## 3.2 Lego NXT Mindstorms 2.0

### 3.2.1 Robotic Requirements

The robots required for Robowars were required to have 4 characteristics which were identified as crucial to the implementation of the project. These characteristics are: zero radius turning, to enable the playing of the “light-cycles” game mode. The second characteristic required independent wheel/motor control to avoid having to build a complex steering system. The third requirement dealt with a wireless communication ability which provided the means to have controls being provided from either a server or mobile device, as well as allowing the robot to communicate back to the server. Finally the last requirement involved robot side position tracking. Robot side tracking was deemed the most crucial of the requirements as should a packet be dropped coming from the robot to the server the model can still remain consistent. Had the server done the position tracking, and a packet had been dropped there could have been a model inconsistency which would render all usage pointless.

### 3.2.2 LEGO NXT Mindstorm 2.0

The NXT 2.0 mindstorm kits are the 3rd generation of LEGO mindstorms. The kit consists of three main types of devices: servo motors, input peripherals, and the NXT brick. These devices, when working together alongside the LEGO bricks offers an almost limitless number of design possibilities. The NXT brick is a 48MHz microprocessor with 64KB of SRAM. It allows up to 3 servo motors and 4 input peripherals. The brick can be connected through a wired USB 2.0 connection or wirelessly over a Bluetooth 2.1. It also has an LCD screen and a speaker capable of 8Hz playback. The servo motors are motors controlled using on board tachometers. They provide up to 180RPM with 15N•cm torque[m1]. The peripherals included in the kit are: two touch sensors, a colour sensor, and an ultrasonic sensor. The sensor used in our design is the LEGO colour sensor. This sensor consists of three devices: a white flood light, an infrared colour reader and a RGB LED bank for colour output and reading. This sensor can read values in full 8bit RGB colour while countering for light saturation, and brightness to return the correct 0-255 RGB value as well, LEGO provides 6 preprogrammed colours: white, black, yellow, green, blue and red [m2].

### 3.2.3 RoboWars Robot Implementation

RoboWars has built its own custom robots for this project. In our design we have used 2 servo motors to provide both drive and steering, balanced over a central steering column. These two servo motors can provide zero radius turning by having them spin in opposite directions. To ensure the robots are able to function as liberally as possible the design has added a gearing system with a 3.38:1 gear ratio. The robots communicate to the server over a custom built messaging protocol over a Bluetooth connection. The robots also have a front mounted colour sensor used to provide readings from the map to provide error corrections.

[m1] <http://philohome.com/nxtmotor/nxtmotor.htm>

[m2] <http://lejos.sourceforge.net/nxt/nxj/api/lejos/nxt/ColorLightSensor.html>

# 6 – Robot Client Implementation

## 6.1 Client Implementation

The robot client is a rather simple structure. It consists of many classaes running independently. They are as follows:

* RobotMain: The main class which is used to initialize the RobotCommandController and the listener for key Interupts.
* RobotCommandController: Robot Command Controller acts as the main thread. It has the main representations of RobotMovement and ColorSensor, further more it contains the thread which will periodically return the position of the robot. RobotCommandController’s main purpose is to receive incoming objects and call the corresponding functions in RobotMovement or ColorSensor see figure 6.1.
* ColorSensor: The class used to generate and posess the RobotMap used to correct the errors through movement. In this class is where the error correction color reads are done and handled
* RobotMovement: This is the thread safe access point for the RobotCommandController to access the navigator classes , it simply forwards the commands onto the navigator through calling the correct methods in a safe manner.
* PositionTracker: This class simply returns the robot’s pose on a periodic interval through the LejosOutputStream and then returns to sleep.

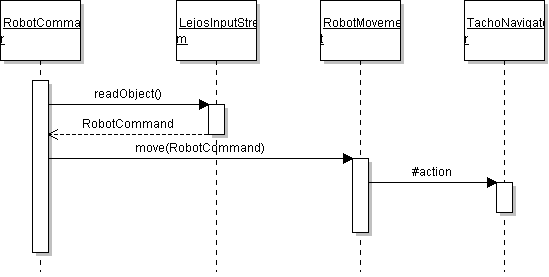


Figure 6.1 How Commands are received

## 6.2 Lejos and Modifications

### 6.2.1 Lejos

Lejos is an open source firmware to provide a JVM which can be embedded on microcontrollers, thus allowing the microcontrollers to be programmed in Java. Originally started in 1999 providing the firmware for devices which were sent to the international space station in 2001 Lejos has continued to grow. Since 2006 Lejos has be focused to the LEGO RCX and LEGO NXT bricks as its primary target microcontrollers. Lejos features a large API as well as a vibrant community which is still active on a daily basis. Furthermore it features a large number of tutorials and reference materials to allow easy development. Although this seems ideal there were many occasions where there were inconsistencies which required major modification, which will be expanded upon in the following sections

### 6.2.2 Bluetooth communication and I/O Modifications

Lejos provides by default an effective handshake system as well as a means with which to provide input and output through subclasses of the Java input and output streams. Lejos also boasts the ability to provide Object passing over these input and output streams. Unfortunately these boasts are unfounded; although a potential system was started it was never fully implemented. Thus a new system was required to be implemented to allow object passing. From this new implementation LejosInputStream and LejosOutputStream were born as well a new message protocol scheme was implemented.

#### 6.2.2.1 LejosInputStream

LejosInputStream is used to receive and decode packets sent over the server to the robot. It receives a byte stream over Bluetooth (or any other inputstream), which it records into a byte array. This array is then converted into a plain text string. The plain text string is then decoded and new objects are created and returned. This input stream though is unable to decode all objects only those which are used in RoboWars (RobotCommand, RobotMap, Vector <Color>, and Pose).

#### 6.2.2.2 LejosOutputStream

LejosOutputStream is used to decode and transfer objects to and from the robot. All objects which can be sent and decoded successfully over the two streams have a function which returns a designated output string in the correct message format. This string is then converted into byte representations of the text. From there the newly formed Byte[] is transferred a byte at a time using the standard OutputStream.write(Byte b) function in java.

#### 6.2.2.3 RoboWars Message Protocol

In RoboWars we have implemented a message protocol to allow the transferring of particular objects over LejosInput/OutputStream. At the moment 4 objects can be sent over the protocol: RobotCommand, RobotMap, Vector<Int>, and Pose. They turned into transferable objects by calling the corresponding toOutputStream function. The message protocol for each object is as follows:

* RobotCommand: [<type>|<speed>|<turnbearing>|<special flags>|<pose>]
* RobotMap: [[MapPoint0],[<x>|<y>|<color>],….,[MapPointN]]
* Vector<int>:[<r-value>|<g-value>|<b-value>|<saturation>]
* Pose: [<x-cooridinate>|<y-coordinate>|<heading>]

## 6.2.3 Navigation and Piloting Modification

Lejos by default provides a system to track position and heading of the robot in multiple ways. The main two tools used are the HiTechnic sensor compass paired the lejos CompassNavigator class or using the onboard tachometers in each servo motor coupled with the lejos Tachonavigator class. Due to the hardware available to the project we were required to use the tachometers embedded in the servo motors to our positioning. These navigators were used to provide high level position tracking as well as act as the access point to the Pilot class which actually operates the motors. As the project progressed there became a series of issues and challenges that needed to be overcome and corrected. They are identified and explained in this section.

#### 6.2.3.1 Blocking On a Steer() Call

In order to implement the rolling turns that are provided to the robot from the server as a representation of the movements done by the telephone we were required to use the steer function provided by lejos. The steer function was designed to have the servo motors rotate with varying degrees of power dependant on a provided ratio. This ratio was from [-200,200] with 0 being drive directly forward, and ±200 being a zero radius turn in either direction. There was a crucial error though in the TachoPilot implementation of steer. The Pilot and all threads would block whenever a non-zero value was provided to steer. This was caused by the default TachoPilot was doing comparisons between the provided float values and hard coded integer values on its branching decisions. This issue resulted in the code driving itself into an infinite loop as it tried to calculate the actual turn ratios this issue was overcome by modifying the hard coded values to reflect the true types of inputs provided. Thus the RoboWarsTachoPilot was born. By correcting this issue the robot was now free to operate in an 180o field as required for the LightCycles game mode.

#### 6.2.3.2 360o Controls and Tracking

One of the two game modes that are implemented for the project it TankSim. TankSim is a simulation as if the robots acted as tanks thus the robots needed to be able to go forwards backwards and any way in between. Unfortunately the code provided in the PilotClass used to drive the motors was only capable to drive the robot in a 1800 range thus not allowing the ability to travel backwards. To overcome this issue a new steer function was implemented which was able to account for the throttle provided by the user. Should the throttle be a value less than zero the parity bits for both servo motors were flipped causing them to rotate in the opposite direction. This allowed for 360o motion and actions to be performed by the robot. There was an unintended consequence of this change though. The position tracking system was not designed to operate in a 3600 environment, it considered everything forwards. Thus resulting in a complete failure of the tracking, caused by two main components:

* The robot no longer knew what way was up: As it had no frame of reference for which way was forwards (usually provided by the parity bit) the robot would always assume that it was travelling forwards. To overcome this UpdatePosition function in SimpleNavigator had to be rewritten to account for the parity of the motors in reference to their original state.
* On a context switch from forwards to backwards the position information would become non-sensical: This issue was caused by other functions that would call UpdatePosition before the change in parity could be taken into account. Thus causing the robots to believe it had turned very far angles (>4000) even though the robot would only track from ±1800, furthermore the robot would lose track of where it was causing the (X,Y) co-ordinate to be meaningless. To overcome this whenever a context switch occurs the pilot is reset and then told where it is. Thus it believes that it started at that point and has travelled no distance at all.

## 6.3 Position Tracking And Error Correction

### 6.3.1 Position Tracking

RoboWars robots use dead reckoning to figure out its location. Dead reckoning uses a starting location and a displacement for the robots to be able to figure out where they are. This is performed by the Navigator classes provided by Lejos, they are used to update the Pose representation of the robot (x co-ordinate, y co-ordinate, and heading). The default system also does not take into account for the geared wheels so the displacement functions in SimpleNavigator had to be rewritten to overcome this difference. The displacements are based off of degree turns of the internal tachometers for each servo motor, and are then converted into units provided by the user, based off the provided dimensions of the robot’s wheel diameter. In our project the units are all based in cm.

### 6.3.2 Error correction and The Grid

As the robots move about the world errors are introduced into the position tracking. Although we have tried to make this as small as possible it is impossible to remove all errors. At the moment there is a drift of ±2mm in either direction as well as a shift of ±2o on the heading per 30cm travelled. Although this is small it is still an issue that needs to be overcome. Despite our grandest efforts there is no way to conclusively correct the heading error if our tracking system is based off of tachometer rotation due to wheel slippage.

#### 6.3.2.1 The Grid

To correct this, a complex system was created to track the robots with in a world. This system is known as the grid. The Grid is a grid of dots with 1.5cm diameter in varying colors placed on the game mat. This is a real world representation of an onboard RobotMap that the robot generates based upon inputs provided by the server or as the default setting. When a robot using its colour sensor drives over a dot and detects its colour the robot will determine where the nearest dot of that colour in the correct direction is and adjust its position accordingly. This “Snapping” action snaps to the location of the center of the robot not the sensor, so this generated value is generated by the Robot using basic trigonometry.

#### 6.3.2.2 Future Suggestions

Due to technology available at the time, we have not been able to create a perfect tracking system. Yet alternatives to our design have been devised should there be a chance to improve upon it. These suggestions are as follows:

* Heading Correction: At the time we started there was not the budget to get the HiTechnic Compass sensor. In any future designs the robots should be equipped with such a sensor the heading error becomes 0. This will help in both the position tracking but also the snapping action done by the colour sensors as they use the heading during trigonometry.
* Digital Correction: When we began certain technologies were not released yet. The main important one would be the Microsoft Kinect. The Kinect projects an RGB grid. If we were able to project a similar grid on the game mat we would be able to get perfect position tracking using similar methods as are currently implemented[mw3].
* Gradient Colour Scheme: Due to the means available to us we are unable to implement this idea in its fullest. By making the grid into a colour gradient we would be able to get the correct position based off the RGB value read by the sensor. Because of the sensitivity we would be able to get an absolute position but will still be unable to correct the heading error.

[mw3] <http://www.techradar.com/reviews/gaming/games-consoles/controllers/microsoft-kinect-905010/review>

## 8.2 Robot Client Testing

As continuing with the projects modularity the client can be identified into two sections, the I/O streams that are connecting the robot to the server and the code to run the robots. Both systems have been given the means to be tested the testing is as follows:

* IOTest.java: This script runs a simulation to check that the LejosIn/OutStreams are able to successfully encode and decode every type of object that is used in communication for RoboWars matches.
* TestColourSensor.java: This script is used to check if the Colour sensor is still working properly. ColorSensor in RoboWars can be run in test mode where it returns the read and these reads are output on a server screen. Unfortunately this output must be approved by a human oracle.
* KeyController.java: This script is used to check that the robot is moving correctly. It acts as an interface to run all commands used by RoboWars over Bluetooth. The movements must be verified by a human oracle as there is no way to assert that the correct movement occurred programmatically.

``