Turtlebot Control in the DIARC Robotic Architecture

CSM Williams
Duncan Thomas, Joseph Kim,
Evan Lim, Matt Clough

June 18, 2019

Introduction

Our client is Dr. Tom Williams, an assistant professor at the School of Mines and the director of the Mines Interactive Robotics Research Lab (MIRRORLab). The MIRRORLab, located in Brown Building room 339 at Mines, develops intelligent systems to study human-robot interaction (HRI).

The project we were given was to integrate the Kobuki *Turtlebot 2* into the *Distributed, Integrated, Affect, Reflection, Cognition* (DIARC) theoretical architecture to enable control via natural language commands and create a Wizard-of-Oz (WoZ) interface to allow manual remote control of the robot.

One way to control the *Turtlebot's* movement would be through the Robot Operating System (ROS). ROS is a collection of libraries and tools that have been developed over time to create a robust operating system for a wide variety of robots. This is how the MIRRORLab has been using the *Turtlebot* up to this point.

For their HRI experiments and exploration research, the MIRRORLab uses the Agent Development Environment (ADE), an implementation of DIARC. The main points of DIARC are emotion recognition and expression, natural language understanding, action execution, perceptual learning, and the integration challenge of developing all of those points at the same time. ADE provides a way for the concepts of DIARC to be tested in the real world.

For our project, one way that the *Turtlebot* could be interfaced into the ADE architecture would be to have ADE and the ROS run in parallel when running experiments. Both architectures would need to be used because ADE would parse human speech commands and then ROS would consequently move the *Turtlebot*. However, for experiments with the *Turtlebot*, only movement control would be necessary. A large amount of ROS's functionality would be unused and simply add bulk to the product. To get around using ROS at all, we needed to interface the *Turtlebot's* C++ driver library directly with the ADE Java library.

Requirements

Functionally, the drivers for the Kobuki *Turtlebot 2*, which are in C++, needed to be wrapped to make them accessible from Java. To manually control the robot, a GUI needed to be developed in Java; more specifically, the GUI must connect to a webcam on the robot and be able to send movement commands through a set of buttons or keyboard input. Lastly, the *Turtlebot* drivers and the GUI must be worked into ADE for use by the MIRRORLab.

Functional Requirements :

- Be able to call C++ functions in Java
- Directly interface Kobuki driver functions with ADE architecture
- Robot should be able to run with a controller GUI or through speech commands
- GUI should be modular to only display relevant controls for currently-running robot

Non-functional Requirements :

- GUI should be user friendly and intuitive
- GUI should look elegant (stretch goal)
- The project needs to be built using ANT

System Architecture

Figure 1 below is a representation of how information flows through the system. At the start we have the operator, which is our Wizard of Oz, a term we use to describe the remote nature of controlling the robot. The operator gives input to the GUI to parts on the robot and receives the feed from a webcam on the robot to control it without having to see the robot itself. The GUI has a class that extends ADE's Component class in order to communicate with the rest of the architecture, sends the operator's commands into the architecture where other components complete the actions, like the TurtlebotComponent that had to be made. The TurtlebotComponent calls methods from the Java-wrapped C++ code to tell the *TurtleBot 2* what to do. The *TurtleBot* should also be able to be controlled through speech commands, this is accomplished by multiple components in the DIARC Architecture that all together eventually call the wrapped methods as well.

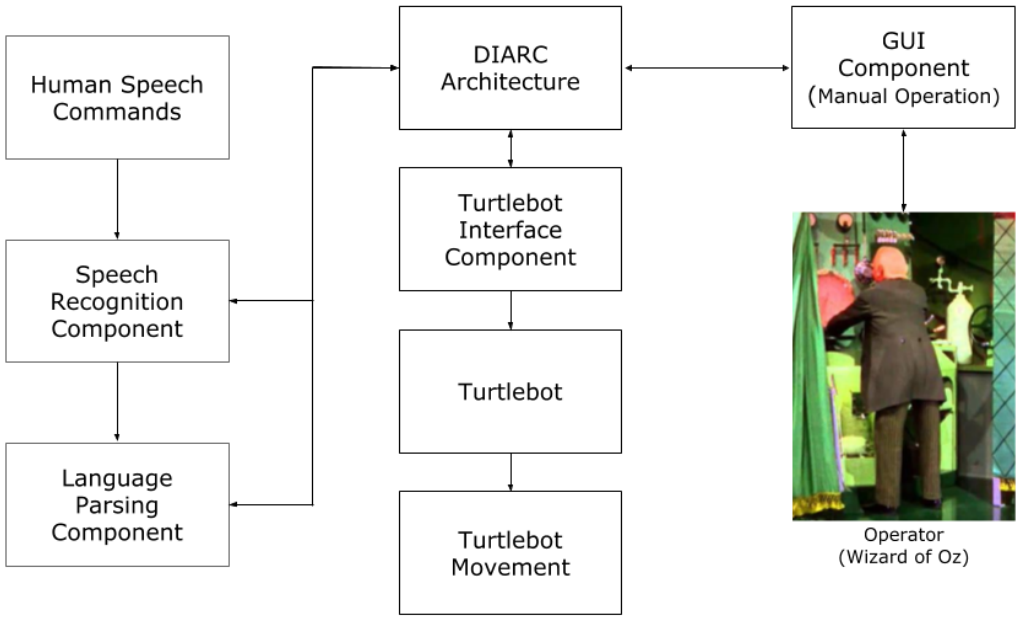

Figure 1 : System Design

ADE components each operate in parallel, send messages to each other, have their own update loop, and are all connected to a central registry component. This registry component is how all other individual components communicate with each other. *Figure 2* is a window from the ADE registry showing the connections between the individual components running for our system. The registry is represented by the blue component in the middle, with all of the other components stemming from it. The components we created, the *TurtlebotComponent* and *WizardGUIComponent*, are the two rightmost components; all of the other components besides the GoalManager component are required for speech commands.

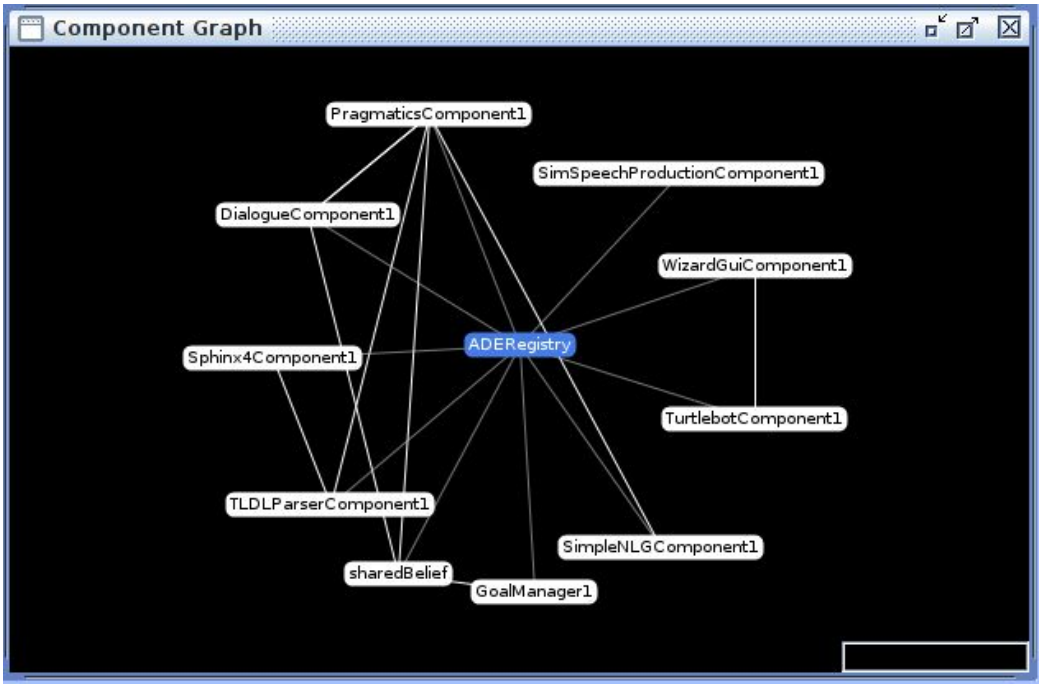

Figure 2 : ADE Registry Component Graph

The design of the GUI is shown in *Figure 3*. It is broken into three different classes: individual modules, a container for those modules, and the display for those modules. Each module has its own specific functionality, whether that be for a camera stream for an operator to see what a robot might see or for an operator to control the basic movements of a robot. Modules like these are then gathered into the Parameters class to let the GridPane know which modules it should load and show the operator.

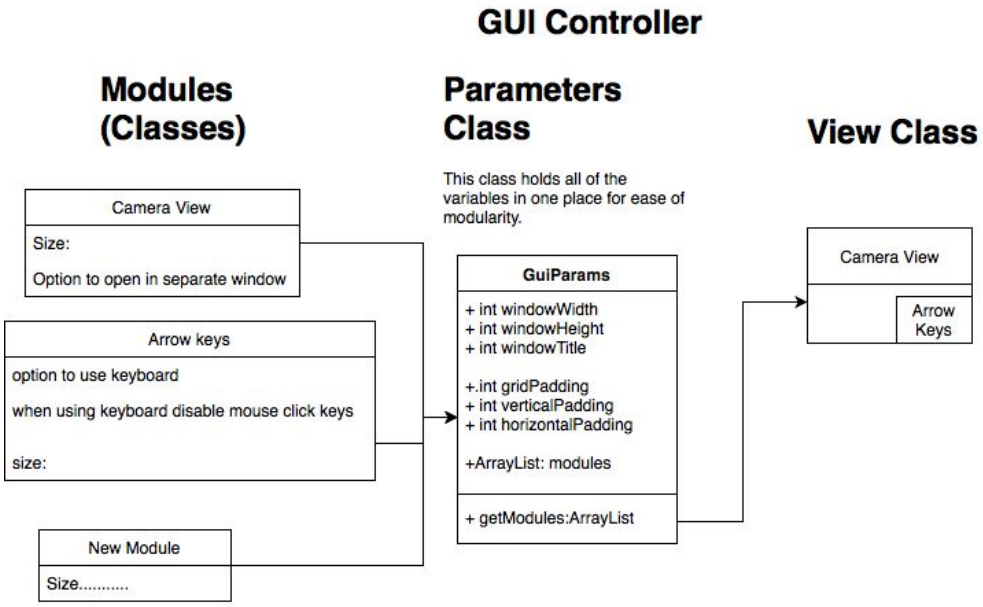

Figure 3 : GUI Controller Design

Technical Design

As mentioned in the introduction, even though ROS has a lot of functionality built into its library, we only needed movement control for the *Turtlebot 2* from ROS. Using ROS for our project would have lead to inefficiency for the runtime of our TurtlebotComponent within ADE because both ROS and ADE would have to run in parallel. Instead, we chose to directly interface the C++ driver library for the *Turtlebot* with a new ADE component to integrate the bot into ADE. There are a few options for how to do this interfacing between Java code and C++ code, such as the Simplified Wrapper and Interface Generator (SWIG) and the Java Native Interface (JNI).

Both of these options would allow the TurtlebotComponent in Java to call C++ functions from the driver library. SWIG actually uses JNI for the core functionality of wrapping the C++ code and is supposed to automate tedious parts of the process of using JNI. However, we could not get SWIG to work with the driver library and we instead explored using JNI itself. *Figure 4* is a diagram detailing how JNI provides a connection between the native C++ code and the JRE running Java code. We wrote the Java interface code that provides the Java code with a pointer to a C++ object so that the Java code can call C++ functions through these objects.

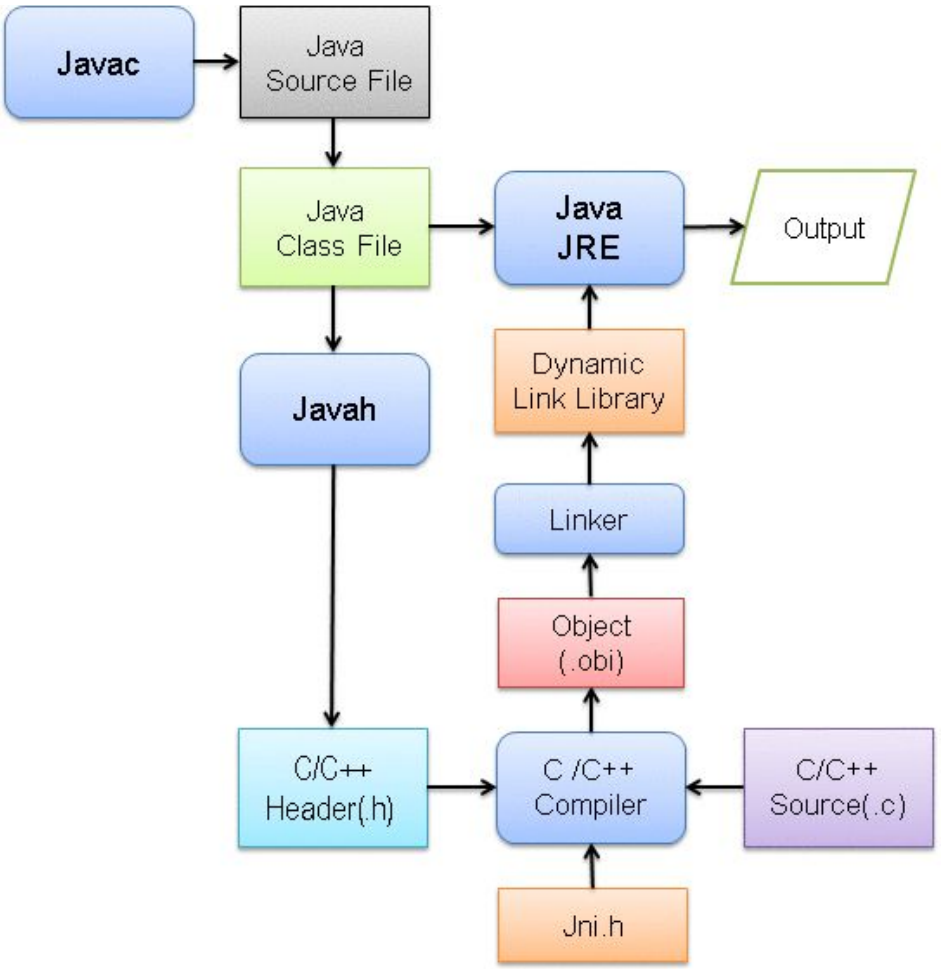

Figure 4 : JNI Block Diagram
(eQuestionAnswers : How to access a C function from Java?)

While wrapping up development and moving onto product deployment, we realized that our system design at that point was with everything running on one computer. With a moving robot, connected by a cable, operation was awkward, and the WoZ interface was irrelevant. If the operator was tethered to the *Turtlebot* by a short cable, they would have to move with the robot and would already be able to see what the robot sees; we needed to figure out a way to fully operate the robot wirelessly. The first part of this issue we focused on was the manual operation of the *Turtlebot* through the use of the GUI.

From ADE, a solution presented itself. Each component is essentially independent, meaning a complete system can be run across multiple machines. We thought about launching the GUI on a computer not connected to the robot, but we were unable to access the view from the camera on the robot in this fashion. We believe that the optimal solution would be to send the video stream from the computer on the robot to the operator's computer. However, due to time constraints, we were not able to explore this option. We instead opted to remotely view the other computer's desktop using a program, like *VNC Viewer*, to use the GUI from the other computer. This allowed for wireless manual control, but voice commands could not be transmitted through the desktop viewer using the operator's microphone; they would still need to go through the computer on the robot's microphone.

We were able to get around this by running the speech recognition component on the operator's computer, which would then send its understanding of vocalizations over the network to the rest of the components. These issues and their solutions resulted in our current operation design, which can be seen in *Figure 5*: all components but the speech recognition component are run on a computer connected to the robot while the operator's computer runs the speech recognition component and remote into the robot-connected computer to use the GUI.

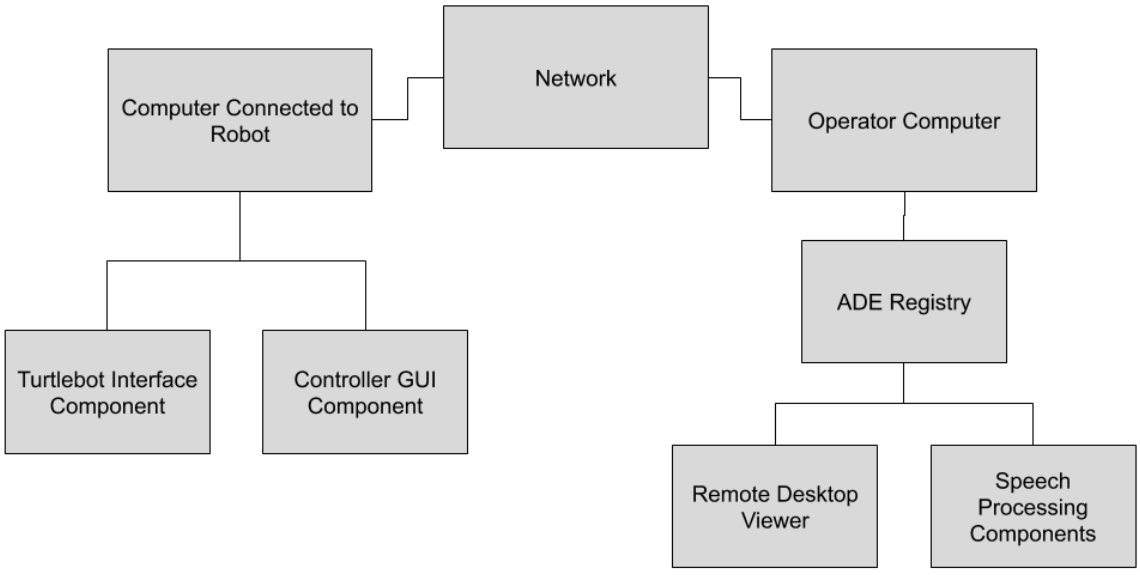

Figure 5 : Operation Design

QA

To ensure that our project is not a black box, we provided adequate documentation to explain different aspects of set up, execution, and extension. As we are developing with JNI, a framework that none of us were familiar with, we needed to make sure that any one else could pick up from where we left off, regardless of their familiarity with JNI. Regular commenting was done to detail how a piece of code works and what it achieves. This should serve as a small learning tool if someone wanted to modify our code or expand on it.

A larger learning tool is provided within the MIRRORLab's GitLab repository wiki page. We added pages to the wiki that details the steps necessary to run our system. If we do not provide adequate documentation, our project does not have an impact outside of its own functionality. Explanations on lower-level details, like JNI, and architectural details like build paths and config files allows for future members to apply our groundwork to other projects.

We used Git as version control, and before code is committed we review it as a team. This helps to reduce the frequency of bugs. In the GUI, because of the modular development, we were able to create modules and review individual modules. Each module has its own container and doesn't affect the other modules so we only needed to review the current module. In this way, we are able to review specific pieces of code.

Working on the interface most of the code review happened over multiple ways to implement the system. We had a lot of trouble with this part of the project. To combat this the team was broken up into parallel working teams and individuals. Different members would explore different ways to solve the same issue to cut down on overhead switching time. When a solution was found to be likely the code was reviewed to determine if it would work before adding it to the repository.

In order to understand whether our product is easy to use we employed user testing. The code we received for the Kobuki drivers included a number of demos which needed to appear to function the same, and by rewriting and comparing them to the old demos we believe they are accurate to the original. Another example of user testing is when we had to use the GUI to test movement control of the *Turtlebot* and show our client this movement in action. After this, we received feedback from Dr. Williams and made improvements based on his suggestions.

Results

Based on minimal requirements, the project can be considered a success. The *Turtlebot* is integrated into the ADE architecture and there is a controller GUI for robots in ADE, through which the operator can view a camera stream.

The *Turtlebot* is interfaced through the *TurtlebotComponent* we wrote and the component interfaces the C++ driver library with the Java ADE code. The *TurtlebotComponent* can also link up and communicate with other components in ADE like the speech recognition component for natural language interaction.

The GUI can be used to control multiple robots through the ADE architecture by connecting the controller to existing components like the *VelocityComponent*. An operator can also see in the GUI what the robot sees, if there is a camera for the robot, through a camera stream module.

Some features that we were not able to complete due to time constraints are a module in the GUI to change the color of the LED's on the Turtlebot, a module to display the battery power of the robot, and getting data from the bumper sensor on the robot. When we started this project we expected it to be rather simple on what we needed to do. While they were, we spent a lot of time on chasing an avenue that ended up not panning out for interfacing C++ code with Java code. In our stubbornness to get that method to work, we didn't ask questions until three weeks into the field session. If we had asked sooner, we could have gotten more things done. From that, we learned to ask more questions, and ask them soon, to keep us from getting stuck.