# RoboCup Soccer Simulator Client Framework Users Guide

by

redCard_Linux




Spring/Summer 2003

# Table of Contents


Introduction ..... 3
System Requirements ..... 4
Installation ..... 4
Getting Started with RoboCup ..... 5
Client Loader ..... 6
Soccer Client Overview ..... 7
Soccer Client Framework ..... 7
    Client Module ..... 7
    ClientDlg Module ..... 7
    Network Module ..... 7
    Parser Module ..... 8
    Player Module ..... 9
Developing Clients ..... 10
Developing a Coordinate System ..... 12
Appendix A: Soccer Client → Soccer Server Messages (Commands) ..... 15
Appendix B: Soccer Server → Soccer Client Messages ..... 17
Appendix C: Player Class Data Members and Methods ..... 19
Appendix D: Network Class Data Members and Methods ..... 23
Appendix E: Parser Class Data Members and Methods ..... 25
Appendix F: Soccer Client Dialog ..... 30
Appendix G: Client Loader ..... 33
Appendix H: RoboCup Files/Folders ..... 35

# Introduction

RoboCup is an international project aimed at fostering research in Artificial Intelligence (AI), robotics, autonomous agents, and related fields. To this end, The RoboCup Federation promotes several different leagues of soccer competition, including the humanoid league and the simulation league. For competition in the humanoid league, very expensive robots are required. For competition in the simulation league, the RoboCup Soccer Simulator is required.

The RoboCup Soccer Simulator is a platform for research into the AI aspects of RoboCup. While not as glamorous as humanoid robots, the Simulator allows researchers to concentrate on developing the intelligence domain without having to be concerned with the mechanics and expense of physical robots.

The RoboCup Soccer Simulator consists of three main components:

- A Soccer Server
- A Soccer Monitor
- A set of Soccer Clients

This RoboCup Soccer Simulator package includes the following components: a Soccer Server (executable file only); a Soccer Monitor (executable file only); and a Soccer Client Framework (complete C++ source code). Practice Teams of Soccer Clients (executable files only) are also included. See Appendix H for a complete listing of included files.

The Soccer Server is the controller of the simulation. It imposes all rules of play and communication protocols. The Soccer Monitor and Soccer Clients establish a UDP socket connection with the Soccer Server. Via these connections, messages are exchanged between the Soccer Monitor and the Soccer Server and between the Soccer Clients and the Soccer Server.

The Soccer Monitor provides a visual interface – it provides the graphical soccer field and the soccer player images on the field.

Soccer Clients are independent, autonomous agents representing the brains of the soccer players. The Soccer Client Framework is the infrastructure for a Soccer Client. It provides all the necessary networking support and message parsing and formatting support. It's left to the students in the AI courses to extend the C++ code of the framework to develop a team of intelligent, competitive soccer players. Developing intelligent behavior in the Soccer Clients is the AI aspect of RoboCup.

## System Requirements

**Computer**: IBM or compatible PC, 600MHz or higher
**Memory**: Minimum 128 MB RAM
**Operating System**: Windows98/2000/XP*
**Monitor**: SVGA color monitor, 800X600 or higher screen resolution
**Additional software**:
Microsoft Visual Studio 6.0 Professional*
Adobe Acrobat Reader (for user manuals)

*As a student, you have access to these Microsoft products at no charge through the Microsoft Developer Network Academic Alliance (MSDNAA).
Sign up at: http://www.cis.usouthal.edu/documents/msdnaasignup.html

## Installation

1. Download RoboCup file (probably a zip file) from location provided by your instructor (or copy RoboCup folder from CD if provided).

2. Unzip or save files to your hard drive. We recommend you save the RoboCup folder to your C:\ drive. It is important that you maintain the RoboCup files in the same folder/directory structure on your computer as provided to you. For example, check **Use Folder Names** if you are using WinZip.

3. If you are copying these files from a CD, you must change the **Read-only** property for all files. Right click the RoboCup folder, click **Properties**, **General** tab, Uncheck **Read-only**, click **Yes** to apply to all subfolders and files.

4. See **Appendix H** for more information on RoboCup files/folders.

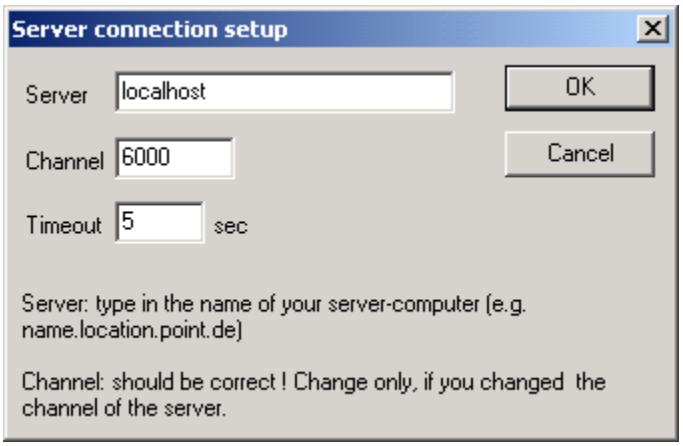
# Getting Started with RoboCup

1. Start the Soccer Server (..\RoboCup\SoccerServer\rcssserver.exe).

2. Start the Soccer Monitor (..\RoboCup\SoccerMonitor\TsinghuAeolus_monitor.exe).

3. Click **Connect**  on the Soccer Monitor. This will allow you to connect the Soccer Monitor to the Soccer Server.

   If you are running the Soccer Server and Soccer Monitor on the same computer, type **localhost** in **Server**.




   If you want to connect the Soccer Monitor to the Soccer Server running on a different computer, type the *IP address of the computer running the Soccer Server* in **Server**.

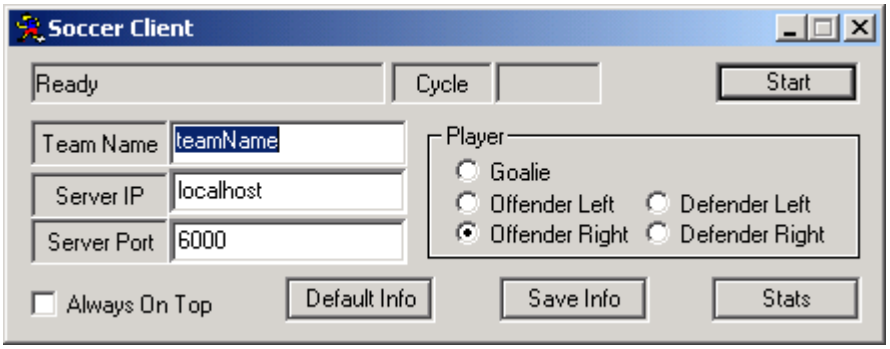
4. Start Microsoft Visual C++ 6.0.

5. Open the Soccer Client workspace (..\RoboCup\SoccerClient\client.dsw).

6. Edit the code for your players as desired (See the **Developing Clients** section for more information).

7. Execute your new Soccer Client.

8. The following Soccer Client Dialog will appear. Now you can start loading your players on the field.




   Type your team's name in **Team Name**.

   Type **localhost** or the *IP address of the computer running the Soccer Server* in **Server IP** (This will be the same as in step 3 above).

   Select the player you wish to load (Goalie, Offender Left, etc).

   **Note**: Each player runs the code from the .cpp file of the same name. For example, put the code for **Goalie** in **Goalie.cpp**, **Offender Left** in **OffenderLeft.cpp**, etc. Place code common to all players in **Player.cpp**. See the **Developing Clients** section for more information.

9. Click **Start**.

10. Repeat steps 7-9 as desired to load additional soccer players.

11. Click **Kick off**  on the Soccer Monitor to start the game. You can utilize the features of the Soccer Client Dialog(s) to debug/improve your player(s) or simply minimize them for an unobstructed view of the playing field. See **Appendix F** for more information on the Soccer Client Dialog.

    **Note**: You must close any running Soccer Clients before recompiling your code.

## Client Loader

Utilize the Client Loader (..\RoboCup\ClientLoader\Client Loader.exe) to load your players/team in game mode. This is used for tournament play or at any time you want to run your players without the overhead of the full Client Dialog debugging window. See **Appendix G** for more information on the Client Loader.

# Soccer Client Overview

From a high level abstract point of view, a RoboCup Soccer Client consists of four components – a Networking Function, a Parsing Function, an AI (Thinking) Function, and a Data Structure that represents the Soccer Client's perspective of the game.

The Networking Function is responsible for handling message traffic between the Soccer Client and the Soccer Server via a UDP socket connection. The Networking Function receives various messages from the Server, which it then passes on to the Parsing Function to extract the relevant information held in the messages. The extracted information is stored in the Data Structure for subsequent use by the AI Function. The AI Function uses the sensory information in the Data Structure to make decisions on how to proceed towards its objectives. The AI Function sends command messages to the Networking Function, which in turn transmits the messages to the Soccer Server.

# Soccer Client Framework

This Soccer Client Framework consists of three major modules – the Network Module, the Parser Module, and the Player Module. Additionally there are two supporting modules – the Client Module, and the ClientDlg Module. Each of these modules exists as a number of .ccp and .h files.

## Client Module

The Client Module simply serves as the entry point of the application. It consists of the files Client.cpp and Client.h.

## ClientDlg Module

The ClientDlg Module provides a user interface which is used to start the player. During Soccer Client development, ClientDlg can display statistical information such as stamina and the ball's distance and direction from the player. During tournament play, this feature can be turned off. The ClientDlg Module consists of the files ClientDlg.cpp and ClientDlg.h.

## Network Module

The Network Module is responsible for communications to and from the Soccer Server. It consists of the files Network.cpp and Network.h. The Network Module establishes and maintains a UDP socket connection with the Soccer Server. The Network Module transmits messages to the Soccer Server. These messages originate as action commands (Dash, Kick, Turn, etc.) from the Player Module. The Network Module also receives messages from the Soccer Server. These text string messages must be parsed before the information within can be made available to the Player Module.

## Parser Module

The Parser Module is responsible for breaking down the messages coming into the Soccer Client from the Soccer Server and extracting the relevant information. Depending on the message type and the information within, the extracted information will either trigger an event in the Player Module or be stored for subsequent use by the Player Module. The Parser Module consists of the files Parser.cpp and Parser.h.

There are five different types of messages that the Parsing Function must handle: (init …), (reconnect …), (hear …), (sense_body …), and (see …). Each message contains a varying number of atomic pieces of information called tokens. The following are examples of each of the five types of messages:

- `(init l 6 before_kick_off)`
- `(reconnect r before_kick_off)`
- `(hear 200 referee play_on)`
- `(sense_body 2766 (view_mode high normal) (stamina 4000 1) (speed 0) (head_angle 0) (kick 24) (dash 200) (turn 188) (say 0) (turn_neck 0) (catch 0) (move 3) (change_view 0))`
- `(see 3193 ((flag l b) 47.5 -32) ((flag g l b) 29.4 -1) ((goal l) 27.1 11 -0.542 0.2) ((flag g l t) 26.6 26 -0.532 0.4) ((flag p l c) 11.5 -4 -0.46 -0.2) ((flag b l 50) 50.4 -38) ((flag l 0) 32.1 13 -0.642 0.2) ((flag l t 10) 32.1 31 -0.642 0.4) ((flag l b 10) 35.2 -2) ((flag l b 20) 40.4 -15) ((flag l b 30) 47.5 -25) ((ball) 10 -10 -0.4 -0.4) ((line l) 29.1 -66))`

Every complete message begins with an opening parenthesis and ends with a closing parenthesis. For each incoming message, the first thing the Parser does is call the function isCompleteMessage(...) to attempt to ensure the message is complete, by checking for every opening parenthesis there is a matching closing parenthesis. Next the parser determines which of the five possible types the message is by examining the fourth character in the string, since this character is unique to each type of message. With the message type determined, a function to parse that particular message type is called: parseInitMessage(...), parseReconnectMessage(...), parseSenseBodyMessage(...), parseHearMessage(...), or parseSeeMessage(...).

The vast majority of the messages represent the eyes and ears of the soccer player. Through these messages the player can ‘see’ objects on the field, such as the ball and other players. The player can also ‘hear’ audio messages from the referee and other players. Once the information has been extracted from a message, it is stored in the appropriate data structure for subsequent use by the Player Module.

## Player Module

The Player Module is the nucleus of the Soccer Client. All other modules exist to support the Player Module. The Player Module consists of the following files: Player.cpp, Goalie.cpp, DefenderLeft.cpp, DefenderRight.cpp, OffenderLeft.cpp, OffenserRight.cpp, and associated .h files. To create a particular player type, such as goalie or left defender, the Player object is inherited by the particular instance of player. Therefore, each player inherits characteristics common to all players and only requires additional code for player specific characteristics.

The Player Module is responsible for the "intelligence" of the player. This where AI students working to develop intelligent Soccer Clients will be working 99.99% of the time. Most of the code in the Player Module consists of skeletal functions. However, there is also some infrastructure code to handle sending and receiving messages to and from the Soccer Server via the Network Module.

The Player Module receives notifications of new data available, such as new sensory information and new audio messages. It must then analyze the information and make intelligent decisions on how to act. Once the information is analyzed and decisions made, the Player module will perform actions by sending one or more command messages to the Soccer Server via the Network Module.

# Developing Clients

Typically a team will consist of a goalie, two defenders and two offenders, however it's left to the students' discretion how to populate its five-player team (exception: no more than one goalie is allowed – this rule is imposed by the Soccer Server.)

All of the source code for the Soccer Client Framework is provided, however you should only have to work within the Player Module code to develop your Soccer Clients. The Player Module consists of the files: Player.cpp, Goalie.cpp, DefenderLeft.cpp, DefenderRight.cpp, OffenderLeft.cpp, OffenserRight.cpp, and associated .h files. The classes Goalie, DefenderLeft, etc. inherit from the Player class. Each player inherits characteristics common to all players and only requires additional code for player specific characteristics. Therefore, if you write code for all players, add it to the Player class. However, code you write for a specific player such as OffenderRight, goes only in that class.

To make your player behave as an intelligent competitive soccer player, use the information in the GameData.h file to "see" the situation on the field and determine what actions to perform. This information is updated every 150ms when the Soccer Client receives a '(see …)' message from the Soccer Server. To perform the actions, invoke the Dash, Kick, Turn, etc. functions. The following code sample causes the player to simple-mindedly find the ball and kick it towards the goal. This simple code illustrates how to use the information in GameData.h and issue action commands.

```
int i = gameData.game.cMySide == 'l' ? 1 : 0;

bool   ballVisible   = gameData.ball.bVisible;      //
double ballDirection = gameData.ball.dDirection;    //
double ballDistance  = gameData.ball.dDistance;     // These variables are not
bool   goalVisible   = gameData.goal[i].bVisible;   // necessary, but they make
double goalDirection = gameData.goal[i].dDirection; // the code easier to read.
double goalDistance  = gameData.goal[i].dDistance;  //
int myStamina        = gameData.senseBody.nStamina; //

if(ballVisible)
    if(ballDirection > -15 && ballDirection < 15 )
        if(ballDistance < 1)
            if(goalVisible)
                if(goalDistance < 25)
                    Kick(100, goalDirection);   // Shoot the goal
                else
                    Kick(30, goalDirection);
            else
                Kick(30,90);
        else
            if(myStamina > 2000)
                Dash(100);
            else
                Dash(70);
    else
        Turn(ballDirection);
else
    Turn(90);
```

To act on the ball the player must be able to see it. gameData.ball.bVisible will be true if the ball is in the player's field of view. The player's field of view defaults to 90°. However, you can change this angle to 45° or 180° by issuing a ChangeView(*Quality, Width*) command. Quality is either 'high' or 'low' and Width is either 'narrow' (45°), 'normal' (90°), or 'wide' (180°). The Quality parameter default is 'high'. Changing it to 'low' doubles the frequency of '(see …)' messages but decreases the distance and detail the player can see. The Width parameter also affects the distance and detail the player can see. The wider the angle the shorter the distance and the lower the detail. You will have to experiment with these parameters to determine what works best for the particular situation. Of course you can always just leave them at the defaults.

In general, with default view parameters, the ball can be seen anywhere on the field if it's in the player's field of view. If gameData.ball.bVisible is false, the Turn(*Angle*) or TurnNeck(*Angle*) command should be issued to locate the ball. The Turn(*Angle*) command turns the player the specified number of degrees and thus changes its heading. The TurnNeck(*Angle*) command turns the player's head relative to its body.

**(Note: unless an object is visible, its direction, distance, and other parameters are meaningless, and therefore should not be used).**

Once the ball is visible, the player knows its direction and distance. If the player is not heading in the ball's general direction, the player can turn towards the ball's direction by issuing the Turn(ballDirection) command. If the ball's distance is within about one unit, the player can kick the ball by issuing the Kick(*Power, Angle*) command. If the ball is further away, the player can run to the ball by issuing a Dash(*Power*) command.

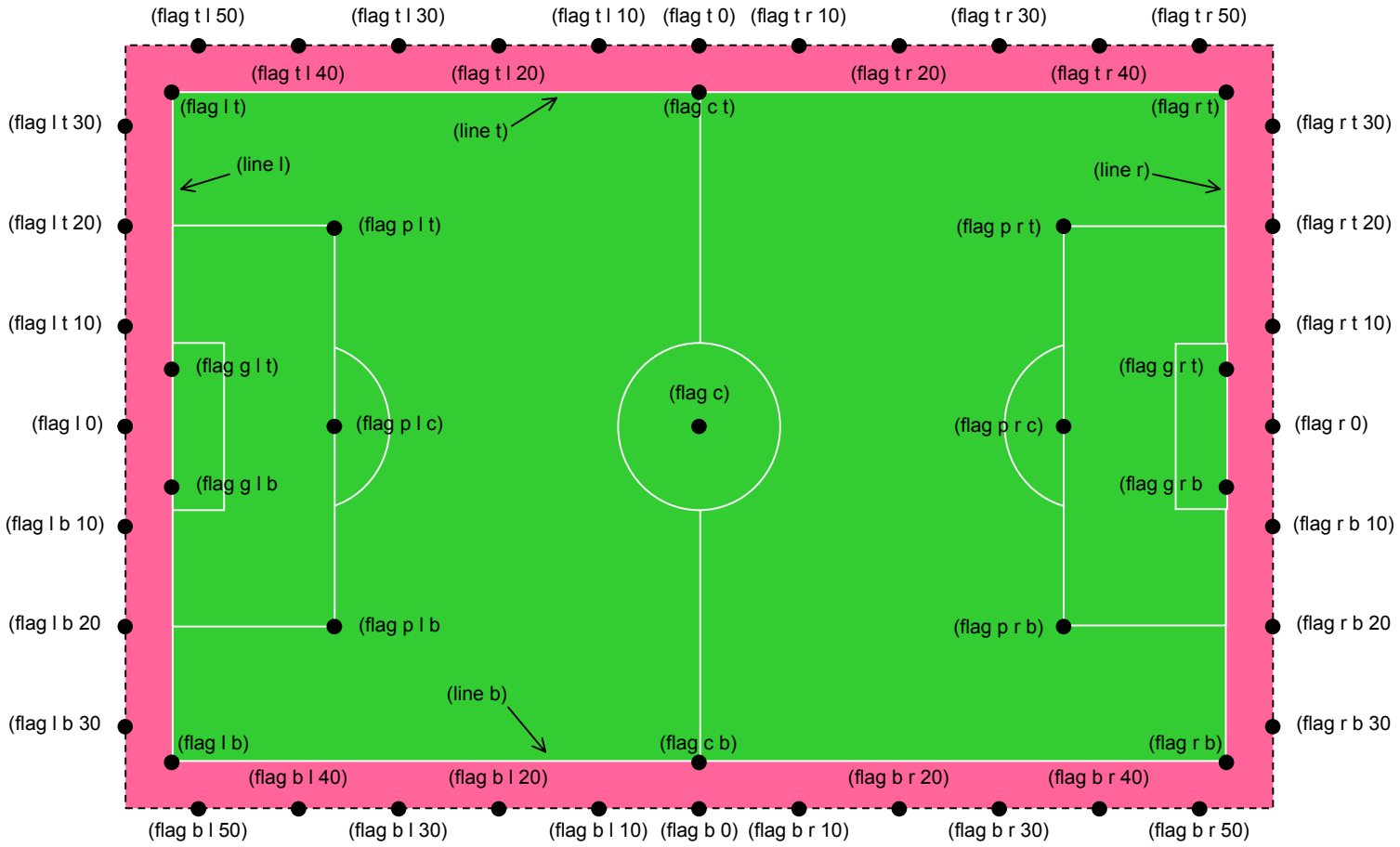
The Dash(*Power*) command moves the player in the direction it is heading (the player moves backwards if *Power* is negative). The Dash(*Power*) command consumes stamina; the higher the *Power*, the more consumed. Stamina is continuously being restored, but if it's being consumed to rapidly it will be depleted. Stamina is the player's energy**. If the player runs out of energy it can't run.** Note that the Dash(*Power*) command only moves the player one unit each time it is issued, so it must be issued continuously (once per cycle) to make the player run.

The Dash, Kick, Turn, and TurnNeck commands should only be issued once per cycle (100ms). The Soccer Server ignores all but one of these commands if more than one are issued in a cycle, and the one that is actually accepted is random. The '(see …)' message is received every 150ms and updates the visual information in gameData. The '(sense_body)' message is received every 100ms. These are your timing signals.

Now it's up to you to use these tips to implement your strategies. Good Luck!

# Developing a Coordinate System

There are 53 flags on the soccer field and 4 lines bordering the field. The player can use these flags and lines to calculate its heading and its X and Y coordinates.




To take full advantage of the flags and lines, dust off your Trig book and use them to develop your coordinate system. The coordinate system would give the player X and Y coordinates. With X and Y coordinates, you can create functions such as gotoXY() to send the player to any location on the field. Once the player knows its coordinates, it can calculate the coordinates of other objects on the field, such as the ball and other players.

While it's not absolutely necessary to have a coordinate system, teams that do have a distinct advantage.

The first step in creating a coordinate system is to determine the orientation of the system and whether it's an absolute or relative system. Possibilities include an absolute system with (0,0) either at center field or at one of the corners of the field. Another possibility is a relative system with (0,0) at the center of the field and the coordinates relative to the player. There are advantages and disadvantages to both systems. You will have to decide which is best for your team.

The next step in creating a coordinates system is to calculate the player's heading using the lines around the field. The player can always see exactly one line and knows the angle of incidence to that line and the distance from the line at the point of incidence. You may find the following code snippet useful.

```
char mySide = gameData.game.cMySide;
char  lineSide = gameData.line.cSide;
double lineAngle = gameData.line.dAngle;
double headAngle = gameData.senseBody.dHeadAngle;
double X;
double Y;
double heading;


// calculate heading based on lines
// heading equals zero when player is heading straight down field from its side
// of the field to the opponents' side of the field

if (lineSide == 'r')
{
   if (lineAngle < 0)  heading = -lineAngle - headAngle - 90;
   else                heading = -lineAngle - headAngle + 90;
}
else
if (lineSide == 'b')
{ . . . . . .
```

Once you know the heading, you can calculate the player's X and Y coordinates by triangulating on the flags. You can use all the flags on the field or just the flags around the field. The player can almost always see at least one of the flags around the field. You only need to calculate on any one flag the player can see at that moment. It may be advantageous (perhaps more accurate) to use the closest flag. You may want to consider creating a table (array) containing all of the flags' coordinates. The flag coordinates in this table would depend on the orientation of your coordinate system. Here are a few coordinates based on standard Cartesian coordinates with the center of the field being (0,0):

```
flag c – (0,0)
flag t 0 – (0,39)
flag b 0 – (0,-39)
flag l 0 – (-57.5,0)
flag r 0 – (57.5,0)
flag l t – (-52.5,34)
flag r b – (52.5,-34)
flag t r 30 – (30,57.5)
flag l b 20 – (-57.5,-20)
```

You may find the following code snippet useful. Note that this a small fragment of code that was used on a relative coordinate system with the center of the field at (0,0). This system only used some of the flags around the field and did not use a table for flag coordinates (although in hind-site it should have).

```
const double PI = 3.14159;
const double RAD = PI / 180;   // Radians

if(gameData.flag[47].bVisible)   // Flag_R_0: Right Center [47]
{
   Y = -gameData.flag[47].dDistance * sin((gameData.flag[47].dDirection +
       headAngle + heading) * RAD);

   X =  gameData.flag[47].dDistance * cos((gameData.flag[47].dDirection +
        headAngle + heading) * RAD);

   if (mySide == 'l')  X = -(X - 57.5);
   else                X = -(X + 57.5);
}
else
if(gameData.flag[29].bVisible)   // Flag_B_0: Bottom Center [29]
{
   Y = gameData.flag[29].dDistance * cos((gameData.flag[29].dDirection +
       headAngle + heading - 90) * RAD);
          X = gameData.flag[29].dDistance * sin((gameData.flag[29].dDirection +
       headAngle + heading - 90) * RAD);

   if (mySide == 'l')  Y = -(Y - 39);
   else                Y = -(Y + 39);
}
else
   . . . . . .
```

Creating a coordinate system is one of the more challenging aspects of developing Soccer Clients. Have fun!!!

# APPENDIX A

## Soccer Client → Soccer Server Messages (Commands)

**Initialization**

- **`(init TeamName [(version VersionNumber) [(goalie)])`**
  VersionNumber = the protocol version

- **`(reconnect TeamName UniformNumber)`**

To get the player to perform actions, you will call the following command functions. (Note, the function calls will be of the format "`Command(`*`Parameter`*`, …)`". The individual functions will format the command messages sent to the Soccer Server as "(command *Parameter* …)". Example: when you issue the function call "Kick(100, 0)" the Soccer Server will be sent the message "(kick 100 0)".)

- **`Dash(`*`Power`*`)`**
  Use the Dash command to move the player. The player always moves in the direction its body is facing. A single dash command causes the player to move only a very short distance (one unit). For the player to actually run, dash commands must be issued continuously. *Power* is a value from -100 to 100. This value affects the speed of the player. As the player dashes, the player's stamina is consumed. (When the player is at rest, stamina is restored.) The higher the power value, the more stamina consumed. However, for values up to about 75, the player can run around all day without significant loss of stamina. At full power (100), the player will quickly run down its stamina. Note also that backwards dashes (-*Power*) consumes double stamina.

- **`Turn(`*`Direction`*`)`**
  Use the Turn command to turn the player. *Direction* is an angular value from -180° to 180°. Negative values cause the player to turn to the left and positive values cause the player to turn to the right. The turn command causes the player to turn its entire body, including its head, to the specified direction.

- **`TurnNeck(Direction)`**
  Use the TurnNeck command to turn the player's head relative to its body, thus allowing the player to see in a direction different from the direction it is heading. *Direction* is an angular value from -90° to 90°. Negative values turn the player's head to the left and positive values turn the player's head to the right.

- **`Kick(Power, Direction)`**

  Use the Kick command to have the player kick the ball. *Power* is a value from -100 to 100, and *Direction* is an angular value from -180° – 180°. The ball can be kicked forward (*Power* > 0) or backward (*Power* < 0). If *Direction* = 0, the ball is kicked in the direction the player is currently heading. If *Direction* is negative, the ball is kicked left of the player's heading, and if *Direction* is positive, the ball is kicked right of the player's heading. The ball can be kicked towards an entity, such as a goal, by setting *Direction* equal to the entity's current direction value.
  Example: Kick(100, gameData.goal[i].dDirection).

- **`Catch(Direction)`**

  Use the Catch command to have the goalie catch the ball. This command is permitted only for goalies while they are in their penalty box – it is ignored by the Soccer Server if issued by any non-goalie player. *Direction* is an angular value from -180° – 180°.

- **`Move(X,Y)`**

  Use the Move command to instantly move the player to the position (X, Y). This command is available only in the before_kick_off mode, except the goalie can also invoke this command immediately after catching the ball. *X* is a value from –52.5 to 52.5, and *Y* is a value from –34 to 34. The coordinate system for this command is as follows: the center of the field is (0,0); with respect to the player standing on his side of the field, facing downfield, the player's upper left quadrant is (X , -Y), its upper right quadrant is (X, Y), its lower right quadrant is (-X, Y), and its lower left quadrant is (-X, -Y). This coordinate system is relative to the player, regardless of which side of the field it is on.

- **`ChangeView(Width Quality)`**

  Use the ChangeView command to change the player's field of view. Width = narrow, normal, or wide; Quality = high or low.

- **`Say(Message)`**

  Use the Say command to broadcast audio messages to other players
  **Warning: Do not use parentheses in Message**

- **`SenseBody)`**

  Use the SenseBody command to request a sense_body message from the Soccer Server (you probably do not ever need to use this command since the Soccer Server automatically sends a sense_body message every 100ms anyway).
  The server returns: `(sense_body Time (view_mode {high | low} {narrow | normal | wide}) (stamina Stamina Effort) (speed AmountOfSpeed) (head_angle HeadAngle) (kick KickCount) (dash DashCount) (turn TurnCount) (say SayCount)(turn_neck TurnNeckCount))`

# APPENDIX B

## Soccer Server → Soccer Client Messages

### Initialization

- **`(init Side UniformNumber PlayMode)`**
  Side = l | r
  UniformNumber = 1 – 11
  PlayMode = a valid play mode

- **`(reconnect Side PlayMode)`**

### Audio Sensor

- **`(hear Time Sender Message)`**
  Time = current game cycle
  Sender = self | referee | *direction*
  (*direction*: when the sender is another player, the relative direction of that player is given instead of who it is)
  Message = alphanumeric text string

  Referee messages:
  - `before_kickoff`
  - `kickoff`
  - `kick_in`
  - `play_on`
  - `goal_scored`
  - `goal_kick`
  - `game_over`
  - `game_in_overtime`
  - `half_time`
  - `offsides`
  - `corner_kick`
  - `catch_ball`
  - `free_kick`
  - `foul`

## Visual Sensor

- **(see Time ObjName Distance Direction**
  **[DistChanged DirChanged**
  **[BodyFacingDir HeadFacingDir]]))**

  or

  **(see Time (ObjName Direction))**

  **ObjName = (player [TeamName [UniformNumber [goalie]]])**
  or **(ball)**
  or **(line l|r|t|b)**
  or **(goal l|r)**
  or **(flag c [t|b])**
  or **(flag p l|r c|t|b)**
  or **(flag g l|r t|b)**
  or **(flag l|r 0|t|b [10|20|30])**
  or **(flag t|b 0|l|r [10|20|30|40|50])**
  or **(Player)**
  or **(Ball)**
  or **(Goal)**
  or **(Flag)**

  Note that the amount of information given for each object depends on how well the player "sees" the object. Generally the more distant the object the less information given.

## Body Sensor

- **(sense_body Time (view_mode {high | low} {narrow | normal | wide})(stamina Stamina Effort)(speed Speed)(head_angle Angle)(kick Count)(dash Count)(turn Count)(say Count)(turn_neck Count))**

  The last five parameters are counters of commands received by the Soccer Server. They can be used to keep track of lost or delayed messages.

# APPENDIX C

## Player Class Data Members and Methods

### Public Data Members:

**bool m_bConnected**
Set to true when the Soccer Client receives the '(init …)' message from the Soccer Server – indicates the Soccer Client is connected to the Soccer Server

**CString m_sLastRefMsg**
Holds the most recent referee message for display in the Debug Tool

**CString m_sLastMsgSent**
Holds the last message sent to the Soccer Server from the Soccer Client for display in the Debug Tool

**CGameData gameData**
Instance of the GameData class

**CNetwork m_network**
Instance of the Network class

**Parser parser**
Instance of the Parser class

### Public Methods:

**CPlayer()**
Default constuctor

**virtual ~CPlayer()**
Default destructor

**BOOL Connect(CString serverIp, int serverPort, CString teamName, bool bGoalie)**
Calls the Network's Start function to connect the Soccer Client to the Soccer Server. Sends an '(init …)' command to the Soccer Server if a successful connection is established. Returns true if successful, false otherwise.

**BOOL Reconnect(CString serverIp, int serverPort, CString teamName)**

Similar to the Connect function. Calls the Network's Start function to reconnect the Soccer Client to the Soccer Server. Sends a '(reconnect …)' command to the Soccer Server if a successful connection is established. Should only be called following a disconnect after a successful connection. Returns true if successful, false otherwise.

**void Disconnect()**

Disconnects the Soccer Client from the Soccer Server. Sends a '(bye)' command to inform the Soccer Server that the Soccer Client is leaving, then calls the Network's Stop function to close the socket.

**virtual void InitPlayer()**

Called to handle an '(init …)' message and should be overridden by every inherited player.

**virtual void Action()**

Called whenever a '(see …)' message is received (every 150ms). This method should be overridden by every inherited player.

**void OnReceivedMessage(CString msg)**

Called by the Network Module in respond to receiving a new message from the Soccer Server. Passes the new message on to the Parser Module. Is returned a char indicating the type of message received. Then invokes the corresponding notification function.

Call the following methods to make the player perform a specified action. Each of these methods formats the command into a message suitable for transmission to the Soccer Server. You should NOT alter these methods. See Appendix B for more on these commands.

**void Kick(double power, double direction)**

Call this method to kick the ball

**void Turn(double direction)**

Call this method to turn player

**void TurnNeck(double angle)**

Call this method to turn player's head relative to body

**void Dash(double power)**

Call this method to move player forward or backwards during play

**void Move(double x, double y)**

Call this method to move player to (x,y) position instantly during before_kick_off mode

**void SenseBody()**

Call this method to request a sense_body message from the Soccer Server (you probably do not ever need to use this command since the Soccer Server automatically sends a sense_body message every 100ms anyway)

**void Catch(double direction)**
Call this method to make the **goalie** catch the ball

**void Say(CString Message)**
Call this method to broadcast an audio message to other players

**void ChangeView(CString Width, CString Quality)**
Call this method to change the player's field of view

The following methods are called by the refereeMsg(…) method in response to receiving a referee audio message from the Soccer Server. Override these methods for individual players and add your own code to respond to the specific event. Hint: although you are not required to include code to respond to these events, you will probably want to respond to the first four events to properly position your players. (After each goal is scored, there is a short "dead-time" to allow the players to use the Move(x, y) command to instantly reposition themselves prior to the ensuing kickoff. You should Move your players and make sure they stay where you moved them, because the Soccer Server will randomly position any player that is not on its side of the field, usually nowhere near where you want them to be.)

**virtual void eventBeforeKickoff()** – received a before_kick_off referee message
**virtual void eventKickoff(char cSide)** – received a kick_off referee message
**virtual void eventGoalScored(char cSide)** – received a goal_scored referee message
**virtual void eventPlayOn()** – received a play_on referee message
**virtual void eventHalfTime()** – received a half_time referee message
**virtual void eventOffsides(char cSide)** – received an offsides referee message
**virtual void eventCornerKick(char cSide)** – received a corner_kick referee message
**virtual void eventGoalKick(char cSide)** – received a goal_kick referee message
**virtual void eventFreeKick(char cSide)** – received a free_kick referee message
**virtual void eventFoul(char cSide)** – received a foul referee message
**virtual void eventKickIn(char cSide)** – received a kick_in referee message
**virtual void eventGameOver()** – received a game_up referee message
**virtual void eventGameInOvertime()** – received a game_overtime referee message

## Private Methods:

**void OnInitPlayer()**
Notification that a '(init …)' message has been received. Invokes the InitPlayer() function for the inherited player to override.

**void OnReconnect()**
Notification that a '(reconnect …)' message has been received.

**void OnSenseBody()**
Notification that a '(sense_body …)' message has been received.

**void OnVisualData()**
Notification that a '(see…)' message has been received.

**void OnAudioMessage()**
Notification that a '(hear…)' message has been received.

**void refereeMsg(string msg)**
Parses referee audio messages. Invokes the appropriate event method.

# APPENDIX D

## Network Class Data Members and Methods

### Private Data Members:

**CPlayer* m_pPlayer**
A pointer to the instance of the player. Used to transfer new messages from the Soccer Server to the player.

**Int m_port**
Contains the port that the server is running on. Default value = 6000.

**CString m_ip**
Contains the ip address that the server is running on.

**SOCKADDR_IN sockAddr**
Contains the server info – used by the ReceiveFrom() function.

**UINT m_uiPort**
Contains the port that the server is running on. Default value = 6000.

**Cstring m_sSocketAddress**
Contains the ip address that the server is running on.

### Public Methods:

**CNetwork()**
Default constructor.

**virtual ~CNetwork()**
Default Destructor.

**bool Start(CString sServerIp, int nServerPort,CPlayer* pPlayer)**
Invokes the Connect() function and set the pointer to the Player.

**bool Stop()**
Invokes the Disconnect() function.

**void SendMessage(CString& strText)**
Used to send messages to the soccer server. Invokes the Sendto function to send the message which returns the length of the message sent. That value is then compared to the original message length. If it matches, the message was sent successfully, otherwise there was an error sending the message.

**void Disconnect()**

Simply invokes the socket's disconnect function so that the socket will no longer be bound to the server address.

## Private Methods:

**BOOL Connect(CString serverIp, int serverPort)**

Takes as parameters the server IP address and the server port. Then uses these parameters to bind the socket to the IP address and port specified for the Soccer Server. Returns a Boolean value based on the success of the connection.

**BOOL Bind(UINT nSocketPort, LPCTSTR lpszSocketAddress)**

Attempts to bind the socket to the specified port and address and returns true is successful.

**int ReceiveFrom(void* lpBuf, int nBufLen, CString& rSocketAddress, UINT& rSocketPort, int nFlags)**

Used to actually receive the message.

**void OnReceivedMessage(CString msg)**

Once a message is received, transfers message to the player.

**void OnConnect(int nErrorCode)**

Notification that the socket has connected.

**void OnReceive(int nErrorCode)**

Invoked when a new message is waiting to be received. Calls the receiveFrom() function to actually receive the message. Then invokes the OnReceivedMessage once the message is successfully received.

# APPENDIX E

## Parser Class Data Members and Methods

### Public Data Members:

**CGameData* pGameData**
Pointer to GamaData data structure.

### Public Methods:

**Parser()**
Default constructor.

**virtual ~Parser()**
Default destructor.

**char parseMessage(string sMsg)**
Entry point for Parser. Determines which of the five message types the incoming message is and calls the appropriate parsing function. First, the function calls isCompleteMessage() to validate message - rejects message if it is not validated. Uses the fourth character of the message string to determine message type, since this character position is unique for all valid messages types. Returns a char indicating type of message.

### Private Methods:

**bool isCompleteMessage(string sMsg)**
Performs a rudimentary message validation by testing that for every opening parenthesis there is a closing parenthesis. (Every complete valid message will have matching sets of parentheses.) The function walks through the message string, incrementing nCount for each opening parenthesis encountered, and decrementing nCount for each closing parenthesis. If nCount becomes negative, a closing parenthesis was encountered without a previous opening parenthesis, and therefore the function preemptively returns false. If, after traversing the entire string, nCount is zero, the function returns true, otherwise it returns false.

**int extractTokens(vector<string> &vsTokens, string sStr)**
Called by all parsing functions to extract the individual tokens from the passed in string. Places each token into a cell of a vector local to the calling function. Returns the number of tokens extracted.

The next five functions parse the specified incoming messages. The first four of these functions operate in exactly the same way. First, they establish a vector of strings to hold the tokens found in the message. Next, they call the extractTokens() function to get the tokens from the message and store them in the vector. Lastly, they populate the appropriate data structure in GameData. Note that some data in GameData is stored as strings, while other data is stored as numerical data. The numerical data is converted from type string to integer or floating point types using the atoi() and atof() functions. These conversion functions require C-style zero-terminated character-array strings as input. Therefore the c_str() function is called to convert from type string to the C-style zero-terminated character-array string.

**void parseInitMsg(string sMsg)**

Parses (init ...) messages: (init Side UniformNumber PlayMode)
Every valid (init ...) message contains exactly 4 tokens:
0) the message name "init"
1) Side = l|r
2) UniformNumber = 1-11
3) PlayMode = a valid play mode

**void parseReconnectMsg(string sMsg)**

Parses (reconnect ...) messages: (reconnect Side PlayMode)
Every valid (reconnect ...) message contains exactly 3 tokens:
0) the message name "reconnect"
1) Side = l|r
2) PlayMode = a valid play mode

**void parseSenseBodyMsg(string sMsg)**

Parses (sense_body ...) messages:
(sense_body Time (view_mode ViewQuality ViewWidth) (stamina Stamina Effort)
(speed Speed)(head_angle HeadDirection) (kick KickCount) (dash DashCount)
(turn TurnCount) (say SayCount) (turn_neck TurnNeckCount))
Every valid (sense_body ...) message contains exactly 22 tokens:
0) the message name "sense_body"
1) Time = game cycle
3) ViewQuality = high | low
4) ViewWidth = narrow | normal | wide
6) Stamina
7) Effort
9) Speed
11) HeadAngle = relative angle of head facing direction to body facing direction
13) KickCount = # of kick commands received by server
15) DashCount = # of dash commands received by server
17) TurnCount = # of turn commands received by server
19) SayCount = # of say commands received by server
21) TurnNeckCount = # of turn_neck commands received by server
Tokens 2,5,8,10,12,14,16,18,20 are just headers for the various values.

**void parseHearMsg(string sMsg)**

Parses (hear ...) messages: (hear Time Sender Message)
Every valid (hear ...) message contains exactly 4 tokens:
0) the message name "hear"
1) Time = current game cycle
2) Sender = referee|self|Direction
3) Message = a message text string

**void parseSeeMsg(string sMsg)**

Parses (see ...) messages:
(see Time (ObjName Distance Direction DistChanged DirChanged BodyFacingDir HeadFacingDir))
(see Time (ObjName Distance Direction DistChanged DirChanged))
(see Time (ObjName Distance Direction))
(see Time (ObjName Direction))
ObjName = (player [TeamName [UniformNumber [goalie]]])
(ball)
(line l|r|t|b)
(goal l|r)
(flag c [t|b])
(flag p l|r c|t|b)
(flag g l|r t|b)
(flag l|r 0|t|b [10|20|30])
(flag t|b 0|l|r [10|20|30|40|50])
(Player)
(Ball)
(Goal)
(Flag)

example:
(see 3004 ((flag l t) 37 19) ((goal r) 25 -43 -0 -0) ((flag g l t) 24.3 -27 -0 0) ((flag p l t) 16 31 -0 0.1) ((flag t l 40) 34.8 40) ((flag t l 50) 39.3 26) ((flag l 0) 30 -41 -0 -0) ((flag r b) 29.4 -22 -0 0) ((ball) 32.5 -4 -1 12) ((line l) 27.9 60))

Each (see ...) message consist of a number of entities and the data for each entity. An entity is an object on the field the player “sees” - player, ball, line, goal or flag. If an entity is out of the player’s field of view but is in close proximity (within 2.2 units), the player “senses” the entity. In the (see ...) message, the name of the sensed entities begins with an upper case character – Player, Ball, Goal, and Flag. (Line entities are not sensed by the player.) The number of entities in each message may vary greatly.

First, the function calls resetVisualInfo() to clear out old data, since the old data may no longer be valid. The function steps through the message searching for each entity and its data. Each entity is preceded by a pair of opening parentheses "((". As each entity is located, a specific function is called to parse that entity.

**void parseEntity(string sEntity)**

Determines which of the five entity types the incoming entity is and calls the appropriate parsing function.

The next five functions parse the specified entities of the (see ...) messages. Basically, all of these functions operate in the same way. First, they establish a vector of strings to hold the tokens found in the entity. Next, they call the extractTokens() function to get the tokens from the message and store them in the vector. Lastly, they populate the appropriate data structure in GameData. The number of tokens (i.e., the amount of information given) for each entity varies depending on how well the player "saw" the entity. If the player was close enough to the entity, all data for that type of entity is given. However, as the distance to the entity increases less data is given. Note that some data in GameData is stored as strings, while other data is stored as numerical data. The numerical data is converted from type string to integer or floating point types using the atoi() and atof() functions. These conversion functions require C-style zero-terminated character-array strings as input. Therefore the c_str() function is called to convert from type string to the C-style zero-terminated character-array string.

**void parseFlagEntity(string sEntity)**

Parses ((flag) ...) & ((Flag) ...) entities:
((flag x [x [x]]) Distance Direction [DistChanged DirChanged])
((Flag) Distance Direction)
Before extracting the tokens, the name of the flag must be extracted. The flag's name is passed to the getFlagIndex() function to get the index for storing the data into an array in GameData. There are 53 flags on the field, each with a unique name. (example: "flag t l 40" is located 40 yards from center field along the top left boundary.) The flag array in GameData has 54 cells (0 – 53). Cell 0 is reserved for a non-named proximity flag – a flag that is out of the player's field of view but the player "senses" due to its close proximity. (Note that the proximity flag is actually one of the 53 uniquely named flags but the name is not known.) Each of the cells 1 through 53 are reserved for a specific flag. See the "constants.h" file for an enumeration of the flags.

**void parsePlayerEntity(string sEntity)**

Parses ((player) ...) & ((Player) ...) entities:
((player [TeamName [UniformNumber [goalie]]])
Distance Direction [DistChanged DirChanged [BodyFacingDir HeadFacingDir]])
((Player) Distance Direction)
The player entity contains two types of tokens – tokens that identify the player and data tokens: distance, direction, etc.. The number of each type of token (i.e., the amount of information given) depends on how well the player was "seen".

**void parseBallEntity(string sEntity)**

Parses ((ball) ...) & ((Ball) ...) entities :
((ball) Distance Direction [DistChanged DirChanged])
((Ball) Distance Direction)

**void parseGoalEntity(string sEntity)**

Parses ((goal) ...) entity:
((goal l|r) Distance Direction [DistChanged DirChanged])

**void parseLineEntity(string sEntity)**

Parses ((line) ...) entity:
((line l|r|t|b) Distance Angle)

**void resetVisualInfo()**

Resets data fields in GameData to default values. Called each time a new (see …) message is parsed, since the old data is no longer valid.

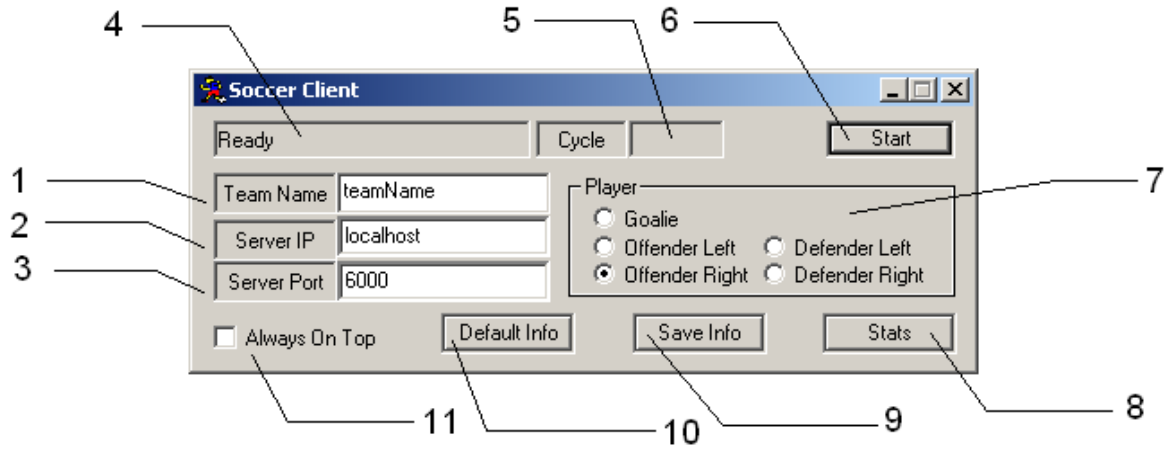
**int getFlagIndex(string sFlagName)**

Assigns a unique integer value to each flag. This value is used as the index to the flag array in GameData.

# APPENDIX F

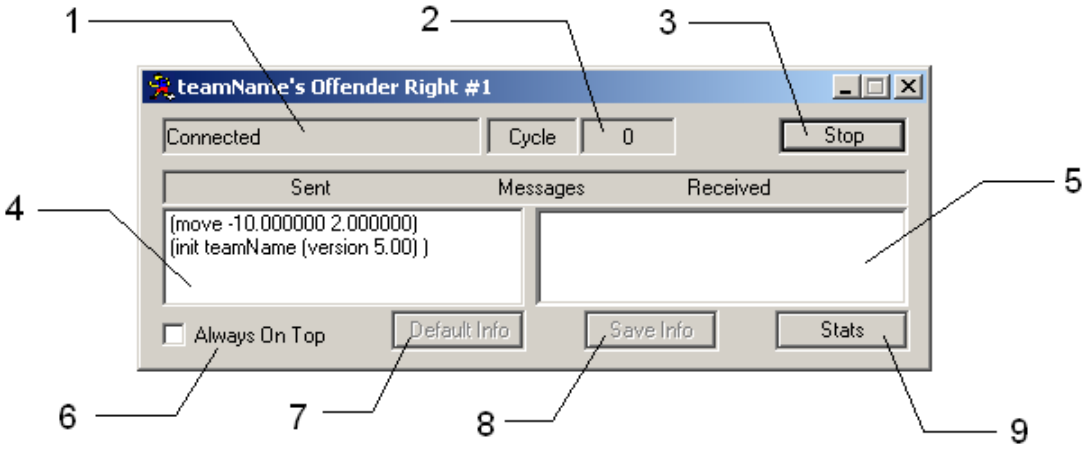
## Soccer Client Dialog

### Soccer Client Dialog at Start Up



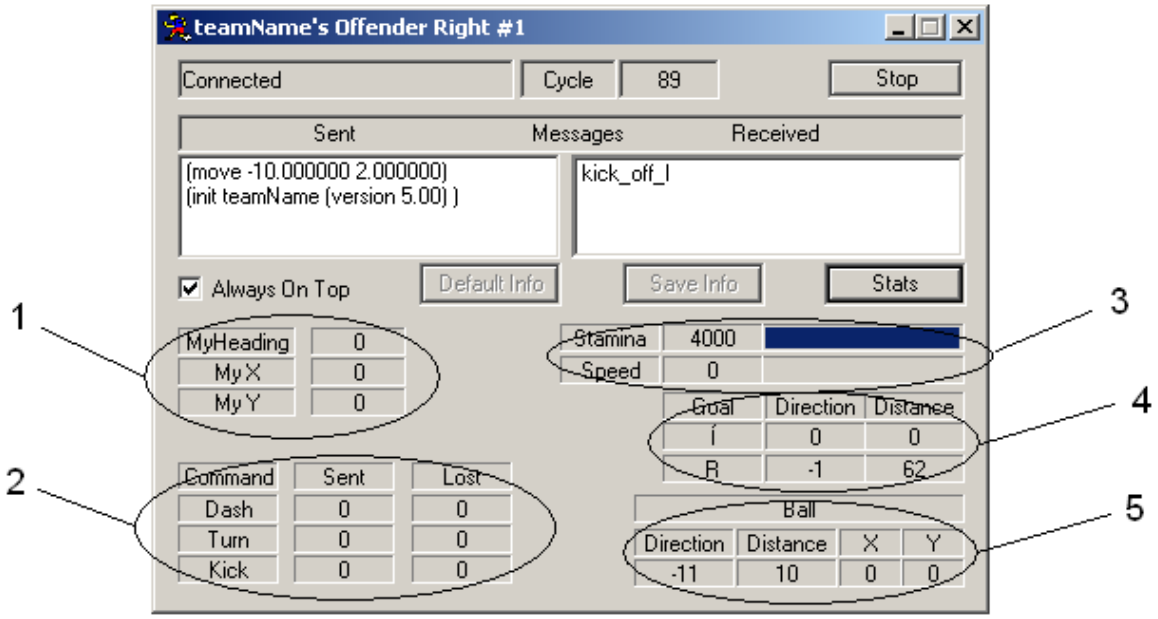

1. Enter your team name here. Must be all characters with no spaces.
2. Enter your server IP or **localhost**.
3. Enter your server port here.
4. Displays the status of the client.
5. Displays the current cycle of the game.
6. Start (connect) and stop (disconnect) client.
7. Select the player to load.
8. Toggle stats view on/off.
9. Save dialog information to client.ini file.
10. Restore default values to dialog.
11. Keeps dialog on top of other windows.

## Soccer Client Dialog after Starting Player




1. Displays the status of the client.
2. Displays the current cycle of the game.
3. Start (connect) and stop (disconnect) client.
4. Displays messages sent by the client.
5. Displays messages received from the referee.
6. Keeps dialog on top of other windows.
7. Restore default values to dialog.
8. Save dialog information to client.ini file.
9. Toggle stats view on/off.

## Soccer Client Dialog Stats View

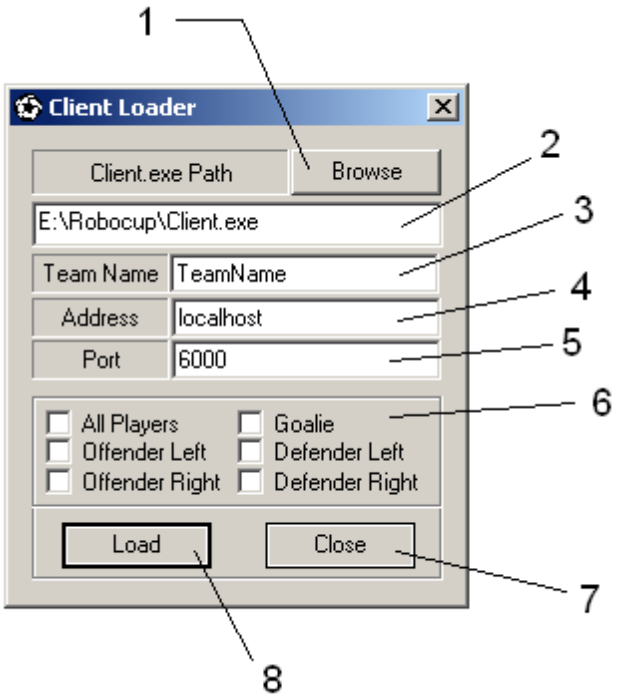



1. Displays information about the player's heading and position. The default client that comes with the package does not contain a coordinate system, however, these fields will aid in the development of one. Just be sure to store the coordinates in the appropriate variables. See CPlayer::calculateCoordinates()
2. Displays the counts for the commands sent to the server it also displays the counts of lost messages. This will show how effective the commands are and which ones need improvement. Note: lost messages do not necessarily indicate the server did not receive them, but may indicate the server did not accept them.
3. Displays information about the client's stamina and speed. Use this information to help manage stamina.
4. Displays information about the goals.
5. Displays information about the ball, as with the player's position the ball's position must also be calculated. See CPlayer::calculateCoordinates()

# APPENDIX G

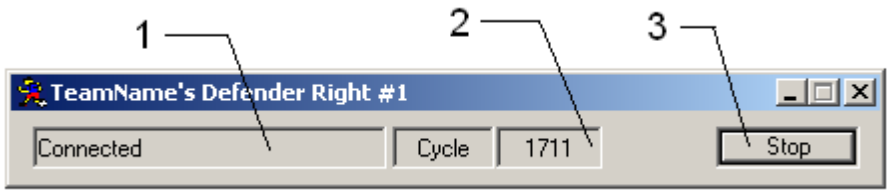
## Client Loader

### Client Loader




1. Use this button to browse for the Soccer Client (client.exe).
2. Shows the path of the Soccer Client (client.exe).
3. Enter team name here.
4. Enter server IP here.
5. Enter server port here.
6. Check all boxes for players to load.
7. Use this button to close the Client Loader.
8. Use this button to load the selected players.

## Soccer Client Dialog in Game Mode




1. Displays the status of the client.
2. Displays the current cycle of the game.
3. Start (connect) and stop (disconnect) client.

This interface only appears if client.exe is loaded (using *Client Loader* or *Command Line*) for tournament play.

**The Command Line for loading the Client.exe is as follows:**

Client.exe TeamName Player ServerIP ServerPort

TeamName::= $(-|_|a\text{-}z|A\text{-}Z|0\text{-}9)^{+}$

Player::= goalie | offenderleft | offenderright | defenderleft | defenderright

ServerIP::= localhost | “a valid IP address that robocup soccer server is running on”

ServerPort::= 6000 | “a valid port that a robocup soccer server is running on”

**Command Line Examples:**

Client.exe myTeam goalie localhost 6000
E:\Robocup\Client.exe Team8 offenderright 192.168.1.110 6000

# APPENDIX H

## RoboCup Files/Folders

**RoboCup –** Contains all RoboCup files. We recommend you copy to C:\ on your computer.

**RoboCup\ClientLoader** – Contains Client Loader files. See **Client Loader** section and **Appendix G** for more information on the Client Loader.

**RoboCup\Documentation** – Contains documentation from the RoboCup Federation (RoboCupFederationManual.pdf) and the redCard_Linux Users Guide (redCard_LinuxManual.pdf).

**RoboCup\LogFiles** – Contains saved RoboCup game files. Open and run saved games from the Soccer Monitor. It is not necessary to run or connect to the Soccer Server to playback logfiles.

**RoboCup\LogFiles\MonitorToViewUSALogFiles –** Contains an older Soccer Monitor (Soccermonitor.exe) to use when viewing the logfiles located in USARoboCup1999 and USARoboCup2000.

**RoboCup\LogFiles\RoboCupFinal2001** – Logfile from the RoboCup World Championship 2001 final game.

**RoboCup\LogFiles\RoboCupFinal2002** – Logfile from the RoboCup World Championship 2002 final game.

**RoboCup\LogFiles\ USARoboCup1999** – Logfiles from the 1999 University of South Alabama A.I. classes.

**RoboCup\LogFiles\ USARoboCup2000** – Logfiles from the 2000 University of South Alabama A.I. classes.

**RoboCup\MyLogFiles** – Logfiles from your RoboCup games will be saved in this folder. This will only work correctly if RoboCup folder structure is maintained. This setting can be changed in the RoboCup\SoccerServer\rcssserver-server.conf file. Edit this file at your own risk.

**RoboCup\PracticeTeams** – Contains several developed RoboCup teams for observation or to test your own teams. Use the Client Loader to load these teams. See **Client Loader** section and **Appendix G** for more information on the Client Loader.

**RoboCup\PracticeTeams\Basic** – Level 1 redCard_Linux team.

**RoboCup\PracticeTeams\Advanced** – Level 2 redCard_Linux team.

**RoboCup\PracticeTeams\TurboAdvanced** – Level 3 redCard_Linux team.

**RoboCup\PracticeTeams\GoodLuck** – This is the 2002 RoboCup World Championship Team. Run RoboManager.exe to load.

**RoboCup\SoccerClient** – Contains Soccer Client Framework files.

**RoboCup\SoccerMonitor** – Contains Soccer Monitor files.

**RoboCup\SoccerServer** – Contains Soccer Server files.