Human Robot Interface

Field Session 2016

Zoe Nacol
Alexander Hintz
Ryan Bailey
Amanual Le Nard

Client: Hao Zhang
Human Centered Robotics Lab

Colorado School of Mines
Golden, CO

Contents

Introduction

Client Hao Zhang is an Assistant Professor in the Computer Science and Electrical Engineering Department at the Colorado School of Mines. He has a Ph.D. in Computer Science and a M.S. degree in Electrical Engineering. He founded the Human-Centered Robotics Lab and focuses his research on Human-Robot teaming, which is the act of humans and robots working together in time and safety critical circumstances.

Project The primary goal for this project was to build a functional interface between humans and robots that can be used to control robots in both search and rescue operations and pipeline inspections. The client requested that we build an interface that runs on Ubuntu and can communicate with and control multiple robots using an open source robotics framework called the Robot Operating System (ROS). Through this interface, the user is able to connect to multiple robots, control multiple and individual robots simultaneously (movement only), display data on the interface from the Inertial Measurement Unit (IMU) and camera for each robot, and save data from the IMU.

ROS Nodes A ROS application is rarely a single program in the traditional sense. Instead, ROS operates as more of a distributed network than as a monolithic program. Each application is made up of several running executables, called nodes, which communicate back and forth with each other, with each node having its own purpose and thread. This is done to promote code reuse, as it is easy to reconfigure nodes or simply reuse nodes in different applications. Nodes are also used to make communication with the robots seamless. After all, a robot is just a collection of nodes, which can publish and subscribe just like any other node. A node can be accessed in C++ code through an object called a NodeHandle which takes in the path of the node in its constructor. The NodeHandle, once initialized, can be used to set up subscribers, publisher and other assorted ROS functions. Many of the custom ROS objects are required to be initialized from a NodeHandle.

ROS Namespaces A running ROS program has the nodes organized in something best described as a file structure. Each node is by default listed in the root of this pseudo file structure. However, just as folders can be in folders, nodes can be in other nodes, and they work much the same way. A namespace is just a way of placing nodes under a common name or node category. The main method of communicating between nodes is through topics. Topics carry information which can be accessed by nodes subscribing to a topic, and broadcasted by nodes publishing to a topic. Topics can be thought of as files: you can have files in folders, but you cannot have a folder in a file.

Publisher and Subscriber Model ROS uses a publisher and subscriber model for communication between nodes. A publisher, on initialization, advertises that it is going to be publishing to a specific topic. Elsewhere in the code, it actually publishes information to that topic. A subscriber, on initialization, requests that information from a specific topic is passed to the subscriber. If and when information is published to that topic, the subscriber will receive the information, and a given function will be called every time, with the topic's message as an input. This function is known as a callback. Each node can contain as many publishers and subscribers as needed. The ROS master node handles sending the messages between the publishers and subscribers.

CHAPTER 1

Requirements

1.1 Functional Requirements

- Select which robot(s) to control (multiple robots can be controlled simultaneously)
- For each robot, the ability to toggle the display of:
  - IMU data on scrolling graph
  - RGB and Depth camera views (user can only display one at a time)
- Ability to toggle saving all IMU data through the interface
- Add, show, hide, and select robots in UI
- Selected robot(s) can be controlled through the following forms:
  - Virtual joystick
  - PS3 controller
  - Foot pedal[0]
  - Voice control[0]

1.2 Non-Functional Requirements

- Must be in C++ and Python
- All code must be compatible with Robot Operating System (ROS) Indigo Igloo
- Interface compatible with Ubuntu 14.04
- Must be compatible with specified hardware, including but not limited to:
  - Microphone / speech recognition device(s)
  - Joystick
  - Foot Pedal

[0]Neither the foot pedal nor voice control were actually implemented; they are currently only dummy options on the interface

1.3 Non-Required Features

We were able to implement several features not asked of us, including:

- Automatic connected robot identification
- Created scripts to make connecting to robots and running the interface easier
- Modified robot launch files to smooth robot motion

CHAPTER 2

System Architecture

2.1 Node Background

At its base, ROS acts as a communication platform between different sections of code, or "nodes". Each node runs in a separate thread, and as such usually has a specific set of tasks or processes that can be run independently. For example, a camera node may only be tasked with communicating with the camera hardware, getting data from it, and passing the data to any nodes that request it. Another node may take this camera data to do image recognition, then pass any notable features to a controller node for other processing.

During each ROS session, a new Master node is created to monitor all other nodes, and to act as a central base for communication. When a section of code is run that defines a node, ROS automatically creates, initializes, and starts the node, and begins to create links between other nodes that are created. Each node can publish or subscribe to a "topic". In other words, a node can send out or receive a specific message. These messages are passed under a specific name, or "topic". When a publisher or subscriber is declared in a node, the ROS Master node is notified, and dynamically links together the publishers and subscribers so that the messages can be passed. When a node receives a message, a callback is called in the receiving node so that the message can be dealt with.

Nodes may also provide services to one another. Services are usually messages or requests that are only passed once in awhile, or messages that require a response, such as when requesting a variable from another node. Because they run parallel to other processes, services may be used to accomplish a task that may take longer to complete, without suspending other code. A service publishes "true" when it is complete, and any response data can be gathered by the calling node.

2.2 System

There are four main components of our system: mhri_plugin, mux, bot_window, and mhri_joy. The sections below discuss what each component is responsible for, as well as how it connects to the others.

mhri_plugin The main user interface window is mhri_plugin. The Qt library is used for the GUI and automatically generates a ROS Qt node out of mhri_plugin. This is the initial window seen when launching the interface. From this window, the user can add bots, select bots, move selected bots, and select the control type. When a bot is selected (through the check box next to its name in the connected bot list), a bot_window appears, which holds information about a specific robot. mhri_plugin publishes the list of selected robots to all bot_windows so that each bot window knows when to pass on movement information to the robot. Finally, When the control type is changed, mhri_plugin publishes the new control type to mux.

mux Movement of the robot(s) is handled by mux. It takes control inputs from all controller forms (virtual joystick and PS3 controller are the only ones working currently) but only focuses on the input from the

selected controller type, set by mhri_plugin. It then converts the input message into the standard movement message type for the Turtlebots, called a twist message. This twist message is then published to all the bot_windows.

bot_window Each robot is given a bot_window when added to the GUI. The window can vary depending on the robot type, but there is currently only one window implementation, created for Turtlebots. To add robots of other types, a new window that extends QDialog must be created (assuming it's not compatible with bot_window). QDialog is a member of the Qt framework. In this window, the user can toggle both data streaming and data saving for the robot. Currently, this window allows the user to stream IMU and camera data from the Turtlebot sensors. Users can view the camera data in two forms: RGB and depth images. The IMU data is displayed as a scrolling graph showing the linear velocity. bot_window subscribes to two topics: twist messages from mux, and the selected robot list from mhri_plugin. When a twist message is published, each bot_window cross references the list from mhri_plugin to see if it is selected. If selected, it publishes the twist message to its corresponding robot.

mhri_joy To allow the user to control the robot using a PS3 controller, something must be used to convert the USB signals into data that mux can use. That is what the mhri_joy node does. While a joy node is included as part of ROS, mhri_joy was used instead so that it could easily be modified if any new features were desired. When mhri_joy is launched, it is immediately suspended to prevent it from constantly searching for USB joystick devices. When "PS3 Controller" is selected from the interface, the mhri_joy node is unsuspended, and tries to connect to the selected joystick device. The node then takes in data from the USB device and publishes it to mux under the topic mux_in/ps3joy.

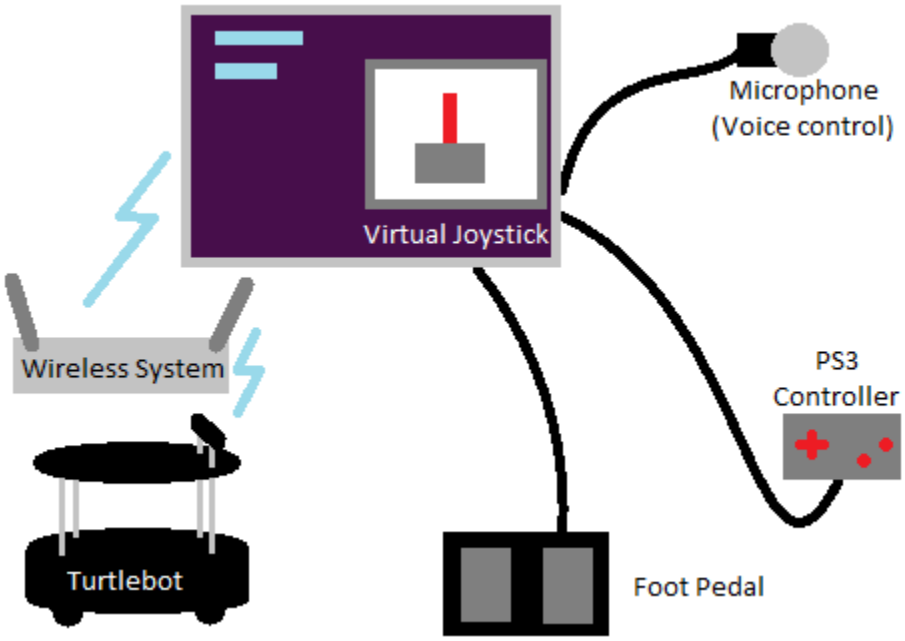


Figure 2.1: Diagram representing physical connections in the system

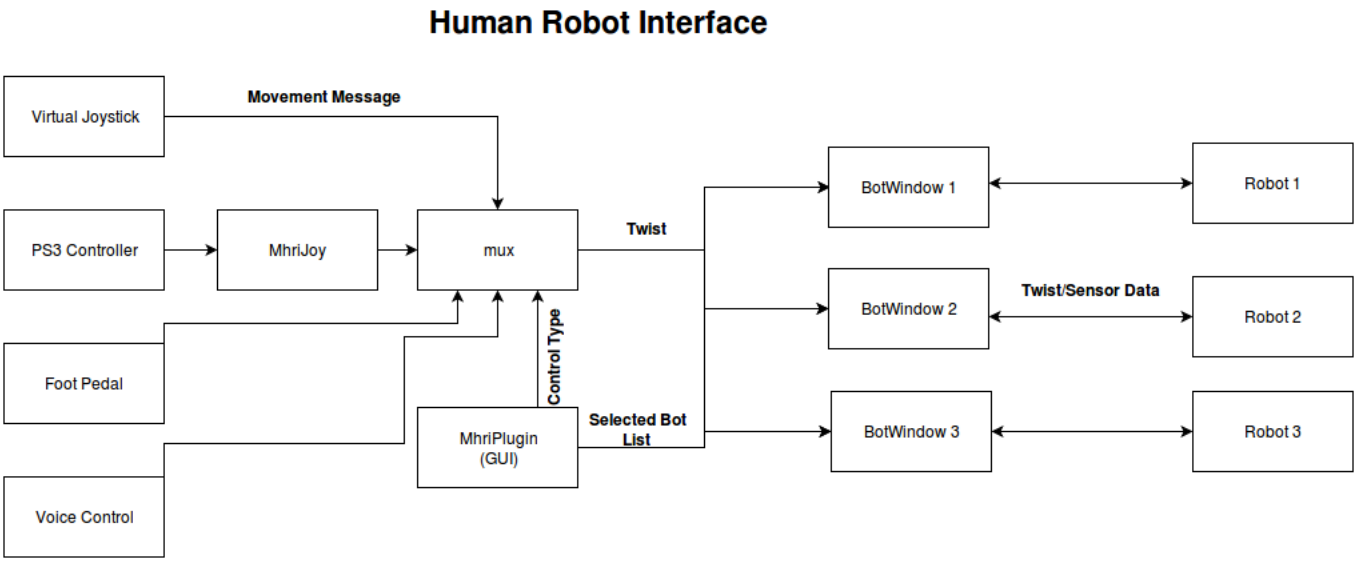


Figure 2.2: Node graph of system; bold text over arrows represents message type

CHAPTER 3

Technical Design

3.1 Multiple Robots in ROS

3.1.1 Overview

One of the main components of this project was figuring out how to connect multiple robots to one master at the same time. This is difficult, as by default, all robots that connect to the system listen to the same topics, and thus the same commands. Therefore, we had to find a way to distinguish each robot, and to set each robot to listen to different commands, all without knowing the order in which the robots are connected. Once we found a way to separate the robot signals, we could then broadcast to multiple robots at a time to achieve simultaneous control. We achieved this distinction using namespaces. ROS allows nodes to be pushed into a custom namespace via the use of groups, or by editing the code directly. However, because we wanted to keep the Turtlebot code unmodified so that the project would be more generally applicable, we simply modified the launch files on the Turtlebot instead of modifying the Turtlebot code. This allows us to put all nodes running on the Turtlebot into a namespace, so that nodes and topics within the Turtlebot can be identifiable. Throughout the documentation and this report, we generally refer to this namespace as the "robot identifier" or "robot name".

3.1.2 Modifying Launch Files

A ROS program is launched by calling a XML-based launch file. One aspect of the launch file syntax has a structure called a "group"; any launch file details inside of a group will become part of that group's namespace. The launch file syntax also supports calling a launch file from within another launch file. Utilizing this, we created a custom launch file that takes in a name as an argument, and creating a group under that name. Our custom launch file then calls the built-in launch files for the robot. Using this, we modify the Turtlebot code as little as possible. To actually launch a robot, we use a custom script to set the environmental variables, then call the launch files to set the namespace and start the robot. These files could be easily modified to allow for different ways to set the name, or to even create a permanent name.

3.1.3 Auto Robot Add

By giving each robot an identifier through the use of custom launch files, we were able to utilize our knowledge over the other nodes in our system to create an interesting and useful capability. When the auto connect button is pressed in our main screen, the system queries the master node for all nodes running on the system. We then remove any nodes that are in a hard-coded blacklist, which contains nodes we know are not robots. Next, we remove any nodes from the list that are part of the namespaces of the robots we already know about. Therefore, any remaining nodes must belong to a new robot namespace, assuming the user does not add any new custom nodes without adding them to the blacklist. We parse the node name for the namespace, and spawn a new robot window for that namespace. At first, we had the auto connect function running on a one second timer. However, we were worried about any performance impacts that

might occur from this, and moved it to a button-activated capability. While this addition is a small one, it has gone a long way towards increasing the usability of our system.

Listing 3.1: Topic List

```
/mux_in/joy
/mux_in/mode_select
/mux_in/ps3joy
/robot_controller/robot_select
/robot_controller/twist
/turtlebot/incompatible_rapp_list
/turtlebot/rapp_list
/turtlebot/status
```

Listing 3.2: Topic List for 2 different robots

```
/bot1/camera/depth_registered/image_raw      /bot2/camera/depth_registered/image_raw
/bot1/camera/rgb/image_raw                   /bot2/camera/rgb/image_raw
/bot1/cmd_vel_mux/active                     /bot2/cmd_vel_mux/active
/bot1/cmd_vel_mux/input/navi                 /bot2/cmd_vel_mux/input/navi
/bot1/cmd_vel_mux/input/safety_controller    /bot2/cmd_vel_mux/input/safety_controller
/bot1/cmd_vel_mux/input/teleop               /bot2/cmd_vel_mux/input/teleop
/bot1/mobile_base/commands/controller_info   /bot2/mobile_base/commands/controller_info
/bot1/mobile_base/commands/digital_output    /bot2/mobile_base/commands/digital_output
/bot1/mobile_base/commands/external_power    /bot2/mobile_base/commands/external_power
/bot1/mobile_base/sensors/imu_data           /bot2/mobile_base/sensors/imu_data
/bot1/mobile_base/sensors/imu_data_raw       /bot2/mobile_base/sensors/imu_data_raw
/bot1/odom                                   /bot2/odom
/bot1/teleop/input                           /bot2/teleop/input
/bot1/teleop_velocity_smoother/param_descrip /bot2/teleop_velocity_smoother/param_descrip
/bot1/teleop_velocity_smoother/param_update  /bot2/teleop_velocity_smoother/param_update
```

3.2 Bot Window and Class Based Robots

3.2.1 Overview

One of our main goals during this project was to leave the code base open to expansion in the future. Throughout this entire project the only robots we were working with were Turtlebots, but we did not want to hard code only for them. Furthermore as each robot gets its own pop-up window it made sense for each window to directly publish and subscribe to the robots thus avoiding all middlemen. In order to make this work we leverage the power of class based design to design a template for the Turtlebot which we create an instance of for each robot we add to the GUI. In order for our program to be open to expansion we set a ROS parameter in the Turtlebot launch file. The main GUI can access these parameters which tells the main GUI that it is a Turtlebot. Thus, the GUI knows to spawn a Turtlebot template for that robot. All one has to do to add additional types of robots is to make a template for them and modify the robot launch file.

3.2.2 How Templates Work

Our client requested that each robot has their own window. Having the UI structured this way easily fit with the object oriented design. For each type of supported robot, of which there is only one in the current code, gets their own window object in the code. This window object contains all necessary information to publish and receive data from the robot requiring only the namespace of the robot to be given to it. These template will differ for each type of robot based on the main aspects of said robot. An example would be the fact that the Turtlebots have an IMU in their base that records data as they rotate. Some robots would not have this feature, such as a pipeline robot, which does not rotate on an axis. Each template is specialized for each type of robot and their unique UI needs. This way it is very easy to add support for new robots requiring only a custom launch file and a custom GUI object.

3.2.3 Turtlebot Template

The only current template we have is for the Turtlebot which was the type of robot supplied for the development of this project. The Turtlebot template comes with a few features. The base Turtlebot is simplistic with a few main sensors for IMU data including velocities and positioning information and a camera feed that has multiple modes including RGB and depth camera. For this project we created a template that utilizes these features and displays them. Our template takes the IMU messages and image messages that are being returned from the robot. We subscribe to these messages from a NodeHandle and then transform the data being returned from these messages. For the IMU data, we create a scrolling graph which uses float data values from the IMU messages as data for the graph. The image data is returned in a form initially incompatible with our interface. However, converting the image message into another more standard form, we can successfully display the image in the interface.

CHAPTER 4

Decisions

4.1 Using C++ over Python

The code we were provided with to start was all in C++. While we did rewrite many sections of the code, we stayed with C++ for both consistency and ease. This is because most example code we found when doing research was in C++ rather than Python. C++ is also more compatible with Qt, the graphics library we are using. Most of our group had more experience with C++ than with Python.

4.2 Using bot_window instead of dedicated Robot Class

Originally we worked on a design which used a robot class that could be expanded to accommodate more types of robots. However, we decided to remove our robot class and instead use windows to represent each robot. This is still just as expandable as a robot class. However, it seemed redundant to have a robot class when we represent each of the robots through its own independent windows. We made this decision first and foremost because our client preferred separate windows each robot rather than everything appearing on one window. It was also easier to manipulate each robot's camera and IMU data through a new window rather than on the main interface.

4.3 Not Using Rocon

Rocon (an extension of ROS) allows you to control multiple robots at the same time- one of the most important requirements for our project. However, due to the little, outdated documentation and lack of support and examples online we ended up spending two days attempting to learn this framework. We made little to no headway and ran into numerous issues in those two days. Eventually we decided that Rocon was not something we could use.

4.4 Using mhri_joy node instead of ROS joy_node

Although ROS comes with an embedded joystick node we decided to create our own joystick node for the following reasons:

- This was modified to allow easy switching between input devices, something that was not easily done using the original code. A service was also added to suspend readings and restart them on demand, as to avoid unnecessary CPU load.
- Because these two features were desired, and it proved easier to directly modify this code instead of finding other solutions, then for any future changes required, this can be directly edited rather than relying on other work-arounds.

4.5 Creating custom scripts

The custom scripts were, like all great innovations, born out of laziness. Each time we ran our program we had to make sure the environmental variables were set properly on both the robot and the computer and had not changed between runs. As the variables are local only to the terminal window they were set in and we were often in the course of testing switching what computer the robots were connected to this got to be very timing consuming and frustrating. If the variables are not set properly the program will not work at all and print out a cryptic error message saying that the ROS master could not be found or that the robot could not ping its own server. In order to reduce this point of failure and to speed up testing we decided to create our own bash scripts that would set the environmental variables for us and then call the appropriate ROS commands. More info about all the scripts we wrote and what they do is in Appendix A

4.6 Manual testing over ROS unit testing

Although ROS has a unit testing framework that we could have used to test our code, we decided to forego this framework in favor of manual testing. The decision was made based on the fact that ROS has little documentation, and online examples were not exceedingly helpful. Due to this, learning the testing framework for ROS would have taken a large chunk of our project time that we did not have.

CHAPTER 5

Results

5.1 Future Work

- Automatically detect and switch between input control (Voice, Pedal, etc.)
- Increase supported camera image data (laserscan, point clouds)
- Support for viewing camera stream using virtual-reality headset
- Ability to add robots of type other than Turtlebot
- Expansion via additional robot accessories (other sensors and tools)

5.2 Unimplemented Features

More Control Types

One of the given goals of this project was to be able to control the robots though several different control types, including a virtual joystick, a PS3 controller, foot pedals, and voice control. Due to time constraints and to limit the project scope, we only chose to implement and test the virtual joystick, the PS3 controller, and control switching functionality. We are leaving it up to future teams to integrate the remaining control types.

Depth Image streaming

There is limited bandwidth to connect and control the robots. The camera streaming is very bandwidth intensive, and as such, it is very important to limit its use by only sending the compressed image instead of the raw picture. We managed to do this with the RGB image, but could not get it to work with the depth image. The compressed data should be used in the future for any camera data sent back to the user.

5.3 Performance Testing Results

All performance testing was based on observations made while using and developing the application. Performance was rated on how well the robot was able to respond to user input, as well as the latency of the data that was being returned to the user. Performance results were as follows.

- Network lag was found to be the most limiting testing result. We would test the system by moving the robot and checking if there was any noticeable lag in the camera view on the interface. Overall, the camera image was smooth and had almost no delay when being updated on the GUI.
- The application is able to handle multiple data input streams from all sensors simultaneously and smoothly, even when using multiple robots.

5.4 Summary of Testing

All of the testing was done manually, even though ROS provides a unit testing framework. Testing was primarily focused on the stability of the interface when connected to robot(s), to ensure no bugs were found or any unexpected crashing occurs.

- In an attempt to crash the program, we tested the interface thoroughly by toggling sensor data streaming and data saving, including:
  - Nothing on with/without robot movement
  - Camera on with/without robot movement
  - IMU on with/without robot movement
  - Camera and IMU with/without robot movement
  - Camera, IMU, and saving data with/without robot movement
- We tested the data saving capability by trying to save data to new files, existing files, and to files within directories that did not exist.

Overall, the testing process revealed bugs that we were able to fix. We were able to prevent the program from crashing unexpectedly, and defined a default location for saved data to be stored when an invalid location was entered into the save file path box.

5.5 Lessons Learned

ROS is an outdated, open source framework with little documentation or support online. As a result, we had an extremely difficult time both learning and using the ROS framework. Looking back, we should have attempted to gain a better understanding of ROS before diving into major changes and refactoring. However, this is difficult due the time sensitive nature of the project.

Originally, the interface window would crash if left open long enough while both the IMU and the camera feed were on. It was later discovered that the QT GUI cannot be edited from callback functions (a function that is called when a message is received). Instead, slots and signals had to be implemented. The callback functions that modified the GUI were then re implemented to emit signals when a new message is received.

APPENDIX A

Startup Guide

A.1 Introduction

This document provides instructions on setting up and starting the Human Robot Interface as well as connecting the interface to Turtlebots. It is divided into two main parts: background and guide. The background provides a general overview of the Human Robot Interface (HRI). The guide offers step-by-step instructions in order to both setup and use the HRI. This document assumes that the reader has a basic understanding of ROS and its underlying structure (more specifically, nodes and namespaces). It also assumes a minimal amount of knowledge of the Linux command line. Due to the many environmental variables needed, the preferred way to start both the interface and connect to Turtlebots are through the provided scripts (discussed later). However, in order to properly use the scripts, understanding the manual set up procedure is highly recommended.

All terminal commands in this document will begin with a $. Ignore this character when typing a command into the terminal. Some commands may have something like (... IP address) in them. If this is the case, follow the steps in "Getting IP Addresses" below to find the appropriate IP address needed.

Note: ROS and the HRI work best when running on Ubuntu 14.04 LTS. While other distributions may work, 14.04 is highly recommended, especially if new to ROS. Some of the provided scripts will not work in non-Debian distributions.

A.2 Background

A.2.1 Getting IP Addresses

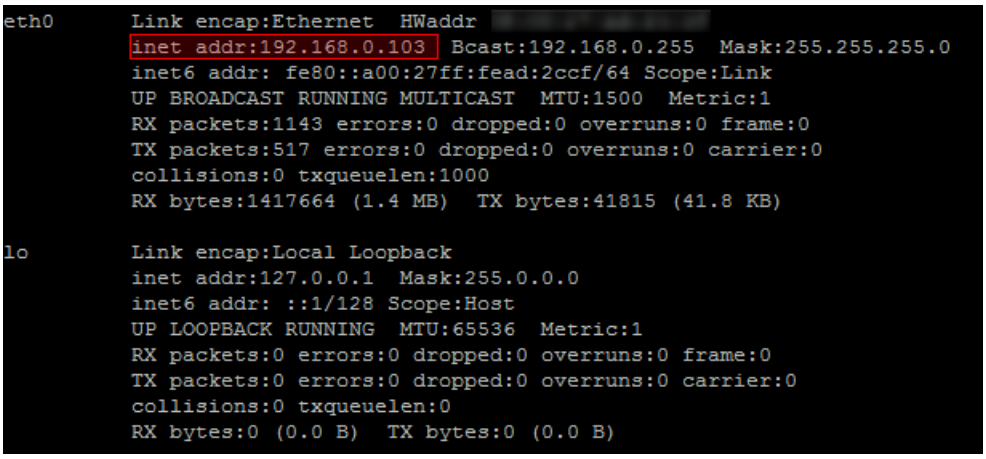
In order to get the IP address for any computer, run the following command (remember to ignore the $):

```
$ ifconfig
```

You should get an output something like the below image. The IP address is highlighted in red.

A.2.2 Turtlebots and ROS

One of the goals of this project was to minimize modifications made to the Turtlebot code as much as possible. However, using the base launch files for the Turtlebot prevented connecting to multiple robots at one time due to all robots using the same node name. As a result, each robot launches under their own group with a unique group name for each robot. This is done by creating a custom launch file that creates a group name and then calls the built in Turtlebot launch files. There are two custom launch files in this project: the minimal Turtlebot launch and the Turtlebot sensor launch.



ROS by default will automatically send data between the launched nodes in a network. The way ROS does this is by having each node send its information to a central node called master that figures out how to hook up all the publishers and subscribers. In order to get this to happen ROS needs to know where the master node is and where nodes are in the network. This is done through environmental variables set on the command line. The ROS Master URI tells the nodes where the master node is. The ROS Hostname is needed for the master to know which computer a nodes is on and to assign URIs for the nodes. The ROS IP simply supplies the nodes with their IP address.

A.2.3 Scripts

Because environmental variables are one of the most common sources of connection issues if not setup properly, it is highly recommended to use the provided scripts. They get the IP address of the computer through the 'hostname -I' command. In order to launch both of the Turtlebot launch scripts in one shell scripts, the launch files are run as background processes.

A.2.4 Further Reading

- `http://wiki.ros.org/roscpp/Overview/NodeHandles`
- `http://wiki.ros.org/Names`
- `http://wiki.ros.org/roslaunch/XML/group`
- `http://wiki.ros.org/roslaunch`
- `http://wiki.ros.org/ROS/NetworkSetup`
- `http://wiki.ros.org/ROS/EnvironmentVariables`

A.3 Guide

A.3.1 Host Computer

This section describes how to run the HRI on the host computer (the machine used to control all the robots). It assumes that the user is in the catkin_ws folder of this project in a fresh terminal.

Manual start up procedure

1. Source base ROS files

```
$ source opt/ros/indigo/setup.bash
```

2. Source local ROS files

```
$ source devel/setup.bash
```

3. Configure the ROS environmental variables

```
$ export ROS_HOSTNAME=(Host Computer's IP address)
$ export ROS_MASTER_URI=http://(Host Computer's IP address):11311
$ export ROS_IP=(Host Computer's IP address)
```

4. Launch the application

```
$ roslaunch mhri mhri.launch
```

Script based start up procedure

1. Source base ROS files

```
$ source opt/ros/indigo/setup.bash
```

2. Source local ROS files

```
$ source devel/setup.bash
```

3. Start the application using the mrhi_run script in the host folder of utils.

```
$ ./.../utils/host/mhri_run.sh
```

A.3.2 Remote Computers (Robots)

This section explains how to start and connect a Turtlebot to the host computer. It assumes the user is connected to the Turtlebot either physically or through SSH. In order to connect multiple robots, repeat the below steps but use a different identifier for each bot.

Note: The HRI must be running on the host computer before a connection to a robot is possible.

Manual start up procedure

1. ROS environmental variables

```
$ export ROS_HOSTNAME=(Turtlebot's IP address)
$ export ROS_MASTER_URI=http://(Host Computer's IP address):11311
$ export ROS_IP=(Turtlebot's IP address)
```

2. Navigate to the Turtlebot's launch files

```
$ roscd turtlebot_bringup/launch
```

3. Check to see if the custom launch files are present. You should see mhri.launch and mhri_sensor.launch.

```
$ ls
```

4. If the two custom launch files are present proceed to the next item else copy the two launch files from Human-Robot-Interface/utils/remote to the folder.

5. Launch minimal launch

```
$ roslaunch turtlebot_bringup mhri.launch name:=(Robot Identifier)
```

6. In a separate terminal set the environmental variables as you did in step one and the launch the camera using the following command

```
$ roslaunch turtlebot_bringup mhri_sensor.launch name:=(Robot Identifier)
```

Script based start up procedure

1. Navigate to the Turtlebot's launch files

   ```
   $ roscd turtlebot_bringup/launch
   ```

2. Check to see if the custom launch files along with the script are present. You should see mhri.launch, mhri_sensor.launch and turtle_remote.sh.

   ```
   $ ls
   ```

3. If the two custom launch files and the script are present proceeded to the next item else copy files from Human-Robot-Interface/utils/remote to the folder.

4. Launch the script

   ```
   $ ./turtle_remote.sh (Host computer's IP address) (Robot Identifier)
   ```

A.3.3 Adding Bots To Interface

After following all previous steps to launch the interface and connect to Turtlebots, the user has two options for adding the robots to the interface. Pressing the "Look for Connected Bots" button will automatically add all connected bots when there is no text entered into the field labeled "Enter Robot Name". The user can also manually add a robot by typing the unique identifier in the text box and pressing "Add Bot". If they do not appear or there are issues controlling the robots after they have been added, double check the steps above to ensure the environmental variables are set up correctly. If the problem persists, refer to the troubleshooting documentation.

A.4 Script documentation

This section is separated into two parts: scripts for the host computer and those for the remote computers. Below each description is the command used to invoke the script.

A.4.1 Host

These scripts are designed to be run on the Host computer.

connect_turtlebot.sh

This script will ssh into the given Turtlebot and run the two launch scripts. On connecting the user will be prompted to enter the password for the Turtlebot.

```
$ ./connect_turtlebot.sh (User name on Robot) (Robot IP address) (Robot Identifier)
```

mrhi_run.sh

This script will set the ROS environmental variables using the IP address returned by 'hostname -I' before launching the application.

```
$ ./mrhi_run.sh
```

turtlebot_deploy.sh

This script will automatically transfer all the files in the remote folder over scp to the Turtlebot. Sadly due to to where the files are stored it requires entering the password three times.

```
$ ./turtlebot_deploy.sh (User name on Robot)@(Robot IP address)
```

A.4.2 Remote

These scripts are designed to be run on the Turtlebots either in person or over SSH.

mhri.launch

Custom launch file which launches minimal.launch under the group passed in by name:= and also launches the velocity smoother.

```
$ roslaunch turtlebot_bringup mhri.launch name:=(Robot Identifier)
```

mhri_sensor.launch

Custom launch file which launches 3dsensor.launch under the group passed in by name:=

```
$ roslaunch turtlebot_bringup mhri_sensor.launch name:=(Robot Identifier)
```

turtle_remote.sh

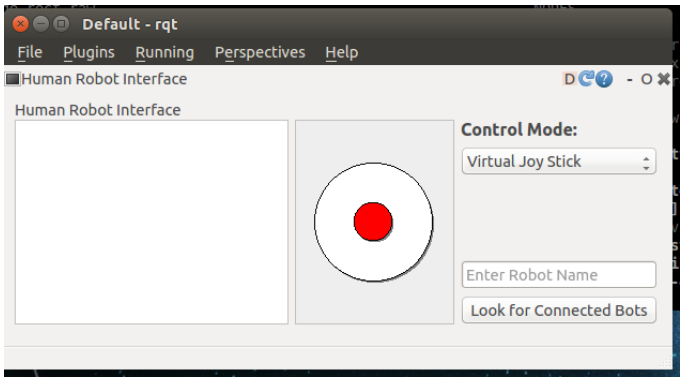
This script will set the ROS environmental variables using the IP address returned by ‘hostname -I‘ and the passed in host IP address before calling both mhri.launch and mhri_sensor.launch and passing the robot identifier to each launch file.

```
$ ./turtle_remote.sh (Host computer IP address) (Robot Identifier)
```

APPENDIX B

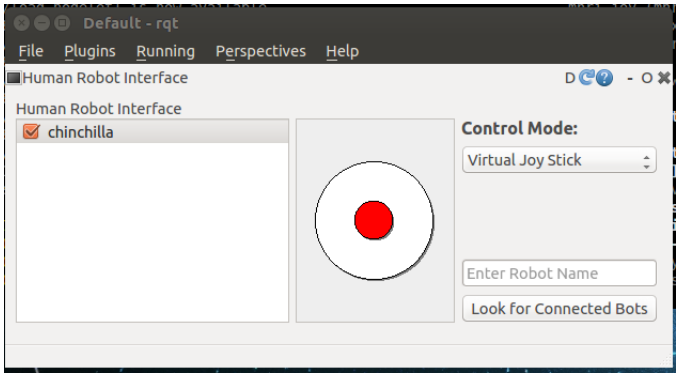
GUI Overview

B.1 Main Interface



The design of the interface was kept minimal for ease of use. On the left hand side of the screen is a list that displays all the robots currently connected to the interface. By hitting the "Look for Connected Bots button the all connected bots will be added to list if not already there. To add bots manually, the user can type the name of the bot into the "Enter Robot Name" text box and press "Add Bot".

Note: The robot's name cannot contain any special characters. The robot name text edit field will clear and will fail at adding the robot properly.

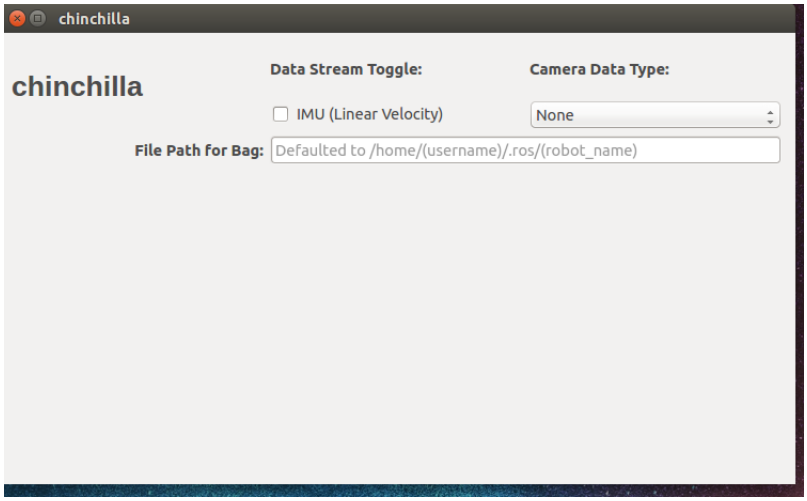


After adding the robot to the list the user can select or deselect the robots using the checkbox next to the robot's name. Selecting the checkbox on any robot will bring up another window for that particular robot and will allow users to control all selected robots through the selected mode control type.

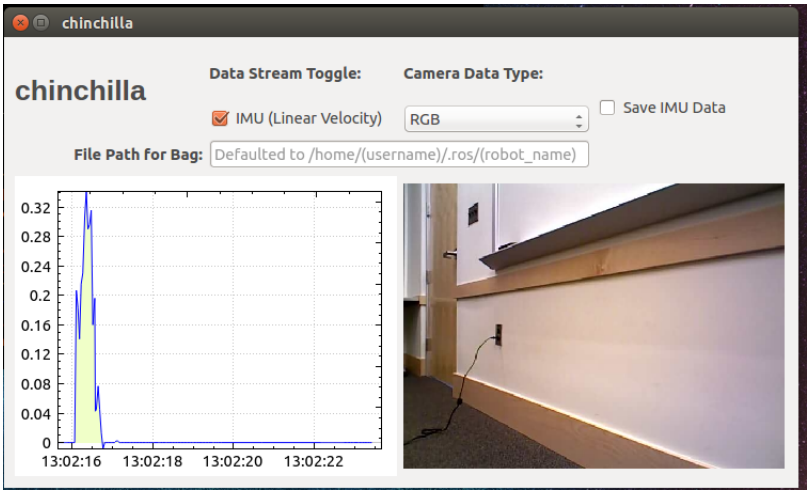
To the right of the robot list, there is a drop down menu that allows users to choose the mode control type (currently, only virtual joystick and PS3 controller are functional). The virtual joystick is the default mode and allows non-rigid control so that rotation and movement can occur simultaneously.

Note: If multiple robots are checked at the same time they will move the same direction and speed.

B.2 Bot Window Interface



When a robot is checked for the first time, a separate window appears for it. If the window is closed, the window will reopen if the robot is checked again or highlighted if already checked. Highlighting a robot in the list makes its window come into focus. Users can still have bot windows open even if the robot is not selected for controlling. Each bot window will vary depending on the type of robot. Currently, only Turtlebots are supported. The Turtlebot window includes the identifier of the robot in the top right hand corner. To the right of the robot identifier are several toggles. The user can toggle IMU data streaming (the graph displaying linear velocity) as well as camera streaming. There are two supported camera views: RGB and depth. Only one can be streamed at a time. If the IMU data streaming is on, the user also has the option of saving this data and giving the file a specific location and name. The IMU data is saved in a rosbag file and defaults to /home/(username)/.ros/(robot identifier). Once the "Save IMU Data" box is checked, the data will begin to save and will terminate once unchecked.



APPENDIX C

Troubleshooting

Below are some general tips to help troubleshoot any issues as well as some common questions.

- Reboot the robot. This almost always helps, no matter the issue (especially sensors not working or being unable to connect and control the robot).
- Check that all of the environmental variables are set on ALL machines (host and remote). See question 5 below and startup documentation.
- Make sure to source properly. Refer to the startup guide for more detail.
- Ensure all machines are connected to the same network (CSMWireless is different than CSMGuest).

1. **What if the sensors are not working?**

   Look at the verbose output on the terminal window used to launch the robot (either through ssh or on the robot). If you see something similar to "Device not found", try rebooting the computer.

   The Asus XTion cameras on Turtlebots must be plugged into USB 2.0, not 3.0 ports. This might be another cause.

   On the host computer, run `$ rostopic list` in a fresh terminal. You should see robot_name/odom/ and robot_name/camera/{depth_registered, rgb}/image_raw for the IMU and camera, respectively. If not seen, they are not connected. Try rebooting the robot and reconnecting the robot.

2. **I can't automatically find or connect to the Turtlebot manually?**

   If you are unable to find the Turtlebot or connect manually ensure that both you and the Turtlebot are connected to the same network. (CSM wireless and CSM guest are different)

3. **I got an error that I cannot find the launch files when attempting to run the interface?**

   Check that you have set the sources for the workspace and launch files correctly. Run the commands `$ source devel/setup.bash && source /opt/ros/indigo/setup.bash` in the catkin_ws folder. This must be done in every new terminal before launching.

4. **I am getting unclear errors what should I check?**

If you are getting unclear error messages when running you may not have set your environmental variables correctly. See the below question to check if they are all correctly set. Remember, each variable is set by running the following:

- `$ export ROS_IP = (computer's IP)`
- `$ export ROS_HOSTNAME = (computer's IP)`
- `$ export ROS_MASTER_URI = (computer running interface IP)`

Note: environmental variables are not permanent, they are strictly local to the terminal window they were set in

5. **How do I find my ROS environmental variables?**

   Run the command `$ printenv | grep ROS` to list all environmental variables related to ROS.

6. **The code won't compile?**

   In the root of catkin_ws, run the following commands

   - `$ rm -rf build/ src/CMakeLists.txt`
   - `$ cd src`
   - `$ catkin_init_workspace`
   - `$ cd ..`

   Then recompile.