See What Your Robot Sees
Matthew Li, Ashvin Nair
Winchester High School
matthew.yj.li@gmail.com, nair.ashvin@gmail.com

See What Your Robot Sees

Introduction

One of the major pitfalls of embedded systems such as the CBC is that code is hard to run and debug. The classical way of running the CBC involves using a flash drive to transfer code between the computer and the CBC. The creation of this application is in large part due to being able to use a friendly development environment. We write our code in Java through CBCJVM [1], and run it through WiFi using SSH [2]. Java makes it easy to, among other things, make a socket server that sends information between the CBC and the computer running the program. Furthermore, Java's emphatically object oriented structure makes code easier to write, share, and reuse than code written in C.

We made a program that allows the robot and a host computer to talk with each other. While running the Java program from a SSH terminal we can start a socket server that can send messages from within the program to any client connected. For example, every time our robot's location is updated, the robot sends out a message saying something like "Location (24,56)". Our graphical client parses through this information and uses it; for location, it will update the on-screen robot to the new location. This system useful for interacting with the robot and debugging code by visually comparing the robot's commands to its actual actions.

Purpose

The primary purpose of our program is to provide a look at the robot's internal map. Our robot includes a self-updating map which records the approximate location of the robot. Both the location and direction the robot is facing is automatically sent to the graphical client. Often there are issues where the robot does not behave as expected while attempting to navigate around the game board. Through the graphical client, one may view where the robot thinks it is located and what direction it thinks it is facing in real time. Instead of reading through hundreds of lines printed out one looks at a relatively easy to comprehend GUI.

Code Explained

The client uses java's ServerSocket and Socket classes [3]. The ServerSocket is part of the program run by the CBC. You must include the ServerSocket in your program. To run

on our robot, we created a Communicator class which has a ServerSocket. Communicator has the following fields:

```
public class Communicator {
...
   final int PORT = 4444;
   ServerSocket serverSocket;
   Socket clientSocket;
   PrintWriter out;
   BufferedReader in;
...
}
```

A socket server writes to a port on the host computer. Clients must find the socket stream by IP address and port number. In our program, the CBC acts as the host and the computer is the client. The CBC must be connected to the users computer, usually through wifi, for the ServerServer to communicate to the Socket. To initialize the communication, a socket server is initialized at the given port and begins to scan the port for any outside clients trying to establish a connection. From this client, output streams are abstracted to make reading and writing easier.

```
   serverSocket = new ServerSocket(PORT);
   Socket clientSocket = null;
   clientSocket = serverSocket.accept();
   out = new PrintWriter(clientSocket.getOutputStream(), true);
   in = new BufferedReader(
      new InputStreamReader(clientSocket.getInputStream()));
```

Each of these statements must be wrapped in a try/catch-block in the actual program. If you have installed CBCJVM [4] and are using ssh you can use the following commands to send over and run your program:

```
scp Program root@192.168.0.100:/mnt/user/code
ssh root@192.168.0.100 "/mnt/user/jvm/java /mnt/user/Program"
```

Once the program is run it waits until a socket is run from the user's computer, which communicator's clientSocket field is initialized too. The ServerSocket sends messages to the socket through wifi. This is done through the send() method in Communicator, which simply calls out.print().

We use a class called CommClient, which has a socket. CommClient has a main method and is the class we run from our computer to provide communicator with a socket. CommClient's fields are similar to those of Communicator.

```
public class CommClient {
...
   final int PORT = 4444;
   Socket echoSocket = null;
   // Communicator's clientSocket refers to the same Socket as echoSocket.
   PrintWriter out = null;
   BufferedReader in = null;
...
}
```

CommClient performs actions based on the messages it receives from the robot, usually either changing the location of the robot in our GUI or adding/removing objects from the

GUI. CommClient "reads" the messages using it's getNextLine method and processes those messages in the main method within a while loop.

```
public class CommClient {
...
   public String getNextLine() {
      try {
         return in.readLine();
      } catch (IOException e) {
         e.printStackTrace();
      }
      return "No input";
   }
...
   public static void main(String[] args) throws IOException {
   ...
      String robotInput;
      while ((robotInput = client.getNextLine()) != null) {
         System.out.println("received: " + robotInput);
         if (robotInput.startsWith("Robot moved to ")) {
            double x = Double.parseDouble(robotInput.substring(
               robotInput.indexOf("(") + 1, robotInput.indexOf(",")));
            double y = Double.parseDouble(robotInput.substring(
               robotInput.indexOf(",") + 1, robotInput.indexOf(")")));
            double direction = Double.parseDouble(robotInput.substring(
               robotInput.indexOf("[") + 1, robotInput.indexOf("]")));
            // direction is in radians
            board.updateRobot(new Point2D.Double(x,y), direction);
            System.out.println("robot position updated");
         }
...
}
```

The code within the while-loop takes an input such as "Robot moved to (31,15) [0.34]" and calls the GUI method to update the robot position to (31,15) with direction 0.34. In turn, the on-screen robot is moved to the correct position. Our robot sends the direction it is facing and its location every time either of those changes. One can also add additional messages easily to the graphical client. To demonstrate the simplicity of adding new features, we quickly built a system to map walls. First, in the main loop of the robot's program, a section is added to send the a message to the server if the Create's bumpers were triggered, similar to adding a "print" statement:

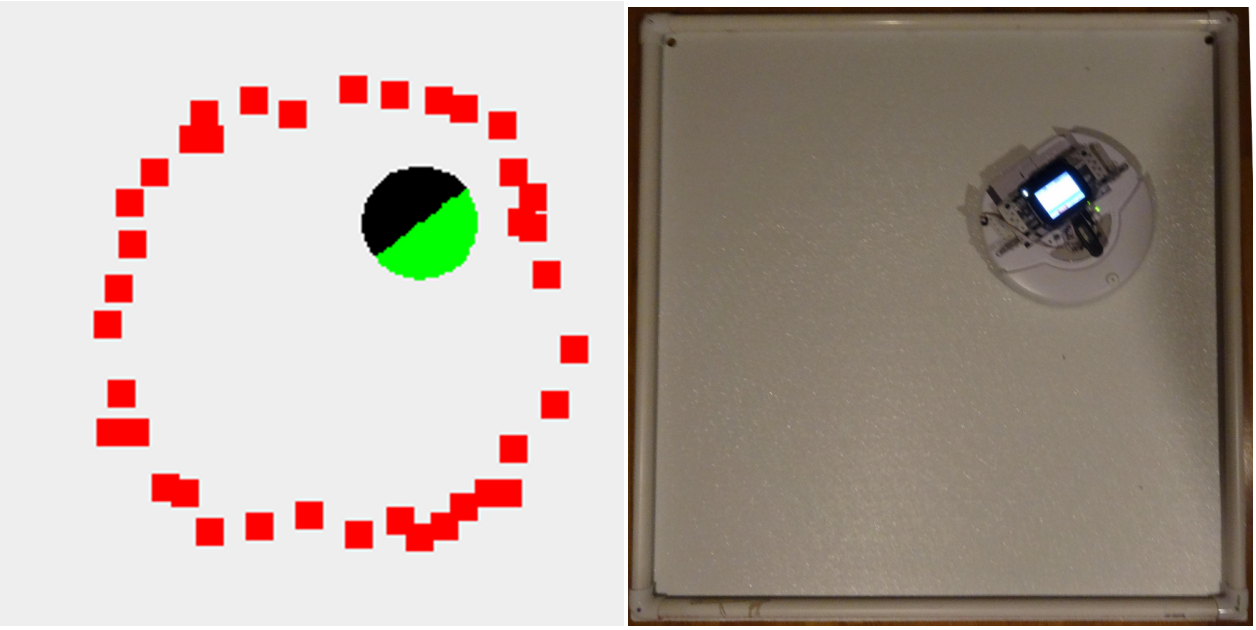
```
if (robot.leftBumper() || robot.rightBumper()) {
   Core.send("Hit wall at (" + robot.getX() + "," robot.getY() + ")");
   // Core.send() calls our communicator.send() method
}
```

Next, parsing code is added to the server's main loop to add a red block in front of the robot on the GUI. For instance, the message "Hit wall at (0,0)" will currently place a block representing the wall at the given location, in this case (0,0).

```
while ((robotInput = client.getNextLine()) != null) {
   ...
   if(robotInput.startsWith("Hit wall at ")){
```

```
            double x = Double.parseDouble(robotInput.substring(
               robotInput.indexOf("(") + 1, robotInput.indexOf(",")));
            double y = Double.parseDouble(robotInput.substring(
               robotInput.indexOf(",") + 1, robotInput.indexOf(")")));
            board.drawObject(new GuiBlock(x, y));
            System.out.println("added block");
        ...
        }
    }
```

This expandability allows us to add additional debugging features such as adding a green pompom at the location the robot thinks it sees one. Once again, instead of reading through a mess of printed lines about the various locations of objects, one can simply see how the robot interprets the object, its location on the board, and its relative location to all the other objects including the robot. Here is what our demo looks like:



(a) A zoomed-in screenshot of the GUI client

(b) The robot running and mapping

Our full Communicator and CommClient classes, along with the working GUI, can be found at `https://github.com/anair13/CBC-communicator` [5].

Conclusion

The GUI Client is still in its infancy. There are still many unimplemented features to be added and creativity is the only limitation of the client. Here are some of our immediate goals to create:

- A graph to read out sensor values, improving the client's usefulness as a debugger.
- Use a camera to constantly identify and localize objects and display them on screen.

- Be able to control the robot remotely from the host computer.

We hope we have encouraged you to go out and make or adapt your own GUI client for your own team to use, play around with, improve, and share with the world. If you have ideas for the GUI or code improvements we would be happy to hear about them. You can email us at whsbotball@gmail.com. Good luck with your GUI Client!

References

[1] B. McDorman, B. Woodruff, A. Joski, J. Frias. CBCJVM Applications of the Java Virtual Machine with Robotics: Concepts and Features. Proceedings of the 2010 Global Conference on Educational Robotics, July 2010.

[2] J. Rand, M. Thompson, B. McDorman. Hacking the CBC Botball Controller: Because It Wouldn't Be a Botball Controller if It Couldn't Be Hacked. Proceedings of the 2009 Global Conference on Educational Robotics, July 2009.

[3] Oracle. Reading from and Writing to a Socket. http://docs.oracle.com/javase/tutorial/networking/sockets/readingWriting.html, March 2012

[4] B. Woodruff. CBCJVM. https://github.com/CBCJVM/CBCJVM, June 2011

[5] A. Nair. CBC-communicator. https://github.com/anair13/CBC-communicator, June 2012