect() {
        //open socket
        if ("WebSocket" in window) {
            websocket = new WebSocket("ws://derekbot.servegame.com:8888/ws");
            //note this server does nothing but echo what was passed
            //use a more elaborate server for more interesting behaviour
            output.innerHTML = "connecting...";

            //attach event handlers
            websocket.onopen = onOpen;
            websocket.onclose = onClose;
            websocket.onmessage = onMessage;
            websocket.onerror = onError;
        } else {
            alert("WebSockets not supported on your browser.");
        } // end if
    } // end connect
    function onOpen(evt) {
        //called as soon as a connection is opened
                    websocket.send("user")
        output.innerHTML = "<p>CONNECTED TO SERVER</p>";
    } // end onOpen
    function onClose(evt) {
        //called when connection is severed
        output.innerHTML += "<p>DISCONNECTED</p>";
    } // end onClose;
    function onMessage(evt) {
        //called on receipt of message
        output.innerHTML += "<p class = 'response'>RESPONSE: "
                + evt.data + "</p>";
        document.getElementById('output').scrollTop = 90000;
        //resize();
    } // end onMessage
    function onError(evt) {
        //called on error
        output.innerHTML += "<p class = 'error'>ERROR: "
                + evt.data + "</p>";
    } // end onError
    function sendMessage() {
        //get message from text field
        txtMessage = document.getElementById("txtMessage");
        message = txtMessage.value;
        //pass message to server
        websocket.send(message);
        output.innerHTML += "<p>MESSAGE SENT: " + message + "</p>";
    } // end sendMessage

    function resize() {
        // 200 is the total height of the other 2 divs
        var height = document.getElementById('output').offsetHeight - 200;
        document.getElementById('rest').style.height = height + 'px';
    };
    setInterval("loop()", 100);

    me.input.bindKey(me.input.KEY.A, "left");
    me.input.bindKey(me.input.KEY.D, "right");
    me.input.bindKey(me.input.KEY.W, "up");
    me.input.bindKey(me.input.KEY.S, "down");

    function loop() {
        if (me.input.isKeyPressed('left'))
        {
            websocket.send("f 1 255")
            websocket.send("b 2 255")
        }
        else
        if (me.input.isKeyPressed('right'))
        {
            websocket.send("b 1 255")
            websocket.send("f 2 255")

        }
        else
        if (me.input.isKeyPressed('up'))
```

```
{
    websocket.send("f 1 255")
    websocket.send("f 2 255")
}
else
if (me.input.isKeyPressed('down'))
{
    websocket.send("b 1 255")
    websocket.send("b 2 255")
} else
{
    websocket.send("r 1 255")
    websocket.send("r 2 255")
```

I omitted the CSS code and chose only to display the logic code in the example in order to conserve space. The full html file can be found at `usersite.html` with CSS intact.

The above code connects to the relay server when the user presses the `connect` button. Upon a successful connection the client identifies itself as a user in the `onOpen` function. Once connected the client checks to see if any arrow keys have been pressed every 100 milliseconds. If they have it sends the appropriate command to relay server. This enables the user to control the robot through use of the arrow keys on their keyboard. This may need to be updated in the near future to accommodate for mobile devices who do not have a physical keyboard. It also allows the user to send commands manually via the text box below the output field. The arrow keys however significantly increase the speed at which the user may react. Typing in command manually will enable the user to have very precise control over the robot. If the user fails to press any key the client will send the message to indicate the robot should move no motors. This still allows the robot to coast if it was moving before, but will prevent it from veering off uncontrollably if it loses connection from the relay server.

If I wish for the response time between the user pressing the arrow keys and the robot responding to be shorter, I could change the function `loop` to run at shorter intervals than every 100 milliseconds.

3.1.3 Connecting

After testing `relayserver.py` with `usersite.html` a problem soon became evident, the protocol which websockets uses and the raw TCP protocol which `relayserver.py` uses are not initially compatible. After further research I found a website[5] which explains the differences between the two and offers a possible solution. However after many attempts I could not replicate Yang Zhang's method successfully. Instead I opted for the Tornado library which promised to make the connecting to WebSockets a lot quicker and cleaner than my previous attempts. I wrote a simple server application which will echo any received messages. This was based on the echo test provided by the Tornado documentation[3]

Listing 3.11: websocketserver.py

```
import tornado.websocket
```

[3]http://www.tornadoweb.org/en/stable/

```
import tornado.ioloop
import tornado.httpserver
import tornado.web


class WSHandler(tornado.websocket.WebSocketHandler):
    def open(self):
        print 'New␣connection␣was␣opened'
        self.write_message("Welcome␣to␣my␣websocket!")

    def on_message(self, message):
        print 'Incoming␣message:', message
        self.write_message("You␣said:␣" + message)

    def on_close(self):
        print 'Connection␣was␣closed...'

application = tornado.web.Application([(r"/ws", WSHandler)])
http_server = tornado.httpserver.HTTPServer(application)
http_server.listen(25565)
http_server.start()
tornado.ioloop.IOLoop.instance().start()
```

I used WebSocket echo test website[4] in order to test whether the server was working. Initially there was no response on either side, but after an hour of debugging I found out that is was the function call `tornado.web.Application([(r"/ws", WSHandler)])` causing the problem, particularly the `"/ws"` string. The `"/ws"` string made it so that in order to successfully connect to the server you have to add `"/ws"` to the end of the address. After changing `ws://derekbot.servegame.com:25565` to `ws://derekbot.servegame.com:25565\ws` everything worked perfectly.

After determining the success of the echo test, I then implemented it into the original `relayserver.py`. The updated changes have been saved as `relayserverfinal.py`. I also implemented the error catching as seen `piclient.py` and for the same reasons as listed before.

Listing 3.12: relayserverfinal.py

```
"""
A server which echos any messages sent to it.
"""

import socket
import thread
import time
import tornado.websocket
import tornado.ioloop
import tornado.httpserver
import tornado.web
import sys
command = "0"
host = ''
port = 25565
backlog = 5
size = 1024
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(backlog)


class WSHandler(tornado.websocket.WebSocketHandler):
    def open(self):
```

[4]http://www.websocket.org/echo.html

```
        print 'New␣connection␣was␣opened'
        self.write_message("Welcome␣to␣my␣websocket!")

    def on_message(self, message):
        print 'Incoming␣message:', message
        command = message
        client.send(command)

    def on_close(self):
        print 'Connection␣was␣closed...'

def listenOnClient(client):
    """Listens for incomming data from the
    client and overrides the command variable
    once data has been recived."""
    while True:
        data = client.recv(size)
        if data:
            print "Received:␣\"" + data + "\"␣from␣user"
            command = data


def relayOnClient(client):
    """Sends the last received command to the robot, effectively relaying the command."""
    while True:
        time.sleep(0.1)
        #Constantly sends the last received command to the client


application = tornado.web.Application([(r'/ws', WSHandler)])
http_server = tornado.httpserver.HTTPServer(application)
http_server.listen(8888)
http_server.start()
thread.start_new_thread(tornado.ioloop.IOLoop.instance().start, ())

while True:
    try:
        print "Listening"
        #Constantly listens for new clientswhile 1:
        client, address = s.accept()
        print "Found␣client␣at:␣" + str(address)

        print "new␣robot␣connected"
        #Starts a new thread for sending to the robot
        thread.start_new_thread(relayOnClient(client), ())

    except socket.error, (value, message):
        if s:
            s.close()
        print "Connection␣Failed:␣" + message
```

3.1.4 Assembly

The first component assembled was the chassis. The chassis are in the form of a 110x85x70mm 3mm acrylic box. Each side of the box was first designed in the computer program called 2D Design[5].

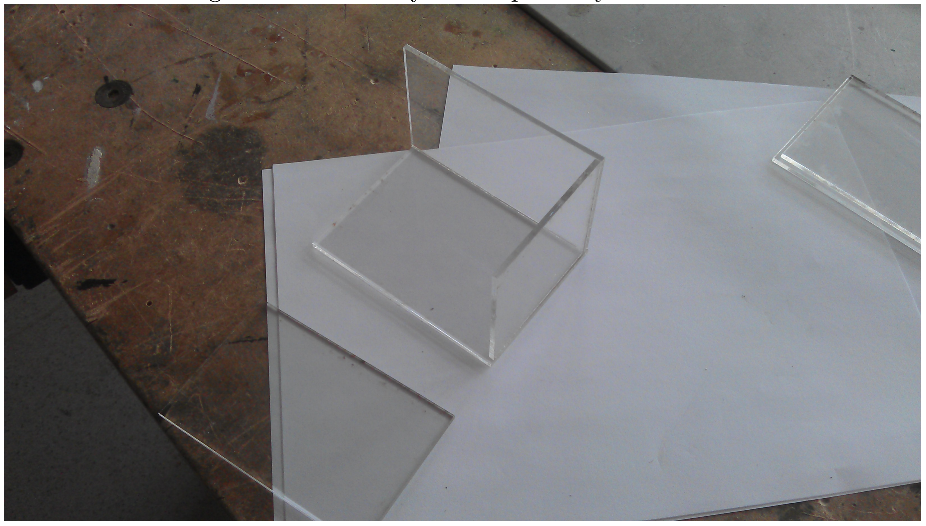
The design was then printed onto a laser printer. The laser printer cut out the designs to a degree of accuracy that is impossible for any human hand.

The box was then assembled using a type of glue called Acrylic Cement. Special care was taken to align the sides of the box, but due to the thinness

[5]http://www.techsoft.co.uk/products/software/2D_Design_V2.asp

of the acrylic (only 3mm) perfect aligning on the box was not accomplished. However this is only a minor detail and it is easily overlooked.

Figure 3.1: The acrylic box partially assembled.



I then had to drill the wormdrive gear boxes to the chassis. Unfortunately during the drilling a minor crack formed around the edges of one of the holes. I did not fully tighten the nut and bolt around this hole in a somewhat hopeful attempt to prevent the crack from spreading any further.

After drilling the holes into the box I attached the wormdrive gear motors and the accompanying wheels, axis and tracks. I noticed that the tracks were a little tight, possibly a symptom of making the box a little bit to long. But at this point there was little to be done about that.

I then places the components inside the chassis, using bubble wrap for scratch protection. I was a little concerned about the use of bubble wrap, as it may cause the components inside the box to overheat. I may have to change the method of protection at a later date if this becomes a problem.

Figure 3.2: Holes in the box being drilled.

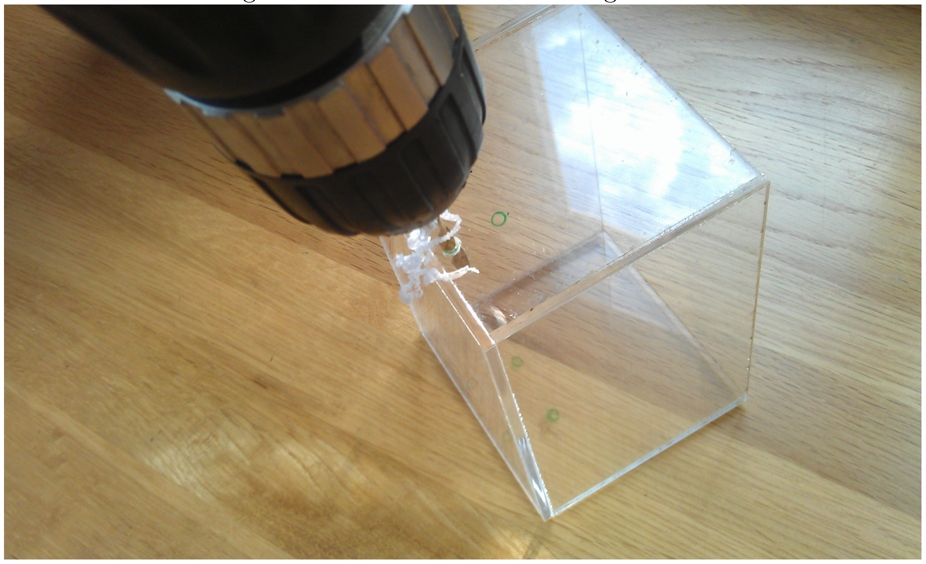


Figure 3.3: Robot with assembled chassis and tracks.

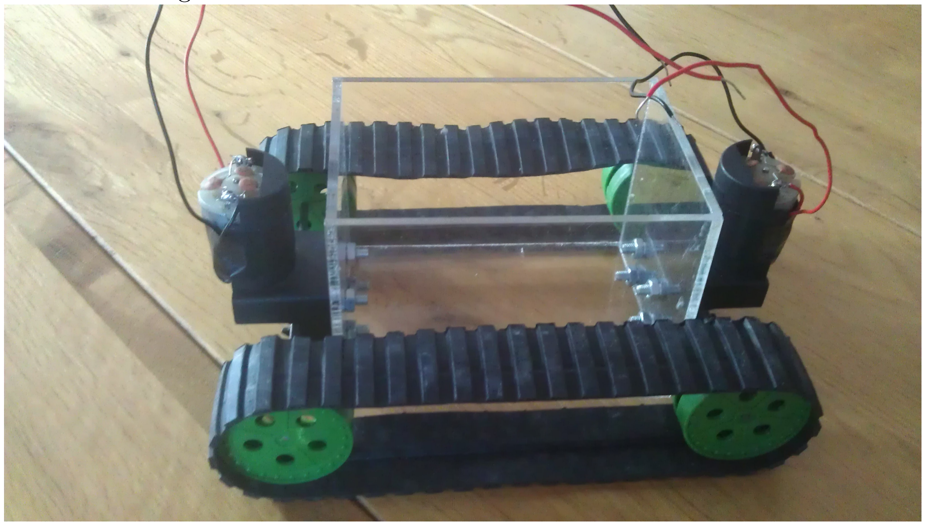
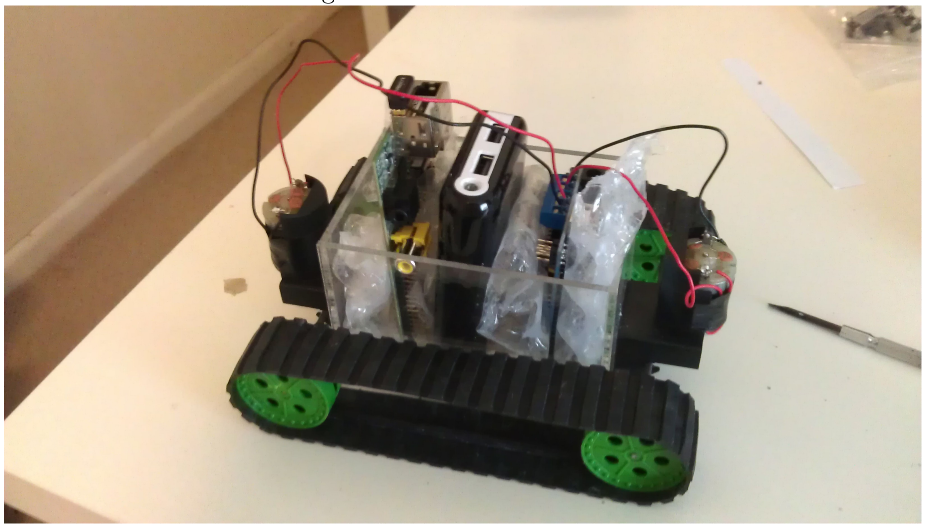


Figure 3.4: Assembled robot



Bibliography

[1] Raspberry Pi Config on Github (May 2013). Retrieved from `https://github.com/asb/raspi-config/blob/master/raspi-config`.

[2] The Oxford Compact English Dictionary (1996). Definition of *robot*.

[3] Elinux.org, May 2012. Raspberry Pi Buying Guide.

[4] Chris Liechti (2010). pySerial's documentation. Retrieved from `http://www.wikibooks.org`.

[5] Yang Zhang (Dec 18, 2009). Web Sockets tutorial with simple Python server. Retrieved from `http://yz.mit.edu/wp/web-sockets-tutorial-with-simple-python-server/`

[6] Mushfiq Sarker (June 5th, 2012). Reading Serial Input with Commands and Numbers. Retrieved from `http://www.inventige.com/arduino-reading-serial-input-with-commands-and-numbers/`.