The George Washington University – CSCI 188

BlackBerry Robotics

Research in developing a Microsoft Robotics Development Studio (MRDS) client dashboard for a J2ME-based mobile device

Frank Ervin

12/14/2009

Summary

The MRDS Dashboard application leverages the Research in Motion (RIM) BlackBerry API in order to read data from and send control messages to projects developed using the Microsoft Robotics Developer Studio (MRDS). The MRDS Dashboard is capable of replacing (or augmenting) the included dashboard functionality from a BlackBerry device. The dashboard displays camera images, plots the location of obstacles detected using LIDAR, and sends rudimentary control commands to the MRDS Generic Differential Drive (GDD) service.

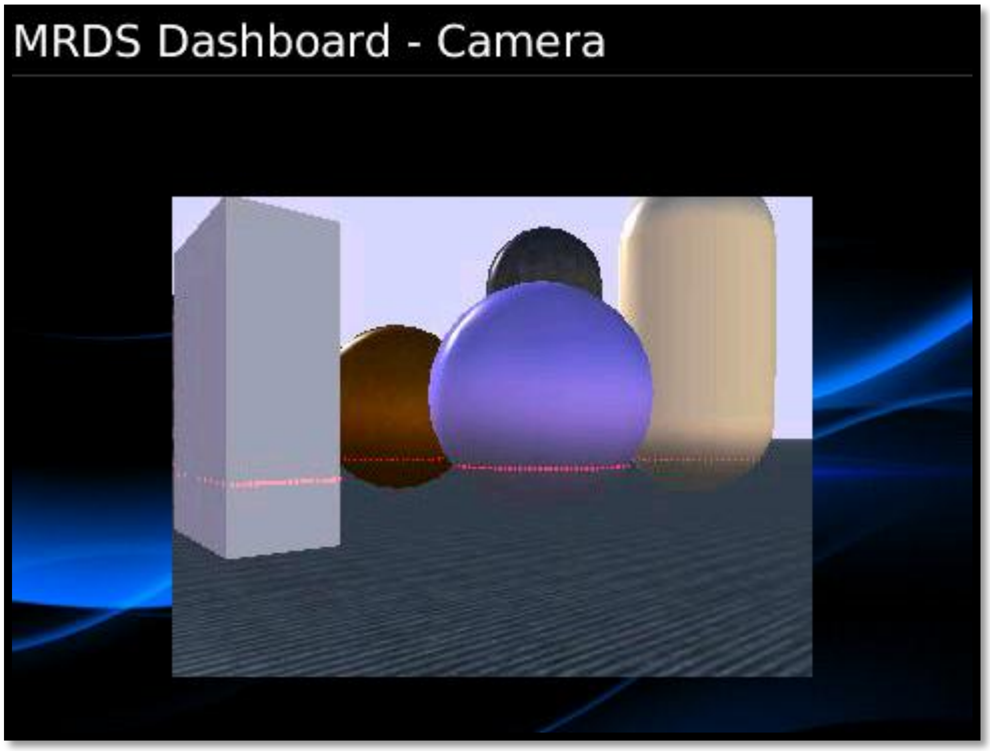


Figure 1: WebcamField rendering the area in front of a MRDS robot simulation

Methodology

The MRDS Dashboard for BlackBerry was written using the BlackBerry JDK Plug-in for Eclipse, version 4.7 with the Beta 4 component pack, which enables v5.0 API compatibility. I found with a little trial and error that this version of the plug-in required Eclipse Ganymede, and not the newer Galileo release. Although RIM has released a Beta 5 release of the plug-in designed for Galileo, I wasn't able to get it running correctly quickly, so I stayed with Beta 4.

After getting the IDE up correctly, I worked on learning BlackBerry development fundamentals and applied these to my in class presentation example. This gave me some workable MainScreens, a Thread implementation, and a MRDSConfig PersistentStore. This is used to store MRDS service endpoint information such as Address, Port, service endpoints, and a connection timeout value.

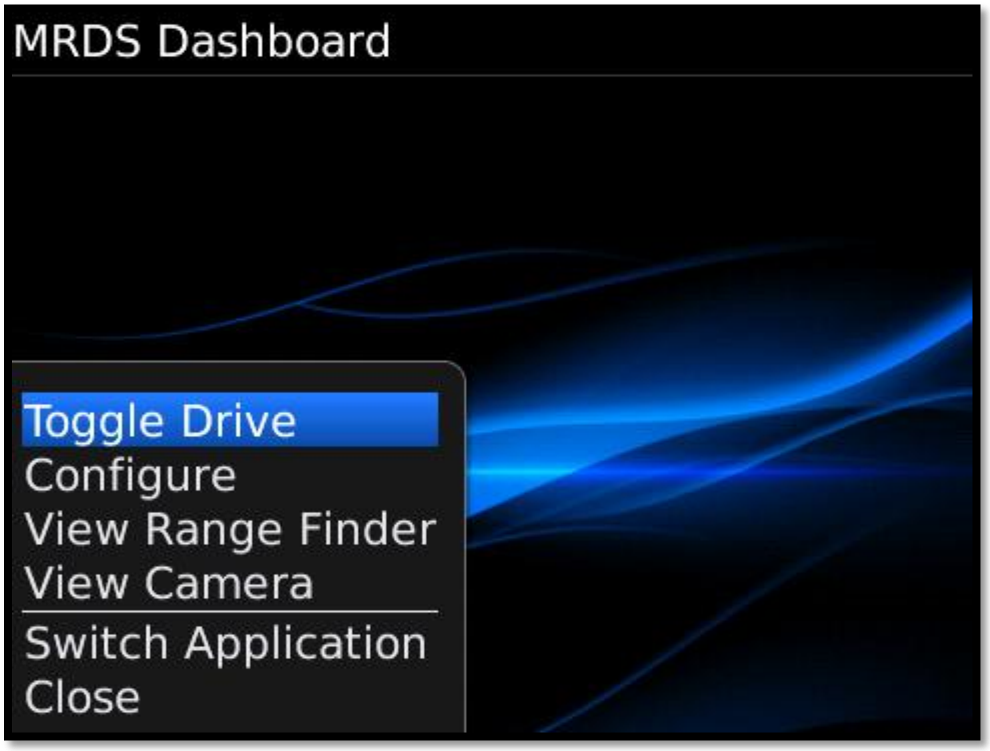
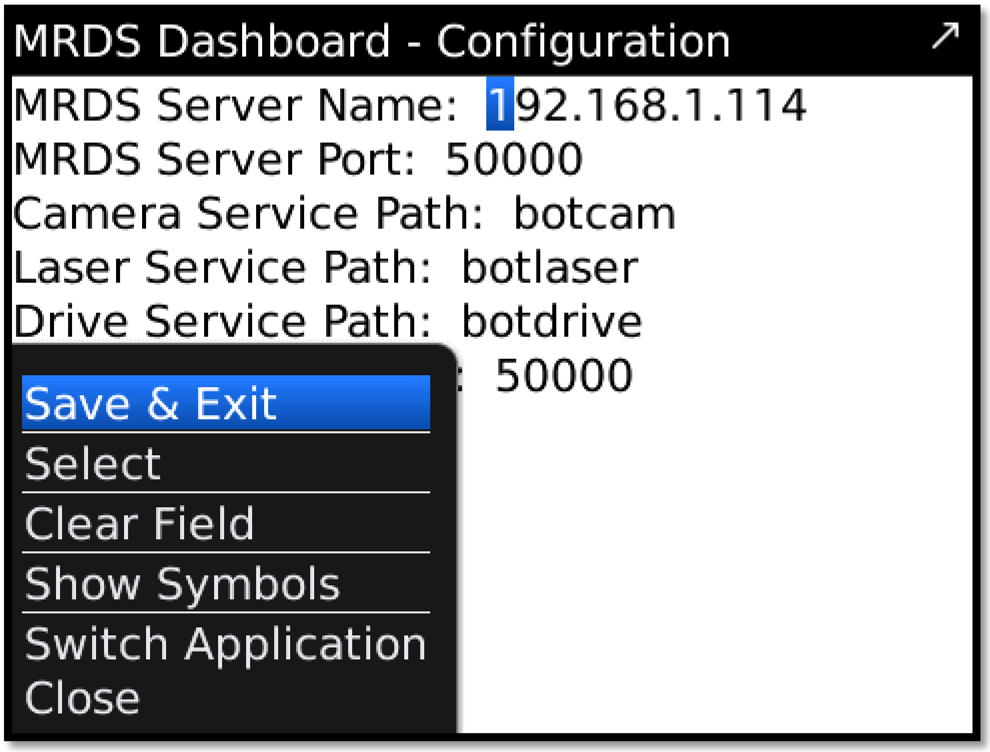


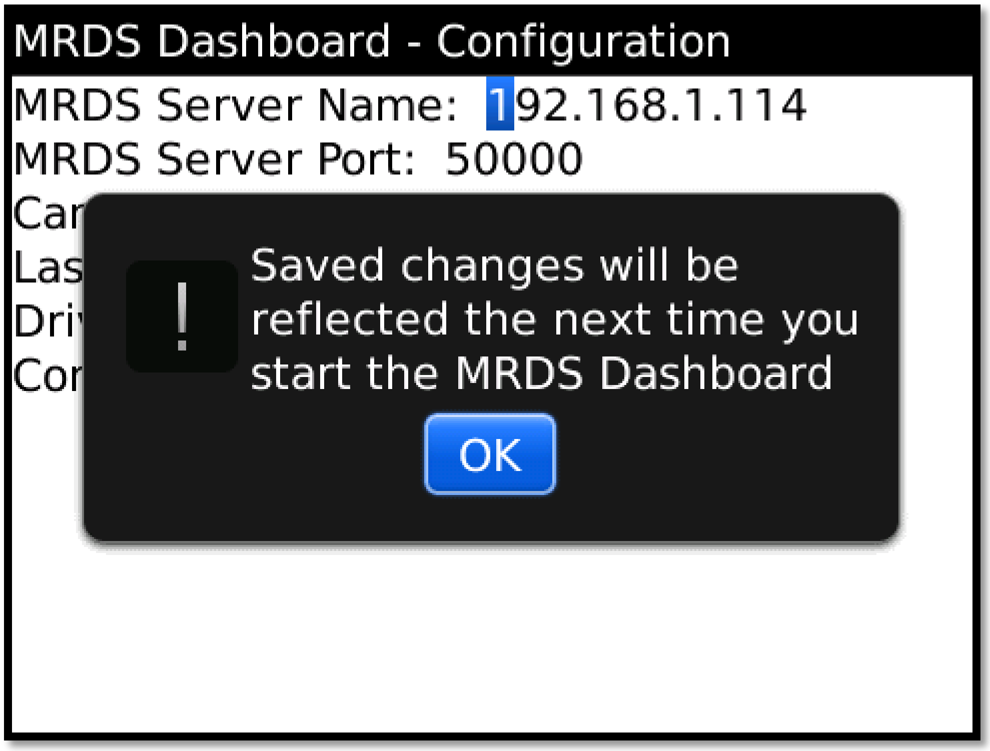
Figure 2: MainScreen of the application, with a listing of all of the MenuItems

At this point, I was on my way but having difficulties understanding exactly how MRDS works. It took a couple of weeks of working in tandem in Eclipse and in the MRDS to get the two to interface correctly. I had to learn about how the DSS manifests work in order to hard-set service endpoint URLs. The MRDS defaults to using GUIDs generated at runtime for service endpoints, which can then be queried using a full DSSP implementation. Since no DSSP implementation exists for Java (yet!), hard-coding the service endpoints was a functional requirement. I also had to learn about how HttpGet and HttpPost service handlers can be used to facilitate non-DSSP interop with MRDS. I struggled attempting to implement a HttpQuery handler for quite some time before I realized that it was not going to be straightforward to implement for all of the services in the MRDS simulation examples, especially the drive service that was built on the Generic Differential Drive service contract.


MRDS Dashboard - Configuration

MRDS Server Name: 192.168.1.114
MRDS Server Port: 50000
Camera Service Path: botcam
Laser Service Path: botlaser
Drive Service Path: botdrive
: 50000

Figure 3: Configuration screen and Save & Exit MenuItem


MRDS Dashboard - Configuration

MRDS Server Name: 192.168.1.114
MRDS Server Port: 50000

Saved changes will be reflected the next time you start the MRDS Dashboard

Figure 4: Dialog after saving

Once I got a basic understanding of MRDS, the first piece I implemented was the WebcamField. This was accomplished by building upon a common RIM API example, the WebBitmapField. This example was limited because it would not update the image when the screen was invalidated. I added some code to facilitate re-requesting the webcam image when the control was invalidated, and found that the control behaved erratically and used far too many threads. To remedy this, I added logic to track the number of background threads and only open them one a time. I also used a RIM API helper function I discovered - net.rim.device.api.io.IOUtilities.streamToBytes(), that could ease in converting an InputStream into a byte array and modified the callback function such that casting to and from a String and a byte array was not needed anymore. This gave the field the responsiveness required to rapidly request and encode image data and have the effect of a video feed.

After completing the WebcamField, I began implementing a custom field to retrieve and display the data from a simulated SICK LIDAR sensor. The field implements the same WebCallback of the WebcamField, and uses a custom LaserHandler SAXParser in the callback handler to populate a custom LaserRecord data structure containing all of the LIDAR measurements. The LaserHandler was inspired by the same SAXParser used in Lab 5. In the field's paint method, the data is used to calculate the Cartesian coordinates of the LIDAR output. They are represented by small circles, and connected using lines. A helper function scaleRange was used to scale the sensor data such that the plot would fit within the size of the LaserField. Getting the entire plotted output to fit correctly and be arranged in the proper perspective took a lot of trial and error.

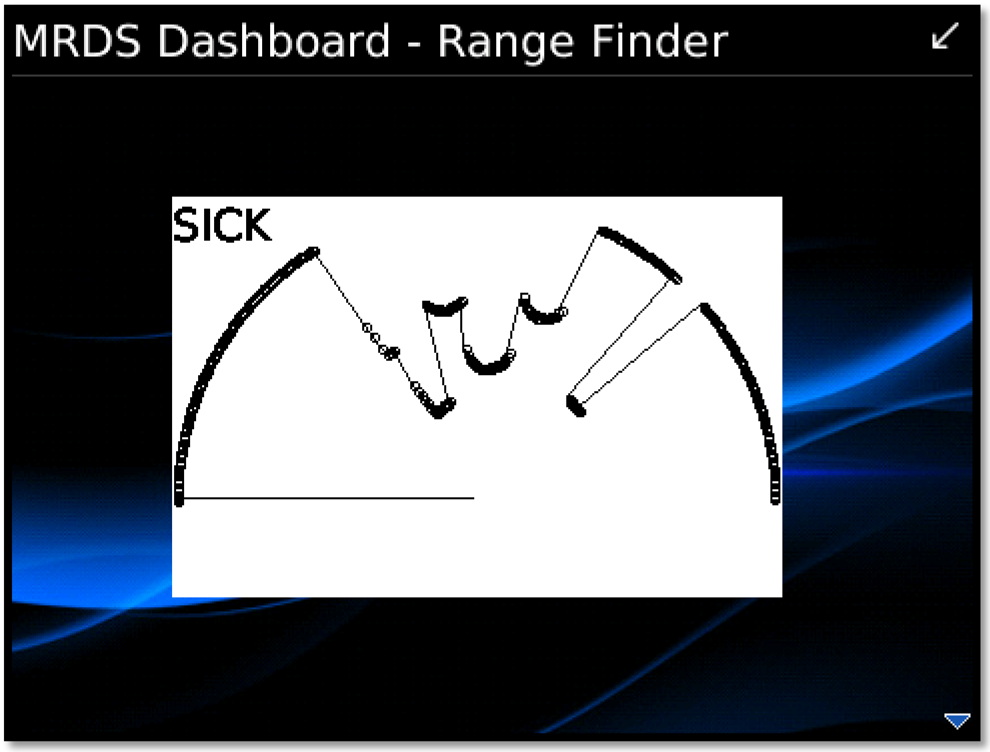


Figure 5: LaserField plotting the LIDAR data

Once I was satisfied with the LaserField, I began trying to control the movement of the GDD using HTTP-standard commands. It took a lot of troubleshooting, but I figured out what needed to be done on both the MRDS and BlackBerry side of things. The MRDS portion still doesn't handle all HttpPosts exactly, and I've been working through that with some assistance on the MSDN forums, but the application is still able to do basic things like enable/disable the motors and rotate the robot. Neither the HttpPost parameter syntax for GDD nor the generic DSSP HttpPost handler itself are described in any Microsoft documentation, so this was a challenge. Ultimately, I was very surprised at how much slower the HttpPost request/response was when compared to the other HTTP operations in the application, so much so that I had to throttle connection timeout settings or risk saturating the thread pool. This flexibility was added to the CfgScreen.

For clarity and reusability, I collapsed all of the pieces of the dashboard which communicate with MRDS into the MRDSProxy helper class. This class encapsulates all of the runnables, including the GetWebcamData, GetLaserData, toggleDrive, and rotateRobot pieces. This library can easily be expanded to include other DSSP operations for MRDS, such as more granular drive control.

Further Research

The work accomplished as part of this project in interfacing J2ME with MRDS could be continued by implementing fields for other sensor and drive types, such as contact bumpers and articulated arms. It would also be beneficial to implement the Decentralized Software Services Protocol (DSSP) in Java. This would make it possible for Java applications to have 100% feature parity with the Microsoft Robotics Developers Studio. Microsoft has released an open source reference implementation in C++ and it's conceivable that this could be used as a basis for such work.