Felix, A Rich Learning Experience in Mobile Robotics

Andrew Tu

***Abstract*—Felix was a mobile robotics platform designed to serve drinks at CES 2019. Despite it's early termination, Felix provided a rich learning opportunity on how to design maintainable robotic systems. The breadth of software and system admin related components of this project was non-trivial. Software was written in six different programming languages, to run on seven unique pieces of hardware and interfacing with ten types of I/O and sensors.**

In this paper, we discuss our architecture, our development workflow, and the lessons we learned.

I. Introduction

The goal of Felix was to develop an autonomous mobile robot capable of serving drinks at CES 2019. This project showcased Flex's ability to develop a mobile robot and various HMI capabilities. Felix is filled at the bar by the operator/bartender before being set to "serve". Felix then begins autonomously navigating the showcase floor, serving drinks. Felix stops when people reach out to take a drink, or call out the wakeup phrase "Hello Blue Genie" [1]. Once all the drinks have been taken, Felix returns home to the bartender to be refilled and sent back out. Along the way, Felix collects data about where in the room drinks are served. From this data, a heatmap is built and used to inform "smart" serving behaviors (e.g. serving "hotter" or "cooler" areas, exploring new places, etc.).

Felix is a roughly $3.5'$ tall, 27" diameter robot with a differential wheel drive. A 360° planar 3D laser scanner (LIDAR) supplemented by eight sonar sensors arranged across the top and bottom of the front face of the robot provide input about the environment. The sensor data ultimately fed into the Slamware Core: a processing module from a Chinese based robotics company. On the top, Felix has a drink tray with "cup holder" slots for up to nine drinks. Each slot contains a force sensitive resistor sensor (FSR) used to identify the presence of a drink. LED strips are used around the top of the robot to give visual feedback to the user and operator. On each side of the tray, IR sensors are used to detect user interactions. Felix uses four directional microphones attached to an Amlogic board to detect the wake up phrase "Hello Blue Genie". Felix's front face, is an Android based operator tablet. Felix can also be controlled remotely using the same Android based application on a remote device.

The project was ultimately cancelled during the final stages of testing and development despite the core functionality of the robot having been completed. The project team was comprised of roughly 15 software, electrical and mechanical engineers. The project started in June/July 2018 and was scheduled to be completed by the end of December of the same year. In mid September, the project deadline was moved up to November 30th with the first fully functioning robot due on October 31st before the project was completely canceled on October 19th. The total budget of the project was on the order of $\$500,000$.

Fig. 1: A rendered image of Felix without the fabric exterior.

This purpose of this paper is to give a high level overview of the hardware and software architecture of Felix, to explain project design choices and methodologies, to describe and analyze the project workflow, and to solidify lessons learned while working on this project.

II. Architecture

We can break down the primary pieces of hardware in our project into two categories: peripherals and processors. The peripherals usually serve a single purpose of producing data to be read and utilized by the system. Additionally, they may be used to output information to the user. The processors in this system are the components directly controlling the peripherals. Figure 2 displays an overview of our hardware stack.

A. Tu, is with the Flex Innovation and Design Lab in Milpitas, CA, USA. Corresponding authors e-mail: tu.a+felix@husky.neu.edu.

[1]A wake up phrase or wake up word (WuW) is a word used to trigger an action in a system. Common examples of WuW's are "Hey Siri", "Hello Google", and "Alexa"

A large portion of our software stack is built on using the Robotic Operating System (ROS) framework. ROS is a widely used open source collection of tools and libraries that abstract and greatly simplify developing software of robotic systems. For more information regarding ROS and its capabilities, see [1].

In this Section, we discuss the purpose of each hardware component and how it connects to the neighboring systems using a top down approach.

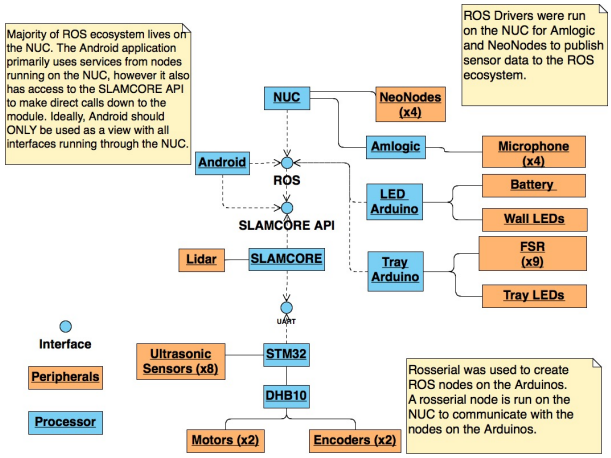

Fig. 2: An overview of the different hardware components in our system.

A. Amlogic

The Amlogic board is responsible for directional wake-up-word (WuW) detection from the microphones. When the WuW is detected, message is sent over a serial connection to the NUC where a node is running to publish the message to the ROS ecosystem.

B. LED Arduino

Despite its name, the LED Arduino serves a dual purpose: LED actuation and battery readings. This Arduino was initially meant to only drive the LEDs but we eventually realized it had extra ports available to also read the voltage from the batteries. The Arduino is connected over serial via USB to the NUC. The Arduino runs a ROS node that connects to a corresponding serial node "server" on the NUC that bridges the serial connection to the ROS ecosystem.

C. Tray Arduino

The tray Arduino is set up similar to the led Arduino, the code on the Arduino written as a ROS serial client and a corresponding server on the NUC. The nine FSR sensors and tray LEDs are fed into the tray Arduino. All processing regarding the tray sensing, computation and actuation is done locally on the Arduino. Tray state information is published on every state change through the ROS channel. The tray receives reset commands through the reset tray topic. Processing on the Arduinos posed a number of challenges towards the system development. These challenges are discussed in-depth in section V-C

D. Android

The Android component of this system served a number of purposes including viewing state information and sending control commands. The tablet was connected over ethernet/wireless to both the ROS ecosystem and also directly to the SLAMCORE API. Throughout the development process, the Android component presented significant challenges to the organizational structure and stability of the system. These difficulties are discussed in-depth in section V-B.

E. NUC

We used an Intel NUC to serve as the linux host connecting large parts of the system together. The NUC was chosen specifically for its small form factor. A number of peripherals were connected over USB to the NUC including both Arduinos, all four IR sensors, and the Amlogic board. The NUC connected to the Slamcore wirelessly or via ethernet.The NUC has a 7th Gen i5 processor with 8GB of RAM and a 256 GB of flash storage.

The NUC was running Ubuntu 16.04 with a full ROS Kinetic Desktop installation. In Felix's configuration, the NUC both served as ROS master as well as running an number of the nodes onboard.

F. SLAMCORE

Felix's navigation system is built around Slamtec's Slamware Core reference board: the sdp mini. The sdp mini comes with a breakout board, connecting the Slamware Core (Slamcore) to an STM32 micro-controller. The kit also comes with Slamtec's RPLIDAR A2, a planar 360° LIDAR, and out of box support for up to 4 sonar sensors.

Slamcore communicates down to the STMicro (STM32) over a serial connection using the Slamtec's Ctrl Bus Communication Protocol. The Slamcore module is configured using the RoboStudio GUI tool which produces a C configuration file. This file is compiled on the STM32 and given to Slamcore at runtime.

Slamcore itself is a proprietary system, taking in data from the lidar to build a map of the environment. **Sonar sensors are used for obstacle detection but not for map building.** The Slamware core exposes an API for control, giving developers the ability to plot navigation points, draw virtual walls, access state data, etc.

Given a point, to navigate to, Slamware core will use a D* algorithm to calculate a path and then create velocity commands (in the form of $(motor1_{vel}, motor2_{vel})$) to drive the robot. Odometry along the X and Y is calculated from the wheel encoders. Rotational information is generated by the onboard IMU.

While at the surface Slamtec appears to present an all encompassing solution, working with the system posed a number of serious issues. These issues are discussed in depth in section V-D.

G. STM32

Slamtec's sdp mini reference solution uses two chips to drive the system. While Slamcore is responsible for navigation

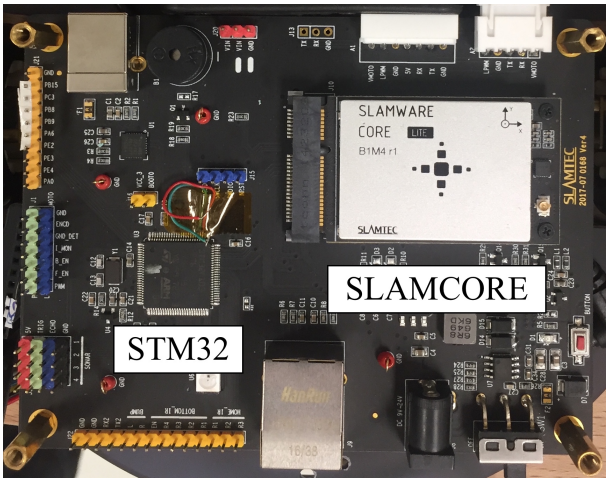

Fig. 3: The Slamtec sdp mini board containing the SLAMCORE module and the STM32.

and routing, the STM is responsible for driving the motors, pulling in sensor data, and providing it to the Slamware core. Because we were transferring the drive train from the sdp minis small PWM driven motors to a larger motor controller driven by the DHB10 board, large modifications were made to the STM code. The STM32 communicates with the DHB10 motor controller board over a serial connection. To increase the visibility of our environment, we doubled the number of ultrasonic sensors requiring existing pinnouts to be remapped for use by the sonar sensors.

H. DHB10

The DHB10 is the motor controller used as part of the Arlo base drive system. The DHB10 accepts a set of serial commands that can be used to set motor speeds, access and clear odometry information, and even reboot the system (although we had to modify the stock DHB10 motor controller firmware to add a soft reboot functionality). The code from the motor controller is written in SPIN, a high level object based language designed specifically by Parallax for the Propeller chipset [2].

III. Software Methodologies

The structure of a well designed software architecture should adhere to a set of design methodologies that maximize its performance and portability without sacrificing the productivity of the developers. These philosophies give rise to a number of design patterns that are used to solve known problems, allowing code to be reused from application to application.

The value of design patterns is succinctly described by the Gang of Four in the introduction to their seminal work "Design patterns: Elements of Reusable Object-Oriented Software". *"Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reusable and avoid alternatives that compromise reusability. Design patterns can even improve the documentation and maintenance of existing systems by furnishing an explicit specification of class and object interactions and their underlying intent. Put simply, design patterns help a designer get a design 'right" faster'."*[3]

Code that is written, but not designed (notoriously known as spaghetti code), can pose a danger to the rest of the system. This code often proves difficult to maintain or extend by the original developer(s), and simply impossible to onboard new engineers into the code base. Furthermore, bugs prove incredibly difficult to track down and fix. In the worst case scenario, a single bug can snowball into a multitude of crashes, bringing down the entire system, spaghetti code or otherwise.

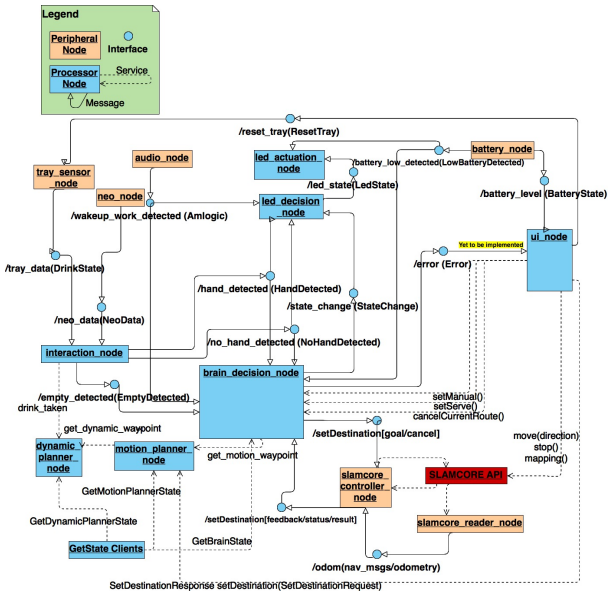

Fig. 4: An overview of the ROS node infrastructure. Dotted lines represent services with the client pointing to the server. Circles represent topics with arrows coming from the publisher and pointing to the subscriber.

Within Felix, we tried to adhere as closely as possible to the following design philosophies. Many of these ideas were strongly influenced by the ROS Best Practices [4] and existing large scale ROS projects like [5] and [6].

- Follow the "See, Think, Act" paradigm
- Keep nodes small and independent
- One node per sensor
- Maximize code reuse, nodes should be specialized at construction
- Maximize parameterization through global server
- Opt for Python when possible

For each methodology, we give a description, and a reasoning why this was important to our architecture.

A. See, Think, Act

The "See, Think, Act" paradigm is commonly recommended by the autonomous vehicle community. Nodes are broken up into a pipelined system with each node handling a specific processing stage. The "See" stage represents the phase of data collection, reading the data directly from the

sensors and publishing it raw to the rest of the system. The "Think" stage represents all of the processing done on the raw data. There can be any number of nodes within this stage depending on the complexity of the processing. Ultimately, a decision is made and a command is passed down to the "Act" phase. In the Act phase, control commands are executed, e.g. wheels turned, arms raised, lights turned on, etc.

Dividing nodes based on these distinct phases allows an individual node to be replaced to upgrade a specific functionality. Changed the motor controller? Only update the corresponding actuation node. New paper on more efficient processing? Only update the corresponding computation node(s).

In our architecture, we use this pattern repeatedly.

- Multiple Sensor Nodes → Interaction Node → LED Decision Node → LED Actuation Node
- Multiple Sensor Nodes → Interaction Node → Brain Decision Node → Slamcore Controller
- UI Node → Motion Planner → Brain Decision Node → Slamcore Controller
- etc.

The clearest example in the breakdown of nodes is the LED pipeline. Multiple sensor nodes fed into the interaction node where some level of computation is done. The interaction node publishes useful, actionable events, that are subscribed to by the next level of computation nodes. One of these nodes is the LED decision node which takes in meaningful event data (from the brain nodes, audio nodes, interaction node) and produces an actionable command, i.e. set the state of each LED to a given color. That command is received by the actuation node whose sole job is to set the attached LEDs to the given list of colors.

B. Small Independent Nodes

> Brevity is the soul of wit
>
> *Shakespeare, Hamlet*

Nodes should be written as concisely as possible. When the functionality of a single node begins to stretch past tolerable limits or a single function contains four layers of nested if statements, a refactoring is in order. While there are not hard limits on lines, no one who has to read and try to understand a 1000+ line class is going to have a good day. Not only does a refactor improve the readability of the code, it also heavily promotes code reuse.

Perhaps the best example of how using a modular structure improves code reuse is the motion planning dataflow. With this flow, the motion planner chooses to serve destination waypoints from either a list of static waypoints or pull from the dynamic planner. In practice, static waypoints are waypoints that were manually added while dynamic waypoints are auto-generated. For this iteration of Felix, the dynamic waypoint generation was based on the heatmap of drinks taken, however future applications may choose to generate waypoints through other means: random choice, least recently visited, vision based estimates, etc. Other strategies for dynamic waypoint generation can be substituted in for the current method, as long as it provides the appropriate `get_dynamic_waypoint` service. Furthermore, with small modifications, to the existing motion planner, a full strategy pattern [7] approach to selecting dynamic waypoints can be implemented. A proposed architecture is shown in fig 5.

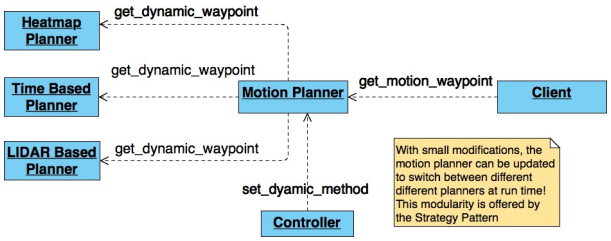

Fig. 5: A proposed motion planner pipeline using the strategy pattern to change how dynamic waypoints are served.

Special attention should be made when when developing nodes that are running as subprocess of another program, i.e. a node that is a member variable to another class. It is easy to let nodes begin pushing or pulling information to the "global scope" of the program, outside of the established interfaces. While this may get the program to compile in the short term, there are serious long term ramifications to the stability, maintainability, and debugging of the system.

C. One Node Per Sensor & Specialize in the Constructor

These two points are closely related and align with the idea of small independent nodes. The "One Node Per Sensor" practice dictates that a new node should be started for each hardware interface connected to a computer. This allows the node written to remain as simple as possible, focusing on performing a single job: publishing data from a single sensor. This idea was strongly influenced by [8].

This practices goes hand in hand with the second practice of specializing the node in the constructor. Instead of hard coding a specific node to an interface, the desired interface should be passed in as a parameter to the executable and used to construct the node. This allows the same executable to be used to start multiple instances of the code, connected to different sensors.

The primary example of this practice in action is with the NeoNodes. A single NeoNode ROS node is written to publish data on a given hardware interface. The hardware is differentiated by specifying the interface to use within the launch file.

Listing 1: NeoNode Launch Configuration

```
<node name="neonode_front" pkg="
    flex_felix_sensing" type="neo_node">
        <remap from="~node_id" to="/
            neonode_id_mapping/front" />
</node>

 <node name="neonode_right" pkg="
     flex_felix_sensing" type="neo_node">
     <remap from="~node_id" to="/
        neonode_id_mapping/right" />
</node>
```

This snippet from the launch file shows two `neo_node` nodes being started with a private parameter `~node_id` being set from another global variable (use of the global param server is discussed in section III-D). Within the node constructor, a different interface is selected corresponding to the node id. This logic is handled by the NeoNode library.

We also more explicitly set the appropriate interfaces based on parameter values.

Listing 2: Sensing Launch Configuration

```
<node name="slamcore_reader" pkg="
    flex_felix_sensing" type="
    slamcore_reader_node">
     <remap from="~ip" to="/slamtec/ip" />
     <remap from="~port" to="/slamtec/port
         " />
</node>

<node pkg="flex_felix_sensing" name="
    amlogic" type="audio_node">
     <remap from="~iface" to="/amlogic/
         iface" />
     <remap from="~baud" to="/amlogic/baud
         " />
</node>
```

In this example, the explicit ip, port, interface, and baud rate are specified for the appropriate nodes. These parameter values are all set in the configuration YAML file that's loaded in at the top level launch file.

D. Using the Global Parameter Server

The global parameter server is a server responsible for setting, storing, and providing state variables across the system. We mainly used the server to provide nodes with system level constants. While the server can be used to pass information back and forth between nodes, it should not be used for high throughput applications [9] [10].

We preload the values in the server by loading a YAML configuration in the launch file of the system. We eventually split variables into two launch files, one of global constants that were system agnostic (colors, timer values, states, number of LEDS, etc.), and one that was system dependent (ip addresses, ports, interface names, etc.).

By shifting all constants to the global parameter server, and having all constants loaded in from one of two files, we could ensure our entire system could be configured in a single place. This significantly reduces the amount of time we needed to spend updating code, instead allowing us to change the configuration file at launch time.

E. Opt for Python When Possible

This was less of a philosophy and more of a development decision. We opted to shift as much of the code base as possible into Python since it's the easiest to code in, and has the fewest issues with dependencies. We handled dependencies using Python virtual environments and a requirements file. All of our Python code is written in Python 2.7 (supported by ROS Kinetic).

The parts of the code we could not write in Python were nodes who depended on a language specific SDK or needed to run on specific hardware. Any nodes running on the Android were naturally written in Java, nodes running on the Arduinos were written in C++. The NeoNode SDK and the Slamware SDK were both written in C++, so the nodes using those SDKs are were also written in C++. The only node that breaks from this convention is the Amlogic node which was written before this decision was made.

IV. Project Workflow

The breadth of software and system admin related components of this project was non-trivial. Software was written

- In **6 different programming languages**
  - C, C++, Python, Java, Bash, and a tiny bit of Spin
- To run on **7 unique pieces of hardware**
  - Amlogic, DHB10, STM32, NUC, FSR Arduino, LED Arduino
- Interfacing with **10 types of I/O and sensors**
  - Sonar, Lidar, Motors, Encoders, FSRs, Microphones, NeoNodes, Slamware Core, LEDs, Android Platform

One of the most challenge components of this project was understanding how to synchronize the development and testing of code by our engineering and testing teams. This problem was compounded by the fact that we had difficulty finding resources describing how a ROS based project should be divided at a node, package, and repo level.

Many of the practices we describe here were taken and scaled down from standard development practices in larger, software focused companies. While seemingly cumbersome, these practices should continued to be used in some capacity to promote efficient collaboration and maintainability of the code base.

A. Project Structure

Felix consists of three repos: Felix, Felix_Common, and Felix_AD. Felix_AD contains all code needed to compile and run the Android application. Felix contains the rest of the code running on the actual robot: everything including the firmware of the DBH10, STM32, ROS Stack running on the NUC and Arduino units. Felix also contains useful scripts that were written for testing and analysis including a script that translates the Chinese source comments to English, provides a Matplotlib dashboard analysis of sonar sensors, and battery data. An overview of the ROS part of the system is shown in Fig. 6

The flex_felix_common package in the Felix_Common repo holds all of the action, message, and service declarations. As a a result, all other felix packages depend on flex_felix_common. Consolidating these dependencies into a single package prevents issues of circular dependencies. The flex_felix_bringup package contains the system launch and configuration. Once the code has been built, all configurations should be able to be changed through modifications to the

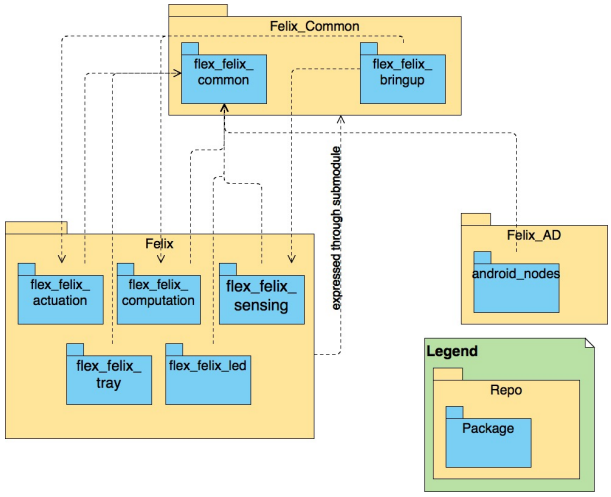

Fig. 6: An overview of the ROS Stack at a repo and package level. All ROS packages have a dependency on flex_felix_common because it defines all actions, services and messages

configuration and launch files. As the bringup package needs to access the executables built by the rest of Felix, it is dependent on all packages of the Felix repo. While this technically creates a circular dependency at the **repo** level, there is not a dependency at the **package** level.

Breaking the common dependencies out into their own repo allowed developers on other portions of the project, (namely developers working on Android) to only need to clone the Felix_Common repo as opposed to the entire Felix repo, i.e. downloading ~ 1000 lines of code instead of $\sim 308,000$ lines of code. Ideally, the Felix repo would have been split up further to pull out the non-ROS related firmware into its own repo, however we were given limitations on number of repos for this project.

Because of the complexity surrounding the on-boarding process, an installation script was written to handle installation of dependencies and configuration of environment. The `install.sh` script is on the root level of the Felix repo. The script handles everything from generating an appropriate environment file (sourced by running `$ source environ`, to installing project dependencies through pip and apt-get. The script even goes as far as installing ROS Kinetic if it detects ROS is not installed on the system. Once the installation is run, contributors will still need to source the generated environment file and enter the generated python virtual environment. A number of custom aliases are located in the environment file including commands for entering the appropriate environments and building the project.

B. Git Submodules

As shown in Fig. 6, a number of dependencies exist between packages living in different repositories; primarily a dependency of packages in all repositories on the flex_felix_common package in the Felix_Common repo. A major concern with having code split between different repositories is maintaining a level of synchronization between these dependent codes. If a message defined in Felix_Common is updated, how do you ensure that nodes in the Felix repo can still be worked on before the new version of the messages are applied? In essence, we need a method of capturing the history of Felix_Common, independent of any repos who depend on it. Furthermore, any repos who depend on Felix_Common should understand which **version** of Felix_Common they depend on.

Fortunately, git has support for this class of problem: submodules. A git submodule is a way of NESTING repositories within one another [11]. It offers its users a way of linking a specific commit of a given repository to the commit of a repository it depends on. While we initially struggled with working with this feature (pushing broken commit values, not synchronizing Felix_Common with remote properly, etc.), submodules eventually made it easier for us to maintain and synchronize the Felix code base. Early on, Felix_Common was added to Felix as a submodule, ensuring that each commit of Felix refers to a specific commit of Felix_Common. Whenever changes needed to be rolled back or a looked at again, we knew exactly which version of common to use.

The Felix_AD (Android) repo took a different approach. Instead of creating a submodule, a separate version of the Felix_Common was downloaded and used to compile a .jar file with the necessary ROS definitions. Once the .jar was compiled, it was copied back into the Felix_AD repo and used until the next time code was updated. While this approach works, manually recompiling, copying, and uploading the new code is fairly involved, especially if commits to common are regular. A preferable approach would to have common be made a submodule of Felix_AD and add the .jar generation to the compilation process of the project.

We updated certain git configurations in order to improve our workflow with submodules.

Listing 3: Git Submodule Configurations

```
# set some configurations
git config status.submodulesummary 1
git config push.recurseSubmodules check
```

The first configuration adds the status of all submodules to the `git status` command. The second configuration ensures that the commit referenced by a submodule can be found in the remote of the submodule repo. This prevents the case where a commit is made locally in the submodule but has not yet been pushed to remote.

C. Feature Branches

Feature branches are at the heart of many workflows. Whenever a new feature needs to be created, a new branch is made from the current working branch (usually master or dev). The work for the new feature is completed on its own branch. Once the work has been completed and tested, the main branch is again pulled down into the feature branch. Conflicts are resolved on the feature branch and the code is retested. Once completed, the feature branch can be merged back into the main branch.

The advantage of feature branches is that new work can completed independent of the rest of the project. This ensures

the incremental commits and changes made for a specific feature do not create issues with work other people are doing on the main branch. Furthermore, it ensures that work other are doing on the main branch do not affect work on the feature branch unless those changes are explicitly pulled in.

D. When in doubt, diagram it out

Drawing out diagrams of the expected behavior of they system, node topologies, and finite state machines made coding to a design significantly easier. Initially, behaviors and requirements were discussed but not recorded in a unified format. Until a full Felix storyboard was written, behaviors and requirements were somewhat fluid, with decisions made on the spot. Once a concrete storyboard was drawn, the development team had a foundation to begin designing the rest of the system. The original storyboard is shown in Fig 7.

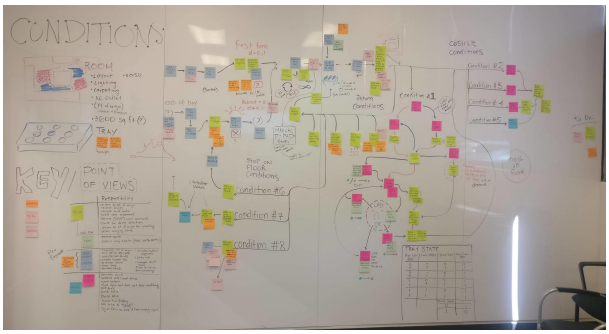

Fig. 7: Version 1.0 of the Felix Storyboard. The storyboard describes Felix's behavior throughout CES.

Certain features like the brain decision node expressed incredibly intricate behaviors based on a number of different inputs. To handle this, we drew out a finite state machine (FSM) to capture how every input affects the decision making process of the node. Once the FSM was drawn, we used a python module to translate the FSM, action for actionto code. The FSM is shown in Fig. 8.

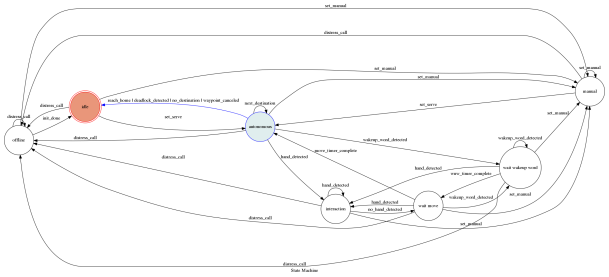

Fig. 8: The finite state machine describing the behavior of the Brain Decision Node.

E. Unit Testing

Unit testing is an invaluable resource to verify code works exactly as expected. Automated unit tests speed up development time be giving developers a fast and reliable metric to determine whether the code they've written works. In an ideal scenario, testing is first done at a function level, then at a node level, and then finally at a system level. In the interest of time, we sacrificed testing at the function level and went straight to node level testing. Tests were conducted via the ROS interfaces for most nodes. **Testing at a node level should be completed before testing at an integration level**[12]. When errors occur at a system level during integration, it is difficult to determine which node is at fault since any node in the system could have sent data that caused the exception or unexpected behavior. By conducting node level test before integration level tests, it is possible to catch and solve many issues before they arise. During this project, we skipped this step a number of times leading to hours of trying to trace back exceptions before finding simple mistakes like checking for `None` in a python call. For more information on the importance of unit testing, please see [12].

F. Debug Order

When debugging robotic systems, there is an order that should be followed.

1) Dirt: stuff blocking/covering sensors, alignment issues
2) Connectors/Wires: are things plugged in correctly?
3) Code: now look for bugs

V. Lessons and Future Work

> Experience is what you get when you didn't get what you wanted. And experience is often the most valuable thing you have to offer.
>
> *Randy Pausch, The Last Lecture*

A. Sensor Coverage

A major issue we had with Felix was navigating obstacles. Felix's obstacle detection and avoidance behaviors are a features the SLAM algorithms running on Slamcore. The version of Slamcore we use only takes inputs from two types of sensors: a planar lidar (Slamtec's RPLIDAR A2) and a series of ultra sonic sensors. Unfortunately, this is simply not enough data to understand our environment to a high enough degree to navigate successfully.

According to [13], *"they key to successful navigation in [an office environment without human intervention] is the robot's ability to reason about its environment in three dimensions, to handle unknown space in an effective manner, and to detect and navigate in cluttered environments with obstacles of varying shapes and sizes"*. The planar LIDAR and sonar sensors only provide the robot with reasoning in two dimensions. While sonars are technically spaced in three dimensions, sonar sensors are best adept to detecting large planar surfaces as opposed to small, round or oddly shaped surfaces. In fact, [13] goes as far as to describe the **EXACT** problems we've been experiencing. *"Most indoor robots rely on a planar or quasi-planar obstacle sensor, such as a laser rangefinder or sonar array. Because vertical structure dominates man-made environments, these sensors are positioned on the robot to detect obstacles along a horizontal slice of the world.* ***The result is a robot that can easily avoid walls but will miss chair legs, door handles, table tops, feet, etc."*** This paper

describes our exact sensor layout and the exact problems we're facing as a result of using the described sensors.

It is important to note that Slamtec offers its own "mule" platform meant for larger robots to navigate through more crowded environments. Both the Zeus [14] (large scale) and Apollo [15] (mid scale) models use an RGB-D camera in addition to a planar LIDAR at the base. Performing a sensor fusion between LIDAR and RGB-D camera provides their advanced platforms the full 3D reasoning suggested by [13]. For comparison, the Apollo and Zeus platforms only use three sonar sensors with a maximum detection distance of $40cm$.

The lack in performance we experience is likely because we are using the Slamcore *LITE* module instead of the more powerful Slamcore *PRO* module which has support for an RGB-D sensor. From our experience the *LITE* module is likely meant to be used in small scale vacuum robots (like iRobot's Roomba) where slightly bumping into objects is an acceptable behavior.

While future work may explore upgrading the *LITE* modules to the *PRO* modules, the issues discussed in section V-D suggest it would be better to move away from the Slamcore platform all together. There are countless examples of open source visual SLAM algorithms with built in ROS support that would better suit our purpose including [16] [17] [18].

B. Android Challenges

By project's end, the main issues we faced besides sensor coverage were related to the stability and performance of the Android application. The responsibility of the app included displaying various views (historical tracks, drink-taken heatmap, virtual walls, etc.), setting and deleting virtual wall elements, manually driving the robot, setting waypoints, and handling the dynamic serve behaviors.

1) Scope: If the primary purpose of the application was to serve as a pure "view" of the system, the tendency of the application to crash, though an inconvenience, would not be fatal. Somewhere during the design process, the application's scope was expanded to handle the dynamic serve behavior. This decision tightly coupled the application's lifecycle to the lifecycle of Felix. Instead of the app being able to be opened and closed at will, or even having the entire tablet removed from the robot, the tablet became an essential element: without the tablet connected and the application running, the robot could not be used.

Knowing that the dynamic planner was prone to crashing, certain safeguards were put in place to protect against a system deadlock. Clients that depended on the dynamic planner were given timeout functions. If the dynamic planner failed to respond, the motion planner node would send Felix home so the application could be restarted by the operator.

2) Stability through Organization: We believe the volatility of the application was in part due to the questionable behavior of the Slamcore Android SDK. From our experience, the same volatility was not seen when working with the Linux SDK. Therefore, **if** we wrote a full set of ROS drivers and ran them on the NUC, and replaced all SDK calls in the Android application with ROS service calls, we could expect to see a reasonable boost in reliability.

This is because processes on Android are bound to the lifecycle of an application. If one part of the application encounters an error, or the system decides to kill the application, everything tied to that application is also killed. In our situation, we believe part of the Slamcore code is causing the system to kill the application, bringing down the rest of our system. In contrast, ROS running on a straight Linux OS spins nodes up in **independent** processes. If a single node crashes, the rest of the nodes in the system remain unaffected. Additionally, ROS has built in support to respawn the affected node(s).

3) Reusibility: Another benefit offered by decoupling the Android application from the Slamtec API is reusability of the application. In our current implementation, we have tightly coupled our application implementation to the Slamtec API. If we the Slamtec system, the entire Android application is rendered useless. A refactoring to the ROS APIs would allow the application to be used with any ROS based systems with the appropriate interfaces.

C. Arduino Challenges

While ROS has support for the Arduino platform through the rosserial libraries, we experienced a number of difficulties when developing for the platform. The first hurdle we experienced was the compilation process. To streamline our development workflow, we moved from manual compilation through the Arduino IDE to compiling through Catkin. Here, we experienced issues when trying to compile with our custom messages defined in flex _felix_common. Even with dependencies expressed on the common package, messages were still not being built in the correct order. This turned out to be a known problem with an issue currently open on the Rosserial Github [19]. The intermediate solution was to build the common package *before* building the rosserial packages. This ensures the most recent changes to common were applied to the messages defined for rosserial at compile time. We automated this work around by scripting it through the `felix_make` command.

Another issue we experienced with the Arduino was hardware limitations. Through experimentation, we found we could only send messages of ~ 24 bytes at a time to the Arduino. While there is likely a configuration somewhere in rosserial that can be tweaked to increase the buffer size, we were not able to find it. We also noticed that long blocking calls (such as delays or long loops) on the Arduino could cause messages to be dropped. Most notably was the code to turn on individual LED's within a loop. A simplified example is explained below.

Listing 4: Long Blocking Code Causes Messages to be Dropped

```
void main() {
  for(int i = 0; i < NUM_LED; i++) {
   // Write to buffer and show each led
       individually
    set_led(i, GREEN);
  }
```

```
}

void set_led(int index, rgb_color color)
    {
  // Writes the LED Color to a buffer
  led_config_rgb.setPixelColor(index, c);
  // Sets changes to take affect
  led_config_rgb.show();
}
```

In this example, the `led_config_rgb.show()` function is called on every iteration of the loop causing unnecessary overhead on the Arudino. This overhead caused roughly 10% of packets to be dropped when sent at a rate of 10hz.

Listing 5: Long Blocking Code Cause Messages to be Dropped

```
void main() {
  for(int i = 0; i < NUM_LED; i++) {
    // Writing to buffer only
    set_led(i, GREEN);
  }
    // Turns on ALL LEDS at once
    led_config_rgb.show();
}

void set_led(int index, rgb color) {
  // Writes the LED Color to a buffer
  led_config_rgb.setPixelColor(index, c);
}
```

By coalescing the new LED states before running the show command, we could reduce the number of writes to a single write at the end of the loop. This provided a tremendous boost in performance, allowing us to receive 100% of messages published at $10hz$. In the interest of time, we did not stress test the new configuration.

Our experiences with the Arduinos in this project indicate they should primarily be used to publish data and perform very simple actions. As much processing as possible should be shifted off board onto a more powerful system. Even with these precautions, special care should be taking to avoid long running blocking calls such as "delays" or processing in long loops to avoid loss of data.

These limitation coupled with difficulties in compiling suggest using a more robust small form factor compute device like a RaspberryPi or BeagleBone Black. These machines are significantly more powerful and run a full Linux OS meaning code can be developed in Python and deployed via SSH without sacrificing access to GPIO pins.

D. Slamcore Issues

We experienced a number of issues while working with the Slamtec platform. One of the earliest issues we encountered was in the reference code written for the STM32 on the sdp mini platform. While the code was fairly legible, all of the comments and the majority of the documentation was written in Chinese. Online documentation was fairly easy to translate using the Google Translate extension on Google Chrome, however translating the comments was a more involved task. Eventually we settled on writing a python script that recursively combed through a given directory to translate all Chinese characters back to English.

Lack of updated documentation was a constant issue while working with Slamtec. Slamtec did not appear to have a version tracking system in place that could effectively connect documentation versions to software versions to hardware versions. At one point we needed to manually probe all pins on the Slamtec breakout board to ensure proper connections were being made after we were told the engineer who had drawn the necessary diagrams had left the company.

Slamtec's loose version tracking made it particularly difficult to know whether code written for different board versions were compatible. We initially started with a version three and two version four boards. Code was written and tested on the version four board. Furthermore, a breakout board was designed to accommodate the pinnouts of version four board. Whenf we tried to procure a third version four board for assembly, we found we could only purchase version six boards. Lack of documentation between the version four and six boards made it very difficult to know whether the code or the designed breakout board would be compatible with the newer versions of the board. Fortunately, we were able to coordinate with some of Slamtec's engineers and determined minimal code needed to be updated and our breakout board was still compatible.

Slamtec's out of the box tools and software also had a number of bugs. The RoboStudio platform was prone to freezing and crashing. There were numerous inconsistencies in units and bugs in the configuration tools provided. Even the generated configurations had to be modified by hand in order for the code to compile. Changing the network configurations through the web portal or attempting to update the firmware on the Slamcore unit was nearly impossible, often requiring multiple attempts before changes successfully loaded.

One of the most critical problems we experienced was compiling the Slamcore SDK within the Catkin build system. The publicly available Slamcore SDKs were compiled with either GCC 4.6 or 4.8. The version of Catkin that ships with ROS Kinetic comes prebuilt with GCC 5.4 and is not backwards compatible with libraries built with previous versions of GCC. Fortunately, Slamtec engineers were able to provide a version of the SDK that was compiled with 5.4, enabling us to proceed with the ROS integration.

Overall, time differences, language barriers, and lack of adequate knowledge over the different parts of their product made interfacing with Slamtec somewhat difficult and very time consuming. As mentioned in section V-A, future work should closely evaluate alternatives to the Slamtec platform.

E. ROS Lessons

Felix was in invaluable learning opportunity to further our knowledge of ROS. Through this project we gained insights on designing ROS projects and infrastructure, developing workflows, and devising best practices for future use. We

have already discussed many of these points in this paper; this section highlights two more important lessons learned on this project.

1) ActionLib: The ROS ActionLib is a topic not covered in the ROS beginner tutorials that played an important role in this project. The first two means of communication taught in ROS are topics and services. Topics are the core of the ROS pub/sub infrastructure: messages of a specific type are published to a topic, anyone subscribed to that topic can receive the message and access its information. Services act as an remote procedure call (RPC) in ROS: a client posts a command to a server and blocks until the server completes its task and responds to the client.

ROS Actions are something in between a service and a topic. The ActionLib gives users the ability to issue a command (generally something time intensive and long running), receive feedback on the status of the command, and eventually be notified of an end state. The client (command issuer) also has the ability to preempt (cancel) the issued command at any point in time. The actions are built on top of ROS topics allowing them to be used in a non-blocking fashion and are capable of triggering callbacks on feedbacks or results.

We used the ROS actions when sending navigation goals from the brain node to the Slamcore controller. Using an action allowed us to "set and forget" a destination as a goal and return to processing other pieces of information from within the brain node. At any point in time, we could go back and cancel the current destination in response to events that may have occurred (NeoNodes or WuW detected, Set Manual commands received, etc.). We were able to use the feedback from the Slamcore Controller to detect "stuck" states in the robot and call for help when needed.

2) Custom Messages: Custom messages played a crucial role in the workflow of our project. When we started, many of our messages were simple enough to the point where we considered using the messages straight from the `std_msgs` package. After careful consideration, we realized that generating custom messages (even if they overlapped with the existing standard messages) offered better portability and maintainability. This decision saved us countless hours of refactoring as our project grew.

We quickly realized that messages where we though a simple boolean would suffice grew to contain larger, nested structures. Had we been interfaced directly to the standard message structures, modifications and refactors would have been much more significant – likely significant enough where the refactor would be deemed too much work to undertake. In total, we had close to 70 commits to the Felix_Common repo, with each commit potentially representing changes to one or more message types.

VI. Conclusion

Despite his early termination, Felix provided a valuable experience, ripe with lessons in software design practices that can be applied to future projects. These lessons covered a wide range of topics from working with ROS, to improving our workflow, to experiencing the consequences of premature development.

References

[1] About ros. [Online]. Available: http://www.ros.org/about-ros/

[2] About spin. [Online]. Available: https://www.parallax.com/propeller/qna/Content/QnaTopics/QnaSpin.htm

[3] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994.

[4] Ros best practices. [Online]. Available: http://wiki.ros.org/BestPractices#Existing_best_practices

[5] Pr2. [Online]. Available: http://wiki.ros.org/Robots/PR2

[6] Autoware github. [Online]. Available: https://github.com/CPFL/Autoware

[7] Strategy pattern. [Online]. Available: https://en.wikipedia.org/wiki/Strategy_pattern

[8] Ros nodes per sensor. [Online]. Available: https://answers.ros.org/question/272719/dealing-with-several-sensors/

[9] Max frequency to look up parameters. [Online]. Available: https://answers.ros.org/question/192223/max-frequency-to-look-up-parameters/

[10] Ros cpp param server. [Online]. Available: http://wiki.ros.org/roscpp/Overview/Parameter%20Server

[11] Git submodules. [Online]. Available: https://www.git-scm.com/docs/gitsubmodules

[12] Unit testing in ros. [Online]. Available: http://wiki.ros.org/Quality/Tutorials/UnitTesting

[13] E. Marder-Eppstein, E. Berger, T. Foote, B. P. Gerkey, and K. Konolige, "The office marathon: Robust navigation in an indoor office environment." in *ICRA*. IEEE, 2010, pp. 300–307. [Online]. Available: http://dblp.uni-trier.de/db/conf/icra/icra2010.html#Marder-EppsteinBFGK10

[14] Slamtec zeus spec. [Online]. Available: https://www.slamtec.com/en/Zeus/Spec

[15] Slamtec apollo spec. [Online]. Available: https://www.slamtec.com/en/Apollo/Spec

[16] R. Mur-Artal, J. M. M. Montiel, and J. D. Tardós, "ORB-SLAM: a versatile and accurate monocular SLAM system," *CoRR*, vol. abs/1502.00956, 2015. [Online]. Available: http://arxiv.org/abs/1502.00956

[17] R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *CoRR*, vol. abs/1610.06475, 2016. [Online]. Available: http://arxiv.org/abs/1610.06475

[18] C. Forster, M. Pizzoli, and D. Scaramuzza, "SVO: Fast semi-direct monocular visual odometry," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[19] Ros serial compilation bug. [Online]. Available: https://github.com/ros-drivers/rosserial/issues/239

[20] Slamtec api documentation. [Online]. Available: https://wiki.slamtec.com/pages/viewpage.action?pageId=1016252

[21] Slamware control bus protocol. [Online]. Available: https://wiki.slamtec.com/pages/viewpage.action?pageId=1016210

[22] Ros actionlib tutorials. [Online]. Available: http://wiki.ros.org/actionlib_tutorials/Tutorials

[23] Slamtec wiki. [Online]. Available: https://wiki.slamtec.com

[24] Ros tutorials. [Online]. Available: http://wiki.ros.org/ROS/Tutorials

Biographies

Andrew Tu is a fourth year undergraduate student at Northeastern University majoring in computer engineering and computer science. Tu has been active in undergraduate research at Northeastern since Fall of 2015. To date, he has contributed to three different labs and has received 3 NSF REU grants to conduct research as an undergrad and received a GENI SAVI grant to conduct research in Rome, Italy during the summer of 2016. Tu completed his first co-op at MIT Lincoln Laboratory where he worked on high performance computing on a radar signal processing chain. He is currently on his second co-op at the Innovation and Design Labs at Flex in Milpitas, CA.